anima
Internet-Draft
Intended status: Standards Track
Expires: 21 April 2026

E. Dijk, Ed.
IoTconsultancy.nl
M. Richardson
Sandelman Software Works
P. van der Stok
vanderstok consultancy
P. Kampanakis
Cisco Systems
18 October 2025

Join Proxy for Bootstrapping of Constrained Network Elements
draft-ietf-anima-constrained-join-proxy-18

Abstract

This document extends the constrained Bootstrapping Remote Secure Key Infrastructures (cBRSKI) onboarding protocol by adding a new network function, the constrained Join Proxy. This function can be implemented on a constrained node. The goal of the Join Proxy is to help new constrained nodes ("Pledges") securely onboard into a new IP network using the cBRSKI protocol. It acts as a circuit proxy for User Datagram Protocol (UDP) packets that carry the onboarding messages. The solution is extensible to support other UDP-based onboarding protocols as well. The Join Proxy functionality is designed for use in constrained networks, including IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN) based networks in which the onboarding authority server ("Registrar") may be multiple IP hops away from a Pledge. Despite this distance, the Pledge only needs to use link-local communication to complete cBRSKI onboarding. Two modes of Join Proxy operation are defined, stateless and stateful, to allow different trade-offs regarding resource usage, implementation complexity and security.

About This Document

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

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-anima-constrained-join-proxy/.

Discussion of this document takes place on the anima Working Group mailing list (mailto:anima@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/anima/. Subscribe at https://www.ietf.org/mailman/listinfo/anima/.

Source for this draft and an issue tracker can be found at https://github.com/anima-wg/constrained-join-proxy.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The Bootstrapping Remote Secure Key Infrastructure (BRSKI) protocol described in [RFC8995] provides a solution for a secure zero-touch (automated) onboarding of new, unconfigured devices. In the context of BRSKI, new devices, called "Pledges", are equipped with a factory-installed Initial Device Identifier (IDevID) [ieee802-1AR], and are enrolled into a network. BRSKI makes use of Enrollment over Secure Transport (EST) [RFC7030] with [RFC8366bis] signed vouchers to securely enroll devices. A Registrar provides the trust anchor of the network domain to which a Pledge enrolls.

[cBRSKI] defines a version of BRSKI that is suitable for constrained nodes ([RFC7228]) and for operation on constrained networks ([RFC7228]) including Low-Power and Lossy Networks (LLN) [RFC7102]. It uses Constrained Application Protocol (CoAP) [RFC7252] messages secured by Datagram Transport Layer Security (DTLS) [RFC9147] to implement the BRSKI functions defined by [RFC8995].

In this document, cBRSKI is extended such that a cBRSKI Pledge can connect to a Registrar via a constrained Join Proxy. In particular, this solution is intended to support IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN) [RFC4944] mesh networks. 6TiSCH networks are not in scope of this document since these use the CoJP [RFC9031] proxy mechanism.

The Join Proxy as specified in this document is one of the Join Proxy options referred to in Section 2.5.2 of [RFC8995] as future work.

However, in IP networks that require node authentication, such as those using 6LoWPAN [RFC4944], data to and from the Pledge will not be routable over the IP network before it is properly authenticated to the network. A new Pledge can initially only use a link-local IPv6 address to communicate with a mesh neighbor [RFC6775] until it receives the necessary network configuration parameters.

Before it can receive these parameters, the Pledge needs to be authenticated and authorized to onboard the network. This is done in cBRSKI through an end-to-end encrypted DTLS session with a domain Registrar.

When this Registrar is not a direct (link-local) neighbor of the Pledge but several hops away, the Pledge needs to discover a link-local neighbor that is operating as a constrained Join Proxy, which helps forward the DTLS messages of the session between Pledge and Registrar.

Because the Join Proxy is a regular network node that has already been onboarded onto the network, it can send IP packets to the Registrar which are then routed over one or more hops over the mesh network -- and potentially over other IP networks too, before reaching the Registrar. Likewise, the Registrar sends its response IP packets which are routed back to the Join Proxy over the mesh network.

Once a Pledge has enrolled onto the network in this manner, it can optionally be configured itself as a new constrained Join Proxy. In this role it can help other Pledges perform the cBRSKI onboarding process.

Two modes of operation for a constrained Join Proxy are specified:

1. A stateful Join Proxy that locally stores UDP connection state per Pledge.

2. A stateless Join Proxy that does not locally store UDP connection state, but stores it in the header of a message that is exchanged between the Join Proxy and the Registrar.

Similar to the difference between storing and non-storing Modes of Operations (MOP) in RPL [RFC6550], the stateful and stateless modes differ in the way that they store the state required to forward return UDP packets from the Registrar back to the Pledge.

2. Terminology

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

The following terms are defined in [RFC8366bis] and [RFC8995], and are used identically in this document: artifact, Circuit Proxy, Join Proxy, domain, imprint, Registrar, Pledge, and Voucher.

The term "installation" refers to all devices in the network and their interconnections, including Registrar, enrolled nodes (with and without constrained Join Proxy functionality) and Pledges (not yet enrolled).

(Installation) IP addresses are assumed to be routable over the whole installation network, except for link-local IP addresses.

The term "Join Proxy" is used in this document with the same definition as in [RFC8995]. However, in this document it refers specifically to a Join Proxy that can support Pledges to onboard using a UDP-based protocol, such as the cBRSKI protocol [cBRSKI]. This protocol operates over an end-to-end secured DTLS session between a Pledge and a cBRSKI Registrar.

The acronym "JPY" is used to refer to a new protocol and JPY message format defined by this document. The message can be seen as a "Join Proxy Yoke": connecting two data items and letting these travel together over a network.

Because UDP does not have the notion of a connection, the term "UDP connection" in this document refers to a pseudo-connection, whose establishment on the Join Proxy is triggered by receipt of a first UDP packet from a new Pledge source.

The term "endpoint" is used as defined in [RFC7252].

The terms "6LoWPAN Router" (6LR), "6LoWPAN Border Router" (6LBR) and "6LoWPAN link" are used as defined in [RFC6775].

Details of the IP address and port notation used in the Join Proxy specification are provided in Section 4.2.

3. Join Proxy Problem Statement and Solution

3.1. Problem Statement

As depicted in Figure 1, the Pledge (P), in a network such as a 6LoWPAN [RFC4944] mesh network can be more than one hop away from the Registrar (R) and it is not yet authenticated to the network. Also, the Pledge does not possess any key material to encrypt or decrypt link-layer data transmissions.

In this situation, the Pledge can only communicate one-hop to its neighbors, such as the constrained Join Proxy (J), using link-local IPv6 addresses and using no link-layer encryption. However, the Pledge needs to communicate with end-to-end security with a Registrar to authenticate and obtain its domain identity/credentials. In the case of cBRSKI, the domain identity is an X.509 certificate. Domain credentials may include key material for network access.

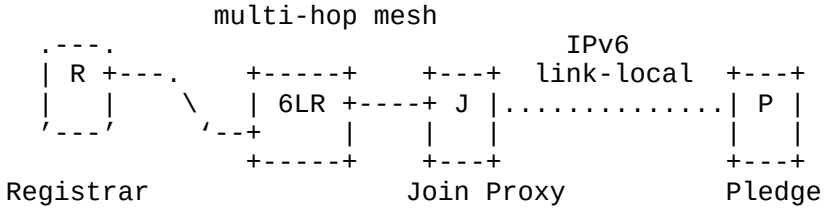

Figure 1: Multi-hop cBRSKI onboarding scenario in a 6LoWPAN mesh network

So one problem is that there is no IP routability between the Pledge and the Registrar, via intermediate nodes such as 6LoWPAN Routers (6LRs), despite the need for an end-to-end secured session between both.

Furthermore, the Pledge is not be able to discover the IP address of the Registrar because it is not yet allowed onto the network.

3.2. Solution

To overcome these problems, the constrained Join Proxy is introduced. This is specific functionality that all, or a specific subset of, authenticated nodes in an IP network can implement. When the Join Proxy functionality is enabled in a node, it can help a neighboring Pledge securely onboard the network.

The Join Proxy performs relaying of UDP packets from the Pledge to the intended Registrar, and relaying of the subsequent return packets. An authenticated Join Proxy can either be configured with the routable IP address of the Registrar, or it can discover this address as specified in this document. Other methods of Registrar discovery (not yet specified in this document) can also be easily added.

The Join Proxy acts as a packet-by-packet proxy for UDP packets between Pledge and Registrar. The cBRSKI protocol between Pledge and Registrar [cBRSKI] which this Join Proxy supports uses UDP messages with DTLS-encrypted CoAP payloads, but the Join Proxy as described here is unaware of these payloads. The Join Proxy solution can therefore be easily extended to work for other UDP-based protocols, as long as these protocols are agnostic to (or can be made to work with) the change of the IP and UDP headers performed by the Join Proxy.

