Simple One-directional File Transfer

Version 2.0

Protocol Design (IN2333), SoSe 2020

Jade Riedel, Pooja Parasuraman, Dominik Pham,
Joseph Birkner, Johannes Abel, Peter Okelmann
(danield.riedel, pooja.parasuraman, dominik.pham,
joseph.birkner)@tum.de + (abel, okelmann)@in.tum.de

August 13, 2020

ABSTRACT

The Simple One-directional File Transfer (SOFT) protocol is a file sharing mechanism where a Client can request one or more files from a Server which responds back with the requested files. The unique feature of this file sharing protocol in comparison to other protocols in existence is that it uses UDP as its transport protocol but is still able to guarantee reliability. Additionally, SOFT also comes with flow control as well as a flexible congestion control mechanism. This document provides a specification of SOFT to enable third-party implementations and to solicit community feedback through experimentation on the performance of SOFT.

TABLE OF CONTENTS

1. INTRODUCTION

The Simple One-directional File Transfer (SOFT) protocol is a UDP-based protocol to transfer files from a Server to a Client. Although it uses UDP as its underlying transport protocol, SOFT guarantees reliable file transfer and is even able to resume a disrupted file transfer or recover from a connection drop. Additionally, SOFT supports flow control by enabling the Client to announce its maximum buffer size in a file request so that the Server can adjust its packet size accordingly. The protocol also supports a flexible loss-based congestion control by increasing or decreasing the number of packets that can be sent by the Server without an acknowledgment being required. The amount of increase or decrease can vary and is dependent on the implementation of this protocol.

2. CONVENTIONS

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [1] [2] when, and only when, they appear in all capitals, as shown here.

3. TERMINOLOGY

Server
: An entity that owns the file(s) and transfers them upon a request.

Client
: An entity that initiates the file transfer and explicitly requests the file(s).

Host
: A host is a SOFT entity denoting either a Client or a Server.

Receiver/Sender
: Servers and Clients both send and receive application layer data. Therefore, each Server and Client may be referred to as both receiver and sender in their respective function.

Object
: A transport layer unit of data that can be transferred from one Soft entity to another.

Chunk
: A file will be split into a set of smaller units called Chunks that can be transferred. The default size of a Chunk is 512 bytes.

session
: A unique Client ID is assigned to a Client by the handshake procedure. A session is identified by the assigned Client ID until the Server holds the Client ID for the Client.

IN-ORDER
: Ascending order without any missing numbers in between starting from 0.

implementation-specific
: Certain parts of this specification might not have concrete definition or a defined algorithm for implementation. These areas are denoted with the keyword "implementation-specific" so that any algorithm can be used to achieve the required functionality that aligns with the protocol specification.

connection loss
: A connection is considered to be lost between the Server and a Client when no data is able to flow between the Server and the Client.

version
: The version of the SOFT protocol specification.

Ordering Constraints
: Constraints for the ordering of packets based on the Chunks present within them

4. SOFT PROTOCOL TRANSPORT LAYER

4.1. Procedure Overview

This section describes the basic procedure of abstract data transfer, acknowledgement and error handling between a Server and a Client in the SOFT Protocol.

- Client begins a Session by initiating the Handshake procedure per section 4.2.
- During the Handshake, the Server generates a unique Session ID. This ID is needed to identify Clients during the error handling and acknowledgement process.
- The Client and Server are now able to exchange SOFT messages which may contain Chunks of Objects from the application layer, errors or acknowledgments.
- The Client and Server are able to exchange Objects by sending SOFT Messages with an Object Header TLV per section 6.1.3 and an Object Chunk TLV per table section 6.1.4 as part of the payload.
- The Server may indicate that a Chunk needs to be acknowledged by setting the "ACK Required" bit in the Object Chunk TLV accordingly.
- The Client is able to acknowledge a Chunk of an Object by sending an ACK TLV per section 6.1.6.
- The Server is able to explicitly request the acknowledgment of a Chunk by sending an ACK Request TLV per section 6.1.8.
- If an error occurs on the SOFT Transport Layer both Client and Server are able to send an Error TLV per section 6.1.7.

4.2. Handshake

This section describes the Handshake procedure to establish a connection between a Server and the Client and its contents.

The Client is able to establish a connection by sending a SOFT Message per section 6.1.1 with the Session ID being set to zero and a payload containing exactly one Host Information TLV as per section 6.1.2 as the first TLV in that SOFT Message. This Host Information TLV shall contain information about the Client.

If the Server receives a SOFT message with the Session ID being set to zero, the first TLV of that SOFT Message must be a Host Information TLV. The Server must respond with a SOFT Message containing a Host Information TLV as its first TLV. This Host Information TLV shall contain information about the Server.

The Host Information TLV contains information to facilitate a reliable communication between Server and Client. The Host Information TLV includes the maximum buffer size of the Host, the maximum number of out-of-order packets that the Host can handle (a value of 0 means the Host is not able to perform out-of-order processing of packets), as well as the Host's operating system. The field "Frequency of Acknowledgement" is used to request an acknowledgement frequency strategy. A Host may choose a different acknowledgement frequency strategy than the one requested in the Host Information TLV. The Host Information TLV also carries the ID and version of the used Application.

If a Client receives a SOFT Message from a Server, which has not yet sent a Host Information TLV, and this SOFT Message does not contain a Host Information TLV, the Client shall send a Host Information TLV again.

If a Server receives a SOFT Message from a Client, which has previously sent a Host Information TLV, and this SOFT Message contains a Host Information TLV, the Server shall send a Host Information TLV.

4.3. Object Transfer

This section describes the contents and procedure of an Object Transfer.

4.3.1. Object Composition

Objects are transferrable data abstractions of the SOFT Transport Layer. In this abstraction everything the Application layer is transferring through the SOFT Transport Layer is an Object.

Objects have a unique Object ID. The Object ID of any Object must remain static for the lifetime of this Object. The lifetime of any Object must be at least sufficient to transfer this Object. For Objects of long lifetime (such as files), hashes shall be used to assign its Object ID. For Objects of short lifetime, a sufficiently random integer shall be used to assign its Object ID.

Any Object shall be static during its transfer. The SOFT Transport Layer shall have random access into the Chunks of the Object during its transfer. The number of chunks shall also be fixed during an Object's transfer and known at the start of this transmission.

Any Object has an Object Type. The interpretation of an Object Type is defined by the application layer as per section 6.2.1.

An Object may be composed of heterogenous data fields (an Object of Type File may consist of Metadata and File Content). To disambiguate these Fields an array gives their length (in number of Chunks) and Type. The interpretation of a Field's Type is defined by the application layer as per section 6.2.1. A Chunk of an Object may not contain more than one field's data.

The content of any Object Chunk must be unambiguous. The content of any Object Chunk must not contain incomplete Application Layer TLVs.

An Object is separated into 2 distinct parts, the Object Header and the Object Chunk of the Object. An Object must have exactly one header. An Object's Content may consist of any number of Object Chunks.

