Fraunhofer
ISST

FRAUNHOFER INSTITUTE FOR SOFTWARE AND SYSTEMS ENGINEERING

ISST REPORT

A LIGHTWEIGHT INDUSTRIAL DATA SPACE SENSOR CONNECTOR

INDUSTRIAL DATA SPACE

A LIGHTWEIGHT INDUSTRIAL DATA SPACE SENSOR CONNECTOR

Concept, Implementation and First Experiences

Dr. Bernhard Holtkamp

Dr. Jan Cirullies

Tim Dahlmanns

Sebastian Steinbuß

Fraunhofer-Institut für Software- und Systemtechnik, ISST
in Dortmund.

April 2018

Content

Figures

1 Introduction

Digitization is a megatrend and the Internet of Things (IoT) is an essential part of it. Market forecasts predict more than 20 billion of connected devices until 2020 [1]. For most business applications, a secure transfer of data from IoT devices to applications in the cloud is a crucial issue. The Industrial Data Space initiative has been set up in 2015 to develop concepts and technology for secure data transfers between data suppliers and data consumers, preserving data sovereignty of the data suppliers. The reference architecture model [2] describes the interaction of key components from different views. The model aims at covering a broad range of application scenarios and the resulting requirements. A complementary document [3] describes implementation aspects of a connector in more detail.

From an IoT perspective, however, the approach taken imposes heavier burdens on connectors for IoT devices than necessary for many applications. Therefore, we present a lightweight model for an IoT sensor connector that fits with the core principles of the Industrial Data Space.

The rest of the paper is organized as follows. In a first step, we describe the key concepts and components of the Industrial Data Space. The focus here is on connectors and security requirements. In a next step, we describe an IoT use case to illustrate the need for a lightweight sensor connector. Then we describe the architecture of such a connector and check it against Industrial Data Space requirements. A description of a connector implementation follows, including first practical experiences. The paper closes with a summary of results and an outlook to future work.

2
Industrial Data Space

Conceptually, the Industrial Data Space consists of the set of all connectors (i.e. data endpoints), brokers as a federated registry and an app store. Connectors are certified components that implement data endpoints at the sites of data suppliers and data consumers. Each connector has a unique identifier and has to enrol at a broker. A broker maintains a registry of participants, e.g. data suppliers and data providers, and of available data sources, based on a defined information model for participants and data services. An app store provides certified apps for data processing, pluggable into a connector.

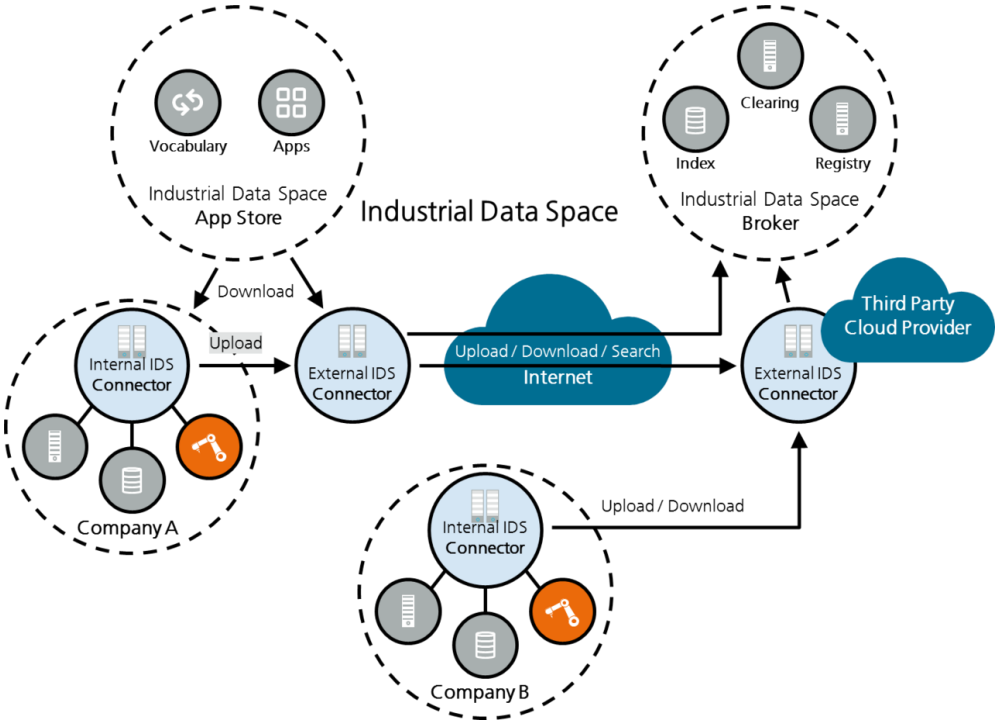

Figure 1 Industrial Data Space interaction of constituents

Figure 1 shows the key components and their interaction. The Industrial Data Space Connector is discussed in more detail in the following section.

