Table of Contents

FIDO 2.0: Client To Authenticator Protocol

FIDO Alliance Review Draft 18 October 2016

This version:
https://fidoalliance.org/specs/fido-undefined-v2.0-rd-20161018/fido-inter-device-protocol-v2.0-rd-20161018.html

Editors:
Rolf Lindemann, Nok Nok Labs
Vijay Bharadwaj, Microsoft
Alexei Czeskis, Google
Michael B. Jones, Microsoft
Contributors:
Mirko J. Ploch, SurePassID
Matthieu Antoine, Gemalto
Jakob Ehrensvärd, Yubico

Abstract

This specification describes an application layer protocol for communication between an external authenticator and another client/platform. This protocol can be run over a variety of transport protocols using different physical media. This specification defines requirements for such transport protocols, but does not specify the details of how such transport layer connections should be set up.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://www.fidoalliance.org/specifications/.

This document was published by the FIDO Alliance as a Review Draft. This document is intended to become a FIDO Alliance Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

Table of Contents

1. Overview

This section is non-normative.

This protocol is intended to be used in scenarios where a user interacts with a relying party (a website or native app) on some platform (e.g., a PC) which prompts the user to interact with an external authenticator (e.g., a smartphone).

In order to provide evidence of user interaction, an external authenticator implementing this protocol is expected to have a mechanism to obtain a user gesture. Possible examples of user gestures include: as a consent button, password, a PIN, a biometric or a combination of these.

Prior to executing this protocol, the client/platform (referred to as *host* hereafter) and external authenticator (referred to as *authenticator* hereafter) must establish a confidential and mutually authenticated data transport channel. This specification does not specify the details of how such a channel is established, nor how transport layer security must be achieved.

2. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words must, must not, required, should, should not, recommended, may, and optional in this specification are to be interpreted as described in [RFC2119].

3. Protocol Structure

This section is non-normative.

This protocol is specified in three parts:

- **Authenticator API**: At this level of abstraction, each authenticator operation is defined similarly to an API call - it accepts input parameters and returns either an output or error code. Note that this API level is conceptual and does not represent actual APIs. The actual APIs will be provided by each implementing platform.
- **Message Encoding**: In order to invoke a method in the authenticator API, the host must construct and encode a request and send it to the authenticator over the chosen transport protocol. The authenticator will then process the request and return an encoded response.
- **Transport-specific Binding**: Requests and responses are conveyed to external authenticators over specific transports (e.g., USB, NFC, Bluetooth). For each transport technology, message bindings are specified for this protocol.

This document specifies all three of the above pieces for external FIDO 2.0 authenticators.

4. Authenticator API

Each operation in the authenticator API can be performed independently of the others, and all operations are asynchronous. The authenticator may enforce a limit on outstanding operations to limit resource usage - in this case, the authenticator is expected to return a busy status and the host is expected to retry the operation later. Additionally, this protocol does not enforce in-order or reliable delivery of requests and responses; if these properties are desired, they must be provided by the underlying transport protocol or implemented at a higher layer by applications.

Note that this API level is conceptual and does not represent actual APIs. The actual APIs will be provided by each implementing platform.

The authenticator API has the following methods and data structures.

4.1 authenticatorMakeCredential

This method is invoked by the host to request generation of a new credential in the authenticator. It takes the following input parameters:

Parameter name	Data type	Required?	Definition
rpId	String	Required	Identity of the relying party. See [FIDOPlatformApiReqs]
clientDataHash	Byte Array	Required	Hash of the ClientData contextual binding specified by host. See [FIDOSignatureFormat].
accountInformation	AccountInfo	Required	Friendly UI details to be used by the authenticator when displaying the credential to the user for selection and usage authorization. See [FIDOWebApi] for AccountInfo type specification.
cryptoParameters	sequence of FIDOCredentialParameters	Required	A sequence of FIDOCredentialParameters structures, as specified in [FIDOWebApi].
blacklist	Sequence of Credentials	Optional	A sequence of Credential structures, as specified in [FIDOWebApi]. The authenticator is requested to return an error (see Section TBD) if it recognizes any of them.
extensions	FIDOExtensions	Optional	Parameters to influence authenticator operation. These parameters might be authenticator specific.

When such a request is received, the authenticator performs the following procedure:

1. If the blacklist parameter is present and contains a credential ID that is present on this authenticator, terminate this procedure and return error code TDB.
2. If the cryptoParameters parameter does not contain a valid AlgorithmIdentifier structure that is supported by the authenticator, terminate this procedure and return error code TBD.
3. Optionally, if the extensions parameter is present, process any extensions that this authenticator supports.
4. If the authenticator has a display, show the contents of the accountInformation and rpId parameters to the user. Request permission to create a credential. If the user declines permission, return an error code.
5. Generate a new cryptographic key pair for the algorithm specified.
6. Associate the newly-created key pair with the rpId.
7. Generate an attestation statement for the newly-created key using clientDataHash.

On success, the authenticator must return the following structure in its response:

Member name	Data type	Required?	Definition
credential	Credential	Required	A credential type and a byte string that must be used by the host to identify this key for future operations. From the perspective of the host, this is simply an opaque identifier for the key.
publicKey	ByteArray	Required	The DER encoding of the SubjectPublicKeyInfo structure from [RFC5280] (Section 4.1.2.7) generated for the new credential.
rawAttestation	Byte Array	Optional	The raw attestation statement. Its structure is opaque to the Platform/Client. See [FIDOKeyAttestation] for structure details.

4.2 authenticatorGetAssertion

This method is used by a host to request cryptographic proof of user authentication as well as user consent to a given transaction, using a previously generated credential that is stored by the authenticator. provide. It takes the following input parameters:

Parameter name	Data type	Required?	Definition
rpId	String	Required	Identity of the relying party. See [FIDOPlatformApiReqs]
clientDataHash	Byte Array	Required	Hash of the ClientData contextual binding specified by host. See [FIDOSignatureFormat].
whitelist	Sequence of Credentials	Optional	A sequence of Credential structures, as specified in [FIDOWebApi]. The authenticator is requested to only generate a FIDOAssertion using one of them.
extensions	FIDOExtensions	Optional	Parameters to influence authenticator operation. These parameters might be authenticator specific.

When such a request is received, the authenticator performs the following procedure:

1. Locate all FIDO 2.0 credentials that are eligible for retrieval under the specified criteria:
    - If a whitelist is present and is non-empty, locate all mentioned credentials which are present on this authenticator. Discard any for credential which is not bound to the specified rpId.
    - If a whitelist is not present, locate all credentials which are present on this authenticator which are bound to the specified rpId.
2. Optionally, if the extensions parameter is present, process any extensions that this authenticator supports.
3. Display all these credentials to the user, using their friendly name. If the authenticator stored an AccountInfo object during the creation of this credential, display the stored account information. Also, display the rpId of the requester (specified in the request) and ask the user to select a credential. If the user declines to select a credential or takes too long (as determined by the authenticator), terminate this procedure and

return an error code.
4. If the user selects a credential, sign the clientDataHash with it, using the structure specified in [FIDOSignatureFormat].

