TF PROTOCOL

ver 2.4.1

**“To my wife Yanin for her infinite patience
for my time consuming projects.”**

Overview

The TF Protocol works by sending text commands from client to the server in a TCP connection. Every time a command is received at the server it responds with a command execution status. The buffer used for all the communication must be of 512 * 1024 bytes.

The communication between the server and the client occurs in an encrypted way. This encryption is a client-to-server one. It uses at first public key encryption to exchange a session key. This session key is used for the encryption to transmit and receive commands, data, etc.

This protocol has two works modes. The standard mode, and the notification mode. In the standard mode the server holds on for commands from the client, then it executes it, and it returns the result and the possible extra data depending on the command. In the notification mode the client holds on for the server to receive data.

Every sent or received data must be preceded by a header indicating the size of the message that will be received. The only exception of this, are for those modules and commands that implements their own header mechanism. The header for the current version of TF PROTOCOL looks like this:

```
struct tfhdr {
   int32_t sz;
}
```

It only contains one field of 32bits (integer). Before sending it must be converted to network byte order (big-endian). It must be two of

this header to allow multi-threaded communication (full-duplex comm). One used for TX ‘transmit data’ and the other one used for RX ‘receive data’.

Starting communication

The server has the fallowing internal information structure.

```
/* Protocol version length */
#define PROTOLEN 32
/* Hash application identity lentgth. */
#define HASHLEN 256

struct tfproto {
    /* Protocol version */
    char proto[PROTOLEN];
    /* Hash identity */
    char hash[HASHLEN];
    /* Directory for the database. */
    char dbdir[PATH_MAX];
    /* Number of port for comm. */
    char port[PORTLEN];
}
```

The **proto** member holds the protocol version. The **hash** member holds a message digest that identifies the client in order to allow the use of the protocol. The **dbdir** member holds the directory for the data that clients require to store. The **port** member indicates the number of ports in which the server will listen for connections. As stated before the **hash** has to be less than 256 bytes long; and the **proto** less than 32 bytes long.

In order to match this structure when a client starts a connection, there are some steps that must be follow.

Beginning the connection

The client starts the connection by sending the version of the protocol that it expects to use. The exact string of this depends of the configuration of the protocol. There is no restriction to numbers.

Once the server validates the protocol version it responds with a command status or an error condition.

OK for approved.
FAILED 2 : Incompatible protocol.

If the server returns with a failure status it has three parts. The string FAILED which indicates an error. The number after one space character could be used to identify the error with integer type so there is no need to parse the following string. The next string after the number is a text that describes, mostly for debug purpose, the nature of the error.

This dynamic of sending commands from client and once the server processes it, it returns an exit status, it repeats for every command of the protocol. There is one exception, some command may not indicate and error code and the string indicating the error.

After the server validates the protocol version and sends back to the client the OK command, it goes to wait again, it is time for the public key of the client to start the encryption system. If the protocol validation fails the connection is terminated by the server.

The encryption system

The TF PROTOCOL transmits and receives data encrypted. The client must have the public key of the server. The RSA must be 2048 bit and RSA_PKCS1_OAEP_PADDING padding. The procedure of exchanging keys and generating session keys occurs as follows:

1) The client generates an arbitrary session key which max length cannot exceed 2048 / 8 – 42 bytes, and must be at least 16 bytes.
2) The client encrypts the session key with the server's public key and sends it.
3) The server decrypts the session key with its own private key.
4) The server sends back OK if the process has been successful, otherwise sends back FAILED, yet in plain text.

If there is an error in this process the connection is terminated by the server.

The encryption session key is an array of at least 16 bytes up to 2048 / 8 – 42 that is generated in a random way and it will be xored with the data sent or received from the server. As soon as the server and the client are in possession of the session key, it must be extracted from it a **seed**. The seed are the first 8 bytes as a 64bit integer.

The algorithm to encrypt and decrypt data with the session key in C looks like this one.

```
#define KEYLEN 32
char rndkey[KEYLEN] = “the encryption key”;
char data[] = “the data”;
int len = sizeof data;
int decrypt;
```

```
        int c = 0, keyc = 0;
        for (; c < len; c++, keyc++) {
                if (keyc == KEYLEN)
                        keyc = 0;
                if (decrypt) {
                        /* Unpack received data */
                        data[c] = data[c] – (seed >> 56 & 0xFF);
                        /* Decrypt received data */
                        data[c] ^= rndkey[keyc];
                } else if (encrypt) {
                        /* Encrypt data to send */
                        data[c] ^= rndkey[keyc];
                        /* Pack data to send */
                        data[c] = (seed >> 56 & 0xFF) + data[c];
                }
                /* Change seed and encryption key */
                seed = seed * (seed >> 8 & 0xFFFFFFFF) +
                    (seed >> 40 & 0xFFFF);
                if (seed == 0)
                    seed = *(int64_t *) rndkey;
                rndkey[keyc] = seed % 256;
}
```

From this point forward, all the communication will be encrypted and decrypted with the session key through an algorithm like the above. As the session key changes over time, it must be a copy of those two factors: **session key** and **seed**. This allows in a multi-thread environment perform reading and writing operations simultaneously without interference to each other.

It is also required to preserve the session key in its original form, this is, without any transformation. This is a provision in case any extended subsystem requires the original key.

The next step is to send to the server the hash identity. It responds as usual with OK or FAILED exit status.

The summary.

1) The client starts a connection.
2) The client sends the protocol version.
3) The server responds one of these:
    OK
    FAILED 2 : Incompatible protocol.
4) The client sends the session key encrypted with the server's public key.
5) The server decrypts the session key which will be used to encrypt the rest of the communication. The response is one of these:
    OK
    FAILED 25 : Bad public rsa encryption key.

From this point on, encrypted.

6) The client sends the hash.
7) The server validates it and responds one of these:
    OK
    FAILED 3 : Invalid hash string.

Once all the above steps are passed successfully, the communication begins with the command interface. In case of a failure the communication is terminated by the server.

Security enhancement

This security enhancement provides a real user ‘system user’ mechanism to enforce trough the file system driver the directory and file access permission. It works in one of the following ways:

Single-user:
The TFProtocol deployment creates a default user for the protocol daemon. The daemon works on behalf this user and there is no need for further settings, and login it is not required. In this case every connection to the protocol will work with that user permission.

