UNICORE UFTPD SERVER

UNICORE Team

Document Version:	1.0.0
Component Version:	2.6.0
Date:	11 07 2018

Contents

This is the UFTPD user manual providing information on running and using the UNICORE UFTP server *uftpd*. 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

UFTP issue tracker: https://sourceforge.net/p/unicore/uftp-issues

UNICORE UFTP

UFTP is a data streaming library and file transfer tool.

It can be used integrated into UNICORE, allowing to transfer data from client to server (and vice versa), as well as providing data staging between UFTP-enabled UNICORE sites. UFTP can also be used independently from UNICORE, requiring a authentication server and a standalone UFTP client.

A full UFTP server installation consists of two parts

- the "uftpd" file server
- either a UNICORE/X server, or a standalone authentication service.

This manual covers the UFTP file server "uftpd".

UFTP features

- dynamic firewall port opening using a pseudo FTP connection. UFTPD requires only a single open port.
- parallel input/output streams based on code from the JPARSS library, Copyright (c) 2001 Southeastern Universities Research Association, Thomas Jefferson National Accelerator Facility
- optional encryption of the data streams using a symmetric key algorithm
- optional compression of the data streams (using gzip)
- command port protected by SSL
- partial reads/writes to a file. If supported by the filesystem, multiple UFTP processes can thus read/write a file in parallel (striping)
- supports efficient synchronization of single local and remote files using the rsync algorithm
- integrated into UNICORE clients for fast file upload and download

- integrated with UNICORE servers for fast data staging and server-to-server file transfers
- standalone (non-UNICORE) client available
- data upload/download with *curl* or *ftp* possible in conjunction with the auth server
- written in Java

How does UFTP work

The UFTP file server, called *uftpd*, listens on two ports (which may be on two different network interfaces):

- the command port receives control commands
- the listen port accepts data connections from clients.

The uftpd server is "controlled" (usually by UNICORE/X) via the command port, and receives/sends data directly from/to a client machine (which can be an actual user client machine or another server). Data connnections are made to the "listen" port, which has to be accessible from external machines. Firewalls have to treat the "listen" port as an FTP port.

A UFTP file transfer works as follows:

- the UNICORE/X server sends a request to the command port. This request notifies the UFTPD server about the upcoming transfer and contains the following information
  - the source/target file name
  - whether to send or receive data
  - a "secret", i.e. a one-time password which the client will send to authenticate itself
  - how many data connections will be opened
  - the user and group id for who to create the file (in case of send mode)
  - an optional key to encrypt/decrypt the data
  - the client's IP address
- the UFTPD server will now accept an incoming connection from the announced IP address, provided the supplied "secret" matches the expectation.
- if everything is OK, the requested number of data connections from the client can be opened. Firewall transversal will be negotiated using a pseudo FTP protocol.
- the file is sent/received using the requested number of data connections
- to access the requested file, uftpd attempts to switch its user id to the requested one prior to reading/writing the file. This uses a C library which is accessed from Java via the Java native interface (JNI). See also the installation section below.

[NOTE] .IMPORTANT SECURITY NOTE

The UNICORE UFTPD server is running with root privileges. Make sure to read and understand the section below on protecting the command socket. Otherwise, users logged on to the UFTPD machine can possibly read and write other user's files.

Installation and use

Prerequisites

- Java 8 (or later) runtime is required
- the server "listen" port needs to be accessible through your firewalls, declaring it an "FTP" port (FTP connection tracking)
- the UFTPD server needs access to the target file systems
- a server certificate for the UFTPD server is strongly recommended for production use (see the section on SSL below)

A functional UFTP installation requires either a full UNICORE server or the *auth server*.

