YXORP
Guide and Reference Manual

yxorp version 2.33 and later

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'.

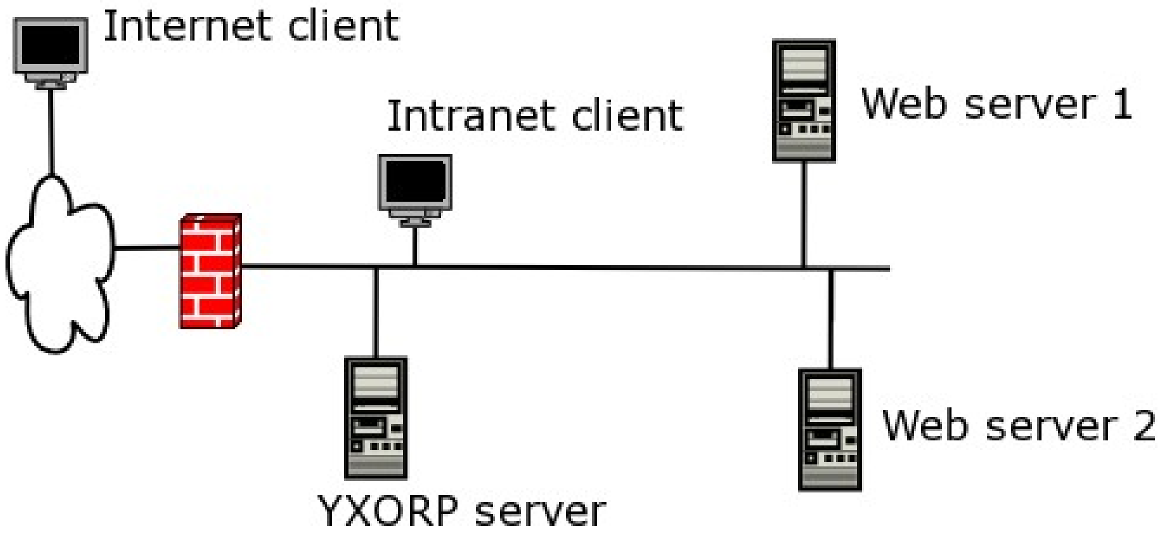
Table of Contents

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. Clients, i.e. systems running browsers etc., reside on the Internet.



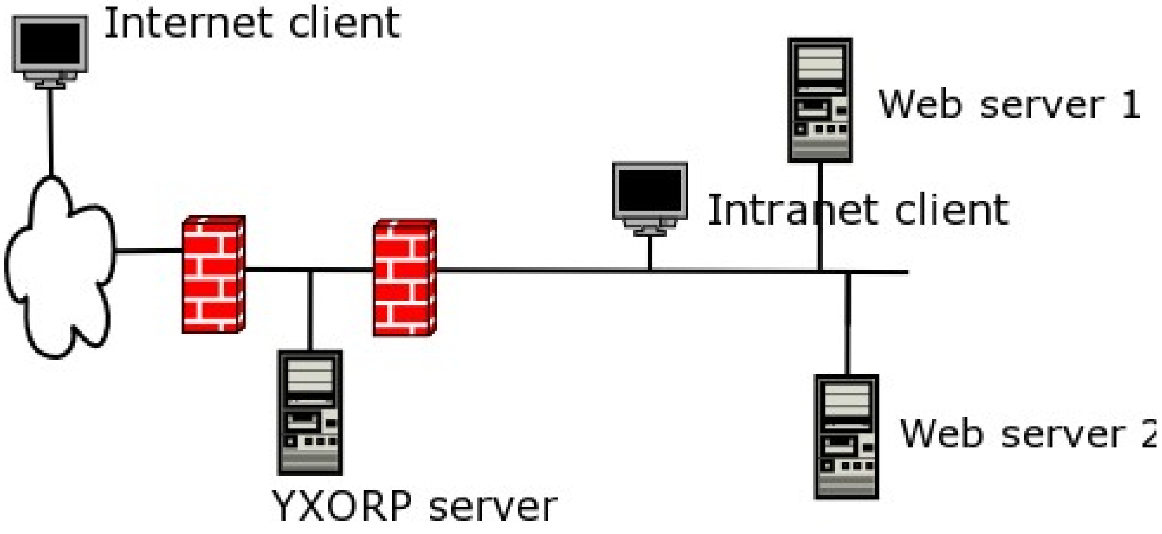
The firewall is, in this kind of setup, most probably running NAT (and thus will probably have only one public Internet address).

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, for example by an attacker exploiting an error in Yxorp's code, or by a mistake in the configuration, the attacker might able to insert another configuration or bypass Yxorp. By changing the configuration, it might 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. One way of doing that is to make sure each box runs firewalling software and is adequately patched, like you would normally do for a box that is directly connected to the Internet.

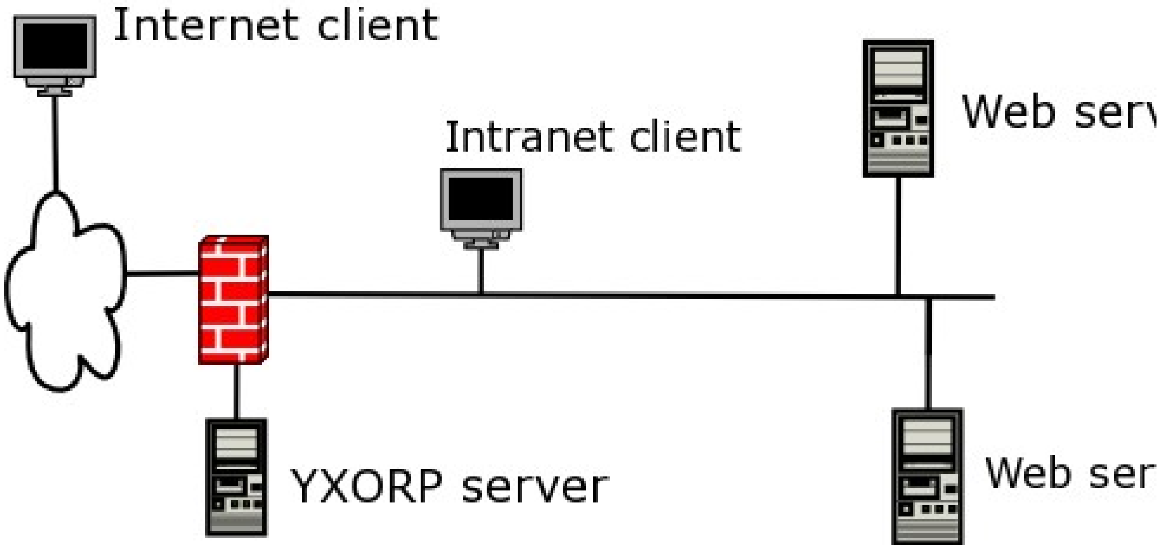
Secure network setup

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.



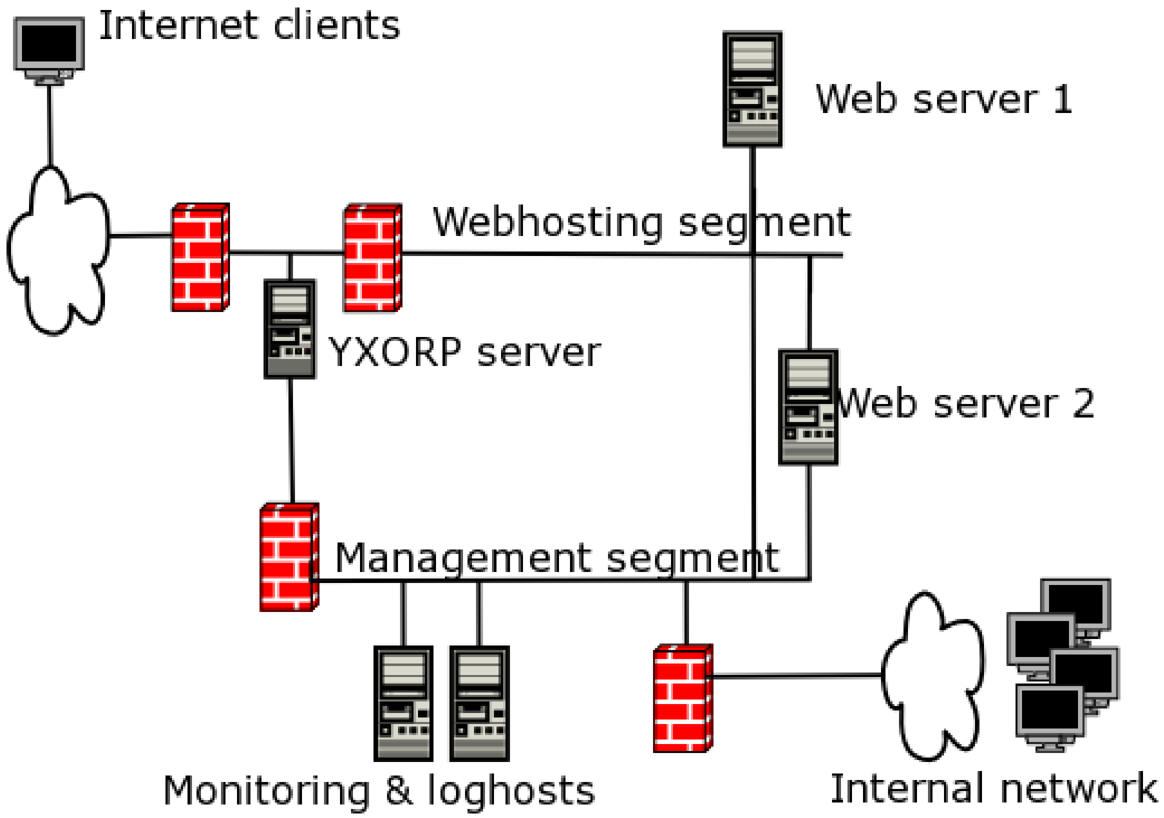
The difference with the minimal setup is that the traffic between Yxorp and the web servers is now restricted by the second firewall; this means that even if Yxorp is compromised, it is still not possible to use that vulnerability to reach the Intranet clients or other systems on the internal network (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 that 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, retaining 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). The philosophy of minimizing a system is based on the idea that software that is not installed on the box cannot be vulnerable to attacks, and does not need security patches.

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 (which you probably wouldn't need) and telnet (which you wouldn't want). 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. Typically, 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 you 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, the Yxorp configuration needs to list the IP-addresses of the real servers, and the actual web server(s) need to be configured to serve content for the domain name.

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 described in the INSTALL file contains some steps that help in generating a test certificate authority, server certificates, and optionally client certificates.

Yxorp can deal with the TLS extension called server name indication. SNI makes it possible to use virtual hosting in combination with SSL. SNI works by allowing a client to send the desired server name before the SSL handshake has completes, and, more to the point, before the server has to present it's certificate (which must contain the server name).

Current major browsers in general seem to support SNI – if they are running on a current major operating system. To enable SSL with SNI for Yxorp, it needs to be built with OpenSSL 0.9.8g or later. The current major Linux distributions currently include a suitable version of OpenSSL; there is no need to build it specifically to enable SNI any longer. However, there are also many distributions that are not yet at a suitable level; check the output from the ./configure to be sure (especially the messages saying something like “checking if openssl version is high enough to support TLS extensions... yes” and “checking if openssl version is high enough to support TLS extensions... yes”). If necessary follow instructions on how to enable SNI on backlevel distributions in the file INSTALL.tlsext.

Basic configuration tutorial

Basic setup

Since version 2.25, the normal open-source stanza of configure, make, make install no longer installs the default configuration. You will need to read the file INSTALL that came with the distribution to install the default configuration that this tutorial uses, as well as for things like generating certificates.

The default configuration you get by following the instructions in the file INSTALL is used in this tutorial to explain the basics of how Yxorp works and to show some of the things Yxorp can do.

There are no complicated things necessary to run this configuration; it only needs a machine with Internet access to run Yxorp and a machine (possibly the same) that runs browsers and can connect to the Yxorp machine. You don't need to set up a web server, because the tutorial uses Yxorp's web site at http://yxorp.sourceforge.net/ (this is why the system needs Internet access).

