The eXtended subsystem 1

The idea behind this subsystem is to provide a greater granularity in the operations on files, not a huge throughput and outstanding performance. Be aware that the implementation of the server is not compelled to implement any of the extended subsystems. The client can test if this subsystem is implemented by using a command of this subsystem. If the server responses “UNKNOWN”, you will realize that the subsystem is not implemented.

[Supports TFPSFS]
XS1_OPEN “path/to/file”

XS1_OPEN opens the indicated file in the first parameter and store the file descriptor if it succeeds. The return status for this command could be:

OK “file-descriptor index”

FAILED 1 : Access denied to location.
FAILED 30 : Directory is locked.
FAILED 17 : Source file does not exist.
FAILED 8: Requested file is a directory.
FAILED XS1 1 : No file descriptor hole available.
FAILED XS1 2 : Failed to open the requested file.

For security reasons, the returned value is an array index instead of the file descriptor itself. The file descriptor index is a 32bit positive integer. The maximum permitted opened file descriptors per process by the OS are implementation-dependent. The maximum number of file descriptors that XS1 API allows is INT_MAX which is 2147483647 (or 2^31-1).

The opened file descriptors by XS1_OPEN remains open until they are explicitly closed or the connection ends.

XS1_CLOSE “file-descriptor index”

XS1_CLOSE closes the file descriptor specified in the first parameter. The return status for this command could be:

OK

FAILED XS1 3: Invalid file descriptor.

XS1_TRUNC “file-descriptor index” “new-size”

XS1_TRUNC truncates the file associated with the file descriptor specified by the first parameter. The second parameter specifies the new size for the file. This parameter must be a number between 0 and 2^63-1, “a 64bit signed integer”. If the specified “new-size” is less than the file size, “new-size” becomes the size of the file. If the specified “new-size” is greater than the file size, according to the implementation, the “new-size” becomes the size filling with holes or the file if left unchanged. All this process does not update the file position. For example, if the file position is “n” and a truncation to zero takes place, the next write operation will size the file up to n+1, with the above behavior of zero-filling.
The return status of this command could be:

OK

FAILED XS1 3 : Invalid file descriptor.
FAILED 16 : Missing parameter from command.
FAILED XS1 4 : Value for truncation not specified.
FAILED XS1 5 : Error while truncating the file.

XS1_SEEK “file-descriptor index” “offset” “whence”

XS1_SEEK sets the file’s position of the specified file descriptor by the first parameter. The second parameter is the offset and could range from -9223372036854775808 or “-(2^63)” to 9223372036854775807 or “2^63-1”. The last parameter specifies a relative position in the file and could be one of this three possibilities: **CUR** stands for current position; **SET** stands for the beginning of the file; **END** stands for the end of the file. The resulting position will be the offset from one of these constants, measured in bytes, from the beginning of the file.

If the combination of “offset” and “whence” results in a position beyond the end of the file, the operation success. However, it is not going to take place a size increase of the file. If a write operation is done, bytes between the previous end of file and the new position are filled with zeros. Those zeros may be not written to disk; instead, the file could create a hole.
The return status of this command could be:

OK “new-position”

The “new-position” is a 64bit integer.

FAILED XS1 3 : Invalid file descriptor.
FAILED 16 : Missing parameter from command.
FAILED XS1 6 : Invalid value for seek.

FAILED XS1 7 : Invalid whence constant.
FAILED XS1 6 : Error while seeking in the file.

As examples, we can take:

XS1_SEEK “file-descriptor index” 0 CUR
This will return the current position in the file.
XS1_SEEK “file-descriptor index” -4 END
This will return the number of bytes from the beginning up to the END minus 4 bytes.

XS1_LOCK “file-descriptor index” “lock-type”

XS1_LOCK locks the file associated with “file-descriptor index”. The “lock-type” parameter indicates the type of the lock, which could be: **LOCK**, **LOCKW, UNLOCK**. LOCK tries to lock the file without blocking. LOCKW locks the file and waits until it has been succeeded. UNLOCK unlocks the locked file. This locking system has nothing to do with the LOCK command of the standard mode of the TF PROTOCOL. While that lock does not eliminate race conditions, XS1_LOCK it does. Once XS1_LOCK has been succeeded, it is guaranteed that only one client owns the file’s operations due to the mutual exclusive access. The lock operation is an atomic one and no interleaving will occur due to thread/process scheduler algorithm of the operating system.

Be aware that if the second parameter is specified by LOCKW, on heavy loaded systems with great parallelism, the process could be blocked for long periods of times. This mode can be used when such a risk does not exist, or it can be assumable. Another risk of using LOCKW is that deadlocks can occur if the client does not design well the “locking protocol”.

We use the “locking protocol” term because the lock that XS1_LOCK implements is an advisory lock mechanism. The operating system does not enforce it. Clients are the ones that get agree to lock the access to the file. For example, lock the file first, use it, and then unlock it. That is a common sequence of operations, but again, clients must agree on it.
The lock set by XS1_LOCK is not persistent, once the network connection ends, the lock will be released. A persistent and free rece-condition lock can be created by combining XS1_LOCK and LOCK command from the standard interface.

