UNICORE GATEWAY

UNICORE Team

Document Version:	1.1.0
Component Version:	7.2.0
Date:	09 02 2015

This work is co-funded by the EC EMI project under the FP7 Collaborative Projects Grant Agreement Nr. INFSO-RI-261611.

Contents

This user manual provides information on running and using the UNICORE gateway. Please note also the following places for getting more information:

UNICORE Website: http://www.unicore.eu

Support list: unicore-support@lists.sf.net

Developer's list: unicore-devel@lists.sf.net

1 Introduction

The Gateway is the entry point into a UNICORE site, routing HTTPS traffic to servers like UNICORE/X. It is installed in front of any networking firewall. It authenticates incoming messages and forwards them to their intended destination. The Gateway receives the reply and sends it back to the client. In this way, only a single open port in a site's firewall has to be configured.

LIMITATIONS

This forwarding process only works for "most" HTTP requests, and is not a complete HTTP reverse proxy implementation. For example, it is not possible to run a full, complex web application like the UNICORE portal "behind" the gateway. Check the respective components's manual whether it can be run behind the Gateway.

In effect, traffic to a *virtual* URL, e.g. *https://mygateway:8088/Alpha* is forwarded to the real URL, e.g. *https://host1:7777*.

The mappings of virtual URL to real URL for the available sites are listed in a configuration file `connections.properties`. Additionally, the gateway supports dynamic registration of sites.

The second functionality of the gateway is authentication of incoming requests. Connections to the gateway are made using client-authenticated SSL, so the gateway will check whether the caller presents a certificate issued by a trusted authority. Information about the authenticated client is forwarded to services behind the gateway in UNICORE proprietary format (as SOAP header element).

IMPORTANT NOTE ON PATHS
The UNICORE Gateway is distributed either as a platform independent and portable bundle (as a part of the UNICORE core server package) or as an installable, platform dependent package format such as RPM.
Depending on the installation method, the paths to various Gateway files are different. If installing using a distribution-specific package the following paths are used:

```
CONF=/etc/unicore/gateway
BIN=/usr/sbin
LOG=/var/log/unicore/gateway
```

If installing using the portable bundle all Gateway files are installed under a single directory. Path prefixes then are as follows, where INST is a directory where the Gateway was installed:

```
CONF=INST/conf
BIN=INST/bin
LOG=INST/log
```

The above variables (CONF, BIN and LOG) are used throughout the rest of this manual.

2 Installation

The UNICORE Gateway is distributed in the following formats:

1. As a part of platform independent installation bundle called UNICORE core server bundle. The UNICORE core server bundle is provided in two forms: one with a graphical installer and one with a command line installer.
2. As a binary, platform-specific package available currently for RedHat (Centos) and Debian platforms. Those packages are not tested on all possible platforms, but should work without any problems with other versions of similar distributions, e.g. SL6, Centos, or Fedora.

2.1 Installation from the core server bundle

Download the core server bundle from the UNICORE project website.

If you use the graphical installer, follow the on-screen instructions and do not forget to enable the Gateway checkbox when prompted.

If you use the console installer, please review the README file available after extracting the bundle. You don't have to change any defaults as the Gateway is installed by default.

2.2 Installation from a Linux package (rpm or deb)

Use your distribution's package manager to install.

3 Upgrading

The general update procedure is presented below, with possible variations:

1. Stop the old Gateway.
2. Update the server package. This step mostly applies for RPM/DEB managed installations. For Quickstart installation it is enough to replace the *.jar files with the new ones.
3. Start the newly installed Gateway.
4. Verify log file and fix any problems reported.

4 Configuration

The gateway is configured using a set of configuration files, which reside in the `CONF` subdirectory.

4.1 Java and environment settings: startup.properties

This file contains settings related to the Java VM, such as the Java command to use, memory settings, library paths etc.