First, we will look at the contents of the default configuration file.

```
<?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. The file names set here are the files Yxorp will actually write to; log file rotation will cause Yxorp to close the current file, rename it, and open a fresh one with the original name.

The format of the access and error logs can be changed, but for this tutorial, 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, the command “yxorprealserver” can be used to set servers in or out of service, “yxorpclientstate” can be used to look at client states, etc.

```
<!-- 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. Which rule to run on a request is defined on a listener, as we will see later in this example. This particular rule looks at the value of the Host: header (every variable name ending in a colon ':' is interpreted as a header name). If, in this rule, the Host: header 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. Below the multiple-line string, the special variables errorcode and rejectreason are added; this will generate a simple HTML page showing a text showing which error code Yxorp generated (or received from the server), and a text message giving the reason why the request was rejected. In a real situation, you may not want to give out this information; however, the choice is yours, and for this tutorial it makes sense not to hide the reason why Yxorp rejects a request.

```
<!-- 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. There are several rule types you can set on the listener, and in other places; most go beyond the scope of this chapter. Further on, though, we will deal with special rules that run if servers are not available.

```
<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 (i.e. what would normally be the domain name of a server). The Host: header value is matched to a <server> definition (i.e. the <server> definition should normally have the same name as the domain name). The <server> in this example is named “yxorp.sourceforge.net”, and thus will be selected if the Host: header contains this domain name (remember we used the request rule above to make sure it does). The <server> maps to a <virtualserver> group; <virtualservers> are used to create the link between one “logical” <server> and one or more “physical” <realserver>. This is the way Yxorp can be configured for load balancing.

This virtualserver group defines only one real server (more will be added later on in this tutorial). 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 to be translated to an IP address. This is useful for examples and testing, but in a real life situation it is better to define IP addresses for reasons of 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.

If you followed the instructions in the file INSTALL that comes with the source code, the complete configuration file for this example should have been installed on your system. You can also pick it up from http://yxorp.sourceforge.net/examples/yxorpconfig.xml

Testing the configuration

As root, start yxorp with the following command:

```
# yxorp
```

Yxorp needs to be started by root; however, it will release most privileges the root user has (on Solaris and Linux). You can also configure Yxorp to change to another userid and group and/or run it in a chroot jail, see the configuration reference for details.

Since version 2, Yxorp starts as a background daemon by default (so it may appear as though nothing happened). You can use any of the following commands to verify that Yxorp is running:

```
# yxorpconfig -r             (reads the actual configuration from the daemon)
or
# ps -e| grep yxorp          (lists the process table and looks for yxorp)
```

If you did not install Yxorp into some directory in your path, you'll have to include a path to the yxorp command. Yxorp uses a default configuration file name to start from; the default filename is “yxorpconfig.xml”, and Yxorp looks for this file in the default location for configuration files. On most systems, this is /usr/local/etc. Both can be changed as in the following example:

```
# yxorp -c my-yxorpconfig.xml -s /directory-for-yxorp-configurations
```

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/

You can leave the Yxorp daemon running for the next steps in this tutorial, however, if you want to stop the daemon for some reason, use the following command:

```
# yxorp -K
```

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 installed if you followed the instructions in the INSTALL file, but you can also use your own certificate. Yxorp tries to read the file from the same directory as it's configuration is in; usually this is /usr/local/etc.

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 ~/^\/yxorp*$/) {
   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 /yxorp 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, and 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; Yxorp will always report the base64 form.