Multi-user:
The TFProtocol does not define any default user; instead, it defines a set of user/password pairs. In this case, once the connection is established, the client should call as a first command ‘LOGIN’ to login at the server. If the user/password combination is a valid one, the server will use that user as the persona of the process. The fundamental difference in regards to the ‘Single-user’ mentioned above is that every connection can login with a different user and thus its respective access permission.
This way every client using the protocol can create directories and files with ownership and security access enforced by the file-system driver. This allows clients, if used properly, to have directories and files that no one else can read/write.

Only the following commands can be called before **LOGIN**:

END, **KEEPALIVE**, and **PROCKEY**

In order to allow this security enhancement to work, the protocol defines a few special commands for this task: **LOGIN**, **CHMOD**, **CHOWN.** You can find more information in the next section.

In-jail:
The TFProtocol deployment sets whether in-jailing will be required or not. The term in-jail refers to restrict the access of the TFProtocol daemon to a particular directory. Is in-jailing is required, it has to be called after a possible login requirement, but before every other command. For further information see the **INJAIL** command documentation.

Only the following commands can be called before **INJAIL** if multi-user is set:

END, **LOGIN, KEEPALIVE**, and **PROCKEY**

If single-user is set, only the **END** command can be called before **INJAIL.**

The command interface

The server can receive the fallowing commands. Every command is formed by a name and optional arguments. The double quote of the command arguments in the examples, as in the return status examples, are not part of the command syntax, it is just for reading convenience.

UNKNOWN

This is the response sent by the server if the command is not recognized.

END

END command is sent by the client to the server in order to terminate the TCP connection.

ECHO “example”

ECHO command is for debug propose. Once sent the server replays back the exact same thing the client sent, including the ECHO word.

MKDIR “path/to/new/dir”

MKDIR command creates a directory at the specified path. The return status of this command could be:

OK
FAILED 4 : Directory already exist.
FAILED 1 : Access denied to location.
FAILED 30 : Directory is locked.

If the parameter is left empty the daemon protocol directory will be used. So it responds the FAILED 4 return status.

Secured Directory

This feature allows the creation of a kind of directory that cannot be listed with any command. In order to access it, the name must be known in advance. The directory name may contain any character, except the decimal zero “0” and the path separator “/”, and must end with an “.sd” extension. The length of the directory name must not exceed 255 characters including the extension.
In order to create strong secured directories, the following technique could be used: Combine a user´s name with a very strong password and apply to them the SHA256 or any other secure hash function. The resulting hash must be scanned to remove or change from it any occurrence of the decimal zero ‘0’ -the null character- or the path separator ‘/’. If the hash is converted to hexadecimal format -which is usual- , it is guaranteed that the resulting string will contain only numbers and letters. If more security is needed, this process can be repeated recursively.

DEL “/path/to/file/to/del”

DEL command deletes the specified file. This is a special command because is the only one who can operate in a locked directory. If it is the case, then DEL can only delete exactly one file: The locking file. The return status of this command could be:

OK
FAILED 1 : Access denied to location.
FAILED 9 : File does not exist.
FAILED 8 : Requested file is a directory.
FAILED 30 : Directory is locked.

FAILED

If used without an argument the protocol daemon directory is used. So it return the FAILED 8 return status.

RMDIR “/path/to/dir/to/remove”

RMDIR removes a specified directory recursively. The return status of this command could be:

OK
FAILED 1 : Access denied to location.
FAILED 30 : Directory is locked.
FAILED 15 : Directory to remove still exist.

If “/path/to/dir/to/remove” has secure directories inside of it, they will be not removed.
If used without an argument the protocol daemon directory is used. So it returns the FAILED 15 return status due the impossibility to delete the daemon directory. However, every other directory will be deleted.

COPY “/path/to/source/file” | “/path/to/new/file”

COPY copy the file indicated by the first parameter of the command to the file indicated by the second one. The return status of this command could be:

OK
FAILED 16 : Missing parameter from command.
FAILED 1 : Access denied to location.
FAILED 19 : Directory can't be linked.

FAILED 12 : File already exist.
FAILED 17 : Source file does not exist.
FAILED 18 : Copy general error.
FAILED 30 : Directory is locked.

TOUCH “path/to/file/to/create”

TOUCH creates a new file in the specified directory. The return status of this command could be:

OK
FAILED 1 : Access denied to location.
FAILED 12 : File already exist.
FAILED 24 : Error creating new file.
FAILED 30 : Directory is locked.

If used without an argument the protocol daemon directory is used. So it return the FAILED 12 return status.

DATE

DATE returns the number of elapsed seconds since the epoch, and arbitrary point in the time continuum, which is the Gregorian calendar time Jan 1 1970 00:00 UTC. The return status of this command could be:

OK “timestamp”

UDATE

UDATE returns the number of elapsed seconds and microseconds since the epoch -separated by dot-, and arbitrary point in the time

continuum, which is the Gregorian calendar time Jan 1 1970 00:00 UTC. The return status of this command could be:

OK “seconds.microseconds”

NDATE

NDATE returns the number of elapsed seconds and nanoseconds since the epoch -separated by dot-, and arbitrary point in the time continuum, which is the Gregorian calendar time Jan 1 1970 00:00 UTC. The return status of this command could be:

OK “seconds.nanoseconds”

DATEF

DATEF returns the current date of the server in human readable format “yyyy-mm-dd HH:MM:SS” UTC. The return status of this command could be:

OK “yyyy-mm-dd HH:MM:SS”

DTOF “timestamp”

DTOF converts date in Unix timestamp format -seconds since the epoch- in human readable format “yyyy-mm-dd HH:MM:SS” UTC. The return status of this command could be:

OK “yyyy-mm-dd HH:MM:SS”
FAILED 16 : Missing parameter from command.
FAILED 26 : Date is not representable.

FTOD “ yyyy-mm-dd HH:MM:SS”

FTOD converts date in human readable format “ yyyy-mm-dd HH:MM:SS” to its Unix timestamp format. The return status of this command could be:

OK “timestamp”
FAILED 16 : Missing parameter from command.

FSTAT “/path/to/file/or/directory”

FSTAT returns statistics of a file or directory in the form "D | F | U FILE-SIZE LAST-ACCESS LAST-MODIFICATION" where D stands for directory; F stands for file; and U stands for unknown, only one of them is reported. The “FILE-SIZE” is reported in bytes in a integer of 64 bits and the number could be up to 20 digits long. The “LAST-ACCESS” and “LAST-MODIFICATION” are both timestamps. The return status of this command could be:

OK “D | F | U FILE-SIZE LAST-ACCESS LAST-MODIFICATION”
FAILED 16 : Missing parameter from command.
FAILED 1 : Access denied to location.
FAILED 9 : File does not exist.
FAILED 30 : Directory is locked.