4.2 Configuring sites: connections.properties

This is a simple list connecting the names of sites and their physical addresses. An example is:

```
DEMO-SITE = https://localhost:7777
REGISTRY = https://localhost:7778
```

If this file is modified, the gateway will re-read it at runtime, so there is no need to restart the gateway in order to add or remove sites.

Optionally administrator can enable a possibility for dynamic site registration at runtime, see Section 4.4.2 for details. Then this file should contain only the static entries (or none if all sites register dynamically).

Further options for back-end sites configuration are presented in Section 6.

4.3 Main server settings: gateway.properties

Use the `gateway.hostname` property to configure the network interface and port the gateway will listen on. You can also select between https and http protocol, though in almost all cases https will be used.

Example:

```
gateway.hostname = https://192.168.100.123:8080
```

Note
If you set the host to `0.0.0.0`, the gateway will listen on all network interfaces of the host machine, else it will listen only on the specified one.

If the scheme of the hostname URL is set to https, the Gateway uses the configuration data from `security.properties` to configure the HTTPS settings.

4.4 Certificate-less end-user access

With UNICORE 7, it is possible that end-users do not have client certificate. To enable them to connect, the Gateway needs to accept TLS connections without a client certificate. To configure this, set the following in `gateway.properties`

```
gateway.httpServer.requireClientAuthn=false
```

4.4.1 Scalability settings

To fine-tune the operational parameters of the embedded Jetty server, you can set advanced HTTP server parameters. See Section 4.4.5 for details. Among others you can use the non-blocking IO connector offered by Jetty, which will scale up to higher numbers of concurrent connections than the default connector.

The gateway acts as a https client for the VSites behind it. The number of concurrent calls is limited, and controlled by two parameters:

```
# maximum total number of concurrent calls to Vsites
gateway.client.maxTotal=100
# total number of concurrent calls per site
gateway.client.maxPerService=20
```

You can also control the limit on the maximum SOAP header size which is allowed by the gateway. **Typically you don't have to touch this parameter**. However if your clients do produce very big SOAP headers and gateway blocks them, you can increase the limit. Note that such a giant SOAP header usually means that the client is not behaving in a sane way, e.g. is trying to perform a DoS attack.

```
# maximum size of an accepted SOAP header, in bytes
gateway.soapMaxHeader=102400
```

Note that gateway may consume this amount of memory (plus some extra amount for other data) for each opened connection. Therefore, this value multiplied by the number of maximum allowed connections, should be significantly lower, then the total memory available for the gateway.

4.4.2 Dynamic registration of Vsites

Dynamic registration is controlled by three properties in `CONF/gateway.properties` file:

```
gateway.registration.enable=true
```

If set to true, the gateway will accept dynamic registrations which are made by sending a HTTP POST request to the URL /VSITE_REGISTRATION_REQUEST

Filters can be set to forbid access of certain hosts, or to require certain strings in the Vsite addresses. For example:

```
gateway.registration.deny=foo.org example.org
```

will deny registration if the remote hostname contains *foo.org* or *example.org*. Conversely,

```
gateway.registration.allow=mydomain.org
```

will only accept registrations if the remote address contains *mydomain.org*. These two (deny and allow) can be combined.

4.4.3 Web interface ("monkey page")

For testing and simple monitoring purposes, the gateway displays a website showing detailed site information (the details view can be disabled). Once the Gateway is running, open up a browser and navigate to https://<gateway_host>:8080 (or whichever URL the gateway is running on). As the gateway usually runs using SSL, you will need to import a suitable client certificate into your web browser.

A HTML form for testing the dynamic registration is available as well, by clicking the link in the footer of the main gateway page.

To disable the Vsite details page, set

```
gateway.disableWebpage=true
```

4.4.4 Main options reference

4.4.5 Configuring advanced HTTP server settings

UNICORE servers are using an embedded Jetty HTTP server. In most cases the default configuration should be perfectly fine. However, for some sites (e.g. experiencing an extremely high load) HTTP server settings can be fine-tuned with the following parameters.

