Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 8 December 2023

V. Birk
B. Hoeneisen
H. Marques
pEp Foundation

6 June 2023

pretty Easy privacy (pEp): Key Synchronization Protocol (KeySync) draft-pep-keysync-03

Abstract

This document describes the pEp KeySync protocol, which is designed to perform secure peer-to-peer synchronization of private keys across devices belonging to the same user.

Modern users of messaging systems typically have multiple devices for communicating, and attempting to use encryption on all of these devices often leads to situations where messages cannot be decrypted on a given device due to missing private key data. Current approaches to resolve key synchronicity issues are cumbersome and potentially insecure. The pEp KeySync protocol is designed to facilitate this personal key synchronization in a user-friendly manner.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-pep-keysync/.

Discussion of this document takes place on the medup non-WG mailing list (mailto:medup@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/medup/.

Source for this draft and an issue tracker can be found at https://gitea.pep.foundation/pEp.foundation/internet-drafts.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Copyright Notice

Table of Contents

1. Introduction

The pretty Easy privacy (pEp) [I-D.pep-general] protocols describe a set of conventions for the automation of operations traditionally seen as barriers to the use and deployment of secure end-to-end interpersonal messaging. These include, but are not limited to, key management, key discovery, and private key handling.

This document specifies the pEp KeySync protocol, a means for secure, decentralized, peer-to-peer synchronization of private keys across devices belonging to the same user, allowing that user to send and receive encrypted communications from any of their devices.

For pEp implementations, pEp KeySync is a critical part of the broader pEp Sync protocol, which is designed to be extensible to allow for the synchronization of additional user data, such as configuration settings and peer trust status information across a single user's devices.

This document will provide a general description of pEp KeySync, including idealized use cases, diagrams, and examples of messages that may be generated during the KeySync process.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. Terms

The following terms are defined for the scope of this document:

* pEp Handshake: The process of one User contacting another over an independent channel in order to verify Trustwords (or fingerprints as a fallback). This can be done in-person or through established verbal communication channels, like a phone call. [I-D.pep-handshake]

Note: In pEp KeySync, the Handshake is used to authenticate own devices (the User normally compares the Trustwords directly by looking at the screens of the devices involved).

* Trustwords: A representation of 16-bit natural numbers (0 to 65535) as natural language words: For each natural language a fixed number-to-word map can be defined as convention and registered with IANA. Trustwords are generated from the combined public key fingerprints of a both communication partners. Trustwords are used for verification and establishment of trust (for the respective keys and communication partners). [I-D.pep-trustwords]

* Transport: A general description of what Transport means in pEp context can be found in [I-D.pep-general].

  For pEp Sync there are special requirements to the Transport, i.e., that messages can be sent to ones own devices and that every own device (including the sender) receives the messages in the same order.

* Trust On First Use (TOFU): cf. [RFC7435], which states: "In a protocol, TOFU calls for accepting and storing a public key or credential associated with an asserted Identity, without authenticating that assertion. Subsequent communication that is authenticated using the cached key or credential is secure against an MiTM attack, if such an attack did not succeed during the vulnerable initial communication."

* Man-in-the-middle (MITM) attack: cf. [RFC4949], which states: "A form of active wiretapping attack in which the attacker intercepts and selectively modifies communicated data to masquerade as one or more of the entities involved in a communication association."

  Note: Historically, MITM has stood for '_Man_-in-the-middle'. However, to indicate that the entity in the middle is not always a human attacker, MITM can also stand for 'Machine-in-the-middle' or 'Meddler-in-the-middle'.

* User: An individual entity using pEp. A User may have one or more Identities.

* User-ID: A unique identifier for a given User.

* Address: An Address in pEp means the designator of a destination where messages can be routed to and accessed from, e.g., email address, Uniform Resource Identifier (URI), Network Access Identifier (NAI), phone number, etc. An Address may belong to one or more Users. A User may have multiple Addresses.

* Identity: A binding between a User (unique User-ID) and an Address (email, network ID, URI, etc.). Each Identity is uniquely identified by this binding. Identities contain a number of different pieces of information, often including, but not limited to:

  - User-ID

  - Address

  - Default Key

  - Username

  - Preferred encryption format

  - Information about whether this is an Own Identity

  - ...

  A single User may have multiple identities. See also [RFC4949].

* Own Identity: An Identity corresponding to one of the Own User's Addresses.

* Device Group: A set of devices controlled by one pEp User that have successfully completed the KeySync setup process and synchronize Identity information, such as cryptographic keys. This data is synchronized through a common channel for a given Identity. For example, if a User's Identity is tied to a specific email address, the common channel for this Identity could be an inbox.

* Sole Device: A device which is not part of a Device Group.

* Grouped Device: A device which is already part of a Device Group.

* Beacon (message): A technical text message that is broadcast by Sole Devices and transmitted through a message sent to the channel of an Identity. Other Sole Devices, or a Grouped Device of the same unique Identity and using that Identity's channel, can interpret this Beacon in order to initiate negotiation for the formation of a Device Group.

* Transaction ID (TID): A UUID version 4, variant 1 number generated by each device during the pEp KeySync process in order to identify the respective devices involved.

* Default Key: The Identity's or User's key that is marked as the key to be used to encrypt to (in the case of communications partners) or to sign with (in the case of the Own User).

* Own Key: A public/private keypair corresponding to a User's Own Identity.

1.3. Problem Statement

Secure and private digital communication is becoming a necessity for many people. Encryption protocols which utilize key pairs are the most popular and easily implemented methods to ensure a message is authentic and can be trusted.

However, most modern users have multiple devices for communicating, and attempting to use encryption on all of these devices often leads to situations where messages cannot be decrypted on a given device due to missing private key data. For example, Alice sends an encrypted message to Bob, using the public key of a key pair that Bob generated on his laptop. When Bob attempts to decrypt the message on his mobile phone, the private key that he generated on his laptop is not available. As a result, Bob must either use his laptop to decrypt the message, or attempt to copy the correct private key to his mobile device, which may expose his private key to potential leaks or theft. Using, in turn, centralized solutions to share the missing private key data has the disadvantage to be prone to infrastructure attacks and also leads to availability issues.

1.4. Main Challenge

The main challenge that pEp KeySync is designed to overcome is to perform the synchronization in a secure manner so that private keys are not leaked or exposed to theft.

Note: The case of an adversary getting physical access to the device itself is beyond the scope of this document.

1.5. Approach

The basic approach to solving the multiple-device decryption problem is to synchronize private keys among the devices of a User in a secure manner. pEp achieves this by giving Users the option to form a Device Group with their devices. When the User initiates this process, a Handshake occurs, and the User is presented with a Trustwords dialog for pairing purposes (cf. [I-D.pep-trustwords]). Simply put, the User MUST complete this Trustwords dialog (to confirm for the authenticity of the transport channel) before the automatic and security-sensitive transfer of private key information can occur.

2. General Description

The pEp KeySync protocol allows a User to securely synchronize private key data for multiple Identities across their various devices. This synchronization process is decentralized and performed as a two-phase commit (2PC) protocol structure. This structure ensures consensus among the devices at all stages of the KeySync process.

KeySync's 2PC transaction is accomplished through the implementation of a Finite State Machine (FSM) on each pEp-enabled device. This FSM not only sends and receives network traffic, which allows devices to communicate with each other throughout the KeySync process, but also interacts with the core pEp implementation itself.

Once activated by the User, pEp KeySync initiates the formation of a Device Group, and the User is guided through a Handshake process on its respective devices. A User can choose to reject or cancel this process at any time, from either device, and private key data is not exchanged until the group formation process is verified on both devices.

Once a Device Group is formed, a User can add additional devices to its group through the same joining procedure. Upon adding the new device to the existing Device Group, key data is synchronized among all Grouped Devices, allowing a User to communicate privately from any of its secure Identities.

2.1. Use Cases for pEp KeySync

This section describes ideal-condition use cases for pEp KeySync. The focus is on the core procedures and on the scenarios where everything works. Unexpected user behavior, error handling, race conditions, etc., are generally omitted from this section in order to focus on the general concepts of pEp KeySync. Additional use cases will be discussed in further detail throughout Appendix A.

2.1.1. Form Device Group

Our User, Alice, has two devices that are configured with pEp-implementing messaging clients and share the same Identity for her preferred communication channel. In our example, this communication channel is the inbox for a specific email address, alice@example.org, which Alice has configured on each device. Let us call these devices Alice_Mobile and Alice_Tablet. Each device already has its own dedicated key pair, which was automatically generated by the pEp protocol when Alice configured her email inbox on her respective devices.

When Alice sends an email from Alice_Mobile, it is encrypted by the key for that specific device, as are any replies she might receive. If she wishes to read that email (or replies to it) on Alice_Tablet, she is unable to do so because the key pair for Alice_Tablet is different. Alice wants to read all of her encrypted communications on both of her devices, but currently cannot do so, as the devices do not have any authenticated and secure established connection to each other and thus cannot share key pair data without compromising her privacy.

Alice will use pEp KeySync to form a Device Group and add her devices to it. pEp KeySync provides an authenticated and secure connection for Alice to exchange private key data among her devices, which will allow her to have full access to all of her encrypted messages on both devices.

2.1.2. Add New Device to Existing Device Group

Sometime after devices Alice_Mobile and Alice_Tablet have formed a Device Group (cf. Section 2.1.1), Alice buys another device, Alice_Laptop, which is also configured with pEp-implementing messaging clients and shares the same Identity for her preferred communication channel (the aforementioned email address). Alice_Laptop also has a key pair, which was automatically generated by the pEp protocol, just as the Grouped Devices Alice_Mobile and Alice_Tablet have. But while the Grouped Devices know each other and have exchanged private keys, Alice_Laptop and the Grouped Devices don't have any connection to each other. Thus, Alice does not have full, encrypted communication capability across the three devices.

As before with devices Alice_Mobile and Alice_Tablet, Alice will use pEp KeySync to add device Alice_Laptop to the existing Device Group, allowing all three devices to exchange private key information, and Alice to have full access to her messages from any of them.

2.1.3. Exchange Private Keys

All devices from Alice are part of a Device Group (cf. Section 2.1.1 and Section 2.1.2). However, as keys may expire or get reset (cf. [I-D.pep-keyreset]), it is inevitable that new key pairs will be generated. For Alice to maintain her ability to read all encrypted messages on all devices, any new private key needs to be shared with the other devices in the Device Group.

All devices in Alice's Device Group will share the latest private keys as they are generated, keeping all of her devices up to date and functioning as desired.

2.1.4. Leave Device Group

Alice decides that her mobile phone, Alice_Mobile, should no longer have access to private keys of the Device Group.

Alice will use pEp KeySync on her mobile phone to leave the Device Group. This also initiates the pEp KeyReset protocol, which resets keys for all Own Identities (cf. [I-D.pep-keyreset]) on the remaining devices. Furthermore, Sync is deactivated on Alice_Mobile.

In the future, if Alice desires, she can re-add Alice_Mobile to a Device Group. If Alice wants to do this, she will first have to re-enable Sync on Alice_Mobile and then initiate the joining procedure (cf. Section 2.1.2) again. If there was only one device left, no Device Group exists anymore. In this case Alice will have to initiate the Form Device Group (cf. Section 2.1.1) instead of the joining procedure.

2.1.5. Remove other Device from Device Group

Let's assume one of Alice's devices, Alice_Tablet, was stolen or became otherwise compromised. To limit the damage, she needs to ensure that Alice_Tablet no longer receives updates to private keys from other Device Group members. Furthermore, she needs to reset the keys for all Own Identities (cf. [I-D.pep-keyreset]) on the remaining devices, which includes informing communication partners to no longer use the (potentially) compromised keys.

Note: In order to prevent any reset (new) keys to reach Alice_Tablet, the channel credentials (e.g., IMAP password) should be changed before this step.

On all of her remaining Grouped Devices, Alice needs to initiate the Leave Device Group procedure as described in Section 2.1.4. As a result, the Device Group will be dissolved.

For KeySync to work again on her remaining devices, a new Device Group needs to be formed. Therefore, Alice will first have to re-enable Sync on her remaining devices and then initiate the Form Device Group procedure (cf. Section 2.1.1) again. For every additional remaining device (if any), she will have to initiate the joining procedure (cf. Section 2.1.2) again.

2.2. Interaction Diagrams

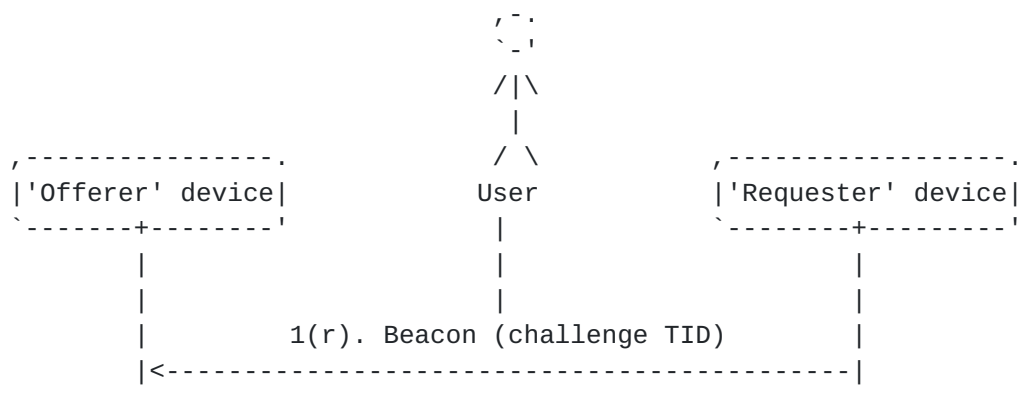
The following interaction diagrams depict what happens during Alice's KeySync scenarios in a simplified manner. For each scenario, we first present a successful case, then an unsuccessful case and, finally, a case that has been interrupted, or discontinued. Some details are skipped here for the sake of readability. Descriptions of the interactions are included after each diagram.

