FAST Protocol Specification

Hossam Mabed

April 2020

File Asynchronous Secure Transfer
Design Document

1 Introduction

This document describes the File Asynchronous Secure Transfer (FAST) protocol. FAST describes a secure file transfer application, where multiple clients and a file server can server communicate together securely.

The user interacts with the client using a CLI or a GUI and the client speaks to the server. The user can use the client to execute file commands such as uploading or downloading files, removing files, creating directories, etc. The user can access and modify only their files.

FAST consists of two separate protocols: FAST Handshake, and FAST Tunnel. FAST Handshake is essentially a key exchange protocol, which authenticates the client to the server, using a secret password, then establishes a temporary shared session key between the client and the server. Then, the server and the client use this key to communicate commands over a secure channel. The client sends commands to the server, and the server responds with an acknowledgement and potential output. The FAST Tunnel protocol describes this secure channel channel, which takes control after the establishing of a shared session key using the Handshake.

The following file commands are supported:

1. MKD – creating a folder on the server
2. RMD – removing a folder from the server
3. GWD – asking for the name of the current folder (working directory) on the server
4. CWD – changing the current folder on the server
5. LST – listing the content of a folder on the server
6. UPL – uploading a file to the server
7. DNL – downloading a file from the server
8. RMF – removing a file from a folder on the server

These commands, their arguments, their formats, and their expected responses are described later in more detail, along with the specification of the protocol.

2 Attacker Models and Trust Assumptions

We have two different attacker models, which we design the protocol to protect against. They are described here.

2.1 Outsiders

Outsiders are non-participants in the protocol communications.

2.1.1 Goals

The goals for the outsiders can be any or all of the following:

- Modify contents of the files sent by the client to the server
- Obtain the password of a client
- Impersonate a client
- Delete or modify files of a client
- Create new files in the name of a client
- Read contents of files belonging to a client

2.1.2 Capabilites

Outsiders have all of the following abilities:

- Eavesdrop all protocol communications
- Intercept/block all protocol communications
- Modify all protocol communications

However, outsiders cannot break any cryptographic primitives.

2.2 Legitimate clients

Legitimate clients, who are legitimate participants in the protocol, can also be seen as attackers, if they have goals that are not they are not allowed under normal protocol rules.

2.2.1 Goals

The goals for legitimate clients, who behave as attackers, can be any or all of the following:

- Impersonate a different user
  - Delete or modify files of another client
  - Create new files in the name of another client
- Obtain the password of another client
- Read contents of files belonging to another client

2.2.2 Capabilites

Legitimate users have all the capabilities of outsiders, except they also have the following capabilities:

- Participate in protocol as legitimate user
- Login into the file server with their own password
- Execute protocol commands related to their own files
- Have their own ID
- Know the ID of any other legitimate client

However, legitimate users also cannot break any cryptographic primitives.

2.3 Assumptions

The following trust assumptions are made:

1. The file server is trustworthy
2. The file server conducts file handling properly, and does not abuse its power by not conducting the instructions it is instructed to.
3. All clients have access to the public key of the server, obtained through a secure out-of-band channel, and all clients can trust that this public key belongs to the server

4. The user can trust the client they're interacting with
5. Clients **cannot** trust each other (other clients can be attackers)

Furthermore, the following assumptions, unrelated to trust, are also made:

1. The server has a way of maintaining file structure and storage that is outside the scope of this protocol
2. Once a client is authenticated, the server has a way of allowing the client to interact only with the files and folders belonging to that client. This is outside the scope of this protocol.
3. All clients have IDs that uniquely identify them, established beforehand and outside the scope of this protocol
4. Clients have established a password with the server, outside the scope of this protocol
5. The server and all clients know the ID of every client
6. The server maintains some data structure that maps every client ID to a password hash, a session key (which can be null), and a session message counter (which can be null)

3 Security Requirements

Based on these attacker models and these trust assumptions, the following security requirements are made for the protocol:

3.1 Confidentiality

1. No one should be able to read the contents of a client's file except the client
2. No one should be able to know the password of a client except that client

3.2 Integrity

1. Contents of a file or instruction sent by a client should not be modified in any means

3.3 Authentication

1. Only the user should be able to modify their files
2. Only the user should be able to create files under their name
3. Only the user should be able to delete their files
4. File origin should be verifiable to the server
5. Acknowledgement origin should be verifiable to the client

3.4 Replay Protection

Replayed instructions should be detected and discarded by the server

3.5 Instruction Freshness

Instructions sent to the server should be fresh

3.6 Instruction order

Instructions should be executed by the server in the order they are sent by the client

3.7 Non-repudiation of instruction origin

4 Protocol Specifications

The following sections describe in detail the two protocols.