FUPD “/path/to/file/or/directory”

FUPD update the timestamps of the file or directory to server current time. The return status could be:

OK

FAILED 16 : Missing parameter from command.
FAILED 1 : Access denied to location.
FAILED 9 : File does not exist.
FAILED 27 : Timestamp updating failed.
FAILED 30 : Directory is locked.

CPDIR “/path/to/source/directory” | “/path/to/destination/directory”

CPDIR copy recursively the source directory into a new created directory specified in the second parameter of the command. The return status of this command could be:

OK
FAILED 16 : Missing parameter from command.
FAILED 1 : Access denied to location.
FAILED 20 : Source path is not a directory.
FAILED 4 : Directory already exist.
FAILED 21 : Error replicating directory tree.
FAILED 30 : Directory is locked.

XCOPY “newname” “/path/to/source/file” | “pattern”

XCOPY copies the source file specified in the second parameter into every directory in the tree that matches “pattern” with the name “newname” specified in the first parameter.

This is a best-effort command instead of super-reliable one. If any one of the destinations are locked or the file exists instead of return a failed status code, XCOPY skips it silently. The return status of this command are relative to the source file/directory.
The return status of this command could be:

OK
FAILED 16 : Missing parameter from command.
FAILED 1 : Access denied to location.
FAILED 17 : Source file does not exist.
FAILED 23 : Source path is not a file.
FAILED 30 : Directory is locked.

XDEL “/path/to/directory/” “filename”

XDEL deletes all files that match “filename” in the specified path at first parameter and it does recursively starting at the specified directory.
This is a best-effort command instead of super-reliable one. If any one of the path are locked or even and error occurs, instead of return a failure return status, XDEL skips it silently. The return status of this command are relative to the path specified.
The return status of this command could be:

OK
FAILED 16 : Missing parameter from command.
FAILED 1 : Access denied to location.
FAILED 20 : Source path is not a directory.
FAILED 30 : Directory is locked.

XRMDIR “/path/to/directory/” “directory-name”

XRMDIR deletes all directories recursively that match “directory-name” starting at the specified path at first parameter.
This is a best-effort command instead of super-reliable one. If any one of the path are locked or even an error occurs, instead of return a failure return status, XRMDIR skips it silently. The return status of this command are relative to the path specified.
The return status of this command could be:

OK
FAILED 16 : Missing parameter from command.
FAILED 1 : Access denied to location.
FAILED 10 : Directory does not exist.
FAILED 20 : Source path is not a directory.
FAILED 30 : Directory is locked.

XCPDIR “new-directory-name” “/path/to/source/directory/” | “destination-directory-pattern”

XCPDIR copy recursively source directory specified in the second parameter to every directory in the tree that matches “destination-directory-pattern” with the name “new-directory-name” specified at first parameter.
This is a best-effort command instead of super-reliable one. If any one of the destinations are locked or the directory or file exists, instead of return a failure return code, XCPDIR skips it silently. The return status of this command are relative to the source file/directory
The return status of this command could be:

OK
FAILED 16 : Missing parameter from command.
FAILED 1 : Access denied to location.
FAILED 20 : Source path is not a directory.
FAILED 29 : Source path does not exist.
FAILED 30 : Directory is locked.

LOCK “lock-filename”

LOCK specifies a filename that when it exists in a directory no operation can be done on it, except to delete the lock file. This allows a temporary blocking of a directory for any sort of operations

except the one eliminating the locking file which is the command **DEL** with the path to the locking file. The above statement does not apply for commands that do not specify “FAILED 30” as possible return status. The return status of this command could be:

OK

Specify LOCK with an empty string “” or without argument is the way to turn off the locking mechanism regardless of how many locking files are installed in the directories. The notification system is not affected by the locking mechanism.

This is not a synchronization mechanism. A synchronization mechanism eliminates race conditions, LOCK does not. Don’t try to use it as such, there is a lot of windows in between. Even it is possible that some operations are completed successfully in the directory by other clients after it has been locked. The only way the client realizes of this is by comparing the files and directories timestamps against the lock file timestamp. If the lock file and any other file or directory in the locked directory has the same timestamp the client can decide whether to accept that file or directory as valid one.

SNDFILE “0 | 1” “/path/to/filename”

SNDFILE sends a file to the server. The first parameter could be either “0” for false or “1” for true. If “/path/to/filename” it exists in the server, this flag indicates whether should be overwritten or not. Once the transmission loop starts there are three keywords to continue or stop the flow: **CONT BREAK** and **OK.** The CONT keyword followed by a whitespace “ ” and the payload tells to the server that the chunk of data is part of file. The BREAK keyword

tells to the server to abort the operation and it deletes the already stored data. The OK keyword tells to the server that there is no more data to be send. Every time the server receives a chunk of data it responds with the flow control keyword CONT or and status returns in case of a failure. When the SNDFILE command is sent the server responds with CONT flow control to signal that is ready for receiving the file.
The return status of this command could be:

CONT “To indicate the willingness to continue receiving the file”

OK “In response to the OK sent by the client signaling the EOF -end of file- condition or the BREAK to tell the server to stop receiving”

FAILED 13 : Quota exceeded.
FAILED 16 : Missing parameter from command.
FAILED 1 : Access denied to location.
FAILED 12 : File already exist.
FAILED 8 : Requested file is a directory.
FAILED 7 : File transfer API error.
FAILED 30 : Directory is locked.

As an illustration of the dynamic of this command we can use this example:

CLIENT SEND: SNDFILE 0 /path/to/new/file

SERVER RESPONDS: CONT
CLIENT SEND: CONT PAYLOAD-OF-DATA

“This process repeats until the server returns and failure status or the client sends BREAK to stop the flow or OK to indicate that the

operation is done. In this example the first parameter which is 0 tells to the server that if the /path/to/new/file exists must not be overwritten". The maximum number of bytes for payload is BUFSZ - CONT word and the whitespace. If buffer size is 512, then the payload cannot exceed 507 bytes. As benchmarks have been made we suggest that this command should be used only to transfer small files '100 MB or less'. In a1Gbps network testing the operation is completed in eleven seconds for a 100 MB file.