In summary, the following steps are typically taken for the onboarding process of a Pledge:

1. Join Proxies in the network learn the IP address and UDP port of the Registrar.

2. A new Pledge arrives: it discovers one or more Join Proxies and selects one.

3. The Pledge sends a link-local UDP message to the selected Join Proxy.

4. The Join Proxy relays the message to the Registrar (and port) of step 1.

5. The Registrar sends a response UDP message back to the Join Proxy.

6. The Join Proxy relays the message back to the Pledge.

7. Step 3 to 6 repeat as needed, for multiple messages, to complete the onboarding protocol.

8. The Pledge uses its obtained domain identity/credentials to join the domain network.

To reach the Registrar in step 4, the Join Proxy needs to be either configured with a Registrar address or needs to dynamically discover a Registrar as detailed in Section 5.1. This configuration/discovery is specified here as step 1. Alternatively, in case of automated discovery it can also happen on-demand in step 4, at the moment that the Join Proxy has data to send to the Registrar.

3.3. Solution for Multiple Registrars

The solution description in Section 3.2 assumes there is only one Registrar service configured or discovered by a Join Proxy, defined by a single IP address and single UDP port.

However, there may be multiple Registrars present in a network deployment. There may be multiple Registrars supporting the exact same onboarding protocol, or multiple Registrars supporting different onboarding protocols, or a combination of both. Such cases are all supported by this specification, to enable redundancy, backward-compatibility, and introduction of new variants of onboarding protocols over time. Further information about the "BRSKI variants" concept can be found in [I-D.ietf-anima-brski-discovery].

See Section 4.6 for the specific requirements on the Join Proxy for supporting multiple Registrars or multiple onboarding protocol variants.

3.4. Forming 6LoWPAN Mesh Networks with cBRSKI

The Join Proxy has been specifically designed to set up an entire 6LoWPAN mesh network using cBRSKI onboarding. This section outlines how this process works and highlights the role that the Join Proxy plays in forming the mesh network.

Typically, the first node to be set up is a 6LoWPAN Border Router (6LBR) which will form the new mesh network and decide on the network’s configuration. The 6LBR may be configured using for example one of the below methods. Note that multiple methods may be used within the scope of a single installation.

1. Manual administrative configuration

2. Use non-constrained BRSKI [RFC8995] to automatically onboard over its high-speed network interface when it gets powered on.

3. Use cBRSKI [cBRSKI] to automatically onboard over its high-speed network interface when it gets powered on.

Once the 6LBR is enabled, it requires an active Registrar reachable via IP communication to onboard any Pledges. Once cBRSKI onboarding is enabled (either administratively, or automatically) on the 6BLR, it can support the onboarding of 6LoWPAN-enabled Pledges, via its 6LoWPAN network interface. This 6LBR may host the cBRSKI Registrar itself, but the Registrar may also be hosted elsewhere on the installation network.

At the time the Registrar and the 6LBR are enabled, there may be zero Pledges, or there may be already one or more installed and powered Pledges waiting - periodically attempting to discover a Join Proxy over their 6LoWPAN network interface.

A Registrar hosted on the 6LBR will, per [cBRSKI], make itself discoverable as a Join Proxy so that Pledges can use it for cBRSKI onboarding over a 6LoWPAN link (one hop). Note that only some of Pledges waiting to onboard may be direct neighbors of the Registrar/6LBR. Other Pledges would need their traffic to be relayed by Join Proxies across one or more enrolled mesh devices (6LR, see Figure 1) in order to reach the Registrar/6LBR. For this purpose, all or a subset of the enrolled Pledges start to act as Join Proxies themselves. Which subset is selected, and when the Join Proxy function is enabled by a node, is out of scope of this document.

The desired end state of the installation includes a network with a Registrar and all Pledges successfully enrolled in the network domain and connected to one of the 6LoWPAN mesh networks that are part of the installation. New Pledges may also be added by future network maintenance work on the installation.

Pledges employ link-local communication until they are enrolled, at which point they stop being a "Pledge". A Pledge will periodically try to discover a Join Proxy using for example link-local discovery requests, as defined in [cBRSKI]. Pledges that are neighbors of the Registrar will discover the Registrar itself (which is posing as a Join Proxy) and will be enrolled first, using cBRSKI. The Pledges that are not a neighbor of the Registrar will at first fail to find a Join Proxy. Later on, they will eventually discover a Join Proxy so that they can be enrolled with cBRSKI too. While this continues, more and more Join Proxies with a larger hop distance to the Registrar will emerge. The mesh network auto-configures in this way, such that at the end of the onboarding process, all Pledges are enrolled into the network domain and connected to the mesh network.

4. Join Proxy Specification

A Join Proxy can operate in two modes:

1. Stateful mode

2. Stateless mode

The advantages and disadvantages of the two modes are presented in Section 6.

4.1. Mode Implementation and Configuration Requirements

For a Join Proxy implementation on a node, there are three possible scenarios:

1. Both stateful and stateless modes are implemented. The Join Proxy can switch between these modes, depending on configuration and/or auto-discovery of Registrar(s) for each option.

2. Only stateful mode is implemented.

3. Only stateless mode is implemented.

Option 2 and 3 have the advantage of reducing code size, testing efforts and deployment complexity, but requires all devices in the deployment to standardize on the same choice.

A standard for a network-wide application or ecosystem profile, that integrates the Join Proxy functionality as defined in this document, MAY specify the use of any of these three options. It is expected that most deployments of constrained Join Proxies will be in the context of such standards and that these standards will be able to pick either option 2 or 3 based on considerations such as those in Section 6.

A Join Proxy that is not adhering to such an additional standard MUST implement both modes (option 1). A Join Proxy or Registrar not adhering to such additional standards is called "generic".

If a Join Proxy implements both modes but does not implement methods to discover available Registrars (for either method), then it MUST use only the mode that is currently configured for the network, or configured individually for the device. The method or profile that defines such a configuration is outside the scope of this document. If the mode is not configured and also can not be discovered automatically, then the device MUST NOT operate as a Join Proxy.

For a Join Proxy to be operational, the node on which it is running has to be able to communicate with a Registrar (that is, exchange UDP messages with it). Establishing this connectivity can happen fully automatically if the Join Proxy node first enrolls itself as a Pledge, and then discovers the Registrar IP address/port, and if applicable its desired mode of operation (stateful or stateless), through a discovery mechanism (see Section 5). Other methods, such as provisioning the Join Proxy are out of scope for this document but equally feasible. Such methods would typically be defined by a standard or ecosystem profile that integrates Join Proxy functionality. Such provisioning can also be fully automated, for example if the Registrar IP address/port are included in the network configuration parameters that are disseminated to each trusted network node.

Independent of the mode of the Join Proxy or its discovery/ configuration methods, the Pledge first discovers (see Section 5.2) and selects the most appropriate Join Proxy. From the discovery result, the Pledge learns a Join Proxy's link-local IP address and UDP join-port. Details of this discovery are defined by the onboarding protocol and are not in scope of this document. For cBRSKI, this is defined in Section 10 of [cBRSKI].

A generic cBRSKI Registrar by design necessarily implements the stateful mode, and it SHOULD implement support for Join Proxies operating in the stateless mode. Support for only the stateless mode is considered not to bring significant simplifications to a generic cBRSKI Registrar implementation. However, the generic cBRSKI Registrar MAY offer a configuration option to disable either the stateful or stateless mode, which can be useful in a particular deployment. A cBRSKI Registrar that is only implemented to support an aforementioned network-wide application or ecosystem profile MAY implement either stateful and/or stateless mode.

4.2. Notation

The following notation is used in this section in both text and figures:

* The colon (:) separates IP address and port number (<IP>:<port>).

* IP_P denotes the link-local IP address of the Pledge. For simplicity, it is assumed here that the Pledge only has one network interface.

* IP_R denotes the routable IP address of the Registrar.

* IP_Jl denotes the link-local IP address of the Join Proxy on the interface that connects it to the Pledge.

* IP_Jr denotes the routable IP address of the Join Proxy.

* p_P denotes the UDP port used by the Pledge for its onboarding/ joining protocol, which may be cBRSKI. The Pledge acts in a UDP client role, specifically as a DTLS client for the case of cBRSKI.

* p_Jl denotes the join-port of the Join Proxy.

* p_Jr denotes the client port of the Join Proxy that it uses to forward packets to the Registrar.

* p_R denotes the server port of the Registrar on which it serves the onboarding protocol, such as cBRSKI.

* p_Rj denotes the server port of the Registrar on which it serves the JPY protocol.

* JPY[H( ),C( )] denotes a JPY message, as defined by the JPY protocol, with header H and content C indicated in between the parentheses.