4.3.2. Transfer Process

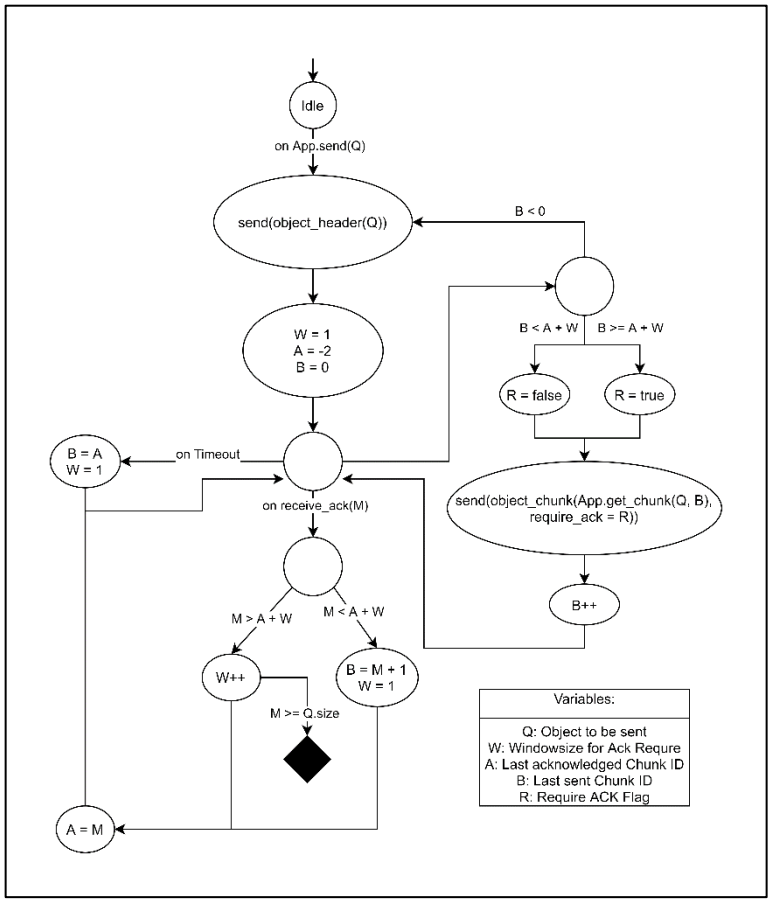

Figure 4.1 State Diagram of an Object being sent

To transfer an Object both its Header and Content must be transferred. The Header of an Object shall be transferred via a SOFT Object Header TLV as per section 6.1.3. The Content of an Object shall be transferred via a SOFT Object Chunk TLV as per section 6.1.4.

The Header of an Object shall be sent before sending that Object's Content. The Content of an Object must be sent in complete and increasing order. The first Chunk of an Object to be sent after that Object's Header is Chunk 0. A Chunk with Chunk ID N can only be sent if the Chunk with

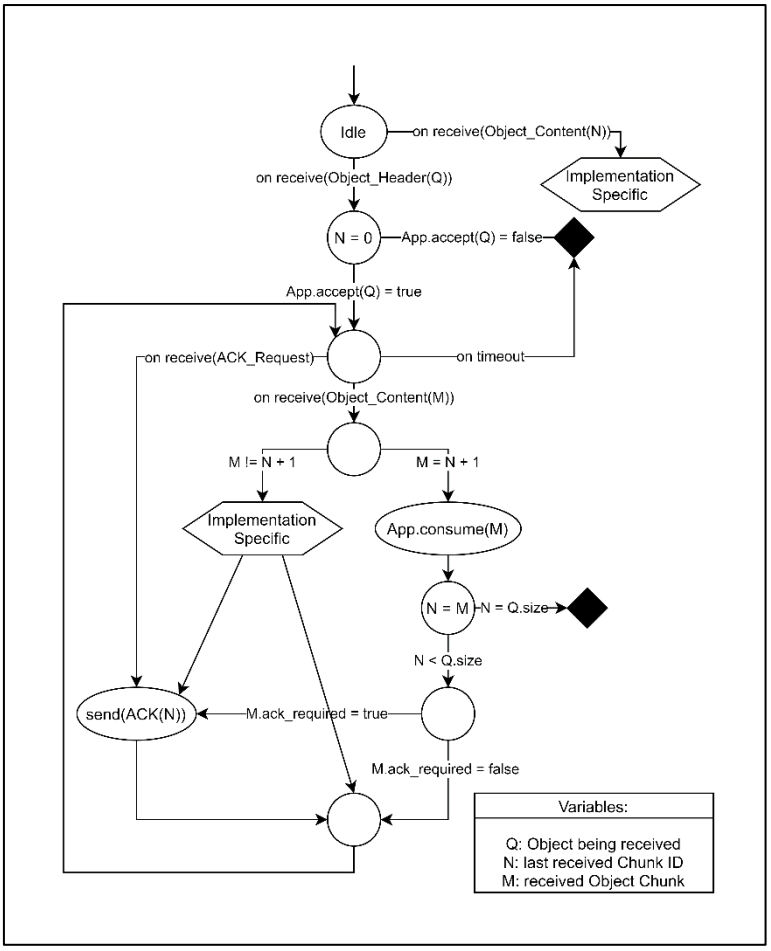

Figure 4.2 State Diagram of an Object being received

Chunk ID N-1 has been sent (excluding the 0th Chunk). For the purposes of acknowledgement, the Object Header is considered to have Chunk ID -1.

If a SOFT Object Chunk TLV of an Object is received without receiving that Object's Header, the ordering constraints are considered violated.

If a SOFT Object Chunk TLV with Chunk ID N is received without receiving that Object's Chunk with Chunk ID N-1, the ordering constraints are considered violated.

4.4. Object Skip

If a Host is sending an Object and is triggered to skip to a different Chunk ID a SOFT Object Skip TLV shall be sent as per section 6.1.5. She Object ID field and the Chunk ID field shall be set accordingly.

If a Host that previously sent an Object Skip TLV for an Object receives an ACK TLV for that Object with Chunk ID less than the Chunk ID of the most recent Object Skip TLV for that Object, the Host shall end an Object Skip TLV for that Object with the same Chunk ID as the most recent Object Skip TLV for that Object.

If a Host receives an Object Skip TLV for an Object, the Host shall send an Acknowledgement for that Object with the Chunk ID set to the Chunk ID in

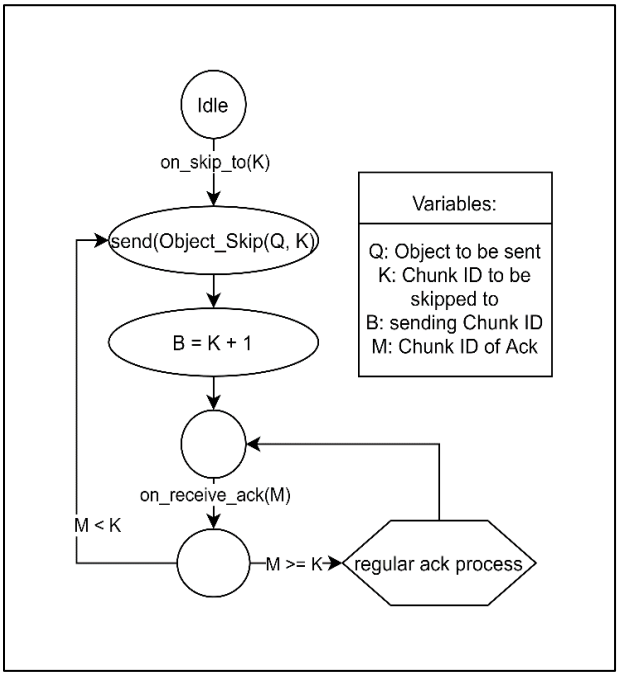

Figure 4.3 Extended State Diagram of receiving an Object with the skip feature

the Object Skip TLV. Furthermore, the Host shall continue receiving as if it had received all Chunks up to the Chunk ID in the Object Skip TLV.

4.4.1. Reliability of Object Transfer and Congestion Control

The transport layer MUST guarantee for each SOFT Object that all its chunks are received (completeness). Furthermore, it MUST guarantee that each chunk has no bit errors according to the FNV hash checksum of the SOFT network packet (correctness).

As reliability is provided on a per-object-basis, the acknowledgement process defined here is instantiated per Object as well.

Completeness

For each SOFT Object Hosts keep track of the Chunk ID up to which all other Chunks have been received without checksum error. When sending an acknowledgement for an Object, this Chunk ID is included int the SOFT ACK TLV as per section 6.1.6.

A Host may request an acknowledgement either through setting the ACK required flag of an Object Chunk TLV to '1' as per section 6.1.4 or by sending an ACK Request TLV as per section 6.1.8. If a Host receives an Object Chunk TLV with the ACK required flag set to '1' or an ACK Request TLV the Host shall send an acknowledgement for that Object as described in section 4.4.2.

Acknowledgements for chunks which were already acknowledged, may be used as a hint towards packet loss. It is up the sender, to adjust the frequency of requiring acknowledgements from the receiver. The receiver may use the frequency of acknowledgement field in the Host Information TLV as per section 6.1.2 to indicate the preferred acknowledgement frequency strategy.