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). By default, 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 as in the following example:

```
<?xml version="1.0"?>
<yxorpconfig>
<globalconfiguration>
   <header id="X-Pad:" xlateid="X_Pad:" client="1" server="1"
      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 chapter 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 length parameter, which is the maximum length of this header that Yxorp will accept. For most header attributes, you can specify if

Yxorp should ignore (and discard) the header, or reject the request.

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 by default have the name “<your-hostname>_state”, 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 cookies and associated client state information are timed out after a configurable time of inactivity. See the chapters on client state for more details.

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 -v
<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.

Client states (and following from that, also the sticky load balancing mappings) are cleaned up automatically by Yxorp after a configurable time of inactivity, see the chapter on client state for more details.

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.

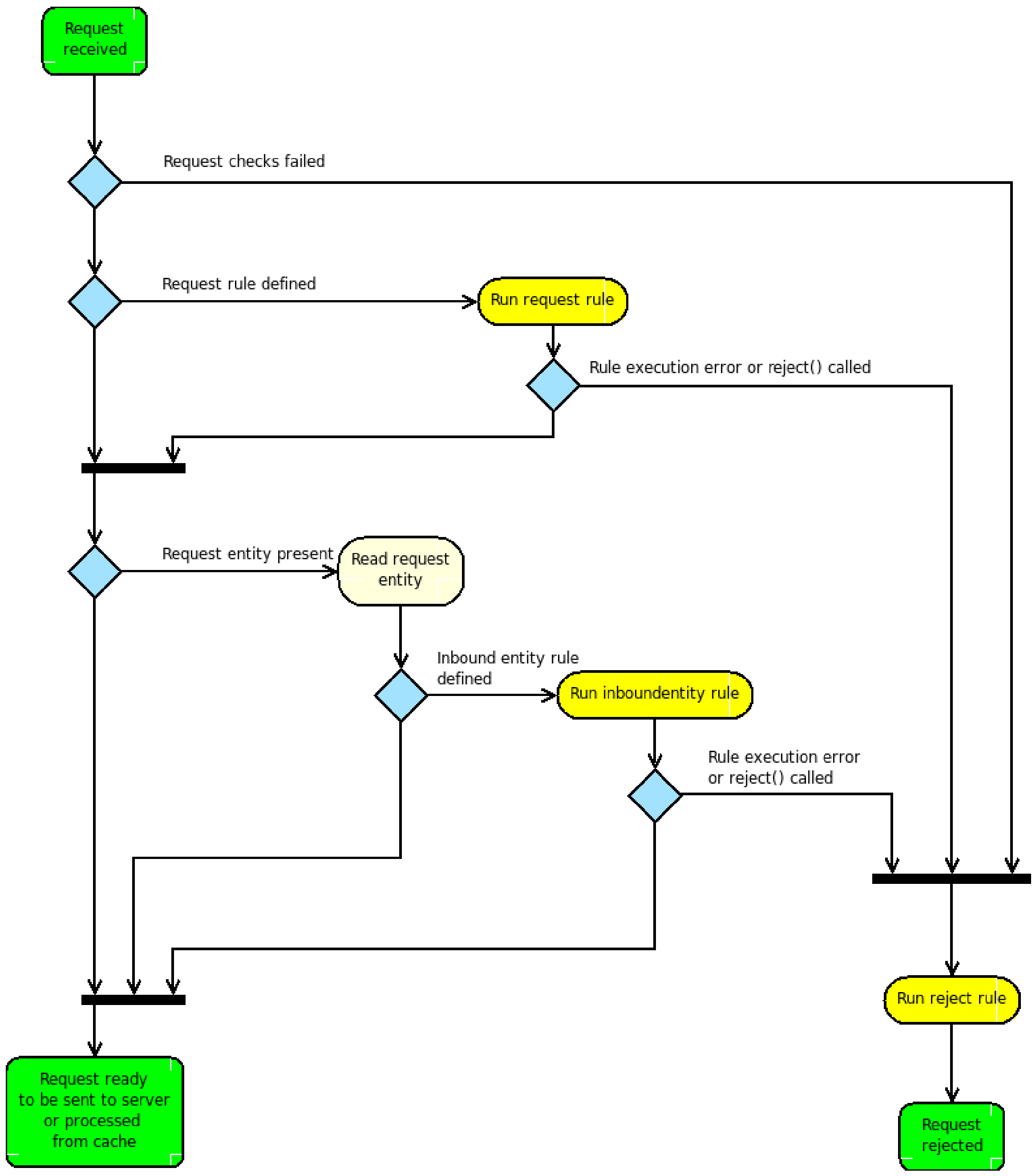
Rule programming concepts

Yxorp has several types of rule, each of which is applicable to a specific phase of processing a request or reply, or addresses a specific situation that may occur during the processing of a request or response.

For the purpose of documentation, it is easiest to deal with the rule types by dividing the processing of a single request and the response to that request in three groups: Request processing, server selection, and response processing.

Request processing

The drawing below shows which rules are run during Yxorp's request processing, i.e. the process that happens from the point when Yxorp has received a request from a client, to the point where Yxorp needs to figure out which server to send that request to.



In the drawing, the observant reader will note that the green 'state' box in the lower left includes 'request ready to be processed from cache'. The current version of Yxorp does not define rules to run if a request is served from Yxorp's cache; this will be added in some future version.

Request rules

The request rule is run whenever a request is processed, and a request rule is defined. The request rule must be defined on the listener that the request comes in on. Because the request rule is by definition the first rule that runs in the processing of a request, it has no context (i.e. no variables set by other rule types can exist).

The request rule has access to all headers that have been taken from the request, and the following special variables:

Name	***Use***
uri	The uri variable has the uri for the current request. It is in the form as taken from the request, except that it is normalized to a relative uri, i.e. the protocol and host parts are removed if the request contained them. In RFC2396 terms, everything up to the path specification is removed. The uri variable may be assigned a different value. If this is done, the request that is sent to the server will contain the modified value.
method	Set to the method name. Use for reference only; changing this variable will have no effect.
rejectedheaders	If Yxorp rejects a header (because it does not know this header, it is overlength, contains illegal characters, etc) the header name is appended to this variable to form a space-separated list of all rejected variables.

Inboundentity rules

An inboundentity rule is used when a request is processed, and an inboundentity is present on the request, and the rule is set for this request. The rule may either be inherited from a definition on the listener that this request came in on, or it may be set by a request rule by calling the setinboundentityrule() function.

The inboundentity rule runs in the context of the request; that means that it has access to all the header variables and also non-header variables that were set in the request rule. All values for the variables will be as they were left by the request rule.

Inboundentity rules also have access to the same special variables as request rules do. Beyond that, the entity itself is also available to this rule type, but currently this is limited to access through function calls only (there is an experimental function that takes the entity and puts it in a variable, but that function will be deprecated in a future version).

Reject rules

The reject rule is run whenever the base code of Yxorp detects an error during the processing of a request, or when a rule executes a reject() function call. The context of the reject rule depends on the rules that ran before it. It inherits all non-header variables from those rules.

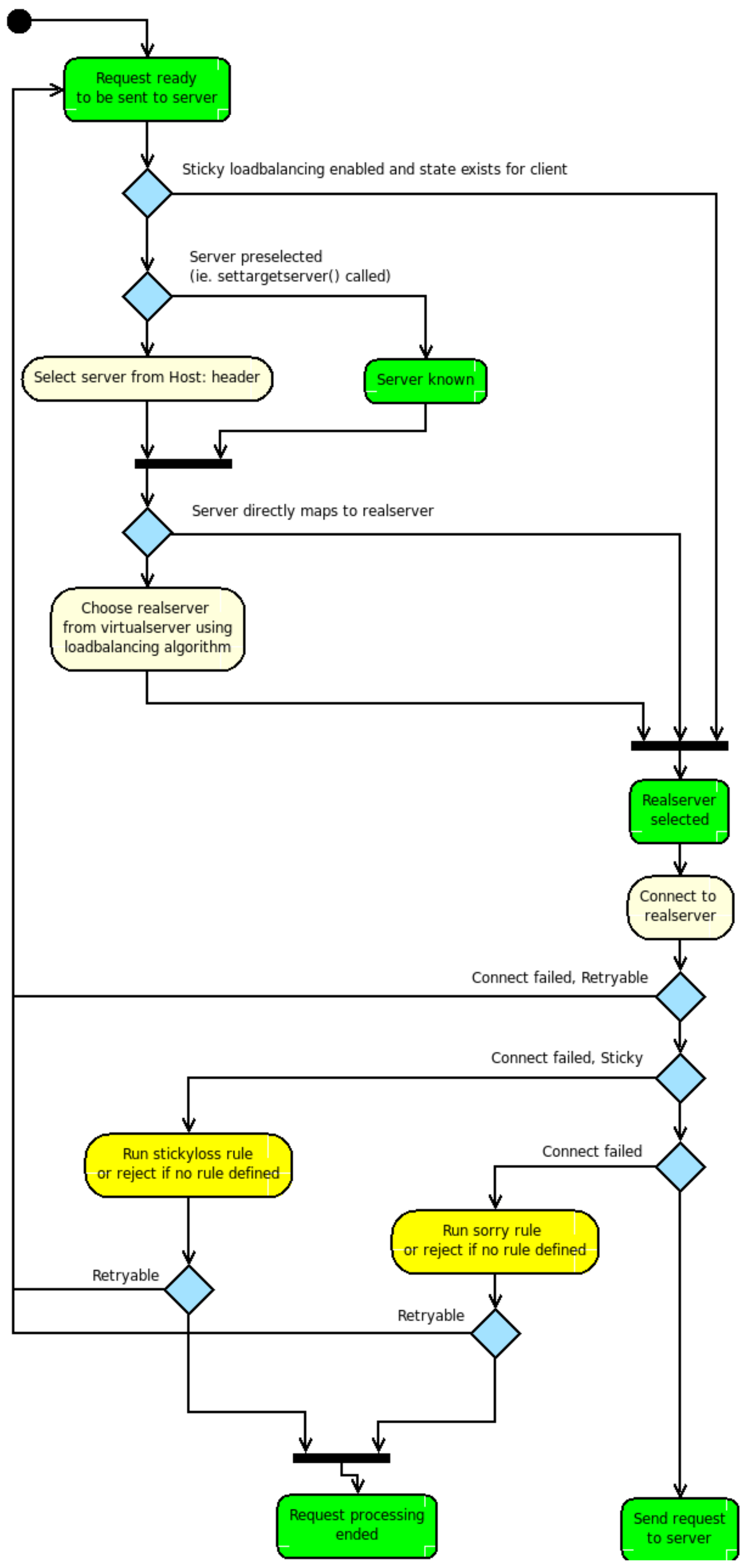
If a reject rule runs (either by a call to the reject() function or triggered by the base code), the effect is ultimately that the request or response is not processed normally. If a connection to the client exists at the time that the reject is processed, an error message will be sent to the client. In some cases, this will include a meaningful error code as defined in the HTTP specification. In many other cases, the error code will be a default of 400 – Bad request, because the HTTP specification does not include status codes for all reasons that Yxorp may reject a request for.

Reject rules have access to a number of special variables that allow you to change the error that is sent to the client. It is also possible to include HTML into the error message, so that a simple page can be displayed by a browser.

Name	***Use***
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.

Server selection

The diagram below shows how Yxorp selects to which server a request should be sent, and which rules may run during this process. Rules only run during the server selection process if something went wrong; either Yxorp is unable to send the request to a server, or the server that the user was sticky-loadbalanced to is no longer available.


Request ready
to be sent to server
Sticky loadbalancing enabled and state exists for client
Server preselected
(ie. settargetserver() called)
Select server from Host: header
Server known
Server directly maps to realserver
Choose realserver
from virtualserver using
loadbalancing algorithm
Realserver
selected
Connect to
realserver
Connect failed, Retryable
Connect failed, Sticky
Run stickyloss rule
or reject if no rule defined
Connect failed
Run sorry rule
or reject if no rule defined
Retryable
Retryable
Request processing
ended
Send request
to server

Sorry rules

A sorry rule is run by Yxorp if it can not connect to a server, respecting the in- or out-of-service and available status, and retries to the maximum configured in the globalconfiguration item maxserverconnectionattempts.

The sorry rule can call functions that change the processing of the request (for instance sending it to another server by calling the settargetserver() function); or call the redirect() function that will cause the client to redirect to another URL (for instance a 'sorry' page), or call the reject() function, causing an error to be returned to the client.

Both the reject() and redirect() function calls cause the reject rule defined for this request to be run. If neither of these functions is actually called during the execution of the sorry rule, Yxorp assumes that some other aspect of the request was changed by the rule and will retry to setup a session, up to and until the maximum specified in maxserverconnectionattempts. If the maximum retries are exhausted, Yxorp ends the request by causing a reject, and causing a reject rule, if defined, to be run.

Sorry rules inherit variable tables, including header variables, from earlier rules. Sorry rules may also access the following special variables:

connectionretries	holds the number of times that yxorp attempted to set up a connection to a server during the processing of this request.
maxconnectionretries	holds the number configured in globalconfiguration item maxserverconnectionattempts.

Stickyloss rules

A stickyloss rule is run by Yxorp if it can not contact, within the current retry limits set in the globalconfiguration, and respecting the available status, the specific server that this specific request should be send to according to the sticky load balancing state that Yxorp maintains for this client.

A stickyloss rule can call functions to change the processing of the request (exactly like sorry rules can). In this specific situation however, changing something about the request may not make sense; the specific server maintaining the state can not be made contactable in this way.

Both the reject() and redirect() function calls explicitly or implicitly cause the reject rule defined for this request to run. If neither of these functions is called, Yxorp assumes that some other aspect of the request was changed by the rule and will retry to setup a session, up to and until the maximum specified in maxserverconnectionattempts. Keep in mind however that this may not make sense, because the specific server maintaining the state will most probably still not be contactable.

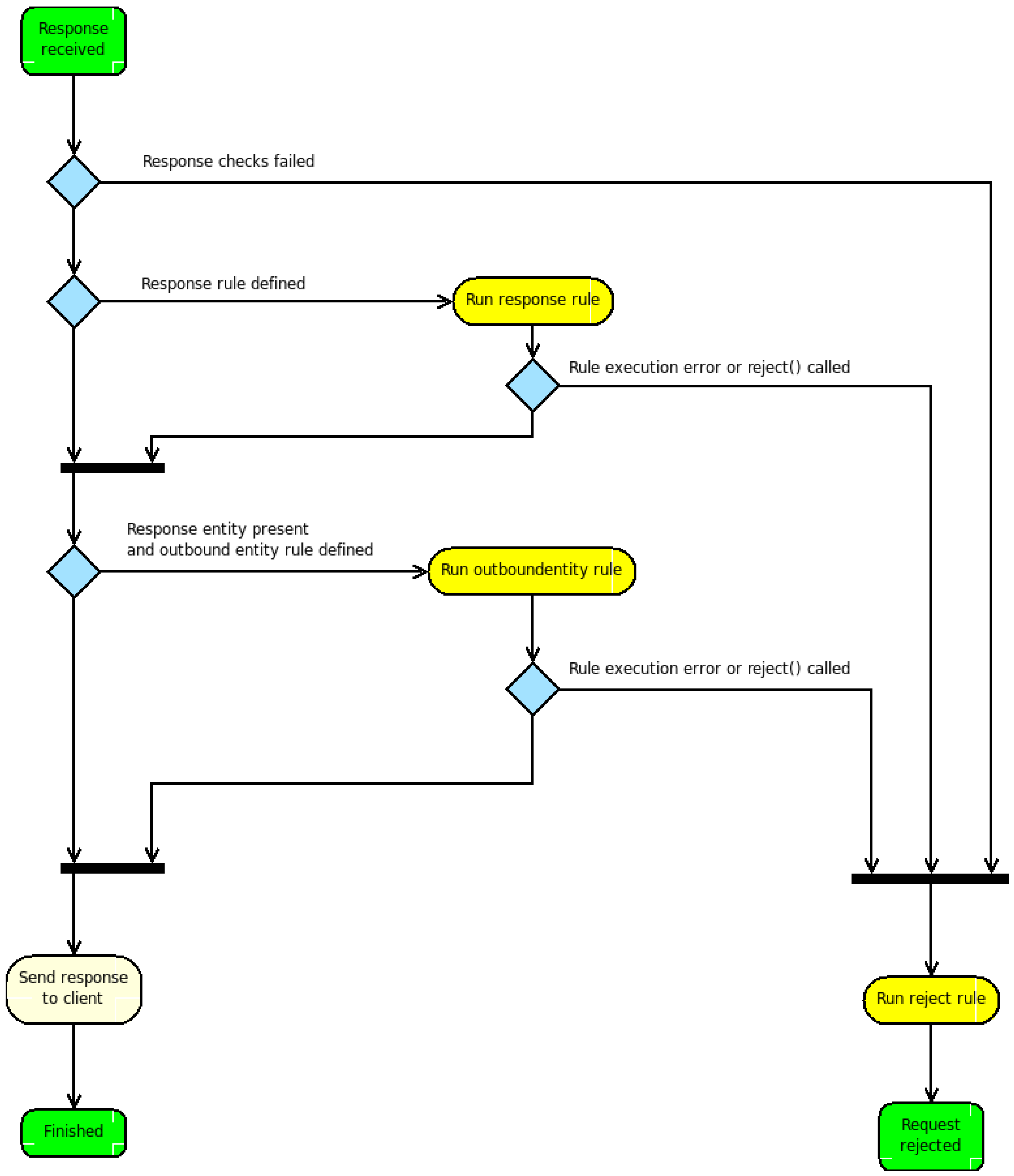
If the maximum retries are exhausted, Yxorp ends the request by causing a reject, and causing a reject rule, if defined, to be run.

Stickyloss rules inherit variable tables, including header variables, from earlier rules. Stickyloss rules may also access the following special variables:

connectionretries	holds the number of times that yxorp attempted to set up a connection to a server during the processing of this request.
maxconnectionretries	holds the number configured in globalconfiguration item maxserverconnectionattempts.
stickylost	set to 1 if a stickyloss situation has occurred for this request; this variable may be used to disambiguate where the same rule code is shared by sorry or stickyloss rules

Response processing

The diagram below shows which rules Yxorp runs during the processing of a response. This is the final stage of Yxorp's processing of a request.



Response rules

Response rules are run by Yxorp when a response rule is defined for this request, either inherited from the listener the request originally came in on, or set by calling the setresponserule() function in a rule previously run for this request.

The response rule inherits any non-header variables from earlier rules. Header variables are taken from the response as sent by the server. The response rule also has access to the following special variables:

Name	***Use***
uri	for reference only
method	Set to the method name. Use for reference only; changing this variable will have no effect.
rejectedheaders	If Yxorp rejects a header (because it does not know this header, it is overlength, contains illegal characters, etc) the header name is appended to this variable to form a space-separated list of all rejected variables.

Outboundentity rules

Outboundentity rules are run by Yxorp when a response rule is defined for this request, either inherited from the listener the request originally came in on, or set by calling the setoutboundentityrule() function in a rule previously run for this request.

The outboundentity rule inherits header variables from the response rule that ran before it. If no such rule exists, no header variables are available to the outboundentity rule. Other, non-header variables are inherited from any other rule that ran before the outboundentity rule.

Rule progamming examples

This chapter deals with rules, and how they can be used to change requests and responses. The focus in this chapter is on real-world situations that are applicable to many Yxorp users. 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.

If you run Yxorp as a security device, 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. Always be sure that you understand what you're dealing with, read up on RFC's and other documentation, make traces to see what is happening on the wire.

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 - to to 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 by default check if the certificate that the backend server presents is valid. That means it could be a self-signed certificate, or it could be out of date, it could be issued for another domain name, etc. You could be accepting a certificate that is forged to make you think you are connecting to a verified site, while instead you are connecting to something else. If you are not in complete control of the server(s), use the isscertverified() function to verify the server certificate, and read the Advanced Concepts chapter on SSL processing.

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); you could use several constructions in your regular expressions to match in the desired way, but this may prove to be tricky to get right. Alternatively, an easy way out is to use one of the functions in the rule language to strip unwanted parameters off the URI before matching against the white- or blacklist.

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 (pun intended... I mean too coarse-grained). 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.

Advanced concepts

This chapter provides examples and reference information about complex reverse proxy processing.

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 for a dissecting proxy:

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.

Attribute	*Format*	
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 the rule that will be executed for requests.
resprule	string	The name of the rule that will be executed for responses. This setting can be changed by using rule functions in the request rule.
rejrule	string	The name of the rule that will be executed if a request is rejected. This setting can be changed by using rule functions.
inboundentityrule	string	The name of the rule that is run after an inbound entity (ie. client-to-server) has been received. Setting this causes inbound entities to be buffered, and the request to be processed store-and-forward. This setting can be changed by using rule functions.
outboundentityrule	string	The name of the rule that is run after an outbound entity (ie. server-to-client) has been received. Setting this causes outbound entities to be buffered, and the request to be processed store-and-forward. This setting can be changed by using rule functions.
cachearea	string	The name of the cache area that is applicable for requests coming in on this listener. This setting can be changed by using rule functions.
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. The file must also contain the private key associated with the certificate, or the pvtkeyfile attribute must be used.
pvtkeyfile	string	Name of a file containing the private key associated with the certificate set by the certfile attribute. If the pvtkeyfile attribute is not used, Yxorp expects to find the private key in the file set by the certfile tag.
certpasswd	string	Optional password to decrypt the certificate with
cafile	string	Name of a file containing the certificate authority information that is used to verify certificates. The file must be in PEM format.
requestclientcertificate	number	If nonzero, Yxorp will request a client to send a client certificate during the setup phase of a SSL connection between a client and this listener.
verifyclientcertif	number	If nonzero, Yxorp will verify client certificates it receives from

Attribute	*Format*	
icate		sessions from clients to this listener. If the certificates do not verify, the session setup will be aborted.

For a tunneling proxy, the following attributes are applicable:

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	ipaddre ss	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.
tunneldest	ipaddre ss	If mode=”tunnel”, IPv4 address of the tunnel destination host; if mode=”dissect”, this attribute has no meaning.
tunnelport	ipaddre ss	If mode=”tunnel”, TCP port number on the tunnel destination host; if mode=”dissect”, this attribute has no meaning.

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.

In case a certificate file modification time stamp has changed, the certificate file will be reloaded.

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 TLS extension Server Name Indication

When built with OpenSSL version 0.9.8g or later, Yxorp can support the TLS extension Server Name Indication. This means that one of several server certificates may be presented to the client, according to the server name that the client includes in the TLS handshake (assuming, of course, that the client also supports SNI).

The certificate, CA file etc. configured on the <listener> tag is the default certificate in the SNI context; that is, the certificate on the <listener> tag is presented to the client if no server name is included in the TLS handshake. If the server name is present, this name is used to map into a list of <certlist> items that can be configured inside the <listener> tag. If a match is found, the certificate defined by the <certlist> is used for the TLS/SSL handshake; if no match is found, the default defined on the <listener> tag is used.

The <certlist> tag takes the following arguments:

Attribute	***Format***	
hostname	string	The host name, or server name in the TLS context, that will be used to map to this certificate list entry.
certfile	string	Name of a file containing the certificate to be used. The certificate must be in PEM format. The file must also contain the private key associated with the certificate, or the pvtkeyfile attribute must be used.
pvtkeyfile	string	Name of a file containing the private key associated with the certificate set by the certfile attribute. If the pvtkeyfile

Attribute	*Format*	
		attribute is not used, Yxorp expects to find the private key in the file set by the certfile tag.
certpasswd	string	Optional password to decrypt the certificate with
cafile	string	Name of a file containing the certificate authority information that is used to verify certificates. The file must be in PEM format.

Reconfiguring

All attributes on a <certlist> item can be reconfigured. In case a certificate file modification time stamp has changed, the certificate file will be reloaded.

Example

The following example shows how <certlist> items are configured:

```
<listener
             id="tlslistener"
             ipaddress="127.0.0.1"
             port="443"
             certfile="test.pem"
>
             <certlist hostname="www.example.com"
certfile="examplecert.pem" />
             <certlist hostname="secure.example.com"
certfile="secureexamplecert.pem" />
</listener>
```

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.
virtualserv	string	Identifies the virtual server that will process the request.

Attribute	***Format***	
er		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", "lru", and "weightedrandom".
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.
clearrealservers	number	If non-zero, all <real> tags within this virtualserver will be cleared.

The <real> tag takes the following attributes:

Attribute	***Format***	
id	string	Must be set on each server. The id must be unique.
weight	number	If weightedrandom scheduling is used, this attribute is used to define the weight of this real server mapping. A weight of zero defines a "server of last resort", that will be scheduled only if no servers with a higher weight are in service and available.
add	number	(default non-zero) If non-zero, this <real> tag will be added.
remove	number	If non-zero, this <real> tag will be removed.

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.
verifyserve rcertificate	number	If non-zero, SSL connections to this real server will verify the server certificate. If the certificate does not verify, the connection setup will be aborted.
cafile	string	File containing the certificate authority information against which the server certificates will be checked.
available	string	“yes” or “no” to set the intended availability state. If unavailable, Yxorp will not send requests to this server, except requests from sticky-loadbalancing sessions that have a sticky state mapping to this server.
inservice	string	Possible values are “yes” or “no”; automatically changed by server wakeup mechanisms (see virtualserver). If out-of-service, Yxorp will not send requests to this server.
reason	string	(display only) The source of the information in the inservice

Attribute	***Format***	
		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 available, that is, the server is presumed to be able to process requests; the server is also set inservice, which means that new requests can be scheduled to this server.

```
<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”, “inboundentity”, “outboundentity”, “stickyloss” 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. There are also files for debugging output, and for request details.

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 configuration actually cause debugging output. If no file name has been set, and debug output is generated, it will be sent to stdout.
requestdetaillog	string	File name for the request detail log. This log file can be used to capture detail information about each request, or alternatively about failed requests only. The amount of information logged is much larger than with the accesslog or debuglog. The <requestdetaillog> tag can be used to tailor what information is logged; also, function calls in the rule language allow to make specific exceptions for specific requests.
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  %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, taken from the Host: header (optionally modified by rule programs by assigning the Host: variable)
%targetserver	Host name of server, taken from the Host: header (optionally modified by rule programs by using the settargetserver() function call)
%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. Same as %clientprotocol.
%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
%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. If the request was served from a cache area, the name of the area is set instead.

Token	***Field value***
%clientprotocol	Returns “http://” or “https://” according to the current protocol used between yxorp and the client
%serverprotocol	Returns “http://” or “https://” according to the current protocol used between yxorp and the server

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 request detail logging

The requestdetaillog is a log file, defined in the <log> tag described above. It can be used to log details about each request, or only about requests that ended in error. The request or status line (the first line in an HTTP message) and the headers can be logged in each stage of Yxorp's processing.

The <requestdetaillog> tag takes the following attributes:

Attribute	***Format***	
log	number	Master flag; if set to 0, no requestdetaillog actions are taken.
logerror	number	If set to 0, all requests are logged to the requestdetaillog. If set to non-zero, only requests for which the end status is in error are logged to the requestdetaillog.
logreceivedrequest	number	If set to non-zero, received request data (request line and message headers) is added to the requestdetaillog. The hex dump may also contain the first part of the request entity, depending on what is in the buffer at the time.

Attribute	***Format***	
		The following bitmasks apply: • 0x01 : Text dump • 0x02 : Hex dump • 0x04 : State of the internal data structures at time of dump
logtransmittedreque st	number	If set to non-zero, transmitted request data (request line and message headers) is added to the requestdetaillog. The hex dump may also contain the first part of the request entity, depending on what is in the buffer at the time. The following bitmasks apply: • 0x01 : Text dump • 0x02 : Hex dump • 0x04 : State of the internal data structures at time of dump
logreceivedresponse	number	If set to non-zero, received response data (status line and message headers) is added to the requestdetaillog. The hex dump may also contain the first part of the response entity, depending on what is in the buffer at the time. The following bitmasks apply: • 0x01 : Text dump • 0x02 : Hex dump • 0x04 : State of the internal data structures at time of dump
logtransmittedrespo nse	number	If set to non-zero, transmitted response data (status line and message headers) is added to the requestdetaillog. The hex dump may also contain the first part of the response entity, depending on what is in the buffer at the time. The following bitmasks apply: • 0x01 : Text dump • 0x02 : Hex dump • 0x04 : State of the internal data structures at time of dump
logfinalinternaldata	number	If set to non-zero, a formatted copy of some of Yxorp's internal data structures are addedd to the requestdetaillog.

Reconfiguring

All flags in the requestdetaillog tag may be reconfigured.

Example

The following example shows how to configure the request detail log to log the data received for requests that ended in error:

```
<requestdetaillog log=”1” logerror=”1” logreceivedrequests=”1” />
```

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 following 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.
startatoverrun	number	If the minfree threshold is reached, this amount of new threads will be started. The default and minimum is 1; the maximum is 10.
actual	number	(display only) Actual number of active threads.
actualfree	number	(display only) Actual number of free (waiting) threads.

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. Because of this, the number of threads or LWPs reported by operating system commands may be higher than the value of max or actual.

In general, and assuming that clients ie. browsers behave according to RFC2616, each client can use two parallel sessions to retrieve content. During the handling of requests, and for some time after that according to the session timeout settings, this translates into two active threads for each client. The number of actual threads you need depends very much on the 'think time' of your users, that is, how long users will stay on the same web page. If this is shorter than the session timeout settings, each user will use up to two threads. If the 'think time' is longer than the session timeout settings, the actually needed number of threads will be lower. The timer settings that influence this are readfromclienttimeout, sslclientreadtimeout, and completerequesttimeout.

The defaults should suffice for normal web sites. If you run a heavily loaded site (ie. serving hundreds of concurrent sessions) you may have to increase the max attribute or lower the timer settings. Be careful about changing these settings though.

Reconfiguring

It is possible to reconfigure threads; however, reducing 'max' will not lower the number of active threads. Threads do however stop after having been idle for a configurable time.

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”. In this way, Yxorp does not need to run as root to be able to open low sockets (ie. <1024).

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. Because of privileges dropped on Solaris and Linux, Yxorp will not have omnipotent file system access even when it runs as root.

Other operating systems (like FreeBSD, NetBSD, MacOS etc) currently have no working privilege model. On those operating systems Yxorp needs to run as root to be able to bind to ports lower than 1024. It is still possible to have Yxorp run as non-root on those operating systems, but all listeners will need to bind to high ports.

Another security setting is “chroot”; if this is set, yxorp will change root to the specified directory. Within the jail, you will need the following:

File name or description	
/dev/random	A source of entropy for SSL.
/dev/urandom	If your build includes SSL support, Yxorp needs read access to at least one of these.
/dev/null	Yxorp needs /dev/null to run as a daemon.
<ssl certificates>	The certificates for your SSL-enabled listeners. Yxorp needs read access to these files. Make sure that you do not allow world access to these files! Normally, yxorp tries to find this file within the <$sysconfdir> directory that was set during configure (ie. /usr/local/etc).
<log files>	Any of yxorp's log files you specified in your configuration. Yxorp needs permission to create and write these files.

You do not have to copy any dynamic or static libraries into the jail (as yxorp will already have the handles open before changing to the jail), but for OpenSSL you will need to make a source of entropy available within the jail (usually /dev/urandom or /dev/random). Also, the certificate must be accessible within the jail.

Many people insist on statically linking applications that will run inside a chroot jail. This is not in our view preferable, because you will need to relink the executables any time a bug fix is applied to any of the libraries, and you will need to do this by hand. Chances are very big you will forget this, and thus remain vulnerable for longer than needed. Besides, dynamic libraries were invented for a reason.