4.1 FAST Handshake Protocol

4.1.1 Overview

The purpose of the FAST Handshake protocol is to authenticate client to the server. The client is able to login to the server with password and establish session key. Establishing a session key takes 1-RTT length.

All clients are assumed to know the server's public key through a secure out-of-band channel, and we assume that clients trust the key is in fact the public key of the server.

Client starts the protocol by sending a message to the server with the ID of the client, the client's password, the proposed key for the session, and a timestamp. The entire message is encrypted with the server's public key.

The server decrypts the received message with its private key. It accepts the message if the timestamp is recent enough and the password is correct. If the session is accepted, the server sends a "session_start" message, encrypted with the newly established session key to the client. If the session is not accepted, the server drops the message and doesn't do any further actions.

4.1.2 Assumptions

We assume the client can trust the server (the client is okay with the server knowing the client's password). Furthermore, the client obtains an encryption-capable public RSA key of the server, through some secure out-of-band channel. The client is sure that the key is indeed the key of the server. Furthermore, the clocks are synchronized between all clients and the server. The client already knows its unique ID that uniquely identifies it to the server. Setting up a new client with a new ID is outside the scope of this protocol

The server stores some data structure with all client information, which maps client IDs to the hash of the password and the symmetric key of the currently active session. If there is no active session, that field is NULL. Similarly, the IDs are also mapped to a current session message sequencing number, which is NULL if there is no active session.

4.1.3 Message Sequence Diagram

The FAST Handshake protocol is described at a high level by the following message sequence diagram:

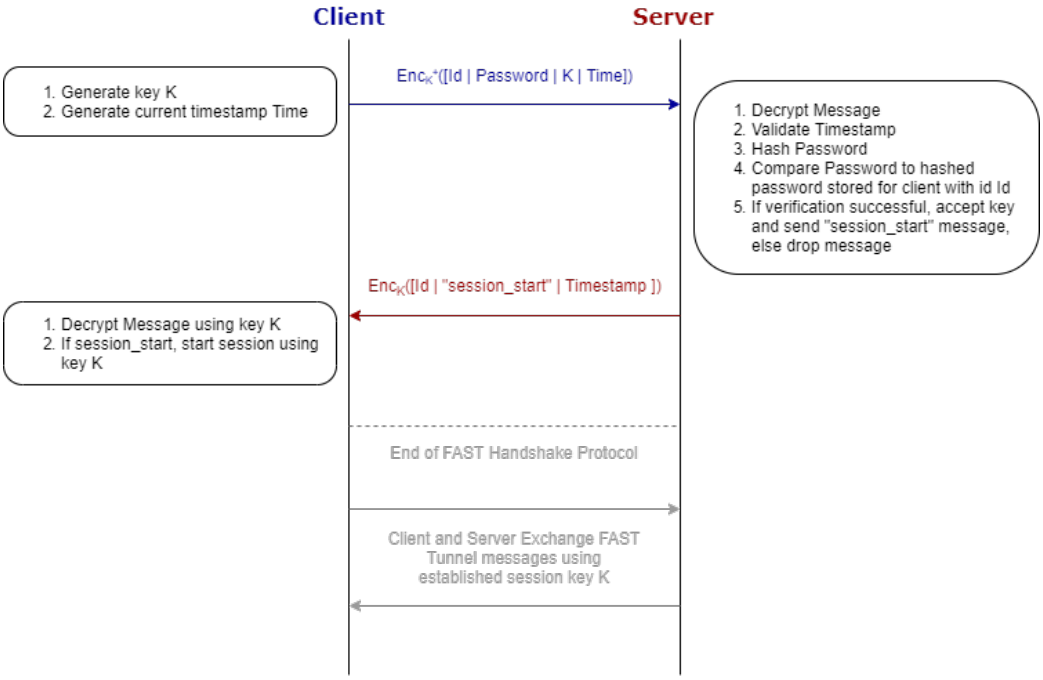

Figure 1: Handshake protocol Message Sequence Diagram

4.1.4 Message Format and Message Processing Rules

We describe the message formats and processing rules of this protocol in three phases

