Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 15, 2021

V. Birk
B. Hoeneisen
K. Bristol
pEp Foundation
July 14, 2020

pretty Easy privacy (pEp): Key Synchronization Protocol (KeySync) draft-pep-keysync-02

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.

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/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 15, 2021.

Copyright Notice

Table of Contents

1. Introduction

The pretty Easy privacy (pEp) [I-D.birk-pep] 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", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

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.marques-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 scalar-to-word representation of 16-bit numbers (0 to 65535) to natural language words. When doing a Handshake, peers are shown combined Trustwords of both public keys involved to ease the comparison. [I-D.birk-pep-trustwords]

- 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."

- Identity: The combination of a unique user identifier plus a specific address (email, network ID, URI, etc.). A single user may have multiple identities. See also [RFC4949].

- Device Group: All of a user's devices which have successfully completed the KeySync process, and are now configured to share user data, such as cryptographic keys, trust information, calendars, configurations, and other data as a result of that process. 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: A key which is actually used for a given identity.

- Own Key: A Default Key for an 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.

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.birk-pep-trustwords]) Simply put, the user MUST complete this Trustwords dialog 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 protocol structure (2PC). 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 pEp engine 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 their 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 their 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 their 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 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 a 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, 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 all private keys of the Device Group. Alice can manually tell her mobile phone to leave the Device Group by turning

off the pEp Sync feature on her device, which deactivates KeySync. The Device Group is dissolved, and Sync is disabled on her mobile phone. This action also initiates the pEp KeyReset protocol, which resets keys for all own identities.

In the future, if Alice desires, she can re-add her mobile phone to a Device Group, but she will first have to re-enable Sync, and then initiate the joining procedure again (cf. Section 2.1.1 and Section 2.1.2).

2.1.5. Remove other Device from Device Group

One of Alice's devices may be stolen or become otherwise compromised. She needs to ensure that the affected device no longer receives updates to private keys from the other devices in her Device Group. Using one of her remaining Grouped Devices, Alice can disable pEp Sync (and thus KeySync) on her remaining devices. This action dissolves the Device Group and initiates the pEp KeyReset protocol.

2.2. Interaction Diagrams

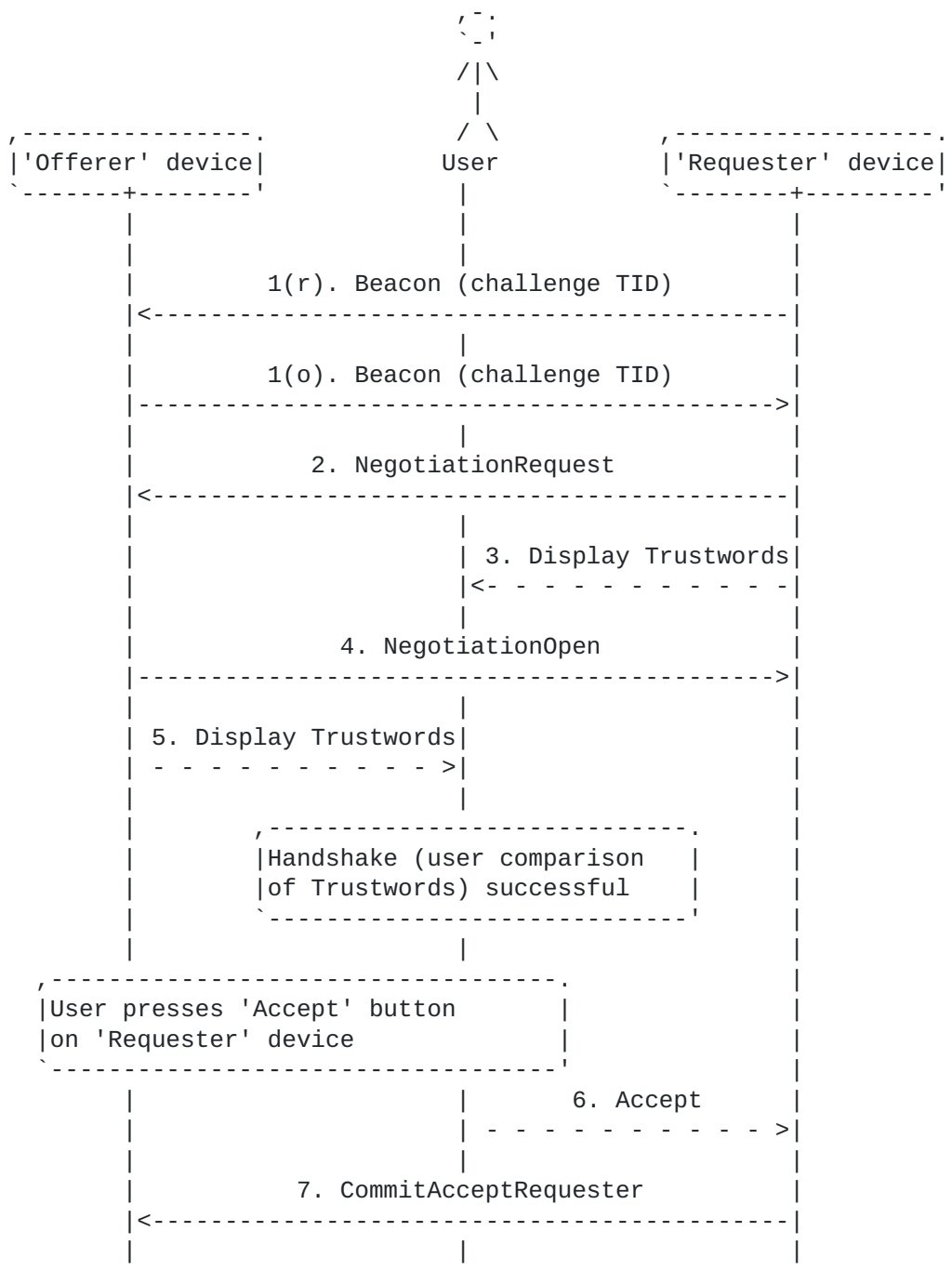
The following interaction diagrams depict what happens during Alice's KeySync scenarios in a simplified manner. For each scenario, we first present a successful case, then an unsuccessful case and, finally, a case that has been interrupted, or discontinued. Some details are skipped here for the sake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