RCVFILE "0 | 1" "/path/to/filename"

RCVFILE receives a file from the server. The first parameter could be either "0" for false or "1" for true and tells the server whether the file must be deleted after successfully received by the client. Once the transmission loop starts there are three keywords to continue or stop the flow: **CONT BREAK** and **OK.** The CONT keyword followed by a whitespace " " and the payload tells to the client that the chunk of data is part of file. The BREAK keyword tells to the server to abort the operation. The OK keyword tells to the client that there is no more data to receive. Every time the client receives a chunk of data it responds with the flow control keyword CONT or BREAK to stop the operation. When the RCVFILE command is sent the server responds with CONT flow control and the payload or a return status for failure.
The return status of this command could be:

CONT PAYLOAD-OF-DATA

OK "In response to the BREAK sent by the client or to signal the EOF -end of file- condition indicating that the operation is done."

FAILED 16 : Missing parameter from command.

FAILED 1 : Access denied to location.
FAILED 9 : File does not exist.
FAILED 8 : Requested file is a directory.
FAILED 7 : File transfer API error.
FAILED 30 : Directory is locked.

As an illustration of the dynamic of this command we can use this example:

CLIENT SEND: RCVFILE 0 /path/to/filename

SERVER RESPONDS: CONT PAYLOAD-OF-DATA
CLIENT SEND: CONT

“This process repeats until the server returns OK for -end of file-condition or a failure status, or the client sends BREAK to stop the flow. In this example the first parameter which is 0 tells to the server that /path//to/filename must not be deleted after the operation is done”. The maximum number of bytes for payload is BUFSZ - CONT word and the whitespace. If buffer size is 512, then the payload cannot exceed 507 bytes. As benchmarks have been made we suggest that this command be used only to transfer small files ‘100 MB or less’. In a1Gbps network testing the operation is completed in eleven seconds for a 100 MB file.

LS “/path/to/directory/to/list”

LS command list the directory entries for the indicated path, if the argument is missing, it lists the root directory of the protocol daemon. The return value of this command is a file with the listed content. In fact it is like issuing the command RCVFILE to a temporary file with the listed content of the directory.

The file returned by LS has the following syntax.

F | D | U /path/to/file-or-directory

The F stands for “file”; the D for “directory” and the U for “unknown”. The return status of this command beside those for RCVFILE because RCVFILE works on behalf LS could be:

FAILED 1 : Access denied to location.
FAILED 30 : Directory is locked.
FAILED 10 : Directory does not exist.
FAILED 11 : Failed to make a temporary file.

LSR “/path/to/directory/to/list”

This command is exactly like LS but it lists recursively the specified directory.

RENAM “path/to/oldname” | “path/to/newname”

RENAM renames the file or directory specified at first parameter into the name specified at second parameter. RENAM operates atomically; there is no instant at which “newname” is non-existent between the operation’s steps if “newname” already exists. If a system crash occurs, it is possible for both names “oldname” and “newname” to still exist, but “newname” will be intact.

RENAM has some restrictions to operate.
1) “oldname” it must exist.
2) If “newname” is a directory must be empty.
3) If “oldname” is a directory then “newname” must not exist or it has to be an empty directory.

4) The “newname” must not specify a subdirectory of the directory “oldname” which is being renamed.

KEEPALIVE “1|0” “idle-time” | “interval” | “count”

KEEPALIVE sets the configuration parameters for the TCP keepalive feature. This is especially useful for clients behind NAT boxes. If there is some idle time in the established connection -no data transmission- the NAT box could close or unset the connection without the peers knowing it. In contexts where it is predictable that an established connection could be ‘in silent’ for long periods of time, and it is possible that clients are behind NAT boxes, it is necessary to set the TCP keepalive packets.

The first parameter of the command could be 0 or 1, meaning on or off.
The second parameter is the time (in seconds) the connection needs to remain idle before TCP starts sending keepalive probes.
The third parameter is the time (in seconds) between individual keepalive probes.
The fourth parameter is the maximum number of keepalive probes TCP should send before dropping the connection.

The return status of this command could be:

OK
FAILED 32: Failed to set TCP keepalive parameters.

Whether the server implements a default timeout of keepalive to declare a connection dead, it is up to the implementation. In order to be sure that keepalive feature is turned off or on, it must be done explicitly.

There are two more commands ADDNTFY and STARTNTFY. We will see them in the next section.

PROCKEY

PROCKEY retrieves a unique key generated by the server's instance that communicates with the client. This unique key could be used later to identify that instance. One of these uses, but not the only one, is to test whether the server or even the socket communication line is still opened, in other words: the keepalive mechanism from the client side perspective.

The return status of this command could be:

OK
FAILED 33 : Process without unique key.

FREESP

FREESP retrieves the available space in the partition where the protocol folder is located. This command should be used only after a failure to find out if such a failure is due to the lack of space. It should not be used to guess how much space remains, and then proceed with a writing operation. The described scenario could potentially lead to a race condition. If both clients at the same time retrieve the free space and then write according to that value, one of them will fail.

The return status of this command could be:

OK "free-space"

“free-space” will be a 64bit signed integer indicating how much space left -in bytes- in the protocol partition directory at the time of the command execution.

PUT

```
struct hpfile {
    uint64_t offst;
    int64_t bufsz;
} ;
```

PUT “/path/to/filename” hpfile

PUT is part of the High Performance File Interface. It can be used to transfer files from the client to the server.

The first parameter is the file to be written, it could exist or not. If the file does not already exist, then it is created. If the file already exists, the command will overwrite it.

The second parameter goes after the “/path/to/filename” and a whitespace. It consists of two 64bit integers normalized ‘converted to network byte-order’. The first one should be unsigned and the second one, signed. The first one must range from 0 to 2^64-1. The second one must range from -(2^63) to 2^63-1.

The “offst” integer -the first one- is the desire offset in which the data should be written in the opened file, being a new file or an existing one.

The “bufsz” integer -the second one- is the proposed buffer size by the client.

Once the command is constructed like above, it can be sent to the server. On the server side, the protocol will evaluate the parameters and will return one of the following conditions:

OK “definitive-buffer”

FAILED 1 : Access denied to location.
FAILED 16 : Missing parameter from command.
FAILED 34 : H-P interface failed to open file descriptor.
FAILED 35 : H-P interface failed to set file position.
FAILED 36 : H-P interface failed to allocate the buffer.