The <security> tag takes the following attributes:

Attribute	***Format***	
user	number/string	The numerical userid or userid from /etc/passwd that will be set
group	number/string	The numerical group or group name from /etc/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” />

<security user=”nobody” group=”nobody” />
```

Configuring basic authentication

Yxorp can store a table of basic authentication credentials in its configuration. Basic authentication credentials can also be retrieved from LDAP, see the configuration sections on <ldapserver> and <ldapsource> for more information.

Note that basic authentication is not very secure. To recover userid and password from the encoded form is trivial. Generally, it is a good idea to enforce SSL in combination with basic authentication; however, you should be aware that browsers cache or store the userid and password, and some browsers even show the password in clear text in a configuration window.

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.

Attribute	***Format***	
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 digest authentication

Yxorp can store a table of digest authentication credentials in its configuration. Digest authentication credentials can also be retrieved from LDAP, see the configuration sections on <ldapserver> and <ldapsource> for more information.

Yxorp must be built with OpenSSL to support digest authentication.

Digest authentication is in theory much stronger than basic authentication. However, client applications are known that do not correctly implement digest authentication. For this reason, Yxorp currently does not currently enforce the nc parameter to be ever-increasing. This severly limits the cryptographic strength of the algorithm.

The <digestauth> tag takes the following attributes:

Attribute	***Format***	
realm	string	The realm. See also the first parameter on the digest_auth_check function.
userid	string	The user name
passwd	string	The password
ha1	string	The encrypted form of userid, realm, and password (set either realm and ha1 or realm, userid and passwd). Generated configurations from yxorp will always contain the ha1 form.
remove	number	If nonzero, remove this digest authentication credential from the table.

Reconfiguring

It is possible to reconfigure all digest authentication settings.

Example

```
<digestauth realm="mystuff" userid="baas" passwd="moresecure" />
```

Configuring ldap servers

Yxorp can access LDAP servers for retrieving information, for example to support the basic_auth_check() and digest_auth_check() functions. Yxorp must be built with ldap to support this.

LDAP support associated with basic authentication is considered stable functionality as of Yxorp version 2.33 and later. Support of digest authenticate with LDAP however is not; the way Yxorp implements that will change in later versions.

The <ldapserver> tag takes the following attributes:

Attribute	***Format***	
id	string	The name by which this ldapserver is referred to (usually from an ldapsource definition).
hostname	string	The name of the host that the ldap server runs on
port	string	The port on which the ldap server listens
binddn*	string	The dn (distinguished name) to use in binding to the ldap server. Only required for digest authentication support, not relevant for basic authentication support.
password*	string	The password to use in the bind operation. Only required for digest authentication support, not relevant for basic authentication support.
maxsessions*	number	The maximum number of sessions that can be concurrently open to this ldap. This should be equal to the maximum number of threads you are running. Only required for digest authentication support, not relevant for basic authentication support.

* in the table above signifies attributes that are part of the experimental support of digest authentication.

Reconfiguring

It is possible to reconfigure all ldapserver settings; however, existing sessions to an ldap server are not taken down on reconfiguration.

Example

```
// for basic authentication support
<ldapserver
  id=”ldapserver1”
  hostname=”localhost”
  port=”389”
/>

// for digest authentication support
<ldapserver
   id=”ldapserver2”
   hostname=”localhost”
   port=”389”
   binddn=”cn=root,dc=example,dc=com”
   password=”secret”
   maxsessions=”100”
/>
```

Configuring ldap sources

An ldapsource is an abstraction layer on top of ldapservers to set retrieval parameters with. Yxorp must be built with ldap support to enable the <ldapsource> tag; otherwise, the <ldapsource> tag will be rejected from the configuration.

LDAP support associated with basic authentication is considered stable functionality as of Yxorp version 2.33 and later. Support of digest authenticate with LDAP however is not; the way Yxorp implements that will change in later versions.

The ldapsource tag can be used with the basic_auth_check function; in this case, the settings on the ldapsource tag will be used to perform a simple bind to the ldap server. If however a digest_auth_check function is used, an administrative bind will be done to the ldap server using the DN set on the <ldapserver> tag. This administrative connection will then be used to retrieve the password for the userid retrieved from the digest_auth_check from the ldap server. For this to work, the ldap server needs to store the user passwords as clear text.

The <ldapsource> tag takes the following attributes:

Attribute	*Format*	
id	string	The name by which this ldapsource is referred to.
serverid	string	The serverid maps to a ldapserver definition to be used for this ldapsource
bindfmt	string	A string containing exactly one occurrence of “%s”. The username as passed from a basic authentication dialog will be substituted into this “%s”; the resulting string will be used to perform a simple bind to the ldap server mapped by the serverid attribute. The result of this simple bind will be passed back to the basic_auth_check function; also, the result will be cached according to the cachetime setting.

Attribute	***Format***	
cachetime	number	Number of seconds that the result of a simple bind to the ldap is cached by Yxorp. The minimum setting is 10 seconds.
base*	string	The base DN to be used for search operations
scope*	string	The scope to be used for search operations. Defined values are “base”, “onelevel”, and “subtree”.

* in the table above signifies attributes that are part of the experimental support of digest authentication.

Reconfiguring

It is possible to reconfigure all ldapsource settings.

Example

```
// for basic authentication
<ldapsource
   id=”ldapsource1”
   serverid=”ldapserver1”
   bindfmt=”cn=%s,ou=People,dc=example,dc=com”
   cachetime=”60”
/>

// for digest authentication

<ldapsource
   id=”ldapsource1”
   serverid=”ldapserver1”
   base=”ou=People,dc=example,dc=com”
   scope=”subtree”
/>
```

Configuring cache areas

To control the caching processes, Yxorp uses the concept of areas. A request that is eligible for caching is mapped to a “cache area”, which can be defined as a set of attributes to control how that request is cached. Cache areas are configured through the <cachearea> tag.

The <cachearea> tag takes the following attributes:

Attribute	***Format***	
area	string	The name of the cache area
maxage	number	The maximum time in seconds that an entity is cached. After this time, it will not be used for generating cached responses, and it will be removed from the cache.
maxobjectsize	number	The maximum size of an object that is accepted into this cache area. Larger objects will not be stored, and thus, no caching will be done for these.

Attribute	***Format***	
refreshage	number	If an object is older than this time in seconds, the first reference will not be served from the cache but directed to the real server. The entity returned from the real server will be stored in the cache, thus effectively refreshing the cache. This mechanism is intended to keep the cache consistenly fresh, but limit the maximum load on the real servers.

Reconfiguring

It is possible to reconfigure all cachearea settings.

Example

