Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 8, 2020

B. Hoeneisen
Ucom.ch
H. Marques
pEp Foundation

July 07, 2019

pretty Easy privacy (pEp): Key Synchronization Protocol
draft-hoeneisen-pep-keysync-00

Abstract

Modern users of messaging systems usually have multiple devices, and often desire to send and receive encrypted messages on some or all of their devices. Using encryption on multiple devices often results in situations where messages cannot be decrypted on the device used to read the message due to a missing private key.

This document specifies a protocol for secure peer-to-peer synchronization of private keys across devices belonging to the same user (pEp Key Synchronization Protocol).

The pretty Easy privacy (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.

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 8, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 3
  1.1. Problem Statement 3
  1.2. Approach 4
  1.3. Main Challenge 4
  1.4. Requirements Language 4
  1.5. Terms 4
2. General Description 5
  2.1. Use Cases for pEp KeySync 6
    2.1.1. Form Device Group 6
    2.1.2. Add New Device to Existing Device Group 6
    2.1.3. Exchange Private Keys 6
    2.1.4. Leave Device Group 7
    2.1.5. Remove other Device from Device Group 7
  2.2. Interaction Diagrams 7
    2.2.1. Form Device Group 8
    2.2.2. Add New Device to Existing Device Group 16
    2.2.3. Exchange Private Keys 23
    2.2.4. Leave Device Group 23
    2.2.5. Remove other Device from Device Group 23
  2.3. Simplified Finite-State Machine 23
3. Reference Implementation 24
  3.1. Full Finite-State Machine 24
    3.1.1. States 24
    3.1.2. Actions 43
  3.2. Messages 69
    3.2.1. Format 69
    3.2.2. List of Messages Used in Finite-State Machine 69
    3.2.3. Example Messages 72
4. Security Considerations 72
5. Privacy Considerations 72
6. IANA Considerations 72

7. Acknowledgments 72
8. References 73
8.1. Normative References 73
8.2. Informative References 73
Appendix A. Document Changelog 74
Appendix B. Open Issues 74
Authors' Addresses 74

1. Introduction

This document specifies the pEp Key Synchronization (KeySync) Protocol, a means for secure peer-to-peer synchronization of private keys across devices belonging to the same user.

This part of the pretty Easy privacy (pEp) protocols [I-D.birk-pep] presents a way to synchronize private key material in a decentralized manner that is easy to implement. This network protocol is designed as a finite-state machine (FSM) along with the exchange of specific KeySync messages.

For pEp implementations, pEp KeySync for key synchronization 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 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. Problem Statement

Modern users of messaging systems usually have multiple devices, and often desire to send and receive encrypted messages on some or all of them.

Using encryption on multiple devices often results in situations where messages cannot be decrypted on the device used to read the message due to a missing private key. These messages were likely encrypted with a key that was generated on another device belonging to the same user. For example, the sender encrypts with a key that was generated on the home laptop of the recipient. The recipient then attempts to decrypt the message on their mobile phone, where the corresponding private key is not available. As a result, the recipient either must return to their laptop to decrypt the message, or attempt to copy the correct private key to their mobile device, which may expose the user's private key to potential leaks or theft.

1.2. 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 aims to do this by using Trustword confirmation (cf. [I-D.birk-pep-trustwords]) between any two devices at a time for pairing purposes. Simply put, a user needs to manually compare and confirm Trustwords before the automatic and security-sensitive transfer of private key information can occur.

1.3. 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 access to the device itself is beyond the scope of this document.

1.4. 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].

1.5. 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 by fallback: fingerprints). 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 ID plus a specific address (email, network ID, URI, etc.). A single user may have multiple identities.

- Device Group: All devices that are grouped to share data like cryptographic keys, trust information, calendar, configuration and other data. The data is synchronized through a common channel for a given identity, e.g., an identity can be an email address and the common channel a mailbox in email.

- Group Key: A key pair primarily used by a Device Group for a certain identity. A Device Group has a Group Key for each of a user's identities.

- 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.

2. General Description

This section describes an ideal-condition use case for pEp KeySync. It focuses on the main procedures and on the scenarios where everything works. Unexpected user behavior, error handling, race conditions and the like are mostly omitted from this section in order to facilitate a better understanding of the general concepts of pEp KeySync. Additional use cases will be discussed in further detail throughout Section 3.

2.1. Use Cases for pEp KeySync

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: a communication channel with an email address). Let us call these devices Alice_R and Alice_O. Each device already has its own key pair, which was automatically generated by the pEp protocol. Neither device knows anything about the other.

Alice wants full communication capability from both of her devices, but currently cannot do so, as the devices do not know about each other. Alice will use pEp Keysync to form a Device Group and add her devices to it. This allows for the exchange of private key data among its devices, allowing Alice to have full communication capability on both of her devices.

2.1.2. Add New Device to Existing Device Group

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

As before with devices Alice_R and Alice_O, Alice will use pEp Keysync to add device Alice_J to the existing Device Group, allowing all three devices to exchange private key information, and Alice to have 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, occasionally new key pairs are generated. For Alice to be able 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 may decide that one of her devices (e.g., her mobile phone) should no longer have access to all private keys of the Device Group. Alice can manually tell that device to leave the Device Group. The Device Group will ensure that further communication among the remaining Grouped Devices is private.

2.1.5. Remove other Device from Device Group

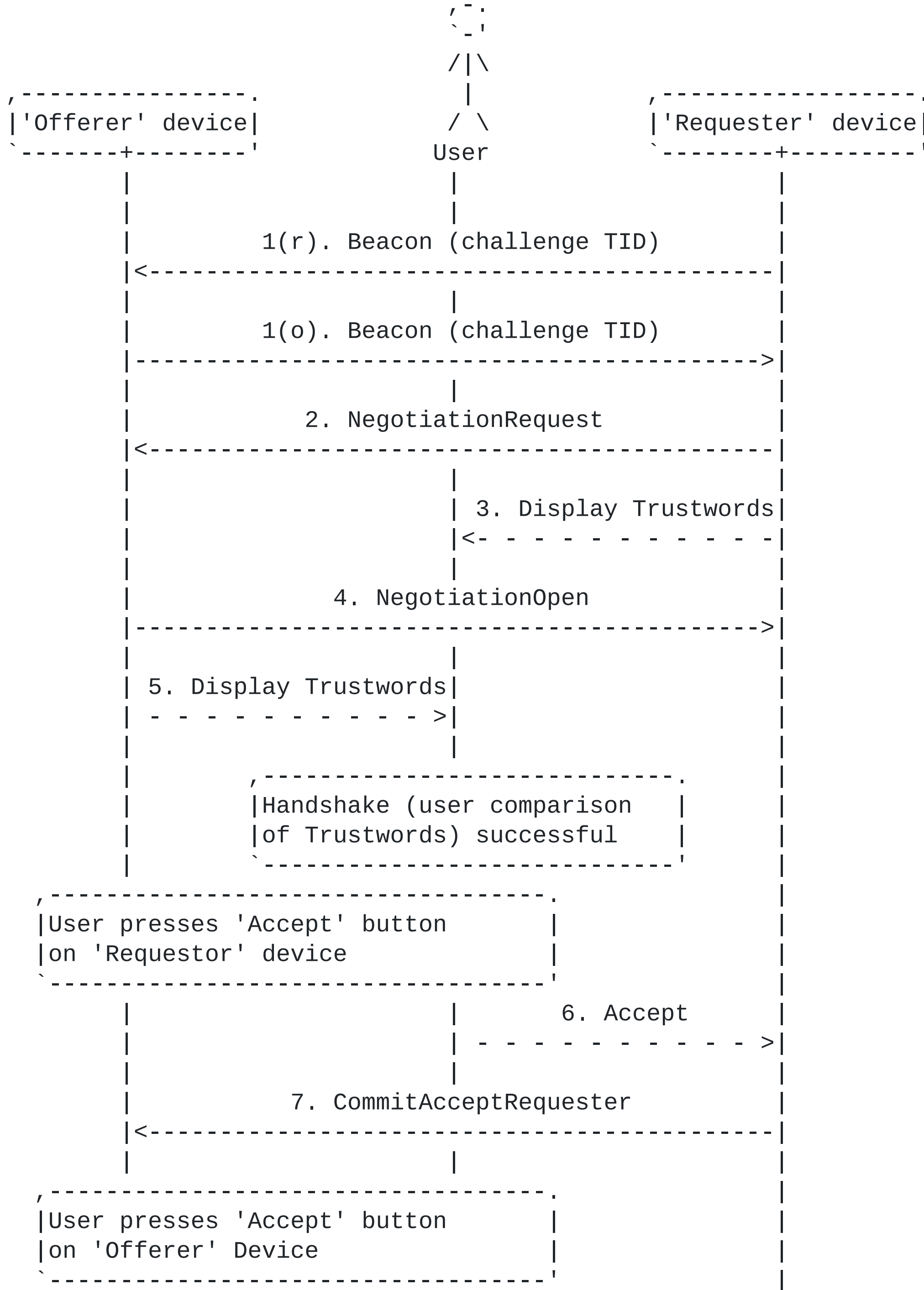
One of Alice's devices may be stolen or become otherwise compromised. She needs to ensure that the affected device no longer receives updates to private keys from the other devices in her Device Group. Alice can initiate actions to mitigate the damage, including the revocation of her private keys, as well as forcibly removing the compromised device from her Device Group.