Correctness

Each SOFT Message Frame contains a 32-bit FNV-Hash [3] as per section 6.1.1. When a Host receives a SOFT Message, it shall compare the Checksum field to the FNV-Hash of the message (excluding the Checksum field). If a Host detects an Error, the host MUST drop that packet as none of the contained information may be trusted. The Host SHOULD respond with an Error TLV with error type Checksum Error.

Congestion- and Flow-Control

At connection establishment a receiver provides information like the maximum number of out-of-order packets it is able to receive, and its buffer size measured in bytes. Both numbers may be used by the sender for transmission optimization. The latter one for example may be used to initialize the maximum number of inflight packets.

The sender shall change its sending rate over time according to an implementation specific algorithm: On every received acknowledgement the sender shall increase its rate for example by the number of newly acknowledged packets. The sender shall decrease its sending rate for every packet loss it detects for example by a factor of two.

Hence, if the link is congested and packets are therefore lost, the sender reduces its sending rate. If the receiver is overloaded and therefore cannot or chooses not to send acknowledgements, the sender assumes packet loss as well and therefore reduces its sending rate.

4.4.2. SOFT Acknowledgement Process

The SOFT Acknowledgement Process is used to ensure the reliable transfer of Objects.

If a Host receives an Object Chunk TLV as per section 6.1.4 with the ACK Required flag set to '1', or the Host receives an ACK Request TLV as per section 6.1.8 including that Object’s Object ID, the Host shall send an ACK TLV as per section 6.1.6 including that Object’s Object ID. The ACK TLV must contain the Chunk ID of the Chunk of that Object, which has the highest ID that was last received IN-ORDER without checksum error.

If a Host does not receive an ACK TLV as per section 6.1.6, within some implementation-specific timeout window, the Server may decide that the packet(s) is/are lost and may send an ACK Request TLV or restart sending Chunks beginning at the last successfully acknowledged Chunk of the affected files.

If a Host receives an ACK TLV as per section 6.1.6 with a Chunk ID (M) that is less or equal to the last successfully acknowledged Chunk ID (A) of that Object, the Host shall restart its sending at the following Chunk (M+1).

4.4.3. Fragmentation

In order to use the largest possible unfragmented packets, multiple TLVs may be sent per Message. Any Host shall avoid composing any packets of sizes that are likely to be fragmented. Path MTU Discovery (over ICMP) may be used to find the largest unfragmented packet size [4]. The chunk size of 512 Bytes should be reasonably small enough to avoid most fragmentation [5]. No packet shall exceed 1500 Bytes in size [6].

4.5. Interface

The following section defines the interface of the SOFT Transport Layer and an Application Layer.

Initialization

Interface	Description
void connect(target, host_information)	called by the Application Layer to establish a connection to target
void on_connect(connectee, host_information)	event triggered by an incoming Handshake

void connect(target, host_information)
called by the Application Layer to establish a connection to target

void on_connect(connectee, host_information)
event triggered by an incoming Handshake

Sending

Interface	Description
object_ptr send(object_descriptor, starting_chunk = 0)	called by the Application Layer to begin sending an Object
chunk get_chunk(object_ptr, chunk_id)	called by the Transport Layer to request a chunk from the Application Layer
void abort_sending(object_ptr)	called by the Application Layer to stop sending an Object
void finished_sending(object_ptr)	called by the Transport Layer when sending of an Object is complete
void skip_to(object_ptr, chunk_id)	called by the Application Layer to continue sending from a certain Chunk

object_ptr send(object_descriptor, starting_chunk = 0)
called by the Application Layer to begin sending an Object

chunk get_chunk(object_ptr, chunk_id)
called by the Transport Layer to request a chunk from the Application Layer

void abort_sending(object_ptr)
called by the Application Layer to stop sending an Object

void finished_sending(object_ptr)
called by the Transport Layer when sending of an Object is complete

void skip_to(object_ptr, chunk_id)
called by the Application Layer to continue sending from a certain Chunk

Receiving

Interface	Description
<bool, object_ptr> accept(object_header)	called by the Transport Layer when an Object Header is received. Application Layer may send false if it wishes to not receive that Object
void consume(object_ptr, chunk_id, object_chunk)	called by the Transport Layer when an Object Chunk is received to trigger the Application Layer to process that chunk

void abort_receiving(object_ptr)	called by the Application Layer to stop receiving an Object
void finished_receiving(object_ptr)	called by the Transport Layer when receiving of an Object is complete

<bool, object_ptr> accept(object_header)
called by the Transport Layer when an Object Header is received. Application Layer may send false if it wishes to not receive that Object

void consume(object_ptr, chunk_id, object_chunk)
called by the Transport Layer when an Object Chunk is received to trigger the Application Layer to process that chunk

void abort_receiving(object_ptr)
called by the Application Layer to stop receiving an Object

void finished_receiving(object_ptr)
called by the Transport Layer when receiving of an Object is complete

4.6. Error Handling

This section describes how errors in the SOFT protocol transport layers are handled. The sender of an error can either be a Client or the Server. The same applies to the receiver of an error.

If an error occurs on the SOFT transport layer, the sender shall notify its communication partner about it by sending a SOFT Message containing an Error TLV as per section 6.1.7. The sender must specify what type of error occurred by setting the "Error Code" Field in the TLV accordingly.

Packet Loss:
If the Client receives an Object Chunk TLV with the ACK Request bit set to '1', or receives an Ack Request TLV, and the Client is missing Chunks of this file with Chunk ID smaller than this the Client shall send an acknowledgement for the Chunk, which has the highest ID that was last received IN-ORDER without checksum error, as per section 4.4.2.

Checksum Error:
If the Checksum of a SOFT Message is incorrect the Message shall not be processed, as its content cannot be trusted to be correct. A Transport Error TLV with the Error Code set to Checksum Error '0x04' shall be sent as per section 6.1.7.

If a Host receives a Transport Error TLV with the Error Code set to Checksum Error '0x04' the Host may deal with these in an implementation specific way. Possible actions might include continue sending as usual or send ACK Request TLVs as per section 6.1.8.

Double Receive:
If a Host receives an ACK TLV as per section 6.1.6 with a Chunk ID (M) that is less or equal to the last successfully acknowledged

Chunk ID (A) of that Object, the Host shall restart its sending at the following Chunk (M+1) as per section 4.4.2.

No ACK received:
If the Server requests an Acknowledgement for an Object by sending an ACK Request TLV or setting the ACK required bit and the Server does not receive an Acknowledgement for this file within an implementation-specific time limit the Server may send an ACK Request TLV or restart sending the Chunks of this file beginning at the last successfully acknowledged Chunk of that file as per section 4.4.2.

Unsupported Version:
A Host shall always initiate its communication with its highest supported version of the SOFT Transport Layer. If a Host receives a SOFT Message with a Version number outside its supported version range an Transport Error TLV shall be sent with the Error Code set to Unsupported Version '0x05' as per section 6.1.7. The Error Data shall contain the maximum and minimum supported range of version.

If a Host receives a Transport Error TLV with the Error Code set to Unsupported Version '0x05' communication shall be proceeded with the highest common version number. If no version number can be found in common, communication shall be terminated in an implementation specific way.

Session Unknown:
When a Host receives a SOFT Message with a Session ID that is either unknown or is already dropped a Transport Error TLV with the Error Code set to Session Unknown '0x06' shall be sent as per section 6.1.7.

If a Host receives a Transport Error TLV with the Error Code set to Session Unknown '0x06', all in flight Objects shall be aborted and a new Session shall be established via the Handshake Procedure as per section 4.2.

Transfer of an Object cannot be resumed after a Session expired.

Object Abort:
If a Host wants to stop the transmission of an Object a Transport Error TLV shall be sent with the Error Code set to Object Abort '0x08' shall be sent as per section 6.1.7. The Error Data shall contain a list of Object IDs that shall be aborted.

If a Host receives a Transport Error TLV with the Error Code set to Object Abort '0x08' the Host shall cease the sending of all Objects with Object ID in the Error Data and report their abort to the Application.