On success, the authenticator must return the following structure in its response:

Member name	Data type	Required?	Definition
credential	Credential	Optional	Credential whose private key was used to generate the assertion. May be omitted if the whitelist has exactly one Credential.
authenticatorData	Byte Array	Required	Authenticator's raw contextual binding, as specified in [FIDOSignatureFormat].
signature	Byte Array	Required	Raw signature from the authenticator, as specified in [FIDOSignatureFormat].

4.3 authenticatorCancel

Using this method, the host can request the authenticator to cancel all ongoing operations are return to a ready state. It takes no input parameters and returns success or failure.

4.4 authenticatorGetInfo

Using this method, the host can request that the authenticator report a list of all supported protocol versions (currently, "FIDO_2_0" is the only supported version) and extensions. This method takes no inputs.

On success, the authenticator must return:

Member name	Data type	Required?	Definition
versions	Sequence of strings	Required	List of supported versions.
extensions	Sequence of strings	Optional	List of supported extensions.
aaguid	String	Optional	The claimed AAGUID.

5. Message encoding

Many transports (e.g., Bluetooth Smart) are bandwidth-constrained, and serialization formats such as JSON are too heavy-weight for such environments. For this reason, all encoding is done using the concise binary encoding CBOR [RFC7049].

Messages from the host to authenticator are called "commands" and messages from authenticator to host are called "replies". All values are big endian encoded.

5.1 Commands

All commands are structured as:

Name	Length	Required?	Definition
Command Value	1 byte	Required	The value of the command to execute
Command Parameters	variable	Optional	CBOR [RFC7049] encoded set of parameters. Some commands have parameters, while others do not (see below)

The assigned values for commands and their descriptions are:

Command Name	Command Value	Has parameters?
authenticatorMakeCredential	0x01	yes
authenticatorGetAssertion	0x02	yes
authenticatorCancel	0x03	no
authenticatorGetInfo	0x04	no

Command parameters are encoded using a CBOR map (CBOR major type 5). The CBOR map must be encoded using the definite length variant.

Some commands have optional parameters. Therefore, the length of the parameter map for these commands may vary. For example, authenticatorMakeCredential may have 4, 5, or 6 parameters, while authenticatorGetAssertion may have 2, 3, or 4 parameters.

All command parameters are CBOR encoded following the*JSON to CBOR* conversion procedures as per the CBOR specification [RFC7049]. Specifically, parameters that are represented as DOM objects in the *Authenticator API* layers (formally defined in the Web API [FIDOWebApi]) are converted first to JSON and subsequently to CBOR.

EXAMPLE 1
An AccountInfo DOM object defined as follows:

```
var userAccountInformation = {
      rpDisplayName: "ACME",
      displayName: "John P. Smith",
      name: "johnpsmith@example.com",
      id: "1098237235409872",
      imageUri: "https://pics.acme.com/00/p/aBjjjpqPb.png"
    };
```

would be CBOR encoded as follows:

```
a5                                              # map(5)
    6d                                          # text(13)
        7270446973706c61794e616d65              # "rpDisplayName"
    64                                          # text(4)
        41636d65                                # "Acme"
    6b                                          # text(11)
        646973706c61794e616d65                  # "displayName"
    6d                                          # text(13)
        4a6f686e20502e20536d697468              # "John P. Smith"
    64                                          # text(4)
        6e616d65                                # "name"
    76                                          # text(22)
        6a6f686e70736d697468406578616d70        # "johnpsmith@example.com"
        6c652e636f6d                            # ...
    62                                          # text(2)
        6964                                    # "id"
    70                                          # text(16)
        31303938323337323335343039383732        # "1098237235409872"
    68                                          # text(8)
        696d61676555524c                        # "imageURL"
    7828                                        # text(40)
        68747470733a2f2f706963732e61636d        # "https://pics.acme.com/00/p/aBjjjpqPb.png"
        652e636f6d2f30302f702f61426a6a6a        # ...
        707150622e706e67                        # ...
```

EXAMPLE 2
A DOM object that is a sequence of FIDOCredentialParameters defined as follows:

```
        var cryptoParams = [
            {
              type: "FIDO",
              algorithm: "ES256",
            },
            {
              type: "FIDO",
              algorithm: "RS256",
            }
          ];
```

would be CBOR encoded as:

```
    82                                              # array(2)
        a2                                          # map(2)
            64                                      # text(4)
                74797065                            # "type"
            6a                                      # text(10)
                53636f70656443726564                # "ScopedCred"
            69                                      # text(9)
                616c676f726974686d                  # "algorithm"
            65                                      # text(5)
                4553323536                          # "ES256"
        a2                                          # map(2)
            64                                      # text(4)
                74797065                            # "type"
            6a                                      # text(10)
                53636f70656443726564                # "ScopedCred"
            69                                      # text(9)
                616c676f726974686d                  # "algorithm"
            65                                      # text(5)
                5253323536                          # "RS256"
```

For each command that contains parameters, the parameter map keys and value types are specified below:

Command	Parameter Name	Key	Value type
authenticatorMakeCredential	rpId	0x01	UTF-8 encoded text string (CBOR major type 3).
	clientDataHash	0x02	byte string (CBOR major type 2).
	accountInformation	0x03	CBOR definite length map (CBOR major type 5).
	cryptoParameters	0x04	CBOR definite length array (CBOR major type 4) of CBOR definite length maps (CBOR major type 5).
	blacklist	0x05	CBOR definite length array (CBOR major type 4) of CBOR definite length maps (CBOR major type 5).
	extensions	0x06	CBOR definite length map (CBOR major type 5).
authenticatorGetAssertion	rpId	0x01	UTF-8 encoded text string (CBOR major type 3).
	clientDataHash	0x02	byte string (CBOR major type 2).
	whitelist	0x03	CBOR definite length array (CBOR major type 4) of CBOR definite length maps (CBOR major type 5).
	extensions	0x04	CBOR definite length map (CBOR major type 5).

EXAMPLE 3
The following is a complete encoding example of the `authenticatorMakeCredential` command (using same account and crypto parameters as above) and the corresponding `authenticatorMakeCredential_Response` response:

```
    01                                              # authenticatorMakeCredential command
    a4                                              # map(4)
        01                                          # unsigned(1) -- rpId
        68                                          # text(8)
            61636d652e636f6d                        # "acme.com"
        02                                          # unsigned(2) -- clientDataHash
        2420                                        # byte string(32)
            5a81483d96b0bc15ad19af7f5a662e14        # "5a81483d96b0bc15ad19af7f5a662e14b275729fbc05579b18513e7f550016b1"
            b275729fbc05579b18513e7f550016b1        # ...
        03                                          # unsigned(3) -- accountInformation
        a5                                          # map(5)
            6d                                      # text(13)
                7270446973706c61794e616d65          # "rpDisplayName"
            64                                      # text(4)
                41636d65                            # "Acme"
            6b                                      # text(11)
                646973706c61794e616d65              # "displayName"
            6d                                      # text(13)
                4a6f686e20502e20536d697468          # "John P. Smith"
            64                                      # text(4)
                6e616d65                            # "name"
            76                                      # text(22)
                6a6f686e70736d697468406578616d70    # "johnpsmith@example.com"
                6c652e636f6d                        # ...
            62                                      # text(2)
                6964                                # "id"
            70                                      # text(16)
                31303938323337323335343039383732    # "1098237235409872"
            68                                      # text(8)
                696d61676555524c                    # "imageURL"
            7828                                    # text(40)
                68747470733a2f2f706963732e61636d    # "https://pics.acme.com/00/p/aBjjjpqPb.png"
                652e636f6d2f30302f702f61426a6a6a    # ...
                707150622e706e67                    # ...
        04                                          # unsigned(4) -- cryptoParameters
        82                                          # array(2)
            a2                                      # map(2)
                64                                  # text(4)
                    74797065                        # "type"
                6a                                  # text(10)
                    53636f70656443726564            # "ScopedCred"
                69                                  # text(9)
                    616c676f726974686d              # "algorithm"
                65                                  # text(5)
                    4553323536                      # "ES256"
            a2                                      # map(2)
                64                                  # text(4)
                    74797065                        # "type"
                6a                                  # text(10)
                    53636f70656443726564            # "ScopedCred"
                69                                  # text(9)
                    616c676f726974686d              # "algorithm"
                65                                  # text(5)
                    5253323536                      # "RS256"
```