“definitive-buffer” is a 64bit signed integer normalized -converted to network byte-order- with the buffer that the server finally decided to use. The server tries to create a buffer with the size indicated by the client. If that request cannot be satisfied, the server will create a buffer according to implementation-dependent criteria. In any case, if the server replies OK, the 64bit integer after the whitespace will be the buffer that should be used. The server will never return a buffer’s size greater than the one requested by the client.

After this, the server starts accepting the data flowing from the client. It works as follow:

The client must send a header before every payload. This header contains how many bytes will be sent next. The header is a 64bit signed integer normalized -converted to network byte-order.

This header is a window that allows the client to make some operations during the flow. The client can send the following codes using the header:

0 This is the HPFEND code which means that there is no more data to be sent.

-1 This is the HPFSTOP code which means that the client wants to stop sending data and will continue later. In this case the server just stops writing the file without removing it.

-2 This is the HPFCANCEL code which means that the client is canceling the operation. The server stops the operation and deletes the file.

-127 This is the HPFFIN code and is used to tell the server that the operation has finished.

No matter which code is used, the operation always must be ended by the HPFFIN code. Both sides must send it. First the server, then the client.

After the server receives one of the above codes -except HPFFIN-, it sends HPFFIN code, then the client responds with another HPFFIN code. Once this termination handshake is done, the server returns to listen again for new commands.

About cancellation:

This command is an asynchronous one in the sense that either side can send an HPFCANCEL code in the middle of the flow. As a result, the server can send, while receiving data, any code to request

cooperation from the client. The server can send the following codes:

HPFCANCEL

Under certain conditions the server can decide to cancel the operation and will request the client to do so. After the server sends the HPFCANCEL code, all data received in the server will be discarded. The file will be removed too.

Once the HPFCANCEL code reaches the client, it must stop sending data. After the server sends this code, as stated before, it will send the HPFFIN code which indicates that the operation is done. Then the client responds with another HPFFIN code to confirm the end of the operation.

If the client sends, at any point, HPFSTOP, it can be safely assumed that the amount of data sent to the server is already stored in the file.

The above assumption does not apply when after the client sends HPFSTOP it receives from the server an HPFCANCEL. In this case the server deletes the file.

Note of caution:
As the server can cancel the operation asynchronously, the client may send some data before realizing the cancellation.

GET

```
struct hpfile {
    uint64_t offst;
    int64_t bufsz;
```

} ;

GET “/path/to/filename” hpfile

GET is part of the High Performance File Interface. It can be used to transfer files from the server to the client.

The first parameter is the file to be read.

The second parameter goes after the “/path/to/filename” and a whitespace. It consists of two 64bit integers normalized ‘converted to network byte-order’. The first one should be unsigned and the second one, signed. The first one must range from 0 to 2^64-1. The second one must range from -(2^63) to 2^63-1.

The “offst” integer -the first one- is the desire offset in which the data should start to be read in the opened file.

The “bufsz” integer -the second one- is the proposed buffer size by the client.

Once the command is constructed like the above, it can be sent to the server. On the server side, the protocol will evaluate the parameters and will return one of the following conditions:

OK “definitive-buffer”

FAILED 1 : Access denied to location.
FAILED 16 : Missing parameter from command.
FAILED 34 : H-P interface failed to open file descriptor.
FAILED 35 : H-P interface failed to set file position.
FAILED 36 : H-P interface failed to allocate the buffer.

“definitive-buffer” is a 64bit signed integer normalized -converted to network byte-order- with the buffer that the server finally decided to use. The server tries to create a buffer with the size indicated by the client, if that request cannot be satisfied, the server will create a buffer according to implementation-dependent criteria. In any case, if the server replies OK, the 64bit integer after the whitespace will be the buffer that should be used. The server will never return a buffer’s size greater than the one requested by the client.

After this, the server starts sending data to the client. It works as follows:

The server sends a header before every payload. This header contains how many bytes will be sent next. The header is a 64bit signed integer normalized -converted to network byte-order.

This header is a window that allows the server to do some operations during the flow. The server can send the following codes using the header:

0 This is the HPFEND code which means that there is no more data to be sent.

-2 This is the HPFCANCEL code which means that the server is canceling the operation.

-127 This is the HPFFIN code and is used to tell the client that the operation has finished.

No matter which code is used, the operation must always end with the HPFFIN code. Both sides must send it. Unlike the PUT command, first the client, then the server.

After the server sends one of the above codes -except HPFFIN-, it waits for HPFFIN code, then it responds with another HPFFIN code. Once this termination handshake is done, the server returns to listen again for new commands.

About cancellation:

This command is an asynchronous one in the sense that either side can send a HPFCANCEL code in the middle of the flow. As a result, the client can send, while receiving data, any code to request cooperation from the server. The client can send the following codes:

HPFCANCEL
HPFFIN

Under certain conditions the client can decide to cancel the operation and will request the server to do so. After the client sends the HPFCANCEL code, all data received by the client should be discarded.

Once the HPFCANCEL code reaches the server, it will stop sending data. After the server receives this code, as stated before, it will wait for the HPFFIN code which indicates that the operation is done. Then the server responds with another HPFFIN code to confirm the end of the operation.

Note of caution:

As the client can cancel the operation asynchronously, the server may send some data before realizing the cancellation.

PUTCAN

```
struct hpfile {
    uint64_t offst;
    int64_t bufsz;
    uint64_t canpt;
} ;
```

PUTCAN “/path/to/filename” hpfile

PUTCAN is part of the High Performance File Interface. It can be used to transfer files from the client to the server. It differs from PUT because the cancellation, rather than asynchronous, are predefined points.

The first parameter is the file to be written, it could exist or not. If the file does not already exist, then it is created. If the file already exists, the command will overwrite it.

The second parameter goes after the “/path/to/filename” and a whitespace. It consists of three 64bit integers normalized ‘converted to network byte-order’. The first one should be unsigned; the second one, signed; and the third one, unsigned. The first one must range from 0 to 2^64-1. The second one must range from -(2^63) to 2^63-1. The third one must range from 0 to 2^64-1.

The “offst” integer -the first one- is the desire offset in which the data should be written in the opened file, being a new file or an existing one.

The “bufsz” integer -the second one- is the proposed buffer size by the client.