5. SOFT PROTOCOL APPLICATION LAYER

5.1. Procedure Overview

This section describes the basic procedure of transferring a file with this protocol.

- The Client initiates a connection with the Server.
- The Client may request files at any time.
- The Server identifies the requested files. If files cannot be served an Error Report Message is sent.
- The Server determines the order in which the files need to be sent, based on file size.
- The Server sends Chunks of one or multiple files.
- The Client may try resuming a transmission after timeout or error occurs by sending a File Resume Message.
- If the Client gets an Invalid File Resume Request error from the Server, the Client may resend request for files to restart the connection.
- A Client may send a file list request for a certain path and recursion depth. A Sever responds with a list of the files in the directory at the given path, or an Error Report Message.

5.2. Interaction with the Transport Layer

When the app layer wants to send a File Metadata TLV and Content TLVs, the following object would be presented to the transport layer:

- Object
  - Type: File Response Message (Section 6.2.3), 0x11 (Table 6.10)
  - Size: 10 Chunks
  - List of fields:
    - Field:
      - Type: File Metadata, 0x22 (Table 6.15)
      - Length: 2 Chunks
      - List of chunks:
        - Chunk 0 (a transport layer chunk). List of TLVs:
          - Metadata Entry TLV: (Table 6.16)
          - Metadata Entry TLV ...
        - Chunk 1 (another transport layer chunk)
          - Metadata Entry TLV ...
    - Field:
      - Type: File Content, 0x23 (Table 6.14)
      - Length: 8 Chunks
      - List of chunks:
        - Chunk 2: List of TLVs:
          - File Content TLV (Table 6.14 as well)
        - ...
        - Chunk 9: List of TLVs:

- File Content TLV

Chunks will be given to the transport layer when it asks for it. The type information around it is given to the application as dedicated object type information (referenced as object_descriptor in Section 4.5).

5.3. File Request

This section describes the contents and procedure of a File Request.

If triggered, a Client shall send a File Request Message as per section 6.2.2 with a File Request TLV included in the message. A File Request Message must contain at least one file. Each requested file must have a file name or a file path. The filename is delimited by its length field and need not be delimited by a null byte. The Client may request more than one file in a single File Request Message.

If the name of the file is not known to the Client, then the Client may send a File List Request as per section 5.7.1 to get the list of files held by the Server.

5.4. Resuming File Transfer

A File Request Message is used to resume at least one of the previously disrupted file transfers. A File Resume Request TLV shall be used for this purpose.

If triggered, the Client shall send a File Request Message to the Server per section 6.2.2. The Client shall include a File Resume Request TLV in the File Request Message.

The File Resume Request TLV must contain the File ID and Chunk ID of the Chunk, which has the highest ID that was last received IN-ORDER, for each file. The File Resume Request TLV must contain Chunk ID and File ID of at least one of the files for which the transfer was disrupted.

If the Server receives a File Request Message from the Client, with a File Resume Request TLV included in the message, within some implementation-specific timeout window, it shall then resume the file transfer by sending the Chunk which have ID(s) equal to one greater than the Chunk ID (e.g., $(n+1)^{th}$ Chunk with Chunk ID = n+1) for the file(s) received in the File Resume Request TLV. The timeout window shall be initiated immediately after the file transfer(s) were disrupted and must be maintained locally by the Server.

The server must notify the transport layer to ignore requesting chunks from chunk id 0 and shall start requesting the chunks from the ID mentioned in the interface call **"skip_to"** as per section 4.5.

If the Server receives a File Request message from the Client, with a File Resume Request TLV included in the message, after the implementation-specific timeout window, then the Server shall send an

Error Report Message per section 6.2.4 with error code set to 0x06 indicating that the resume request is invalid for the files mentioned in the tlv.

If the Client receives an Error Report Message which includes in the message with the Error Code set to 0x06, the Client may send a new File Request Message per section 6.2.2 for the file(s).

5.5. File Response

If the Controller receives a File Request Message containing a File Request TLV included in the message per section 5.3, and if the Server has access to the requested file(s), then the Server shall start the transmission of the file(s) as units of chunks by sending the File Response message.

The Server must send a File Response message with at least one Metadata TLV or one File Content TLV.

5.5.1. File Metadata

If the Controller receives a File Request message for a file from a Client, then the Server shall start the transfer of file by sending a File Response message with a Metadata TLV per Table 6.15 containing the metadata of the requested file.

The metadata of a file contains information about that file. Depending on operating system it may contain information such as

- File size
- Filename
- Permissions
- Dates of creation, modification, change
- A hash of the file

An entry of type file size is required to be present per file. An entry of type SHA3-512 [7] is required to be present per file. All other Metadata contents are optional. If the Server cannot determine the size of a file, '-1' shall be used to denote unknown file size. The Client may reject a file of unknown size with a File Abort Error as per section 5.6.5.

The Metadata TLV may contain more than one metadata entry. The metadata is in the Soft Metadata format. The Server can decide what Metadata to include in any File Metadata TLV. The Client may choose to ignore any Metadata. The types of Metadata entries are defined in Table 6.16.

Metadata of a file MUST be sent before its content. If the contents of the file are sent before the file Metadata, then it is considered as the Metadata Ordering constraints are violated.

There can be more than one Metadata TLV for a file.

Number of metadata TLVs = m where m > 0

If the Client receives a File Metadata TLV with a File ID the Client did not previously request, it may send a File Abort Error for that file as per section 5.6.5.

5.5.2. File Content

Once all the metadata information of the file has been sent, the Server shall then start the transmission of the actual contents of the file by using the File Content TLV of the File Response message per Table 6.14.

The Server splits large files into units of chunks. The chunk size (s_{chunk}) is by default fixed to 512 bytes as recommended in section 4.4.3.

All files larger in size than the default chunk size must be split in contiguous chunks of that size. Only the final Chunk of a file may be smaller in size than s_{chunk}.

The first Chunk of a file's content shall be assigned with the chunk id as m, where (m-1) denotes the chunk ID of the last Metadata TLV. The Chunks of any single file must be sent in complete and ascending order

The nth Chunk of the file Contents shall contain byte $(n)*s_{chunk}$ to $\min((n+1)*s_{chunk}-1$, End of File) of that file (beginning at 0). The chunk ID of the nth Chunk of the File Contents shall be m+n+1.

A File Response may contain any number of Chunks of any number of files. Chunks of different files do not need to be synchronized.

The Server shall include a unique File ID for every file. File ID shall be generated using the SHA3-512 [7] hash of the entire file contents.

If the Client receives a File Content TLV with a Chunk ID m+n+1 and the Client has not received ALL of the file's Chunks with Chunk ID 0 to m+n+1, then the file's contents are considered to be received out-of-order.

If the Client receives a File Content TLV that does not violate the Metadata Ordering Constraints the chunks of content shall be copied into the file.

If the Client receives a File Content TLV for a file that the Client did not previously request, it may send a File Abort Error for that file per section 5.6.5.

5.6. Error Report Message

An Error Report Message shall be sent to notify a Client or the Server that an error condition has occurred. It should contain an Error Report TLV with code and optional content to tell the Client that any of the following error conditions has occurred.

5.6.1. File Not Found (0x01)

If a Server receives a File Request Message per section 6.2.2 from the Client, with a File Request TLV included in the message, for one or more files and if the Server could not find the location of the file(s) requested by the Client, the Server shall then send an Error Report Message.

The Error TLV must contain the File ID(s) in the error data field for all the requested files that could not be located by the Server. The Error Code must be set to 0x01 in the Error TLV to denote that the requested file(s) is/are unknown to the Server.

5.6.2. File Changed Before Transfer Resume (0x02)

If a Server receives a File Request Message per section 6.2.2 from the Client, with a File Resume Request TLV included in the message, for one or more files for which the transfer had been previously disrupted and if any change(s) to the requested file(s) has/have occurred, the Server shall then send an Error Report Message.

The Error TLV must contain the File ID(s) in the error data field for all the requested files which have changed before the transfer resume request. The Error Code must be set to 0x02 in the Error TLV to denote that the requested file(s) has/have been changed before resuming the transfer.