authenticatorMakeCredential_Response response:

```
00                                              # status = success
a3                                              # map(3)
    01                                          # unsigned(1) -- credential
    a2                                          # map(2)
        64                                      # text(4)
            74797065                            # "type"
        6a                                      # text(10)
            53636f70656443726564                # "ScopedCred"
        62                                      # text(2)
            6964                                # "id"
        7824                                    # text(36)
            38444437343134442d454534332d3437    # "8DD7414D-EE43-474C-A05D-FDDB828B663B"
            34432d413035442d4644444238323842    # ...
            36363342                            # ...
    02                                          # unsigned(2) -- credentialPublicKey
    a5                                          # map(5)
        63                                      # text(3)
            6b7479                              # "kty"
        63                                      # text(3)
            525341                              # "RSA"
        63                                      # text(3)
            616c67                              # "alg"
        65                                      # text(5)
            5253323536                          # "RS256"
        63                                      # text(3)
            657874                              # "ext"
        20                                      # false
        61                                      # text(1)
            6e                                  # "n"
        790156                                  # text(342)
            6c4d5234586f78526959356b70746748    # "lMR4XoxRiY5kptgHhh1XLKnezHC2EWPIImlHS-iUMSKVH32WWUKfEoY5Al_exPtcVuUfcNGtMoys
            686831584c4b6e657a48433245575049    # ...
            496d6c48532d69554d534b5648333257    # ...
            57554b66456f5935416c5f6578507463    # ...
            56755566634e47744d6f7973414e3635    # ...
            505a7a634d4b5861512d326138416562    # ...
            4b776538715147426334795930456b50    # ...
            393953676238307241663153377 32d4a    # ...
            524e56744e5452623471725856434d78    # ...
            5a48753375626a73646579624d492d66    # ...
            464b7a596739495636444506f744a7978    # ...
            314f704e536469625377574b44546335    # ...
            597a47666f4f473376412d316165396f    # ...
            464f68355a6f6c68486e7235556b6f64    # ...
            464b5561784f4f486650724142304d56    # ...
            5435593555374766f5f5a5f317146444f    # ...
            4c794f57646878787a6c326174334b39    # ...
            74795143386b674a434e4b597371372d    # ...
            45467a7641395139305043365378 4741    # ...
            54516f49434b6e3276434e4d42715648    # ...
            4c6c547964426d50372d384d6f4d7865    # ...
            664d32373777                        # ...
        03                                      # text(1)
            65                                  # "e"
        64                                      # text(4)
            41514142                            # "AQAB"
    03                                          # unsigned(3) -- rawAttestation
    22                                          # null
```

EXAMPLE 4

The following is a complete encoding example of the authenticatorGetAssertion command and the corresponding authenticatorGetAssertion_Response response:

```
02                                              # authenticatorGetAssertion command
a3                                              # map(3)
    01                                          # unsigned(1) -- rpId
    68                                          # text(8)
        61636d652e636f6d                        # "acme.com"
    02                                          # unsigned(2) -- clientDataHash
    7840                                        # text(64) TODO: byte string
        65336230633434323938666331633134        # "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855"
        39616662663463383939366662393234        # ...
        32376165343165343634396239333463        # ...
        61343935393931623738353262383535        # ...
    03                                          # unsigned(3) -- whitelist
    a2                                          # map(2)
        64                                      # text(4)
            74797065                            # "type"
        6a                                      # text(10)
            53636f70656443726564                # "ScopedCred"
        62                                      # text(2)
            6964                                # "id"
        7824                                    # text(36)
            38444437343134442d454534332d3437    # "8DD7414D-EE43-474C-A05D-FDDB828B663B"
            34432d413035442d4644444238323842    # ...
            36363342                            # ...
```

authenticatorGetAssertion_Response response:

```
00                                              # status = success
a4                                              # map(4)
    01                                          # unsigned(1) -- credential
    a2                                          # map(2)
        64                                      # text(4)
            74797065                            # "type"
        6a                                      # text(10)
            53636f70656443726564                # "ScopedCred"
        62                                      # text(2)
            6964                                # "id"
        7824                                    # text(36)
            38444437343134442d454534332d3437    # "8DD7414D-EE43-474C-A05D-FDDB828B663B"
            34432d413035442d4644444238323842    # ...
```

```
           36363342                         # ...
   02                                       # unsigned(2) -- clientData
   7830                                     # text(48) TODO: byte string
       6577304b43534a6a6147467362475675     # "ew0KCSJjaGFsbGVuZ2UiIDogImFiYzEyM2RlZjQ1NiINCn0A"
       5a32556949446f67496d4669597a4579     # ...
       4d32526c5a6a51314e69494e436e3041     # ...
   03                                       # unsigned(3) -- authenticatorData
   67                                       # text(7)
       41514141414141                       # "AQAAAAA"
   04                                       # unsigned(4) -- signature
   790156                                   # text(342) TODO: byte string
       6732326e683157772d715a4179737569     # "g22nh1Ww-qZAysuizkugZGmEisax3dtoUNzIl2LWOSARzeZxm_-nQoHfKyo8b8_XnxXwuLlW8RXB
       7a6b75675a476d4569736178336474 6f     # ...
       554e7a496c324c574f5341527a655a78     # ...
       6d5f2d6e516f48664b796f3862385f58     # ...
       6e785877754c6c57385258424c414e33     # ...
       38443356325042756750526c6f567a45     # ...
       31676e34566c37526f3132344771 5079     # ...
       55524e6c6c764e6b443345416c363462     # ...
       48504b2d4556494f6d49387a6b30514b     # ...
       5f5a6f7166414b595f52664d4c4e4f62     # ...
       536e3437485f6864412d595a5547456b     # ...
       57746379556743363548397866684657     # ...
       4f5164672d725f704859355f54786764     # ...
       534e523869746b426232785a474b6167     # ...
       466e475574646d4f5352524f56774b39     # ...
       41616c4a77734a443157346c46355f34     # ...
       4a66756d736231594a36795177725068     # ...
       4a75594e434365564858496168584455     # ...
       4b54645474575173304d546a376b4769     # ...
       316a2d5f6c6b4e706c3772456e535634     # ...
       777177384b35534563484d2d6d454259     # ...
       582d664d4477                         # ...
```

5.2 Responses

All responses are structured as:

Name	Length	Required?	Definition
Status	1 byte	Required	The status of the response. 0x00 means success; all other values are errors. See table TBD for error values.
Response Data	variable	Optional	CBOR encoded set of values.