```
<cachearea area=”area51” maxage=”120” maxobjectsize=”128000” />
```

Configuring global settings

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

The default values for the settings are set to reasonable defaults that should work well in almost all circumstances. In general, you should probably not make any changes to the defaults, with a few exceptions that are documented elsewhere in this document. If you do decide to make changes, make sure you understand what it is you are changing, and test what the effects are before deploying to production systems.

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.
maxchunkheaderlength	number	In chunked content processing, the maximum chunk header length that will be accepted. If a header is longer than this, the request will be rejected.
maxchunkheaderreadretries	number	The maximum number of retries that will be done in trying to read a complete chunk header, in chunked header processing. If this amount of retries does not produce a valid chunk header, the request will be rejected.
maxserverconnectionattempts	number	The maximum number of times connection to a server will 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

Attribute	*Format*	
		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.
sslclientaccepttimeout	number	The time (in milliseconds) that the SSL exchanges may take in total for the SSL connection between a client and yxorp to be accepted. This includes handshaking, selecting a cypher algorithm, and exchanging keys. If this timer expires, the request will be rejected.
sslclientreadtimeout	number	The time (in milliseconds) that the SSL exchanges may take in total to read data from the SSL connection between a client and yxorp. This may include a key renegotiation. If this timer expires, the request will be rejected.
sslclientwritetimeout	number	The time (in milliseconds) that the SSL exchanges may take in total to write data to the SSL connection between a client and yxorp. This may include a key renegotiation. If this timer expires, the request will be rejected.
sslclientclosetimeout	number	The time (in milliseconds) that the SSL exchanges may take in total to close the SSL connection between a client and yxorp. If this timer expires, the request will be rejected.
sslserverconnecttimeout	number	The time (in milliseconds) that the SSL exchanges may take in total for the SSL connection between yxorp and a server to be accepted. This includes handshaking, selecting a cypher algorithm, and exchanging keys. If this timer expires, the request will be rejected.
sslserverreadtimeout	number	The time (in milliseconds) that the SSL exchanges may take in total to read data from the SSL connection between yxorp and a server.

Attribute	***Format***	
		This may include a key renegotiation. If this timer expires, the request will be rejected.
sslserverwritetimeout	number	The time (in milliseconds) that the SSL exchanges may take in total to write data from the SSL connection between yxorp and the server. This may include a key renegotiation. If this timer expires, the request will be rejected.
sslserverclosetimeout	number	The time (in milliseconds) that the SSL exchanges may take in total to close the SSL connection between yxorp and a server. If this 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. Setting this to a longer time also has the effect of causing longer response times if one or more servers in a virtualserver load balancing group becomes unavailable. See also maxserverconnectionattempts.
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.
completerequesttimeout	number	(milliseconds) Wait this long before a request must be completely received. A request is considered completely received in this context when both the start-line and message-headers have been read, or, less formally, when the <CR><LF><CR><LF> after the last request header has been read. If this timer expires, the request will be rejected.

Attribute	***Format***	
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. If you are using sticky load balancing, you may need to increase this time to the maximum time you want the sticky mapping to be remembered for idle sessions.
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 (both from client and from server), but it should also be small enough not to waste 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 97; 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).

Attribute	*Format*	
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 or other functionality that uses clientstate dvars. 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).
maxdvarchunksize	number	The maximum size of one memory allocation “chunk” in a dvar vector. In practice, this is the maximum size that the value of a variable can ever have; requests for storage over this size are rejected and cause the process requiring the variable to fail. The default is set to 16M, which should be adequate for normal use. If you need to tune this, you are probably doing something that Yxorp was not meant to do. See also maxrcmpchunksize.
maxrcmpchunksize	number	The maximum size of one memory allocation “chunk” used for the rule vm. In practice, this is the maximum size that the value of a variable can ever have; requests for storage over this size are rejected and cause the process requiring the variable to fail. The default is set to 16M, which should be adequate for normal use. If you need to tune this, you are probably doing something that Yxorp was not meant to do. See also maxdvarchunksize.
entitybuffertotalsizelimit	number	The total amount of memory that Yxorp can use for entity buffers. Yxorp uses these buffers to temporarily store HTTP entities that are subject to rule processing

Attribute	*Format*	
		(ie. inboundentity and outboundentity rules) and caching. If you don't use these rule types, and don't use caching, you do no need entity buffers. To determine the number you need if the default is insufficient, consider the average size of entities that need to be buffered, and the maximum number of threads you are using. Multiply these, and set the limit to twice that amount to be on the safe side. If yxorp runs out of entity buffer space, requests subject to rule processing will be rejected, and requests that would be eligible for caching will not be cached. As a rule of thumb, do not define more than a quarter of the physical memory that is available in your system and not used in any other processes. If you exceed this, paging may occur to the point of greatly reducing system throughput. If you don't need entity buffers, leave this setting at it's default.
entitybufferinitial	number	The initial size of an entitybuffer if the required size can not be determined from the request or response. This occurs if no Content-Length header is present in a request or response, which is generally the case with dynamically generated content. See also entitybuffertotalsizelimit and entitybufferextend.
entitybufferextend	number	The number of bytes added to an entitybuffer if it is allocated without knowledge of the required size, and if the initial allocation (ie. entitybufferinitial) is not sufficient. Since the reallocation this causes is an expensive operation, set entitybufferinitial to a value that is sufficient for something like 80-90% of all requests. Setting to 0 prevents reallocation (and causes requests that require a larger amount to fail). See also entitybuffertotalsizelimit and entitybufferinitial.
entitybuffermaxobjectsize	number	The maximum size of one entitybuffer, both for fixed-length requests and variable length requests (that are allocated through entitybufferinitial and entitybufferextend).

Attribute	*Format*	
		Requests that require a larger buffer are rejected (for inboundentity and outboundentity rules) or not cached.
cachemaxobjectsize	number	The maximum size of an HTTP entity that can be cached.
cachetotalsizelimit	number	The maximum size that the cache can use. If this is exceeded, no new entries will be added to the cache. As a rule of thumb, don't set this to more than half the physical memory that is available in your system and not used in any other processes. If you exceed this, paging may occur to the point of greatly reducing system throughput.
cachemaintenanceinterval	number	The time (in seconds) between runs of the maintenance processes for the cache.
basicauthmaintenanceinterval	number	The time (in seconds) between runs of the maintenance processes for the basic authentication cache.
enabledheadergroups	string	The header groups that are enabled. Available groups are “std” which is always enabled, “dav” for WebDAV related headers, and “msext” for Microsoft extensions to WebDAV. To enable these groups, comma-separate the groups you require.
enabledmethodgroups	string	The method groups that are enabled. Available groups are “std” which is always enabled, “rfc2616” for additional less-used methods defined in rfc2616, “dav” for WebDAV related headers, and “msext” for Microsoft extensions to WebDAV. To enable these groups, comma-separate the groups you require.
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.
clearthreadpoolgracetime	number	(milliseconds) Time that Yxorp waits for threads

Attribute	*Format*	
		to finish after a stop command has been given. After this time expires, threads will be forcibly stopped to ensure orderly shutdown.
closelogdaemongracetime	number	(milliseconds) Time that Yxorp waits for it's internal log daemon to finish after a stop command has been given. This allows the log daemon to flush its buffers and ensure that logging is complete.
starterwaitforsignaltime	number	(milliseconds) Time that the command starting Yxorp waits for its daemonized child to become active.
threadinactivitylimit	number	(seconds) Time a thread waits for work before exiting. Thread resources are released after being inactive for this length of time. Currently, this comprises LDAP sessions maintained per thread.
doclientheaderenabledchecks	number	(default non-zero) If zero, Yxorp will not check if headers are enabled for client. All headers that are defined in Yxorp's table will be accepted from a client. Do not set this to zero if you run Yxorp as a security device; instead, change individual header definitions to accommodate your needs.
doclientheaderlengthchecks	number	(default non-zero) If zero, Yxorp will not check the length of headers received from a client. Do not set this to zero if you run Yxorp as a security device; instead, change individual header definitions to accommodate your needs.
doclientheaderbufchecks	number	(default non-zero) If zero, Yxorp will not check the contents of headers received from the client against a table of allowed characters. Do not set this to zero if you run Yxorp as a security device; instead, change individual header definitions to accommodate your needs.
doclientheaderduplicatechecks	number	(default non-zero) If zero, Yxorp will not check headers received from the client for duplicates. Do not set this to zero if you run Yxorp as a security device; instead, change individual header definitions to accommodate your needs.
doclientheaderunknownchecks	number	(default non-zero) If zero, Yxorp will accept from clients any unknown header (ie. headers that are not defined in Yxorp's header table). The

Attribute	***Format***	
		unknown headers will be passed through without any checking (regardless of other settings). However, all known headers will be checked as defined by the header table or by other do[client|server]header settings. Do not set this to zero if you run Yxorp as a security device; instead, change individual header definitions to accommodate your needs.
doserverheaderenabledchecks	number	(default non-zero) If zero, Yxorp will not check if headers are enabled for server. All headers that are defined in Yxorp's table will be accepted from a server. Do not set this to zero if you run Yxorp as a security device; instead, change individual header definitions to accommodate your needs.
doserverheaderlengthchecks	number	(default non-zero) If zero, Yxorp will not check the length of headers received from a server. Do not set this to zero if you run Yxorp as a security device; instead, change individual header definitions to accommodate your needs.
doserverheaderbufchecks	number	(default non-zero) If zero, Yxorp will not check the contents of headers received from the server against a table of allowed characters. Do not set this to zero if you run Yxorp as a security device; instead, change individual header definitions to accommodate your needs.
doserverheaderduplicatechecks	number	(default non-zero) If zero, Yxorp will not check headers received from the server for duplicates. Do not set this to zero if you run Yxorp as a security device; instead, change individual header definitions to accommodate your needs.
doserverheaderunknownchecks	number	(default non-zero) If zero, Yxorp will accept from servers any unknown header (ie. headers that are not defined in Yxorp's header table). The unknown headers will be passed through without any checking (regardless of other settings). However, all known headers will be checked as defined by the header table or by other do[client|server]header settings. Do not set this to zero if you run Yxorp as a security device; instead, change individual header definitions to accommodate your needs.

Attribute	*Format*	
workdatadissectbufsize	number	(default 8192) The buffer size used to receive requests in. A complete request (excluding entity) must fit into this buffer; if not, Yxorp will reject the request. The part of the request that this applies to is made up by the request line (method name, uri, http version) and all headers, including the terminating CRLF after each line and the CRLF terminating the request. The entity (the data retrieved by a request, or the data sent in by for example a POST request) is **not** limited by this buffer size. Do not change this unless you really have requests this big, and can not change them to become smaller. Also note workdatadissectwbufsize, which must be defined larger than workdatadissectbufsize.
workdatadissectwbufsize	number	(default 10240) Internal work buffer. Must be set bigger than workdatadissectbufsize (normally at least 2K over this size). Do not change this unless you have changed workdatadissectbufsize.
maxlenrequesturi	number	(default 2048) The maximum size of the URI that is received by Yxorp. This size includes any parameter fields sent by clients (ie. http://www.example.com/some.cgi?parameter1=value1). Requests with longer URI's will be rejected by Yxorp.
maxlenheaderline	number	(default 2048) The maximum length of any header that Yxorp can process. Note that some headers (eg. Authorization:, Referer:, Location: may include URI's, and thus may become quite long. Requests with longer headers will be rejected by Yxorp.
rewritelocationheader	number	If nonzero (default), yxorp will rewrite Location: headers coming from the server that contain an absolute form URI to the domain name that the request associated with the response originally contained.
renameonrotate	number	(default zero). If nonzero, yxorp will rename its log files when rotating them. If zero, the files

Attribute	*Format*	
		will just be closed and then reopened.
basicauthmaintenanceinterval	number	(default 30) Number of seconds between runs of the basic authentication maintenance process.
basicauthcachetime	number	(default 60) Number of seconds that credentials added by use of the basicauth_add function are accepted as valid. After this time, the entry may still be present in the basicauth table; the credentials will be removed on first reference after the time elapsed, or by the next basic authentication maintenance process.
generatexcache	number	(default nonzero) If nonzero, Yxorp will generate X-Cache headers if serving from a cache was applicable to this request. If serving from a cache was not possible, no header will be generated irrespective of this setting.
viareportversionstringtoclient	number	If zero, do not report the version string in Via: headers sent to the client.
viareportpackagenametoclient	number	If zero, do not report the package name in Via: headers sent to the client. Irrelevant if viareportversionstringtoclient is nonzero.
viareporthosttoclient	number	If zero, do not generate a Via: header to send to the client. Irrelevant if viareportversionstringtoclient or viareportpackagenametoclient are nonzero.
viareportversionstringtoserver	number	If zero, do not report the version string in Via: headers sent to the server.
viareportpackagenametoserver	number	If zero, do not report the package name in Via: headers sent to the server. Irrelevant if viareportversionstringtoserver is nonzero.
viareporthosttoserver	number	If zero, do not generate a Via: header to send to the server. Irrelevant if viareportversionstringtoserver or viareportpackagenametoserver are nonzero.
workerthreadstacksize	number	(default zero) If nonzero, the value defined here will be used to override the platform's default for the stack size of a worker thread. The stack size should not be defined smaller that 128000. This setting should only be used if no alternatives exist, and the number of threads required consume all available virtual memory. Normally, either the available virtual memory should be increased (ie. ulimit -v) or the defined stack space should be lowered (ie. ulimit -s).

Attribute	***Format***	
		Note that the ulimit command only applies within the shell in which the command was issued; for ulimit to have effect, it must either be included in the script used to start yxorp, or be issued manually before yxorp is started.
statecookiename	string	This string will be prepended to the name of the cookie variable that Yxorp will set if it needs to track clients. The special value of "(auto)" derives a unique name from the hostname that Yxorp runs on.
localhostname	string	The string defined here will be used in reporting the host name of the system that Yxorp runs on; for instance in X-Cache: and Via: headers. The default special value of "(auto)" can be used to derive the name from the host name as reported by the operating system; or any other name can be configured.

The method table

In the globalconfiguration section is the table of accepted methods. The header table can be changed by including "<method>" tags in the globalconfiguration section.

By default, only a restrictive set of methods consisting of "GET", "HEAD", and "POST" is active; but many more methods are defined. These can be enabled with the "enabledmethodgroups" setting in the global configuration.

The <method> tag takes the following attributes:

Attribute	***Format***	
id	string	The name of the method, as it would appear in a request. Yxorp processes this value case-insensitively.
enable	number	Whether or not this method is enabled. Remember that a method must also be in an enabled method group to be active.
inboundentity	number	If non-zero, Yxorp expects an entity to be sent client-to-server with this method.
outboundentity	number	If non-zero, Yxorp expects an entity to be sent server-to-client with this method.
cacheable	number	If non-zero, cache processing is enabled for this method. This should only be set to 1 for GET and HEAD; caching is not defined by rfc2616 for other methods. Changing this attribute from the provided defaults may cause undesired effects.
methodgroups	string	Defines to which method groups this method belongs. Recognized values are "all", "rfc2616", "dav", and "msext". Refer to enabledmethodgroups in globalconfig.

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. The name must include a colon (':') as the last character.
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 changed to '_'; this is usually required because the minus sign has a specific meaning in the rule language.
client	number, (“reject” | ”ignore”)	If non-zero, this header is accepted from a client connection. If 0, the request is rejected if “reject” is set, or the header is discarded if “ignore” is set. Reject is default.
server	number, (“reject” | ”ignore”)	If non-zero, this header is accepted from a server connection. If 0, the request is rejected if “reject” is set, or the header is discarded if “ignore” is set. Reject is default.
maxlen	number [“, ignore”]	The maximum length of this header line (so including the header name itself). If this length is exceeded the request is rejected, unless “ignore” is set, in which case the header is discarded.
check	string, (“reject” | ”ignore”)	Takes the values “none”, “numeric”, “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; “numeric” is the set of characters between 0 and 9.
allowduplicates	number, (“reject” | ”ignore”)	Whether or not multiple occurrences of this header are allowed within a single request or response. If 0, the request is rejected if “reject” is set, or the header is discarded if “ignore” is set. Reject is default.
hop-by-hop	numeric	As defined in rfc2616, some headers have meaning only on one

Attribute	*Format*	
		session. Yxorp base code controls the values of these headers. This attribute should not be changed.
folding	numeric	As defined in rfc2616, a gateway may decide to combine headers with the same name into one, if the header value is composed of a comma-separated list. Setting the folding attribute to “1” enables this.
hdrgroups	string	Defines to which header groups this header belongs. Recognized values are “all”, “dav”, and “msext”. Refer to enabledheadergroups in globalconfig.

The status code table

In the globalconfiguration section is the table of status codes. The table can be changed by including “<statuscode>” tags in the globalconfiguration section.

By default, all status codes in rfc2616 and rfc2518 (WebDAV) are defined. You can add to these, or change the standard reason texts with the codes for your amusement.

The <method> tag takes the following attributes:

Attribute	*Format*	
id	number	The numeric status code.
reason	string	The textual explanation of the status code

Reconfiguring

All global settings can be reconfigured. However, header, method, and status code entries, once they are created, may be modified but not completely removed.

Example

The example below sets the global configuration values for clientstatecleanupmaxage to 900 seconds. The header table is updated 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 the entire request will be rejected if this request is overlength, has any characters in it outside of the reduced set, or appears multiple times in the request.

The status code of “200” is changed from it's default “OK” to say “Yo”.

```
<globalconfiguration
                    clientstatecleanupmaxage="900"
                    >
<header id=”Example”
        xlateid=”bad_example”
        maxlen=”1, reject”
        client=”1”
        server=”0, ignore”
        check=”reduced, reject”
        allowduplicates=”0, reject”
/>
<statuscode id=”200” reason=”Yo”>
</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 terminated by a binary zero (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.
- Variable names may *not* contain the minus ('-'). This is very awkward – especially because several header names contain a minus, and thus need to be translated. Normally, the default translation for a minus is an underscore ('_') and vice versa. This is default translation is done automatically for all header-type variables (those ending in a colon) that have no predefined translation (as in: defined in the table of headers in the globalconfiguration).
- Variables at runtime comprise the actual value and a number of attributes; these determine the actual header name (which can be different from the variable name in the rule), the order of the header in the list of headers to be sent, the maximum acceptable length, 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' statement, 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 RFC-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. As some HTTP clients and servers are sensitive to the order in which headers are included in requests or responses, Yxorp keeps track of the original order of the headers. During processing of a request or response, Yxorp sets the order number of the first header in a request or response 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.

  Yxorp uses duplicate variables to accommodate the fact that many HTTP clients and servers do not support header folding, as described in rfc2616. Headers are converted into variables exactly as they are sent out by clients and servers; if a client sends two header lines for the same header line, that is exactly how these will be represented as header variables.

All variables and variable attributes may be referenced or modified in a rule program. Variables can be directly accessed in the rule language; duplicate variables and attributes can be accessed through function calls and rule language constructs.

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 counterintuitive 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

Name	***Use***
	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.

Since version 2.27, a rule may be empty (i.e. contain no statements).