5.6.3. No Space Left on Device (0x03)

If a Client receives a File Response Message per section 6.2.3 from the Server, with a File Content TLV included in the message, containing Chunk(s) of the previously requested file(s) and if the Client could not process the received file contents due to space constraint on the device on which the Client is operating, then the Client shall send an Error Report Message.

The Error Code shall be set to '0x03' in the Error TLV to denote that further Chunks of any file sent by the Server cannot be processed by the Client.

If the Server receives an Error Report Message with an Error TLV included in the message with the Error Code set to '0x03', the Server shall then stop all the file transfers to the Client until any one of the following conditions happens.

- A File Resume Request for the aborted file is received from the Client within some implementation-specific timeout window.
- A File Request Message for any file is received from the Client for one or more files.

5.6.4. File Hash Error (0x04)

Once a Client has received all chunks that belong to a specific file object, it MAY calculate the hash of the file content and compare it against the hash which was advertised in the file metadata.

IF the computed and advertised values match, no further action is necessary. OTHERWISE, a Client MAY notify the Server about the error condition, by sending an Error Report Message with an Error TLV included in the message per Table 6.17. The Client must set the Error Code in the Error TLV to 0x04 indicating that there is a mismatch in the hash of the file included in the Error TLV.

Additionally, the Client may request to send the file again by sending a File Request Message per section 5.3.

If the Server receives an Error Report Message with an Error TLV included message with the Error Code set to 0x04, it may choose to ignore the error or may start the file transfer from chunk 0, including the metadata.

5.6.5. File Abort (0x05)

A Client MAY send this error to the Server to denote that the transfer of file ID(s) included in the Error data SHALL be aborted.

If triggered, a Client shall send a File Report Message with an Error TLV included in the message with the Error Code set to '0x05' to denote that the transfer of the File ID(s) included in the Error Data shall be aborted. The Error Data must include at least one File ID.

If the Server receives a File Report Message with an Error TLV included in the message with the Error Code set to '0x05', the Server shall then stop the file transfers for all the files that are included in the Error data field. The Server may not support resuming the transfer for aborted files.

5.6.6. Invalid Resume Request (0x06)

A Server MUST send this error to a Client with IDs of files that have been previously issued in a file resume request message, for each File ID for which the Server does not remember the interrupt point.

The Server shall send an Error Report Message per section 6.2.4 with an Error TLV included in the message with the Error Code set to 0x06, indicating that the resume request is invalid.

If the Client receives an Error Report Message which includes in the message with the Error Code set to ‘0x06’, the Client may send a new File Request Message per section 6.2.2 for the file(s).

5.6.7. Invalid Depth for List (0x07)

If a Server receives a File List Request message with the level of recursion (=n) and if the Server could not handle the level of recursion, then the Server shall send an Error Report Message with an Error TLV included in the message. The Error Code must be set to ‘0x07’ in the Error TLV to denote that the request has invalid depth field for the specific directory path.

When the Client receives an Error Report Message with the Error Code set to '0x07' in the Error TLV in the message, then the Client may resend the File List Request message with a lesser level of recursion.

5.6.8. Unknown Format Code (0x08)

A Server MUST send this Error Code if it does not support the Format Code in the request.

If the Server receives a File List Request message with a File List Request TLV included in the message and if the Server does not support the format code denoted in the File List Request TLV, then the Server shall send an Error Report Message with an Error TLV included in the message. The Error Code must be set to '0x08' and it shall contain all Format Codes that are supported by the Server (empty list if none is supported).

If the Client receives an Error Report Message with the Error Code set to '0x08' in the Error TLV, then the Client shall not send further list requests or may send a File List Request message with one of the Format Codes supported by the Server.

5.7. File List

File List enables a Client to retrieve information about the files provided by a Server.

File List is OPTIONAL for the Client, i.e. the Client can safely ignore implementation of this feature and this section.

File List is partly optional for the Server. The Server MAY choose not to support any Format Code and always respond with an Error Report Message as per section 5.7.2. The Server MUST respond, however.

This document defines Format Code 0x01 in section 5.7.3. Other Format Codes may be subject to future specification in separate documents.

5.7.1. File List Request

A Client MAY ask a Server to provide an overview of the files it provides, by sending a File List Request Message with a File List Request TLV as per section 6.2.5, which must contain a directory path, a level of recursion and a format code.

The directory path indicates where the Server should look to generate a file list response. A directory path of "." shall indicate the working directory of the Server. Generally, directory paths MUST be in the format of Unix-style path names with directory names like **([^/]|\\/)+** and full paths like **(/([^/]|\\\\|\\/)+)+/?**. The trailing slash of a path name SHALL be optional.

The level of recursion indicates an upper bound for the depth until which the contents of subdirectories of the given path should be traversed. The value of 0xFF should be considered as infinity, i.e. the Server may respond with a full depth-first traversal.

The Format Code indicates what and how the content of the file list response shall be formatted and transmitted by the Server. If the Server does not support the given Format Code it will respond with an Unknown Format Code Error as per section 5.6.8, containing a list of supported Format Codes. Format Code 0x00 is reserved and MUST NOT be supported by the Server. It can thus be used to request supported Format Codes by triggering an Error Report Message. These may be used to re-request some supported format.

5.7.2. File List Response

If the Server receives a File List Request as defined in section 5.7.1, it must decide whether this request is valid and MUST respond either with a File List Response Message as per section 6.2.6 or an Error Report Message as per section 6.2.4.

If the request is valid, the Server should respond with information of the files and subdirectories in the path requested in the file list request TLV, with the level of recursion and the format of the response dependent on the Format Code as given by the File List Request.

The level of recursion indicates the depth until which the contents of subdirectories should be traversed. The Server MUST at least list the directory given by the path and MAY choose to not go as deep as the requested level of recursion, but it MUST NOT do a deeper recursion than the one requested. The value 0xFF should be handled special and trigger a potentially unlimited traversal.

If the request is invalid, an Error Report Message containing an error TLV should be sent.

If the path is invalid, the Server should respond with a File Not Found Error as described in section 5.6.1.

If the Server does not support the given Format Code, it must respond with an Unknown Format Code Error as per section 5.6.8. The Error Data shall contain a list of Format Codes supported by the Server. A Server MAY choose not to support any Format Code.

5.7.3. Specification of SOFT format for code 0x01

The file list information is transmitted via a file list response message containing a file list response TLV as per section 6.2.6.

The File List Request TLVs consist of a list of file or directory names (not paths). For each name there is information on its type (being file or directory), its ID, as per 6.3.2, and the ID of its parent, i.e. the ID of the directory a file or subdirectory is located in. Including the parent ID allows for sending names instead of paths as the hierarchy and paths can be deduced from them.

Directories and files MUST be listed in preorder Depth First Traversal, but there is no additional constraint on what directory must be listed first if there are multiple (implementation-specific).

The length of each file list response TLV should not exceed the default chunk size. Each file list response TLV should contain as much file and directory elements as possible, without exceeding the length (except for the last file list response TLV), according to the implementation specific ordering.

6. SOFT MESSAGING

The SOFT Protocol consists of the SOFT Transport and the SOFT Application. The SOFT Transport is defined in section 6.1. The SOFT Application is defined in section 6.2.

Where not otherwise noted or required integers use Little Endian.

6.1. SOFT Transport Layer

The SOFT Transport operates on top of the UDP Datagram format [8]. The SOFT Transport forms an application-agnostic abstraction on top of UDP which ensures reliable exchange of application-specific Objects between two Hosts. Through the SOFT Transport the SOFT Application can focus on application logic and does mostly not have to deal with undesirable characteristics of UDP.