The “canpt” -the third one- is the number of transmitted buffers in which there will be a window to cancel the operation. 0 means no cancellation window.

Once the command is constructed like above, it can be sent to the server. On the server side, the protocol will evaluate the parameters and will return one of the following conditions:

OK “definitive-buffer”

FAILED 1 : Access denied to location.
FAILED 16 : Missing parameter from command.
FAILED 34 : H-P interface failed to open file descriptor.
FAILED 35 : H-P interface failed to set file position.
FAILED 36 : H-P interface failed to allocate the buffer.

“definitive-buffer” is a 64bit signed integer normalized -converted to network byte-order- with the buffer that the server finally decided to use. The server tries to create a buffer with the size indicated by the client. If that request cannot be satisfied, the server will create a buffer according to implementation-dependent criteria. In any case, if the server replies OK, the 64bit integer after the whitespace will be the buffer that should be used. The server will never return a buffer’s size greater than the one requested by the client.

After this, the server starts accepting the data flowing from the client. It works as follow:

The client must send a header before every payload. This header contains how many bytes will be sent next. The header is a 64bit signed integer normalized -converted to network byte-order.

This header is a window that allows the client to make some operations during the flow. The client can send the following codes using the header:

0 This is the HPFEND code which means that there is no more data to be sent.

-1 This is the HPFSTOP code which means that the client wants to stop sending data and will continue later. In this case the server just stops writing the file without removing it.

-2 This is the HPFCANCEL code which means that the client is canceling the operation. The server stops the operation and deletes the file.

About cancellation:

This command is intended for scenarios where multi-threading is not available, so asynchronous cancellation is not possible. In such a case, the client and the server must agree after how many bytes sent, there will be a chance to cancel the operation.

The “canpt” parameter specifies the number of buffers that will be sent before switching the way the data is flowing. Once the flow is inverted, the server has the chance to send one of the following codes:

HPFCANCEL

-3 HPFCONT (This code tells the other end that flow can continue).

GETCAN

```
struct hpfile {
    uint64_t offst;
    int64_t bufsz;
    uint64_t canpt;
} ;
```

GETCAN “/path/to/filename” hpfile

GETCAN is part of the High Performance File Interface. It can be used to transfer files from the server to the client. It differs from PUT because the cancellation, rather than asynchronous, are predefined points.

The first parameter is the file to be read.

The second parameter goes after the “/path/to/filename” and a whitespace. It consists of three 64bit integers normalized ‘converted to network byte-order’. The first one should be unsigned; the second one, signed; and the third one, unsigned. The first one must range from 0 to 2^64-1. The second one must range from -(2^63) to 2^63-1. The third one must range from 0 to 2^64-1.

The “offst” integer -the first one- is the desire offset in which the data should start to be read in the opened file.

The “bufsz” integer -the second one- is the proposed buffer size by the client.

The “canpt” -the third one- is the number of transmitted buffers in which there will be a window to cancel the operation. 0 means no cancellation window.

Once the command is constructed like the above, it can be sent to the server. On the server side, the protocol will evaluate the parameters and will return one of the following conditions:

OK “definitive-buffer”

FAILED 1 : Access denied to location.
FAILED 16 : Missing parameter from command.
FAILED 34 : H-P interface failed to open file descriptor.
FAILED 35 : H-P interface failed to set file position.
FAILED 36 : H-P interface failed to allocate the buffer.

“definitive-buffer” is a 64bit signed integer normalized -converted to network byte-order- with the buffer that the server finally decided to use. The server tries to create a buffer with the size indicated by the client, if that request cannot be satisfied, the server will create a buffer according to implementation-dependent criteria. In any case, if the server replies OK, the 64bit integer after the whitespace will be the buffer that should be used. The server will never return a buffer’s size greater than the one requested by the client.

After this, the server starts sending data to the client. It works as follows:

The server sends a header before every payload. This header contains how many bytes will be sent next. The header is a 64bit signed integer normalized -converted to network byte-order.

This header is a window that allows the server to do some operations during the flow. The server can send the following codes using the header:

0 This is the HPFEND code which means that there is no more data to be sent.

-2 This is the HPFCANCEL code which means that the server is canceling the operation.

About cancellation:

This command is intended for scenarios where multi-threading is not available, so asynchronous cancellation is not possible. In such a case, the client and the server must agree after how many bytes sent, there will be a chance to cancel the operation.

The “canpt” parameter specifies the number of buffers that will be sent before switching the way the data is flowing. Once the flow is inverted, the client has the chance to send one of the following codes:

HPFCANCEL
-3 HPFCONT (This code tells the other end that flow can continue).

LOGIN ‘user’ ‘password’

LOGIN evaluates the combination of ‘user’ -as first parameter- and ‘password’ -as second parameter- against a database in the server. Note that there is a whitespace between the first and the second parameter. If a match is found, then the protocol sets the real-user id and real-group id of the process to that user. From this point on, all

executed commands at the server are done on behalf of that logged user and the access to the file system will be restricted to the permission of that user. The return status of this command could be:

OK
FAILED 39 : The process is already logged.
FAILED 16 : Missing parameter from command.
FAILED 40 : Failed to open users database.
FAILED 37 : Login failed.

CHMOD “/path/to/file” “octal-mode”

CHMOD changes the file access permission in the same fashion that the UNIX system call does it. The first parameter is the file or directory in which the access bits are about to be changed. The second parameter is the mode ‘permission’ in octal notation to be used. Between the first and the second parameter there is a whitespace.

The complete explanation of composing the mode is out of the scope of this document. But here are a few useful examples:

CHMOD “path-to-file” 0400
Only owner can read the file.
CHMOD “path-to-file” 0200
Only owner can write the file.
CHMOD “path-to-file” 0100
Only owner can execute the file.

Now if we do summation we can obtain the following combinations:

CHMOD “path-to-file” 0600

Only owner can read and write the file.
CHMOD “path-to-file” 0500
Only owner can read and execute the file.
CHMOD “path-to-file” 0300
Only owner can write and execute the file.

So far, we have seen the second digit -from left to right- which represents the owner´s permission. The third digit represents the group´s permission. The fourth digit represents the permission for ‘others’ (anybody else). The same summation logic explained before for the second digit applies to the third and the fourth.

About the first digit, it could be:

4 (SUID) Set user-id bit.
2 (SGID) Set group-id bit.
1 Sticky bit.
Here the summation only applies to SGID (2) and Sticky bit (1). So, the value for the first digit could be: 4, 2, 1, or 3.