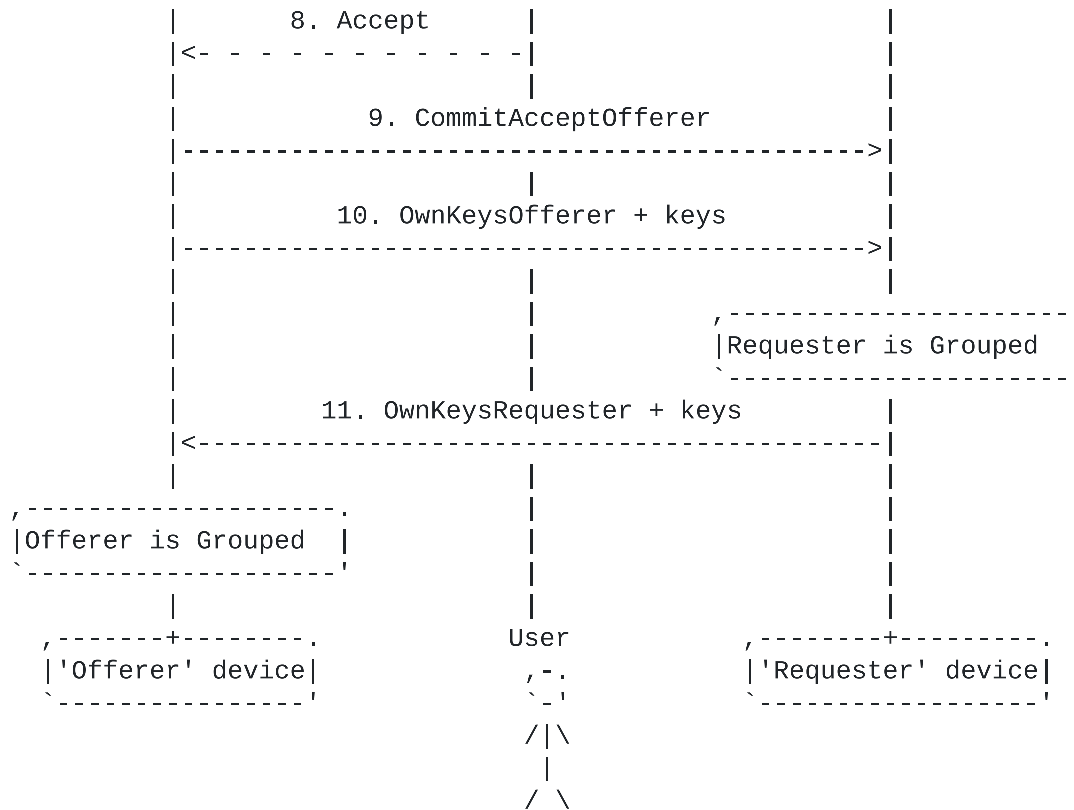
2.2. Interaction Diagrams

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.

2.2.1. Form Device Group

2.2.1.1. Successful Case




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

As depicted above, a user intends to form a Device Group in order to securely share key material among its members. The group is formed by an 'Offerer' device and a 'Requester' device. The names 'Offerer' and 'Requester' are derived from the FSM (cf. Section 2.3), 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 UUID version 4, variant 1 number, which is called 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.

Note: All messages are 'broadcast'. The TIDs added to each message allow the identification of received messages which pertain to the ongoing transaction and its sender.

1. Every device sends a Beacon message containing a challenge TID. Upon receipt of a Beacon message from another device, the 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. Thus, it re-sends its own Beacon (for robustness in case earlier Beacon message got lost) and waits. 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 the user.

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

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

6. The user compares the Trustwords of both devices. As the Trustwords are the same on both devices, the user presses the 'Accept' button on the 'Requester' device.

   Note 1: The user may also press the 'Accept' button on the 'Offerer' device first. The end result is not affected by which

'Accept' button is pressed first. However, the order of the messages slightly changes.

Note 2: The user may also choose to press the 'Cancel' button or the 'Reject' button (see below).

7. On receipt of the user's 'Accept', the 'Requester' device sends a CommitAcceptRequester message.

   The 'Offerer' device receives this message and waits for the user to press the local 'Accept' button.

8. The user compares the Trustwords of both devices and presses the 'Accept' button on the 'Offerer' device.

   Note: The user may also choose to press the 'Cancel' button or the 'Reject' button (see below).

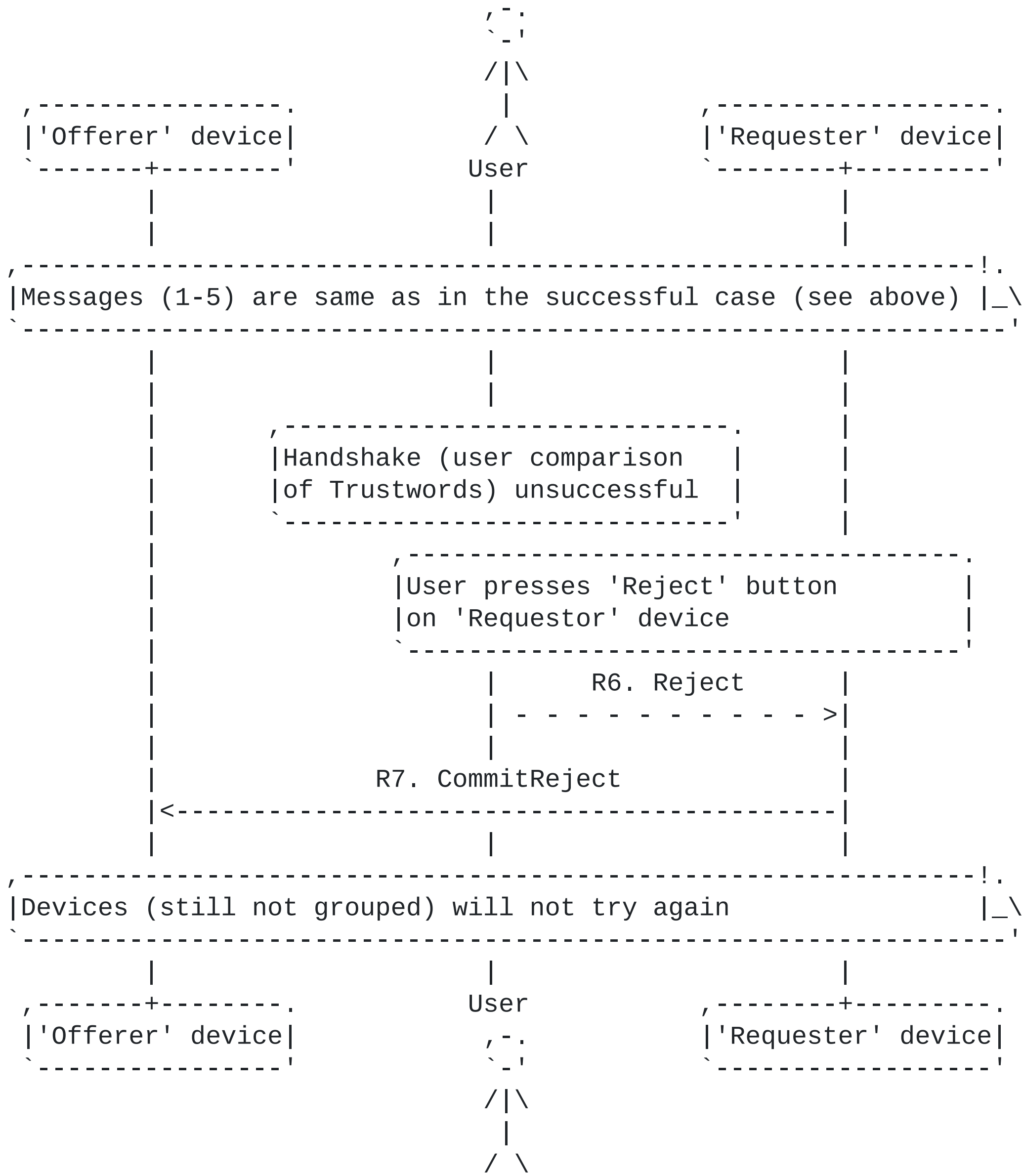
9. On receipt of the user's 'Accept', the 'Offerer' device sends a CommitAcceptRequester message.

10. The 'Offerer' device then sends an OwnKeysOfferer message along with the user's local key pairs (private and public keys) to to be synchronized.

11. Upon receipt of the OwnKeysOfferer message, the 'Requester' device is Grouped and sends an OwnKeysRequester message along with the user's local key pairs (private and public keys) to be synchronized.

    Upon receipt of the OwnKeysRequester message, also the 'Offerer' device is 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:

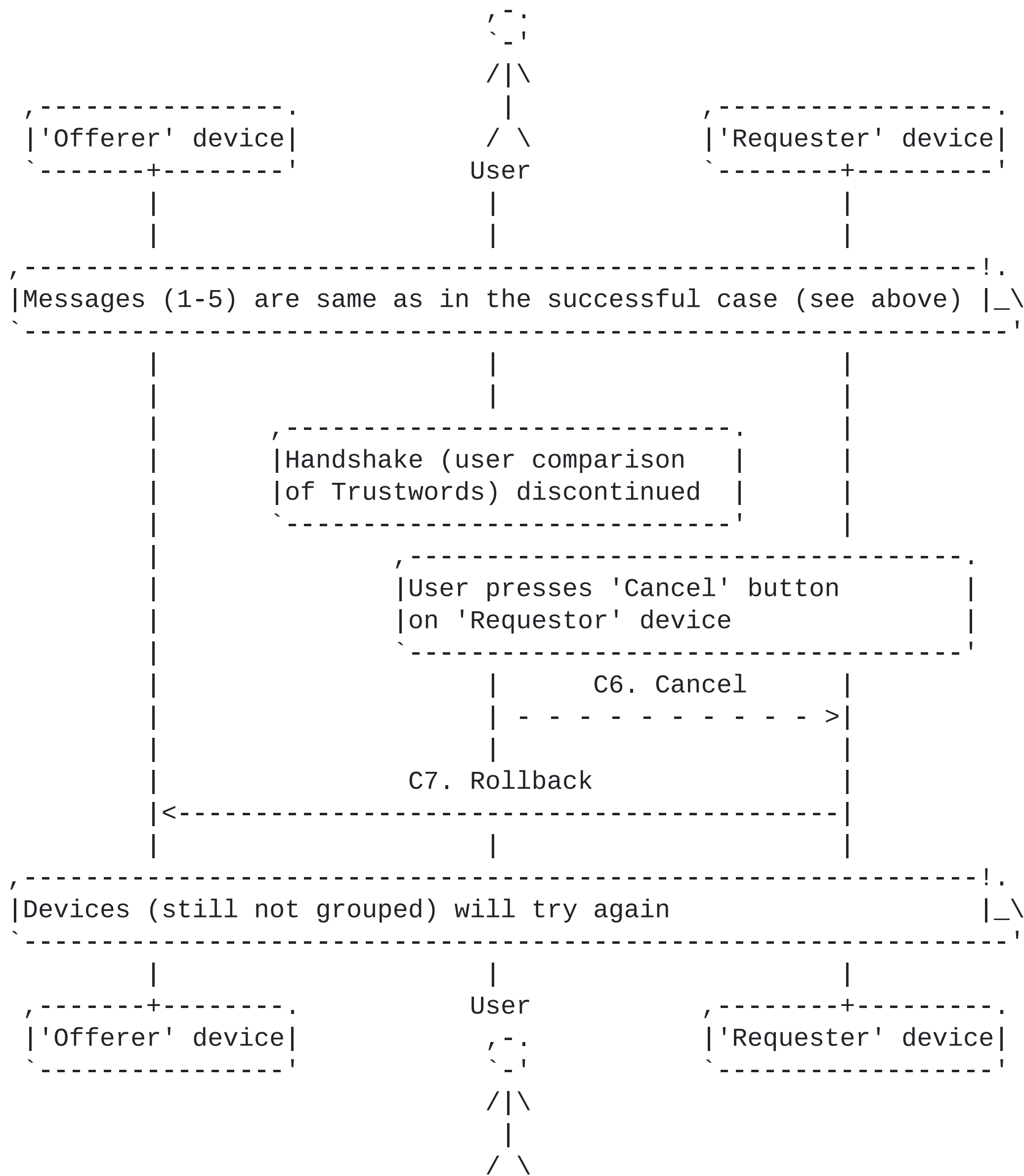
R6. The user compares the Trustwords of both devices. As the Trustwords do not match, the user presses the 'Reject' button on the 'Requester' device.