6.1.1. SOFT Message Frame

Table 6.1 SOFT Message Frame Layout

Length	Name	Description
1 octet	Version number	0x01: SOFT Version 1.0 0x02: SOFT Version 2.0 0x03-0xff: Reserved
8 octets	Session ID	Session ID, initially set to 0, determined by Server during Handshake as per section 4.2
1 octet	Number of TLVs	Number of TLVs present in the message
m bytes	Payload	List of TLVs
4 bytes	Checksum	FNV Hash Checksum for the entire message

6.1.2. SOFT Host Information TLV

Table 6.2 Host Information TLV Format

Field	Length	Value	Description
Type	1 octet	0x50	Host Information TLV
Length	2 octets	Variable	Number of octets in ensuing field.
Value	n octets	Variable	Maximum Buffer size in Bytes as unsigned LEB128 number

	1 octet	Variable	Maximum number of out-of-order packets that can be handled. 0 denotes the Host does not support reordering
	1 octet	0x00: Default 0x01-0x0f: Reserved 0x10: Minimal 0x11: Maximal 0x12-0xff: Reserved	Frequency of Acknowledgement
	1 octet	0x00: Reserved 0x01: Linux 0x02: Windows 0x03: MacOS 0x04: FreeBSD 0x05: Android 0x06: iOS 0x07-0xff: Reserved	Operating System supported (for metadata decoding)
	1 octet	0x00: Reserved 0x01: SOFT 0x02-0xff: Reserved	Application ID
	1 octet	Variable	Application Version

6.1.3. SOFT Object Header TLV

Table 6.3 SOFT Object Header TLV Format

Field	Length	Value	Description
Type	1 octet	0x51	Object Header TLV.
Length	2 octets	Variable	Number of octets in ensuing field.
Value	8 octets	Variable	Object ID
	n octets	Variable	Number of chunks as unsigned LEB128
	1 bit	0: Ack Required 1: Ack Not Required	Denote ACK Request
	7 bits	Reserved	Reserved
	1 octet	Variable	Object Type
	1 octet	m	Number of Fields
	1 octet	Variable	Type of Field

	n octets	Variable	Length of Field as LEB128 in number of Chunks
	The above 2 fields are repeated for m-1 times		

6.1.4. SOFT Object Chunk TLV

Table 6.4 SOFT Object Chunk TLV Format

Field	Length	Value	Description
Type	1 octet	0x52	Object Chunk TLV
Length	2 octets	Variable	Number of octets in ensuing field.
Value	8 octets	Variable	Object ID
	n octets	Variable	Chunk ID as signed LEB128 number
	1 bit	0: Last Chunk 1: More Chunks to come	Denote last Chunk of a File
	1 bit	0: Ack Required 1: Ack Not Required	ACK Required
	3 bits	Reserved	Reserved
	11 bits	n	Size of the following Chunk
	1 octet	Variable	Number of TLVs in the Content
	n octets	Variable	Contents of Object

6.1.5. SOFT Object Skip TLV

Table 6.5 SOFT Object Skip TLV Format

Field	Length	Value	Description
Type	1 octet	0x53	Object Skip TLV
Length	2 octets	Variable	Number of octets in ensuing field.
Value	8 octets	Variable	Object ID
	n octets	Variable	Chunk ID as signed LEB128 number

6.1.6. SOFT ACK TLV

Table 6.6 SOFT ACK TLV Format

Field	Length	Value	Description
Type	1 octet	0x30	ACK TLV.
Length	2 octets	Variable	Number of octets in ensuing field.
Value	1 octet	m	Number of Objects present in the TLV
	8 octets	Variable	Object ID
	n octets	Variable	Last received Chunk ID of the Object as signed LEB128 number
	The above 2 fields are repeated for m-1 times		

6.1.7. SOFT Transport Error TLV

Table 6.7 SOFT Transport Error TLV Format

Field	Length	Value	Description
Type	1 octet	0x31	Error TLV.
Length	2 octets	Variable	Number of octets in ensuing field.
Value	1 octet	0x00: Reserved 0x01: Reserved 0x02: Reserved 0x03: Reserved 0x04: Checksum Error 0x05: Unsupported Version 0x06: Session Unknown 0x07: Reserved 0x08: Object Abort 0x09 – 0xFF: Reserved	Error Code
	0 or n octets	Variable	Error data per Table 6.8

Table 6.8 SOFT Transport Error Data Format

Error Code	Error Data Format
0x04	The Error Data field must be empty
0x06	

<table>
<tr><td>0x05</td><td><table><tr><th>Length</th><th>Field</th></tr><tr><td>1 octet</td><td>Maximum Supported Version</td></tr><tr><td>1 octet</td><td>Minimum Supported Version</td></tr></table></td></tr>
<tr><td>0x08</td><td><table><tr><th>Length</th><th>Field</th></tr><tr><td>1 octet</td><td>Number of Objects in the data (= m)</td></tr><tr><td>8 octets</td><td>Object ID</td></tr><tr><td colspan="2">The above field is repeated for m-1 times</td></tr></table></td></tr>
<tr><td>0x00 – 0x03</td><td rowspan="3">The error data field is ignored</td></tr>
<tr><td>0x07</td></tr>
<tr><td>0x09- 0xff</td></tr>
</table>

6.1.8. SOFT ACK Request TLV

Table 6.9 SOFT ACK Request TLV Format

Field	Length	Value	Description
Type	1 octet	0x32	ACK Request TLV.
Length	2 octets	Variable	Number of octets in ensuing field.
Value	1 octet	m	Number of objects present in the TLV
	8 octets	Variable	Object ID
	n octets	Variable	Chunk ID of the Object for which ACK is required as signed LEB128 number
	The above 2 fields are repeated for m-1 times		

6.2. SOFT Application Layer

The primary application of SOFT is the unidirectional transfer of static file data. Therefore, the SOFT application layer describes the decomposition of files, metadata and errors into objects which can be transported in application layer TLVs (object layer chunks).

This version of the Soft Application Layer uses the Application ID 0x01 and Application Version 0x01. These will be used to identify the SOFT Application as per section 6.1.2.

6.2.1. SOFT Object Types

This section defines the application layer message type for SOFT messages which will be incorporated as objects in the transport layer packets.

Table 6.10 Message Types

Message/Object type	Description
File Request message	A message to request Server for specified files or to resume a file transfer
File Response message	A message to send requested files to Client
Error Report Message	A message by Client or Server to send errors
File List Request message	A message by Client to request the list of files held by Server.
File List Response message	A message to send the list of files present in a directory

Table 6.11 List of Fields of an Object Type

Message/Object type	Value	Field Types
File Request message	0x10	amount: 0 or 1, value: 0x20, name: File Request amount: 0 or 1, value: 0x21, name: File Resume
File Response message	0x11	amount: 0 or 1, value: 0x22, name: File Metadata amount: 0 or 1, value: 0x23, name: File Content
Error Report Message	0x12	amount: 1, value: 0x24, name: Error
File List Request message	0x13	amount: 1, value: 0x25, name: File List Request
File List Response message	0x14	amount: 1, value: 0x26, name: File List Response

6.2.2. File Request Message

The following TLV(s) shall be included in this message.

- Zero or One File Request TLV (see Table 6.12)
- Zero or One File Resume Request TLV (see Table 6.13)

Table 6.12 File Request TLV format

Field	Length	Value	Description
Type	1 octet	0x20	File Request TLV.
Length	2 octets	Variable	Number of octets in ensuing field.
Value	1 octet	m	Number of files requested
	n octet	q	Length of File Path as unsigned LEB128 number
	q octets	String	File Path
	The above 2 fields are repeated for m-1 times		

Table 6.13 File Resume TLV format

Field	Length	Value	Description
Type	1 octet	0x21	File Resume TLV.
Length	2 octets	Variable	Number of octets in ensuing field.
Value	1 octet	m	Number of files present in the TLV
	8 octets	Variable	File ID
	n octets	Variable	Last received Chunk ID of the file segment as signed LEB128 number
	The above 2 fields are repeated for m-1 times		

6.2.3. File Response Message

The following TLV(s) shall be included in this message.