Example

Note
Various UNICORE modules use different property prefixes. Here we don't put any, but in practice you have to use the prefix (see the reference table above for the actual prefix). Also properties might need to be provided using different syntax, as XML.

In this example we will turn on compression of all responses bigger then 50kB (assuming that the client supports decompression). Additionally we are limiting the number of concurrent clients that can be served to more or less 50, while keeping 10 threads ready all the time to server new clients quicker.

```
jetty.gzip.enable=true
jetty.gzip.minGzipSize=51200
jetty.maxThreads=50
jetty.minThreads=10
```

4.5 security.properties

In the `security.properties` file, the trust store and gateway credential is configured.

4.5.1 Configuring PKI trust settings

Public Key Infrastructure (PKI) trust settings are used to validate certificates. This is performed, in the first place when a connection with a remote peer is initiated over the network, using the SSL (or TLS) protocol. Additionally certificate validation can happen in few other situations, e.g. when checking digital signatures of various sensitive pieces of data.

Certificates validation is primarily configured using a set of initially trusted certificates of so called Certificate Authorities (CAs). Those trusted certificates are also known as *trust anchors* and their collection is called a *trust store*.

Except of *trust anchors* validation mechanism can use additional input for checking if a certificate being checked was not revoked and if its subject is in a permitted namespace.

UNICORE allows for different types of trust stores. All of them are configured using a set of properties.

- *Keystore trust store* - the only format supported in older UNICORE versions. Trusted certificates are stored in a single binary file in JKS or PKCS12 format. The file can be only manipulated using a special tool like JDK *keytool* or openssl (in case of PKCS12 format). This format is great if trust store should be in a single file or when compatibility with other Java solutions or older UNICORE releases is desired.
- *OpenSSL trust store* - allows to use a directory with CA certificates stored in PEM format, under precisely defined names: the CA certificates, CRLs, signing policy files and namespaces files are named <hash>.0, <hash>.r0, <hash>.signing_policy and <hash>.namespaces. Hash is the old hash of the trusted CA certificate subject name (in Openssl version > 1.0.0 use -suject_hash_old switch to generate it). If multiple certificates have the same hash then the default zero number must be increased. This format is the same as used by other then UNICORE popular middlewares as Globus and gLite. It is suggested when a common trust store with such middlewares is needed.
- *Directory trust store* - the most flexible and convenient option, suggested for all remaining cases. It allows to use a list of wildcard expressions, concrete paths of files or even URLs to remote files as a set of trusted CAs and in the same way for the CRLs. With this trust store administrator can simply configure all files (or all with a specified extension) in a directory to be used as a trusted certificates.

In all cases trust stores can be (and by default are) configured to be automatically refreshed.

Property name	**Type**	**Default value / mandatory**	**Description**
`gateway.truststore.allowProxy`	[ALLOW, DENY]	`ALLOW`	Controls whether proxy certificates are supported.
`gateway.truststore.type`	[keystore, openssl, directory]	*mandatory to be set*	The truststore type.
`gateway.truststore.updateInterval`	integer number	`600`	How often the truststore should be reloaded, in seconds. Set to negative value to disable refreshing at runtime. *(runtime updateable)*
--- Directory type settings ---			
`gateway.truststore.directoryConnectionTimeout`	integer number	`15`	Connection timeout for fetching the remote CA certificates in seconds.

Property name	Type	Default value / mandatory	Description
gateway.truststore.directoryDiskCachePath	filesystem path	-	Directory where CA certificates should be cached, after downloading them from a remote source. Can be left undefined if no disk cache should be used. Note that directory should be secured, i.e. normal users should not be allowed to write to it.
gateway.truststore.directoryEncoding	[PEM, DER]	PEM	For directory truststore controls whether certificates are encoded in PEM or DER.
gateway.truststore.directoryLocations.*	list of properties with a common prefix	-	List of CA certificates locations. Can contain URLs, local files and wildcard expressions. *(runtime updateable)*
--- Keystore type settings ---			
gateway.truststore.keystoreFormat	string	-	The keystore type (jks, pkcs12) in case of truststore of keystore type.
gateway.truststore.keystorePassword	string	-	The password of the keystore type truststore.
gateway.truststore.keystorePath	string	-	The keystore path in case of truststore of keystore type.
--- Openssl type settings ---			
gateway.truststore.opensslNewStoreFormat	[true, false]	false	In case of openssl truststore, specifies whether the trust store is in openssl 1.0.0+ format (true) or older openssl 0.x format (false)
gateway.truststore.opensslPath	filesystem path	/etc/grid-security/certificates	Directory to be used for opeenssl truststore.
--- Revocation settings ---			