Note: The user may also press the 'Reject' button on the 'Offerer' device first. The end result is not affected by which 'Reject' button is pressed first. However, the order of the messages slightly changes.

R7. On receipt of the user's 'Reject', the 'Requester' device sends a CommitReject message.

Once the CommitReject message is sent or received, respectively, the devices 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.

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. The user decides to discontinue the process and presses the 'Cancel' button on the 'Requester' device.

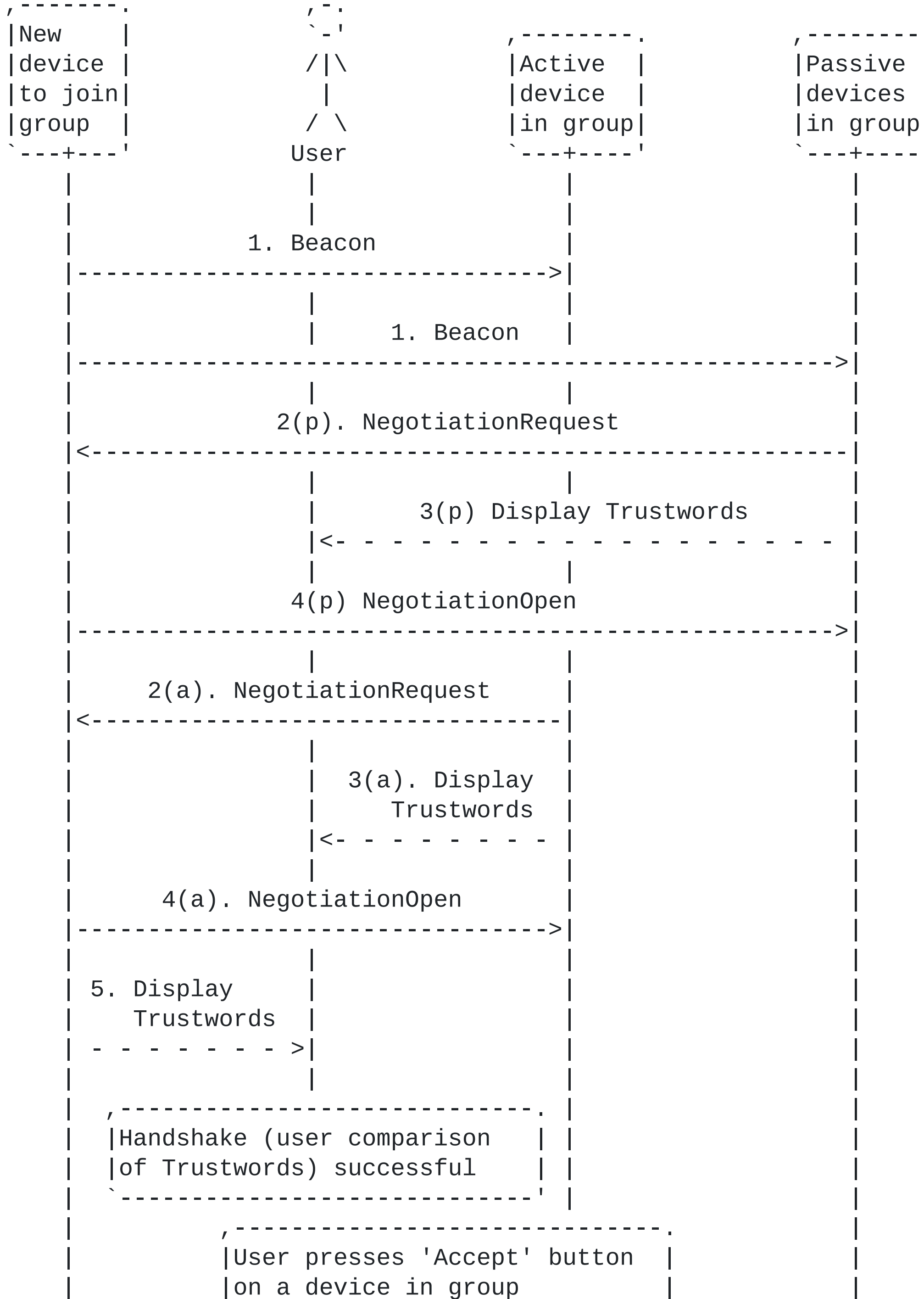
Note: The user may also press the 'Cancel' button on the 'Offerer' device first. The end result is not affected by which 'Cancel' button is pressed first. However, the order of the messages slightly changes.

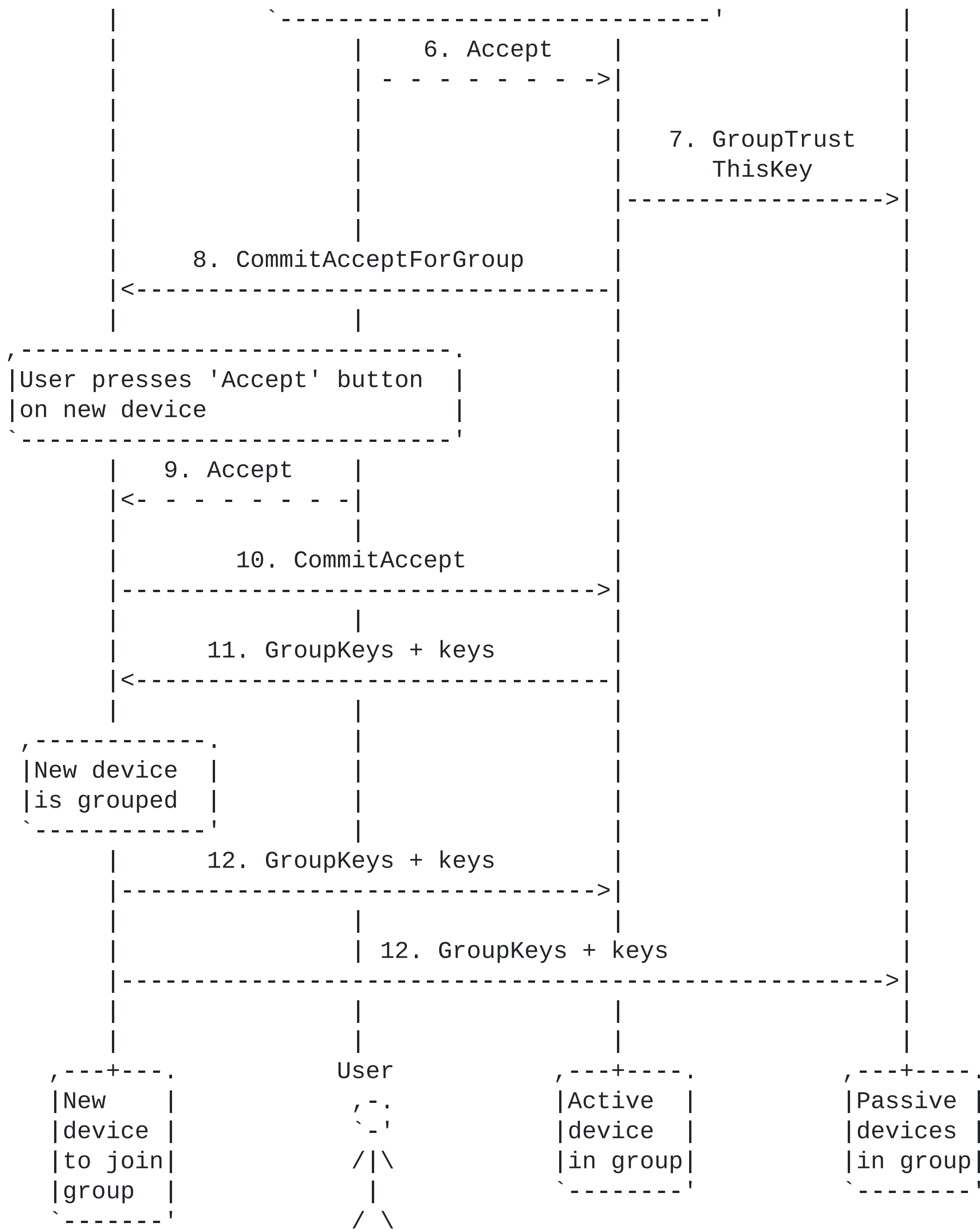
C7. On receipt of the user's 'Cancel', the 'Requester' device sends a rollback message.

The devices do not form a Device Group. KeySync remains enabled and forming a Device Group can start again.