Response data is encoded using a CBOR map (CBOR major type 5). The CBOR map must be encoded using the definite length variant.

For each response message, the map keys and value types are specified below:

Response Message	Member Name	Key	Value type
authenticatorMakeCredential_Response	credential	0x01	CBOR definite length map (CBOR major type 5).
	credentialPublicKey	0x02	byte string (CBOR major type 2).
	rawAttestation	0x03	byte string (CBOR major type 2).
authenticatorGetAssertion_Response	credential	0x01	CBOR definite length map (CBOR major type 5).
	authenticatorData	0x02	byte string (CBOR major type 2).
	signature	0x03	byte string (CBOR major type 2).
authenticatorGetInfo_Response	versions	0x01	CBOR definite length array (CBOR major type 4) of UTF-8 encoded strings (CBOR major type 3).
	extensions	0x02	CBOR definite length array (CBOR major type 4) of UTF-8 encoded strings (CBOR major type 3).
	aaguid	0x03	CBOR UTF-8 encoded string (CBOR major type 3).

6. Transport-specific Bindings

6.1 USB

6.1.1 Design rationale

CTAP messages are framed for USB transport using the HID (Human Interface Device) protocol. We henceforth refer to the protocol as CTAPHID. The CTAPHID protocol is designed with the following design objectives in mind

- Driver-less installation on all major host platforms
- Multi-application support with concurrent application access without the need for serialization and centralized dispatching.
- Fixed latency response and low protocol overhead
- Scalable method for CTAPHID device discovery

Since HID data is sent as interrupt packets and multiple applications may access the HID stack at once, a non-trivial level of complexity has to be added to handle this.

6.1.2 Protocol structure and data framing

The CTAP protocol is designed to be concurrent and state-less in such a way that each performed function is not dependent on previous actions. However, there has to be some form of "atomicity" that varies between the characteristics of the underlying transport protocol, which for the CTAPHID protocol introduces the following terminology:

- Transaction
- Message
- Packet

A **transaction** is the highest level of aggregated functionality, which in turn consists of a request, followed by a response message. Once a request has been initiated, the transaction has to be entirely completed before a second transaction can take place and a response is never sent without a previous request.

Request and response **messages** are in turn divided into individual fragments, known as **packets**. The packet is the smallest form of protocol data unit, which in the case of CTAPHID are mapped into HID reports.

6.1.3 Concurrency and channels

Additional logic and overhead is required to allow a CTAPHID device to deal with multiple "clients", i.e. multiple applications accessing the single resource through the HID stack. Each client communicates with a CTAPHID device through a logical **channel**, where each application uses a unique 32-bit **channel identifier** for routing and arbitration purposes.

A channel identifier is allocated by the FIDO authenticator device to ensure its system-wide uniqueness. The actual algorithm for generation of channel identifiers is vendor specific and not defined by this specification.

Channel ID 0 is reserved and `0xffffffff` is reserved for broadcast commands, i.e. at the time of channel allocation.

6.1.4 Message and packet structure

Packets are one of two types, **initialization packets** and **continuation packets**. As the name suggests, the first packet sent in a message is an initialization packet, which also becomes the start of a transaction. If the entire message does not fit into one packet (including the CTAPHID protocol overhead), one or more continuation packets have to be sent in strict ascending order to complete the message transfer.

A message sent from a host to a device is known as a**request** and a message sent from a device back to the host is known as a **response**. A request always triggers a response and response messages are never sent ad-hoc, i.e. without a prior request message.

The request and response messages have an identical structure. A transaction is started with the initialization packet of the request message and ends with the last packet of the response message.

Packets are always fixed size (defined by the endpoint and HID report descriptors) and although all bytes may not be needed in a particular packet, the full size always has to be sent. Unused bytes should be set to zero.

An initialization packet is defined as

Offset	Length	Mnemonic	Description
0	4	CID	Channel identifier
4	1	CMD	Command identifier (bit 7 always set)
5	1	BCNTH	High part of payload length
6	1	BCNTL	Low part of payload length
7	(s - 7)	DATA	Payload data (s is equal to the fixed packet size)

The command byte has always the highest bit set to distinguish it from a continuation packet, which is described below.

A continuation packet is defined as

Offset	Length	Mnemonic	Description
0	4	CID	Channel identifier
4	1	SEQ	Packet sequence 0x00..0x7f (bit 7 always cleared)
5	(s - 5)	DATA	Payload data (s is equal to the fixed packet size)

With this approach, a message with a payload less or equal to (s - 7) may be sent as one packet. A larger message is then divided into one or more continuation packets, starting with sequence number 0, which then increments by one to a maximum of 127.

With a packet size of 64 bytes (max for full-speed devices), this means that the maximum message payload length is 64 - 7 + 128 * (64 - 5) = 7609 bytes.

6.1.5 Arbitration

In order to handle multiple channels and clients concurrency, the CTAPHID protocol has to maintain certain internal states, block conflicting requests and maintain protocol integrity. The protocol relies on each client application (channel) behaves politely, i.e. does not actively act to destroy for other channels. With this said, a malign or malfunctioning application can cause issues for other channels. Expected errors and potentially stalling applications should however be handled properly.

6.1.5.1 Transaction atomicity, idle and busy states.

A transaction always consists of three stages:

1. A message is sent from the host to the device
2. The device processes the message
3. A response is sent back from the device to the host

The protocol is built on the assumption that a plurality of concurrent applications may try ad-hoc to perform transactions at any time, with each transaction being atomic, i.e. it cannot be interrupted by another application once started.

The application channel that manages to get through the first initialization packet when the device is in idle state will keep the device locked for other channels until the last packet of the response message has been received. The device then returns to idle state, ready to perform another transaction for the same or a different channel. Between two transactions, no state is maintained in the device and a host application must assume that any other process may execute other transactions at any time.

If an application tries to access the device from a different channel while the device is busy with a transaction, that request will immediately fail with a busy-error message sent to the requesting channel.

6.1.5.2 Transaction timeout

A transaction has to be completed within a specified period of time to prevent a stalling application to cause the device to be completely locked out for access by other applications. If for example an application sends an initialization packet that signals that continuation packets will follow and that application crashes, the device will back out that pending channel request and return to an idle state.

6.1.5.3 Transaction abort and re-synchronization

If an application for any reason "gets lost", gets an unexpected response or error, it may at any time issue an abort-and-resynchronize command. If the device detects a SYNC command during a transaction that has the same channel id as the active transaction, the transaction is aborted (if possible) and all buffered data flushed (if any). The device then returns to idle state to become ready for a new transaction.

6.1.5.4 Packet sequencing

The device keeps track of packets arriving in correct and ascending order and that no expected packets are missing. The device will continue to assemble a message until all parts of it has been received or that the transaction times out. Spurious continuation packets appearing without a prior initialization packet will be ignored.

6.1.6 Channel locking

In order to deal with aggregated transactions that may not be interrupted, such as tunneling of vendor-specific commands, a channel lock command may be implemented. By sending a channel lock command, the device prevents other channels from communicating with the device until the channel lock has timed out or been explicitly unlocked by the application.

This feature is optional and has not to be considered by general CTAP HID applications.