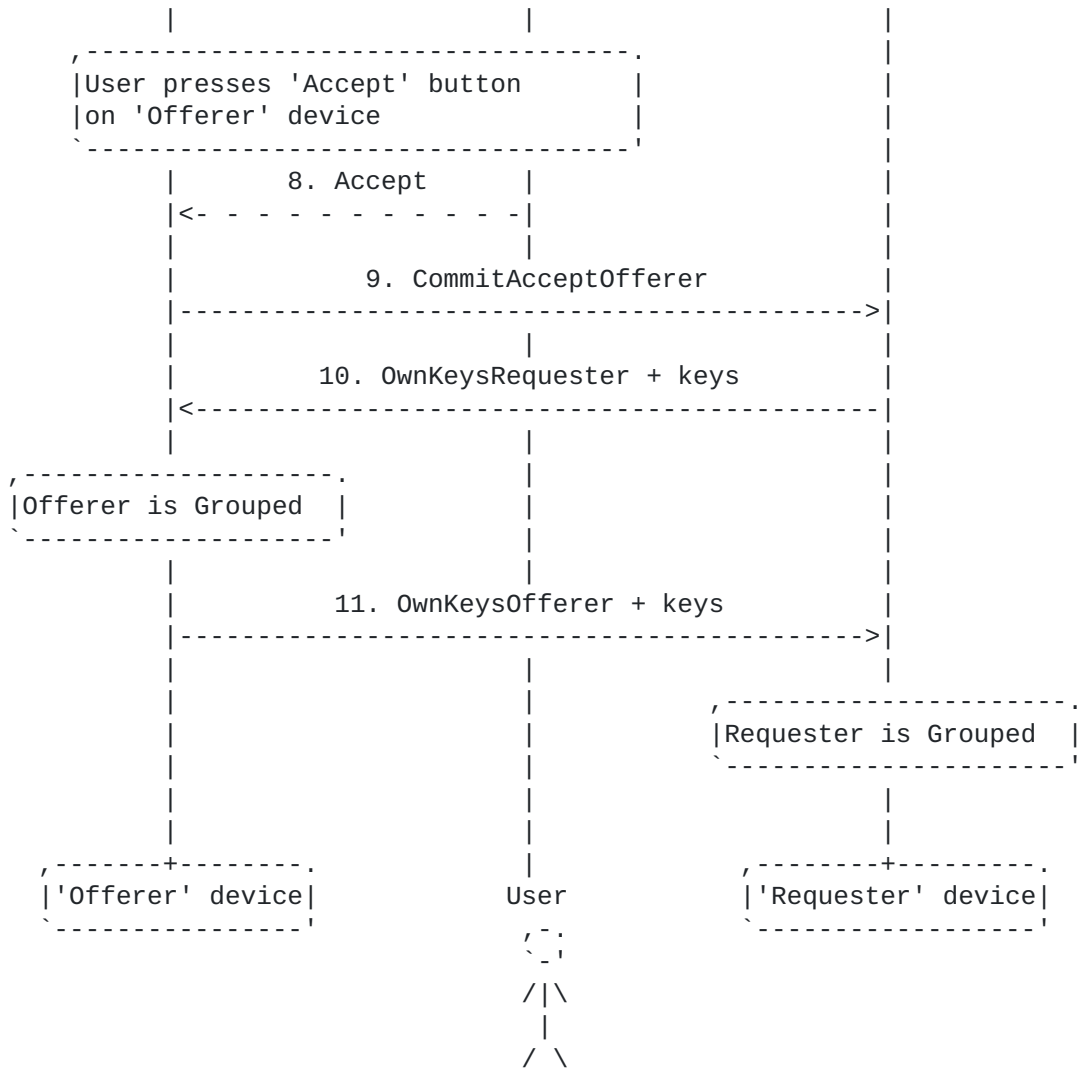
Each pEp-enabled device runs its own Finite State Machine (FSM), which interact with each other throughout the KeySync process, and drive the UI options presented to Alice (the 'User' in all diagrams, unless otherwise noted). All messages are 'broadcast' between devices. The TIDs added to each message 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




User presses 'Accept' button
on 'Offerer' device
8. Accept
9. CommitAcceptOfferer
10. OwnKeysRequester + keys
Offerer is Grouped
11. OwnKeysOfferer + keys
Requester is Grouped
'Offerer' device
User
'Requester' device

As depicted above, our user, Alice, intends to form a Device Group in order to securely share key material between her devices. 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.