4.3. Stateful Join Proxy

In stateful mode, the Join Proxy acts as a UDP circuit proxy that does not change the UDP payload (called "data octets" in [RFC768]) but only rewrites the IP and UDP headers of each UDP packet it forwards between a Pledge and a Registrar.

The UDP flow mapping state maintained by the Join Proxy can be represented as a list of tuples, one for each active Pledge, as follows:

```
   Local UDP state              Routable UDP state     Time state
 (IP_P:p_P, IP_Jl:p_Jl) <===> (IP_Jr:p_Jr, IP_R:p_R)  (Exp-timer)
```

In case a Join Proxy has multiple network interfaces that accept Pledges, an interface identifier needs to be added on the leftmost tuple component. If a Join Proxy has multiple network interfaces to connect to (one or more) Registrars, an interface identifier needs to be added to the rightmost tuple component. Both of these are not shown further in this section, for better readability.

The establishment of the UDP connection state on the Join Proxy is solely triggered by receipt of a UDP packet from a Pledge with an IP_P:p_P link-local source and IP_Jl:p_Jl link-local destination for which no mapping state exists, and that is terminated by a connection expiry timer.

Figure 2 depicts an example DTLS session via the Join Proxy, to show how this state is used in practice. In this case the Join Proxy knows the IP address of the Registrar (IP_R) and the default CoAPS port (P_R = 5684) on the Registrar is used to access cBRSKI resources.

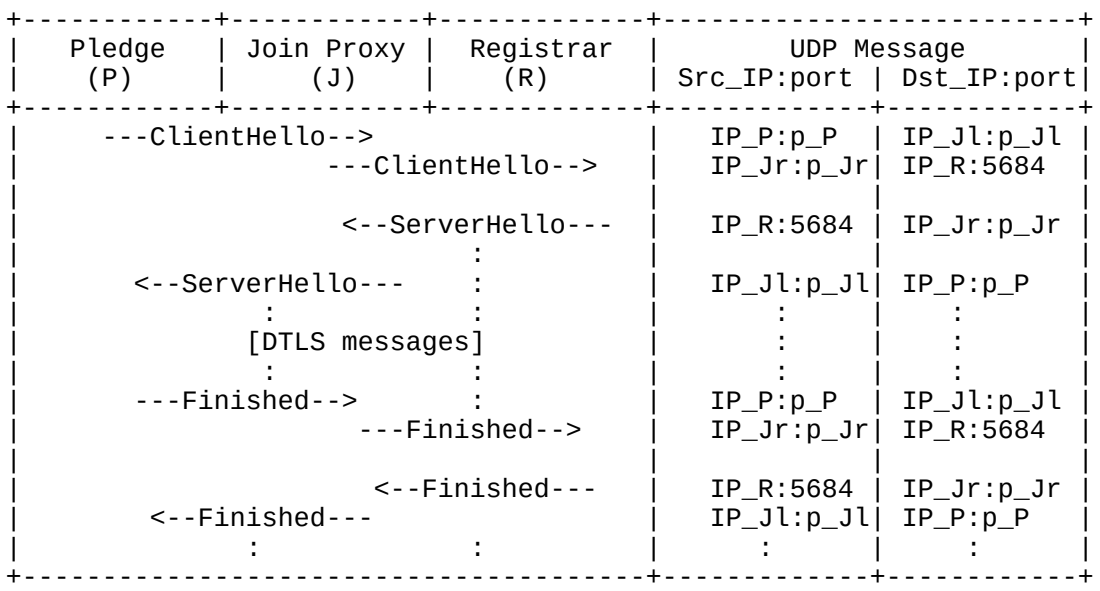

Figure 2: Example of the message flow of a DTLS session via a stateful Join Proxy

The Join Proxy MUST allocate a unique IP_Jr:p_Jr for every unique Pledge that it serves. This is typically done by selecting a unique available port P_Jr for each Pledge. Doing so enables the Join Proxy to correctly map the UDP packets received from the Registrar back to the corresponding Pledges. Also, it enables the Registrar to correctly distinguish multiple DTLS clients by means of IP address/port tuples.

The default timeout for clearing the state for a Pledge MUST be 30 seconds after the last relayed packet was sent on a UDP connection associated to that Pledge, in either direction. The default timeout MAY be overridden by another value that is either configured, or discovered in some way out of scope of this document.

When a Join Proxy receives an ICMP [RFC792] / ICMPv6 [RFC4443] error from the Registrar, this may signal a permanent change of the Registrar's IP address and/or port, or it may signal a temporary disruption of the network. In such case, the Join Proxy SHOULD send an equivalent ICMP error message (with same Type and Code) to the Pledge. The specific Pledge can be determined from the IP/UDP header information that is contained in the ICMP error message body, if included. In case the ICMP message body is empty, or insufficient information is included there, the Join Proxy does not send the ICMP error message to the Pledge because the intended recipient cannot be determined.

To protect itself and the Registrar against malfunctioning Pledges and/or denial of service (DoS) attacks, the Join Proxy SHOULD limit the number of simultaneous state tuples for a given IP_p to at most 2, and it SHOULD limit the number of simultaneous state tuples per network interface to at most 10.

When a new Pledge connection is received and the Join Proxy is unable to build new mapping state for it, for example due to the above limits, the Join Proxy SHOULD return an ICMP Type 1 "Destination Unreachable" error message with Code 1, "Communication with destination administratively prohibited".

4.4. Stateless Join Proxy

Stateless Join Proxy operation eliminates the need and complexity to maintain per-Pledge UDP connection mapping state on the proxy and the machinery to build, maintain and remove this mapping state. It also removes the need to protect this mapping state against DoS attacks and may also reduce memory and CPU requirements on the proxy.

Stateless Join Proxy operations work by introducing a new JPY message used in communication between Proxy and Registrar. This message will store the state "in the network". It consists of two parts:

* Header (H) field: contains state information about the Pledge (P) such as the link-local IP address and UDP port.

* Contents (C) field: the original UDP payload (data octets according to [RFC768]) received from the Pledge, or destined to the Pledge.

When the join proxy receives a UDP message from a Pledge, it encodes the Pledge’s link-local IP address, interface ID and UDP (source) port of the UDP packet into the Header field and the UDP payload into the Contents field and sends the packet to the Registrar from a fixed source UDP port. When the Registrar sends packets for the Pledge, it MUST return the Header field unchanged, so that the join proxy can decode the Header to reconstruct the Pledge’s link-local IP address, interace and UDP (destination) port for the return UDP packet. Figure 3 shows this per-packet mapping on the join proxy for a DTLS session.

The Registrar transiently stores the Header field information. The Registrar uses the Contents field to execute the Registrar functionality. When the Registrar replies, it wraps its DTLS message in a JPY message and sends it back to the Join Proxy. The Registrar SHOULD NOT assume that it can decode the Header Field of a received JPY message, it MUST simply replicate it when responding. The Header of a reply JPY message contains the original source link-local address and port of the Pledge from the transient state stored earlier and the Contents field contains the DTLS payload created by the Registrar.

On receiving the JPY message, the Join Proxy retrieves the two parts. It uses the Header field information to send a link-local UDP message containing the (DTLS) payload retrieved from the Contents field to a particular Pledge.

When the Registrar receives such a JPY message, it MUST treat the Header H as a single additional opaque identifier of all packets associated to a UDP connection with a Pledge. Whereas in the stateful proxy case, all packets with the same 4-tuple (IP_Jr:p_Jr, IP_R:p_R) belong to a single Pledge’s UDP connection, in the stateless proxy case only the packets with the same 5-tuple (IP_Jr:p_Jr, IP_R:p_Rj, H) belong to a single Pledge’s UDP connection. The JPY message Contents field contains the UDP payload of the packet for that Pledge’s UDP connection. Packets with different header H belong to different Pledge’s UDP connections.

In the stateless mode, the Registrar MUST offer the JPY protocol on a discoverable UDP port (p_Rj). There is no default port number available for the JPY protocol, unlike in the stateful mode where the Registrar can host all its services on the CoAPS default port (5684).