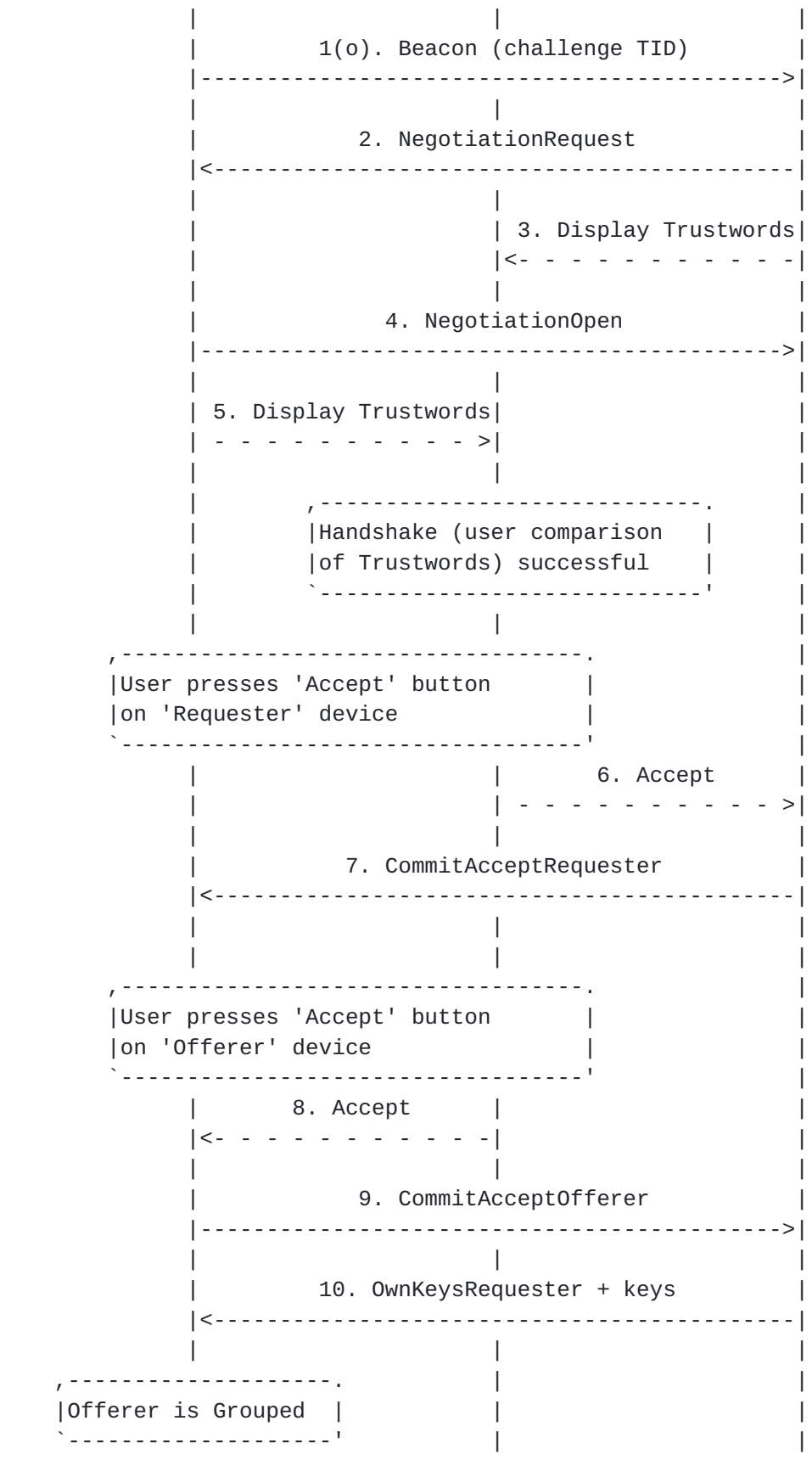
Each pEp-enabled device runs its own Finite State Machine (FSM), which interact with each other throughout the KeySync process, and drive the UI options presented to Alice (the 'User' in all diagrams, unless otherwise noted). All Messages are 'broadcast' between devices. The TIDs added to each Message allow the identification of received Messages which pertain to the ongoing transaction and the device which sent it.

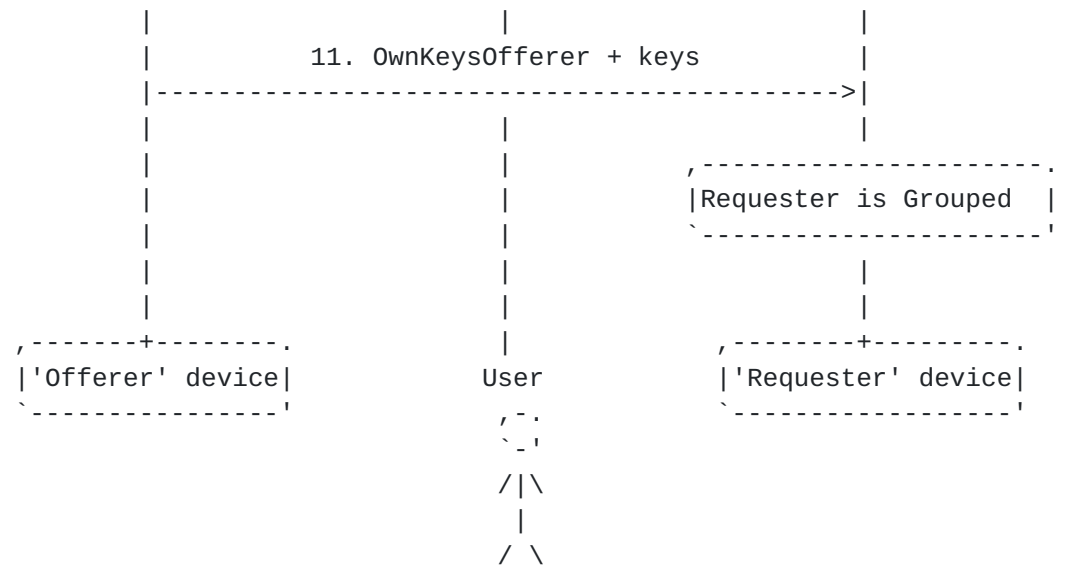
For events requiring Alice's interaction in order to proceed, it does not matter which device has the specified option chosen first unless otherwise indicated. For example, if an event states that Alice must choose 'Accept' on the 'Offerer' device in order to continue, the process will be unaffected if she does so on the 'Requester' device first. The only difference is that the order of the roles for the remainder of the given scenario will be swapped.

2.2.1. Form Device Group

2.2.1.1. Successful Case




1(o). Beacon (challenge TID)
2. NegotiationRequest
3. Display Trustwords
4. NegotiationOpen
5. Display Trustwords
Handshake (user comparison of Trustwords) successful
User presses 'Accept' button on 'Requester' device
6. Accept
7. CommitAcceptRequester
User presses 'Accept' button on 'Offerer' device
8. Accept
9. CommitAcceptOfferer
10. OwnKeysRequester + keys
Offerer is Grouped



As depicted above, our User, Alice, intends to form a Device Group in order to share key material between her devices in an authenticated and secure manner. The group is formed by an 'Offerer' device and a 'Requester' device. The names 'Offerer' and 'Requester' are derived from the FSM (cf. Appendix A.1), in which the device roles are defined during the start sequence, which is necessary for the FSM to work as intended.

During initialization of pEp KeySync, each device generates a Transaction-ID (TID). These TIDs are sent as a Challenge in a Beacon over the mutual channel, and the device roles of 'Offerer' and 'Requester' are determined by the numeric value of each device's unique TID.

1. Every device sends a Beacon Message containing a Challenge TID. Upon receipt of a Beacon Message from another device, the received Challenge TID is compared with the device's own Challenge TID. The device which has a TID with a lower numerical value is assigned as the 'Requester', and the other device is automatically assigned as the 'Offerer'.

   Note: The 'Offerer' device MUST NOT start a negotiation. In the event the earlier Beacon Message is lost, the 'Offerer' device re-sends its own Beacon and waits for a response. Message 1(r) depicts the Beacon Message sent by the 'Requester' device and is not required for the process to continue.

2. After determination of the role, the 'Requester' device sends a NegotiationRequest Message.

3. The 'Requester' device displays the Trustwords to Alice.

4. Upon receipt of the NegotiationRequest Message, the 'Offerer' device sends a NegotiationOpen Message.

5. The 'Offerer' device displays the Trustwords to Alice.

6. Alice compares the Trustwords of both devices. As the Trustwords are the same on both devices, she chooses the 'Accept' option on the 'Requester' device.

   Note: Alice may choose 'Accept' on the 'Offerer' device first, in which case the sequence of the messages is slightly different (i.e. message 8 is sent before message 6). However, the result will be exactly the same.

7. On receipt of Alice's 'Accept' from the 'Offerer' device, the 'Requester' device sends a CommitAcceptRequester Message.

   The 'Offerer' device receives this Message and waits for Alice to choose 'Accept'.

8. Alice compares the Trustwords of both devices and chooses the 'Accept' option on the 'Offerer' device.

9. Once Alice chooses 'Accept', the 'Offerer' device sends a CommitAcceptOfferer Message.

10. Upon receipt of the CommitAcceptOfferer Message, the 'Requester' device sends an OwnKeysRequester Message along with Alice's local key pairs (private and public keys) to be synchronized.

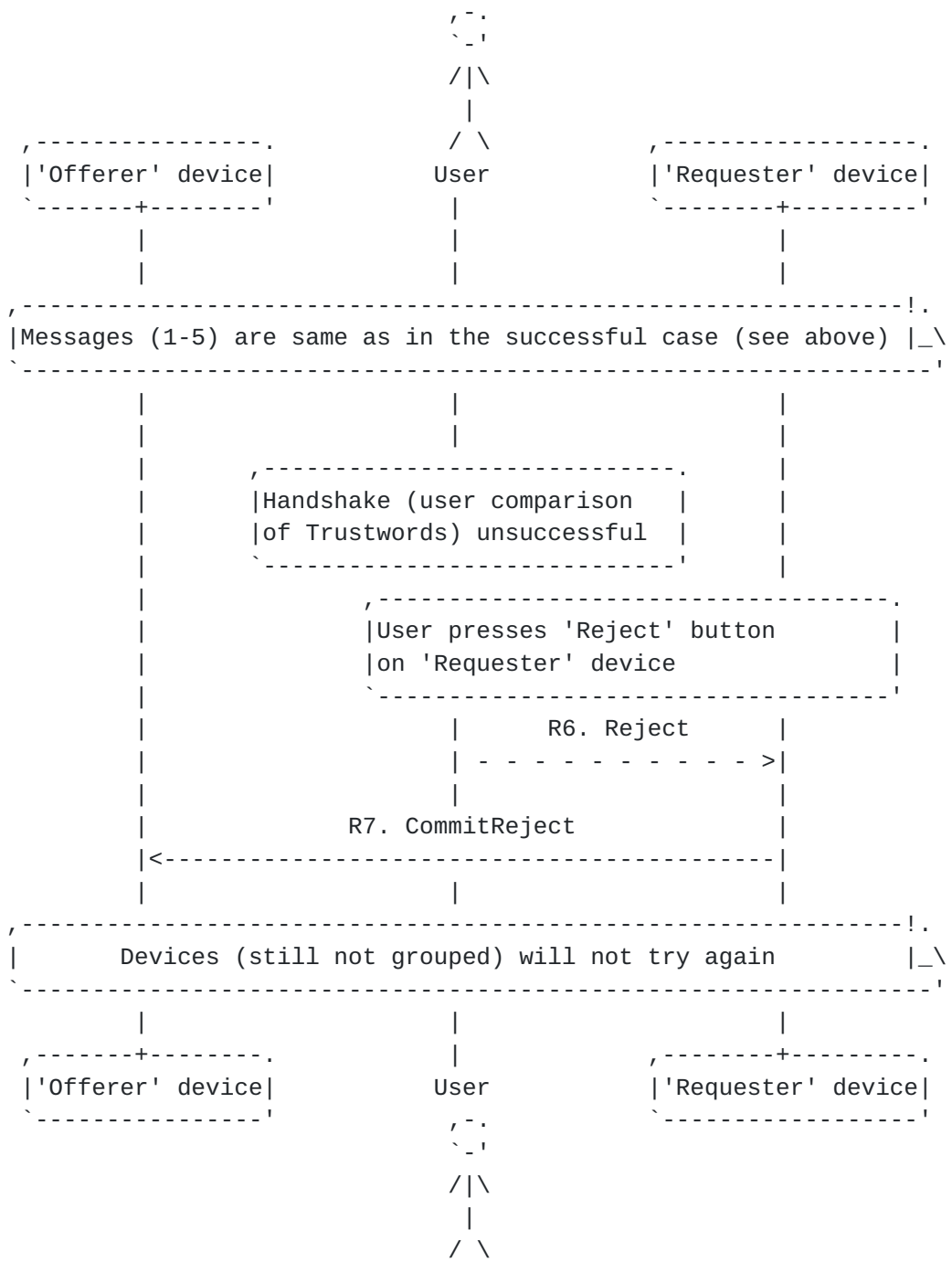
11. Upon receipt of the OwnKeysRequester Message, the 'Offerer' device saves the 'Requester' device keys and combines them with the existing 'Offerer' device keys. This means that the 'Offerer' device is grouped.

    The 'Offerer' device sends an OwnKeysOfferer Message along with its own existing local key pairs (private and public keys) to be synchronized.

    Upon receipt of the OwnKeysOfferer Message, the 'Requester' device saves the 'Offerer' keys combined with the 'Requester' keys. This means that the 'Requester' device is also grouped.

    The formation of the Device Group has been successful.