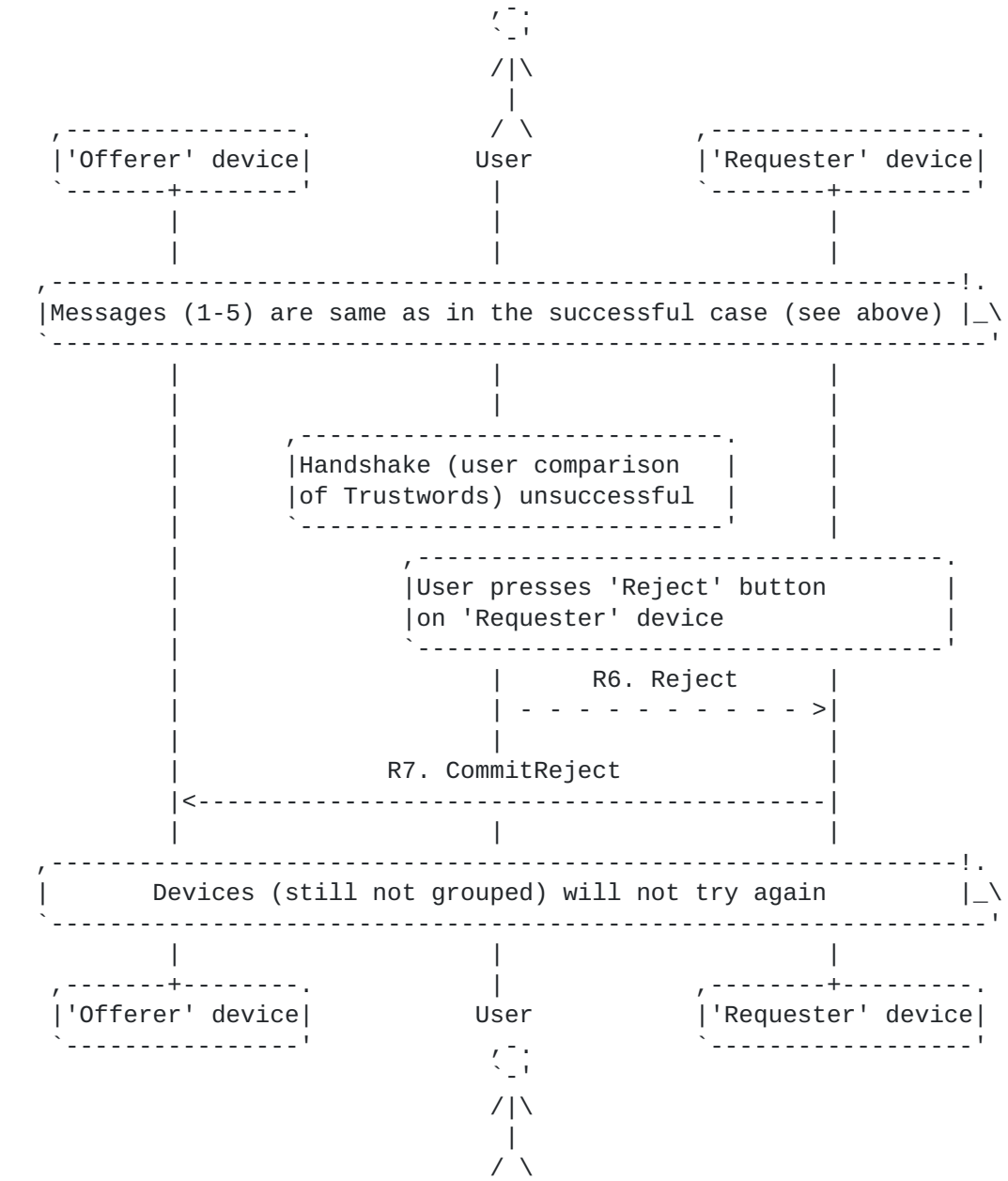
11. Upon receipt of the OwnKeysRequester message, the 'Offerer' device saves the 'Requester' device keys and combines them with the existing 'Offerer' device keys. This means that the 'Offerer' device is grouped.

    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:

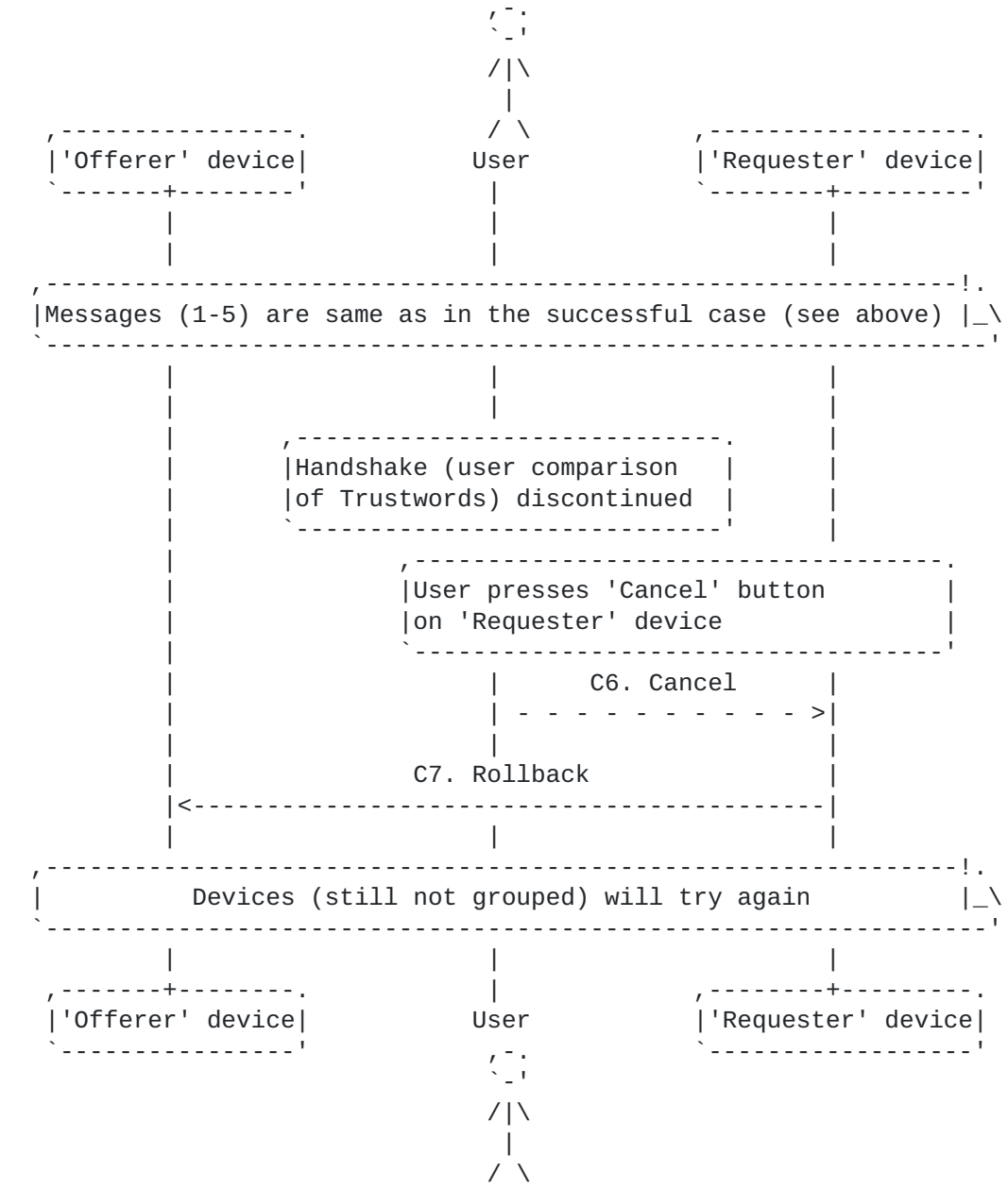
R6. Our user, Alice, compares the Trustwords of both devices. As the Trustwords do not match, she chooses the 'Reject' option on the 'Requester' device.

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.

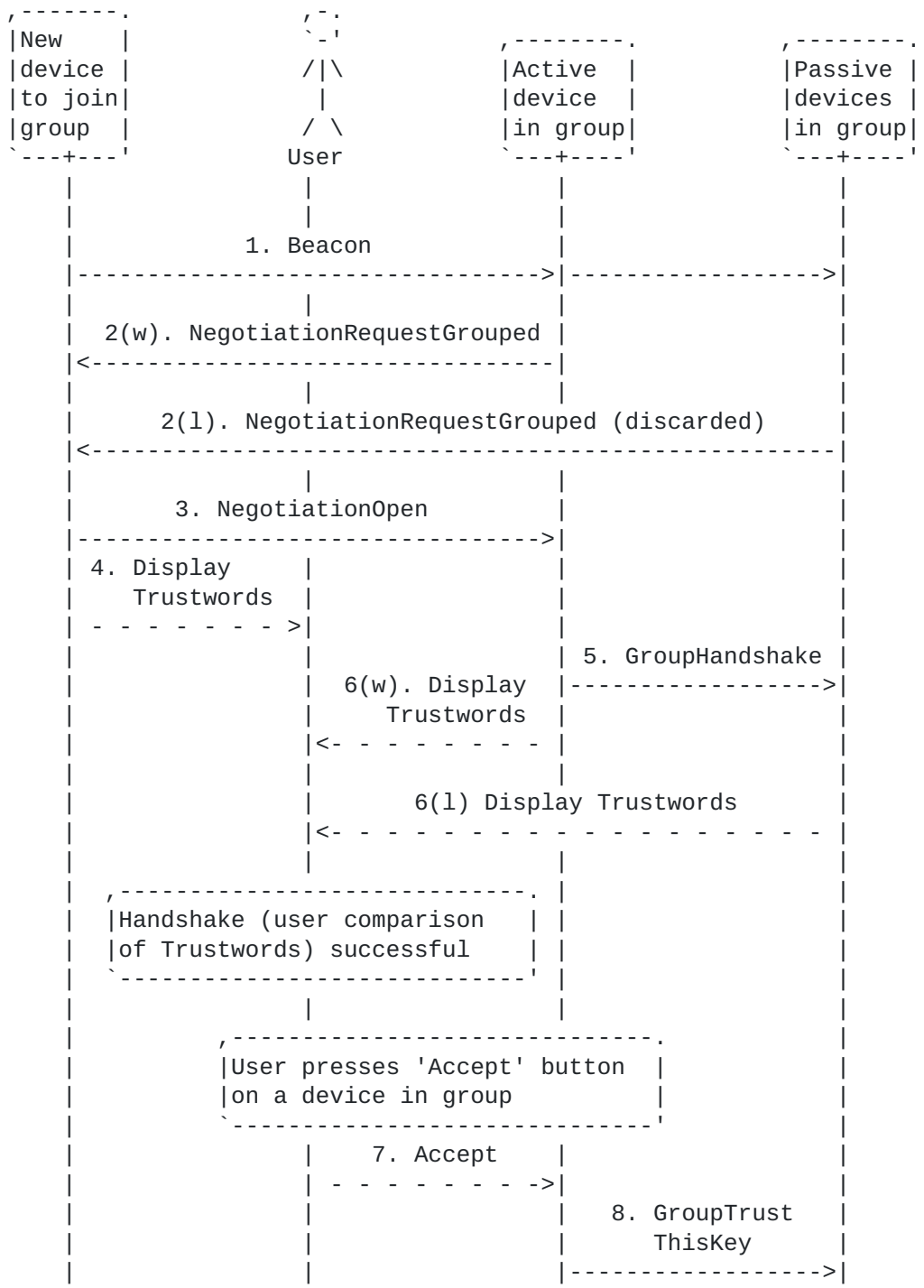
Note: The user may choose 'Cancel' on the 'Offerer' device, in which case the origin and/or destination of the messages change. However, the result will be exactly the same.