2.1
Industrial Data Space Connector Model

The reference architecture model for the Industrial Data Space includes a reference model for Industrial Data Space Connectors. According to that model, a connector consists of a core part, a container for apps from the app store, and of a custom part for the interaction with the local IT infrastructure of the participant. Figure 2 shows the logical architecture of the connector model. In the following, we focus on the execution part and neglect connector configuration.

The primary task of a connector is the management of data transfers from and/or to a peer connector and from and/or to the participant site. A connector on the sending side generally uploads data from the local site (see Company A in figure 1), transfers

the data to the receiving connector that generally downloads the data to its local site (Company B in figure 1). According to the reference model, data transfer can take place via push or pull ([2, p. 20]). Data services in the custom container implement the data transfers between the connector and the local site. Data apps may be plugged into the app store container and may be used for data processing and transformation.

Industrial Data Space guidelines require that all connectors support the Industrial Data Space Information Model and that connectors are compatible with each other. The information model describes the infrastructure of the data space (participant, connector and data endpoint), the metadata to be exchanged with a data transfer (TransferHeader), the message format for messages sent to a broker (BrokerMessage) and the input/output behaviour of data apps.

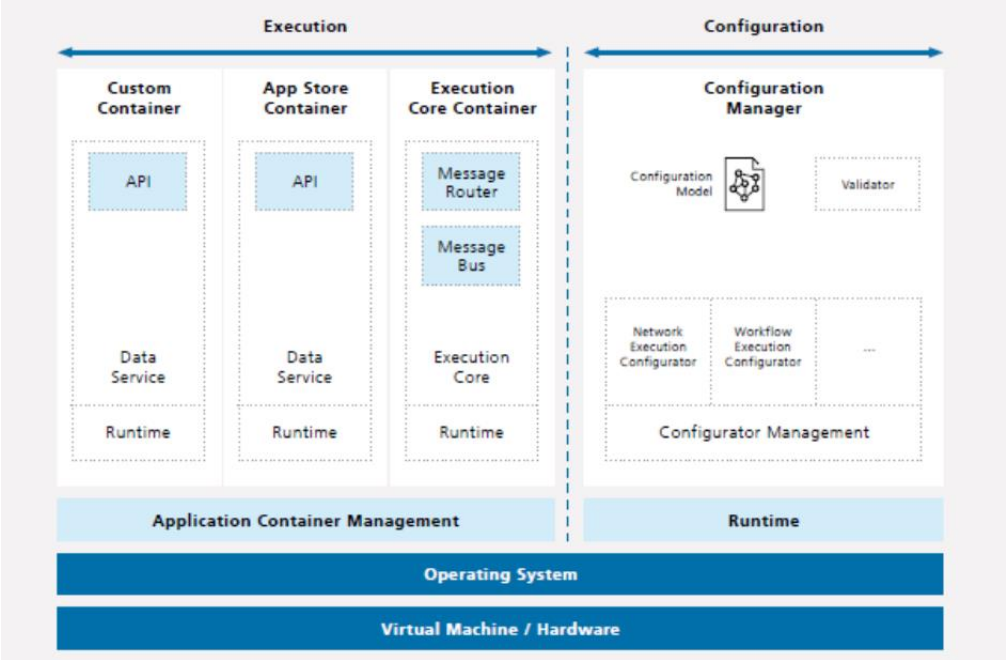

Figure 2 Logical architecture of an Industrial Data Space Connector [2]

In [3] more implementation relevant information is provided. Here it is stated that a connector typically runs on an operating system. For operating systems a set of requirements are defined that have to be fulfilled (e.g. admin user account, regular security patches, standard compliant TCP/IP stack, status monitoring, logging of operations). For the use of data apps from the app store, the Container Management System of the app store must be supported. Normally, a message router is in charge of receiving data and transferring them within the connector. Standard protocols like MQTT or REST are generally supported. A Message Bus can store data intermediately and can serve as a transaction mechanism for workflows. A broker provides a connector's metadata; alternatively, self-declaration by a connector is possible. Self-declaration requires that the connector already knows its unique identification.

2.2 Industrial Data Space Security Requirements

Security is a key issue for the Industrial Data Space to provide trust among the participants and data sovereignty to the data owner. Data sovereignty is defined as "as

a natural person's or legal entity's capability of being entirely self-determined with regard to its data" [9].

As data classification ranges from open accessibly to highly confidential, security profiles have been defined for connectors and each connector must provide its profile on request. Figure 3 shows the defined profiles.