2.2.1.2. Unsuccessful Case



For unsuccessful KeySync attempts, messages 1-5 are the same as in a successful attempt (see above), but once the Trustwords are shown, events are as follows:

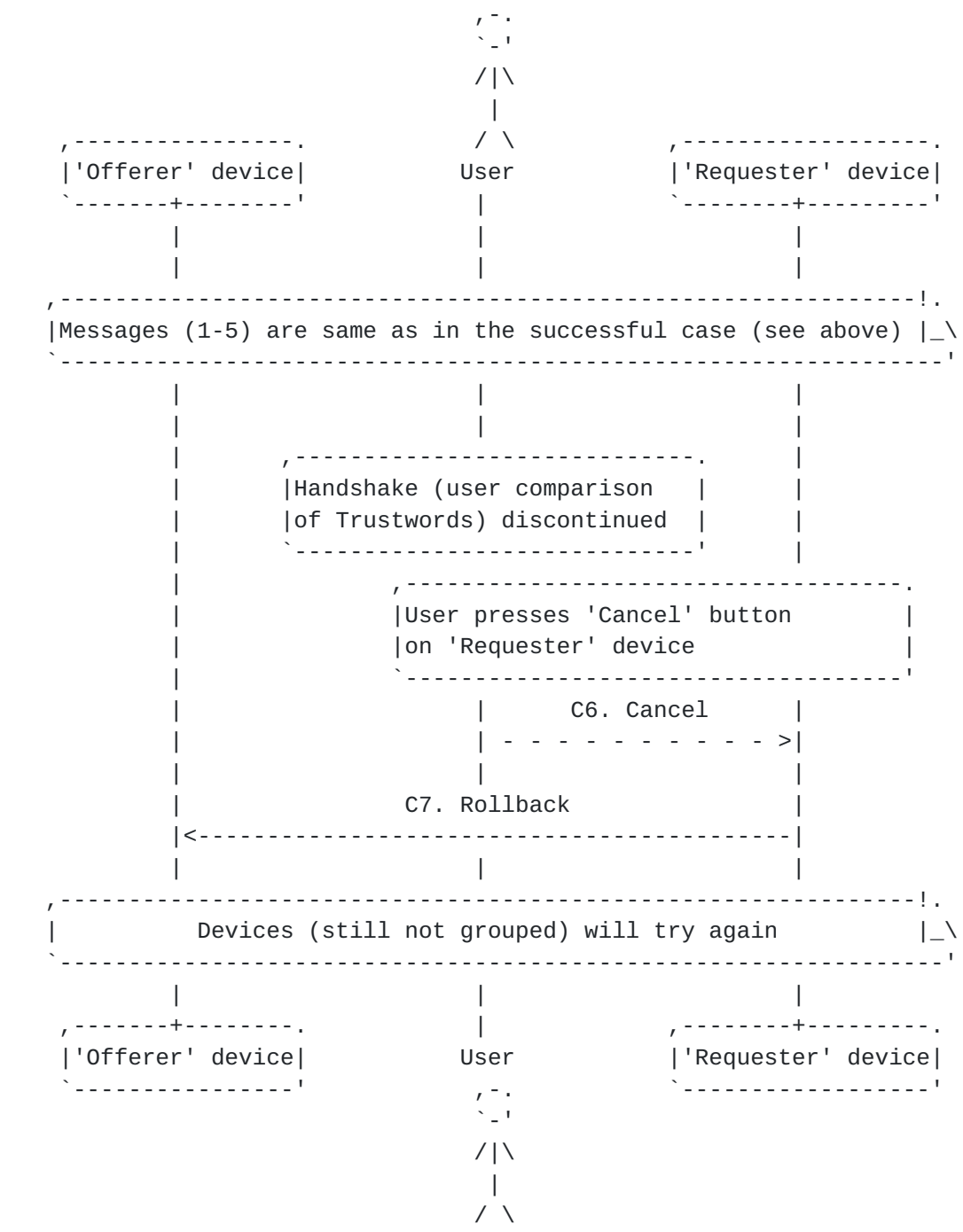
R6. Our User, Alice, compares the Trustwords of both devices. As the Trustwords do not match, she chooses the 'Reject' option on the 'Requester' device.

Note: The User may choose 'Reject' on the 'Offerer' device, in which case the origin and/or destination of the messages change. However, the result will be exactly the same.

R7. Once Alice chooses the 'Reject' option, the 'Requester' device sends a CommitReject Message to the 'Offerer' device.

Once the CommitReject Message is sent and received by the respective devices, they cannot form a Device Group, and pEp KeySync is disabled on both devices. As a result, there are no further attempts to form a Device Group involving either of these two devices. KeySync may be re-enabled in the pEp settings on the affected device(s).

2.2.1.3. Discontinuation Case



For discontinued (canceled) KeySync attempts, messages 1-5 are the same as in a successful attempt (see above), but once the Trustwords are shown, events are as follows:

C6. Our User, Alice, decides to discontinue the process and chooses the 'Cancel' option on the 'Requester' device.

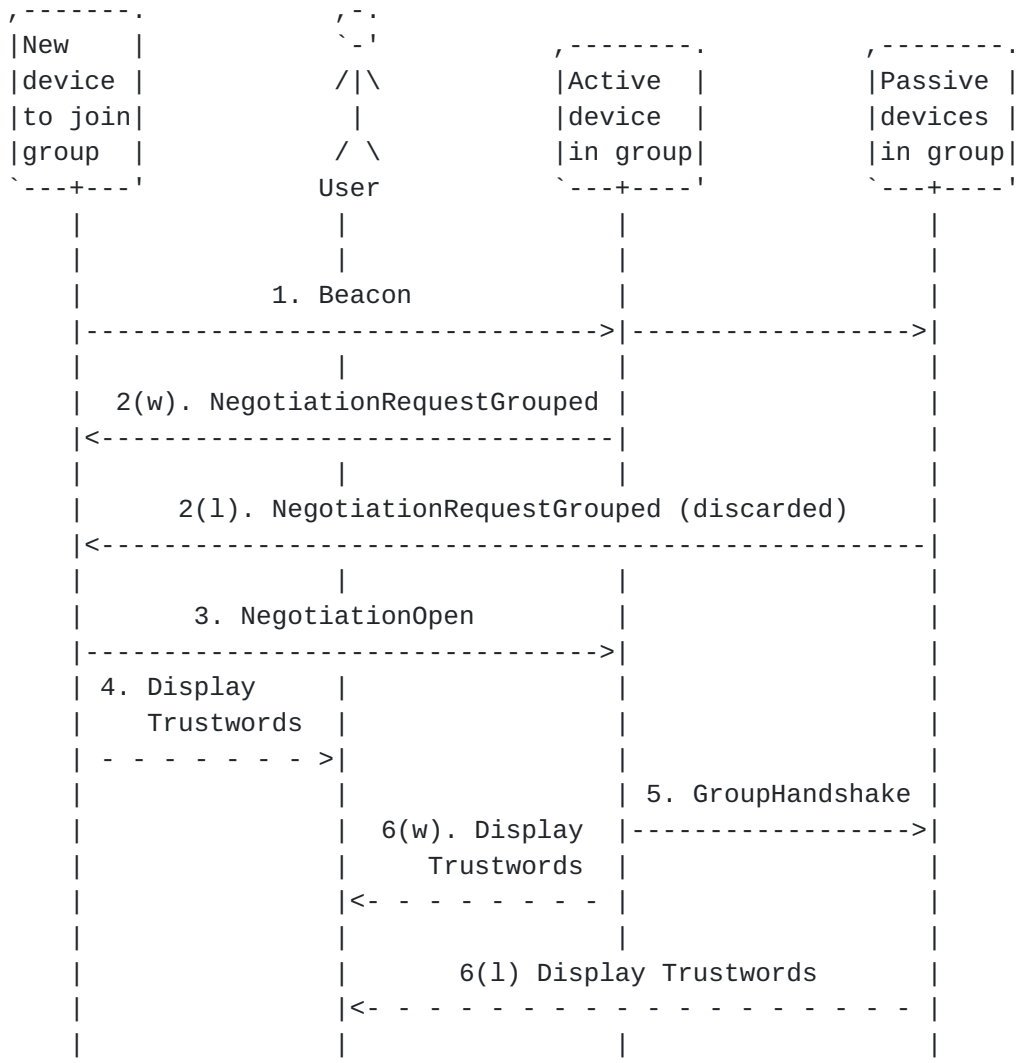
Note: The User may choose 'Cancel' on the 'Offerer' device, in which case the origin and/or destination of the messages change. However, the result will be exactly the same.

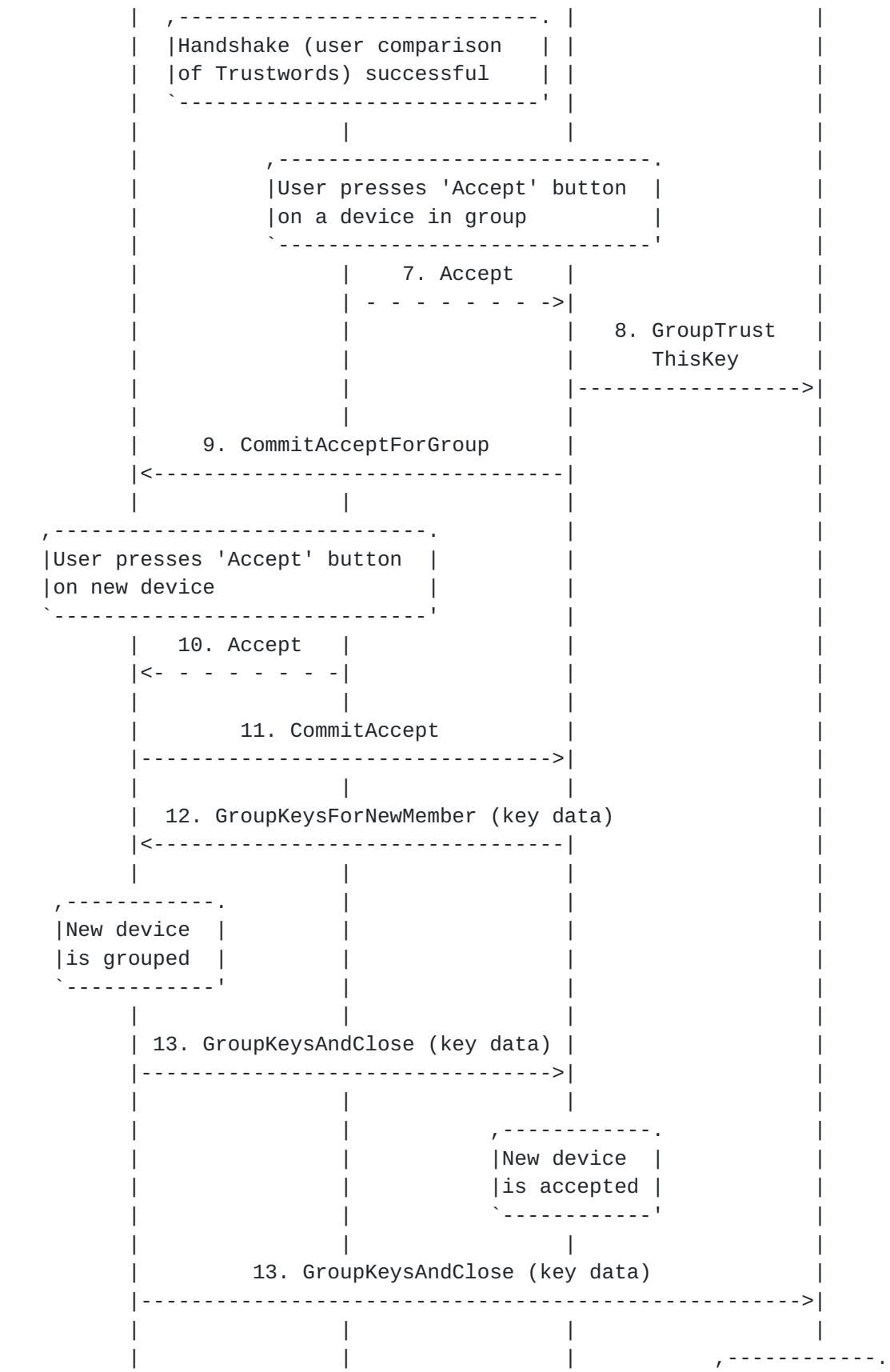
C7. Once Alice chooses the 'Cancel' option, the 'Requester' device sends a rollback Message to the 'Offerer' device.

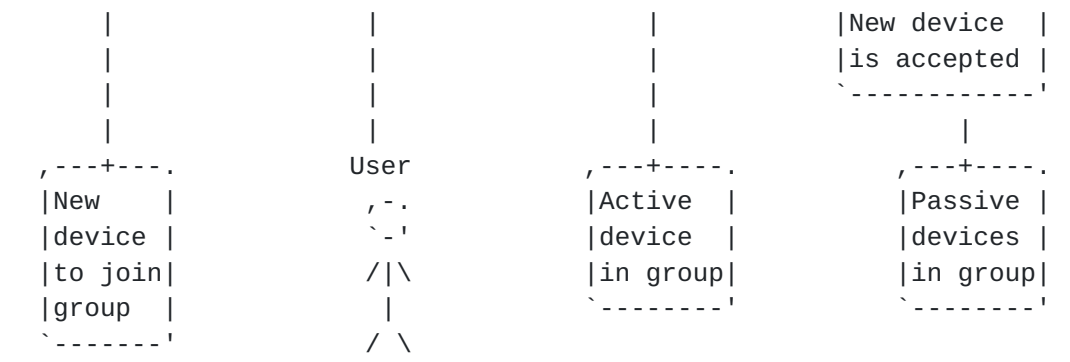
The devices do not form a Device Group. KeySync remains enabled on both devices, and Alice can attempt to form a Device Group again.

2.2.2. Add New Device to Existing Device Group

2.2.2.1. Successful Case




Handshake (user comparison of Trustwords) successful
User presses 'Accept' button on a device in group
7. Accept
8. GroupTrust ThisKey
9. CommitAcceptForGroup
User presses 'Accept' button on new device
10. Accept
11. CommitAccept
12. GroupKeysForNewMember (key data)
New device is grouped
13. GroupKeysAndClose (key data)
New device is accepted
13. GroupKeysAndClose (key data)



As depicted above, our User, Alice, intends to add a new device to her existing Device Group.

1. When Alice initializes the pEp KeySync process, the new device sends a Beacon Message.

2. Upon receipt of a Beacon Message from this new, Ungrouped Device, all Grouped Devices in Alice's existing Device Group send a NegotiationRequestGrouped Message to the New Device.

   Note: Messages 2(w) and 2(l) are instances of the same (NegotiationRequestGrouped) Message type sent from the Grouped Devices. Only the first NegotiationRequestGrouped Message received by the New Device is acknowledged. In this example, 2(w) (the "winner") is processed, while message 2(l) (the "loser") will be ignored and discarded. The result will be the same, no matter which NegotiationRequestGrouped Message is processed first.

3. Upon receipt of the NegotiationRequestGrouped Message 2(w), the New Device answers with a NegotiationOpen Message to the device that issued the "winning" NegotiationRequestGrouped Message.

4. The New Device displays the Trustwords to Alice.

5. Upon receipt of the NegotiationOpen Message, the "winner" device sends a GroupHandshake Message to the "loser" device(s), in order to activate the Trustwords dialog on all Grouped Devices.

6. All Grouped Devices display the Trustwords to the User.

   Note: Messages 6(w) and 6(l) are instances of the same Action on different devices.

7. Alice compares the Trustwords of all devices and chooses the 'Accept' option on any of the Grouped Devices.

Note 1: The Grouped Device that Alice chooses the 'Accept' option on assumes the role of the Active Device for the remainder of the KeySync process, while the other device(s) in the Device Group are assigned the passive role.

Note 2: Alice may choose 'Accept' on the new device first, in which case sequence of the messages is slightly different (i.e., message 10 is sent before message 7). However, the result will be exactly the same.

8. Once Alice chooses the 'Accept' option, the Active Device sends a GroupTrustThisKey Message to the Passive Device(s) in the existing Device Group.

9. The Active Device also sends a CommitAcceptForGroup Message to the new device. Upon receipt, the New Device waits for Alice to choose 'Accept'.

10. Alice compares the Trustwords on both the New Device and the Active Device, then chooses the 'Accept' option on the new device.

11. Once Alice chooses 'Accept', the New Device sends a CommitAccept Message to the Active Device.

12. Upon receipt of the CommitAccept Message, the Active Device device sends a GroupKeysForNewMember Message to the New Device, along with Alice's local key pairs (private and public keys) for synchronization.

13. The New Device receives the GroupKeysForNewMember Message and saves the received keys combined with its Own Keys. The new device has successfully joined the Device Group.