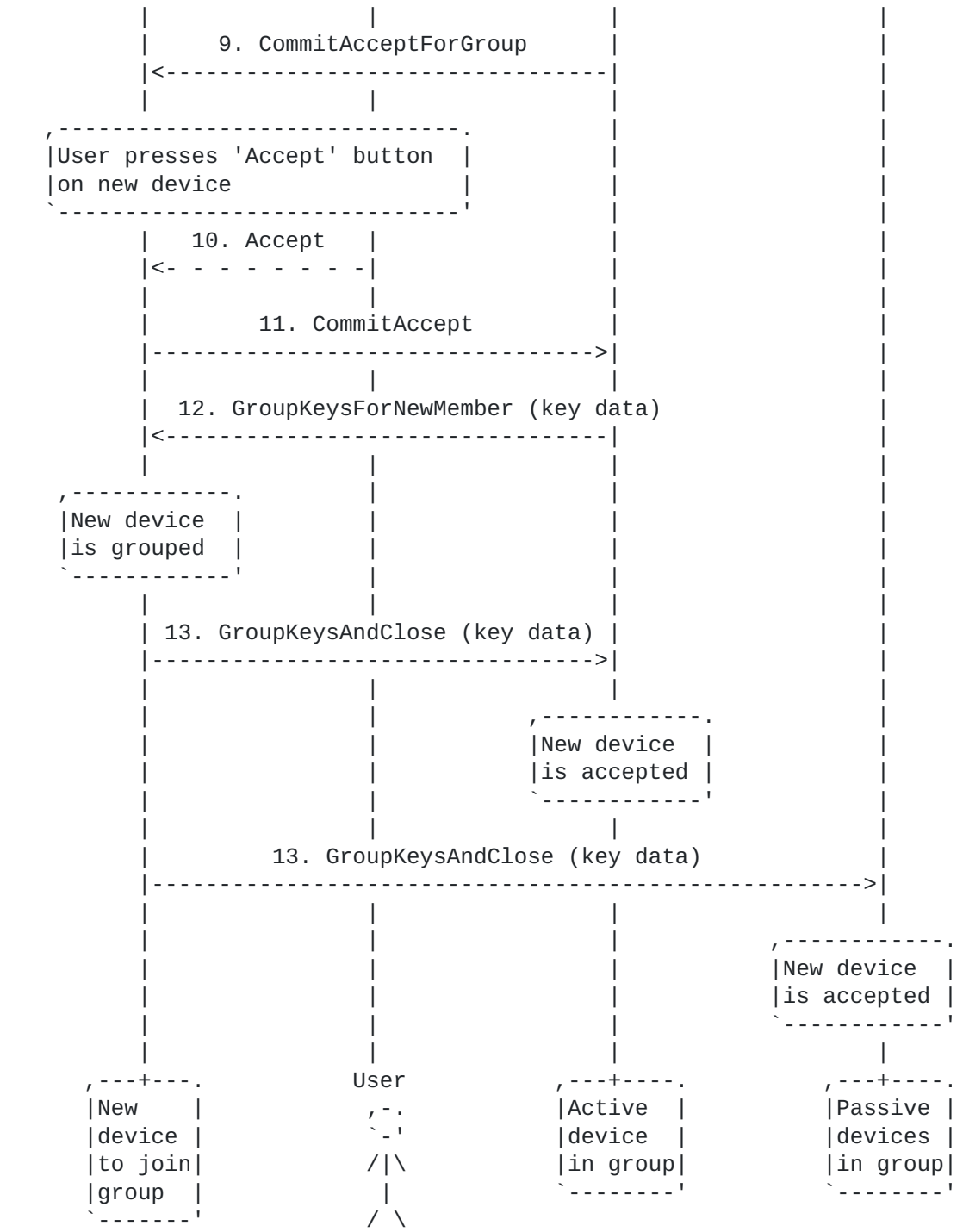
C7. Once Alice chooses the 'Cancel' option, the 'Requester' device sends a rollback message to the 'Offerer' device.

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




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)
New device is accepted
User
New device to join group
Active device in group
Passive devices in group

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.