A bit of theory. (optional reading)
It is possible that some implementations add greater granularity to the lock mechanism that differentiates the read/write locks and the bytes ranges. If this is not well implemented, could result in a producer-consumer problem, even worst with the network in between. To avoid this, it is possible to add mutexes or semaphores, read/write counters, and timeouts. As the result of the increasing complexity of the interface for the client, wrong settings could be specified. This will result in unfair shares for consumers “readers” or producers “writers”. In the worst case-scenario it causes completely starvation.
If this extension to the XS1_LOCK is present in the XS1 subsystem, the command must be XS1_LOCKX, although the syntax and parameters will be implementation-dependent.
Although, the XS1_LOCK eliminates the producer-consumer problem by setting the same type of mutual exclusive lock to any kind of operation, fairness is a different thing. The way in which threads/processes are scheduled is defined by the operating system scheduler algorithm. Lie on this, guarantee that every

thread/process at some point has the chance to acquire the lock. The operating system's scheduler used to adjust fairness too.

The return status of this command could be:

OK

FAILED XS1 3 : Invalid file descriptor.
FAILED 16 : Missing parameter from command.
FAILED XS1 9 : Lock type unknown.
FAILED XS1 10 : Error while locking the file.
FAILED XS1 11 : Unable to acquire the lock.

The XS1 11 return status it's not an error, it is the return status when a lock can't be acquired in the non-blocking mode of XS1_LOCK.

XS1_READ "file-descriptor index" "bytes-to-read"

XS1_READ reads from file descriptor the specified number of bytes in the second parameter. The number of bytes to read must be equal or greater than 0 or equal or less than the communication buffer size minus the number of bytes of "OK" return status minus the white space. If communication buffer is 512 bytes, then the payload must be between 0 and 509. The size of the communication buffer is specified in the TF PROTOCOL documentation. The XS1_READ command updates the file descriptor position unless it fails. In case of a failure, the position of the file is restored.
If EOF "end of file" condition is found or the number of bytes to read is set to 0, then XS1_READ returns just "OK" and the white space that separates commands from payload. The client can realize about the EOF because the payload will be 0. In other words, the

length of the received data at client side will be exactly 3 bytes, which is the OK return status plus the white space.
Be aware that XS1_READ can return less bytes than the requested and it is not necessary an error. This happens if there are not that many bytes left in the file.
The return status of this command could be:

OK “payload”

The “payload” could be empty due to EOF or a zero specified as the second parameter.

FAILED XS1 3 : Invalid file descriptor.
FAILED 16 : Missing parameter from command.
FAILED XS1 12 : Invalid length value for read.
FAILED XS1 13 : Error while reading the file.
FAILED XS1 14 : Network operation error.

XS1_WRITE “file-descriptor index” “bytes-to-write” “payload”

XS1_WRITE writes to the file descriptor the specified number of bytes in the second parameter. The number of bytes to write must be equal or greater than 0 or equal or less than the communication buffer size minus the number of bytes of XS1_WRITE command, minus the white space, minus the “file-descriptor index” -which is a 32 bit integer, 10 digits- minus another whitespace minus the maximum possible digits of the third parameter, minus another white space. If communication buffer is 512 bytes, then the payload must be between 0 and 487. The size of the communication buffer is specified in the TF PROTOCOL documentation. The XS1_WRITE command updates the file descriptor position unless it fails. In case of a failure the position of the file is restored.

The return status of this command could be:

OK

FAILED XS1 3 : Invalid file descriptor.
FAILED 16 : Missing parameter from command.
FAILED XS1 15 : Invalid size value for write.
FAILED 13 : Quota exceeded.
FAILED XS1 16 : Error while writing the file.

XS1_READV2

XS1_READV2 reads from an open file descriptor the specified number of bytes. Unlike XS1_READ, XS1_READV2 uses a 12 byte header. The first 4 bytes of the header is the open file descriptor returned by XS1_OPEN. The next 8 bytes contains the number of bytes to be read. Both integers must be converted to big-endian before send it over the network. After send XS1_READV2 the server expects the previous indicated header. After receives the header the server will response with a 64 bit integer header converted to big-endian. This header may contain 0, -1 or some positive value. In case of -1, this indicates that an error has occurred and a message will come next. This message will be exactly 126 bytes. After that the server will return to listen for commands again. If the server sent 0 it means that there is nothing else to be read from the file indicated in the first 4 bytes of the header sent to the server. At this point the server returns for listening commands again. If the server sent some positive value it means that that number of bytes come next. After the server sent the payload read from the file it return again for listening for commands again.

The header that the client must send after XS1_READV2 looks like the following.

```
typedef struct v2hdr {
    int32_t fdidx;
    uint64_t bytes;
} V2hdr;
```

XS1_WRITEV2

XS1_WRITEV2 writes to an open file descriptor the specified number of bytes. Unlike XS1_WRITE, XS1_WRITEV2 uses a 12 byte header. The first 4 bytes of the header is the open file descriptor returned by XS1_OPEN. The next 8 bytes contains the number of bytes to be written. Both integers must be converted to big-endian before send it over the network. After send XS1_WRITEV2 the server expects the previous indicated header. After receives the header the server will read from the network the indicated number of bytes by the 64bit -8 bytes- that follows the first 4 bytes of the header. After that, the server will tries to write the payload to the file represented by the first 32bit -4 bytes- of the header. At this point the server will return a 64bit integer converted to big-endian that could be -1 or the number of bytes actually written to the file. If is -1 an error messages of exactly 126 bytes come next. After that the server returns again for listening commands. If is any other value it means the bytes written to the file and the server returns again for listening commands.

The header that the client must send after XS1_WRITEV2 looks like the following.

```
typedef struct v2hdr {
    int32_t fdidx;
    uint64_t bytes;
} V2hdr;
```