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.

Statement

```
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. In conjunction with the indirection operator “$”, this can be used to perform an operation on each element in a list of variables.

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

basicauth_add(source, realm) - cache basicauth credentials

The basicauth_add function adds the basicauth credentials, presented in the Authorization: header present in the current request, to the basicauth cache table. The entry will be valid for a limited time only, and after it has become invalid it will be removed from the cache table by the basicauth maintenance process.

The time the cached entry is valid is defined in the globalconfiguration item basicauthcachetime.

basicauth_check(source, realm) - check basicauth credentials

The basicauth_check function checks the credentials presented in the Authorization: header with the current request with the basicauth table. If a match is found, either as a hard coded entry or a cached entry, the function returns true; else, it returns false.

basicauth_getpass() - read password from basicauth credentials

The basicauth_getpass function reads the password from the base64-encoded basic authentication credentials passed with the current request.

basicauth_getuser() - read userid from basicauth credentials

The basicauth_getuser function reads the userid from the base64-encoded basic authentication credentials passed with the current request.

basicauth_reject(realm) - cause a 401 status code

The basicauth_reject function sets up the response for the current request to contain a 401 Unauthorized status code, including a WWW-Authenticate: header containing the realm as set as a parameter to the function. Also, the reject reason is set to the text “basicauth401”.

Please note that it is generally required to explicitly end the rule execution after calling this function, for instance by using the return statement. Failure to do so may have unintended consequences if the status code, reject reason, or WWW-Authenticate header are changed by further statements.

clearclientstate() – remove the client state

Synonym for killclientstate(). This function immediately removes all state information that is kept in the clientstate table for the current request. If a state cookie was previously sent to the client, no attempt is made to remove that state cookie from the client. A next request from this client will either carry a state cookie value that does not map to a valid state or no state cookie at all, and in both cases cause a new state to be created.

In many cases it is preferable to not immediately remove the state information. This situation would for instance occur while running a rule for an URI for a logoff page, which refers to other entities like graphics that still have to be retrieved. If in such a case the state information was immediately removed, retrieving the graphics entities would cause a new state entry to be created in the client state table. For this reason, there is also the setclientstatefastage() function.

clearsticky() – remove a sticky mapping from the client state

Synonym for killsticky(). This function removes the sticky mapping for the current value of the Host: header (as exists at the time of the call, and possibly changed by assignments to the Host: variable in this or previously run rules).

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(range) – 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. The contains function will be deprecated in some future version.

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.

Note you can also use a regular expression; this is a lot more flexible. The contains_characters function will be deprecated in some future version.

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. The containscase function will be deprecated in some future version.

digest_auth_check(realm, authentication source) – enforce digest authentication

checks if valid digest authentication credentials are present in the request (i.e. the Authorization: Digest header). If this header 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.

If the authentication credentials are present in the request, and if the authentication source parameter has the exact value of “local”, the request credentials will be checked to the internal digestauth table (see the decription of the <digestauth> table in the configuration reference). The internal table uses the realm name to enable different realms to use different userid/password combinations, or to enable a given user to be granted access to one resource, but not another.

Versions earlier than 2.33 contained an experimental ldap interface associated with this function call; this has been removed.

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 digest_auth_check fails; this must be done explicitly; normally, a return statement should be used to prevent execution of statements following the call to digest_auth_check. Typical use is as:

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

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_dupvar is especially useful when processing a header for which duplicates may exist (i.e. multiple headers with the same name), as in the following example:

```
// the following is wrong:
if (Cookie: -/blabla/) {
   ... only the first Cookie header is checked
}
// since headers that may occur multiple times are stored in a dupvar group,
// as Cookie:, Cookie:0, Cookie:1 etc.

// check for all cookie headers
foreach i (enumerate_dupvar("Cookie:")) {
   if ($i -/blabla/) {
      ... the cookie contains “blabla”
   }
}
```

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.

Note you can also use a regular expression; this is a lot more flexible. The containscase function will be deprecated in some future version.

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

equal performs a case insensitive comparison of two variables.

Note you can also use a regular expression; this is a lot more flexible. The containscase function will be deprecated in some future version.

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. Currently two values for attr are recognized: “reqhdr” to test if a variable has been created by Yxorp as a request header variable, and “rsphdr”, for a response header variable.

getcachearea() – return cache area name

Returns the cache area configured for this request.

getclientip() – return the client ip address

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

The value of the returned ip address may have been changed as a result of processing an X-Forwarded-For header, if Yxorp is configured to do so. See the chapter on XFF processing for details.

getclientcipherbits() – return the key length used in the current SSL connection

getclientcipherbits returns the key length used in the current SSL connection between a client and Yxorp.

If the current connection is not an SSL connection, or Yxorp is built without SSL support, false is returned.

getclientciphername() – return the cipher used in the current SSL connection

getclientciphername returns the name of the cipher used in the current SSL connection between a client and Yxorp.

If the current connection is not an SSL connection, or Yxorp is built without SSL support, false is returned.

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 when Yxorp is handling a large volume of requests.

getclientcertfailcode() – return the client certificate failure code

getclientcertfailcode returns the code, as defined by OpenSSL, that identifies the failure that occurred while verifying the client certificate presented on the current SSL connection between a client and Yxorp. The code is obtained from the OpenSSL function SSL_get_verify_result.

If the current connection is not an SSL connection, or Yxorp is built without SSL support, false is returned.

getclientcertfailmsg() – return the client certificate failure message

getclientcertfailcode returns the message, as produced by OpenSSL, that identifies the failure that occurred while verifying the client certificate presented on the current SSL connection between a client and Yxorp. The message is obtained from the OpenSSL function X509_verify_cert_error_string.

If the current connection is not an SSL connection, or Yxorp is built without SSL support, false is returned.

getclientcertinfo() – return client certificate information

getclientcertinfo returns information about the client certificate presented on the current SSL connection between a client and Yxorp. The information is in the form returned by the OpenSSL call X509_NAME_oneline.

If the current connection is not an SSL connection, or Yxorp is built without SSL support, false is returned.

getclientcertissuerinfo() – return client certificate issuer information

getclientcertissuerinfo returns information about the issuer of the client certificate presented on the current SSL connection between a client and Yxorp. The information is in the form returned by the OpenSSL call X509_NAME_oneline.

If the current connection is not an SSL connection, or Yxorp is built without SSL support, false is returned.

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 passed as a string. If the variable and the client state exist, the data will be returned; otherwise, an empty string (i.e. false) will be returned.

getentity() – return the current entity

getentity returns the current entity. If there is none, false is returned. Normally, this function can only be used in inboundentity or outboundentity rules.

Note that the rule vm, as it works on zero-terminated strings, cannot currently process binary entities. Thus, if you are processing a request that carries a binary entity (ie. a picture, jpeg, bitmap or other) the entity will be returned by this function, but following statements can not process it completely.

getinboundentityrule() – return inboundentity rule

getinboundentity returns the name of the inboundentity rule. This attribute normally set by the listener definition, and is reset to that value on session reuse; it may have been changed by a call to setinboundentityrule() executed in the context of the current request.

getlength(variablename) – return the length of a variable

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

getlistenerid() – return 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.

getoutboundentityrule() – return outbound entity rule

Returns the name of the outbound entity rule. This attribute normally set by the listener definition, and is reset to that value on session reuse; it may have been changed by a call to setoutboundentityrule() executed in the context of the current request.

getrejectrule() – return reject rule

Returns the name of the reject rule. This attribute normally set by the listener definition, and is reset to that value on session reuse; it may have been changed by a call to setrejectrule() executed in the context of the current request.

getrequestnumber() – returns the ordinal number of this request

getrequestnumber returns the ordinal number of this request, since the start of Yxorp.

getresponserule() – return response rule

Returns the name of the response rule. This attribute normally set by the listener definition, and is reset to that value on session reuse; it may have been changed by a call to setresponserule() executed in the context of the current request.

getservercertfailcode() – return the server certificate failure code

getservercertfailcode returns the code, as defined by OpenSSL, that identifies the failure that occurred while verifying the server certificate presented on the current SSL connection between Yxorp and a (real) server. The code is obtained from the OpenSSL function SSL_get_verify_result.

If the current connection is not an SSL connection, or Yxorp is built without SSL support, false is returned.

getservercertfailmsg() – return the server certificate failure message

getservercertfailcode returns the message, as produced by OpenSSL, that identifies the failure that occurred while verifying the server certificate presented on the current SSL connection between Yxorp and a (real) server. The message is obtained from the OpenSSL function X509_verify_cert_error_string.

If the current connection is not an SSL connection, or Yxorp is built without SSL support, false is returned.

getservercertinfo() – return server certificate information

getservercertinfo returns information about the server certificate presented on the current SSL connection between Yxorp and a (real) server. The information is in the form returned by the OpenSSL call X509_NAME_oneline.

If the current connection is not an SSL connection, or Yxorp is built without SSL support, false is returned.

getservercertissuerinfo() – return server certificate issuer information

getservercertissuerinfo returns information about the issuer of the server certificate presented on the current SSL connection between Yxorp and a (real) server. The information is in the form returned by the OpenSSL call X509_NAME_oneline.

If the current connection is not an SSL connection, or Yxorp is built without SSL support, false is returned.

getservercipherbits() – return the key length used in the current SSL connection

getservercipherbits returns the key length used in the current SSL connection between Yxorp and a (real) server.

If the current connection is not an SSL connection, or Yxorp is built without SSL support, false is returned.

getserverciphername() – return the cipher used in the current SSL connection

getserverciphername returns the name of the cipher used in the current SSL connection between Yxorp and a (real) server.

If the current connection is not an SSL connection, or Yxorp is built without SSL support, false is returned.

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. This difference is at least explained by the network time between Yxorp and the client, but also included is the time it takes the client to render a page.

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.

isccertverified() – return verification state of client certificate

isccertverified() returns true if the certificate used on the SSL session between Yxorp and a (real) server could be verified. If the session does not use SSL, or if Yxorp was built without SSL support, false is returned.

isscertverified() – return verification state of server certificate

isscertverified() returns true if the certificate used on the SSL session between a client and Yxorp could be verified. If the session does not use SSL, or if Yxorp was built without SSL support, false is returned.

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

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

killclientstate() – remove the client state

Synonym for clearclientstate().

killsticky() – remove a sticky mapping from the client state

Synonym for clearsticky().

killvar(variablename) – remove a variable

Synonym for killvariable()

killvariable(variablename) – remove a variable

killvariable kills (removes) the variable of which the name is passed as a parameter from the variable pool. This is especially useful if you want to keep a header from being passed on.