4.1.4.1 Phase 1: Initiation Message (Client)

The Client initiates the protocol. The client sends the initiation message, which includes the client's ID, the client's Password, the client's freshly generated key, and the client's current timestamp, all encrypted with the server's public key. The client does this in the following way:

1. The client starts the message with its ID (32 bytes)

- The ID is a unique 32 byte value that uniquely identifies the client to the server
- The ID of a client is not a secret value

2. The client appends to the message its Password (32 bytes)
    - The password is a secret value that only the client knows, which allows the server to authenticate the client
    - The password must be at least 8 characters long, and must contain at least 1 lowercase letter, 1 uppercase letter, and 1 special character.
    - Passwords cannot be longer than 32 characters
    - Passwords use ASCII encoding. Passwords are not allowed to have non-ASCII characters
    - All passwords, even if they are shorter than 32 characters, are sent in 32 bytes. If the password is shorter than 32 characters, the password is padded to make it fit into 32 bytes
    - Passwords are padded using PKCS#7 padding
3. The client generates a new fresh key and appends it to the message (32 bytes)
    - The key generated must be a pseudorandom stream of 256 bits that can be used as a symmetric key for AES in GCM mode.
    - The key must be generated with an appropriate cryptographic pseudo-random number generator
4. The client appends its timestamp (32 bytes)
    - The client uses a timestamp format that is pre-agreed upon with the server (this is outside the scope of this protocol), but must be 32 bytes
5. The client encrypts the message
    - The entire message (all 128 bytes) is preformatted for RSA encryption using PKCS#1 v2 (OAEP)

- The client encrypts the preformatted message with the public key of the server, using RSA encryption

6. The client sends the encrypted message to the server

4.1.4.2 Phase 2: Verification, Session Creation, and Acknowledgement (Server)

In this stage, the server decrypts the received message from the client. It verifies the timestamp and the password. If the client is authenticated, the server creates a new session with the client using the provided key, and sends an acknowledgement message to the client, encrypted with the new session key. Otherwise, the server drops the message and doesn't send anything to the server. In detail, the server does the following:

1. The server decrypts the message
    - The server decrypts the message received from the client, using its private key
2. The server validates the timestamp
    - The server checks that the timestamp is recent enough. Precisely, the server checks that the timestamp was from less than 1 minute ago. If the timestamp is fresh, verification resumes. Else, the message is dropped and the server goes back to waiting for another initiation message.
3. The server validates the password
    - The server first removes the padding from the password
    - The servers ensures that the password is valid
        - Shorter than 32 bytes and longer than 8 bytes
        - Contains at least 1 character, 1 lowercase letter, and 1 uppercase letter
        - Contains no non-ASCII characters
    - If the password is valid, the password is hashed
    - Passwords are hashed using SHA-224

- The server looks up the password hash stored corresponding to the user ID provided, to the password hash just computed
- The hashed password is compared to the stored hash
- If the hashes match, the password is verified, and the client is authenticated.

4. If the validation or authentication fails, drop the message

    - If, at any stage in the previously mentioned steps of verification, validation, and authentication, that stage fails, the message is dropped, no message is sent to the client, and the server goes back to waiting for initiation messages.