Property name	Type	Default value / mandatory	Description
`gateway.truststore.crlConnectionTimeout`	integer number	`15`	Connection timeout for fetching the remote CRLs in seconds (not used for Openssl truststores).
`gateway.truststore.crlDiskCachePath`	filesystem path	-	Directory where CRLs should be cached, after downloading them from remote source. Can be left undefined if no disk cache should be used. Note that directory should be secured, i.e. normal users should not be allowed to write to it. Not used for Openssl truststores.
`gateway.truststore.crlLocations.*`	list of properties with a common prefix	-	List of CRLs locations. Can contain URLs, local files and wildcard expressions. Not used for Openssl truststores. *(runtime updateable)*
`gateway.truststore.crlMode`	[REQUIRE, IF_VALID, IGNORE]	`IF_VALID`	General CRL handling mode. The IF_VALID setting turns on CRL checking only in case the CRL is present.
`gateway.truststore.crlUpdateInterval`	integer number	`600`	How often CRLs should be updated, in seconds. Set to negative value to disable refreshing at runtime. *(runtime updateable)*
`gateway.truststore.ocspCacheTtl`	integer number	`3600`	For how long the OCSP responses should be locally cached in seconds (this is a maximum value, responses won't be cached after expiration)
`gateway.truststore.ocspDiskCache`	filesystem path	-	If this property is defined then OCSP responses will be cached on disk in the defined folder.

Property name	Type	Default value / mandatory	Description
gateway.truststore.ocspLocalResponders.<NUMBER>	list of properties with a common prefix	-	Optional list of local OCSP responders
gateway.truststore.ocspMode	[REQUIRE, IF_AVAILABLE, IGNORE]	IF_AVAILABLE	General OCSP ckecking mode. REQUIRE should not be used unless it is guaranteed that for all certificates an OCSP responder is defined.
gateway.truststore.ocspTimeout	integer number	10000	Timeout for OCSP connections in miliseconds.
gateway.truststore.revocationOrder	[CRL_OCSP, OCSP_CRL]	OCSP_CRL	Controls overal revocation sources order
gateway.truststore.revocationUseAll	[true, false]	false	Controls whether all defined revocation sources should be always checked, even if the first one already confirmed that a checked certificate is not revoked.

Examples

Note
Various UNICORE modules use different property prefixes. Here we don't put any, but in practice you have to use the prefix (see the reference table above for the actual prefix). Also properties might need to be provided using different syntax, as XML.

Directory trust store, with a minimal set of options:

```
truststore.type=directory
truststore.directoryLocations.1=/trust/dir/*.pem
truststore.crlLocations=/trust/dir/*.crl
```

Directory trust store, with a complete set of options:

```
truststore.type=directory
truststore.allowProxy=DENY
truststore.updateInterval=1234
```

```
truststore.directoryLocations.1=/trust/dir/*.pem
truststore.directoryLocations.2=http://caserver/ca.pem
truststore.directoryEncoding=PEM
truststore.directoryConnectionTimeout=100
truststore.directoryDiskCachePath=/tmp
truststore.crlLocations.1=/trust/dir/*.crl
truststore.crlLocations.2=http://caserver/crl.pem
truststore.crlUpdateInterval=400
truststore.crlMode=REQUIRE
truststore.crlConnectionTimeout=200
truststore.crlDiskCachePath=/tmp
```