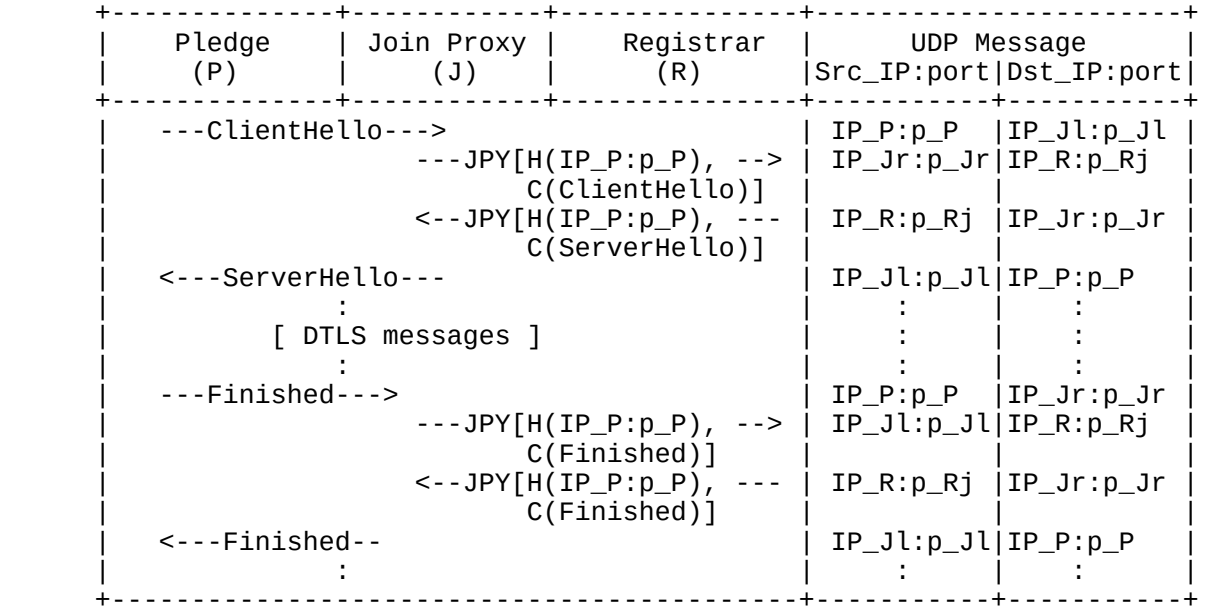

Figure 3: Example of the message flow of a DTLS session via a stateless Join Proxy

When a Join Proxy receives an ICMP [RFC792] / ICMPv6 [RFC4443] error from the Registrar, this may signal a permanent change of the Registrar's IP address and/or port, or it may signal a temporary disruption of the network.

Unlike a stateful Join Proxy, the stateless Join Proxy cannot determine the Pledge to which this ICMP error should be mapped, because the JPY header containing this information is not included in the ICMP error message. Therefore, it cannot inform the Pledge of the specific error that occurred.

4.5. JPY Protocol and Messages

JPY messages are used by a stateless Join Proxy to carry required state information in the relayed UDP messages, such that it does not need to store this state in memory. JPY messages are carried directly over the UDP layer. So, there is no CoAP or DTLS layer used between the JPY messages and the UDP layer.

A Registrar that supports the JPY protocol also uses JPY message to return relayed UDP messages to the stateless Join Proxy, including the state information that it needs.

4.5.1. JPY Message Structure

Each JPY message consists of one CBOR [RFC8949] array with 2 elements:

1. The Header (H) with the Join Proxy's per-message state data: wrapped in a CBOR byte string. The state data SHOULD be at most 32 bytes.

2. The Content (C) field: the binary (DTLS) payload being relayed, wrapped in a CBOR byte string. The payload is encrypted. The Join Proxy cannot decrypt it and therefore has no knowledge of any transported (CoAP) messages, or the URI paths or media types within the CoAP messages.

Using CDDL [RFC8610], the CBOR array that constitutes the JPY message can be formally defined as:

```
jpy_message =
[
   jpy_header  : bstr,
   jpy_content : bstr,
]
```

Figure 4: CDDL representation of a JPY message

The jpy_header state data is to be reflected (unmodified) by the Registrar when sending return JPY messages to the Join Proxy. The header's internal representation is not standardized: it can be constructed by the Join Proxy in whatever way. It is to be used by the Join Proxy to record state for the included jpy_content field, which includes the information which Pledge the data in jpy_content came from.

This state data stored in the JPY message is similar to the "state object" mechanism described in Section 7.1 of [RFC9031]. However, since the CoAP protocol layer (if any) is inside the DTLS layer, so end-to-end encrypted between the Pledge and the Registrar, it is not possible for the Join Proxy to act as a CoAP proxy per Section 5.7 of [RFC7252].

Detailed examples of a complete JPY message are shown in Appendix A.

4.5.2. JPY Message Port Usage

For the JPY messages sent to the Registrar, the Join Proxy SHOULD use the same UDP source port and IP source address for the JPY messages sent on behalf of all Pledges.

Although a Join Proxy MAY vary the UDP source port, doing so creates more local state. A Join Proxy with multiple CPUs (unlikely in a constrained system, but possible) could, for instance, use different UDP source port numbers to demultiplex connections across CPUs.

4.5.3. JPY Message Overhead and MTU Size

The use of the JPY message CBOR encoding adds a 3-6 byte overhead on top of the data carried within the Header and Contents fields. The Header state data itself (up to 32 bytes) also adds an overhead on each UDP message exchanged between Join Proxy and Registrar. Therefore, a protocol using the stateless Join Proxy MUST use (UDP) payloads that are bounded in size, such that the maximum payload length used minus the maximum overhead size (38 bytes) stays below the MTU size of the network. cBRSKI is designed to work even for the minimum IPv6 MTU of 1280 bytes, by configuring the DTLS maximum fragment length and using CoAP blockwise transfer for large resource transfers [cBRSKI].

At the CoAP level, using the cBRSKI [cBRSKI] and the EST-CoAPS [RFC9148] protocols, the CoAP blockwise options [RFC7959] are often used to split large payloads into multiple data blocks. The Registrar and the Pledge MUST select a block size that would allow the addition of the JPY message structure without violating MTU sizes.

4.5.4. JPY Message Security

Application or ecosystem standards adopting the stateless Join Proxy need to determine if there is the potential for attacks originating from the trusted network side of the Join Proxy. Such attacks would involve senders other than a trustworthy Registrar sending packets to the Join Proxy, impersonating the trusted Registrar by using its source address and port. In many well-designed solutions, this attack vector can be excluded because IP source addresses are verified. For example, in Autonomic Networking Infrastructure (ANI) networks, the Autonomic Control Plane (ACP) ([RFC8994]) ensures that only trustworthy nodes can communicate amongst each other. In an ACP, compromising an ACP node may be as hard as compromising the Registrar itself. Likewise, in many Wi-Fi mesh networks and 6LoWPAN mesh networks, link-layer security is applied and claimed to achieve similar levels of secure and trusted communication within the scope of the mesh.

For stateless Join Proxies that only operate in such secured network environments, it can be sufficient to only accept JPY messages originating from a Registrar's IP address and port, and not use any additional encryption or integrity protection of the JPY header. The Registrar's IP address and port are configured on the Join Proxy, or discovered by the Join Proxy, for sending JPY messages.

Generic stateless Join Proxies on the other hand can not assume any such additional security measures for the network that connects the Proxy to the Registrar. For example, a generic Join Proxy's network connection to a Registrar may pass through a lightly protected enterprise network, such as a university or campus network, without additional security. Therefore, a generic stateless Join Proxy SHOULD encrypt and integrity-protect the state data prior to wrapping it in a CBOR byte string in jpy_header.

It SHOULD be encrypted with a symmetric key known only to the Join Proxy itself. When the Join Proxy attempts to decrypt a receiver jpy_header byte string, and either the decryption or the integrity check fails, it MUST silently discard the JPY message.

The symmetric key need not persist on a long-term basis, and MAY be changed periodically. Because a key change during an onboarding attempt of a Pledge could lead to DTLS retransmissions, or even failure of the onboarding attempt, it is RECOMMENDED to change the key infrequently: for example every 24 hours.

4.5.5. Example Format for JPY Header Data

A typical JPY message header format, prior to encryption, could be constructed using the following binary data structure (expressed in C style notation):

```
struct jpy_header_plaintext {
    uint8_t  family;   // Only valid in the range 0...1
    uint8_t  ifindex;  // Only valid in the range 0...MAX_INTERFACES
    uint16_t srcport;  // Only valid > 0
    uint8_t  iid[8];
    uint32_t zero;     // Only valid == 0
};
```

This is illustrative only: the format of the data inside jpy_header is not subject to standardization and may vary across Pledges. It may for example use a CBOR array encoding, formally defined and constrained using CDDL [RFC8610].

The data structure stores the Pledge's UDP source port (srcport), the IID bits of the Pledge's originating IPv6 link-Local address (iid), the IPv4/IPv6 family (as a uint8 value 0 or 1) and an interface index (ifindex) to provide the link-local scope for the case that the Join Proxy has multiple network interfaces. The zero field is both for integrity protection and padding. It is always value zero (before encryption) on sending and MUST be zero after decryption on reception.

The resulting plaintext size is 16 bytes. This size fits into a single AES128 CBC block for instance, resulting in a 16 byte block of encrypted state data, jpy_header_ciphertext. Due to the way that CBC encryption mixes all the contents of a block together, an attacker that modifies any bit of this block will most likely change one of the zero bits in the family and/or zero fields as well.