5. If the verification and authentication succeeds, start the new session

    - If all the previously mentioned steps succeed, the server accepts the new session with that client
    - The server stores the new key K provided by the client in the current session field corresponding to the client ID

6. If the session is accepted, send an acknowledgement message

    - If the session is accepted, the following message is constructed:
    - The ID of the client (provided by the client earlier) (32 bytes)
    - The special message "session_start" in ASCII (13 bytes)
    - The current server's timestamp (32 bytes)
    - The full message (77 bytes) is encrypted
        - Message is encrypted using symmetric key encryption with AES in GCM mode, using the newly established session key K as the symmetric key
        - The Nonce used for this message is 1, signifying the first message sent using this session key
        - The MAC length is 16 bytes
        - There is no header associated data used. Instead, the entire message is encrypted
    - The generated ciphertext and tag are sent to the client

4.1.4.3 Phase 3: Session Start (Client)

Once the client receives the "session_start" message, it also accepts the new session, and stores the generated session key K in its state. It then starts communicating with the server using symmetric encryption with the key K. When the client receives the message:

1. Client decrypts and verifies the message
    - If decryption and verification is unsuccessful, the message is dropped, and a new initiation message is sent with a new key, then client moves back to waiting for the acknowledgement message.
2. Client verifies message content
    - The client verifies that the Id sent is its own ID, that the message is "session_start", and that the timestamp is within 1 minute ago.
    - If this fails, the message is dropped, and we move back to waiting for the acknowledgement message.
3. Client accepts new session
    - If all of the above is successful, the client accepts the new session, and starts sending commands to the server using the FAST Tunnel protocol, using the established symmetric key K.

If 5 minutes have passed since sending the initiation message, and the client has not received the session_start message yet, the client drops the current session, and attempts to initiate a new session using a new freshly generated key.

This is the end of the FAST Handshake protocol, having established the session and the symmetric session keys. All further communication between the client and server uses the FAST Tunnel protocol.

4.1.5 Cryptographic Primitives Used

4.1.5.1 Asymmetric Key Encryption

Used in sending the initiation message from client to server. Uses RSA public key encryption, with messages preformatted using PKCS#1 v2 (OAEP).

Public Key-Private Key pairs for the server are established beforehand and the public key communicated to the clients using a secure out-of-band channel, that is outside the scope of this protocol.

4.1.5.2 Padding

Used to pad the password to be 32 bytes long. PKCS#7 padding is used

4.1.5.3 Hashing

Used to hash passwords. SHA-224 is used

4.1.5.4 Symmetric key encryption

Used to send the session_start message. AES in GCM mode is used, with 256-bit key, and the number 1 as the nonce (only 1 message is sent with this encryption). The MAC Length used is 16 bytes. No header data is used as associated data, since the entire message is encrypted.

4.1.6 Protocol State Machine

The following diagram summarizes the state machine and possible state changes for the server:

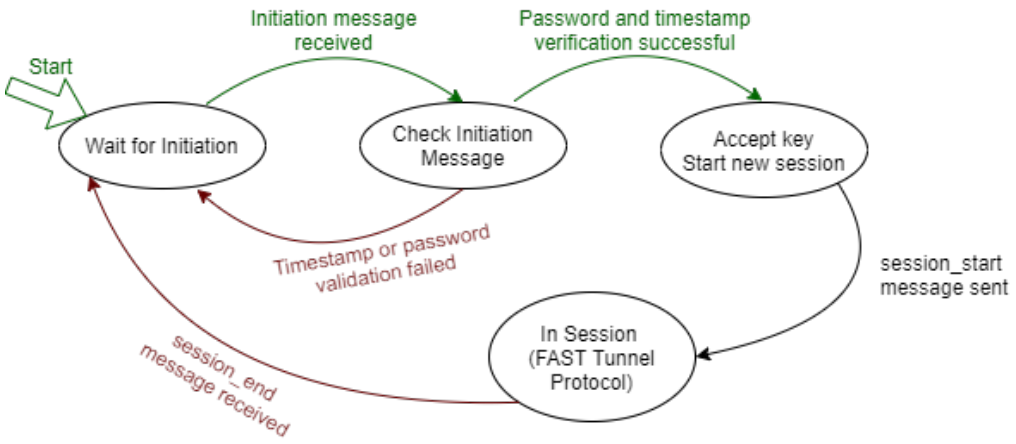

Figure 2: Handshake protocol server state diagram

Meanwhile, the following state machine diagram illustrates the states and state changes in the client