NOTE ON PATHS

The UNICORE UFTPD server is distributed either as a platform independent and portable tar.gz or zip bundle, or as an installable, platform dependent package such as RPM.
Depending on the installation package, the paths to various files are different. If installing using distribution-specific package the following paths are used:

```
CONF=/etc/unicore/uftpd
SBIN=/usr/sbin
BIN=/usr/bin
LOG=/var/log/unicore/uftpd
LIB=/usr/share/unicore/uftpd/lib
```

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

```
CONF=INST/conf
SBIN=INST/bin
BIN=INST/bin
LOG=INST/log
LIB=INST/lib
```

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

C library for switching user ID

It may be required to re-compile the libuftp-unix.so library on your system. This library uses the Java Native Interface (JNI). The version supplied with the distribution has been compiled on a 64bit Linux system. The folder LIB/native contains the required headers and C source files, as well as an exemplary makefile. Please edit the makefile, the following information is required:

- the base directory of our Java installation (JAVA_HOME)
- the location of platform-specific include files
- the location of the `uftp-<version>.jar` file (LIB)

Then, run "make install" to build the library, which will compile the code and install the library into the LIB folder. If any problems occur during this procedure, please consult UNICORE support.

Starting and stopping the UFTPD server

In the SBIN directory, start/stop and status scripts are provided:

- `unicore-uftpd-start.sh` starts the server
- `unicore-uftpd-stop.sh` stops the server
- `unicore-uftpd-status.sh` checks the server status

The parameters such as server host/port, control host/port, and others are configured in the `CONF/uftpd.conf` file

In a production scenario with multiple users, the uftpd server needs to be started as root. This is necessary to be able to set the correct file permissions.

Configuration parameters

The following variables can be defined in the configuration file (uftpd.conf):

```
SERVER_HOST     : the interface where the server listens for ↩
    client data
                  connections

SERVER_PORT     : the port where the server listens for  ↩
   client data
                  connections
```

```
ADVERTISE_HOST  : Advertise this server as having the ↩
   following IP in the
                 control connection. This is useful if the ↩
                    server is behind
                 a NAT firewall and the public address is ↩
                    different from
                 SERVER_HOST.

CMD_HOST        : the interface where the server listens for ↩
    control commands

CMD_PORT        : the port where the server listens for ↩
   control commands

SSL_CONF        : File containing SSL settings for the ↩
   command port

ACL             : File containing the list of server DNs ↩
   that are allowed
                 access to the command port

UFTPD_MEM       : the maximum memory (Java heap size) ↩
   allocated to the UFTPD server

MAX_CONNECTIONS : the maximum number of concurrent control ↩
   connections per client IP

MAX_STREAMS     : the maximum number of parallel TCP streams ↩
    per connection

BUFFER_SIZE     : the size of the buffer (in kilobytes) for ↩
   reading/writing local files

PORT_RANGE      : (optional) server-side port range in the ↩
   form 'lower:upper' that will be
                 used for data connections. By default, any ↩
                     free ports will be used.
                 Example: set to '50000:50500' to limit the ↩
                     port range.
                 NOTE ports in this range *must not* be ↩
                    used by other services!
```

```
DISABLE_IP_CHECK : (optional) in some situations, the client ↩
    IP can be different from
                the one that was sent to the UFTPD server ↩
                   . This will lead to rejected
                transfers. Setting this variable to a non ↩
                   -zero value will disable the
                IP check. Only the one-time password will ↩
                    be checked.
```

As usual if you set the SERVER_HOST to be "0.0.0.0", the server will bind to all the available network interfaces.

If possible, use an "internal" interface for the Command socket. If that is not possible, make sure the Command socket is protected by a firewall!

We strongly recommend enabling SSL for the Command socket. Please refer to the next section.

Protecting the Command socket

Using SSL for the Command port ensures that only trusted parties (i.e., trusted UNICORE servers) can issue commands to the UFTPD server. To further limit the set of trusted users, an access control list (ACL) file is used.

In production settings where users can log in to the UFTPD server machine, SSL MUST be enabled to prevent unauthorized data access!

[NOTE] .IMPORTANT SECURITY NOTE

Without SSL enabled, users logged in to the UFTPD server can potentially read or write files with arbitrary user privileges (except *root*).