```
// Don't send the Server: header at all
killvariable(“Server:”);

// Don't send any Cookie: headers
foreach i (enumerate_dupvar("Cookie:")) {
   killvariable($i);
}
```

ldap_bind(dn, pass)

ldap_bind will execute a bind operation to the ldap server set by a previous ldap_set_serveruri function, to the dn that is passed and using the pass parameter.

ldap_init()

ldap_init causes the internal structures required for interaction with an ldap server to be initialized, and must be called before any ldap library function is invoked through any of the ldap functions other than ldap_set_<whatever>. Although this function may become redundant in a version after Yxorp 2.33, it is good practice to call this function before invoking ldap_search, ldap_bind or any other ldap action function.

ldap_search(filter)

Initiates a seach operation towards the ldap server that has been identified with the ldap_set_serveruri function, respecting the parameters set by the ldap_set_searchbase, ldap_set_searchdn, ldap_set_searchpw, ldap_set_searchscope functions. These parameters will be used to do a bind to the ldap server (a so called 'administrative bind'); the connection that is thus created will be used to do the search operation. The search operation may return one or more dn that match the search criteria set both by the filter and by the ldap_set_<x> functions. These dn will be set in dupvar variables with the basename of “dn”; the ldap_search function returns the number of dupvars that have been set in this way. Normally, for a successful implementation, the number of dn's returned should be exactly one if the user is found or zero if not; all other values (as in, multiple definitions in the ldap server could match to this user definition) are not generally useable.

The filter parameter is a string representation of the filter to apply in the search. The string should conform to the format specified in RFC 4515, as extended by RFC 4526. For instance, "(cn=Jane Doe)".

ldap_set_seachbase(base)

Sets the base dn that will be used in search operations initiated by the ldap_search call following the use of this function.

ldap_set_searchdn(dn)

Sets the dn that will be used in search operations initiated by the ldap_search call following the use of this function.

ldap_set_searchpw(pw)

Sets the password that will be used in search operations initiated by the ldap_search call following the use of this function.

ldap_set_searchscope(scope)

This function sets the seach scope parameter into the ldap structure for use in an ldap_seach function. The scope parameter should be one of the string values LDAP_SCOPE_BASE, to search the object itself, LDAP_SCOPE_ONELEVEL, to search the object's immediate children, LDAP_SCOPE_SUBTREE, to search the object and all its descendants, or LDAP_SCOPE_CHILDREN, to search all of the descendants. Note that the latter requires that the server support the LDAP Subordinates Search Scope extension. If the string value does not match one of the values listed here, exactly and case correct, then the default of LDAP_SCOPE_SUBTREE will be applied.

See the documentation of the ldap client library and ldap server for further details.

ldap_set_serveruri(uri)

This function sets the uri to be used in a subsequent ldap interaction. The format of the uri is as described in the documentation of the ldap_initialize function. The value presented here will be passed to the ldap_initialize function in the ldap client library; see the documentation of the ldap client library for further details.

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.

rdlog(numerical value) – set rdlog master flag for the current request

Calling rdlog() with a non-zero value causes the master rdlog flag to be set for this request. If zero, no request detail log actions will be taken for this request. The effect varies according to in which type of rule the call is executed.

rdlogfinalinternaldata(numerical value) – include dump of internal structures

Calling rdlogfinalinternaldata() with a non-zero value causes Yxorp's internal data structures for this request to be logged at the end of the request. The status of the flag is examined at the end of the request.

rdlogerror(numerical value) – set rdlog error flag for the current request

Calling rdlogerror() with a non-zero value causes the rdlog error flag to be set for this request. If zero, the request detail log actions will be taken for this request, irrespective of the result of the request. If non-zero, only requests ending in an error state will be logged to the rdlog. The status of the flag is examined at the end of the request.

rdlogreceivedrequest(numerical value) – include dump of received request

Calling rdlogreceivedrequest() with a non-zero value causes the data that was received from the client to be logged. The status of the flag is examined during the request phase, but (currently) before any rule executes. Hence, this function makes no sense (yet).

The value passed on this function is a bitmask. See the description of <requestdetaillog> for the meaning of the bits.

rdlogreceivedresponse(numerical value) – include dump of received response

Calling rdlogreceivedresponse() with a non-zero value causes the data that was received from the server to be logged. The status of the flag is examined during the response phase.

The value passed on this function is a bitmask. See the description of <requestdetaillog> for the meaning of the bits.

rdlogtransmittedrequest(numerical value) – include dump of transmitted request

Calling rdlogtransmittedrequest() with a non-zero value causes the data that was sent to the server to be logged. The status of the flag is examined during the request phase.

The value passed on this function is a bitmask. See the description of <requestdetaillog> for the meaning of the bits.

rdlogtransmittedresponse(numerical value) – include dump of transmitted response

Calling rdlogtransmittedresponse() with a non-zero value causes the data that was sent to the server to be logged. The status of the flag is examined during the response phase.

The value passed on this function is a bitmask. See the description of <requestdetaillog> for the meaning of the bits.

reject(reason) – reject request

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

sanitizexforwardedfor(trusted proxy IP ranges) – check a X-Forwarded-For: header

sanitizexforwardedfor scans a received X-Forwarded-For header to verify that all proxy addresses are in the list of IP ranges passed as arguments. A variable number of arguments can be passed.

See the chapter on XFF processing for details.

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.

setcachearea(cachearea) – set cache area name

Sets the cache area to use for this request.

setclientipfromxff() – set client ip from X-Forwarded-For: header

If a X-Forwarded-For: header exists, and it is sanitized by calling one of the sanitizexforwardedfor**() functions, calling setclientipfromxff() will have the effect that yxorp's administration of the client ip address will be changed to the client address taken from the X-Forwarded-For: header. For instance, if you call setclientipfromxff(), then getclientip() or one of the clientiprange**() functions, then these functions will use the client address from the X-Forwarded-For: header.

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).

setclientstatefastage(seconds) – schedule removal of the client state

In many cases it is preferable to not immediately remove the state information. This situation would for instance occur while running a rule for an URI for a logoff page, which refers to other entities like graphics that still have to be retrieved. If in such a case the state information was immediately removed, retrieving the graphics entities would cause a new state entry to be created in the client state table. For this reason, there is also the setclientstatefastage() function.

After the call, the client state will remain active for at least the specified number of seconds. After that time expires, the client state will be cleared by the first client state cleanup run; this cleanup run may occur anywhere between immediately after the seconds expire, and the clientstatecleanupinterval set in the globalconfiguration.

Unless you have clients on very slow links or very complicated logoff pages, 10 seconds should be a reasonable setting for the above scenario.

setclientstateid(id) – manually set clientstate id for this request

When a client state is needed, normally Yxorp will automatically create a client state id, and use a cookie to store the value of the client state id in the client browser. If this is not possible, for instance because the client has cookies disabled, it is possible to use some other attribute of the request as a client state id.

When manually maintaining the client state, the setclientstateid function must be called in a request rule for all requests that need access to the client state (for instance when using the client state for sticky load balancing, this means all all requests). The client state id passed to the functions must be the same exact string for all cases that need mapping to the same client state. Care must be taken that different clients will not use the same state; otherwise, unpredictable results may occur in dealing with the client state.

When manually maintaining client states, the automatic creation of client states must be disabled. See setclientstateidgenerate on how to do this.

setclientstateidgenerate(value) – set automatic client state id generation

When a client state is needed, normally Yxorp will automatically create a client state id, and use a cookie to store the value of the client state id in the client browser. If this must be prevented, for instance because the setclientstateid function is used to manually create the client state id, the setclientstateidgenerate function must be called with a zero argument. All non-zero arguments will result in the default behavior where Yxorp will generate the client state id.

setconnectservertimeout(timeout) – set serverend timeout value

Sets the timeout (in milliseconds) that is used for connecting to a server. The value is valid only during the processing of the request that is being executed. The default value, that applies if no call to this function is executed, is taken from <globalconfiguration>

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).

setinboundentityrule(rulename) – set inbound entity rule

Sets the outbound entity rule to run. This attribute normally set by the listener definition, and is reset on session reuse.

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.

setoutboundentityrule(rulename) – set outbound entity rule

Sets the outbound entity rule to run. This attribute normally set by the listener definition, and is reset on session reuse.

setreadfromservertimeout(timeout) – set serverend timeout value

Sets the timeout (in milliseconds) that is used for reading from a server. The timeout applies to each read operation, not to the processing of the entire response; the server has to send at least some data within the timeout. The value is valid only during the processing of the request that is being executed. The default value, that applies if no call to this function is executed, is taken from <globalconfiguration>

setresponserule(rulename) – set response rule

Sets the response rule to run. This attribute normally set by the listener definition, and is reset on session reuse.

setrejectrule(rulename) – set reject rule to run

Sets the reject rule to run. This attribute normally set by the listener definition, and is reset on session reuse.

setsendxforwardedfor() – send X-Forwarded-For: header to server

Calling setsendxforwardedfor() in a request rule causes Yxorp to send a X-Forwarded-For: header to a server. If the header is considered sanitized (i.e. after calling the sanitizexforwardedfor() function) the sanitized part of the header sent in from a client will be included, and Yxorp will append the IP address of where it received the request from to the header.

If no X-Forwarded-For was sent in from the client, if X-Forwarded-For was disabled as a header in Yxorp's header table, or if no call to the sanitizexforwardedfor() function was executed, Yxorp will send a X-Forwarded-For containing the IP address of the client connection.

See the chapter on XFF processing for details.

setsslbackend() – set ssl backend

Forces the server session for this request to use SSL.

setsslserverconnecttimeout(timeout) – set serverend ssl timer

This function sets the timer that is used when connecting to a server. The entire connect operation, including SSL handshake, certificate validation etc. must complete within this time. The timeout is specified in milliseconds, and valid only during the processing of the request that is being executed. The default value, that applies if no call to this function is executed, is taken from <globalconfiguration>.

setsslserverreadtimeout(timeout) – set serverend ssl timer

This function sets the timer that is used when reading from a server. The timeout is specified in milliseconds, and valid only during the processing of the request that is being executed. The default value, that applies if no call to this function is executed, is taken from <globalconfiguration>.

setsslserverwritetimeout(timeout) – set serverend ssl timer

This function sets the timer that is used when writing to a server. The timeout is specified in milliseconds, and valid only during the processing of the request that is being executed. The default value, that applies if no call to this function is executed, is taken from <globalconfiguration>.

setsslserverclosetimeout(timeout) – set serverend ssl timer

This function sets the timer that is used when closing a server connection. The timeout is specified in milliseconds, and valid only during the processing of the request that is being executed. The default value, that applies if no call to this function is executed, is taken from <globalconfiguration>.

setsticky() – set sticky flag

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

settargetserver(id) – set target server

Normally Yxorp uses the value of the Host: header to determine which server to send a request to. If however the settargetserver() function is used, Yxorp will send a request to the server whose id is passed as a parameter. This can be useful if a large number of virtual hosts is defined on a group of real servers, for which Yxorp is to do load balancing; in this case, Yxorp does not need to be aware of each individual virtual host domain.

setwritetoservertimeout(timeout) – set serverend timeout value

Sets the timeout (in milliseconds) that is used for writing to a server. The timeout applies to each write operation, not to the processing of the entire request. The value is valid only during the processing of the request that is being executed. The default value, that applies if no call to this function is executed, is taken from <globalconfiguration>

statuscodemsg(code) – returns status code textual message

The text message for the HTTP response code, as defined in the global configuration, is returned.

strremove(haystack, needle) – remove matching string

The strremove function returns a copy of the string “haystack” from which the first occurrence of the string “needle” has been removed. If “haystack” does not contain any occurrence of the string “needle”, the entire “haystack” is returned.

The strremove function is especially useful in conjunction with regular expressions, as in the following example:

```
a="aapnootmies";
a -/(aap)(noot)(mies)/;

tmp1=strremove(a, _1);
// resulting value of tmp1 is “nootmies”
```

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.

writetofile(filename, value) – write to a file

writetofile takes the value of the second argument, and writes that to a file. The name of the file is set to the value of the first argument.

As this is one of the very few reasons for Yxorp to access the file system and thus poses security risks, this function may be removed for security reasons. If you do not want this to happen, inform the authors.

writeentitytofile(filename) – writes entity to a file

writeentitytofile writes the current entity (if any) to a file. The name of the file is set to the value of the first argument.

As this is one of the very few reasons for Yxorp to access the file system and thus poses security risks, this function may be removed for security reasons. If you do not want this to happen, inform the authors.

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 very fast - it is designed and tuned to process strings quickly, and it does. The instructions that the virtual machine processes are almost all very simple and straightforward. Even though the rule compiler is simple and straightforward, optimizations are automatically done for both trivial and expensive virtual machine instructions. Most data manipulations are accomplished just by moving pointers around. Temporary variables are allocated in scratch memory pools; this speeds up execution because memory allocation is done few times if not just once; 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, or at handling data that is poorly expressed as strings. Calculations require several conversions. Number handling is quite simplistic; if a string value is used in a numeric operation, the value will just default to 0 instead of causing an error (this behavior is likely to change in a future version). The value of false (i.e. the empty string) may be assumed to be treated as 0.

Also noteworthy, since the introduction of entity rules, is the fact that the virtual machine works on zero-terminated strings. Entities that contain binary zeros cannot be processed directly; specific function calls are necessary to overcome this weakness. These functions need by definition to be included as base code; it is not (yet) possible to define these without including source code.

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 somewhat impacted; however, only in very special cases you would have to worry about the performance penalty incurred with enabling debugging in the build. Unless you are running special hardware (embedded systems, router hardware, etc) or are running a very high volume site (as in, sustained throughput of over 100Mbit) you should by default include the debug code; normal current hardware will handle Yxorp's debugging code easily. You may have to think about dealing with the debugging output though - this will easily run into gigabytes of data.