Openssl trust store:

```
truststore.type=openssl
truststore.opensslPath=/truststores/openssl
truststore.opensslNsMode=EUGRIDPMA_GLOBUS_REQUIRE
truststore.allowProxy=ALLOW
truststore.updateInterval=1234
truststore.crlMode=IF_VALID
```

Java keystore used as a trust store:

```
truststore.type=keystore
truststore.keystorePath=src/test/resources/certs/truststore. ↩
    jks
truststore.keystoreFormat=JKS
truststore.keystorePassword=xxxxxx
```

4.5.2 Configuring the credential

UNICORE uses private key and a corresponding certificate (called together as a *credential*) to identify users and servers. Credentials might be provided in several formats:

- Credential can be obtained from a *keystore file*, encoded in JKS or PKCS12 format.
- Credential can be loaded as a pair of PEM files (one with private key and another with certificate),
- or from a pair of DER files,
- or even from a single file, with PEM-encoded certificates and private key (in any order).

The following table list all parameters which allows for configuring the credential. Note that nearly all options are optional. If not defined, the format is tried to be guessed. However some credential formats require additional settings. For instance if using *der* format the *keyPath* is

mandatory as you need two DER files: one with certificate and one with the key (and the latter can not be guessed).

Property name	Type	Default value / mandatory	Description
`gateway.credential.path`	filesystem path	*mandatory to be set*	Credential location. In case of *jks*, *pkcs12* and *pem* store it is the only location required. In case when credential is provided in two files, it is the certificate file path.
`gateway.credential.format`	[jks, pkcs12, der, pem]	-	Format of the credential. It is guessed when not given. Note that *pem* might be either a PEM keystore with certificates and keys (in PEM format) or a pair of PEM files (one with certificate and second with private key).
`gateway.credential.password`	string	-	Password required to load the credential.
`gateway.credential.keyPath`	string	-	Location of the private key if stored separately from the main credential (applicable for *pem* and *der* types only),
`gateway.credential.keyPassword`	string	-	Private key password, which might be needed only for *jks* or *pkcs12*, if key is encrypted with different password then the main credential password.
`gateway.credential.keyAlias`	string	-	Keystore alias of the key entry to be used. Can be ignored if the keystore contains only one key entry. Only applicable for *jks* and *pkcs12*.

Examples

Note
Various UNICORE modules use different property prefixes. Here we don't put any, but in practice you have to use the prefix (see the reference table above for the actual prefix). Also properties might need to be provided using different syntax, as XML.

Credential as a pair of DER files:

```
credential.format=der
credential.password=the\!njs
credential.path=/etc/credentials/cert-1.der
credential.keyPath=/etc/credentials/pk-1.der
```

Credential as a JKS file (credential type can be autodetected in almost every case):

```
credential.path=/etc/credentials/server1.jks
credential.password=xxxxxx
```

4.5.3 Proxy certificate support

The UNICORE gateway optionally accepts proxy certificates as used by other Grid middleware systems. In general, we think proxies are a bad idea, but for interoperability purposes, proxies support can be enabled. If enabled, the clients using proxies are authenticated as the initial issuer of the presented proxy certificates chain. See the above reference properties table for the actual setting.

4.6 Logging

UNICORE uses the Log4j logging framework. It is configured using a config file. By default, this file is found in components configuration directory and is named `logging.properties`. The config file is specified with a Java property `log4j.configuration` (which is set in startup script).

Several libraries used by UNICORE also use the Java utils logging facility (the output is two-lines per log entry). For convenience its configuration is also controlled in the same `logging.properties` file and is directed to the same destination as the main Log4j output.

Note
You can change the logging configuration at runtime by editing the logging.properties file. The new configuration will take effect a few seconds after the file has been modified.

By default, log files are written to the the `LOGS` directory.