SSL setup

To setup SSL, you need a keystore containing the UFTPD server's credential, and a truststore containing certificate authorities that should be trusted. Keystore and truststore can be the same file.

The following properties can be set in the `CONF/udtpd-ssl.conf` file.

Table 1: Credential properties

Property name	**Type**	**Default value / mandatory**	**Description**
`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.
`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).
`credential. password`	string	-	Password required to load the credential.
`credential. keyPath`	string	-	Location of the private key if stored separately from the main credential (applicable for *pem* and *der* types only),
`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.
`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*.

Table 2: Truststore properties

Property name	**Type**	**Default value / mandatory**	**Description**
`truststore.allowProxy`	[ALLOW, DENY]	`ALLOW`	Controls whether proxy certificates are supported.
`truststore.type`	[keystore, openssl, directory]	*mandatory to be set*	The truststore type.
`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 ---			
`truststore.directoryConnectionTimeout`	integer number	`15`	Connection timeout for fetching the remote CA certificates in seconds.
`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.
`truststore.directoryEncoding`	[PEM, DER]	`PEM`	For directory truststore controls whether certificates are encoded in PEM or DER. Note that the PEM file can contain arbitrary number of concatenated, PEM-encoded certificates.
`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 ---			

Table 2: (continued)

Property name	Type	Default value / mandatory	Description
`truststore.keystoreFormat`	string	-	The keystore type (jks, pkcs12) in case of truststore of keystore type.
`truststore.keystorePassword`	string	-	The password of the keystore type truststore.
`truststore.keystorePath`	string	-	The keystore path in case of truststore of keystore type.
--- Openssl type settings ---			
`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)
`truststore.opensslNsMode`	[GLOBUS_EUGRIDPMA, EUGRIDPMA_GLOBUS, GLOBUS, EUGRIDPMA, GLOBUS_EUGRIDPMA_REQUIRE, EUGRIDPMA_GLOBUS_REQUIRE, GLOBUS_REQUIRE, EUGRIDPMA_REQUIRE, EUGRIDPMA_AND_GLOBUS, EUGRIDPMA_AND_GLOBUS_REQUIRE, IGNORE]	`EUGRIDPMA_GLOBUS`	In case of openssl truststore, controls which (and in which order) namespace checking rules should be applied. The *REQUIRE* settings will cause that all configured namespace definitions files must be present for each trusted CA certificate (otherwise checking will fail). The *AND* settings will cause to check both existing namespace files. Otherwise the first found is checked (in the order defined by the property).
`truststore.opensslPath`	filesystem path	`/etc/grid-security/certificates`	Directory to be used for opeenssl truststore.
--- Revocation settings ---			
`truststore.crlConnectionTimeout`	integer number	`15`	Connection timeout for fetching the remote CRLs in seconds (not used for Openssl truststores).

Table 2: (continued)

Property name	Type	Default value / mandatory	Description
`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.
`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)*
`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.
`truststore.crlUpdateInterval`	integer number	`600`	How often CRLs should be updated, in seconds. Set to negative value to disable refreshing at runtime. *(runtime updateable)*
`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)
`truststore.ocspDiskCache`	filesystem path	-	If this property is defined then OCSP responses will be cached on disk in the defined folder.
`truststore.ocspLocalResponders.<NUMBER>`	list of properties with a common prefix	-	Optional list of local OCSP responders

Table 2: (continued)

Property name	Type	Default value / mandatory	Description
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.
truststore.ocspTimeout	integer number	10000	Timeout for OCSP connections in miliseconds.
truststore.revocationOrder	[CRL_OCSP, OCSP_CRL]	OCSP_CRL	Controls overal revocation sources order
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.

If the credential.path property is NOT set, SSL will be disabled.

[NOTE] .Backwards compatibility to previous versions

Existing configuration files with the javax.net.ssl.* properties used in UFTPD < 2.6 are still supported

ACL setup

The access control list contains the distinguished names of those certificates that should be allowed access.