    The New Device sends a GroupKeysAndClose Message to all devices in the Device Group, along with its own original local key pairs (private and public keys) for synchronization.

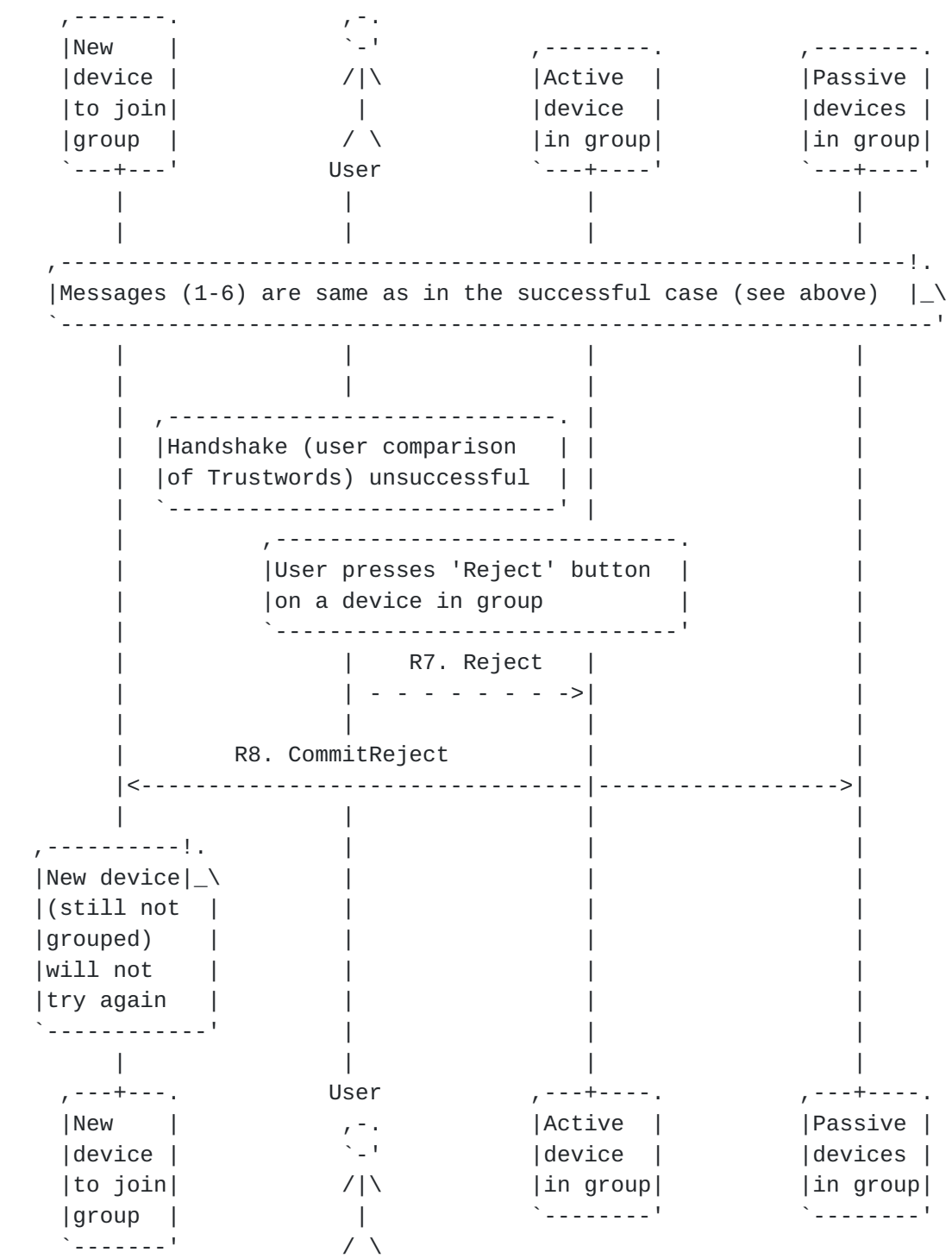
    Note: In the diagram, all messages marked "13. GroupKeysAndClose (key data)" are a single message, but drawn separately in order to convey that the message is sent to all devices in the Device Group.

    Upon receipt of the GroupKeysAndClose Message from the New Device, the Active and Passive Devices save the New Device keys and combine them with their Own Keys. All keys are now synchronized among the devices.

Note: There is no Event Handler to process the GroupKeysAndClose Message explicitly, as all decryptable Messages containing keys are implicitly processed and the received keys saved.

[[TODO: Decide whether the implicit importing keys should rather be replaced by explicit Actions in Event Handlers.]]

2.2.2.2. Unsuccessful Case



For unsuccessful KeySync attempts, messages 1-6 are the same as in a successful attempt (see above), but once the Trustwords are shown, events are as follows:

R7. Our User, Alice, compares the Trustwords displayed on both devices. If the Trustwords do not match, she chooses the 'Reject' option on one of the Grouped Devices (which becomes the Active Device).

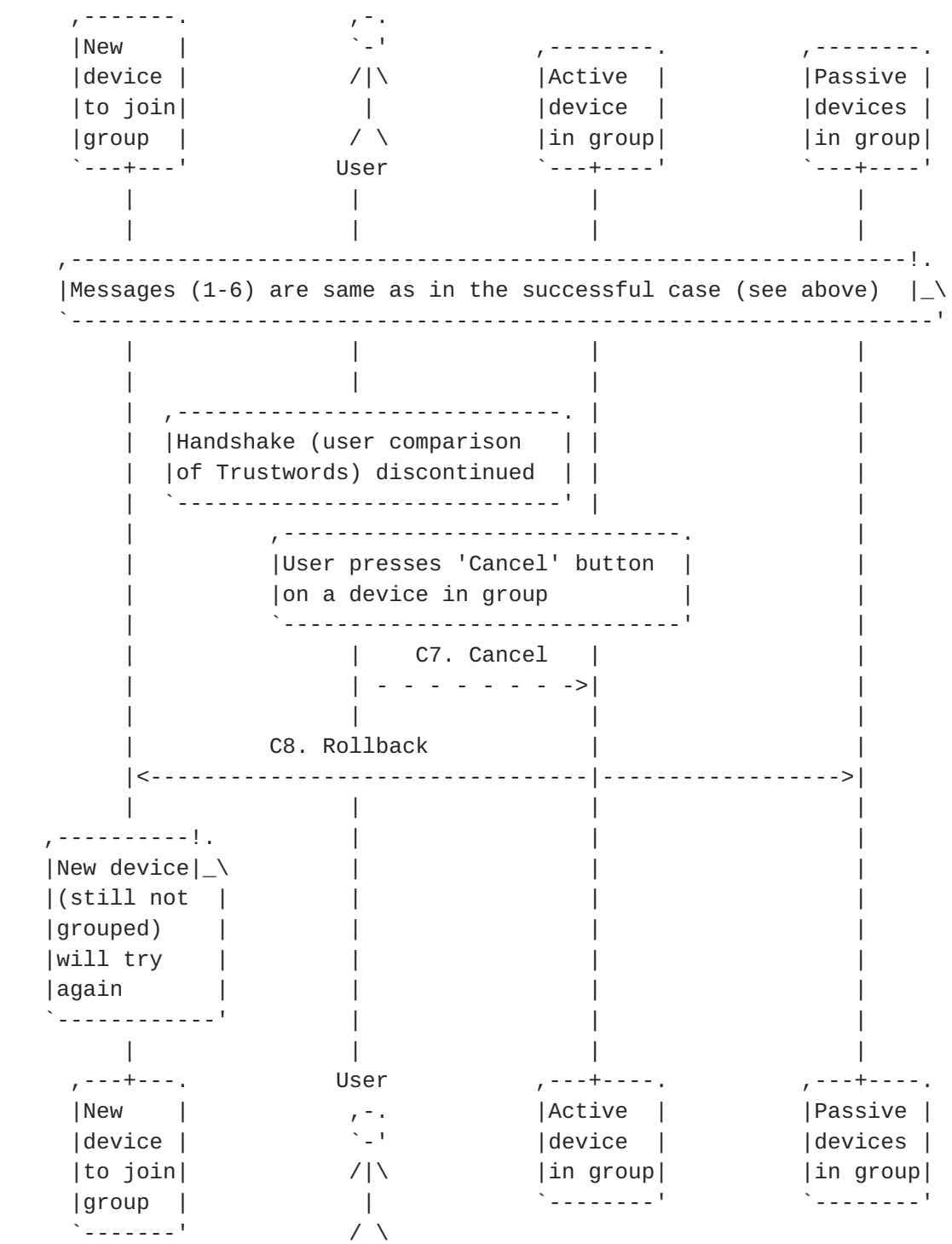
Note: Alice may choose 'Reject' on the new device, in which case the origin and/or destination of the messages change. However, the result will be exactly the same.

R8. Upon receipt of the 'Reject' Event, the Active Device sends a CommitReject Message to both the New Device which attempted to join, and the Passive Device(s) in the Device Group.

Note: In the diagram, "R8. CommitReject" represents the message that is sent to all devices participating in the handshake.

Once the CommitReject Message is sent and received by the respective devices, they cannot form a Device Group, and pEp KeySync is disabled on the New Device. pEp KeySync may be re-enabled in the pEp settings on the affected device.

2.2.2.3. Discontinuation Case



For discontinued (canceled) KeySync attempts, messages 1-6 are the same as in a successful attempt (see above), but once the Trustwords are shown, events are as follows:

C7. Our User, Alice, decides to discontinue the process and chooses the 'Cancel' option on one of the Grouped Devices (which becomes the Active Device).

Note: Alice may choose 'Cancel' on the New Device, in which case the origin and/or destination of the messages change. However, the result will be the same.

C8. When Alice chooses 'Cancel', the Active Device sends a Rollback Message to both the New Device and any Passive Devices in the Device Group.

Note: In the diagram, all messages marked "C8. Rollback" represents the message that is sent to all devices participating in the handshake.

The new device does not join the Device Group. KeySync remains enabled and joining a Device Group can start again at any time.

2.2.3. Exchange Private Keys

[[TODO]]

2.2.4. Leave Device Group

[[TODO]]

2.2.5. Remove other Device from Device Group

[[TODO]]

3. Security Considerations

[[TODO]]

4. Privacy Considerations

[[TODO]]

5. IANA Considerations

This document has no actions for IANA.

6. Acknowledgments

The authors would like to thank the following people who provided substantial contributions, helpful comments or suggestions for this document: Berna Alp, Claudio Luck, Damian Rutz, Damiano Boppart, Itzel Vazquez Sandoval, Kelly Bristol Krista Bennett, Nana Karlstetter, and Sofia Balicka.

This work was initially created by pEp Foundation, and then reviewed and extended with funding by the Internet Society's Beyond the Net Programme on standardizing pEp. [ISOC.bnet]

7. References

7.1. Normative References

[I-D.pep-general]
Birk, V., Marques, H., and B. Hoeneisen, "pretty Easy privacy (pEp): Privacy by Default", Work in Progress, Internet-Draft, draft-pep-general-02, 16 December 2022, <https://datatracker.ietf.org/doc/html/draft-pep-general-02>.

[I-D.pep-keyreset]
Hoeneisen, B., "pretty Easy privacy (pEp): Key Reset", Work in Progress, Internet-Draft, draft-pep-keyreset-00, 15 December 2022, <https://datatracker.ietf.org/doc/html/draft-pep-keyreset-00>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

7.2. Informative References

[I-D.pep-handshake]
Marques, H. and B. Hoeneisen, "pretty Easy privacy (pEp): Contact and Channel Authentication through Handshake", Work in Progress, Internet-Draft, draft-pep-handshake-00, 16 December 2022, <https://datatracker.ietf.org/doc/html/draft-pep-handshake-00>.

[I-D.pep-trustwords]
Hoeneisen, B. and H. Marques, "IANA Registration of Trustword Lists: Guide, Template and IANA Considerations", Work in Progress, Internet-Draft, draft-pep-trustwords-01, 23 December 2022, <https://datatracker.ietf.org/doc/html/draft-pep-trustwords-01>.

[ISOC.bnet]
Simao, I., "Beyond the Net. 12 Innovative Projects Selected for Beyond the Net Funding. Implementing Privacy via Mass Encryption: Standardizing pretty Easy privacy's protocols", June 2017, <https://www.internetsociety.org/blog/2017/06/12-innovative-projects-selected-for-beyond-the-net-funding/>.

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2", FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007, <https://www.rfc-editor.org/info/rfc4949>.

[RFC7435] Dukhovni, V., "Opportunistic Security: Some Protection Most of the Time", RFC 7435, DOI 10.17487/RFC7435, December 2014, <https://www.rfc-editor.org/info/rfc7435>.

Appendix A. Reference Implementation

[[Note: The full Finite State Machine code can be found in Appendix B.1. This section is not a complete reference at this time. The authors intend to refine this section in future revisions of this document.]]

The pEp KeySync Finite State Machine (FSM) is based on a two-phase commit protocol (2PC) structure. This section describes the States, Conditions, Actions, Events, and Messages which comprise the pEp KeySync FSM, and are intended to allow readers to understand the general functionality and Message flow of the FSM.

States are used to direct Actions, Events, and Messages. States may have timeouts. If a timeout is triggered certain functionality is executed.

Conditions are checks performed to determine a specify different behaviors of the FSM depending on the environment, for example, the content of I/O buffers.

Actions describe internal FSM functionality, and fall into two general types. The first Action type directs the State transitions within the FSM, and the second type drives UI functionality. Actions may call internal functions, which are not further described here.

Events are exchanged both between negotiation partners as well as the pEp core implementation itself to trigger Actions and send Messages.

Messages contain information to ensure the integrity of the KeySync session as well as additional data, depending on the type of Message (cf. Appendix A.1.6).

A.1. Description of Finite State Machine

A full diagram of the implemented pEp KeySync FSM can be found at the following URL:

https://gitea.pep.foundation/pEp.foundation/internet-drafts/raw/branch/master/pep-keysync/figures/keysync_fsm_full.svg

For convenience (better readability), there is also a simplified diagram of the implemented pEp KeySync FSM, which does not contain the transitions that occur when choosing the 'Cancel' or 'Reject' options. The simplified diagram can be found at the following URL:

https://gitea.pep.foundation/pEp.foundation/internet-drafts/raw/branch/master/pep-keysync/figures/keysync_fsm_simplified.svg

The first letter of the terms State, Condition, Action, Event and Message is capitalized, whenever it said term refers to the FSM.

A.1.1. States

The FSM has two types of States:

1. Stable States:

   The FSM of KeySync has two Stable States that do not time out:

   * Sole (cf. Appendix A.1.1.2)

   * Grouped (cf. Appendix A.1.1.10)

2. Transitional States:

   All other States (cf. below) are Transitional States that time out.