2.2.2. Add New Device to Existing Device Group

2.2.2.1. Successful Case




6. Accept
7. GroupTrust
ThisKey
8. CommitAcceptForGroup
User presses 'Accept' button
on new device
9. Accept
10. CommitAccept
11. GroupKeys + keys
New device
is grouped
12. GroupKeys + keys
12. GroupKeys + keys
New
device
to join
group
User
Active
device
in group
Passive
devices
in group

As depicted above, a user intends to add a new device to an existing Device Group.

Note: All messages are 'broadcast'. The TIDs added to each message allow the identification of received messages which pertain to the ongoing transaction and its sender.

1. During initialization of pEp KeySync, the new device sends a Beacon message.

   Note: In the diagram, all messages marked "1. Beacon" are a single message, but drawn separately in order to convey that the message is sent to all devices in the Device Group.

2. Upon receipt of a Beacon message from a device not part of a Device Group, all Grouped Devices are send a NegotiationRequest message.

   Note: Messages 2(a) and 2(p) are different instances of the NegotiationRequest message type.

3. All Grouped Devices display the Trustwords to the user.

4. Upon receipt of every NegotiationRequest message, the New Device sends a NegotiationOpen message.

   Note: Messages 4(a) and 4(p) are different instances of the NegotiationOpen message type.

5. The new device displays the Trustwords to the user.

6. The user compares the Trustwords of both devices and presses the 'Accept' button on one of the Grouped Devices.

   Note 1: The Grouped Device that the user presses the 'Accept' button on now assumes the role of the active device in Group, while the other Grouped Devices get the role of passive devices in the group.

   Note 2: The user may also press the 'Accept' button on the new device first. The end result is not affected by which 'Accept' button is pressed first. However, the order and number of the messages change.

   Note 3: The user may also choose to press the 'Cancel' button or the 'Reject' button (see below).

7.   On receipt of the user's 'Accept', the active Grouped Device sends a TrustThisKey message, which is consumed by the passive Grouped Devices.

8.   Then, the Active Grouped Device also sends a CommitAcceptForGroup message.

     The new device receives this message and waits for the user to press the local 'Accept' button.

9.   The user compares the Trustwords of both devices and presses the 'Accept' button on the new device.

     Note: The user may also choose to press the 'Cancel' button or the 'Reject' button. However, these cases are explained later.

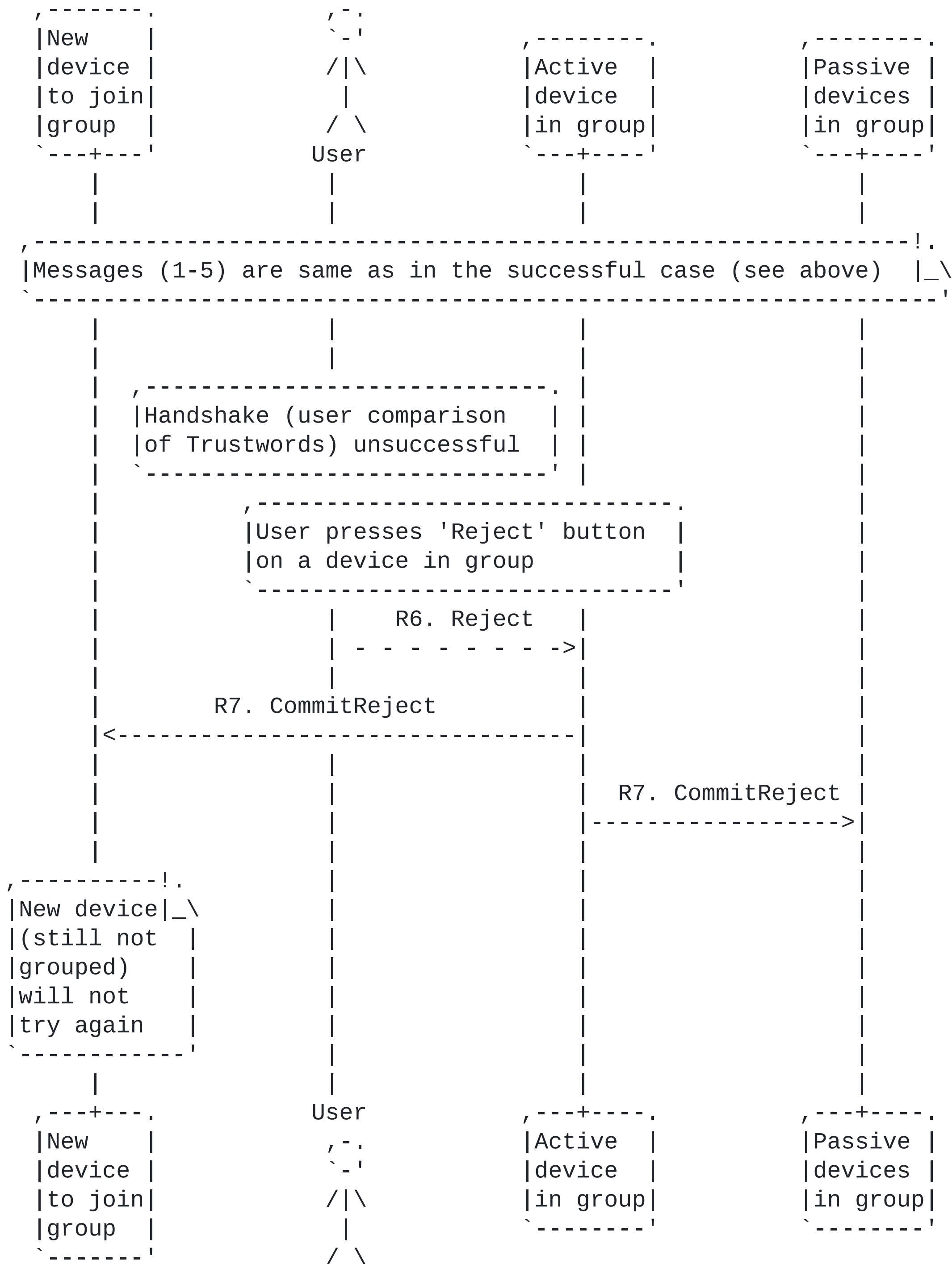
10. On receipt of the user's 'Accept', the new device sends a CommitAccept message.

11. The new device then sends a GroupKeys message along with its own private keys.

12. Upon receipt of the GroupKeys message, the Active Grouped Device is Grouped and sends a GroupKeys message along with its own private keys. The new device has successfully joined the Device Group.

Note: In the diagram, all messages marked "12. GroupKeys + keys" are a single message, but drawn separately in order to convey that the message is sent to all devices in the Device Group.

2.2.2.2. 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. The user compares the Trustwords of both devices. As the Trustwords do not match, the user presses the 'Reject' button on one of the Grouped Devices.

Note: The user may also press the 'Reject' button on the New Device first. The end result is not affected by which 'Reject'-Button is pressed first. However, the order and number of the messages slightly changes.

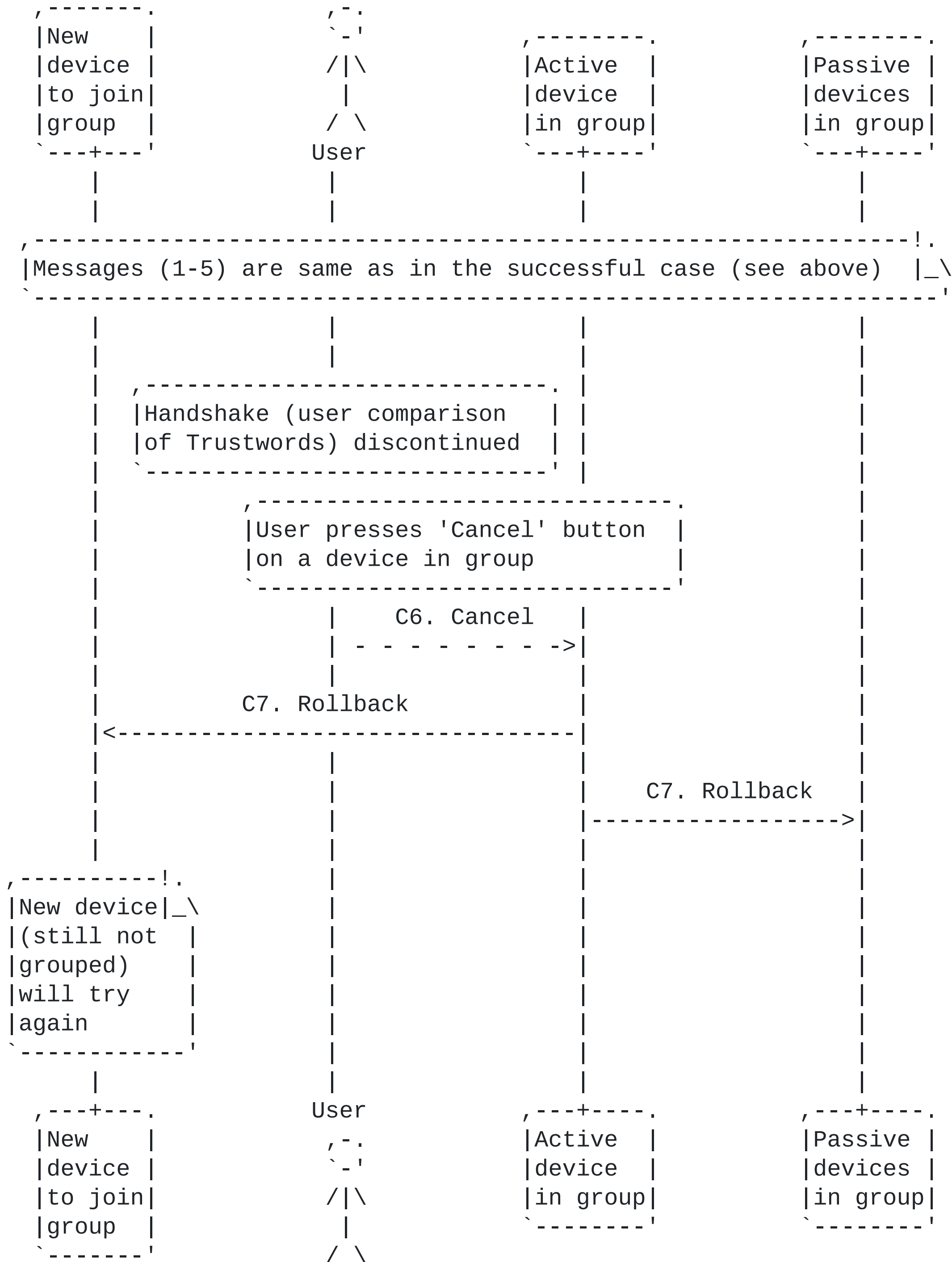
R7. On receipt of the user's 'Reject', the 'Requester' device sends a 'CommitReject' message.