The following example config file configures logging so that log files are rotated daily.

```
# Set root logger level to INFO and its only appender to A1.
log4j.rootLogger=INFO, A1

# A1 is set to be a rolling file appender with default params
log4j.appender.A1=org.apache.log4j.DailyRollingFileAppender
log4j.appender.A1.File=logs/uas.log

#configure daily rollover: once per day the uas.log will be copied
#to a file named e.g. uas.log.2008-12-24
log4j.appender.A1.DatePattern='.'yyyy-MM-dd

# A1 uses the PatternLayout
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d [%t] %-5p %c{1} %x -  ↩
    %m%n
```

Note
In Log4j, the log rotation frequency is controlled by the DatePattern. Check http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/DailyRollingFileAppender.html for the details.

For more info on controlling the logging we refer to the log4j documentation:

- PatternLayout
- RollingFileAppender
- DailyRollingFileAppender

Log4j supports a very wide range of logging options, such as date based or size based file rollover, logging different things to different files and much more. For full information on Log4j we refer to the publicly available documentation, for example the Log4j manual.

4.6.1 Logger categories, names and levels

Logger names are hierarchical. In UNICORE, prefixes are used (e.g. "unicore.security") to which the Java class name is appended. For example, the XUUDB connector in UNICORE/X logs to the "unicore.security.XUUDBAuthoriser" logger.

Therefore the logging output produced can be controlled in a fine-grained manner. Log levels in Log4j are (in increasing level of severity):

`TRACE` on this level *huge* pieces of unprocessed information are dumped, # `DEBUG` on this level UNICORE logs (hopefully) admin-friendly, verbose information, useful for hunting problems, # `INFO` standard information, not much output, # `WARN` warnings are logged when something went wrong (so it should be investigated), but recovery was possible, # `ERROR` something went wrong and operation probably failed, # `FATAL` something went really wrong - this is used very rarely for critical situations like server failure.

For example, to debug a security problem in the UNICORE security layer, you can set:

```
log4j.logger.unicore.security=DEBUG
```

If you are just interested in details of credentials handling, but not everything related to security you can use the following:

```
log4j.logger.unicore.security=INFO
log4j.logger.unicore.security.CredentialProperties=DEBUG
```

so the XUUDBAuthoriser will log on DEBUG level, while the other security components log on INFO level.

Note
(so the full category is printed) and turn on the general DEBUG logging for a while (on unicore). Then interesting events can be seen and subsequently the logging configuration can be fine tuned to only show them.

The most important, root log categories used by the Gateway's logging are:

unicore.gateway	General gateway logging
unicore.connections	Log IPs of clients, and the DN after the SSL handshake
unicore.httpserver	HTTP processing, Jetty server
unicore.security	Certificate details and other security

The Gateway uses the so called MDC (Mapped Diagnostic Context) to provide additional information on the client which is served. In the MDC the client's IP address and client's Distinguished Name is stored. You can control whether to attach MDC to each line by the `%X{entry}` log4j pattern entry. As the `entry` you can use: `clientIP` or `clientName`. For example:

```
log4j.appender.A1.layout.ConversionPattern=%d [%t] [%X{clientIP} %X ↩
    {clientName}] %-5p %c{1} - %m%n
```

5 AJP Connector for using Apache httpd as a frontend

If you wish to use the Apache webserver (httpd) as a frontent for the gateway (e.g. for security or fault-tolerance reasons), you can enable the AJP connector instead of the usual one.

Requirements

- Apache httpd
- mod_jk for Apache httpd

Enabling AJP13 Gateway with Jetty and mod_jk

Enable "mod_ssl" module in httpd configuration files, for instance "ssl.conf":

```
LoadModule ssl_module modules/mod_ssl.so
Listen 443
```

Enable "mod_jk" module in httpd configuration files, for instance "jk.conf":

```
LoadModule jk_module modules/mod_jk.so
JkWorkersFile "conf.d/worker.properties"
```

Define a "mod_jk" worker, to dialog with Gateway's AJP connector, in "worker.properties" configuration file as referred in the above "jk.conf":

```
worker.list=jetty
worker.jetty.port={{gateway_port}}
worker.jetty.host={{gateway_ip}}
worker.jetty.type=ajp13
worker.jetty.lbfactor=1
```