- Zero or One File Metadata TLV (see Table 6.15)
- Zero or One File Content TLV (see Table 6.14)

Table 6.14 File Content TLV format

Field	Length	Value	Description
Type	1 octet	0x23	File Content TLV.
Length	2 octets	n	Number of octets in ensuing field.
Value	n octets	Variable	Contents of File

Table 6.15 File Metadata TLV format

Field	Length	Value	Description
Type	1 octet	0x22	File Metadata TLV.
Length	2 octets	Variable	Number of octets in ensuing field.
Value	1 octet	m	Number of entries present in the TLV
	n octets	Variable	Metadata Entry per Table 6.16
	The above field is repeated for m-1 times		

Table 6.16 Metadata Entry Format

Field	Length	Value	Description
Type	1 octet	0x00: no data 0x01: Name of File as string 0x02: Path of file as string 0x03: Size of File in bytes as 64-bit integer 0x04: Number of Chunks as 64-bit integer 0x05: stat, metadata in the format of stat() in sys/stat.h (92 bytes) 0x06: stat64, metadata in the format of stat64() in sys/stat.h 0x07-0x0F: reserved 0x10: CRC hash (32 bit) 0x11: MD5 hash (128 bit) 0x12: SHA0 hash (160 bit) 0x13: SHA1 hash (160 bit) 0x14: SHA2 hash (224-512 bit) 0x15: SHA3 hash (224-512 bit) 0x16- 0xFF: reserved	Metadata Type
Length	1 octet	m	Number of octets in ensuing field.
Value	m octets	Variable	Metadata

Note: For Sha2 and Sha3 the length of the Metadata determines the length of the hash algorithm. The only defined values are:

- 28 octets: SHA(2|3)-224
- 32 octets: SHA(2|3)-256
- 48 octets: SHA(2|3)-384
- 64 octets: SHA(2|3)-512

6.2.4. Error Report Message

The following TLV(s) shall be included in this message.

- One Error TLV (see Table 6.17)

Table 6.17 Error TLV format

Field	Length	Value	Description
Type	1 octet	0x24	Error TLV.
Length	2 octets	Variable	Number of octets in ensuing field.
Value	1 octet	0x00: Reserved 0x01: File Not Found 0x02: File Changed 0x03: No Space Left on Disk 0x04: File Hash Error 0x05: File Abort 0x06: Invalid File Resume Request 0x07: Invalid Depth for list 0x08: Unknown Format Code 0x09 – 0xFF: Reserved	Error Code
	0 or n octets	Variable	Error data per Table 6.18

Table 6.18 Error Data format

Error Code	Error Data Format
0x01 0x02 0x04 0x05 0x06	Length: 1 octet — Field: Number of Files in the data (= m) Length: n octets — Field: Length of File Name in LEB128 format (= m) Length: m octets — Field: File Name The above field is repeated for m-1 times
0x07	Length: n octets — Field: Length of Path name in LEB128 format (q octets) Length: q octets — Field: Path name
0x08	Size: 1 octet — Field: Number of codes in the data (= m)

	1 octet | Format Code The above field is repeated for m-1 times
0x00	The error data field must be empty.
0x03	
0x09-0xff	

6.2.5. File List Request Message

The following TLV(s) shall be included in this message.

- One File List Request TLV (see Table 6.19)

Table 6.19 File List Request TLV format

Field	Length	Value	Description
Type	1 octet	0x25	File List Request TLV.
Length	2 octets	Variable	Number of octets in ensuing field.
Value	n octets	Variable	Length of Directory Path as unsigned LEB128 number
	n octets	String	Path of the directory
	1 octet	Variable	Level of Recursion (0x00 – No Recursion) (0xff – Full Recursion)
	1 octet	0x00: triggers error 0x01: defined in this document 0x02 – 0xFF: Reserved	Format Code

6.2.6. File List Response Message

The following TLV(s) shall be included in this message.

- Zero or One File List Response TLV (see Table 6.20)
- Zero or One Error TLV (see Table 6.17)

Table 6.20 File List Response TLV format

Field	Length	Value	Description
Type	1 octet	0x26	File List Response TLV.

Length	2 octets	Variable	Number of octets in ensuing field.
Value	1 octet	q	Number of entries present in the TLV
	1 octet	0x00 – File 0x01 - Directory	Type of the entity
	8 octets	Variable	Parent Directory ID
	n octets	m	Length of file or directory name in LEB128 format
	m octets	String	File or directory name
	8 octets	Variable	File or directory ID
	The above 5 fields are repeated q-1 times		

6.3. General Algorithms and Encodings

The following defines general algorithms and Encodings used throughout the specification.

6.3.1. String Encoding

All Strings should be UTF-8 encoded. The corresponding length should be the length of the UTF-8 String in octets.

6.3.2. Object IDs for files and directories

A File’s Object ID MUST be generated by hashing the full path of the file or directory.

The paths to be hashed MUST begin with './' and end without '/', e.g. './path/to/file' and './path/to/directory' would be correct paths, but not '/path/to/file' or './path/to/directory/'.

The hash-function to be used is SHAKE256 [7] with a result size of 64bit.

6.3.3. LEB128

LEB128 (Little Endian Base 128) is a scheme for encoding integers densely that exploits the assumption that most integers are small in magnitude. (This encoding is equally suitable whether the target machine architecture represents data in big-endian or little-endian order. It is "little-endian" only in the sense that it avoids using space to represent the "big" end of an unsigned integer, when the big end is all zeroes or sign extension bits).

Unsigned LEB128 numbers are encoded as follows:

1. Start at the low order end of an unsigned integer and chop it into 7-bit chunks.
2. Place each chunk into the low order 7 bits of a byte. Typically, several of the high order bytes will be zero; discard them.
3. Emit the remaining bytes in a stream, starting with the low order byte; set the high order bit on each byte except the last emitted byte. The high bit of zero on the last byte indicates to the decoder that it has encountered the last byte.

Signed LEB128 numbers are encoded analogously, discarding high order bytes that extend the sign bit of the integer.

The LEB128 encoding was adapted from "DWARF Debugging Information Format Specification Version 3.0" [9].

Table 6.21 Example Values for LEB128 Encoding

LEB128 Encoding	Signed decoding	Unsigned decoding
0x00	0	0x00
0x7f	-1	127
0x80 02	0x01 00	0x01 00
0xff 7f	-1	0x3f ff

6.3.4. FNV-Hash

FNV (Fowler/Noll/Vo) is a fast, non-cryptographic hash algorithm with good dispersion [3] [10].

The FNV-1a Hash algorithm can be easily described by the pseudocode in Figure 6.1.

Figure 6.1 FNV1a Pseudocode

```
hash = offset_basis
for each octet_of_data to be hashed
    hash = hash xor octet_of_data
    hash = hash * FNV_Prime
return hash
```

For the 32bit FNV1a used in SOFT Message the constants used are

- FNV_Prime = 2**24 + 2**8 + 0x93 = 16,777,619 = 0x01000193
- offset_basis = 2,166,136,261 = 0x811C9DC5

7. SECURITY CONSIDERATIONS

This proposal makes no changes to the usage guidelines and security of UDP. More information about UDP security concerns can be found in [11].

7.1. Confidentiality

In order to guarantee confidentiality in a file transfer protocol, SOFT messages sent by a Client or by the Server need to be encrypted. There are three possible methods to establish such a secure communication channel: symmetric key encryption, public-key encryption and a hybrid cryptosystem.

Symmetric encryption is superior to public-key encryption in terms of speed but inferior regarding complexity and therefore security. Additionally, the symmetric encryption key needs to be exchanged securely beforehand which proves to be difficult in a client-server architecture hence this type of encryption should be avoided in a file transfer protocol.

As stated above public-key encryption provides a suffice amount of security but the encryption and decryption of messages takes a lot of time. Since in the SOFT protocol a lot of messages are exchanged the speed of the decryption and encryption mechanism matters a lot therefore public-key encryption is sufficient but not optimal for the SOFT protocol.

A hybrid cryptosystem seems to be the optimal choice for the SOFT protocol. A symmetric key could be agreed upon during the handshake procedure using a key establishment algorithm which then can be used to create a secure communication channel to exchange SOFT messages. Thereby the speed of symmetric encryption and the security of public-key encryption can be fully utilized.