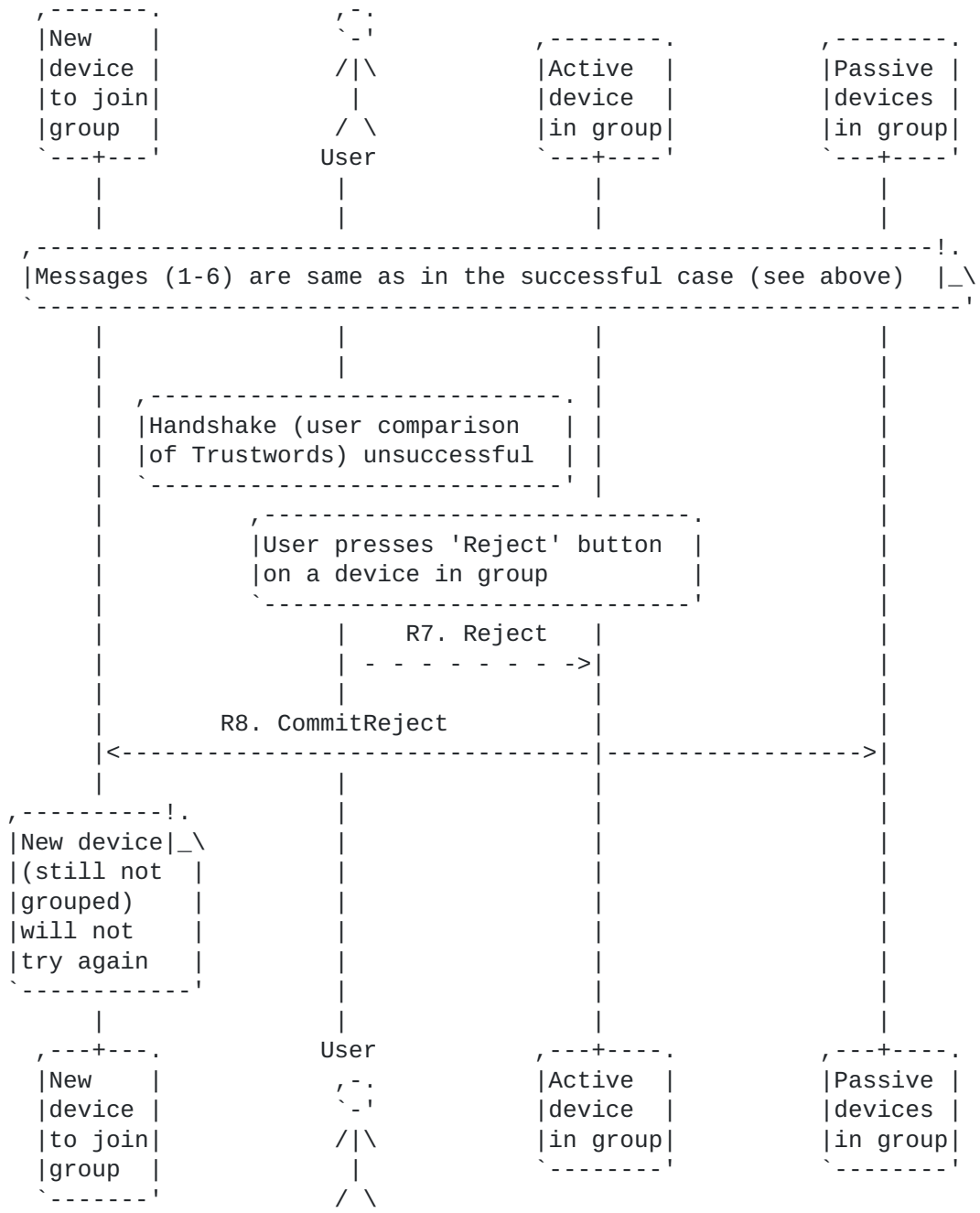
    Note: In the diagram, all messages marked "13. GroupKeysAndClose (key data)" are a single message, but drawn separately in order to convey that the message is sent to all devices in the Device Group.

    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).