Note: In the diagram, all messages marked "R7. CommitReject" are a single message, but drawn separately in order to convey that the message is sent to all devices in the Device Group.

The new device does not join the group and pEp KeySync is disabled on it. As a consequence, there are no further attempts to join the Device Group.

Once the CommitReject message is sent or received, respectively, the new device cannot join the Device Group, and pEp KeySync is disabled on the new device. As a result, there are no further attempts to join a Device Group by the new device.

2.2.2.3. 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. The user decides to discontinue the process and presses the 'Cancel' button on one of the Grouped Devices.

Note: The user may also press the 'Cancel' button on the New Device first. The end result is not affected by which 'Cancel'-Button is pressed first. However, the order and number of the messages slightly changes.

C7. On receipt of the user's 'Cancel', the 'Requester' device sends a rollback message.

Note: In the diagram, all messages marked "C7. Rollback" are a single message, but drawn separately in order to convey that the message is sent to all devices in the Device Group.

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

2.2.3. Exchange Private Keys

[[TODO]]

2.2.4. Leave Device Group

[[TODO]]

2.2.5. Remove other Device from Device Group

[[TODO]]

2.3. Simplified Finite-State Machine

A simplified diagram of the implemented pEp KeySync finite-state machine (FSM), which does not contain the transitions that occur when pressing the 'Cancel' or 'Reject' buttons, can be found at: https://pep.foundation/dev/repos/internet-drafts/raw-file/tip/misc/figures/sync/sync_fsm_simplified.svg

3. Reference Implementation

[[Note: This description of the FSM implementation is Work-In-Progress. For now it is supposed to be sufficient for the reader to understand the general functionality of the FSM, and thus lacking certain details. The authors intend to enhance and refine it in later revisions of this document.]]

3.1. Full Finite-State Machine

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

https://pep.foundation/dev/repos/internet-drafts/raw-file/tip/misc/figures/sync/sync_fsm_full.svg

3.1.1. States

3.1.1.1. 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, the FSM transitions to state Grouped. Otherwise, the FSM transitions to state Sole (cf. Section 3.1.2.1).

Please find more information in the following code excerpt:

```
state InitState {
    on Init {
        if deviceGrouped
            go Grouped;
        go Sole;
    }
}
```

3.1.1.2. Sole

This is the default state for an ungrouped device. On initialization, a challenge TID is created and sent out inside a Beacon message. It waits also 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 becomes 'Requester', the one with higher challenge TID becomes 'Offerer'.

If determined to be 'Requester', a NegotiationRequest message is sent.

The device determined as 'Offerer' receives the NegotiationRequest message, sends a NegotiationOpen message as response and the FSM transitions to state HandshakingOfferer.

On receipt of the NegotiationOpen message, the 'Requester' FSM proceeds in one of two ways. If the NegotiationOpen message comes from a Sole 'Offerer' Device, the FSM transitions to state HandshakingRequester. If the NegotiationOpen message comes from a Grouped Device, the FSM transitions to state HandshakingToJoin.

Please find more information in the following code excerpt:

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

            on KeyGen {
                send Beacon;
            }

            on CannotDecrypt { // cry baby
                send Beacon;
            }

            on Beacon {
                if sameChallenge {
                    // this is our own Beacon; ignore
                }
                else {
                    if weAreOfferer {
                        do useOwnChallenge;
                        send Beacon;
                    }
                    else /* we are requester */ {
                        do openNegotiation;
                        do tellWeAreNotGrouped;
                        // requester is sending NegotiationRequest
                        send NegotiationRequest;
                        do useOwnChallenge;
                    }
                }
            }
           on NegotiationRequest {
                if sameChallenge { // challenge accepted
                    if sameNegotiation {
```

```
                        // this is our own NegotiationRequest; ignore
                    }
                    else {
                        do storeNegotiation;
                        // offerer is accepting
                        // by confirming NegotiationOpen
                        send NegotiationOpen;
                        if partnerIsGrouped
                            go HandshakingToJoin;
                        else
                            go HandshakingOfferer;
                    }
                }
            }

            on NegotiationOpen if sameNegotiationAndPartner {
                // requester is receiving NegotiationOpen
                do storeNegotiation;
                go HandshakingRequester;
            }
        }
```

3.1.1.3. HandshakingOfferer

This state is entered by the 'Offerer' device only. The FSM waits for the user to compare the Trustwords and to press either of the following buttons (on the 'Offerer' device):

- Accept: A CommitAcceptOfferer message is sent and the FSM transitions to state HandshakingPhase1Offerer

- Reject: A CommitReject message is sent, pEp KeySync is disabled, and the FSM transitions to state End

- Cancel: A Rollback message is sent and the FSM transitions to state Sole

If the 'Accept' button was pressed on the 'Requester' device, a CommitAcceptRequester message is received and the FSM transitions to state HandshakingPhase2Offerer.

If the 'Reject' button was pressed on the 'Requester' device, a CommitReject message is received. pEp KeySync is disabled and the FSM transitions to state End.

If the 'Cancel' button was pressed on the 'Requester' device, a Rollback message is received and the FSM transitions to state Sole.

Please find more information in the following code excerpt:

```
        // 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 sameNegotiationAndPartner
                go Sole;

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

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

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

            // got a CommitAccept from requester
            on CommitAcceptRequester if sameNegotiationAndPartner
                go HandshakingPhase2Offerer;
        }
```

3.1.1.4. HandshakingRequester

This state is similar to the HandshakingOfferer (cf. Section 3.1.1.3), but with swapped roles.

This state is entered by the 'Requester' device only. The FSM waits for the user to compare the Trustwords and to press either of the following buttons (on the 'Requester' device):

- Accept: A CommitAcceptOfferer message is sent and the FSM transitions to state HandshakingPhase1Requester

- Reject: A CommitReject message is sent, pEp KeySync is disabled, and the FSM transitions to state End

- Cancel: A Rollback message is sent and the FSM transitions to state Sole

If the 'Accept' button was pressed on the 'Offerer' device, a CommitAcceptOfferer message is received and the FSM transitions to state HandshakingPhase2Requester.

If the 'Reject' button was pressed on the 'Offerer' device, a CommitReject message is received. pEp KeySync is disabled and the FSM transitions to state End.

If the 'Cancel' button was pressed on the 'Offerer' device, a Rollback message is received and the FSM transitions to state Sole.

Please find more information in the following code excerpt:

```
        state HandshakingRequester timeout=600 {
            on Init
                do showSoleHandshake;

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

            on Rollback if sameNegotiationAndPartner
                go Sole;

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

            on CommitReject if sameNegotiationAndPartner {
                do disable;
```

```
                    go End;
                }

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

                // got a CommitAccept from offerer
                on CommitAcceptOfferer if sameNegotiationAndPartner
                    go HandshakingPhase2Requester;
            }

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

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

                on CommitAcceptRequester if sameNegotiationAndPartner {
                    go FormingGroupOfferer;
                }
            }
```

3.1.1.5. HandshakingPhase1Offerer

This state is entered by the 'Offerer' device only. The FSM waits for the user to finish the handshake on the 'Requester' device (i.e. compare the Trustwords and press either of the buttons on the 'Requester' device):

If the 'Accept' button was pressed on the 'Requester' device, a CommitAcceptRequester message is received and the FSM transitions to state FormingGroupOfferer.

If the 'Reject' button was pressed on the 'Requester' device, a CommitReject message is received. pEp KeySync is disabled and the FSM transitions to state End.

If the 'Cancel' button was pressed on the 'Requester' device, a Rollback message is received and the FSM transitions to state Sole.

Please find more information in the following code excerpt:

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

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

            on CommitAcceptRequester if sameNegotiationAndPartner {
                go FormingGroupOfferer;
            }
        }
```

3.1.1.6. HandshakingPhase1Requester

This state is similar to the HandshakingPhase1Offerer (cf. Section 3.1.1.5), but with swapped roles.

This state is entered by the 'Requester' device only. The FSM waits for the user to finish the handshake on the 'Offerer' device (i.e. compare the Trustwords and press either of the buttons on the 'Offerer' device):

If the 'Accept' button was pressed on the 'Offerer' device, a CommitAcceptRequester message is received and the FSM transitions to state FormingGroupOfferer.

If the 'Reject' button was pressed on the 'Offerer' device, a CommitReject message is received. pEp KeySync is disabled and the FSM transitions to state End.

If the 'Cancel' button was pressed on the 'Offerer' device, a Rollback message is received and the FSM transitions to state Sole.

Please find more information in the following code excerpt:

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

               on CommitReject if sameNegotiationAndPartner {
                   do untrustThisKey;
                   do disable;
                   go End;
               }
               on CommitAcceptOfferer if sameNegotiationAndPartner {
                   go FormingGroupRequester;
               }
           }
```

3.1.1.7. HandshakingPhase2Offerer

This state is entered by the 'Offerer' device only. The FSM waits for the user to compare the Trustwords and to press either of the following buttons (on the 'Offerer' device):

- Accept: The key used in the handshake is marked 'trusted', a CommitAcceptOfferer message is sent and the FSM transitions to state FormingGroupOfferer

- Reject: A CommitReject message is sent, pEp KeySync is disabled, and the FSM transitions to state End

- Cancel: A Rollback message is sent and the FSM transitions to state Sole

Please find more information in the following code excerpt:

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

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

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

3.1.1.8. HandshakingPhase2Requester

This state is similar to the HandshakingPhase2Offerer (cf. Section 3.1.1.7), but with swapped roles.

This state is entered by the 'Requester' device only. The FSM waits for the user to compare the Trustwords and to press either of the following buttons (on the 'Requester' device):

- Accept: The key used in the handshake is marked 'trusted', a CommitAcceptOfferer message is sent and the FSM transitions to state FormingGroupRequester

- Reject: A CommitReject message is sent, pEp KeySync is disabled, and the FSM transitions to state End

- Cancel: A Rollback message is sent and the FSM transitions to state Sole

Please find more information in the following code excerpt:

```
state HandshakingPhase2Requester {
     on Cancel {
         send Rollback;
         go Sole;
     }

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

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

3.1.1.9. FormingGroupOfferer

This state is entered by the 'Offerer' device only. The FSM sends a OwnKeysOfferer along with its own keys, and waits to receive the keys from the 'Requester' device. Once received, the 'Requester' keys are saved and marked as Default Keys. The Device Group is created and the FSM transitions to state Grouped.