Furthermore, it should be clarified which SOFT messages need to be encrypted to ensure confidentiality in this file transfer protocol. It is obvious that SOFT messages containing file metadata and file content should be encrypted. If other message types like file request, file list request, etc. need to be encrypted is up to discussion.

7.2. Integrity

Integrity of the file contents sent by the server must be validated. This protocol proposes a simple way to validate the integrity by making use of the SHA3-512 [7] file hash which is an evenly-distributed unique non-reversible cryptographic hash function. This approach helps to detect corruption but does not detect changes to the file hash. If the attacker changes the file content and the corresponding file hash, the Client may still assume that the received file contents are valid.

The attacker, instead of modifying file data, can change the entire file and compute all together new hash and send to the receiver. This

integrity check application is useful only if the user is sure about the authenticity of file.

Hashing is not enough to guarantee file content integrity. One approach will be to make use of a random or predefined SALT string, which can be appended to the data before calculating the hash code. The salt must be a secret string that must be exchanged by the server and the client during handshake.

Using a dynamic salt generated with time seed and the previously agreed secret string, will make it harder for the attacker to modify the contents of the file without getting noticed by the Client.

7.3. Authenticity

Authenticity concerns two topics, authenticity of a Client accessing the Server and authenticity of a Server's responses.

The authenticity of a Server's response can be ensured by a simple digital signature. If a Client does not know the Public Key of a server, digital certificates can be used as in current web server authentication [12].

The Authenticity of a Client can be ensured similarly by giving the Client a Private Key to sign its requests. This requires the Server to know all Clients and their Public Key, or all Clients having some kind of digital certificate for their Public Key. The Server itself may be granter of theses certificates.

If asymmetric cryptography is not desired a password based key derivation function may be used [13]. For this it must be ensured that the initial exchange must be confidential and authentic.

A Challenge-Response authentication may also be used to authenticate Hosts. In such both Server and Client send a cryptographic challenge based on a shared secret. This avoids replay attacks as challenges are unique every time [14].

To avoid replay attacks the Server shall require the Client to prove its authenticity by solving a unique challenge. The challenge consists of a Client giving a particular authenticated response based on the challenge's seed. To avoid man in the middle attacks, a Client may only respond to challenges that have been authenticated by the Server.

Once authenticity of Hosts has been established, authenticity can be further guaranteed via the symmetric encryption channel established in section 7.1.

7.4. Availability

Attacks on resource reservation on the server (and client) may be possible. For many application commands the SOFT protocol expects the

server to maintain per-client (per-session) state, enabling denial-of-service attacks on the server.

For the File list response, the server must calculate the whole file list before transmission. A high requested recursion depth could worsen this. The server may have treated the recursion depth as a hint rather than a must, and abort file list recursion if the depth exceeds buffer size limits.

A single client may start a large number N of file transfers under a large number M of allocated session IDs. Clients can easily DOS servers this way since the server would be asked to remember N*M transmission states. It is therefore necessary to place limits on the number of simultaneous client connections and the total number of file transfer states that may be stored by the server.

7.5. Access Control and Accountability

Care has also to be taken by server implementation with regards to ACCESS CONTROL. Clients can access file contents and metadata via the regular file retrieval functionality and information about what files and directories are available at the server via the file list feature. As there are no additional mechanisms for authentication, all clients will have access to all data. SOFT must not be used for sharing files that require some kind of controlled access, if there are no other mechanisms in place. Server implementations can limit the scope of files and directories which can be accessible. They must make sure that paths sent via the file (list) requests refer to valid data that they want to be shared with all clients. Special care must be taken if a server chooses to accept paths including UNIX path segments like '..' as these may be used to circumvent simple access control. Mechanisms like root jailing may be used.

There is no special consideration with regards to ACCOUNTABILITY (as there is no authentication).

7.6. Privacy

Security goals like authenticity of users and access control have been reflected in this chapter, but there is also another perspective on it: A user may want to download files anonymously for example when they wish to protect their identity.

Therefore, if privacy is of importance, a group-based authentication method shall be used [15].

Therefore, first of all, authentication methods MUST be optional in future SOFT specifications.

Secondly the SOFT protocol gives the client application quite some options to reveal information to the server. This includes operating system and preferences towards transmission like frequency of acknowledgement, maximum number of out-of-order packets and maximum

buffer size (see Host Information TLV per Table 6.2). When those parameters are naively computed by each client, there may be a risk of them being sufficiently unique within the set of clients connecting to a single server. Hence a server might try to use that information to identify a client which could be a violation of privacy requirements of that user. This could even work over an extended period because the less the general situation of the client changes, the less those parameters change as well.

Client implementations might therefore want to mitigate this problem by trying to fill those fields with common and partially or fully randomized data.

8. FUTURE WORK

Currently the protocol implements a congestion control algorithm like TCP Reno [16]. Since this is not the most efficient algorithm to adapt, it is possible for future work to implement a more sophisticated and efficient algorithm (e.g. TCP Cubic [17]).

The current protocol implementation is not able to consume packets that arrive out of order and rather drops them. A more sophisticated implementation may be proposed in future versions to utilize those out of order packets instead of discarding them.

In future versions of the protocol the Client may be able to request the transfer of a folder since the existing implementation only supports the transfer of one or multiple files. It is inefficient to request for multiple files rather than requesting a folder which contains all those files.

9. References

[1] S. Bradner, "RFC 2119: Key words for use in RFCs to Indicate Requirement Levels.," 1997. [Online]. Available: https://tools.ietf.org/html/rfc2119.

[2] B. Leiba, "RFC 8174: Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words.," 2017. [Online]. Available: https://tools.ietf.org/html/rfc8174.

[3] G. Fowler, L. C. Noll and K.-P. Vo, " The FNV Non-Cryptographic Hash Algorithm," 2014. [Online]. Available: https://tools.ietf.org/html/draft-eastlake-fnv-03.

[4] J. Mogul and S. Deering, "RFC 1191: Path MTU discovery," 1990. [Online]. Available: https://tools.ietf.org/html/rfc1191.

[5] J. Postel, "RFC 791: The Internet Protocol," 1981. [Online]. Available: https://tools.ietf.org/html/rfc791.

[6] C. Hornig, "RFC 894: Standard for the transmission of IP datagrams over Ethernet networks," 1984. [Online]. Available: https://tools.ietf.org/html/rfc894.

[7] M. J. Dworkin, "SHA-3 standard: Permutation-based hash and extendable-output functions. No. Federal Inf. Process. Stds.(NIST FIPS)-202," 2015. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[8] J. Postel, "RFC 768: User Datagram Protocol," 1980. [Online]. Available: https://tools.ietf.org/html/rfc768.

[9] Free Standards Group, "DWARF Debugging Information Format Specification Version 3.0," 2005. [Online]. Available: http://dwarfstd.org/doc/Dwarf3.pdf.

[10] "xxHash," [Online]. Available: https://cyan4973.github.io/xxHash/.

[11] L. Eggert, G. Fairhurst and G. Shepherd, "RFC 8085: UDP Usage Guidelines," 2017. [Online]. Available: https://tools.ietf.org/html/rfc8085.

[12] "The network authentication protocol.," 2004. [Online]. Available: https://web.mit.edu/kerberos/.

[13] B. Kaliski, "Pkcs# 5: Password-based cryptography specification version 2.0.," 2000. [Online]. Available: https://tools.ietf.org/html/rfc2898.

[14] R. Blom, "Challenge-response user authentication". U.S. Patent 7,194,765, 2007.

[15] M. Manulis, N. Fleischhacker and F. K. F. P. B. Günther, "Group signatures: Authentication with privacy," Bundesamt fur Sicherheit in der Informationstechnik, Bonn, Germany, 2012. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/GruPA/GruPA.pdf.

[16] M. Allman, V. Paxson and E. Blanton, "RFC 5681: TCP Congestion Control," 2009. [Online]. Available: https://tools.ietf.org/html/rfc5681.

[17] I. e. a. Rhee and R. Scheffenegger, "RFC 8312: CUBIC for Fast Long-Distance Networks," 2018. [Online]. Available: https://tools.ietf.org/html/rfc8312.