A.1.1.1. InitState

On initialization, the FSM enters InitState, which evaluates and determines a device's group status. If the device is detected to belong to a Device Group, it issues a SynchronizeGroupKeys Message to the Grouped Devices (to request an update on the Group Keys), and the FSM transitions to State Grouped (cf. Appendix A.1.2.1). Otherwise, a new Challenge TID is created and sent out inside of a Beacon Message, and the FSM transitions to State Sole.

A.1.1.2. Sole

This is the default FSM State for an Ungrouped Device.

On initialization, this State shows the device as being in the Sole State.

The FSM also listens for Beacons from other devices. Upon receipt of a Beacon Message from another device, the received Challenge TID is compared with the own Challenge. The device with the lower Challenge TID is assigned the 'Requester' role, and the other device is automatically assigned the 'Offerer' role. If a device is determined to be the 'Offerer', it resends the Beacon. If a device is determined to be the 'Requester', it issues a NegotiationRequest Event to the 'Offerer'.

When the 'Offerer' device receives this NegotiationRequest Message, it responds with a NegotiationOpen Message, and the 'Offerer' FSM transitions to State HandshakingOfferer where it awaits the 'Requester' device response.

On receipt of a Grouped device's NegotiationRequestGrouped Message, it responds with a NegotiationOpen Message, and the 'Requester' FSM transitions to State HandshakingToJoin.

On receipt of the 'Offerer' device's NegotiationOpen Message, the 'Requester' FSM transitions to State HandshakingRequester.

In this State, other Events may also be processed, but these Events do not result in a transition to another State.

A.1.1.3. HandshakingOfferer

This State can only be entered by the 'Offerer' device from Sole State.

On initialization, it drives user interface options, including the Trustwords dialog. The User is prompted to compare Trustwords and choose from the following options:

* Accept: The 'Requester' public key used in the Handshake is trusted, and the FSM transitions to State HandshakingPhase1Offerer.

* Reject: A CommitReject Message is sent to the 'Requester' device, pEp KeySync is disabled, and the FSM transitions to State End.

* Cancel: A Rollback Message is sent to the 'Requester' device, and the FSM transitions to State Sole.

If the User selects one of the above options on the 'Requester' device, the 'Requester' FSM sends a response to the 'Offerer' device. When this response is received, the 'Offerer' FSM performs a sameNegotiation Condition on the current negotiation session to verify that the current session has not been disrupted or compromised. If this Condition returns 'true', the FSM proceeds as follows, depending on the Message received:

* CommitAcceptRequester: The 'Requester' FSM transitions to State HandshakingPhase2Offerer.

* CommitReject: pEp KeySync is disabled, and the FSM transitions to State End.

* Rollback: The FSM transitions to State Sole.

A.1.1.4. HandshakingRequester

This State can only be entered by the 'Requester' device from Sole State.

On initialization, it drives user interface options, including the Trustwords dialog. The User is prompted to compare Trustwords, and choose from the following options:

* Accept: The 'Offerer' public key is trusted, a CommitAcceptRequester Message is sent to the 'Offerer' device, and the FSM transitions to State HandshakingPhase1Requester.

* Reject: A CommitReject Message is sent to the 'Offerer' device, pEp KeySync is disabled, and the FSM transitions to State End.

* Cancel: A Rollback Message is sent to the 'Offerer' device, and the FSM transitions to State Sole.

If the User selects the 'Cancel' or the 'Reject' options on the 'Offerer' device, the 'Offerer' FSM sends a response to the 'Requester' device. When this response is received, the 'Requester' FSM performs a sameNegotiation Condition on the current negotiation session to verify that the current session has not been disrupted or compromised. If this Condition returns 'true', the FSM proceeds as follows, depending on the Message received:

* CommitReject: pEp KeySync is disabled, and the FSM transitions to State End.

* Rollback: The FSM transitions to State Sole.

A.1.1.5. HandshakingPhase1Offerer

This State can only be entered by the 'Offerer' device from HandshakingOfferer State.

In this State the FSM awaits and processes the response from a 'Requester' device in State HandshakingRequester. When this response is received, the 'Offerer' FSM performs a sameNegotiation Condition on the current negotiation session to verify that the current session has not been disrupted or compromised. If this Condition returns 'true', the FSM proceeds as follows, depending on the Message received:

* CommitAcceptRequester: A CommitAcceptOfferer Message is sent to the 'Requester' device, and the FSM transitions to State FormingGroupOfferer.

* CommitReject: The 'Requester' public key is mistrusted, pEp KeySync is disabled, and the FSM transitions to State End.

* Rollback: The 'Requester' public key is mistrusted, and the FSM transitions to State Sole.

A.1.1.6. HandshakingPhase1Requester

This State can only be entered by the 'Requester' device from HandshakingRequester State.

In this State the FSM awaits and processes the response from an 'Offerer' device in State HandshakingOfferer or HandshakingPhase2Offerer. When this response is received, the 'Requester' FSM performs a sameNegotiation Condition on the current negotiation session to verify that the current session has not been disrupted or compromised. If this Condition returns 'true', the FSM proceeds as follows, depending on the Message received:

* CommitAcceptOfferer: The FSM prepares the Own Keys on the 'Requester' device for synchronization. The FSM then issues an OwnKeysRequester Message to the 'Offerer', which contains these keys, and transitions to State FormingGroupRequester.

* CommitReject: The 'Offerer' public key is mistrusted, pEp KeySync is disabled, and the FSM transitions to State End.

* Rollback: The 'Offerer' public key is mistrusted, and the FSM transitions to State Sole.

A.1.1.7. HandshakingPhase2Offerer

This State can only be entered by the 'Offerer' device from a HandshakingOfferer State.

In this State the FSM waits for the User's response on the 'Offerer' device. The User is still prompted to compare Trustwords and choose from the following options:

* Accept: The 'Requester' public key used in the Handshake is trusted, a CommitAcceptOfferer Message is issued to the 'Requester', and the FSM transitions to State FormingGroupOfferer.

* Reject: A CommitReject Message is issued to the 'Requester' device, pEp KeySync is disabled, and the FSM transitions to State End.

* Cancel: A Rollback Message is issued to the 'Requester' device, and the FSM transitions to State Sole.

A.1.1.8. FormingGroupOfferer

This State can only be entered by the 'Offerer' device from HandshakingPhase1Offerer or HandshakingPhase2Offerer State.

On initialization, the FSM prepares the Own Keys on the 'Offerer' device for synchronization and makes a backup of these Own Keys. Then it waits for the OwnKeysRequester Message from the 'Requester', which contains the Own Keys and the information about all Own Identities of the 'Requester'.

When this Message is received, the 'Offerer' FSM performs a sameNegotiation Condition on the current negotiation session to verify that the current session has not been disrupted or compromised. If this Condition returns 'true', the FSM saves the 'Requester' keys combined with the 'Offerer' keys in a shared GroupKeys array (saveGroupKeys) and the 'Requester' device keys are

marked as default for those respective Identities (receivedKeysAreDefaultKeys). Then, the FSM prepares the Own Keys on the 'Offerer' device for synchronization. Because the Keys are already set to those of the 'Requester' device, it is taking its former Own Keys and Own Identities from the backup (cf. above). The Offerer sends the OwnKeysOfferer Message (with key material of its Own Keys and Own Identities) to the 'Requester', a UI Event (showGroupCreated) indicates that the Device Group process is complete, and the FSM transitions to State Grouped.

Note: In case the 'Requester' device has transitioned to Sole State due to a Cancel, this OwnKeysOfferer Message will not processed by the 'Requester' device.

In case a (delayed) Cancel arrives (which normally cannot happen), a Rollback Message is issued to the 'Requester' device, and the FSM transitions to State Sole.

In case a (delayed) Rollback Message is received (which normally cannot happen), the FSM transitions to State Sole.

A.1.1.9. FormingGroupRequester

This State can only be entered by the 'Requester' device from a HandshakingPhase1Requester State.

In this State the FSM awaits and processes the Message OwnKeysOfferer from an 'Offerer' device in State HandshakingPhase1Offerer or HandshakingPhase2Offerer.

When this Message is received, the 'Requester' FSM performs a sameNegotiation Condition on the current negotiation session to verify that the current session has not been disrupted or compromised. If this Condition returns 'true', the FSM saves the 'Offerer' keys in a shared GroupKeys array (saveGroupKeys), and prepares the device's Own Keys for synchronization. The 'Requester' device keys are marked as default for those respective Identities (ownKeysAreDefaultKeys). A UI Event (showGroupCreated) indicates that the Device Group process is complete, and the FSM transitions to State Grouped.

In case a (delayed) Cancel arrives (which normally cannot happen), a Rollback Message is issued to the 'Offerer' device, and the FSM transitions to State Sole.

Note: In case the 'Offerer' device has already transitioned to Grouped State, this Rollback Message will not be processed by the 'Offerer' device.

In case a (delayed) Rollback Message is received (which normally cannot happen), the FSM transitions to State Sole.

A.1.1.10. Grouped

This is the default State for any Grouped Device.

On initialization, this State generates a new Challenge TID and shows the device as being in the Grouped State. A UI Event (showBeingInGroup) indicates that the Device is part of a Device Group.

In this State the FSM also listens for Beacons from other devices that are not yet part of the Device Group.

Upon receipt of a Beacon Message from Sole Device, the device sends a NegotiationRequestGrouped Message and waits for the Sole Device to respond with a NegotiationOpen Message.

On receipt of the NegotiationOpen Message from the Sole Device, the FSM of the Grouped Device stores the negotiation information and transitions to State HandshakingGrouped.

If the User requests to leave the device group, LeaveDeviceGroup is triggered, i.e., an InitUnledGroupKeyReset Message is issued to the other Device Group members, Sync is disabled locally, and a resetOwnKeysUngrouped is performed (KeyReset on all Own Keys) .

Upon receipt of an InitUnledGroupKeyReset Message from another member of the Device Group, useOwnResponse is performed (save the response into the I/O Buffer), an ElectGroupKeyResetLeader Message is issued, and the FSM transitions to State GroupKeyResetElection.

In this State, other Events may also be processed, but these Events do not result in a transition to another State, e.g., GroupKeysUpdate.

A.1.1.11. HandshakingToJoin

This State can only be entered by a device in the Sole State that is attempting to join an existing Device Group.

On initialization, this State drives user interface options, including the Trustwords dialog for joining a Device Group. The User on the new device is prompted to compare Trustwords and choose from the following options:

* Accept: The existing Device Group's public key used in the Handshake is trusted, and the FSM transitions to State HandshakingToJoinPhase1.

* Reject: A CommitReject Message is sent to the existing Device Group, pEp KeySync is disabled (on new device), and the FSM transitions to State End.

* Cancel: A Rollback Message is sent to the existing Device Group, and the FSM transitions to State Sole.

If the User selects one of the above options on a device that is part of the existing Device Group, its FSM sends a response to the new device. When this response is received, the FSM of the new device performs a sameNegotiation Condition on the current negotiation session to verify that the current session has not been disrupted or compromised. If this Condition returns 'true', the FSM proceeds as follows, depending on the Message received:

* CommitAcceptForGroup: The FSM of the new device transitions to State HandshakingToJoinPhase2.

* CommitReject: pEp KeySync is disabled (on the new device), and the FSM transitions to State End.

* Rollback: The FSM transitions to State Sole.

A.1.1.12. HandshakingToJoinPhase1

This State is entered by a new device only, i.e., a device that is not yet part of a Device Group.

In this State the FSM awaits and processes the response from a device that is part of the existing Device Group. When this response is received, the FSM of the new device performs a sameNegotiation Condition on the current negotiation session to verify that the current session has not been disrupted or compromised. If this Condition returns 'true', the FSM proceeds as follows, depending on the Message received:

* CommitAcceptForGroup: A CommitAccept Message is sent to the existing Device Group, and the The FSM transitions to State JoiningGroup.

* CommitReject: The existing Device Group's public key is mistrusted, pEp KeySync is disabled (on the new device), and the FSM transitions to State End.

* Rollback: The existing Device Group's public key is mistrusted, and the FSM transitions to State Sole.

A.1.1.13. HandshakingToJoinPhase2

This State is entered by a new device only, i.e., a device that is not yet part of a Device Group.

In this State the FSM waits for the User's response on the new device. The User is still prompted to compare Trustwords and choose from the following options:

* Accept: The existing Device Groups's public key used in the Handshake is trusted, a CommitAccept Message is issued to the 'Requester', and the FSM transitions to State JoiningGroup.

* Reject: A CommitReject Message is issued to the exiting Device Group, pEp KeySync is disabled (on the new device), and the FSM transitions to State End.

* Cancel: A Rollback Message is issued to the existing Device Group, and the FSM transitions to State Sole.

A.1.1.14. JoiningGroup

This State is entered by a new device only, i.e., a device that is not yet part of a Device Group.

On initialization, the FSM prepares the Own Keys on the new device for synchronization and makes a backup of these Own Keys. Then it waits for the OwnKeysForNewMember Message from the exiting Device Group, which contains the Own Keys and the information about all Own Identities of the existing Device Group.

When this Message is received, the FSM of the new device performs a sameNegotiationAndPartner Condition on the current negotiation session to verify that both the current session and negotiation partner have not been disrupted or compromised. If this Condition returns 'true', the FSM saves the 'Requester' keys combined with the keys of the existing group in a shared GroupKeys array (saveGroupKeys) and the Device Group's keys are marked as default for those respective Identities (receivedKeysAreDefaultKeys). Then, the FSM prepares the Own Keys on the new device for synchronization. Because the Keys are already set to the ones of the existing Device Group, it is taking its former Own Keys and Own Identities from the backup (cf. above). The new device sends the GroupKeysAndClose Message (with key material of its Own Keys and Own Identities) to the Device Group, a UI Event (showDeviceAdded) indicates that the join Device Group process is complete, and the FSM transitions to State Grouped.

A.1.1.15. HandshakingGrouped

This State is entered by Grouped Devices only, i.e., devices that are part of a Device Group.

On initialization, this State drives UI options, including the Trustwords dialog. The User is prompted to compare Trustwords, and choose from the following options on any device belonging to the existing Device Group:

* Accept: The new device's public key is trusted, and the FSM transitions to State HandshakingGroupedPhase1.

* Reject: A CommitReject Message is sent to the new device and the FSM transitions to State Grouped.

* Cancel: A Rollback Message is sent to the new device, and the FSM transitions to State Grouped.

If the User selects the 'Cancel' or the 'Reject' options on the new device, the new device's FSM sends a response to the existing Device Group. Whenever this response is received by a Grouped Device, the FSM performs a sameNegotiation Condition on the current negotiation session to verify that the current session has not been disrupted or compromised. If this Condition returns 'true', the FSM proceeds as follows, depending on the Message received:

* CommitReject: The FSM transitions to State Grouped.