For the CHMOD command the first digit -if not needed- could be 0 or it can be completely omitted. For more information see the UNIX man pages.

The return status of this command could be:

OK
FAILED 16 : Missing parameter from command.
FAILED 1 : Access denied to location.
FAILED 9 : File does not exist.
FAILED 41 : Failed to change file mode.

CHOWN “/path/to/file” “user” “group”

CHOWN changes the ownership of a file or a directory. This command in some systems could be only executed by a privileged user (root). In other systems it can be executed too by the owner of the file or directory. The successful execution will depend of the logged user.

The return status of this command could be:

OK
FAILED 16 : Missing parameter from command.
FAILED 1 : Access denied to location.
FAILED 42 : System user does not exist.
FAILED 43 : System group does not exist.
FAILED 44 : Failed to change file ownership.

SHA256 ‘path’

SHA256 makes a sha256 hash of a file indicated by ‘path’. This command is intended to apply a hash to a file before downloading it in order to guarantee its integrity. Be aware that on very large files the time to resolve the digest function could be potentially long. The return status of this command could be:

OK “hash-in-hex-form like 0xff33...”
FAILED 45 : Failed to make SHA256 hash.
FAILED 16 : Missing parameter from command.
FAILED 1 : Access denied to location.

NIGMA ‘keylen’

NIGMA generate a random session key of the specified length in the first parameter. This length should be a number equal or greater than 8 and multiple of 4. If the command fails it return FAILED, otherwise OK. In case of success, after the OK, the server will send a 32bit integer converted to network byte-order indicating the payload size that coming next. This payload will be the new session key and the header must be equal to the keylen indicated by the first parameter. After the client receives the key it should be swapped with the old key, and the new key is the one should be used hereinafter to encrypt/decrypt the data. The return status of this command could be:

OK
-After OK the client must read a 32bit integer header and the session key that coming next, then should be swapped with the old session key-
FAILED 46 : Failed to swap the session key.

RMSECDIR “secure_token” | “/path/to/secure/dir/to/remove”

RMSECDIR removes a specified secure directory recursively. The first parameter is the secure token that allows to remove the directory specified in the second parameter. The secure token could be either a file or a directory named as the first parameter, inside the the directory specified as the second parameter. The return status of this command could be:

OK
FAILED 1 : Access denied to location.
FAILED 30 : Directory is locked.
FAILED 15 : Directory to remove still exist.
FAILED 47 : Failed removing secure directory.

If the second parameter is leaved in blank the protocol daemon directory is used. So it returns the FAILED 15 return status due the impossibility to delete the daemon directory. However, every other directory will be deleted is “secure_token” is found.

INJAIL “secure_token” | “/path/to/dir/to/in-jail/tfprotocol/”

INJAIL in-jails the TFProtocol damon in the directory specified by the second parameter. This is achieved only if “secure_token”, specified as first parameter, is permitted for that directory. The return status of this command could be:

OK
FAILED 1 : Access denied to location.
FAILED 16 : Missing parameter from command.
FAILED 48 : Failed to open ACL file in jail directory.
FAILED 49 : Invalid security token in jail directory.
FAILED 51 : The process is already in-jailed.
FAILED 50 : Invalid command before in-jail.

The keepalive server’s mechanism

The keepalive mechanism that the server provides allows a client to detect whether the server is still reachable; moreover, it allows to test if the socket can be used to transmit and/or receive actual data.
The TCP protocol is designed to allow long time disconnected links without breaking the state of the sockets at both ends. This means that when the line is up again, communication can continue.
However, there are some circumstances where fail-fast scenario is needed. That is, that any of the endpoints can realize that the other end it is not reachable, so communication cannot continue anymore.
The aforementioned scenario is where the keepalive mechanism becomes relevant.

Not every platform supports the native TCP keepalive mechanism; in fact, it is optional in the RFC document. Reached this point, some sort of scheme need to be implemented in order to allow applications to detect dead pears. There is not a straightforward way to do this, especially if the underlying platform does not support it. Therefore, this is where application layer keepalive mechanism takes place.

The TF Protocol server must implement at least one of the following keepalive mechanism.

Out-of-Band data.

The Out-of-Band data –hereinafter the oob -, if present, works as follow:

The client sends an oob byte, then it waits for the respond. Once this byte reaches the server, another oob byte it is sent back to the client immediately.

With this dynamic exchange, on the client´s side some timers should be set, and if they expire after a series of probes, the peer's dead can be assumed.

Of all possible sort of application layer keepalive mechanisms, this is the one that has less performance impact and it is very reliable in the sense that the byte is sent through the same socket without interfering with the normal data. This allows to detect if the socket is not transmitting anymore. Unfortunately, not all TCP implementations treats the oob in the same manner. As a result, some platforms do not support it.

The UDP_HOSTCHECK

This type of keepalive, if present, allows a client to detect if the server is still reachable. This does not imply the TCP connected socket is still able to send/receive data. It only implies that the server address/port can be reached. It works as follows:

The client sends and UDP packet to the same address/port that the TCP socket is connected. This packet contains only one byte and it is the decimal number 0.
Once the server receives the packet, it responds with another UDP packet containing only one bye, the decimal number 1.
If the replay packet that the server sends is received, it means that the server address/port is reachable. Once again, it does not imply that the connected TCP socket can still communicate.

The UDP_PROCHECK

This type of keepalive, if present, allows a client to test whether the server's instance that communicate with the TCP client's socket it is still running. This per-se does not guarantee that the TCP socket can communicate. There are a lot of conditions that produce scenarios where the instance is still running but cannot communicate; however, this is out of the scope of this document.

In order to get reliable feedback about the actual TCP socket, the server's help is needed through the **KEEPALIVE** command. This commands as explained in this document tells the server to send periodically TCP-layer keepalive through the opened socket. In case that all the probes fail, the default behavior is to close that server's instance.
Once that particular server's instance is closed, UDP_PROCHECK will 'see' that the communication can no longer continue and inform it back to the client. It works as follows:

The client requests the server's instance unique key with the **PROCKEY** command.
The client sets the server TCP-layer keepalives through the **KEEPALIVE** command.
Then it sends and UDP packet with the decimal number 1 as first byte and the retrieved key next to it.
As soon as this packet is received by the server, it checks whether that instance -identified by the key- is still running. In order to work properly the timeout and the number of probes of the UDP packets should be equal or slightly greater than the TCP-layer keepalives server setting.
Then the server creates an UDP packet with just one byte and send it back to the client.