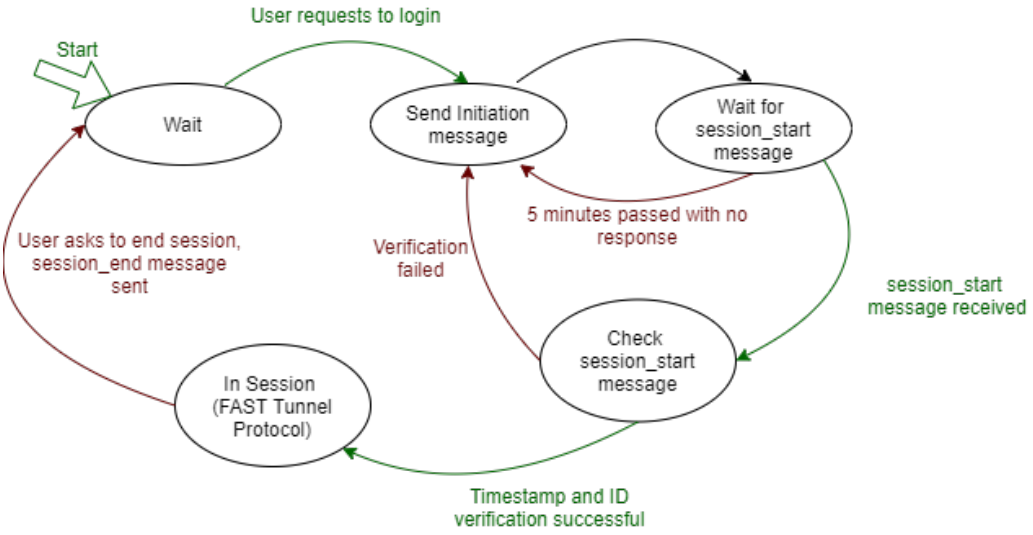

Figure 3: Handshake protocol client state diagram

4.2 FAST Tunnel Protocol

4.2.1 Overview

The second protocol that we describe in this specification is the FAST Tunnel protocol. The FAST Tunnel protocol provides a secure channel that allows communication of commands between the client and the server. The FAST Tunnel protocol takes control as soon as the server switches to the "in session" state, as described earlier, and for the client as soon as it receives and validates the session_start message. The FAST Tunnel is also responsible for the session_end message, as illustrated in the state diagram above.

4.2.2 Assumptions

This protocol is initiated only after a successful handshake of the FAST Handshake protocol, having established a shared fresh session key. This protocol assume a shared session key is already established. Furthermore, The server is trustworthy. Lastly, both the client and the server, upon the

start of the session, maintain a sequence number. It starts at zero when the session is first established.

4.2.3 Message Sequence Diagram

The following message sequence diagram describes at a high level the messages communicated using this protocol:

Figure 4: Tunnel protocol message sequence diagram

4.2.4 Message Format

All messages in this protocol take the following format (not to scale):

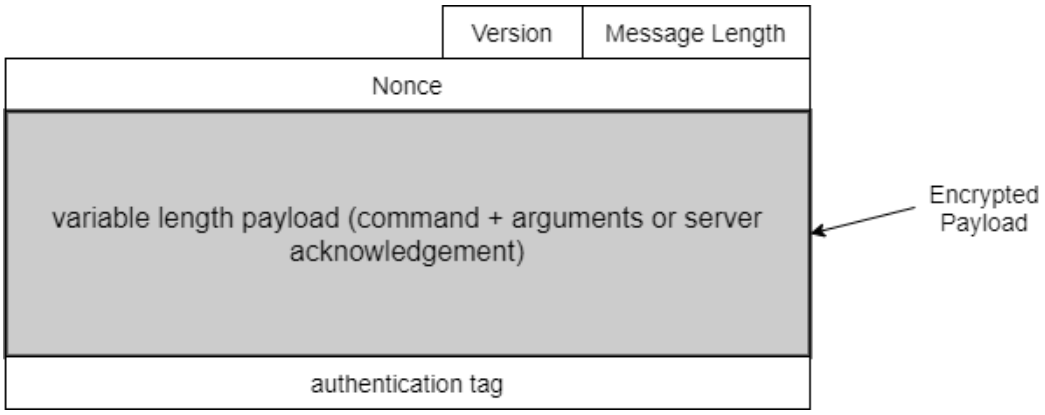

Figure 5: Tunnel protocol message format (not to scale)

where:

4.2.4.1 Version (2 bytes)

Specifies the version number of the FAST Tunnel Protocol used. Each byte specifies a digit in the version number. For example, if the first byte is 1 and the second byte is 0, the version is interpreted as version 1.0

4.2.4.2 Message Length (4 bytes)

Specifies the length of the full message. This length includes the whole message, including version number and message length bytes, as well as the authentication tag. This number is interpreted as integer using big endian encoding.

4.2.4.3 Nonce (16 bytes)

These 16 bytes will be interpreted as an integer stored in big endian, and this number will be used as a nonce (N). N will serve two purposes: it acts as a message sequencing number to verify message order and prevent message replay, and will be used as the Nonce for encrypting the message and decrypting it on the receiving end, using AES encryption in GCM mode.

N must be unique for each message in the session, and must be a strictly increasing counter. Messages will not be accepted if they have a lower Nonce than the most recently received Nonce.

4.2.4.4 Message Payload (variable length)

This payload has two options. If the client is sending the message, it will be a command payload. If the server is sending the message, it will be an acknowledgment payload (with potential output). The exact format of the payload is described later. Regardless of how the payload is constructed, in both cases, the entire payload is encrypted using the session key K with AES encryption in GCM mode and 16 byte MAC.

4.2.4.5 Authentication tag (16 bytes)

The 16 byte authentication tag returned from the AES encryption in GCM mode. Used to verify the authenticity and integrity of the message. The authenticated encryption scheme uses the header date (version, message length, and nonce) as the associated data, and the payload as the plaintext data to be encrypted.

4.2.4.6 Payload formatting

The command payload is constructed as follows:

1. Argument number (1 bytes)
    - Specifies how many arguments the command has. The command itself counts as an argument. Therefore, the minimum number here is 1
2. Command (3 bytes)
    - Specifies which command the client wishes to execute. Commands are three characters encoded in ASCII, and must be from the following
        - MKD – creating a folder on the server
        - RMD – removing a folder from the server

- GWD – asking for the name of the current folder (working directory) on the server
- CWD – changing the current folder on the server
- LST – listing the content of a folder on the server
- UPL – uploading a file to the server
- DNL – downloading a file from the server
- RMF – removing a file from a folder on the server
- END - a special command to end the current session. All sessions must end with this command

3. Arguments (variable length)

    - If more than one argument (the command) is provided, each extra argument must begin with 8 bytes specifying the size of the argument in bytes, followed by the argument itself. The argument size is interpreted as an integer encoded in big endian.
    - The following section specifies how many arguments each command expects

The acknowledgement message is constructed in a similar way except:

1. Argument number signifies how many outputs the server is sending to the client. All commands, even if they don't expect an output, receive the special output "acknowledged" (encoded in ASCII). This special output may be prepended to further output if appropriate. Therefore, the argument number as well must be at least 1

2. The command is replaced with the special acknowledgement message "acknowledged", encoded in ASCII (12 bytes)

3. Any further outputs given by the server must, as before, begin with 8 bytes specifying the size of the output

4. The outputs that the server will send will depend on the commands executed, as described in the following section

4.2.4.7 Command Arguments and Outputs

1. MKD
    - Arguments
        - 1 extra argument follows the command
        - File name: max 120 bytes
        - Must be ASCII-encoded and cannot contain non-ASCII characters
    - Outputs
        - If successful, 1 extra output: "success" (in ASCII)
        - If failure, 2 extra outputs: "failure" in ASCII & Error message in ASCII
2. RMD
    - Arguments
        - 1 extra argument: file name or path
    - Outputs
        - If successful, 1 extra output: "success" (in ASCII)
        - If failure, 2 extra outputs: "failure" in ASCII & Error message in ASCII
3. GWD
    - No extra arguments
    - outputs
        - 1 extra output: the name of the current working directory (in ASCII)
4. CWD
    - Arguments
        - 1 extra argument
        - Directory path: max 120 bytes
        - Must be ASCII-encoded and cannot contain non-ASCII characters