* Rollback: The FSM transitions to State Grouped.

When a GroupTrustThisKey Message is received from another device group member, the key received along with this Message is trusted. If the sameNegotiation Condition returns 'true', the FSM transitions to State Grouped. This latter causes any device in a Device Group, which is not actively taking part in the joining process, to abort the User prompt to compare the Trustwords.

Note: In this State, other Events are processed, but these Events do not result in a transition to another State and are not discussed here.

A.1.1.16. HandshakingGroupedPhase1

This State is entered by Grouped Devices only, i.e., devices that are already part of a Device Group.

On initialization, a Message GroupTrustThisKey is sent to the other members of the Device Group and a Message CommitAcceptForGroup is sent to the new device.

In this State the FSM awaits and processes the response from an new device in State HandshakingToJoin or HandshakingToJoinPhase2. When this response is received, the Grouped Device's FSM performs a sameNegotiation Condition on the current negotiation session to verify that the current session has not been disrupted or compromised. If this Condition returns 'true', the FSM proceeds as follows, depending on the Message received:

* CommitAccept: The FSM prepares the Own Keys on the Grouped Device for synchronization. The FSM then issues a SendGroupKeysForNewMember Message to the new device, which contains these keys. Then a UI Event (showDeviceAccepted) indicates that the new device has been successfully added to the Device Group, and the FSM transitions to State Grouped. [[TODO: Check whether 'go Grouped' should be removed in this Event Handler]]

* CommitReject: The 'Offerer' public key is mistrusted and the FSM transitions to State Grouped.

* Rollback: The 'Offerer' public key is mistrusted, and the FSM transitions to State Grouped.

In case a GroupKeysAndClose Message arrives from another group member, the FSM transitions to State Grouped.

In this State also various other Events are processed, which do not result in a transition to another State.

A.1.1.17. GroupKeyResetElection

This State is entered by Grouped Devices only, i.e., devices that are already part of a Device Group (normally at reception of an InitUnledGroupKeyReset Message). It is used to determine the "leader" for a KeyReset, to avoid multiple executions of KeyReset. Whichever device from a Device Group sends this Message first, will be the "leader" of the KeyReset.

The device waits for the ElectGroupKeyResetLeader Message from any member of the Device Group (that is, including its own ElectGroupKeyResetLeader Message).

If this Message is received, the FSM of the new device performs a sameResponse Condition to determine whether or not the Message was sent by its own device or another Device Group member. If the Message was sent by its own device, a resetOwnGroupedKeys Action is triggered, and the FSM transitions to State Grouped. If the Message was sent by another Device Group member, the FSM just transitions to State Grouped.

Note: All other ElectGroupKeyResetLeader Messages will be ignored, once the FSM is back to State Grouped.

A.1.2. Conditions

Conditions are implemented with the keyword 'condition'. The code of their implementations can contain all elements, which can be contained by the code of Event Handlers (cf. Appendix A.1.5.1), too. All Conditions can either yield 'true' or 'false' on successful execution, or, if the Condition fails, the FSM is brought into an error state and reinitialized.

A.1.2.1. deviceGrouped

The 'deviceGrouped' Condition evaluates 'true' if a device is already in a Device Group. This is determined by checking if there are Group Keys already. This boolean value is available and eventually altered locally on every KeySync-enabled device. For example, in the reference implementation, this boolean value is stored in a local SQL database.

The 'deviceGrouped' value is what the KeySync FSM uses upon initialization (in InitState) to determine whether a device should transition to State Sole or State Grouped.

A.1.2.2. fromGroupMember

The 'fromGroupMember' Condition evaluates 'true' if the incoming Sync Message is coming from a Device Group member. This is used for "double checking".

A.1.2.3. keyElectionWon

The 'keyElectionWon' Condition evaluates 'true', if the fingerprint (FPR) of the Sender Key of the partner is greater than the FPR of our Default Key for the Account, which is being used as Active Transport. In this case our Own Keys are going to be used as Group Keys. Otherwise, it evaluates 'false' and the Own Keys of the partner will be the Group Keys.

A.1.2.4. sameChallenge

The 'sameChallenge' Condition evaluates 'true' if the Challenge of the incoming Sync Message is identical to the Challenge of the Device, i.e., this is a Sync Message that was sent by the device itself.

A.1.2.5. sameNegotiation

The 'sameNegotiation' Condition is dependent upon the 'storeNegotiation' Action, which stores the active negotiation session while the KeySync process is performed. This Condition evaluates 'true' if the 'storeNegotiation' value of the incoming Sync Message is identical to that of the 'storeNegotiation' value that the device is in, i.e. the incoming Sync Message is part of the same Negotiation.

This serves as a session fidelity check. If this boolean evaluates 'true', it confirms that the pEp KeySync session in progress is the same throughout.

A.1.2.6. sameNegotiationAndPartner

Similar to the 'sameNegotiation' Condition, the 'sameNegotiationAndPartner' Condition is dependent upon the 'storeNegotiation' Action, which stores the active negotiation session while the KeySync process is performed. The 'sameNegotiation' Condition evaluates 'true' if both 'storeNegotiation' value of the incoming Sync Message is identical to that of the 'storeNegotiation' value that the Device is in, AND the negotiation partner did not change.

This Condition also serves as a session fidelity check. If this boolean evaluates 'true', it confirms that the pEp KeySync session in progress is the same throughout, and that the negotiation partner has not changed.

A.1.2.7. sameResponse

The 'sameResponse' Condition evaluates 'true' if the Response of the incoming Sync Message is identical to the Response of the device. In this case the Response was correctly echoed.

A.1.2.8. weAreOfferer

The 'weAreOfferer' Condition evaluates 'true' if the Challenge of the incoming Sync Message is greater than the Challenge of the device. Otherwise we are the Requester and the Condition evaluates 'false'.

A.1.3. Actions

Actions are implemented with the keyword 'action'. Actions are unconditionally executing the code of their implementation. Any or all Actions may fail. In the event of failure, Actions bring the FSM into an error state, and the FSM will be reinitialized.

A.1.3.1. backupOwnKeys

The 'backupOwnKeys' Action is to make a backup of all Own Keys, and allows for restoration of the Own Keys.

A.1.3.2. disable

The 'disable' Action does as it implies. This Action shuts down the FSM and disables KeySync functionality on the impacted device. It is most commonly called in 'Reject' scenarios. For example, if a User rejects a pEp Handshake on a device involved in a pEp Handshake, the 'disable' Action is called. Invoking the 'disable' Action results in the FSM transitioning to State End, which automatically disables the KeySync feature.

Note: pEp KeySync can be manually re-enabled in the pEp settings on the disabled device.

A.1.3.3. newChallengeAndNegotiationBase

The 'newChallengeAndNegotiationBase' Action is to randomly compute a new Challenge and a new Response (Negotiation Base). Both are copied into the I/O Buffer.

The 'newChallengeAndNegotiationBase' Action is invoked by a device during an Init Event in either the Sole or Grouped State, and serves to clear and generate a new Challenge TID and negotiation state.

A.1.3.4. openNegotiation

The 'openNegotiation' Action clears Key and Identity of the partner and calculates the Negotiation ID from the Negotiation Base and the Challenge of the partner (by XOR).

An 'openNegotiation' Action is carried out either by a Sole Device in the 'Requester' role, or a Grouped device upon receipt of a Beacon Message from another Sole Device. Most importantly, this Action ensures that the own TID and the Challenge TID of the Sole Device get combined by the mathematical XOR function. In this way, a common TID exists which can be used by both devices a User wishes to pair. This TID is crucial in allowing the devices to recognize themselves in a particular pairing process, as multiple pairing processes can occur simultaneously.

A.1.3.5. ownKeysAreDefaultKeys

The 'ownKeysAreDefaultKeys' Action is to flag Default Keys of Own Identities as Group Keys.

The ownKeysAreDefaultKeys Action is invoked by the 'Requester' device during the final step of Device Group formation between two Sole devices, and ensures that the Own Keys for the Identities on the 'Requester' device are set as the default for those respective Identities.

A.1.3.6. prepareOwnKeys

The 'prepareOwnKeys' Action is to write a list of Own Identities into the I/O Buffer and load the list of Own Keys into the device state.

The prepareOwnKeys Action is invoked during the latter phases of the KeySync protocol for both new and existing Device Group joining processes. This Action indicates to a device that all key information that has been selected for synchronization should be prepared for sending to the other negotiation partner.

A.1.3.7. prepareOwnKeysFromBackup

The 'prepareOwnKeysFromBackup' Action is to restore the formerly backed up Own Keys (cf. Appendix A.1.3.1) into the I/O Buffer. This Action is similar to prepareOwnKeys (cf. Appendix A.1.3.6).

A.1.3.8. receivedKeysAreDefaultKeys

The 'receivedKeysAreDefaultKeys' Action is to set the received Own Keys as Default Keys for the Own Identities.

A.1.3.9. resetOwnGroupedKeys

The 'resetOwnGroupedKeys' Action is to carry out a KeyReset on Own Group Keys (cf. [I-D.pep-keyreset]).

A.1.3.10. resetOwnKeysUngrouped

The 'resetOwnKeysUngrouped' Action is to carry out a KeyReset on all Own Keys (cf. [I-D.pep-keyreset]).

A.1.3.11. saveGroupKeys

The 'saveGroupKeys' Action is to load Own Identities from the I/O Buffer and store them as Own Identities.

The 'saveGroupKeys' Action directs the addition of any keys received during a KeySync process to a GroupKeys array, along with any existing Own or Grouped Device Keys.

A.1.3.12. showBeingInGroup

The 'showBeingInGroup' Action is to signal to the application that the device is member of a Device Group.

The showBeingInGroup Action in State Grouped drives a UI Event that can be used to notify a pEp User that their device belongs to a Device Group.

A.1.3.13. showBeingSole

The 'showBeingSole' Action is to signal to the application that the device is not member of a Device Group.

The 'showBeingSole' Action in State Sole drives a UI Event that can be used to notify a pEp User that its device is Sole (ungrouped).

A.1.3.14. showDeviceAccepted

The 'showDeviceAccepted' Action is to signal to the application that the device has been accepted as member of the Device Group.

The 'showDeviceAccepted' Action drives a UI Event that is used to notify a pEp User that a Sole Device was accepted as member of an existing Device Group.

A.1.3.15. showDeviceAdded

The 'showDeviceAdded' Action is to signal to the application that the device has been added as member of the Device Group.

The 'showDeviceAdded' Action drives a UI Event that is used to notify a pEp User that a Sole Device was added to an already existing Device Group.

A.1.3.16. showGroupCreated

The 'showGroupCreated' Action is to signal to the application that the Device Group has been created.

In either role that a Sole Device can assume ('Requester' or 'Offerer'), the Action 'showGroupCreated' drives a UI Event which notifies a User that a new Device Group was formed from two Sole Devices.

A.1.3.17. showGroupedHandshake

The 'showGroupedHandshake' Action is to signal to the application of a Grouped Device that a new device is about to join that Device Group.

The 'showGroupedHandshake' Action drives a UI Event on a Grouped device, which a pEp implementer uses to display a pEp Handshake dialog. This dialog indicates that there is a new Sole Device that is requesting to join the Device Group that this Grouped Device belongs to.

A.1.3.18. showJoinGroupHandshake

The 'showJoinGroupHandshake' Action is to signal to the application of an Ungrouped Device that it is about to join an existing Device Group.

The 'showJoinGroupHandshake' Action drives a UI Event on a Sole Device attempting to join an existing Device Group, and is used by pEp implementers to show a Handshake dialog on the Sole Device.

A.1.3.19. showSoleHandshake

The 'showSoleHandshake' Action is to signal to the application of an Ungrouped Device that it is about to from a new Device Group.

For cases where two Sole Devices are attempting to form a new Device Group, the showSoleHandshake Action drives a UI Event, which a pEp implementer uses to display a pEp Handshake dialog to each of the devices in negotiation.

A.1.3.20. storeNegotiation

The 'storeNegotiation' Action is to store the Negotiation for the device in the I/O Buffer. Both, the Sender FPR and partner's Identity are stored for later comparison.

The storeNegotiation Action saves the received non-own negotiation information, which is used, e.g., by the sameNegotiation Condition to perform a session fidelity check (cf. Appendix A.1.2.5).

A.1.3.21. storeThisKey

The 'storeThisKey' Action is to load the Sender Key of the partner from the I/O Buffer and store it for later use.

A.1.3.22. tellWeAreGrouped

The 'tellWeAreGrouped' Action is to set the is_grouped Field in the I/O Buffer to 'true'.

The tellWeAreGrouped Action is used by devices already in the Grouped State. It is sent in a Beacon and indicates to Sole Devices that they are entering a negotiation with a Grouped Device. For the Sole Device, receiving this Action determines which State the FSM will transition to next.

A.1.3.23. tellWeAreNotGrouped

The 'tellWeAreNotGrouped' Action is to set the is_grouped Field in the I/O Buffer to 'false'.

The 'tellWeAreNotGrouped' Action is used by Sole Devices which are assigned the role of 'Requester' upon Challenge TID comparison, and is sent along with a NegotiationRequest Event to indicate to the 'Offerer' device that a negotiation request with a Sole Device is being entered.

A.1.3.24. trustThisKey

The 'trustThisKey' Action applies trust to the stored Key of the negotiation partner and loads this Key into the I/O Buffer.

The trustThisKey Action is executed in all States when a User chooses 'Accept' on the Handshake dialog. Trust for the public key from the negotiation partner is granted so the rest of the KeySync process can be conducted securely and authenticated. The trust also extends to the private key portion of the key pair at a later stage in the KeySync process, i.e., after the User has chosen 'Accept' on both devices. If the process is canceled or rejected at any point after the public key trust has been granted, that trust will be removed (cf. Appendix A.1.3.25).

A.1.3.25. untrustThisKey

The 'untrustThisKey' Action is to revoke trust from the formerly stored Key of the partner and clear the Key in the I/O Buffer.

If the 'Cancel' or 'Reject' options are chosen at any point during the KeySync process after a negotiation partner's public key has been trusted, trust on that public key is removed (cf. Appendix A.1.3.24). The 'untrustThisKey' Action ensures that the negotiation partner's public key can never be attached to Messages sent to any outside peers from the recipient's device.

A.1.3.26. useOwnChallenge

The 'useOwnChallenge' Action is to copy the Challenge of the device into the I/O Buffer.

Once a Beacon is received by a device in either the Sole or Grouped State, the 'useOwnChallenge' Action attaches the device's generated Challenge TID to an outgoing Beacon or NegotiationRequest Event for comparison and session verification purposes.

A.1.3.27. useOwnResponse

The 'useOwnResponse' Action is to copy the Response of the device into the I/O Buffer.