The client can tell if the communication channel is still opened by reading the byte sent from the server. The possible values could be:

1 It means that the communication channel is still opened.
0 It means that the socket is no longer usable.

UDP_SOCKCHECK

This type of keepalive, if present, allows a client reliably to check if the TCP socket is still usable, without the server's **KEEPALIVE** command help. However, some platforms may not be able of implement it. It works as follow:

The client sends an Out-of-Band byte (oob) through the TCP socket -this differs from the first explained mechanism because the oob byte responded by the server is not needed-. Once this byte reaches the server, a counter is incremented. This counter is a 32bit unsigned integer and starts from 0. The client, on its side, does the same. There is no problem if the integer overflows.

Then the client sends an UDP packet with the decimal value 2 as the first byte and the unique key retrieved from the server next to it. Just as in the **UDP_PROCHECK.** But this time there is a difference: the counter inside the server.

Once the server receives the UDP packet, it gets the value of the counter inside the server's instance that is connected to the client. Then the server creates an UDP packet that will be at least one byte. There are two possible versions of the packet:

0 It means that the socket is no longer usable.
1 "counter"

It should be noticed that there is no whitespace between the first byte which tells the return status and the counter -without the double quotes-.

Once the client has the value of the counter returned from the server it can compare it with the local copy. If the counter returned by the server is less than the local copy of the client, it means that the last oob byte was not received by the server, and can try again.

With any of the above keepalive methods, after a series of probes the client can assume that the TCP socket is no longer usable.

A note of caution: all of the above mechanisms could be implemented at the server side in different ways and the performance hit will vary from implementation to implementation and from platform to platform. Be aware that any of the keepalive stated above, even the ones with less overhead, have implicit trade-offs. It could be processing-cycles vs bandwidth-usage; fail-fast notice vs resource-consumption, and so on. In any case, they do not come for free, it always implies a performance cost.

The notification system

The notification system is one of the modes in which the protocol works. This mode is invoked by issuing the STARTNTFY command. Once this is done there is no way to go back to the standard mode. In this mode is the client who wait for the server “notifications” instead of issuing command while server waits like in the standard mode. This method allows to avoid the polling every some interval to request the notification to the server with the inherit inefficiency of it.
The notification mechanism works by setting a string pattern and a directory for listening. Every some interval, defined by client side, the server searches for that pattern in the files of the directory associated. Such pattern must be the start part of name of the file to be considered a notification. Once the server finds it, it sends to the client the complete name of that file along with other information. Any directory can have associated severals notification, but the server will return only one of them. The order is undetermined, the only thing that the protocol guarantees is that if the client eliminates the received notification, the server will send the next one found, and so on until there is no one notification in the directory, in which case it sends no more data to the client.
In the same manner that a particular directory can have associated severals notification, the client can specify severals directory to listen.

In order to clarify:
By “notification” we mean a file which name starts with a string that match the specified pattern associated to that directory.
By “listen” we refer to the watchdog that server does in the specified directory each time the interval specified by the client side elapsed.

“Interval” must be taken as a contiguous part of the time continuum between to arbitrary points.
“Elapsed” must be taken as the length of an interval.

If the usage of the protocol requires both modes at the same time, the client must open two connections to the server at the same time, one for the standard mode to issue commands to the server; one to receive notification from the server.

ADDNTFY “token” “/path/to/directory/to/listen”

ADDNTFY sets a new notification which starts name is “token”, without the quotes, and the directory is specified in the second parameter. If “token” is specified by an empty string “” then all the files in the associated directory will be taken as notification regardless of its name. If the second parameter is specified by an empty string “” then the root directory of the protocol daemon is used. The return status of this command could be:

OK
FAILED 5 : Error adding new notification to listen.
FAILED 1 : Access denied to location.

Directories don’t apply as notification entities, so they will not be taken into account.

STARTNFY “interval”

STARTNTFY starts the watchdog to the list of directory specified by the ADDNTFY command. Once STARTNTFY is issued there is no way back to the standard mode. The “interval” parameter, if present, it is a decimal number expressed in seconds as 3.3 which means 3

seconds and 300 miliseconds. The resolution of the interval goes up to nanoseconds, so 3.000000001 is 3 seconds and 1 nanosecond. Of course the integral part of the interval could be 0 too, so 0.000000001 is 1 nanosecond and 0.0001 is 1 microsecond. This number can be expressed in scientific notation as in 1.0e-9. The precision of the interval parameter are 15 digits which is the "double" size specified by the IEEE 754, but the real processor resolution time to sleep could be grater than the requested. So do not assume that the interval specified will be respected. Even if the processor supports such time resolution, mostly operating systems today are preemptive, so the real time waited could be more than the one specified. Another thing to take into account is that the TF PROTOCOL does not specify how to implement the sleep routine for the watchdog, so in event of signals, hardware interruptions or events, the client can't tell whether the sleeping routine will compute the unelapsed time and it will wait for it.
In short, if the client usage of the protocol requires super-reliable time sleeping intervals, this isn't your protocol.

If the "interval" parameter is left empty the server will assign one by default. How many time the server assigns by default to the watchdog mechanism is implementation dependent.
The return status of this command is:

N "file-that-math-notification-token"
The N is the index number of the directory-token list in which the notification has occurred and the string next to it "file-that..." is the complete name of the file.

The client must respond to this with one of two possible ways:

OK
DEL

OK tells the server to continue notifying without deleting the notification. In this case, when the server iterates again over the list could send the same notification for that directory even if there is more to notify. In any case, the client can’t predict if the server will send a new one or and old one, for a particular directory.

DEL tells the server to delete the notification. This ensures that the next time the server iterates over the list it will send other notification than the previous one if there is more for a particular directory.

In the event that there is nothing to notify the server will continue listen in the directories without sending data to the client. The client in this case will continue waiting to receive notification from the server without consuming “quanta” for it.

By “quanta” we mean the time slice assigned by the operating system to a process in a time sharing the system.

It is up to the implementers of the client side of the protocol if there is going to be some protection to the interval specified for the watchdog. This is a special concern with really short intervals because it could lead the server to a quasi busy wait which leads to high consumption of processor time and a considerable drop of server’s performance.