Please find more information in the following code excerpt:

```
state FormingGroupOfferer {
     on Init {
         do prepareOwnKeys;
         send OwnKeysOfferer; // we're not grouped yet,
                              // this is our own keys
     }

     on OwnKeysRequester {
         do saveGroupKeys;
         do receivedKeysAreDefaultKeys;
         do showGroupCreated;
         go Grouped;
     }
 }
```

3.1.1.10. FormingGroupRequester

This state is entered by the 'Requester' device only. The FSM sends a OwnKeysRequester along with its own keys, and waits to receive the keys from the 'Offerer' device. Once received, the 'Offerer' keys are saved. The own keys are marked as Default Keys. The Device Group is created and the FSM transitions to state Grouped.

Please find more information in the following code excerpt:

```
       state FormingGroupRequester {
            on Init {
                do prepareOwnKeys;
                send OwnKeysRequester; // we're not grouped yet,
                                       // this is our own keys
            }

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

3.1.1.11. Grouped

This is the default state for any grouped device. The device also waits 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 NegotiationRequest message and waits for the NegotiationOpen message.

On receipt of the NegotiationOpen message from the Sole Device the FSM of the 'Requester' transitions to state HandshakingGrouped.

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

Please find more information in the following code excerpt:

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

            on GroupKeys
                do saveGroupKeys;

            on KeyGen {
                do prepareOwnKeys;
                send GroupKeys;
            }

            on Beacon {
                do openNegotiation;
                do tellWeAreGrouped;
                send NegotiationRequest;
                do useOwnChallenge;
            }

            on NegotiationOpen if sameNegotiationAndPartner {
                do storeNegotiation;
                go HandshakingGrouped;
            }

            on GroupTrustThisKey
                do trustThisKey;
        }
```

3.1.1.12. HandshakingToJoin

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 to compare the Trustwords and to press either of the following buttons (on the new device):

- o Accept: A CommitAccept message is sent and the FSM transitions to state HandshakingToJoinPhase1

- o Reject: A CommitReject message is sent, pEp KeySync is disabled (on the new device), and the FSM transitions to state End

- o Cancel: A Rollback message is sent and the FSM transitions to state Sole

If the 'Accept' button was pressed on a grouped device, a CommitAcceptForGroup message is received and the FSM transitions to state HandshakingToJoinPhase2

If the 'Reject' button was pressed on a grouped device, a CommitReject message is received. pEp KeySync is disabled (on the new device) and the FSM transitions to state End.

If the 'Cancel' button was pressed on the 'Requester' device, a Rollback message is received and the FSM transitions to state Sole.

Please find more information in the following code excerpt:

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

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

               on Rollback if sameNegotiationAndPartner
                   go Sole;

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

               on CommitAcceptForGroup if sameNegotiationAndPartner
                   go HandshakingToJoinPhase2;

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

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

3.1.1.13. HandshakingToJoinPhase1

This state is entered by a new device only, i.e. a device that is not yet part of a Device Group. The FSM waits for the user to finish the handshake on a grouped device (i.e. compare the Trustwords and press either of the buttons on a grouped device):

If the 'Accept' button was pressed on a grouped device, a CommitAcceptForGroup message is received and the FSM transitions to state JoiningGroup.

If the 'Reject' button was pressed on a grouped device, a CommitReject message is received. pEp KeySync is disabled (on the new device) and the FSM transitions to state End.

If the 'Cancel' button was pressed on the 'Requester' device, a Rollback message is received and the FSM transitions to state Sole.

Please find more information in the following code excerpt:

```
state HandshakingToJoinPhase1 {
    on Rollback if sameNegotiationAndPartner
        go Sole;

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

    on CommitAcceptForGroup if sameNegotiationAndPartner
        go JoiningGroup;
}
```

3.1.1.14. 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 to compare the Trustwords and to press either of the following buttons (on the new device):

- Accept: A CommitAccept message is sent and the FSM transitions to state JoiningGroup

- Reject: A CommitReject message is sent, pEp KeySync is disabled (on the new device), and the FSM transitions to state End

- Cancel: A Rollback message is sent and the FSM transitions to state Sole

Please find more information in the following code excerpt:

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

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

               on Accept {
                   do trustThisKey;
                   go JoiningGroup;
               }
           }
```

3.1.1.15. JoiningGroup

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

The FSM waits to receive the keys from the active grouped device. Once received, these are saved and marked as default keys. Then it sends all keys to the grouped devices and the FSM transitions to state Grouped.

Please find more information in the following code excerpt:

```
          state JoiningGroup {
               on GroupKeys {
                   do saveGroupKeys;
                   do receivedKeysAreDefaultKeys;
                   do prepareOwnKeys;
                   send GroupKeys;
                   do showDeviceAdded;
                   go Grouped;
               }
           }
```

3.1.1.16. HandshakingGrouped

This state is entered by grouped devices only, i.e., devices that are part of a Device Group.

In this state the FSM waits for the user to compare the Trustwords and to press either of the following buttons (on any grouped device, which now becomes the 'Active' grouped device):

- Accept: A CommitAcceptForGroup message is sent and the FSM transitions to state HandshakingGroupedPhase1

- Reject: A CommitReject message is sent and the FSM transitions to state Grouped

- Cancel: A Rollback message is sent and the FSM transitions to state Grouped

If the 'Accept' button was pressed on the new device, a CommitAccept message is received and the FSM transitions to state HandshakingPhase2

If the 'Reject' button was pressed on the new device, a CommitReject message is received and the FSM transitions to state Grouped.

If the 'Cancel' button was pressed on the new device, a Rollback message is received and the FSM transitions to state Grouped.

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

Please find more information in the following code excerpt:

```
          state HandshakingGrouped {
               on Init
                   do showGroupedHandshake;

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

               on Rollback if sameNegotiationAndPartner
                   go Grouped;

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

               on CommitReject if sameNegotiationAndPartner
                   go Grouped;

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

               on CommitAccept if sameNegotiationAndPartner
                   go HandshakingGroupedPhase2;

               on GroupTrustThisKey {
                   do hideHandshakeDialog;
                   do trustThisKey;
               }

               on GroupKeys
                   do saveGroupKeys;
           }
```

3.1.1.17. HandshakingGroupedPhase1

This state is entered by grouped devices only, i.e., devices that are part of a Device Group. The FSM waits for the user to finish the

handshake on the new device (i.e., compare the Trustwords and press either of the buttons on the new device):

If the 'Accept' button was pressed on the new device, a CommitAccept message is received and the FSM transitions to state Grouped

If the 'Reject' button was pressed on the new device, a CommitReject message is received and the FSM transitions to state Grouped.

If the 'Cancel' button was pressed on the new device, a Rollback message is received and the FSM transitions to state Grouped.

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

Please find more information in the following code excerpt:

```
state HandshakingGroupedPhase1 {
    on Rollback if sameNegotiationAndPartner
        go Grouped;

    on CommitReject if sameNegotiationAndPartner
        go Grouped;

    on CommitAccept if sameNegotiationAndPartner {
        do prepareOwnKeys;
        send GroupKeys;
        go Grouped;
    }

    on GroupTrustThisKey {
        do trustThisKey;
    }

    on GroupKeys
        do saveGroupKeys;
}
```

3.1.1.18. HandshakingGroupedPhase2

This state is entered by grouped devices only, i.e. devices that are part of a Device Group.

In this state the FSM waits for the user to compare the Trustwords and to press either of the following buttons (on any grouped device, which now becomes the 'Active' grouped device):

- o Accept: A CommitAcceptForGroup message is sent and the FSM transitions to state HandshakingGroupedPhase1

- o Reject: A CommitReject message is sent and the FSM transitions to state Grouped

- o Cancel: A Rollback message is sent and 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, but in the execution of certain actions (e.g., saveGroupKeys).

Please find more information in the following code excerpt:

```
state HandshakingGroupedPhase2 {
    on Cancel {
        send Rollback;
        go Grouped;
    }

    on Reject {
        send CommitReject;
        go Grouped;
    }

    on Accept {
        do trustThisKey;
        send GroupTrustThisKey;
        do prepareOwnKeys;
        send GroupKeys;
        go Grouped;
    }

    on GroupTrustThisKey {
        do trustThisKey;
    }

    on GroupKeys
        do saveGroupKeys;
}
```

3.1.2. Actions

Actions describe internal FSM functions, and fall into two general types. The first action type is a conditional 'if' statement, which direct the transitional states within the FSM (cf. 'if' statements;

e.g., 'if deviceGrouped'). The second action type directs state changes which KeySync implementers can use to drive UI functionality, such as 'do' statements that trigger dialog box changes (cf 'do' statements; e.g., 'do hideHandshakeDialog').

3.1.2.1. deviceGrouped (conditional)

The 'deviceGrouped' conditional evaluates true if a device is already in a Device Group. 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.

The following code excerpt shows where this action is called:

```
state InitState {
    on Init {
        if deviceGrouped
            go Grouped;
        go Sole;
    }
}
```

3.1.2.2. disable

The 'disable' function may be called in an number of scenarios. For example, a user has rejected a pEp Handshake on either device involved in a pEp Handshake. At this time, in all cases, invoking the 'disable' function results in the FSM transitioning to state End, which disables the KeySync feature.

The following code excerpt shows where this action is called:

```
// 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 sameNegotiationAndPartner
                go Sole;

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

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

        [...]

        // 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 sameNegotiationAndPartner
                go Sole;

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

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

        [...]

        state HandshakingPhase1Offerer {
            on Rollback if sameNegotiationAndPartner {
                do untrustThisKey;
```

```
                go Sole;
            }

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

        [...]