A.1.3.28. useThisKey

The 'useThisKey' Action is to copy the stored Sender Key of the partner into the I/O Buffer.

A.1.4. Transitions

Transitions are changes between States within the FSM, and are indicated by the 'go' command in an Event Handler.

Example:

```
    on Init {
      if deviceGrouped {
          send SynchronizeGroupKeys;
          go Grouped;
      }
      do newChallengeAndNegotiationBase;
      debug > initial Beacon
      send Beacon;
      go Sole;
  }
```

In this example there are two Transitions, one to State Grouped and one to State Sole:

Please see the desired State (Appendix A.1.1 and Appendix B.1) for additional information on why and when these changes are triggered.

A.1.5. Events

While in a State, Events receive incoming Messages and prompt the execution of any Event Handlers contained within (cf. Appendix A.1.5.1).

A.1.5.1. Event Handlers

Event Handlers are code sections (containing Conditions, Actions, Messages, or transitions) executed on receiving an Event. Please refer to the desired State (Appendix B.1) for additional information on specific Event Handlers.

A.1.5.2. Init Event

When the FSM transitions to a new State for the first time, the Init Event (if present) is called. Init Events typically drive UI actions and Event Handlers associated with core functionality of the protocol. All States may have a handler for an Init Event (including the InitState).

Example of an Init Event Handler:

```
  on Init {
      if deviceGrouped {
          send SynchronizeGroupKeys;
          go Grouped;
      }
      do newChallengeAndNegotiationBase;
      debug > initial Beacon
      send Beacon;
      go Sole;
  }
```

A.1.5.3. Message Event

If a Sync Message (cf. Appendix A.1.6) arrives through the network then the Event with the name of the Message occurs.

Example of a Message Event Handler:

In this example an Event Handler is defined, which is executed when a Beacon Message arrives:

```
  on Beacon {
      do openNegotiation;
      do tellWeAreGrouped;
      do useOwnResponse;
      send NegotiationRequestGrouped;
      do useOwnChallenge;
  }
```

A.1.5.4. Signaled Events

Events that are signaled from the core pEp implementation, unless they share their name with a Message.

Example of a Signaled Event Handler:

The KeyGen Event has no corresponding Message. Therefore, it does not occur when a Sync Message arrives, but rather when it is signaled from code:

```
  on KeyGen {
      do prepareOwnKeys;
      send GroupKeysUpdate;
  }
```

A.1.5.5. External Events

External Event may be used to signal a User Interaction to the FSM.

Example:

```
  on Accept {
      do trustThisKey;
      send CommitAcceptRequester;
      go HandshakingPhase1Requester;
  }
```

If Events are part of an API then their IDs must be well-defined. Therefore, it is possible to define such IDs in the FSM.

Example:

```
  external Accept 129;
```

A.1.6. Messages

KeySync is a network protocol, which is implemented using Sync Messages. The Sync Messages for KeySync are defined at the end of the FSM code in Appendix B.1.

Example:

```
message Beacon 2, type=broadcast, ratelimit=10, security=unencrypted {
  field TID challenge;
  auto Version version;
}
```

The wire format of Sync Messages is defined in Abstract Syntax Notation One (ASN.1), cf. Appendix B.2, using Packed Encoding Rules (PER).

Sync Messages are transported inside (e.g., as attachments of) pEp Messages. Hence those are carried by the same Transports, which transmit pEp messages (like, e.g., SMTP and IMAP for email). Some Sync Messages must be sent in copy on all Transports. Others are transported on the Active Transport only. The Active Transport is the transport on which the last Sync Message was received.

A.1.6.1. Message Name and ID

Each Sync Message has a name and an ID.

A.1.6.2. Message Types

There are different types of Messages:

* type=broadcast: Messages, which are meant to be copied on all Transports

* type=anycast: Messages, which are meant to be sent on the Active Transport only (default)

A.1.6.3. Security Context

Each Sync Message has a Security Context. The available Security Contexts are:

* security=unencrypted: send and receive as unencrypted but signed Sync Message

* security=untrusted: only accept when encrypted and signed

* security=trusted (default): only accept when coming over a Trusted Channel and when originating from the Device Group

* security=attach_own_keys_for_new_member: like 'security=trusted' but attach all Own Keys for a new member of the Device Group

* security=attach_own_keys_for_group: like 'security=trusted' but attach all Own Keys for other Device Group members

A.1.6.4. Rate Limit

A Sync Message can have a Rate Limit 'ratelimit=<numeric>'. That means it is only possible to send out one Message each <numeric> second(s). A Rate Limit of 0 means no Rate Limit checking (default).

A.1.6.5. Fields

A Sync Message can have Fields. There are two types of Fields:

1. automatically calculated Fields, defined with the 'auto' keyword, and

2. Fields, which are copied in and out from the I/O Buffer, marked with the 'fields' keyword

The wire format of the Fields is depending on their type. The types are defined in Appendix B.2. Additionally, the two basic types bool (ASN.1: BOOLEAN) and int (ASN.1: INTEGER) are supported.

Example for an 'auto' Field:

```
auto Version version;
```

This Field will be filled with the pEp Sync Protocol version. The Version type is the only automatically calculated type so far.

Example for a Field coming from I/O Buffer:

```
field TID challenge;
```

A.1.6.6. Example

Here an example Message named 'Beacon' with ID=2 (and further attributes) containing 'field' TID and 'auto' Version:

```
message Beacon 2, type=broadcast, ratelimit=10, security=unencrypted {
    field TID challenge;
    auto Version version;
}
```

A.1.6.7. I/O Buffer

There is an I/O Buffer for all Fields which occur in Messages. All Messages share this I/O Buffer. Fields with the same name share one space in the I/O Buffer. Hence, the I/O Buffer is built as superset of all Fields' buffers.

A.1.6.8. Sending

Sending is performed as follows:

1. Calculating all 'auto' Fields and copying the result into the I/O Buffer
2. Loading all Fields of the Message from I/O Buffer
3. Creating a Sync Message
4. Creating a transporting pEp message by attaching the Sync Message using Base Protocol
5. Calling 'messageToSend()' with this pEp message

Example

```
send SynchronizeGroupKeys;
```

A.1.6.9. Receiving

When a Message is being received the field values are being copied into the I/O Buffer and the corresponding Event is being signaled.

A.1.6.10. Messages used in KeySync

In the following, a list of Messages (including format) used by the KeySync FSM as described in Appendix A.1.1 is shown.

A.1.6.10.1. Beacon (ID=2)

Send beacon to everyone on channel.

```
message Beacon 2, type=broadcast, ratelimit=10, security=unencrypted {
    field TID challenge;
    auto Version version;
}
```

A.1.6.10.2. NegotiationRequest (ID=3)

Request negotiation (normally sent after own role has been determined to be the Requester).

```
  message NegotiationRequest 3, security=untrusted {
      field TID challenge;
      field TID response;
      auto Version version;
      field TID negotiation;
      field bool is_group;
  }
```

A.1.6.10.3. NegotiationOpen (ID=4)

Open negotiation (normally sent by Offerer or the device joining a group as a response to 'NegotiationRequest').

```
  message NegotiationOpen 4, security=untrusted {
      field TID response;
      auto Version version;
      field TID negotiation;
  }
```

A.1.6.10.4. Rollback (ID=5)

Rollback the transaction (normally sent after the User has pressed 'Cancel').

```
message Rollback 5, security=untrusted {
    field TID negotiation;
}
```

A.1.6.10.5. CommitReject (ID=6)

Abort the transaction (normally sent after the User has pressed 'Reject').

```
message CommitReject 6, security=untrusted {
    field TID negotiation;
}
```

A.1.6.10.6. CommitAcceptOfferer (ID=7)

Commit the transaction (normally sent by the Offerer after the User has pressed 'Accept' as a response to 'CommitAcceptRequester').

```
message CommitAcceptOfferer 7, security=untrusted {
    field TID negotiation;
}
```

A.1.6.10.7. CommitAcceptRequester (ID=8)

Commit the transaction (normally sent by the Requester after the User has pressed 'Accept').

```
message CommitAcceptRequester 8, security=untrusted {
    field TID negotiation;
}
```

A.1.6.10.8. CommitAccept (ID=9)

Commit the transaction (normally sent by the Sole Device joining a Group, after the User has pressed 'Accept' as a response to 'CommitAcceptGroup').

```
message CommitAccept 9, security=untrusted {
    field TID negotiation;
}
```

A.1.6.10.9. CommitAcceptForGroup (ID=10)

Commit the transaction for the group (normally sent by a Grouped Device after the User has pressed 'Accept').

```
message CommitAcceptForGroup 10, security=untrusted {
    field TID negotiation;
}
```

A.1.6.10.10. GroupTrustThisKey (ID=11)

The whole Device Group can trust this key (normally sent by a Grouped Device to transfer a new key to the other members of the Device Group).

```
message GroupTrustThisKey 11 {
    field Hash key;
    field TID negotiation;
}
```

A.1.6.10.11. GroupKeysForNewMember (ID=12)

Transfer Group Keys and Identities (normally sent by a Grouped Device in reply to a 'CommitAccept' Message from the Sole Device joining the group).

```
message GroupKeysForNewMember 12,
        security=attach_own_keys_for_new_member {
    field IdentityList ownIdentities;
}
```

A.1.6.10.12. GroupKeysAndClose (ID=13)

Transfer Keys and Identities of the new group member (normally sent by the new group member in reply to a 'GroupKeysForNewMember' Message from a Grouped Device).

```
message GroupKeysAndClose 13, security=attach_own_keys_for_group {
    field IdentityList ownIdentities;
}
```

A.1.6.10.13. OwnKeysOfferer (ID=14)

Transfer the Offerer's Keys and Identities (normally sent by the Offerer in reply to a 'OwnKeysRequester' Message from the Requester).

```
message OwnKeysOfferer 14, security=attach_own_keys_for_group {
    field IdentityList ownIdentities;
}
```

A.1.6.10.14. OwnKeysRequester (ID=15)

Transfer the Requester's Keys and Identities (normally sent by the Requester in reply to a 'CommitAcceptOfferer' Message from the Offerer).

```
message OwnKeysRequester 15, security=attach_own_keys_for_new_member {
    field IdentityList ownIdentities;
}
```

A.1.6.10.15. NegotiationRequestGrouped (ID=16)

Request negotiation to join the group (normally sent by a Grouped Device after receiving a 'Beacon' Message from a Sole Device).

```
message NegotiationRequestGrouped 16, security=untrusted {
    field TID challenge;
    field TID response;
    auto Version version;
    field TID negotiation;
    field bool is_group;
}
```

A.1.6.10.16. GroupHandshake (ID=17)

Inform other members of the Device group about a new handshake (normally sent by a Grouped Device after receiving a 'NegotiationOpen' Message from a Sole Device).

```
message GroupHandshake 17 {
    field TID negotiation;
    field Hash key;
}
```

A.1.6.10.17. GroupKeysUpdate (ID=18)

Transfer the Group Keys and Identities (normally sent by a Grouped Device to the other members of the Device Group).

```
message GroupKeysUpdate 18, security=attach_own_keys_for_group {
    field IdentityList ownIdentities;
}
```

A.1.6.10.18. InitUnledGroupKeyReset (ID=19)

Initiate unled group KeyReset, i.e., the initiating device does not perform the KeyReset itself. (This Message is normally sent by a Grouped Device after a 'LeaveDeviceGroup' has been requested by the User.)

Further information on KeyReset can be found in [I-D.pep-keyreset].

```
message InitUnledGroupKeyReset 19 {
}
```

A.1.6.10.19. ElectGroupKeyResetLeader (ID=20)

Initiate determination of "leader" for a KeyReset (normally sent by all Grouped Devices to the all other members of the Device Group in response to an InitUnledGroupKeyReset Message).

Further information can be found in Appendix A.1.1.17 and [I-D.pep-keyreset].

```
message ElectGroupKeyResetLeader 20 {
    field TID response;
}
```

A.1.6.10.20. SynchronizeGroupKeys (ID=21)

Request synchronization of Group Keys (normally sent by a Grouped Device to the other members of the Device Group to trigger a 'GroupKeysUpdate' Message).

```
message SynchronizeGroupKeys 21, ratelimit=60 {
}
```