Dimension	Implementation				
Trusted Platform Module (TPM)	Without TPM		TPM 1.2		TPM 2.0
Authentication	Without certificate	"self-signed" - certificate	Ca-based certificate of internal CA	Ca-based certificate of external CA (cross-certified)	Ca-based certificate of IDS CA
Container Management Layer (CML)	–		Baseline CML (e.g., Docker)		Hardened TrustX CML
Remote Attestation	No RAT		CML & Core Container Attestation		CML & Core Container & Container Attestation
Isolation / Execution Control	–		Basic Runtime Monitoring		Controlled Remote Execution
Software Assurance Level	Unknown software stack			IDS-certified software stack	

Figure 3 Connector security profiles

As can be seen from the figure, in the simplest case a connector does neither use a trusted platform module nor container management or execution control, applies authentication without certificate, does not support remote attestation and runs on an unknown software stack.

2.3 Industrial Data Space Message Structure

So far, the message structure is still under discussion within the Industrial Data Space project. The following proposal is based on an active use case and structures a message into three parts:

1. data transfer header,
2. payload,
3. signature.

In the following, we discuss the three parts.

2.3.1 Data transfer header

The following table shows the proposed attributes of the data transfer header.

Attribute	Type	Description
id	URL	ID of the message
authToken	AuthToken	Authentication token

sender	URL	Reference to self-declaration of the sender
receiver	URL	Reference to the self-declaration of the receiving participant
hashFunction	HashFunction	Algorithm that is used for calculating the hash value (digest) for the payload
digest	byte [ ]	Hash value (checksum) of the payload
CustomAttributes	Collection<Transfer Attribute>	Customer defined attributes
transferCreatedAt	XML Gregorian Calendar	Timestamp

Table 1 Data transfer header of an Industrial Data Space message

The message id is a UUID.

The authentication token has to be provided by an identity provider.[1] Usually, the token has a configurable short lifetime so that each message needs its own authentication token. Only in case of streaming several messages within a short time slot, the same token could be used for multiple messages. In some Industrial Data Space use cases, JSON Web Token[2] is used for the creation of authentication tokens. Such a token consists of three parts again: a header, the payload and a checksum.

The URL of the sender refers to the static self-declaration of the sender; the URL of the receiver refers to that of the receiving site accordingly.

The hashFunction attribute denotes the algorithm that is used for calculating the hash value for the payload. Typical candidates are SHA or MD5. The digest attribute contains the hash value of the payload that is provided by the algorithm when applied to the payload.

CustomAttributes is a collection of customer-defined attributes. For instance, an attribute could be defined (e.g. payloadType) that indicates the content of the payload (e.g. a zip folder or pdf file).

TransferCreatedAt is a timestamp of the message in ISO 8601 format.

2.3.2 Payload

The payload is a sequence of octets in UTF-8 encoding.

[1] At Fraunhofer ISST we use keycloak (www.keycloak.org) for this purpose.
[2] JSON Web Token (JWT): https://tools.ietf.org/html/rfc7519

2.3.3 Signature

The signature is a checksum of the data transfer header to make sure that the header is not manipulated. The following code snippet illustrates how the checksum (i.e. part three of a message) is calculated for the data transfer header.

```
    private static final String ALGORITHM = "SHA256withRSA";
    /*
     * Create signature for data transfer with given private key.
     *
     * @param dataTransfer the object which will be signed
     * @param key the private key used for signature creation
     * @return signature object
     */
    public SignatureData sign(DataTransfer dataTransfer, PrivateKey key) {
        Signature sig;
        try {
            sig = Signature.getInstance(ALGORITHM);
            sig.initSign(key);
            update(sig, dataTransfer);
            return new SignatureData(StringUtil.bytesToHex(sig.sign()),
ALGORITHM);
        } catch (NoSuchAlgorithmException | InvalidKeyException |
SignatureException ex) {
            throw new RuntimeException(ex);
        }
    }

    /**
     * Prefix of private key, which needs to be removed prior to parsing
     */
    private static final String privateKeyPrefix = "-----BEGIN PRIVATE KEY--
---";

    /**
     * Suffix of private key, which needs to be removed prior to parsing
     */
    private static final String privateKeySuffix = "-----END PRIVATE KEY----
-";

    /**
     * Loads private key from PEM[1] file.
     *
     * @param filename filename of private key (located in resources)
     * @param algorithm the algorithm used for this key
     * @return private key in usable form
     */
    private PrivateKey filenameToPrivateKey(String filename, String
algorithm) {
        Resource resource = new ClassPathResource(filename);
        InputStream inputStream;
        try {
            inputStream = resource.getInputStream();
            String privateKeyData =
org.apache.commons.io.IOUtils.toString(inputStream);

            // remove pre-/suffix & decode
            privateKeyData = privateKeyData.replace(privateKeyPrefix, "")
                    .replace(privateKeySuffix, "")
                    .replace("\r\n", "")
                    .replace("\r", "")
                    .replace("\n", "");
                    byte[] decodedPrivateKey =
        Base64.getDecoder().decode(privateKeyData);

            PKCS8EncodedKeySpec keySpec = new
PKCS8EncodedKeySpec(decodedPrivateKey);
```