This jpy_header_ciphertext data is then wrapped in a CBOR byte string to form the jpy_header element. This results in a jpy_header CBOR element of 17 bytes which includes a 1-byte overhead to encode the data as a CBOR byte string of length 16.

Note: when IPv6 is used only the lower 64-bits of the source IPv6 address need to be recorded, because they must be by design all IPv6 link-Local addresses, so the upper 64-bits are just "fe80::" and can be elided. For IPv4, a link-Local IPv4 address [RFC3927] would be used, and it would always fit into the 64 bits of the iid field. On link types where the Interface IDentifier (IID) is not 64-bits, a different field size for iid will be necessary.

Replay protection is not included in this example security solution, because the regular transport layers of cBRSKI and BRSKI, respectively UDP and TCP, also do not provide replay protection. Rather, replay protection is handled by the higher layer protocol, respectively DTLS and TLS. If replay attack protection is desired, AES with GCM [RFC5288] SHOULD be used.

Detailed examples of a complete JPY message are shown in Appendix A.

4.5.6. Processing by Registrar

On reception of a JPY message by the Registrar, the Registrar MUST verify that the number of CBOR array elements is 2 or more. To implement this specification, only the first two elements are used.

The data in the jpy_content field must be provided as input to a DTLS library [RFC9147], which along with the 5-tuple defined in Section 4.4 provides enough information for the Registrar to pick an appropriate (active) client context. Note that the same UDP socket

will need to be used for multiple DTLS flows, which is atypical for how DTLS usually uses sockets. The jpy_context field can be used to select an appropriate DTLS context, as DTLS headers do not contain any kind of per-session context. The jpy_context field needs to be linked to the DTLS context, and when a DTLS message need to be sent back to the client, the jpy_context needs to be included in a JPY message along with the DTLS message in the jpy_content field.

4.6. Handling Multiple Registrars

In a network deployment there MAY be multiple Registrar hosts present, each host operating one or more Registrar service(s). Regardless of the number of (physical or logical) hosts, each of these Registrar services is considered a separate Registrar. One or more of these Registrars MAY be configured in a Join Proxy, by a method out of scope of this specification. Also one or more of these Registrars MAY be found by a Join Proxy using its discovery method(s).

The Join Proxy is not necessarily aware of all onboarding protocol variants that are enabled in its network. Specifically, it may not be aware of the expected communication timing characteristics for the onboarding protocol that it is providing its proxy function for. Therefore, the final selection of onboarding protocol and Registrar is left to the Pledge and not to the Join Proxy. Also the determination of "onboarding progress" and whether the Registrar is considered responsive or not is left to the Pledge performing the onboarding protocol. This is consistent with Section 4.1 of [RFC8995] which defines how a BRSKI Pledge attempts onboarding via multiple Join Proxies and defines the related retry and switching behaviors.

If a Join Proxy discovers more Registrars than it can simultaneously offer to Pledges, given its resource limits or implementation-defined limits, then the Join Proxy MUST select from the discovered Registrars in an implementation-defined manner. Future work such as [I-D.ietf-anima-brski-discovery] may define a selection process for this case.

As an example, a network deployment might include a single Registrar host that offers two Registrar services: cBRSKI and a hypothetical "future BRSKI" (fuBRSKI). Both services are hosted on different UDP ports. Each Join Proxy is configured with these two Registrar services, and when a Pledge is sending CoAP discovery requests each Join Proxy in range will respond with both services in a CoAP discovery response. The Join Proxy is able to distinguish the properties of the two Registrar services by the differences in the CoRE Link Format parameters included in the two responded onboarding service descriptions.

5. Discovery

5.1. Join Proxy Discovers Registrar

In order to accommodate automatic configuration of the Join Proxy, it MUST discover the location and capabilities of the Registrar, in case this information is not configured already.

In BRSKI [RFC8995] the GeneRic Autonomic Signaling Protocol (GRASP) [RFC8990] protocol is supported for discovery of a BRSKI Registrar in an Autonomic Control Plane (ACP). However, this document does not target the ACP context of use. Therefore, the definition of how to use GRASP for discovering a cBRSKI Registrar in an ACP is left to future work such as [I-D.ietf-anima-brski-discovery].

Although multiple discovery methods can be supported in principle by a single Join Proxy, this document only defines one default method for a Join Proxy to discover a Registrar: using CoAP resource discovery queries [RFC6690] [RFC7252].

The CoAP discovery query to use depends on the intended mode of operation of the Join Proxy: stateless or stateful. A stateless Join Proxy needs to discover a UDP endpoint (address and port) that can accept JPY messages, supporting the jpy scheme. On the other hand, a stateful Join Proxy needs to discover a single CoAPS endpoint supporting the coaps scheme that offers the full set of cBRSKI Registrar resources.

5.1.1. Stateless Case

The stateless Join Proxy can discover the JPY protocol endpoint of the Registrar by sending a multicast CoAP GET discovery query to the "/.well-known/core" resource including a resource type (rt) query parameter "brski.rjp". The latter CoAP resource type is defined in Section 8.1.

Upon success, the return payload will contain the port of the Registrar on which the JPY protocol handler is hosted. The resource path returned in this payload is always the root (/) resource, the only resource currently defined for the JPY protocol. This exchange is shown below:

```
REQ: GET coap://[ff05::fd]/.well-known/core?rt=brski.rjp

RES: 2.05 Content
  Content-Format: 40 (application/link-format)
  Payload:
    <jpy://[ipv6_address]:port>;rt=brski.rjp
```

In this case, the multicast CoAP request is sent to the site-local "All CoAP Nodes" multicast IPv6 address ff05::fd. In some deployments, a smaller scope than site-local is more appropriate to reduce the network load due to this CoAP discovery traffic. For example, in a 6LoWPAN mesh network where a JPY protocol endpoint is always hosted on a 6LoWPAN Border Router (6LBR), the realm-local scope "All CoAP Nodes" address ff03::fd can be used.

The reason that the IPv6 address (field ipv6_address) is always included in the link-format result is that in the [RFC6690] link format, and per Section 3.2 of [RFC3986], the authority component cannot include only a port number but has to include also the IP address.

The returned port is expected to process the encapsulated JPY messages described in Section 4.5. The scheme is jpy, described in Section 8.2, and not regular coaps because the JPY messages effectively form a new protocol that encapsulates CoAPS messages.

5.1.2. Stateful Case

The stateful Join Proxy can discover the Registrar’s cBRSKI resource set by sending a multicast CoAP GET discovery query to the "/.well-known/core" resource including a resource type (rt) query parameter "brski". The latter CoAP resource type is defined in [cBRSKI].

Upon success, the return payload will contain the URI path and port of the Registrar on which the cBRSKI resources are hosted. This exchange is shown below:

```
REQ: GET coap://[ff05::fd]/.well-known/core?rt=brski

RES: 2.05 Content
  Content-Format: 40 (application/link-format)
  Payload:
    <coaps://[ipv6_address]:port/uri_path>;rt=brski
```

The port field and its preceding colon are optionally included: if elided, the default CoAPS port 5684 is implied. The uri_path field may be a single CoAP URI path resource label, or it may be a hierarchy of resources. For efficiency, it is RECOMMENDED for the Registrar to configure the URI path as short as possible, for example b.

Note that the Join Proxy does not use the returned uri_path information, while it uses the ipv6_address and port information for its relaying operations.

5.1.3. Examples

A Registrar with address 2001:db8:0:abcd::52, with the JPY protocol hosted on port 7634, and the CoAPS resources hosted on default port 5684 could for example reply to a multicast CoAP query of a stateful Join Proxy as follows:

```
REQ: GET coap://[ff05::fd]/.well-known/core?rt=brski

RES: 2.05 Content
  Content-Format: 40 (application/link-format)
  Payload:
    <coaps://[2001:db8:0:abcd::52]/b>;rt=brski
```

The same Registrar could for example reply to a multicast CoAP query of a stateless Join Proxy as follows:

```
REQ: GET coap://[ff05::fd]/.well-known/core?rt=brski.rjp

RES: 2.05 Content
  Content-Format: 40 (application/link-format)
  Payload:
    <jpy://[2001:db8:0:abcd::52]:7634>;rt=brski.rjp
```

In these examples, the Join Proxy in a specific mode of operation (stateful or stateless) only queries for those cBRSKI services that it minimally needs to perform the Join Proxy function in that mode. For this reason, wildcard queries (such as rt=brski*) are not sent.

5.2. Pledge Discovers Join Proxy

Regardless of whether the Join Proxy operates in stateful or stateless mode, it is discovered by the Pledge identically. Section 10 of [cBRSKI] defines the details of the CoAP discovery request sent by the Pledge.

A Join Proxy implementation by default MUST support this discovery method. If there is another method configured, by some means outside of the scope of this document, the default method MAY be deactivated.