The "ACL" setting in CONF/uftpd.conf is used to specify the location of the ACL file

```
export ACL=conf/uftpd.acl
```

The default ACL contains the certificate DN of the UNICORE/X server from the UNICORE core server bundle. In production, you need to replace this by the actual DNs of your UNICORE/X server(s) and UFTP Authentication server(s).

The ACL entries are expected in RFC2253 format. To get the name from a certificate in the correct format using openssl, you can use the following OpenSSL command:

```
$> openssl x509 -in your_server.pem -noout -subject -nameopt ↩
    RFC2253
```

The ACL file can be updated at runtime.

Firewall configuration

Note
Please consult the firewall documentation on how to enable an "FTP" service on your firewall (or operating system).

With Linux iptables, you may use rules similar to the following:

```
iptables -A INPUT -p tcp -m tcp --dport $SERVER_PORT -j ACCEPT
iptables -A INPUT -p tcp -m helper --helper ftp-$SERVER_PORT -j  ↩
    ACCEPT
```

where $SERVER_PORT is the SERVER_PORT defined in uftpd.conf. The first rule allows anyone to access port $SERVER_PORT. The second rule activates the iptables connection tracking FTP module on port $SERVER_PORT.

On some operating systems it may be required to load additional kernel modules to enable connection tracking, for example on CentOS:

```
modprobe nf_conntrack_ipv4
modprobe nf_conntrack_ftp ports=$SERVER_PORT
```

Logging

UFTPD uses log4j, the same logging system as other UNICORE components. Logging is configured in the `CONF/logging.properties` file.

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 `LOG` directory.

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.

Logger categories, names and levels

Logger names are hierarchical. Prefixes are used (e.g. "uftp.server") to which the Java class name is appended.

The logging output produced can be controlled in a fine-grained manner. Log levels in Log4j are (in increasing level of severity) TRACE, DEBUG, INFO, WARN, ERROR, amd FATAL.

For example, to track the UFTPD server's communication with clients in detail, you can set

```
log4j.logger.uftp.server=DEBUG
```

Here is a table of logger categories

Log category	Description
uftp	All UFTP logging
uftp.security	Security
uftp.server	Server code
uftp.client	Client code

Note
Please take care to not set the global level to TRACE or DEBUG for long times, as this may produce a lot of output.

Usage logging

Often it is desirable to keep track of the usage of your UFTPD server. The server has a special logger category which logs information about finished jobs at INFO level. If you wish to enable this, set the level to INFO or higher:

```
log4j.logger.uftp.server.USAGE=INFO
```

UNICORE Integration

In UNICORE, UFTP can be used for uploading/downloading data from a server, as well as server-to-server transfers, where one UNICORE/X instance acts as UFTP client.

To enable a UNICORE/X server for the UFTP filetransfer the following settings have to be made.

Configuring the UFTP service

The UNICORE/X server needs to know the UFTPD server settings (i.e. host and port of both the command and the main listener socket). If the `coreServices.uftp.server.host` property is set, UFTP will be enabled and advertised to clients.

In the `uas.config` file, the following properties are mandatory for UFTP:

```
#
# REQUIRED parameters
#

# Listener (pseudo-FTP) socket
coreServices.uftp.server.host=...
coreServices.uftp.server.port=...

# Command socket
coreServices.uftp.command.host=...
coreServices.uftp.command.port=...

#
# Full path to the 'uftp.sh' client executable
# on the TSI login node
#
coreServices.uftp.client.executable=...
```

There are further configuration options to tune how UFTP behaves

```
#
# Optional parameters
#

# How many parallel streams to use per file transfer
coreServices.uftp.streams=2

# Limit the maximum number of streams per file transfer
coreServices.uftp.streamsLimit=4

# File read/write buffer in kbytes
coreServices.uftp.buffersize=128
```

UFTPD servers with multiple interfaces

If your UFTPD server is on a machine with multiple network interfaces, you can give a comma-separated list of host names, like so:

```
coreServices.uftp.server.host=net1.domain.org,net2.domain.org
```

The UFTP client in this case will try to connect to the hosts in the specified order, and will use the first "working" one.