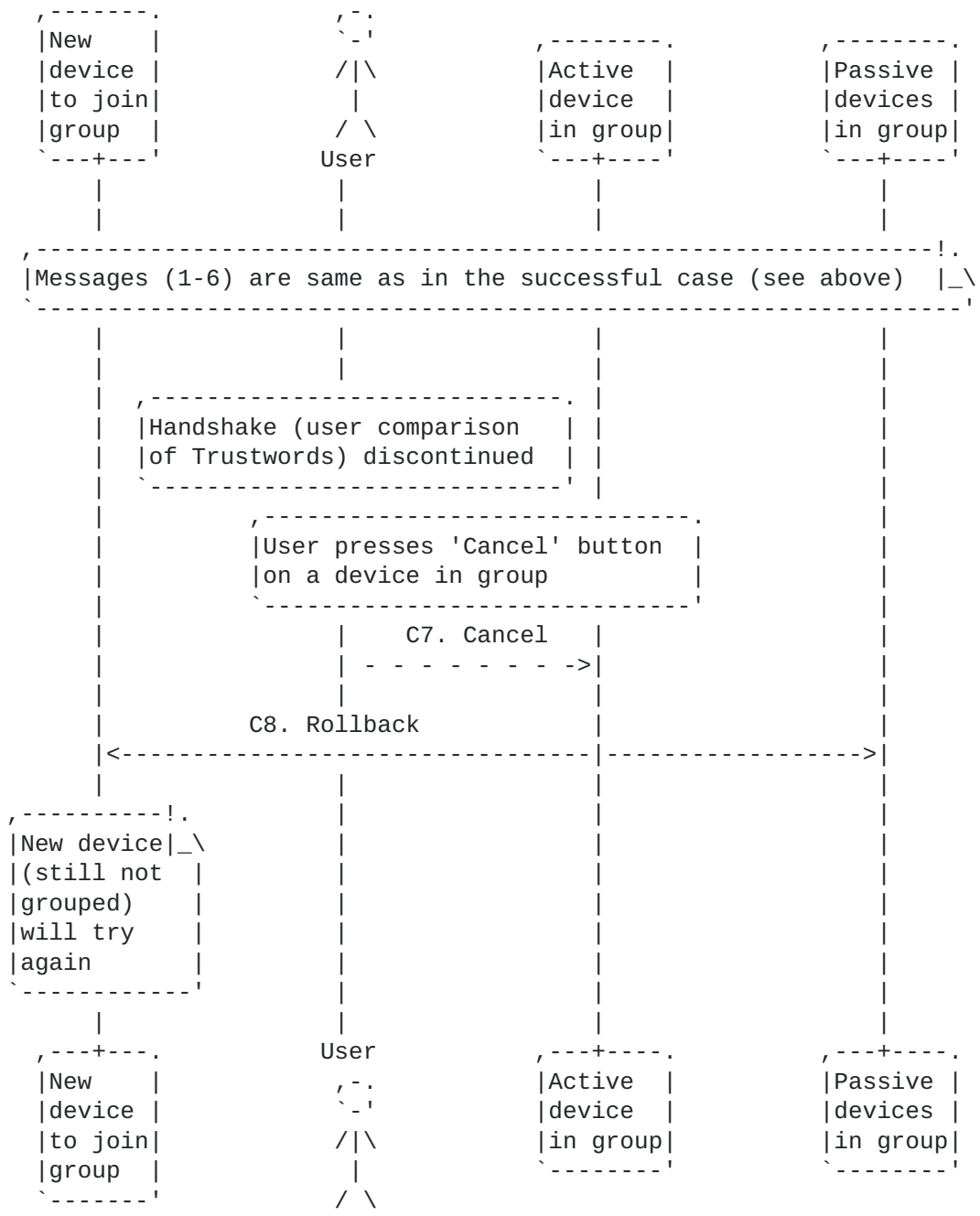
Note: Alice may choose 'Reject' on the new device, in which case the origin and/or destination of the messages change. However, the result will be exactly the same.

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, Hernani Marques, Itzel Vazquez Sandoval, 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.birk-pep] Birk, V., Marques, H., and B. Hoeneisen, "pretty Easy privacy (pEp): Privacy by Default", draft-birk-pep-05 (work in progress), November 2019.

[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>.

[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>.

7.2. Informative References

[I-D.birk-pep-trustwords] Hoeneisen, B. and H. Marques, "IANA Registration of Trustword Lists: Guide, Template and IANA Considerations", draft-birk-pep-trustwords-05 (work in progress), January 2020.

[I-D.marques-pep-handshake]
Marques, H. and B. Hoeneisen, "pretty Easy privacy (pEp): Contact and Channel Authentication through Handshake", draft-marques-pep-handshake-05 (work in progress), July 2020.

[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/>.

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 is based on a two-phase commit protocol (2PC) structure. This section describes the states, 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. Actions describe internal FSM functions, and fall into two general types. The first action type directs the state transitions within the FSM, and the second type drives UI functionality. Events are exchanged both between negotiation partners as well as the pEp engine 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/misc/doc/figures/sync/sync_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/misc/doc/figures/sync/
sync_fsm_simplified.svg

A.1.1. States

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. Otherwise, the FSM transitions to state Sole (cf. Appendix A.1.2.1).

A.1.1.2. Sole

This is the default FSM state for an ungrouped device.

On initialization, a Challenge TID is created and sent out inside of a Beacon message along with the device's current 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 conditional check on the current negotiation session to verify that the current session has not been disrupted or compromised. If this conditional 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 conditional check on the current negotiation session to verify that the current session has not been disrupted or compromised. If this conditional 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 conditional check on the current negotiation session to verify that the current session has not been disrupted or compromised. If this conditional 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 conditional check on the current negotiation session to verify that the current session has not been disrupted or compromised. If this conditional 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 conditional check on the current negotiation session to verify that the current session has not been disrupted or compromised. If this conditional 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 conditional check on the current negotiation session to verify that the current session has not been disrupted or compromised. If this conditional 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.

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

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 conditional check on the current negotiation session to verify that the current session has not been disrupted or compromised. If this conditional 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 conditional check on the current negotiation session to verify that the current session has not been disrupted or compromised. If this conditional 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 conditional check on the current negotiation session to verify that both the current session and negotiation partner have not been disrupted or compromised. If this conditional 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 user interface 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. When this response is received, the grouped devices FSM performs a sameNegotiation conditional check on the current negotiation session to verify that the current session has not been disrupted or compromised. If this conditional 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 conditional check 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 conditional check on the current negotiation session to verify that the current session has not been disrupted or compromised. If this conditional 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 an 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.

It is used to reset keys that have become invalid.

[[TODO: More detailed description]]

A.1.2. Conditions

All Conditions can either be true or false on successful execution, or, if the condition fails, the Finite State Machine is brought into an error state and reinitialized.