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

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

        [...]

        state HandshakingToJoinPhase1 {
            on Rollback if sameNegotiationAndPartner
                go Sole;

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

            on CommitAcceptForGroup if sameNegotiationAndPartner
                go JoiningGroup;
        }

        [...]

        [[ TODO: More occurrences exist; perhaps it's better
                 to move to appendix? ]]
```

3.1.2.3. hideHandshakeDialog

The 'hideHandshakeDialog' function is invoked when a GroupTrustThisKey message is received by a device which is in the Grouped state and negotiating the addition of a new device, but another device within the Device Group has already confirmed the Trustwords dialog to accept the new device.

This action is intended to send an event to the 'Passive' grouped devices that forces the closure of any unnecessary dialog boxes.

The following code excerpt shows where this action is called:

```
          state HandshakingGrouped {
               on Init
                   do showGroupedHandshake;

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

               on Rollback if sameNegotiationAndPartner
                   go Grouped;

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

               on CommitReject if sameNegotiationAndPartner
                   go Grouped;

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

               on CommitAccept if sameNegotiationAndPartner
                   go HandshakingGroupedPhase2;

               on GroupTrustThisKey {
                   do hideHandshakeDialog;
                   do trustThisKey;
               }

               on GroupKeys
                   do saveGroupKeys;
           }
```

3.1.2.4. openNegotiation

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.

The following code excerpt shows where this action is called:

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

            on KeyGen {
                send Beacon;
            }

            on CannotDecrypt { // cry baby
                send Beacon;
            }

            on Beacon {
                if sameChallenge {
                    // this is our own Beacon; ignore
                }
                else {
                    if weAreOfferer {
                        do useOwnChallenge;
                        send Beacon;
                    }
                    else /* we are requester */ {
                        do openNegotiation;
                        do tellWeAreNotGrouped;
                        // requester is sending NegotiationRequest
                        send NegotiationRequest;
                        do useOwnChallenge;
                    }
                }

        [...]

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

```
               }

               on GroupKeys
                   do saveGroupKeys;

               on KeyGen {
                   do prepareOwnKeys;
                   send GroupKeys;
               }

               on Beacon {
                   do openNegotiation;
                   do tellWeAreGrouped;
                   send NegotiationRequest;
                   do useOwnChallenge;
               }
```

3.1.2.5. ownKeysAreDefaultKeys

[[TODO]]

The following code excerpt shows where this action is called:

```
        state FormingGroupOfferer {
            on Init {
                do prepareOwnKeys;
                send OwnKeysOfferer; // we're not grouped yet,
                                     // this is our own keys
            }

            on OwnKeysRequester {
                do saveGroupKeys;
                do receivedKeysAreDefaultKeys;
                do showGroupCreated;
                go Grouped;
            }
        }

        state FormingGroupRequester {
            on Init {
                do prepareOwnKeys;
                send OwnKeysRequester; // we're not grouped yet,
                                       // this is our own keys
            }

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

3.1.2.6. newChallengeAndNegotiationBase

[[TODO]]

The following code excerpt shows where this action is called:

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

        [...]

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

3.1.2.7. partnerIsGrouped (conditional)

[[TODO]]

The following code excerpt shows where this action is called:

```
        state Sole timeout=off {

            [...]

            on NegotiationRequest {
                if sameChallenge { // challenge accepted
                    if sameNegotiation {
                        // this is our own NegotiationRequest; ignore
                    }
                    else {
                        do storeNegotiation;
                        // offerer is accepting by confirming
                        // NegotiationOpen
                        send NegotiationOpen;
                        if partnerIsGrouped
                            go HandshakingToJoin;
                        else
                            go HandshakingOfferer;
                    }
                }
            }
```

3.1.2.8. prepareOwnKeys

[[TODO]]

The following code excerpt shows where this action is called:

```
state FormingGroupOfferer {
    on Init {
        do prepareOwnKeys;
        send OwnKeysOfferer; // we're not grouped yet,
                             // this is our own keys
    }

    [...]

[...]

state FormingGroupRequester {
    on Init {
        do prepareOwnKeys;
        send OwnKeysRequester; // we're not grouped yet,
                               // this is our own keys
    }

    [...]

[...]

state Grouped timeout=off {

    [...]

    on KeyGen {
        do prepareOwnKeys;
        send GroupKeys;
    }

    [...]

[...]
```

3.1.2.9. receivedKeysAreDefaultKeys

The 'receivedKeysAreDefaultKeys' action tells the pEp implementer to set the keys just received as the default for outgoing communications.

[[TODO: Clear out, in which cases which keys are used as default keys.]]

The following code excerpt shows where this action is called:

```
        state FormingGroupOfferer {
            on Init {
                do prepareOwnKeys;
                send OwnKeysOfferer; // we're not grouped yet,
                                     // this is our own keys
            }

            on OwnKeysRequester {
                do saveGroupKeys;
                do receivedKeysAreDefaultKeys;
                do showGroupCreated;
                go Grouped;
            }
        }

        [...]

        state JoiningGroup {
            on GroupKeys {
                do saveGroupKeys;
                do receivedKeysAreDefaultKeys;
                do prepareOwnKeys;
                send GroupKeys;
                do showDeviceAdded;
                go Grouped;
            }
        }

        [...]
```

3.1.2.10. sameChallenge (conditional)

[[TODO]]

The following code excerpt shows where this action is called:

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

```
            on KeyGen {
                send Beacon;
            }

            on CannotDecrypt { // cry baby
                send Beacon;
            }

            on Beacon {
                if sameChallenge {
                    // this is our own Beacon; ignore
                }
                else {
                    if weAreOfferer {
                        do useOwnChallenge;
                        send Beacon;
                    }
                    else /* we are requester */ {
                        do openNegotiation;
                        do tellWeAreNotGrouped;
                        // requester is sending NegotiationRequest
                        send NegotiationRequest;
                        do useOwnChallenge;
                    }
                }

            on NegotiationRequest {
                if sameChallenge { // challenge accepted
                    if sameNegotiation {
                        // this is our own NegotiationRequest; ignore
                    }
                    else {
                        do storeNegotiation;
                        // offerer is accepting by confirming
                        // NegotiationOpen
                        send NegotiationOpen;
                        if partnerIsGrouped
                            go HandshakingToJoin;
                        else
                            go HandshakingOfferer;
                    }
                }
            }

        [...]
```

3.1.2.11. sameNegotiation (conditional)

The 'sameNegotiation' action evaluates true if the FSM finds a NegotiationRequest message that a Sole Device sent out is determined to be self-originating. The Transaction ID (TID) will be an exact match upon comparison, and the NegotiationRequest will be ignored as a result.

The following code excerpt shows where this action is called:

```
        state Sole timeout=off {

            [...]

            on NegotiationRequest {
                if sameChallenge { // challenge accepted
                    if sameNegotiation {
                        // this is our own NegotiationRequest; ignore
                    }
                    else {
                        do storeNegotiation;
                        // offerer is accepting by confirming
                        // NegotiationOpen
                        send NegotiationOpen;
                        if partnerIsGrouped
                            go HandshakingToJoin;
                        else
                            go HandshakingOfferer;
                    }
                }
            }
```

3.1.2.12. sameNegotiationAndPartner (conditional)

[[TODO]]

The following code excerpt shows where this action is called:

3.1.2.13. saveGroupKeys

[[TODO]]

The following code excerpt shows where this action is called:

3.1.2.14. showBeingInGroup

The 'showBeingInGroup' action (in state Grouped) is used to notify a pEp implementer that a device is Grouped and to prepare the necessary structures to potentially carry out KeySync (in case any other device exists and a Beacon from a Sole Device appears at some point).

The following code excerpt shows where this action is called:

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

3.1.2.15. showBeingSole

The 'showBeingSole' action (in state Sole) is used to notify a pEp implementer that a device is ungrouped (or: sole) and to prepare the necessary structures to potentially carry out KeySync (in case any other device exists and a negotiation starts).

The following code excerpt shows where this action is called:

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

3.1.2.16. showDeviceAdded

The 'showDeviceAdded' action is used to notify a pEp implementer that a Sole Device was added to an already existing Device Group.

The following code excerpt shows where this action is called:

```
state JoiningGroup {
    on GroupKeys {
        do saveGroupKeys;
        do receivedKeysAreDefaultKeys;
        do prepareOwnKeys;
        send GroupKeys;
        do showDeviceAdded;
        go Grouped;
    }
}
```

3.1.2.17. showGroupCreated

In both roles a Sole Device can assume (either as Requester or Offerer), this action 'showGroupCreated' ensures that the pEp implementer gets a notification that a new Device Group was formed (both devices coming from Sole state in their respective FSMs).

The following code excerpt shows where this action is called:

```
state FormingGroupOfferer {
    on Init {
        do prepareOwnKeys;
        send OwnKeysOfferer; // we're not grouped yet,
                             // this is our own keys
    }

    on OwnKeysRequester {
        do saveGroupKeys;
        do receivedKeysAreDefaultKeys;
        do showGroupCreated;
        go Grouped;
    }
}

       state FormingGroupRequester {
    on Init {
        do prepareOwnKeys;
        send OwnKeysRequester; // we're not grouped yet,
                               // this is our own keys
    }

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

3.1.2.18. showGroupedHandshake

The 'showGroupedHandshake' action notifies a pEp implementer to show a pEp Handshake on the own, already Grouped Device (which is thus in state Grouped). That means, the Handshake should be displayed in a way indicating there is another (yet) Sole Device willing to join the Device Group one is already a member of.

The following code excerpt shows where this action is called:

```
state HandshakingGrouped {
    on Init
        do showGroupedHandshake;
```

3.1.2.19. showJoinGroupHandshake

The 'showJoinGroupHandshake' action is used to notify pEp implementers to show a Handshake dialog when a Sole Device is to be grouped with an already existing Device Group, such that a user can 'Accept', 'Cancel' or 'Reject' a group joining process.

The following code excerpt shows where this action is called:

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

3.1.2.20. showSoleHandshake

For either the role of the Requester or the Offerer, in case of two Sole Devices, 'showSoleHandshake' notifies a pEp implementer that a pEp Handshake dialog between the two devices in negotiation has to be shown (depending on the role this action is called from two different states; see code below), such that the user (on both Sole Devices in user-defined sequence) can either press 'Accept', 'Cancel' or 'Reject' to decide on the group formation process.

The following code excerpt shows where this action is called:

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

        [...]

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

3.1.2.21. storeNegotiation

[[TODO]]

The following code excerpt shows where this action is called:

```
state Sole timeout=off {

    [...]

    on NegotiationRequest {
        if sameChallenge { // challenge accepted
            if sameNegotiation {
                // this is our own NegotiationRequest; ignore
            }
            else {
                do storeNegotiation;
                // offerer is accepting by confirming
                // NegotiationOpen
                send NegotiationOpen;
                if partnerIsGrouped
                    go HandshakingToJoin;
                else
                    go HandshakingOfferer;
            }
        }
    }

    on NegotiationOpen if sameNegotiationAndPartner {
        // requester is receiving NegotiationOpen
        do storeNegotiation;
        go HandshakingRequester;
    }

[...]

state Grouped timeout=off {

    [...]

    on NegotiationOpen if sameNegotiationAndPartner {
        do storeNegotiation;
        go HandshakingGrouped;
    }

    [...]