Configure a httpd virtual SSL host, dedicated to act as Gateway's frontend, in the httpd configuration files, for instance "unicore.conf". This virtual host lets the "mod_jk" worker defined above in "worker.properties", manage all the requests received and forwards the full client's certificate chain:

```
# Apache VirtualHost configuration to serve as frontend
# for an "AJP connector enabled" UNICORE6 Gateway
<VirtualHost {{frontend_ip}}:{{frontend_port}}>
    # Pass every request on this VirtualHost to the jetty worker
    # defined in mod_jk's worker properties file.
    JkMount /* jetty
    # Pass SSL_CLIENT_CERT environment variable to the AJP ↩
        connector
    JkOptions +ForwardSSLCertChain

    # Log
    ErrorLog "logs/unicore_error_log"
```

```
    TransferLog "logs/unicore_access_log"
    JkLogFile "logs/unicore_mod_jk.log"

    # Enable SSL
    SSLEngine on

    # Export SSL-related environment variables, especially ↩
        SSL_CLIENT_CERT
    # which contains client's PEM-encoded certificate
    SSLOptions +StdEnvVars +ExportCertData

    # Client have to present a valid certificate
    SSLVerifyClient require

    # Server certificate
    SSLCertificateFile "{{/path/to/httpd_cert.pem}}"
    SSLCertificateKeyFile "{{/path/to/httpd_key.pem}}"

    # Trusted CAs
    SSLCACertificateFile "{{/path/to/cacert.pem}}"
</VirtualHost>
```

On the Gateway, enable Jetty AJP connector instead of its HTTP connector in Gateway configuration file `CONF/gateway.properties`:

```
#enable AJP connector
gateway.jetty.ajp=true
```

External references

- Configuring AJP13 using mod_jk or mod_proxy_ajp - Jetty http://wiki.eclipse.org/Jetty/-Tutorial/Apache
- The Apache Tomcat Connector - Webserver HowTo http://tomcat.apache.org/connectors-doc/-webserver_howto/printer/apache.html

6 Using the Gateway for failover and/or loadbalancing of UNICORE sites

The Gateway can be used as a simple failover solution and/or loadbalancer to achieve high availability and/or higher scalability of UNICORE/X sites without additional tools.

A site definition (in `CONF/connections.properties`) can be extended, so that multiple physical servers are used for a single virtual site.

An example for such a so-called multi-site declaration in the connections.properties file looks as follows:

```
#declare a multisite with two physical servers

MYSITE=multisite:vsites=https://localhost:7788 https://localhost ↩
    :7789
```

This will tell the gateway that the virtual site "MYSITE" is indeed a multi-site with the two given physical sites.

6.1 Configuration

Configuration options for the multi-site can be passed in two ways. On the one hand they can go into the connections.properties file, by putting them in the multi-site definition, separated by ";" characters:

```
#declare a multisite with parameters

MYSITE=multisite:param1=value1;param2=value2;param3=value3;...
```

The following general parameters exist

vsites	List of physical sites
strategy	Class name of the site selection strategy to use (see below)
config	Name of a file containing additional parameters

Using the "config" option, all the parameters can be placed in a separate file for enhanced readability. For example you could define in connections.properties:

```
#declare a multisite with parameters read from a separate file

MYSITE=multisite:config=conf/mysite-cluster.properties
```

and give the details in the file "conf/mysite-cluster.properties":

```
#example multisite configuration
vsites=https://localhost:7788 https://localhost:7789

#check site health at most every 5 seconds
strategy.healthcheck.interval=5000
```

6.2 Available Strategies

A selection strategy is used to decide where a client request will be routed. By default, the strategy is "Primary with fallback", i.e. the request will go to the first site if it is available,

otherwise it will go to the second site.