6.1.7 Protocol version and compatibility

The CTAPHID protocol is designed to be extensible, yet maintaining backwards compatibility to the extent it is applicable. This means that a CTAPHID host shall support any version of a device with the command set available in that particular version.

6.1.8 HID device implementation

This description assumes knowledge of the USB and HID specifications and is intended to provide the basics for implementing a CTAPHID device. There are several ways to implement USB devices and reviewing these different methods is beyond the scope of this document. This specification targets the interface part, where a device is regarded as either a single or multiple interface (composite) device.

The description further assumes (but is not limited to) a full-speed USB device (12 Mbit/s). Although not excluded per se, USB low-speed devices are not practical to use given the 8-byte report size limitation together with the protocol overhead.

6.1.8.1 Interface and endpoint descriptors

The device implements two endpoints (except the control endpoint 0), one for IN and one for OUT transfers. The packet size is vendor defined, but the reference implementation assumes a full-speed device with two 64-byte endpoints.

Interface Descriptor

Mnemonic	Value	Description
bNumEndpoints	2	One IN and one OUT endpoint
bInterfaceClass	0x03	HID
bInterfaceSubClass	0x00	No interface subclass
bInterfaceProtocol	0x00	No interface protocol

Endpoint 1 descriptor

Mnemonic	Value	Description
bmAttributes	0x03	Interrupt transfer
bEndpointAdresss	0x01	1, OUT
bMaxPacketSize	64	64-byte packet max
bInterval	5	Poll every 5 millisecond

Endpoint 2 descriptor

Mnemonic	Value	Description
bmAttributes	0x03	Interrupt transfer
bEndpointAdresss	0x81	1, IN
bMaxPacketSize	64	64-byte packet max
bInterval	5	Poll every 5 millisecond

The actual endpoint order, intervals, endpoint numbers and endpoint packet size may be defined freely by the vendor and the host application is responsible for querying these values and handle these accordingly. For the sake of clarity, the values listed above are used in the following examples.

6.1.8.2 HID report descriptor and device discovery

A HID report descriptor is required for all HID devices, even though the reports and their interpretation (scope, range, etc.) makes very little sense from an operating system perspective. The CTAPHID just provides two "raw" reports, which basically map directly to the IN and OUT endpoints. However, the HID report descriptor has an important purpose in CTAPHID, as it is used for device discovery.

For the sake of clarity, a bit of high-level C-style abstraction is provided

EXAMPLE 5

```
// HID report descriptor

const uint8_t HID_ReportDescriptor[] = {
  HID_UsagePage ( FIDO_USAGE_PAGE ),
  HID_Usage ( FIDO_USAGE_CTAPHID ),
  HID_Collection ( HID_Application ),
  HID_Usage ( FIDO_USAGE_DATA_IN ),
  HID_LogicalMin ( 0 ),
  HID_LogicalMaxS ( 0xff ),
  HID_ReportSize ( 8 ),
  HID_ReportCount ( HID_INPUT_REPORT_BYTES ),
  HID_Input ( HID_Data | HID_Absolute | HID_Variable ),
  HID_Usage ( FIDO_USAGE_DATA_OUT ),
  HID_LogicalMin ( 0 ),
  HID_LogicalMaxS ( 0xff ),
  HID_ReportSize ( 8 ),
  HID_ReportCount ( HID_OUTPUT_REPORT_BYTES ),
  HID_Output ( HID_Data | HID_Absolute | HID_Variable ),
HID_EndCollection
};
```

A unique **Usage Page** is defined for the FIDO alliance and under this realm, a CTAPHID **Usage** is defined as well. During CTAPHID device discovery, all HID devices present in the system are examined and devices that match this usage pages and usage are then considered to be CTAPHID devices.

The length values specified by the `HID_INPUT_REPORT_BYTES` and the `HID_OUTPUT_REPORT_BYTES` should typically match the respective endpoint sizes defined in the endpoint descriptors.

6.1.9 CTAPHID commands

The CTAPHID protocol implements the following commands.

6.1.9.1 Mandatory commands

The following list describes the minimum set of commands required by an CTAPHID device. Optional and vendor-specific commands may be implemented as described in respective sections of this document.

6.1.9.1.1 CTAPHID_MSG

This command sends an encapsulated CTAP message to the device. The semantics of the data message is defined in the CTAP/CBOR data encoding specification.

Request

CMD	CTAPHID_MSG
BCNT	1..(n + 1)
DATA	CTAP command byte
DATA + 1	n bytes of CBOR encoded data

Response at success

CMD	CTAPHID_MSG
BCNT	1..(n + 1)
DATA	CTAP status code
DATA + 1	n bytes of CBOR encoded data

6.1.9.1.2 CTAPHID_INIT

This command synchronizes a channel and optionally requests the device to allocate a unique 32-bit channel identifier (CID) that can be used by the requesting application during its lifetime. The requesting application generates a nonce that is used to match the response. When the response is received, the application compares the sent nonce with the received one. After a positive match, the application stores the received channel id and uses that for subsequent transactions.

To allocate a new channel, the requesting application shall use the broadcast channel CTAPHID_BROADCAST_CID. The device then responds the newly allocated channel in the response, using the broadcast channel.

Request

CMD	CTAPHID _INIT
BCNT	8
DATA	8-byte nonce

Response at success

CMD	CTAPHID _INIT
BCNT	17 (see note below)
DATA	8-byte nonce
DATA+8	4-byte channel ID
DATA+12	CTAPHID protocol version identifier
DATA+13	Major device version number
DATA+14	Minor device version number
DATA+15	Build device version number
DATA+16	Capabilities flags

The protocol version identifies the protocol version implemented by the device. An CTAPHID host shall accept a response size that is longer than the anticipated size to allow for future extensions of the protocol, yet maintaining backwards compatibility. Future versions will maintain the response structure to this current version, but additional fields may be added.

The meaning and interpretation of the version number is vendor defined.

The following device capabilities flags are defined. Unused values are reserved for future use and must be set to zero by device vendors.

CAPABILITY_WINK	Implements the WINK function

6.1.9.1.3 CTAPHID_PING

Sends a transaction to the device, which immediately echoes the same data back. This command is defined to be a uniform function for debugging, latency and performance measurements.

Request

CMD	CTAPHID_PING
BCNT	0..n
DATA	n bytes

Response at success

CMD	CTAPHID_PING
BCNT	n
DATA	N bytes

6.1.9.1.4 CTAPHID_ERROR

This command code is used in response messages only.

CMD	CTAPHID_ERROR
BCNT	1
DATA	Error code

The following error codes are defined

ERR_INVALID_CMD	The command in the request is invalid
ERR_INVALID_PAR	The parameter(s) in the request is invalid
ERR_INVALID_LEN	The length field (BCNT) is invalid for the request
ERR_INVALID_SEQ	The sequence does not match expected value
ERR_MSG_TIMEOUT	The message has timed out
ERR_CHANNEL_BUSY	The device is busy for the requesting channel

6.1.9.2 Optional commands

The following commands are defined by this specification but are optional and does not have to be implemented.

6.1.9.2.1 CTAPHID_WINK

The wink command performs a vendor-defined action that provides some visual or audible identification a particular authenticator device. A typical implementation will do a short burst of flashes with a LED or something similar. This is useful when more than one device is attached to a computer and there is confusion which device is paired with which connection.