The join-port of the Join Proxy is discovered by sending a multicast GET request to "/.well-known/core" including a resource type (rt) parameter with the value "brski.jp". This value is defined in Section 8.1. Upon success, the return payload will contain the join-port.

The resource type (rt) "brski.jp" exclusively pertains the empty path resource, and signals that under this root the BRSKI/EST resources of a remote Registrar can be found deeper down in the resource hierarchy under .well-known/brski and .well-known/est.

The meta-example below shows the discovery of the join-port (field join_port) of the Join Proxy:

```
REQ: GET coap://[ff02::fd]/.well-known/core?rt=brski.jp

RES: 2.05 Content
  Content-Format: 40 (application/link-format)
  Payload:
    <coaps://[IP_address]:join_port>;rt=brski.jp
```

In actual examples based on this, the field IP_address would contain the link-local IP address of the Join Proxy and the field join_port would contain the join-port value as a decimal number.

Note that the join_port field and preceding colon MAY be absent in the discovery response: this indicates that the join-port is the default CoAPS port 5684.

In the returned CoRE link format document, discoverable port numbers are usually returned for the Join Proxy resource in the <URI-Reference> of the link (see Section 5.1 of [RFC6690] for details).

5.3. Pledge Discovers Multiple Join Ports

A Pledge MUST be able to handle multiple join-ports being returned in a discovery response sent by a Join Proxy. This can happen if the network supports multiple Registrars and/or multiple Registrar-services as defined in Section 4.6. Then, each Registrar gets assigned its own join-port (up to a limit imposed by Join Proxy implementation) so that a Pledge is enabled to failover to another Registrar if a prior onboarding attempt fails.

How the Pledge selects between the onboarding services matching its query, is implementation-specific and out of scope of this document.

Discovery of multiple Registrars works in the same way as discovery of a single Registrar as defined in Section 5.2, except that multiple links are returned in the CoRE Link Format document.

The meta-example below shows the discovery of two join-ports (fields join_port1 and join_port2) on a Join Proxy, each associated to a different cBRSKI protocol variant, defined by two CoRE Link Format links:

```
REQ: GET coap://[ff02::fd]/.well-known/core?rt=brski.jp

RES: 2.05 Content
  Content-Format: 40 (application/link-format)
  Payload:
    <coaps://[IP_address]:join_port1>;rt=brski.jp,
    <coaps://[IP_address]:join_port2>;rt=brski.jp;
                          param1=value1;param2=value2
```

In actual examples based on this, the field IP_address would contain the link-local IP address of the Join Proxy and the fields join_port1 and join_port2 would contain distinct decimal port number values.

The parameter values (param1 and param2) are included for illustrative purposes. In a real example, these would contain Link Format parameters specifically defined for the brski.jp resource type. Such parameters may be defined in future work ([I-D.ietf-anima-brski-discovery]). These parameters, if understood by the Pledge, help in selecting the optimal matching onboarding protocol variant of cBRSKI. If the Pledge does not understand these parameters, it can select any one of the two join-ports for cBRSKI onboarding. If the attempt subsequently fails, the Pledge repeats the attempt using the other discovered join-port as defined by [cBRSKI].

6. Comparison of Stateless and Stateful Modes

The stateful and stateless mode of operation for the Join Proxy each have their advantages and disadvantages. This section helps operators and/or profile-specifiers to make a choice between the two modes based on the available device resources and network bandwidth.

Stateful mode introduces the complexity of maintaining per-connection state, which can increase processing and memory requirements on the proxy compared to the stateless mode under ideal conditions. Additionally, it opens up a wider range of potential implementation challenges in the presence of misbehaving or malicious Pledges. For example: How can state be effectively limited? How can malicious Pledges be detected—or at least prevented from negatively impacting non-malicious nodes? And so on.

If the proxy is deployed on nodes that support frequent and reliable software updates, then tailoring software enhancements based on the observed attack profile(s) in the deployment scenario is an effective way to improve and harden the implementation. However, many constrained devices either lack this software agility or intentionally avoid it. In such environments, stateless mode becomes advantageous, as it offloads most of the complex hardening responsibilities to the Registrar, allowing the proxy implementation to remain as lightweight as possible. Ultimately, a stateless proxy requires no more protective mechanisms than a basic packet-forwarding router.

The main concern for a stateless Join Proxy is the risk of forwarding an excessive number of packets to the Registrar, particularly over low-bandwidth connections such as 6LoWPAN links. Rate-limiting forwarded packets is the primary defense mechanism in such cases. All other Pledge-specific protections can be delegated to the Registrar, which is expected to have the necessary software agility to handle these.

The following table summarizes more comparison details.

Properties	Stateful mode	Stateless mode
State Information	The Join Proxy needs additional storage to maintain mappings between the address and port number of the Pledge and those of the Registrar.	No information is maintained by the Join Proxy. Registrar transiently stores the JPY message header.
Packet size	The size of a relayed message is the same as the original message.	Size of a relayed message is up to 38 bytes larger than the original: due to additional context data.
Technical complexity	The Join Proxy needs additional functions to maintain state information, and specify the source and destination addresses and ports of relayed messages.	Requires new JPY message structure (CBOR) in Join Proxy. The Registrar requires a function to process JPY messages.
Join Proxy Ports	Join Proxy needs discoverable join-port	Join Proxy needs discoverable join-port
Registrar Ports	Registrar can host on a single UDP port.	Registrar must host on two UDP ports: one for DTLS, one for JPY messages.

Table 1: Comparison between stateful and stateless Join Proxy mode

7. Security Considerations

For a Pledge using a Join Proxy, all the security considerations and requirements in Section 4.1 of [RFC8995] apply. While doing discovery of Join Proxies, the Pledge can be deceived by malicious Join Proxy announcements.

The subsequent communication of a Pledge with a Registrar that flows via the Join Proxy is end-to-end protected by DTLS.

A malicious Join Proxy has a number of relay/routing options for messages sent by a Pledge:

* It relays messages to a malicious Registrar. This is the same case as the presence of a "malicious Registrar" discussed in [RFC8995].

* It does not relay messages, or does not return the responses from the Registrar to the Pledge. This is equivalent to the case of a non-responding Registrar discussed in Section 4.1 of [RFC8995] and Section 5.1 of [RFC8995].

* It uses the returned responses of the Registrar for its own (attack) purposes. This is very unlikely due to the DTLS security.

* It uses the request from the Pledge to take the Pledge certificate and impersonate the Pledge. This is very unlikely because that requires it to acquire the private key of the Pledge, for an attack to be effective.

A malicious Pledge may also craft and send messages to a Join Proxy:

* It can construct an invalid DTLS or UDP message and send it to the open join-port of the Join Proxy. A Join Proxy will accept the message and relay to the Registrar without checking the payload. The Registrar will now parse the invalid message as DTLS protocol payload. Due to the security properties of DTLS, it is highly unlikely that this malicious payload will lead to message acceptance or to the Registrar’s malfunctioning. The Registrar of course MUST be prepared to receive invalid and/or non-DTLS payloads in this way. If the Pledge uses large UDP payloads, the attacker is able to misuse network resources. This way, a DoS attack could be performed by using multiple malicious Pledges, or using a single device posing as multiple Pledges.

For a malicious node that is either a neighbor of a Join Proxy, or is a router on the network path to the Registrar, and the node is part of the trusted network:

* It may sniff the messages routed by the Join Proxy. It is very unlikely that the malicious node can decrypt the DTLS payload. The malicious node may be able to read the inner data structure in the JPY Header field, if that is not encrypted. This does expose some information about the Pledge attempting to join, but this can be mitigated by the Pledge using a new (random) link-local address for each onboarding attempt.

In case the JPY Header is not encrypted, a malicious node has a number of options to craft a JPY message and send it to a stateless Join Proxy:

* It can craft a JPY message with header fields of its choice based on earlier observed contents of JPY messages sent by a stateless Join Proxy. In that case, the Join Proxy would accept the message and send the Content field data to a Pledge as a UDP message. Such a message could disrupt an ongoing DTLS session. It could also allow the attacker to access an unsecured UDP port that a Pledge may have exposed. For this reason, a Pledge MUST NOT accept messages on other UDP ports than its port used for onboarding while an onboarding attempt is ongoing.

It should be noted here that the JPY message CBOR array and the JPY Header field are not DTLS protected. When the communication between stateless Join Proxy and Registrar passes over an unsecure network, an attacker can change the CBOR array, and change the Header field if no encryption is used there. These concerns are also expressed in [RFC8974]. It is also pointed out here that the encryption by the source of the JPY Header, the Join Proxy, is a local matter. Similar to [RFC8974], the use of AES-CCM [RFC3610] with a 64-bit tag is recommended, combined with a sequence number and a replay window.