Enabling data encryption

If you wish to encrypt the data sent/received by UFTPD (in data staging or server-to-server transfers), the following property can be used:

```
# enable data encryption
coreServices.uftp.encryption=true
```

This will by default encrypt data with a symmetric key using the Blowfish algorithm. This costs some performance due to the additional CPU load. Encryption only works in single-stream mode. Users can override this setting.

Limiting bandwidth per transfer

It is possible to limit the bandwidth that is consumed by a single UFTP transfer. It is given in bytes per second.

```
# limit transfer rate (bytes/second)
coreServices.uftp.rateLimit=10000000000
```

Disabling SSL on the command port

While not recommended, it may be sometimes useful to disable SSL for communicating with the UFTPD, e.g. while setting up and testing. To do this add a property in `uas.config` (no UNICORE/X restart is required)

```
coreServices.uftp.command.sslDisable=true
```

Enabling "local" UFTP mode on the UNICORE/X server

In case the UNICORE/X server has direct access to the target file system, and you're not using the UNICORE TSI, it can be an interesting option to run the UFTP filetransfer client code directly in the UNICORE/X server instead of passing it to the TSI. This means more load in the UNICORE/X process. It is also a way to use UFTP to/from a UNICORE/X server running on Windows. To enable local mode, edit `uas.config` and set

```
# enable local client mode
coreServices.uftp.client.local=true
```

Testing the UFTPD server without UNICORE

Testing as described in this section works only if SSL is not enabled. Therefore, you should run these tests as a non-root user. Enable SSL and restart the UFTPD server with root privileges once you are finished with these tests.

The UFTPD distribution contains two scripts that allow you to test the UFP functionality without using UNICORE. Making a data transfer involves two steps:

- invoke `uftp-job.sh` to "announce" an upcoming transfer to the UFTPD server
- invoke `uftp.sh` to initiate the actual transfer

Note, in case you installed from an RPM or DEB package, these files are located in `/usr/bin`.

The following shell commands "transfer" the file `.bashrc` to the /tmp directory.

Assuming you installed from RPM/DEB:

```
. /etc/unicore/uftpd/uftpd.conf
unicore-uftpd-job.sh -c localhost -f ~/.bashrc -s true -x my_secret ↩
     -n 2 -u unicore -g unicore
uftp.sh -r -f /tmp/test -L $SERVER_PORT -l $SERVER_HOST -x  ↩
    my_secret -n 2
```

This should create a file `/tmp/test` identical to `~/.bashrc`. Check the console output and the UFTPD log file `LOG/uftpd.log` in case of errors.

After the transfer finished, check that indeed

```
md5sum /tmp/test ~/.bashrc
```

gives the correct checksum for the newly created file.

It is also possible to enable encryption "manually", by appending "-E <key>" to the commands above, where "key" is a sequence of 12 characters (really a base64-encoded 64 bit key).

Performance measurement hints

To run performance tests, it is possible to read and write from/to pseudo files in "/dev". Since UFTP needs to know the number of bytes to transfer, this can be given using a "pseudo" filename. For example, you could read from "/dev/zero_<number_of_bytes>" and write to "/dev/null".

UFTP will treat any filename that starts with "/dev" and contains an underscore "_" character in this manner.

For example, to transfer a gigabyte of zeros to /dev/null,

```
. /etc/unicore/uftpd/uftpd.conf
unicore-uftpd-job.sh -c localhost -f /dev/zero_1000000000 -s true - ↩
    x my_secret -n 2 -u unicore -g unicore
uftp.sh -r -f /dev/null -L $SERVER_PORT -l $SERVER_HOST -x  ↩
    my_secret -n 2
```