Request

CMD	CTAPHID_WINK
BCNT	0
DATA	N/A

Response at success

CMD	CTAPHID_WINK
BCNT	0
DATA	N/A

6.1.9.2.2 CTAPHID_LOCK

The lock command places an exclusive lock for one channel to communicate with the device. As long as the lock is active, any other channel trying to send a message will fail. In order to prevent a stalling or crashing application to lock the device indefinitely, a lock time up to 10 seconds may be set. An application requiring a longer lock has to send repeating lock commands to maintain the lock.

Request

CMD	CTAPHID_LOCK
BCNT	1
DATA	Lock time in seconds 0..10. A value of 0 immediately releases the lock

Response at success

CMD	CTAPHID_LOCK
BCNT	0
DATA	N/A

6.1.9.3 Vendor specific commands

A CTAPHID may implement additional vendor specific commands that are not defined in this specification, yet being CTAPHID compliant. Such commands, if implemented must have a command in the range between CTAPHID_VENDOR_FIRST and CTAPHID_VENDOR_LAST.

6.2 Near Field Communication (NFC)

6.2.1 Conformance

Please refer to [ISOIEC-7816-4-2013] for APDU definition.

6.2.2 Protocol

The general protocol between a FIDO 2.0 client and an authenticator over NFC is as follows:

1. Client sends an applet selection command
2. Authenticator replies with success
3. Client sends a command for an operation (MakeCredential or GetAssertion)
4. Authenticator replies with response data or error

6.2.3 Applet selection

A successful Select allows the client to know that the applet is present and active. A client shall send a Select to the authenticator before any other command.

The FIDO 2.0 AID consists of the following fields:

Field	Value
RID	0xA000000647
AC	TBD
AX	0x0001

The command to select the FIDO 2.0 applet is:

CLA	INS	P1	P2	Lc	Data In	Le
0x00	0xA4	0x04	0x0C	0x08	AID	TBD (version string length)

In response to the applet selection command, the FIDO authenticatorshall reply with its version string in the successful response. In this writing, the version string is "TBD", hence a successful response to the applet selection command would consist of the following bytes:

```
0xXX..XX9000
```

6.2.4 Framing

Conceptually, framing defines an encapsulation of FIDO 2.0 commands. In NFC, this encapsulation is done in an APDU following [ISOIEC-7816-4-2013]. Fragmentation, if needed, is discussed in the following paragraph.

6.2.4.1 Request

Requests APDU shall have the following format:

CLA	INS	P1	P2	Lc	Data In	Le
0x80	Command value	0x00	0x00	Variable	Command parameters	Variable

6.2.4.2 Response

Response shall have the following format in case of success:

Data	Status word
Response data	"9000" - Success For other values, see [ISOIEC-7816-4-2013]

6.2.5 APDU length

Length fields (Lc and Le) can be short (encoding a length up to 255) or extended (encoding a length up to 65535).

Some responses may not fit into a short APDU response. For this reason, FIDO 2.0 authenticators must respond in the following way:

- If the request was encoded using**extended length** APDU encoding, the authenticator must respond using the extended length APDU response format.
- If the request was encoded using**short** APDU encoding, the authenticator must respond using ISO 7816-4 APDU chaining.

6.3 Bluetooth Smart / Bluetooth Low Energy (BLE)

6.3.1 Conformance

Authenticator and Client devices using BLE shall conform to Bluetooth Core Specification 4.0 or later [BTCORE]

Bluetooth(tm) SIG specified UUID values shall be found on the Assigned Numbers website [BTASSNUM]

6.3.2 Pairing

BLE is a long-range wireless protocol and thus has several implications for privacy, security, and overall user-experience. Because it is wireless, BLE may be subject to monitoring, injection, and other network-level attacks.

For these reasons, Clients and Authenticators must create and use a long-term link key (LTK) and shall encrypt all communications. Authenticator must never use short term keys.

Because BLE has poor ranging (*i.e.,* there is no good indication of proximity), it may not be clear to a FIDO Client with which BLE Authenticator it should communicate. Pairing is the only mechanism defined in this protocol to ensure that FIDO Clients are interacting with the expected BLE Authenticator. As a result, Authenticator manufacturers should instruct users to avoid performing Bluetooth pairing in a public space such as a cafe, shop or train station.

One disadvantage of using standard Bluetooth pairing is that the pairing is "system-wide" on most operating systems. That is, if an Authenticator is paired to a FIDO Client which resides on an operating system where Bluetooth pairing is "system-wide", then any application on that device might be able to interact with an Authenticator. This issue is discussed further in Implementation Considerations.

6.3.3 Link Security

For BLE connections, the Authenticator shall enforce `Security Mode 1, Level 2` (unauthenticated pairing with encryption) before any FIDO 2.0 messages are exchanged.

6.3.4 Framing

Conceptually, framing defines an encapsulation of FIDO 2.0 raw messages responsible for correct transmission of a single request and its response by the transport layer.

All requests and their responses are conceptually written as a single frame. The format of the requests and responses is given first as complete frames. Fragmentation is discussed next for each type of transport layer.

6.3.4.1 Request from Client to Authenticator

Request frames must have the following format

Offset	Length	Mnemonic	Description
0	1	`CMD`	Command identifier
1	1	`HLEN`	High part of data length
2	1	`LLEN`	Low part of data length
3	s	`DATA`	Data (s is equal to the length)

Supported commands are `PING` and `MSG`. The constant values for them are described below.

The data format for the `MSG` command is defined in the Message Encoding section of this document.

6.3.4.2 Response from Authenticator to Client

Response frames must have the following format, which share a similar format to the request frames:

Offset	Length	Mnemonic	Description
0	1	`STAT`	Response status
1	1	`HLEN`	High part of data length
2	1	`LLEN`	Low part of data length
3	s	`DATA`	Data (s is equal to the length)

When the status byte in the response is the same as the command byte in the request, the response is a successful response. The value `ERROR` indicates an error, and the response data contains an error code as a variable-length, big-endian integer. The constant value for `ERROR` is described below.

Note that the errors sent in this response are errors at the encapsulation layer, *e.g.,* indicating an incorrectly formatted request, or possibly an error communicating with the Authenticator's FIDO 2.0 message processing layer. Errors reported by the FIDO 2.0 message processing layer itself are considered a success from the encapsulation layer's point of view, and are reported as a complete `MSG` response.

Data format is defined in the Message Encoding section of this document.

6.3.4.3 Command, Status, and Error constants

The COMMAND constants and values are:

Constant	Value
`PING`	0x81
`KEEPALIVE`	0x82
`MSG`	0x83
`ERROR`	0xbf

The KEEPALIVE command contains a single byte with the following possible values:

Status Constant	Value
PROCESSING	0x01
TUP_NEEDED	0x02
RFU	0x00, 0x03-0xFF

The ERROR constants and values are:

Error Constant	Value	Meaning
ERR_INVALID_CMD	0x01	The command in the request is unknown/invalid
ERR_INVALID_PAR	0x02	The parameter(s) of the command is/are invalid or missing
ERR_INVALID_LEN	0x03	The length of the request is invalid
ERR_INVALID_SEQ	0x04	The sequence number is invalid
ERR_REQ_TIMEOUT	0x05	The request timed out
NA	0x06	Value reserved (HID)
NA	0x0a	Value reserved (HID)
NA	0x0b	Value reserved (HID)
ERR_OTHER	0x7f	Other, unspecified error

6.3.5 GATT Service Description

This profile defines two roles: FIDO Authenticator and FIDO Client.