A "malicious Registrar" (see [RFC8995]) may also be unknowingly selected by a genuine (non-compromised) Join Proxy. This may happen when the malicious Registrar either modifies the network's Registrar address configuration or presents itself as a Registrar using the discovery method used in the network. If the discovery of Registrars is performed in an unsecured manner within the trusted network, it would allow the malicious Registrar to present itself as a Registrar candidate. CoAP discovery defined in Section 5) is, for example, defined without any transport-layer or application-layer security. A trusted Join Proxy may therefore relay a Pledge's messages to it.

It is the responsibility of a Pledge to monitor if an onboarding attempt with the selected Join Proxy and selected join-port on this Proxy (in case of multiple) is proceeding sufficiently quickly. If this is not the case, the Pledge needs to switch to another join-port and/or another Join Proxy to retry its onboarding attempt. See Section 4.6 for specification details on this.

In some installations, layer 2 (link layer) security is provided between all node pairs of a mesh network. In such an environment, in case all mesh nodes are trusted, and the Registrar is also located on the mesh network, and on-mesh attackers are not considered, then encryption of the JPY Header field as specified in this document is not necessary because the layer 2 security already protects it.

8. IANA Considerations

8.1. Resource Type Attributes Registry

This specification registers two new Resource Type (rt=) Link Target Attributes in the "Resource Type (rt=) Link Target Attribute Values" registry under the "Constrained RESTful Environments (CoRE) Parameters" registry group, per the [RFC6690] procedure.

Attribute Value: brski.jp
Description: Constrained Join Proxy for cBRSKI onboarding protocol.
Reference: [This RFC]

Attribute Value: brski.rjp
Description: cBRSKI Registrar Join Proxy endpoint that supports the JPY protocol.
Reference: [This RFC]

8.2. 'jpy' Scheme Registration

This specification registers a new URI scheme per [RFC7595] under the IANA "Uniform Resource Identifier (URI) Schemes" registry.

Scheme name: jpy
Status: Permanent
Applications/protocols that use this scheme name:
cBRSKI, constrained Join Proxy, JPY protocol
Contact: ANIMA WG
Change controller: IESG
References: [This RFC]

The scheme specification is provided below.

* Scheme syntax: identical to the coaps scheme as defined in Section 6.1 of [RFC7252].

* Scheme semantics: JPY protocol as defined in Section 4.5 of [This RFC].

* Encoding considerations: identical to the coaps scheme as defined in Section 12.4 of [RFC7252].

* Interoperability considerations: none.

* Security considerations: all of the security considerations for the coaps scheme as defined in Section 11.1 of [RFC7252] apply. In addition, users of this scheme should be aware that as part of the intended use, a UDP payload that was created under the coaps

scheme is embedded by a Join Proxy into a new UDP message conforming to the jpy scheme, without the Join Proxy being able to reconstruct which CoAPS URI was originally used by the sender of the CoAPS message, since most of the URI information is stored in DTLS-protected data. The receiving server can transform the JPY message sent under the jpy scheme back to a DTLS-encrypted CoAP message that uses the coaps scheme, by extracting the JPY message payload. However, any CoAP-related information not stored in the DTLS-protected data (such as data in UDP/IP headers) is subject to modification by the Join Proxy or other proxies in the communication path to the receiver. Any protocol transported in JPY messages MUST be resilient against such modifications.

8.3. Service Name and Transport Protocol Port Number Registry

This specification registers two service names under the IANA "Service Name and Transport Protocol Port Number" registry.

```
Service Name: brski-jp
Transport Protocol(s): udp
Assignee:  IESG <iesg@ietf.org>
Contact:  IESG <iesg@ietf.org>
Description: Bootstrapping Remote Secure Key Infrastructure
             constrained Join Proxy
Reference:   [This RFC]

Service Name: brski-rjp
Transport Protocol(s): udp
Assignee:  IESG <iesg@ietf.org>
Contact:  IESG <iesg@ietf.org>
Description: Bootstrapping Remote Secure Key Infrastructure
             Registrar join-port, supporting the 'jpy'
             scheme, used by stateless constrained Join Proxy
Reference:   [This RFC]
```

9. References

9.1. Normative References

[cBRSKI] Richardson, M., Van der Stok, P., Kampanakis, P., and E. Dijk, "Constrained Bootstrapping Remote Secure Key Infrastructure (cBRSKI)", Work in Progress, Internet-Draft, draft-ietf-anima-constrained-voucher-29, 18 October 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-anima-constrained-voucher-29>.

[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/rfc/rfc2119>.

[RFC4443] Conta, A., Deering, S., and M. Gupta, Ed., "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification", STD 89, RFC 4443, DOI 10.17487/RFC4443, March 2006, <https://www.rfc-editor.org/rfc/rfc4443>.

[RFC5288] Salowey, J., Choudhury, A., and D. McGrew, "AES Galois Counter Mode (GCM) Cipher Suites for TLS", RFC 5288, DOI 10.17487/RFC5288, August 2008, <https://www.rfc-editor.org/rfc/rfc5288>.

[RFC768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <https://www.rfc-editor.org/rfc/rfc768>.

[RFC792] Postel, J., "Internet Control Message Protocol", STD 5, RFC 792, DOI 10.17487/RFC0792, September 1981, <https://www.rfc-editor.org/rfc/rfc792>.

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

[RFC8366bis] Watsen, K., Richardson, M., Pritikin, M., Eckert, T. T., and Q. Ma, "A Voucher Artifact for Bootstrapping Protocols", Work in Progress, Internet-Draft, draft-ietf-anima-rfc8366bis-14, 1 April 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-anima-rfc8366bis-14>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/rfc/rfc8949>.

[RFC8995] Pritikin, M., Richardson, M., Eckert, T., Behringer, M., and K. Watsen, "Bootstrapping Remote Secure Key Infrastructure (BRSKI)", RFC 8995, DOI 10.17487/RFC8995, May 2021, <https://www.rfc-editor.org/rfc/rfc8995>.

[RFC9147] Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", RFC 9147, DOI 10.17487/RFC9147, April 2022, <https://www.rfc-editor.org/rfc/rfc9147>.

[RFC9148] van der Stok, P., Kampanakis, P., Richardson, M., and S. Raza, "EST-coaps: Enrollment over Secure Transport with the Secure Constrained Application Protocol", RFC 9148, DOI 10.17487/RFC9148, April 2022, <https://www.rfc-editor.org/rfc/rfc9148>.

9.2. Informative References

[I-D.ietf-anima-brski-discovery]
Eckert, T. T. and E. Dijk, "BRSKI discovery and variations", Work in Progress, Internet-Draft, draft-ietf-anima-brski-discovery-07, 20 July 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-anima-brski-discovery-07>.

[I-D.kumar-dice-dtls-relay]
Kumar, S. S., Keoh, S. L., and O. Garcia-Morchon, "DTLS Relay for Constrained Environments", Work in Progress, Internet-Draft, draft-kumar-dice-dtls-relay-02, 20 October 2014, <https://datatracker.ietf.org/doc/html/draft-kumar-dice-dtls-relay-02>.

[I-D.richardson-anima-state-for-joinrouter]
Richardson, M., "Considerations for stateful vs stateless join router in ANIMA bootstrap", Work in Progress, Internet-Draft, draft-richardson-anima-state-for-joinrouter-03, 22 September 2020, <https://datatracker.ietf.org/doc/html/draft-richardson-anima-state-for-joinrouter-03>.

[ieee802-1AR]
"IEEE 802.1AR Secure Device Identity", IEEE Standards Association, 2018, <https://standards.ieee.org/ieee/802.1AR/6995/>.

[RFC3610] Whiting, D., Housley, R., and N. Ferguson, "Counter with CBC-MAC (CCM)", RFC 3610, DOI 10.17487/RFC3610, September 2003, <https://www.rfc-editor.org/rfc/rfc3610>.

[RFC3927] Cheshire, S., Aboba, B., and E. Guttman, "Dynamic Configuration of IPv4 Link-Local Addresses", RFC 3927, DOI 10.17487/RFC3927, May 2005, <https://www.rfc-editor.org/rfc/rfc3927>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/rfc/rfc3986>.

[RFC4944] Montenegro, G., Kushalnagar, N., Hui, J., and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks", RFC 4944, DOI 10.17487/RFC4944, September 2007, <https://www.rfc-editor.org/rfc/rfc4944>.

[RFC6550] Winter, T., Ed., Thubert, P., Ed., Brandt, A., Hui, J., Kelsey, R., Levis, P., Pister, K., Struik, R., Vasseur, JP., and R. Alexander, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks", RFC 6550, DOI 10.17487/RFC6550, March 2012, <https://www.rfc-editor.org/rfc/rfc6550>.

[RFC6690] Shelby, Z., "Constrained RESTful Environments (CoRE) Link Format", RFC 6690, DOI 10.17487/RFC6690, August 2012, <https://www.rfc-editor.org/rfc/rfc6690>.