- Outputs
  - 1 extra output: the name of the new current working directory after executing the command (ASCII)

5. LST
   - No extra arguments
   - Outputs
     - As many outputs as there are files/folders in the current directory. Each output will specify the name of the file/folder in ASCII

6. UPL
   - Arguments
     - 2 extra argument
     - File name: as before
     - File: treated as binary blob, binary posted as-is to the current working directory under the given name
   - Outputs
     - If successful, 1 extra output: "success" (in ASCII)
     - If failure, 2 extra outputs: "failure" in ASCII & Error message in ASCII

7. DNL
   - Arguments
     - 1 extra argument: File Name or File Path:
   - Outputs
     - 2 outputs: the file name in ASCII, and binary payload of the file (treated as a binary blob)

8. RMF
   - Arguments
     - 1 extra argument: file name or path

- Outputs
  - If successful, 1 extra output: “success” (in ASCII)
  - If failure, 2 extra outputs: “failure” in ASCII & Error message in ASCII

9. END
   - No extra arguments
   - No extra outputs

4.2.5 Message Processing Rules

4.2.5.1 Command Messages (Client to Server)

Any interaction in this protocol is initiated by the client to execute a specific command. The Client initiates the protocol by sending a command message to the server, in the format described above.

4.2.5.2 Verification and Command Execution (Server)

Once a command message is received by the server, the server authenticates the message using the tag provided. Then, the server validates the message sequence number (nonce). If it’s successful, the server decrypts the payload and executes the command. Once completed, the server sends an acknowledgement message and any output to the client, in the format described above. In detail, it works in the following way:

1. The server validates the message sequence number
   - In the same table that the server maps client IDs to password hashes and session keys, the server also stores a last received message sequence number (sqn), which is refreshed after each session, and starts at zero
   - The server checks the nonce provided to the stored sqn
     - If it’s higher, sqn is replaced with Nonce
     - If it’s lower, the message is dropped and no further action is taken
2. The server authenticates the message

- The server checks the authentication tag provided by the message. If authentication fails, the message is dropped and no further action is taken.

3. The server decrypts the payload

   - The server uses the stored current session key, as well as the provided nonce, to decrypt the payload usinG AES in GCM mode.

4. Parse commands

   - The server parses the payload following the command structure described above to extract the command requested and any additional arguments for the commands

5. Execute commands

   - The server executes the command requested with the requested arguments using its existing file storage (this is outside the scope of this protocol)
   - If the command is invalid, arguments are invalid, or command execution failed, the server sends an acknowledgement message with two extra outputs, specifying the appropriate error message, as specified above
   - The server manages proper access rights based on the current session's user. This is outside the scope of this protocol.
   - Any files uploaded by the users are handled as-is as binary blobs. The client may wish to encrypt or compress the files beforehand, but that is not handled by the server or by this protocol.

6. If execution successful, send appropriate acknowledgement message.

4.2.5.3 Interpreting Acknowledgement Messages

Upon reception of an acknowledgment message, the client verifies and decrypts the message, and extracts the output. The client also accepts acknowledgement numbers only if the received message nonce is strictly higher than the stored sqn.

To end the FAST Tunnel protocol, the client must send the special command "END" to the server, which prompts it to close the session. Upon

reception of an acknowledgement message from an "END" command, the client logs out the user and deletes session keys.

4.2.6 Cryptographic Primitives Used

An authenticated encryption scheme is used to both authenticate and encrypt message payloads. AES in GCM mode is used, with 256-bit key, and the nonces sent in the header of the message. MAC Length used is 16 bytes. The header data (associated data used) is the version number, message length and nonce.