Primary with fallback

This strategy is suitable for a high-availability scenario, where a secondary site takes over the work in case the primary one goes down for maintenance or due to a problem. This is the default strategy, so nothing needs to be configured to enable it. If you want to explicitely enable it anyway, set

```
strategy=primaryWithFallback
```

The strategy will select from the first two defined physical sites. The first, primary one will be used if it is available, else the second one. Health check is done on each request, but not more frequently as specified by the "strategy.healthcheck.interval" parameter. By default, this parameter is set to 5000 milliseconds.

Changes to the site health will be logged at "INFO" level, so you can see when the sites go up or down.

Round robin

This strategy is suitable for a load-balancing scenario, where a random site will be chosen from the available ones. To enable it, set

```
strategy=roundRobin
```

Changes to the site health will be logged at "INFO" level, so you can see when the sites go up or down.

It is very important to be aware that this strategy requires that all backend sites used in the pool, share a common persistence. It is because Gateway does not track clients, so particular client requests may land at different sites. This is typically solved by using a non-default, shared database for sites, such as MySQL.

Note
Currently loadbalancing of target sites is an experimental feature and is not yet fully functional. It will be improved in future UNICORE versions.

Custom strategy

You can implement and use your own failover strategy, in this case, use the name of the Java class as strategy name:

```
strategy=your_class_name
```

7 Gateway failover and migration

The Section 6 covered usage of the Gateway to provide failover of backend services. However it may be needed to guarantee high-availabilty for the Gateway itself or to move it to other machine in case of the original one's failure.

7.1 Gateway's migration

The Gateway does not store any state information, therefore its migration is easy. It is enough to install the Gateway at the target machine (or even to simply copy it in the case of installation from the core server bundle) and to make sure that the original Gateway's configuration is preserved.

If the new machine uses a different address, it needs to be reflected in the server's configuration file (the listen address). Also, the configuration of sites behind the gateway must be updated accordingly.

7.2 Failover and loadbalancing of the Gateway

Gateway itself doesn't provide any features related to its own redundancy. However as it is stateless, the standard redundancy solutions can be used.

The simpliest solution is to use Round Robin DNS, where DNS server routes the Gateway's DNS address to a pool of real IP addresses. While easy to set up this solution has a significant drawback: DNS server doesn't care about machines being down.

The much better choice is to use the Linux-HA software suite, often known under the name of its principal component, the *heartbeat*. For details see http://www.linux-ha.org

Additionally a more advanced HTTP-aware software can be used, such as HA-Proxy (http://haproxy.1wt.eu). Currently Gateway and UNICORE don't maintain HTTP sessions so usage of the HTTP-aware load-balancer is not strictly required, but such solutions generally provide more general purpose features.

8 Gateway plugins

Note
This functionality is not supported when the NIO connectors are enabled in Jetty.

The gateway supports tunneling other protocols through its socket. This is still experimental, so use at your own risk. To establish the tunnel, a special HTTP message is sent to the gateway:

```
HEAD / HTTP/1.1
Upgrade: YOURPROTOCOLNAME
```

the method must be "HEAD", and the message must contain the "Upgrade" header. The gateway replies:

```
HTTP/1.1 101 Switching protocols
Upgrade: YOURPROTOCOLNAME
```

After this the gateway's socket connection is passed to a custom handler, which can read data from the client and write replies. The handler can be configured in `CONF/gateway.properties`:

```
protocolPlugin.YOURPROTOCOLNAME=your.handler.classname
```

The handler class must implement the eu.unicore.gateway.util.ProtocolPluginHandler interface. For more details please refer to the sourcecode of the plugin interface class.

9 Building the Gateway from source

To checkout the latest version of the Gateway source code, do

```
svn co http://unicore.svn.sourceforge.net/svnroot/unicore/gateway/ ↩
    trunk gateway
```

You will need to install Maven from http://maven.apache.org Compile using

```
mvn install
```

Compiles the code and runs the tests.

```
mvn assembly:assembly
```

builds distribution archives in zip and tar.gz format in the target/ directory