[RFC6775] Shelby, Z., Ed., Chakrabarti, S., Nordmark, E., and C. Bormann, "Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs)", RFC 6775, DOI 10.17487/RFC6775, November 2012, <https://www.rfc-editor.org/rfc/rfc6775>.

[RFC7030] Pritikin, M., Ed., Yee, P., Ed., and D. Harkins, Ed., "Enrollment over Secure Transport", RFC 7030, DOI 10.17487/RFC7030, October 2013, <https://www.rfc-editor.org/rfc/rfc7030>.

[RFC7102] Vasseur, JP., "Terms Used in Routing for Low-Power and Lossy Networks", RFC 7102, DOI 10.17487/RFC7102, January 2014, <https://www.rfc-editor.org/rfc/rfc7102>.

[RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <https://www.rfc-editor.org/rfc/rfc7228>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <https://www.rfc-editor.org/rfc/rfc7252>.

[RFC7595] Thaler, D., Ed., Hansen, T., and T. Hardie, "Guidelines and Registration Procedures for URI Schemes", BCP 35, RFC 7595, DOI 10.17487/RFC7595, June 2015, <https://www.rfc-editor.org/rfc/rfc7595>.

[RFC7959] Bormann, C. and Z. Shelby, Ed., "Block-Wise Transfers in the Constrained Application Protocol (CoAP)", RFC 7959, DOI 10.17487/RFC7959, August 2016, <https://www.rfc-editor.org/rfc/rfc7959>.

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/rfc/rfc8610>.

[RFC8974] Hartke, K. and M. Richardson, "Extended Tokens and Stateless Clients in the Constrained Application Protocol (CoAP)", RFC 8974, DOI 10.17487/RFC8974, January 2021, <https://www.rfc-editor.org/rfc/rfc8974>.

[RFC8990] Bormann, C., Carpenter, B., Ed., and B. Liu, Ed., "GeneRic Autonomic Signaling Protocol (GRASP)", RFC 8990, DOI 10.17487/RFC8990, May 2021, <https://www.rfc-editor.org/rfc/rfc8990>.

[RFC8994] Eckert, T., Ed., Behringer, M., Ed., and S. Bjarnason, "An Autonomic Control Plane (ACP)", RFC 8994, DOI 10.17487/RFC8994, May 2021, <https://www.rfc-editor.org/rfc/rfc8994>.

[RFC9031] Vuini, M., Ed., Simon, J., Pister, K., and M. Richardson, "Constrained Join Protocol (CoJP) for 6TiSCH", RFC 9031, DOI 10.17487/RFC9031, May 2021, <https://www.rfc-editor.org/rfc/rfc9031>.

Appendix A. Stateless Join Proxy JPY Message Examples

This appendix shows an example of a JPY message, sent by a stateless Join Proxy to a Registrar, and an example of the return JPY message sent by the Registrar. The DTLS payload itself, carried in the Content (C) field of the JPY message, is not shown in detail but abbreviated.

First, assume that a Pledge creates a CoAP request to a Join Proxy that it has just discovered and selected for performing [cBRSKI] onboarding.

This request may be a Pledge Voucher Request (PVR) as follows:

```
POST coaps://[fe80::1234:5678]:45965/.well-known/brski/rv
  Content-Format: 836
  Payload:
     <bytes of the COSE-signed PVR>
```

Because a DTLS session is not yet established at this point, the first step for the client is to send the DTLS Client Hello message to the Join Proxy's join-port 45965. When the Join Proxy receives this UDP packet, it creates a JPY message with the following UDP payload:

```
82                                      # array(2)
   50                                   # bytes(16)
      D01914BCC376A88FFECC50CA6017B0C1   #
   59 01AB                              # bytes(427)
      16FEFD000000000000000019E ...
      <further bytes of DTLS 1.2 Client Hello>
```

The same JPY message written in CBOR diagnostic notation [RFC8949] is:

```
[ h'd01914bcc376a88ffecc50ca6017b0c1' ,
  h'16fefd000000000000000019e' ... '3d45' ]
```

Above, the ellipsis ("...") notation in a CBOR diagnostic byte string denotes a further sequence of bytes that is not shown for brevity.

The first CBOR byte string wraps the 16 bytes of encrypted state information of the Header (H) field. The second CBOR byte string wraps the 427 bytes of the received DTLS message.

After the Registrar has processed the received JPY message, it sends a DTLS 1.2 Hello Verify Request in response to the received Client Hello message. This Hello Verify Request is wrapped in a new JPY message that it sends back to the Join Proxy:

```
82                                      # array(2)
   50                                   # bytes(16)
      D01914BCC376A88FFECC50CA6017B0C1   #
   58 3C                                # bytes(60)
      16FEFD000000000000000002F ...
      <further bytes of DTLS 1.2 Hello Verify Request>
```

The same JPY message in CBOR diagnostic notation is:

```
    [ h'd01914bcc376a88ffecc50ca6017b0c1' ,
      h'16fefd000000000000000002f' ... '66c1' ]
```

Acknowledgements

[I-D.richardson-anima-state-for-joinrouter] outlined the various options for building a constrained Join Proxy.

Many thanks for the comments by Bill Atwood, Carsten Bormann, Brian Carpenter, Spencer Dawkins, Toerless Eckert, Russ Housley, Ines Robles, Rich Salz, Jrgen Schnwlder, Malia Vuini, and Rob Wilton.

This document is very much inspired by text published earlier in [I-D.kumar-dice-dtls-relay]. Sandeep Kumar, Sye loong Keoh, and Oscar Garcia-Morchon are the co-authors of this document. Their draft text has served as a basis for this document.

Changelog

-17 to -18

* Changed JPY protocol scheme from coaps+jpy to more generic jpy and rephrased all related definitions (#80).
* Clarified that discovery responses from Join Proxy refer to the root CoAP resource (/) or root JPY resource (#79).
* Assigned editor role (#78).
* Editorial updates.

-16 to -17

* Added security consideration that a genuine Join Proxy may relay to a malicious Registrar (#33, #77).
* Added solution and specification sections on the use of multiple Registrars (#45, #65, #76).
* Added clarification that Registrar address(es) can be configured, or discovered (#76).
* Define conditions for implementing only a single Join Proxy mode - stateful or stateless (#69, #73)
* Improved JPY Header security by adding integrity protection (#74).
* Fixed format definition of example JPY Header (#74).
* Editorial updates.

-15 to -16

* Security considerations text reviewed and expanded with more attack types.
* Define CoAP discovery as default, remove GRASP/6TiSCH (#68).
* Abstract updated to describe higher-level concepts (#47).
* Applied Spencer's TSVART review comment 2022-05-16 in an improved manner.
* Applied Russ' review comments from IOTDIR review 2023-08-09.
* Rewrite Section 4.1 based on Russ' review (#48).
* Applied Toerless' review comments from WGLC (#63).
* Applied review comments of Bill Atwood of 2024-05-21.
* Clarify 'context payload' terminology (#49).
* Use shorter and consistent term for Join Proxy (#58).
* Appendix A corrected to use latest JPY message format.
* Author added.
* Update reference RFC8366 to RFC8366bis.
* Many editorial updates.

-13 to -15

* Various editorial updates and minor changes.

-12 to -13

* jpy message encrypted and no longer standardized

-11 to -12

* many typos fixed and text re-organized
* core of GRASP and CoAP discovery moved to constrained-voucher document, only stateless extensions remain

-10 to -11

* Join Proxy and Registrar discovery merged
* GRASP discovery updated
* ARTART review
* TSVART review

-09 to -10

* OPSDIR review
* IANA review
* SECDIR review
* GENART review

-07 to -09

* typos

-06 to -07

* AD review changes

-05 to -06

* RT value change to brski.jp and brski.rjp
* new registry values for IANA
* improved handling of jpy header array

-04 to -05

* Join Proxy and join-port consistent spelling
* some nits removed
* restructured discovery
* section
* rephrased parts of security section

-03 to -04

* mail address and reference

-02 to -03

* Terminology updated
* Several clarifications on discovery and routability
* DTLS payload introduced

-01 to -02

* Discovery of Join Proxy and Registrar ports

-00 to -01

* Registrar used throughout instead of EST server
* Emphasized Join Proxy port for Join Proxy and Registrar
* updated discovery accordingly
* updated stateless Join Proxy JPY header
* JPY header described with CDDL
* Example simplified and corrected

-00 to -00

* copied from vanderstok-anima-constrained-join-proxy-05

Authors' Addresses

Esko Dijk (editor)
IoTconsultancy.nl
Email: esko.dijk@iotconsultancy.nl

Michael Richardson
Sandelman Software Works
Email: mcr+ietf@sandelman.ca

Peter van der Stok
vanderstok consultancy
Email: stokcons@kpnmail.nl

Panos Kampanakis
Cisco Systems
Email: pkampana@cisco.com