4.2.7 Protocol State Machine

The following diagram summarizes the state machine and possible state changes for the server:

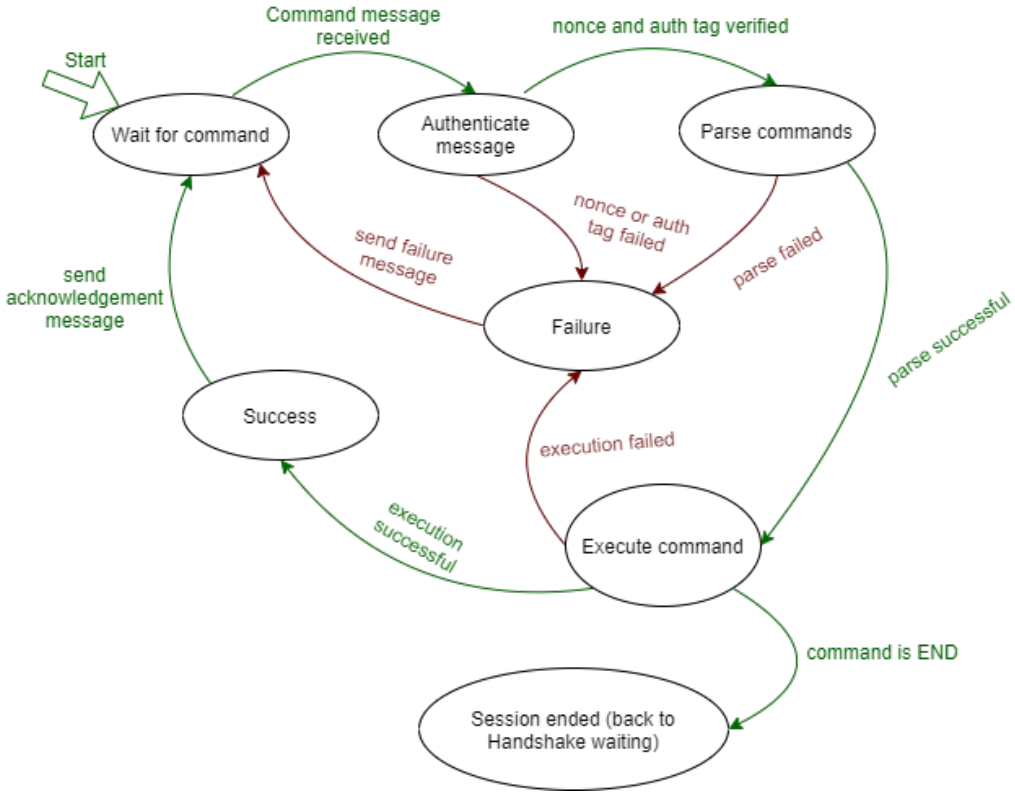

Figure 6: Tunnel protocol server state diagram

Meanwhile, the following state machine diagram illustrates the states and state changes in the client

Figure 7: Tunnel protocol client state diagram

5 Summary

In the Handshake protocol, the confidentiality of the password is provided by the public key cryptography. Because the key is made available in a secure out-of-band channel, we can be sure of its security. Authenticity is provided by the password. Furthermore, replay protection and key freshness is guaranteed by the timestamp, since messages are only accepted if the timestamp falls within a certain short window.

In the second step, the authenticity and integrity of the start_session

message sent from the server to the client is guaranteed by the public key cryptography, since the server is guaranteed to be the only party, other than the client, to have access to the symmetric key. This is because the key was encrypted with the server's public key, and therefore only the server can decrypt it and use it to encrypt an acknowledgement message. Furthermore, replay protection is similarly provided with the timestamp.

Therefore, the Handshake protocol satisfies all its securiry requirements and is able to establish a session key that only the server and the client have access to, to be used for secure communication over the secure channel in the Tunnel protocol.

In the Tunnel protocol, confidentiality of the messages, as well authentication and integrity protection, are provided by the securely established session key, since we use an authenticated encryption scheme, which implements both encryption and message integrity protection.

Furthermore, replay protection of message and message sequencing is handled by the nonce scheme, which is strictly increasing for each message and acts as a message sequence number. Non-repudiation of message origin is provided for both parties by the session key, since only the client and server have access to that shared key.

Therefore, the FAST Handshake and Tunnel protocols satisfy all our security requirements.