Appendix B. Code excerpts

B.1. Finite State Machine

Below you can find the code excerpt for the pEp KeySync FSM, including Messages and external Events:

```
// This file is under BSD License 2.0

// Sync protocol for pEp
// Copyright (c) 2016-2020, pEp foundation

// Written by Volker Birk
```

```
include ./fsm.yml2

protocol Sync 1 {
  // all messages have a timestamp,
  // time out and are removed after timeout

  fsm KeySync 1, threshold=300 {
    version 1, 2;

    state InitState {
      on Init {
        if deviceGrouped {
          send SynchronizeGroupKeys;
          go Grouped;
        }
        do newChallengeAndNegotiationBase;
        debug > initial Beacon
        send Beacon;
        go Sole;
      }
    }

    state Sole timeout=off {
      on Init {
        do showBeingSole;
      }

      on KeyGen {
        debug > key generated
        send Beacon;
      }

      on CannotDecrypt {
        debug > cry, baby
        send Beacon;
      }

      on Beacon {
        if sameChallenge {
          debug > this is our own Beacon; ignore
        }
        else {
          if weAreOfferer {
            do useOwnChallenge;
            debug > we are Offerer
            send Beacon;
          }
          else /* we are requester */ {
```

```
            do openNegotiation;
            do tellWeAreNotGrouped;
            // requester is sending NegotiationRequest
            do useOwnResponse;
            send NegotiationRequest;
            do useOwnChallenge;
          }
        }
      }

      // we get this from another sole device
      on NegotiationRequest {
        if sameChallenge { // challenge accepted
          do storeNegotiation;
          // offerer is accepting by confirming NegotiationOpen
          // repeating response is implicit
          send NegotiationOpen;
          go HandshakingOfferer;
        }
      }

      // we get this from an existing device group
      on NegotiationRequestGrouped {
        if sameChallenge { // challenge accepted
          do storeNegotiation;
          // offerer is accepting by confirming NegotiationOpen
          // repeating response is implicit
          send NegotiationOpen;
          go HandshakingToJoin;
        }
      }

      on NegotiationOpen {
        if sameResponse {
          debug > Requester is receiving NegotiationOpen
          do storeNegotiation;
          go HandshakingRequester;
        }
        else {
          debug > cannot approve NegotiationOpen
        }
      }
    }

    // handshaking without existing Device group
    state HandshakingOfferer timeout=600 {
      on Init
        do showSoleHandshake;
```

```
        // Cancel is Rollback
        on Cancel {
          send Rollback;
          go Sole;
        }

        on Rollback {
          if sameNegotiation
            go Sole;
        }

        // Reject is CommitReject
        on Reject {
          send CommitReject;
          do disable;
          go End;
        }

        on CommitReject {
          if sameNegotiation {
            do disable;
            go End;
          }
        }

        // Accept means init Phase1Commit
        on Accept {
          do trustThisKey;
          go HandshakingPhase1Offerer;
        }

        // got a CommitAccept from requester
        on CommitAcceptRequester {
          if sameNegotiation
            go HandshakingPhase2Offerer;
        }
      }

      // handshaking without existing Device group
      state HandshakingRequester timeout=600 {
        on Init
          do showSoleHandshake;

        // Cancel is Rollback
        on Cancel {
          send Rollback;
          go Sole;
        }
```

```
        on Rollback {
          if sameNegotiation
            go Sole;
        }

        // Reject is CommitReject
        on Reject {
          send CommitReject;
          do disable;
          go End;
        }

        on CommitReject {
          if sameNegotiation {
            do disable;
            go End;
          }
        }

        // Accept means init Phase1Commit
        on Accept {
          do trustThisKey;
          send CommitAcceptRequester;
          go HandshakingPhase1Requester;
        }
      }

      state HandshakingPhase1Offerer {
        on Rollback {
          if sameNegotiation {
            do untrustThisKey;
            go Sole;
          }
        }

        on CommitReject {
          if sameNegotiation {
            do untrustThisKey;
            do disable;
            go End;
          }
        }

        on CommitAcceptRequester {
          if sameNegotiation {
            send CommitAcceptOfferer;
            go FormingGroupOfferer;
          }
```

```
        }
      }

      state HandshakingPhase1Requester {
        on Rollback {
          if sameNegotiation {
            do untrustThisKey;
            go Sole;
          }
        }

        on CommitReject {
          if sameNegotiation {
            do untrustThisKey;
            do disable;
            go End;
          }
        }

        on CommitAcceptOfferer {
          if sameNegotiation {
            do prepareOwnKeys;
            send OwnKeysRequester;
            go FormingGroupRequester;
          }
        }
      }

      state HandshakingPhase2Offerer {
        on Cancel {
          send Rollback;
          go Sole;
        }

        on Reject {
          send CommitReject;
          do disable;
          go End;
        }

        on Accept {
          do trustThisKey;
          send CommitAcceptOfferer;
          go FormingGroupOfferer;
        }
      }

      state FormingGroupOfferer {
```

```
      on Init {
        // we need to keep in memory which keys
        // we have before forming a new group
        do prepareOwnKeys;
        do backupOwnKeys;
      }

      on Cancel {
        send Rollback;
        go Sole;
      }

      on Rollback
        go Sole;

      on OwnKeysRequester {
        if sameNegotiationAndPartner {
          do saveGroupKeys;
          do receivedKeysAreDefaultKeys;
          // send the keys we had before forming a new group
          do prepareOwnKeysFromBackup;
          send OwnKeysOfferer;
          do showGroupCreated;
          go Grouped;
        }
      }
    }

    state FormingGroupRequester {
      on Cancel {
        send Rollback;
        go Sole;
      }

      on Rollback
        go Sole;

      on OwnKeysOfferer {
        if sameNegotiation {
          do saveGroupKeys;
          do prepareOwnKeys;
          do ownKeysAreDefaultKeys;
          do showGroupCreated;
          go Grouped;
        }
      }
    }
```

```
       state Grouped timeout=off {
         on Init {
           do newChallengeAndNegotiationBase;
           do showBeingInGroup;
         }

         on CannotDecrypt {
           debug > cry, baby
           send SynchronizeGroupKeys;
         }

         on SynchronizeGroupKeys {
           do prepareOwnKeys;
           send GroupKeysUpdate;
         }

         on GroupKeysUpdate {
           if fromGroupMember // double check
             do saveGroupKeys;
         }

         on KeyGen {
           do prepareOwnKeys;
           send GroupKeysUpdate;
         }

         on Beacon {
           do openNegotiation;
           do tellWeAreGrouped;
           do useOwnResponse;
           send NegotiationRequestGrouped;
           do useOwnChallenge;
         }

         on NegotiationOpen {
           if sameResponse {
             do storeNegotiation;
             do useThisKey;
             send GroupHandshake;
             go HandshakingGrouped;
           }
           else {
             debug > cannot approve NegotiationOpen
           }
         }

         on GroupHandshake {
           do storeNegotiation;
```

```
      do storeThisKey;
      go HandshakingGrouped;
    }

    on GroupTrustThisKey {
      if fromGroupMember // double check
        do trustThisKey;
    }

    on LeaveDeviceGroup {
      send InitUnledGroupKeyReset;
      do disable;
      do resetOwnKeysUngrouped;
    }

    on InitUnledGroupKeyReset {
      debug > unled group key reset; new group keys will be elected
      do useOwnResponse;
      send ElectGroupKeyResetLeader;
      go GroupKeyResetElection;
    }
  }

  state GroupKeyResetElection {
    on ElectGroupKeyResetLeader {
      if sameResponse {
        // the first one is from us, we're leading this
        do resetOwnGroupedKeys;
        go Grouped;
      }
      else {
        // the first one is not from us
        go Grouped;
      }
    }
  }

  // sole device handshaking with group
  state HandshakingToJoin {
    on Init
      do showJoinGroupHandshake;

    // Cancel is Rollback
    on Cancel {
      send Rollback;
      go Sole;
    }
```

```
        on Rollback {
          if sameNegotiation
            go Sole;
        }

        // Reject is CommitReject
        on Reject {
          send CommitReject;
          do disable;
          go End;
        }

        on CommitAcceptForGroup {
          if sameNegotiation
            go HandshakingToJoinPhase2;
        }

        on CommitReject {
          if sameNegotiation {
            do disable;
            go End;
          }
        }

        // Accept is Phase1Commit
        on Accept {
          do trustThisKey;
          go HandshakingToJoinPhase1;
        }
      }

      state HandshakingToJoinPhase1 {
        on Rollback {
          if sameNegotiation {
            do untrustThisKey;
            go Sole;
          }
        }

        on CommitReject {
          if sameNegotiation {
            do untrustThisKey;
            do disable;
            go End;
          }
        }

        on CommitAcceptForGroup {
```

```
          if sameNegotiation {
            send CommitAccept;
            go JoiningGroup;
          }
        }
      }

      state HandshakingToJoinPhase2 {
        on Cancel {
          send Rollback;
          go Sole;
        }

        on Reject {
          send CommitReject;
          do disable;
          go End;
        }

        on Accept {
          do trustThisKey;
          send CommitAccept;
          go JoiningGroup;
        }
      }

      state JoiningGroup {
        on Init {
          // we need to keep in memory which keys
          // we have before joining
          do prepareOwnKeys;
          do backupOwnKeys;
        }
        on GroupKeysForNewMember {
          if sameNegotiationAndPartner {
            do saveGroupKeys;
            do receivedKeysAreDefaultKeys;
            // send the keys we had before joining
            do prepareOwnKeysFromBackup;
            send GroupKeysAndClose;
            do showDeviceAdded;
            go Grouped;
          }
        }
      }

      state HandshakingGrouped {
        on Init
```

```
        do showGroupedHandshake;

      // Cancel is Rollback
      on Cancel {
        send Rollback;
        go Grouped;
      }

      on Rollback {
        if sameNegotiation
          go Grouped;
      }

      // Reject is CommitReject
      on Reject {
        send CommitReject;
        go Grouped;
      }

      on CommitReject {
        if sameNegotiation
          go Grouped;
      }

      // Accept is Phase1Commit
      on Accept {
        do trustThisKey;
        go HandshakingGroupedPhase1;
      }

      on GroupTrustThisKey {
        if fromGroupMember { // double check
          do trustThisKey;
          if sameNegotiation
            go Grouped;
        }
      }

      on GroupKeysUpdate {
        if fromGroupMember // double check
          do saveGroupKeys;
      }
    }

    state HandshakingGroupedPhase1 {
      on Init {
        send GroupTrustThisKey;
        send CommitAcceptForGroup;
```

```
    }

    on Rollback {
      if sameNegotiation {
        do untrustThisKey;
        go Grouped;
      }
    }

    on CommitReject {
      if sameNegotiation {
        do untrustThisKey;
        go Grouped;
      }
    }

    on CommitAccept {
      if sameNegotiation {
        do prepareOwnKeys;
        send GroupKeysForNewMember;
        do showDeviceAccepted;
        go Grouped;
      }
    }

    on GroupTrustThisKey {
      if fromGroupMember // double check
        do trustThisKey;
    }

    on GroupKeysUpdate {
      if fromGroupMember // double check
        do saveGroupKeys;
    }

    on GroupKeysAndClose {
      if fromGroupMember { // double check
        // do not save GroupKeys as default keys; key data is
        // already imported
        go Grouped;
      }
    }
  }

  external Accept 129;
  external Reject 130;
  external Cancel 131;
```

```
// beacons are always broadcasted

message Beacon 2, type=broadcast,
        ratelimit=10, security=unencrypted {
  field TID challenge;
  auto Version version;
}

message NegotiationRequest 3, security=untrusted {
  field TID challenge;
  field TID response;
  auto Version version;
  field TID negotiation;
  field bool is_group;
}

message NegotiationOpen 4, security=untrusted {
  field TID response;
  auto Version version;
  field TID negotiation;
}

message Rollback 5, security=untrusted {
  field TID negotiation;
}

message CommitReject 6, security=untrusted {
  field TID negotiation;
}

message CommitAcceptOfferer 7, security=untrusted {
  field TID negotiation;
}

message CommitAcceptRequester 8, security=untrusted {
  field TID negotiation;
}

message CommitAccept 9, security=untrusted {
  field TID negotiation;
}

message CommitAcceptForGroup 10, security=untrusted {
  field TID negotiation;
}

// default: security=trusted
// messages are only accepted when coming from the device group
```

```
message GroupTrustThisKey 11 {
  field Hash key;
  field TID negotiation;
}

// trust in future
message GroupKeysForNewMember 12,
        security=attach_own_keys_for_new_member {
  field IdentityList ownIdentities;
}

message GroupKeysAndClose 13,
        security=attach_own_keys_for_group {
  field IdentityList ownIdentities;
}

message OwnKeysOfferer 14, security=attach_own_keys_for_group {
  field IdentityList ownIdentities;
}

message OwnKeysRequester 15,
        security=attach_own_keys_for_new_member {
  field IdentityList ownIdentities;
}

// grouped handshake
message NegotiationRequestGrouped 16, security=untrusted {
  field TID challenge;
  field TID response;
  auto Version version;
  field TID negotiation;
  field bool is_group;
}

message GroupHandshake 17 {
  field TID negotiation;
  field Hash key;
}

// update group
message GroupKeysUpdate 18, security=attach_own_keys_for_group {
  field IdentityList ownIdentities;
}

// initiate unled group key reset
message InitUnledGroupKeyReset 19 {
}
```

```
      message ElectGroupKeyResetLeader 20 {
        field TID response;
      }

      message SynchronizeGroupKeys 21, ratelimit=60 {
      }

      [...]
    }

    [...]

}
```

B.2. ASN.1 Type Definitions

Below you can find the ASN.1 Type definitions for the Messages used in pEp KeySync FSM:

```
-- This file is under BSD License 2.0

-- Sync protocol for pEp
-- Copyright (c) 2016-2021 pEp foundation

-- Written by Volker Birk

PEP
    { iso(1) org(3) dod(6) internet(1) private(4)
      enterprise(1) pEp(47878) basic(0) }

DEFINITIONS AUTOMATIC TAGS EXTENSIBILITY IMPLIED ::=

BEGIN

EXPORTS Identity, IdentityList, TID, Hash, Version, Rating, PString,
        PStringList, PStringPair, PStringPairList, ISO639-1;

ISO639-1 ::= PrintableString(FROM ("a".."z")) (SIZE(2))
Hex ::= PrintableString(FROM ("A".."F" | "0".."9"))
Hash ::= Hex(SIZE(16..128)) -- 32bit Key ID to SHA512 in hex
PString ::= UTF8String  (SIZE(0..1024))
PStringList ::= SEQUENCE OF PString
TID ::= OCTET STRING (SIZE(16)) -- UUID version 4 variant 1

Identity ::= SEQUENCE {
    address     PString,
    fpr         Hash,
    user-id     PString,
```

```
        username    PString,
        comm-type   INTEGER (0..255),
        lang        ISO639-1
    }

    IdentityList ::= SEQUENCE OF Identity

    Version ::= SEQUENCE {
        major INTEGER (0..255) DEFAULT 1,
        minor INTEGER (0..255) DEFAULT 2
    }

    Rating ::= ENUMERATED {
        -- no color

        cannot-decrypt (1),
        have-no-key (2),
        unencrypted (3),
        unreliable (5),

        b0rken (-2),

        -- yellow

        reliable (6),

        -- green

        trusted (7),
        trusted-and-anonymized (8),
        fully-anonymous (9),

        -- red

        mistrust (-1),
        under-attack (-3)
    }

    PStringPair ::= SEQUENCE {
        key      PString,
        value    PString
    }

    PStringPairList ::= SEQUENCE OF PStringPair

    END
```

Appendix C. Document Changelog

[[RFC Editor: This section is to be removed before publication]]

* draft-pep-keysync-03:
  - Updated Use Cases 'Leave Device Group' and 'Remove other Device from Device Group'
  - Updated States, Conditions, Actions, Transitions, Events, Messages
  - Updated/Added Term definitions
  - Harmonized capitalization
  - Updated to xml2rfc v3
  - Added venue tags
  - Several minor edits
  - Updated authors' list
* draft-pep-keysync-02:
  - Improve clarity and readability
  - Updated Section 2.1.1
* draft-pep-keysync-01:
  - Updated FSM States, Actions, Messages, Events and interaction diagrams to reflect recent design changes
  - added latest revision of code and ASN.1 Type definitions
* draft-pep-keysync-00:
  - Updated docname and author's section
* draft-hoeneisen-pep-keysync-01:
  - Major rewrite of upper sections
  - Adjust to reflect code changes

- Move Finite State Machine reference and code to Appendices A & B

* draft-hoeneisen-pep-keysync-00:

- Initial version

Appendix D. Open Issues

[[RFC Editor: This section should be empty and is to be removed before publication]]

* Resolve several TODOs / add missing text

Authors' Addresses

Volker Birk
pEp Foundation
Oberer Graben 4
CH- 8400 Winterthur
Switzerland
Email: volker.birk@pep.foundation
URI: https://pep.foundation/

Bernie Hoeneisen
pEp Foundation
Oberer Graben 4
CH- 8400 Winterthur
Switzerland
Email: bernie.hoeneisen@pep.foundation
URI: https://pep.foundation/

Hernani Marques
pEp Foundation
Oberer Graben 4
CH- 8400 Winterthur
Switzerland
Email: hernani.marques@pep.foundation
URI: https://pep.foundation/