- The FIDO Client shall be a GATT Client
- The FIDO Authenticator shall be a GATT Server

The following figure illustrates the mandatory services and characteristics that shall be offered by a FIDO Authenticator as part of its GATT server:

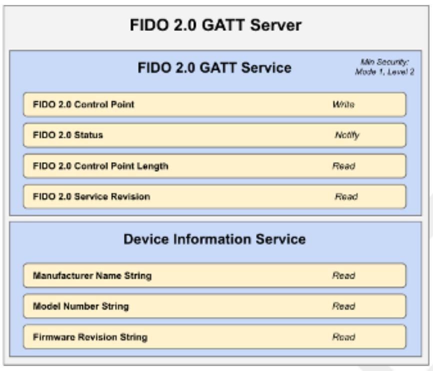

Fig. 1 Mandatory GATT services and characteristics that must be offered by a FIDO Authenticator. Note that the Generic Access Service (GAS) is not present as it is already mandatory for any BLE compliant device.

The table below summarizes additional GATT sub-procedure requirements for a FIDO Authenticator (GATT Server) beyond those required by all GATT Servers.

GATT Sub-Procedure	Requirements
Write Characteristic Value	Mandatory
Notifications	Mandatory
Read Characteristic Descriptors	Mandatory
Write Characteristic Descriptors	Mandatory

The table below summarizes additional GATT sub-procedure requirements for a FIDO 2.0 Client (GATT Client) beyond those required by all GATT Clients.

GATT Sub-Procedure	Requirements
Discover All Primary Services	(*)
Discover Primary Services by Service UUID	(*)
Discover All Characteristics of a Service	(**)
Discover Characteristics by UUID	(**)
Discover All Characteristic Descriptors	Mandatory
Read Characteristic Value	Mandatory
Write Characteristic Value	Mandatory
Notification	Mandatory
Read Characteristic Descriptors	Mandatory
Write Characteristic Descriptors	Mandatory

(*): Mandatory to support at least one of these sub-procedures.

(**): Mandatory to support at least one of these sub-procedures.

Other GATT sub-procedures may be used if supported by both client and server.

Specifics of each service are explained below. In the following descriptions: all values are big-endian coded, all strings are in UTF-8 encoding, and any characteristics not mentioned explicitly are optional.

6.3.5.1 FIDO 2.0 Service

An Authenticator shall implement the FIDO 2.0 Service described below. The UUID for the FIDO 2.0 GATT service is TODO:`0x????`, it shall be declared as a Primary Service. The service contains the following characteristics:

Characteristic Name	Mnemonic	Property	Length	UUID
FIDO 2.0 Control Point	`fido2ControlPoint`	Write	Defined by Vendor (20-512 bytes)	TBD
FIDO 2.0 Status	`fido2Status`	Notify	N/A	TBD
FIDO 2.0 Control Point Length	`fido2ControlPointLength`	Read	2 bytes	TBD
FIDO 2.0 Service Revision	`fido2ServiceRevision`	Read	Defined by Vendor (20-512 bytes)	0x2A28

`fido2ControlPoint` is a write-only command buffer.

`fido2Status` is a notify-only response attribute. The Authenticator will send a series of notifications on this attribute with a maximum length of (ATT_MTU-3) using the response frames defined above. This mechanism is used because this results in a faster transfer speed compared to a notify-read combination.

`fido2ControlPointLength` defines the maximum size in bytes of a single write request to `fido2ControlPoint`. This value shall be between 20 and 512.

`fido2ServiceRevision` defines the revision of the FIDO 2.0 Service. The value is a UTF-8 string. For this version of the specification, the value `fido2ServiceRevision` shall be `FIDO 2.0 Rev 1` or in raw bytes: `0x4649444f20322e30205265762031`.

The `fido2ServiceRevision` Characteristic may include a Characteristic Presentation Format descriptor with format value 0x19, `UTF-8 String`.

6.3.5.2 Device Information Service

An Authenticator shall implement the Device Information Service [BTDIS] with the following characteristics:

- Manufacturer Name String
- Model Number String
- Firmware Revision String

All values for the Device Information Service are left to the vendors. However, vendors should not create uniquely identifiable values so that Authenticators do not become a method of tracking users.

6.3.5.3 Generic Access Service

Every Authenticator shall implement the Generic Access Service [BTGAS] with the following characteristics:

- Device Name
- Appearance

6.3.6 Protocol Overview

The general overview of the communication protocol follows:

1. Authenticator advertises the FIDO 2.0 Service.
2. Client scans for Authenticator advertising the FIDO 2.0 Service.
3. Client performs characteristic discovery on the Authenticator.
4. If not already paired, the Client and Authenticatorshall perform BLE pairing and create a LTK. Authenticator shall only allow connections from previously bonded Clients without user intervention.
5. Client reads the `fido2ControlPointLength` characteristic.
6. Client registers for notifications on the `fido2Status` characteristic.
7. Client writes a request (*e.g.*, an enroll request) into the `fido2ControlPoint` characteristic.
8. Authenticator evaluates the request and responds by sending notifications over `fido2Status` characteristic.
9. The protocol completes when either:
    - The Client unregisters for notifications on the `fido2Status` characteristic, or:
    - The connection times out and is closed by the Authenticator.

6.3.7 Authenticator Advertising Format

When advertising, the Authenticator shall advertise the FIDO 2.0 service UUID.

When advertising, the Authenticator may include the TxPower value in the advertisement (see [BTXPLAD]).

The advertisement may also carry a device name which is distinctive and user-identifiable. For example, "ACME Key" would be an appropriate name, while "XJS4" would not be.

The Authenticator shall also implement the Generic Access Profile [BTGAP] and Device Information Service [BTDIS], both of which also provide a user-friendly name for the device that could be used by the Client.

It is not specified when or how often an Authenticator should advertise, instead that flexibility is left to manufacturers.

6.3.8 Requests

Clients should make requests by connecting to the Authenticator and performing a write into the `fido2ControlPoint` characteristic.

6.3.9 Responses

Authenticators should respond to Clients by sending notifications on the `fido2Status` characteristic.

Some Authenticators might alert users or prompt them to complete the test of user presence (*e.g.*, via sound, light, vibration) Upon receiving any request, the Authenticators shall send KEEPALIVE commands every `kKeepAliveMillis` milliseconds until completing processing the commands. While the Authenticator is processing the request the KEEPALIVE command will contain status `PROCESSING`. If the Authenticator is waiting to complete the Test of User Presence, the KEEPALIVE command will contains status `TUP_NEEDED`. While waiting to complete the Test of User Presence, the Authenticator may alert the user (e.g., by flashing) in order to prompt the user to complete the test of user presence. As soon the Authenticator has completed processing and confirmed user presence, it shall stop sending KEEPALIVE commands, and send the reply.

Upon receiving a KEEPALIVE command, the Client shall assume the Authenticator is still processing the command; the Client shall not resend the command. The Authenticator shall continue sending KEEPALIVE messages at least every `kKeepAliveMillis` to indicate that it is still handling the request. Until a client-defined timeout occurs, the Client shall not move on to other devices when it receives a KEEPALIVE with `TUP_NEEDED` status, as it knows this is a device that can satisfy its request.