A.1.2.1. deviceGrouped

The 'deviceGrouped' conditional 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 to determine whether a device should transition to state Sole or state Grouped.

A.1.2.2. fromGroupMember

The 'fromGroupMember' conditional evaluates true if the incoming Sync Message is coming from a Device Group member.

A.1.2.3. keyElectionWon

The 'keyElectionWon' conditional evaluates true if our Own Keys are going to be used as Group Keys. False if the Own Keys of the partner will be the Group Keys. Calculated by comparing if the FPR of the Sender Key of the partner is greater than our Default Key for the Account, which is being used as Active Transport.

A.1.2.4. sameChallenge

The 'sameChallenge' conditional 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 sent by the originating Device itself.

A.1.2.5. sameNegotiation

The 'sameNegotiation' conditional is dependent upon the 'storeNegotiation' function, which stores the active negotiation session while the KeySync process is performed. This conditional evaluates true if the 'storeNegotiation' value of the incoming Sync Message is identical to that of the 'storeNegotiation' value that the Device is in.

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' conditional, the 'sameNegotiationAndPartner' conditional is dependent upon the 'storeNegotiation' function, which stores the active negotiation session while the KeySync process is performed. The 'sameNegotiation' conditional 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 conditional 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' conditional evaluates true if the Response of the incoming Sync Message is identical to the Response of the Device. In this case the Response is correctly echoed.

A.1.2.8. weAreOfferer

The 'weAreOfferer' conditional evaluates true if the Challenge of the incoming Sync Message is greater than the Challenge of the Device. Otherwise we're the Requester.

A.1.3. Actions

Actions are unconditionally executed. Any or all Actions may fail. In the event of failure, actions bring the Finite State Machine into an error state, and the Finite State Machine 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 Finite State Machine and disables KeySync functionality on the impacted device. It is most commonly called in 'Cancel' or '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' function results in the FSM transitioning to state End, which automatically disables the KeySync feature. 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 (Negotiatial Base).

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 is to clear Key and Identity of the partner and to calculate 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 process 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.

A.1.3.10. resetOwnKeysUngrouped

The 'resetOwnKeysUngrouped' action is to carry out a KeyReset on all Own Keys.

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 their 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 should use to display a pEp Handshake dialog. This dialog should indicate 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 should be 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 a 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 should use 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 The Sender FPR and partner's Identity are both stored for later comparison.

The storeNegotiation action saves the received non-own negotiation information, which is used e.g. by the sameNegotiation conditional 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 they are entering into a negotiation request with a Sole Device.

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. The trust also extends to the private key portion of the key pair at later stage in the KeySync process, so long as the user continues to choose '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 outside peers from the recipient 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 throughout the code. 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 (conditions, actions, messages, or transitions) contained within. Please refer to the desired State (Appendix B.1) for additional information on specific event handlers.

A.1.5.1. 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.

Example of an Init Event Handler:

```
on Init {
    if deviceGrouped {
       send SynchronizeGroupKeys;
       go Grouped;
    }
    go Sole;
}
```

A.1.5.2. 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 an 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.1.5.3. Signaled Events

Events, which don't share their name with a Message, are signaled from engine code.

Example of an 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.1.5.4. External Events

If Events are part of an API then their IDs must be well defined. Therefore, it is possible to define such IDs in the State Machine. External Event may be used to signal a User Interaction to the FSM.

Example:

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

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 Finite State Machine code in Appendix B.1.

The wire format of Sync Messages is defined in ASN.1 (cf. Appendix B.2), using PER.

Sync Messages are transported as Attachments to pEp Messages. Hence they're carried by the same Transports, which transmit pEp Messages. 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 Types

Each Sync Message has a name and an ID. There is 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

A.1.6.2. 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.1.6.3. 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> seconds. A Rate Limit of 0 means no Rate Limit checking.

Example:

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

A.1.6.4. Fields

A Sync Message can have Fields. There is two types of fields: automatically calculated fields, defined with the auto keyword, and 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 yet.

Example for a field coming from I/O Buffer

```
field TID challenge;
```

A.1.6.5. 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.1.6.6. 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.1.6.7. 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.1.6.8. Messages used in KeySync

For more information on the messages used in the KeySync Protocol, see (end of) Appendix B.1.

Appendix B. Code excerpts

B.1. Finite State Machine

Below is the full code 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;
                }
                go Sole;
            }
        }

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

            on KeyGen {
                send Beacon;
            }

            on CannotDecrypt {
                send Beacon;
            }

            on Beacon {
                if sameChallenge {
                }
                else {
                    if weAreOfferer {
                        do useOwnChallenge;
                        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 {
                    do storeNegotiation;
                    go HandshakingRequester;
                }
            }
        }

        // 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 {
                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;
                }
            }

            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 {
            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, 2017 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;

ISO639-1 ::= PrintableString(FROM ("a".."z")) (SIZE(2))
Hex ::= PrintableString(FROM ("A".."F") | FROM ("0".."9"))
Hash ::= Hex(SIZE(16..128)) -- 32bit Key ID to SHA512 in hex
PString ::= UTF8String  (SIZE(1..1024))
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
}

END
```

Appendix C. Document Changelog

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

- 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/

Kelly Bristol
pEp Foundation
Oberer Graben 4
CH-8400 Winterthur
Switzerland

Email: kelly.bristol@pep.foundation
URI: https://pep.foundation/