}
```

3.1.2.22. tellWeAreGrouped

The 'tellWeAreGrouped' action is used in case a device is already is in state Grouped; it is used to modify KeySync state marking one is grouped. This plays a role in negotiation with another device being Sole, such it knows it is about to join an already existing Device Group.

The following code excerpt shows where this action is called:

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

            on GroupKeys
                do saveGroupKeys;

            on KeyGen {
                do prepareOwnKeys;
                send GroupKeys;
            }

            on Beacon {
                do openNegotiation;
                do tellWeAreGrouped;
                send NegotiationRequest;
                do useOwnChallenge;
            }
```

3.1.2.23. tellWeAreNotGrouped

The 'tellWeAreNotGroupd' action is used by Sole Devices (thus in state Sole) which get into the role of being a Requester. This action modifies KeySync state such as to mark one is a Sole Device (and not already in a Device Group). That is, the other Sole Device (in Offerer role) will learn the negotiation is about to form a Device Group.

The following code excerpt shows where this action is called:

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

            on KeyGen {
                send Beacon;
            }

            on CannotDecrypt { // cry baby
                send Beacon;
            }

            on Beacon {
                if sameChallenge {
                    // this is our own Beacon; ignore
                }
                else {
                    if weAreOfferer {
                        do useOwnChallenge;
                        send Beacon;
                    }
                    else /* we are requester */ {
                        do openNegotiation;
                        do tellWeAreNotGrouped;
                        // requester is sending NegotiationRequest
                        send NegotiationRequest;
                        do useOwnChallenge;
                    }
                }
            }
```

3.1.2.24. trustThisKey

The 'trustThisKey' action is executed in all states a user can do 'Accept' of the Handshake dialog. The pEp implementer is told to put trust on the public key received by the other device, which wants to get paired. This means, depending on the role or grouping status the own device is in, this key is candidate to be the new default key. (Note: The trust also extends to the private key part of the public key received at a later stage of the FSM - given the user does effectively 'Accept' on both devices; this trust can also be removed if a 'Reject' on the other device occurs, cf. Section 3.1.2.25.)

The following code excerpt shows where this action is called:

```
        // handshaking without existing Device group
        state HandshakingOfferer timeout=600 {

            [...]

            // Accept means init Phase1Commit
            on Accept {
                do trustThisKey;
                send CommitAcceptOfferer;

        [...]

        // handshaking without existing Device group
        state HandshakingRequester timeout=600 {

            [...]

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

        state HandshakingPhase2Offerer {

            [...]

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

        [...]

        state HandshakingPhase2Requester {

            [...]

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

```
           [...]

           state Grouped timeout=off {

               [...]

               on GroupTrustThisKey
                   do trustThisKey;
           }

          [...]

          state HandshakingToJoin {

               [...]

               // Accept is Phase1Commit
               on Accept {
                   do trustThisKey;
                   send CommitAccept;

           [...]

           state HandshakingToJoinPhase2 {

               [...]

               on Accept {
                   do trustThisKey;
                   go JoiningGroup;
               }
           }

           [...]

           state HandshakingGrouped {

               [...]

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

              [...]
```

```
              on GroupTrustThisKey {
                   do hideHandshakeDialog;
                   do trustThisKey;
               }

           [...]

           state HandshakingGroupedPhase1 {

               [...]

               on GroupTrustThisKey {
                   do trustThisKey;
               }

           [...]

           state HandshakingGroupedPhase2 {

               [...]

               on Accept {
                   do trustThisKey;
                   send GroupTrustThisKey;
                   do prepareOwnKeys;
                   send GroupKeys;
                   go Grouped;
               }
```

3.1.2.25. untrustThisKey

If, after an own 'Accept', the other device returns a 'Reject' action, trust on the other device's public key is removed. The action 'untrustThisKey' ensures that the public key that has already been imported is untrusted (cf. Section 3.1.2.24). As a result, the public key, despite being imported, is never attached to messages sent to outside peers.

The following code excerpt shows where this action is called:

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

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

           [...]

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

               on CommitReject if sameNegotiationAndPartner {
                   do untrustThisKey;
                   do disable;
                   go End;
               }
```

3.1.2.26. useOwnChallenge

[[TODO]]

The following code excerpt shows where this action is called:

```
           state Sole timeout=off {

               [...]

               on Beacon {
                   if sameChallenge {
                       // this is our own Beacon; ignore
                   }
                   else {
                       if weAreOfferer {
                           do useOwnChallenge;
                           send Beacon;
                       }
                       else /* we are requester */ {
                           do openNegotiation;
                           do tellWeAreNotGrouped;
                           // requester is sending NegotiationRequest
                           send NegotiationRequest;
                           do useOwnChallenge;
                       }
                   }

           [...]

           state Sole timeout=off {

              [...]

               on Beacon {
                   if sameChallenge {
                       // this is our own Beacon; ignore
                   }
                   else {
                       if weAreOfferer {
                           do useOwnChallenge;
                           send Beacon;
                       }
                       else /* we are requester */ {
                           do openNegotiation;
                           do tellWeAreNotGrouped;
                           // requester is sending NegotiationRequest
                           send NegotiationRequest;
                           do useOwnChallenge;
                       }
                   }
```

3.2. Messages

[[TODO]]

3.2.1. Format

[[TODO]]

3.2.2. List of Messages Used in Finite-State Machine

3.2.2.1. Beacon

[[TODO]]

Please find more information in the following code excerpt:

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

3.2.2.2. NegotiationRequest

[[TODO]]

Please find more information in the following code excerpt:

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

3.2.2.3. NegotiationOpen

[[TODO]]

Please find more information in the following code excerpt:

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

3.2.2.4. Rollback

[[TODO]]

Please find more information in the following code excerpt:

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

3.2.2.5. CommitReject

[[TODO]]

Please find more information in the following code excerpt:

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

3.2.2.6. CommitAcceptOfferer

[[TODO]]

Please find more information in the following code excerpt:

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

3.2.2.7. CommitAcceptRequester

[[TODO]]

Please find more information in the following code excerpt:

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

3.2.2.8. CommitAccept

[[TODO]]

Please find more information in the following code excerpt:

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

3.2.2.9. CommitAcceptForGroup

[[TODO]]

Please find more information in the following code excerpt:

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

3.2.2.10. GroupTrustThisKey

[[TODO]]

Please find more information in the following code excerpt:

```
// default: security=trusted only
message GroupTrustThisKey 11 {
    field Hash key;
}
```

3.2.2.11. GroupKeys

[[TODO]]

Please find more information in the following code excerpt:

```
// trust in future
message GroupKeys 12, security=attach_own_keys {
    field IdentityList ownIdentities;
}
```

3.2.2.12. OwnKeysOfferer

[[TODO]]

Please find more information in the following code excerpt:

```
message OwnKeysOfferer 13, security=attach_own_keys {
    field IdentityList ownIdentities;
}
```

3.2.2.13. OwnKeysRequester

[[TODO]]

Please find more information in the following code excerpt:

```
message OwnKeysRequester 14, security=attach_own_keys {
    field IdentityList ownIdentities;
}
```

3.2.3. Example Messages

[[TODO]]

4. Security Considerations

[[TODO]]

5. Privacy Considerations

[[TODO]]

6. IANA Considerations

This document has no actions for IANA.

7. Acknowledgments

The authors would like to thank the following people who have provided significant contributions to actual Running Code and the development of this document: Volker Birk and Krista Bennett.

Furthermore, the authors would like to thank the following people who provided helpful comments and suggestions for this document: Claudio Luck, Damian Rutz, Kelly Bristol, and Nana Karlstetter.

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]

8. References

8.1. Normative References

[I-D.birk-pep]
Marques, H. and B. Hoeneisen, "pretty Easy privacy (pEp): Privacy by Default", draft-birk-pep-03 (work in progress), March 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>.

8.2. Informative References

[I-D.birk-pep-trustwords]
Birk, V., Marques, H., and B. Hoeneisen, "IANA Registration of Trustword Lists: Guide, Template and IANA Considerations", draft-birk-pep-trustwords-03 (work in progress), March 2019.

[I-D.marques-pep-handshake]
Marques, H. and B. Hoeneisen, "pretty Easy privacy (pEp): Contact and Channel Authentication through Handshake", draft-marques-pep-handshake-02 (work in progress), March 2019.

[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. Document Changelog

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

- draft-hoeneisen-pep-keysync-00:

  * Initial version

Appendix B. Open Issues

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

- Resolve several TODOs / add missing text

Authors' Addresses

Bernie Hoeneisen
Ucom Standards Track Solutions GmbH
CH-8046 Zuerich
Switzerland

Phone: +41 44 500 52 40
Email: bernie@ietf.hoeneisen.ch (bernhard.hoeneisen AT ucom.ch)
URI: https://ucom.ch/

Hernani Marques
pEp Foundation
Oberer Graben 4
CH-8400 Winterthur
Switzerland

Email: hernani.marques@pep.foundation
URI: https://pep.foundation/