6.3.10 Framing fragmentation

A single request/response sent over BLE may be split over multiple writes and notifications, due to the inherent limitations of BLE which is not currently meant for large messages. Frames are fragmented in the following way:

A frame is divided into an *initialization fragment* and one or more *continuation fragments*.

An initialization fragment is defined as:

Offset	Length	Mnemonic	Description
0	1	`CMD`	Command identifier
1	1	`HLEN`	High part of data length
2	1	`LLEN`	Low part of data length
3	0 to (maxLen - 3)	`DATA`	Data

where `maxLen` is the maximum packet size supported by the characteristic or notification.

In other words, the start of an initialization fragment is indicated by setting the high bit in the first byte. The subsequent two bytes indicate the total length of the frame, in big-endian order. The first `maxLen` - 3 bytes of data follow.

Continuation fragments are defined as:

Offset	Length	Mnemonic	Description
0	1	`SEQ`	Packet sequence 0x00..0x7f (high bit always cleared)
1	0 to (maxLen - 1)	`DATA`	Data

where `maxLen` is the maximum packet size supported by the characteristic or notification.

In other words, continuation fragments begin with a sequence number, beginning at 0, implicitly with the high bit cleared. The sequence number must wrap around to 0 after reaching the maximum sequence number of 0x7f.

Example for sending a `PING` command with 40 bytes of data with a `maxLen` of 20 bytes:

Frame	Bytes
0	`[810028] [17 bytes of data]`
1	`[00] [19 bytes of data]`
2	`[01] [4 bytes of data]`

Example for sending a ping command with 400 bytes of data with a `maxLen` of 512 bytes:

Frame	Bytes
0	`[810190] [400 bytes of data]`

6.3.11 Implementation Considerations

6.3.11.1 Bluetooth pairing: Client considerations

As noted in the Pairing section, a disadvantage of using standard Bluetooth pairing is that the pairing is "system-wide" on most operating systems. That is, if an Authenticator is paired to a FIDO Client which resides on an operating system where Bluetooth pairing is "system-wide", then any application on that device might be able to interact with an Authenticator. This poses both security and privacy risks to users.

While Client operating system security is partly out of FIDO's scope, further revisions of this specification may propose mitigations for this issue.

6.3.11.2 Bluetooth pairing: Authenticator considerations

The method to put the Authenticator into Pairing Mode should be such that it is not easy for the user to do accidentally **especially** if the pairing method is Just Works. For example, the action could be pressing a physically recessed button or pressing multiple buttons. A visible or audible cue that the Authenticator is in Pairing Mode should be considered. As a counter example, a silent, long press of a single non-recessed button is not advised as some users naturally hold buttons down during regular operation.

Note that at times, Authenticators may legitimately receive communication from an unpaired device. For example, a user attempts to use an Authenticator for the first time with a new Client: he turns it on, but forgets to put the Authenticator into pairing mode. In this situation, after connecting to the Authenticator, the Client will notify the user that he needs to pair his Authenticator. The Authenticator should make it easy for the user to do so, e.g., by not requiring the user to wait for a timeout before being able to enable pairing mode.

6.3.11.3 Handling command completion

It is important for low-power devices to be able to conserve power by shutting down or switching to a lower-power state when they have satisfied a Client's requests. However, the FIDO 2.0 protocol makes this hard as it typically includes more than one command/response. This is especially true if a user has more than one key handle associated with an account or identity, multiple key handles may need to be tried before getting a successful outcome. Furthermore, Clients that fail to send follow-up commands in a timely fashion may cause the Authenticator to drain its battery by staying powered up anticipating more commands.

A further consideration is to ensure that a user is not confused about which command she is confirming by completing the test of user presence.

That is, if a user performs the test of user presence, that action should perform exactly one operation.

We combine these considerations into the following series of recommendations:

- Upon initial connection to an Authenticator, and upon receipt of a response from an Authenticator, if a Client has more commands to issue, the Client must transmit the next command or fragment within `kMaxCommandTransmitDelayMillis` milliseconds.
- Upon final response from an Authenticator, if the Client decides it has no more commands to send it should indicate this by disabling notifications on the `fido2Status` characteristic. When the notifications are disabled the Authenticator may enter a low power state or disconnect and shut down.
- Any time the Client wishes to send a FIDO 2.0 message, it must have first enabled notifications on the `fido2Status` characteristic and wait for the ATT acknowledgement to be sure the Authenticator is ready to process messages.
- Upon successful completion of a command which required a test of user presence, e.g. upon a successful authentication or registration command, the Authenticator can assume the Client is satisfied, and may reset its state or power down.
- Upon sending a command response that did not consume a test of user presence, the Authenticator must assume that the Client may wish to initiate another command, and leave the connection open until the Client closes it or until a timeout of at least `kErrorWaitMillis` elapses. Examples of command responses that do not consume user presence include failed authenticate or register commands, as well as get version responses, whether successful or not. After `kErrorWaitMillis` milliseconds have elapsed without further commands from a Client, an Authenticator may reset its state or power down.

Constant	Value
`kMaxCommandTransmitDelayMillis`	1500 milliseconds
`kErrorWaitMillis`	2000 milliseconds
`kKeepAliveMillis`	500 milliseconds

6.3.11.4 Data throughput

BLE does not have particularly high throughput, this can cause noticeable latency to the user if request/responses are large. Some ways that implementers can reduce latency are:

- Support the maximum MTU size allowable by hardware (up to the 512-byte max from the BLE specifications).
- Make the attestation certificate as small as possible; do not include unnecessary extensions.

6.3.11.5 Advertising

Though the standard doesn't appear to mandate it (in any way that we've found thus far), advertising and device discovery seems to work better when the Authenticators advertise on all 3 advertising channels and not just one.

6.3.11.6 Authenticator Address Type

In order to enhance the user's privacy and specifically to guard against tracking, it is recommended that Authenticators use Resolvable Private Addresses (RPAs) instead of static addresses.

7. Bibliography

[ISOIEC-7816-4-2013] ISO 7816-4: Identification cards - Integrated circuit cards; Part 4: Organization, security and commands for interchange

A. References

A.1 Normative references

[FIDOKeyAttestation]
FIDO 2.0: Key attestation format. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-key-attestation-v2.0-ps-20150904.html
[FIDOPlatformApiReqs]
FIDO 2.0: Requirements for Native Platforms. URL: fido-platform-api-reqs.html
[FIDOSignatureFormat]
FIDO 2.0: Signature format. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-signature-format-v2.0-ps-20150904.html
[FIDOWebApi]
FIDO 2.0: Web API for accessing FIDO 2.0 credentials URL: https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-web-api-v2.0-ps-20150904.html
[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119
[RFC5280]
D. Cooper, S. Santesson, s. Farrell, S.Boeyen, R. Housley, W. Polk;*Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, IETF, May 2008, URL:http://www.ietf.org/rfc/rfc5280.txt
[RFC7049]
C. Bormann; P. Hoffman. *Concise Binary Object Representation (CBOR)*. October 2013. Proposed Standard. URL: https://tools.ietf.org/html/rfc7049