[1] PEM: Privacy Enhanced Mail

```
            KeyFactory keyFactory = KeyFactory.getInstance(algorithm);

            // obtain & return private key
            return keyFactory.generatePrivate(keySpec);
        } catch (IOException e) {
            logger.error("Error loading private key file.", e);
        } catch (NoSuchAlgorithmException e) {
            logger.error("Unknown algorithm: " + algorithm + ".", e);
        } catch (InvalidKeySpecException e) {
          logger.error("Could not obtain private key for given keySpec", e);
        }
        // in case of error
        return null;
    }
```

Figure 4 Code sample for checksum generation

2.4 Messaging

We assume that senders and receivers have sent their self-declarations to an Industrial Data Space Broker. We further assume that the participant knows the public key, e.g. passed at registration time or fetched after registration.

2.4.1 Sender

The sender has retrieved the URL of an appropriate receiver's self-declaration. Alternatively, the sender might know the URL of the receiver's self-declaration by definition.

To send an Industrial Data Space compliant message the sender has to perform the following steps:

1. A UUID is generated and the message id attribute is set to the UUID
2. The sender attribute is set to the URL of the sender's self-declaration
3. The receiver attribute is set to the URL of the receiver's self-declaration
4. The CustomAttributes element payloadType is set to the MIME type of the payload (e.g. jpg)
5. The hashFunction attribute is set to a selected hash function (e.g. SHA256withRSA)
6. The hash function is applied to a selected payload. The result is a checksum.
7. The digest attribute is set to calculated checksum of the payload
8. A timestamp is created and the transferCreatedAt attribute is set to the timestamp value
9. A signature is created for the data transfer header. Each connector has its own certificate. This certificate is used to generate the signature for the data transfer header (see code example below).

The message elements are composed into an HTTP as multipart message[1].

```
From: Industrial Data Space sender <sender@acme.com>
To:  Industrial Data Space service provider <idssp@services.com>
Subject: Sample message
MIME-Version: 1.0
Content-type: multipart/mixed; boundary="Industrial Data Space message
boundary"

This is the preamble.  It is to be ignored, though it is a handy place for
mail composers to include an explanatory note to non-MIME compliant readers.

     -- Industrial Data Space message boundary

<Industrial Data Space data transfer header as RDF string in turtle
notation.
It MAY NOT end with a blank line before the boundary>
     -- Industrial Data Space message boundary
<Industrial Data Space payload, e.g. copied file,
It MAY NOT end with a blank line before the boundary >
-- Industrial Data Space message boundary
     Content-type: text/plain; charset=us-ascii

<Checksum (signature of the data transfer header), coded in Base64.
It DOES end with a line break.>

     -- Industrial Data Space message boundary --
End of Industrial Data Space message.
This is the epilogue. It is also to be ignored.
```

Figure 5 Industrial Data Space multipart message structure

Note that the multipart structure is very sensitive to line breaks.

The data transfer header is coded as RDF[2] structure in Turtle[3] notation. The following figure shows a sample data transfer header in Turtle notation, using references to the Industrial Data Space Information Model for data transfer header attributes.

```
<http://industrialdataspace.org/dataTransfer/fa2049be-a089-4b8a-b9e7-
744456031f7b>      a
<http://industrialdataspace.org/2016/10/ids/core#DataTransfer>;

<http://industrialdataspace.org/2016/10/ids/core#authToken>
<http://industrialdataspace.org/authToken/2360d56f-6f35-4a83-a568-
5d64c18d202e>;
<http://industrialdataspace.org/2016/10/ids/core#hashFunction>
<http://industrialdataspace.org/2016/10/ids/ext/hashfunction#SHA-512>;
<http://industrialdataspace.org/2016/10/ids/core#payloadDigest>
"ko0+TuFXfpHwkVfo6s1CVITGAv7txskPYyHrYMx0ueZVGWzEisMO5BnC9wW9LUNHMq9HXL4ZvuH
lBWL51cwcfg=="
^^<http://www.w3.org/2001/XMLSchema#base64Binary>;
<http://industrialdataspace.org/2016/10/ids/core#receiver>
<http://www.isst.fraunhofer.de/Broker>;
<http://industrialdataspace.org/2016/10/ids/core#sender>
<http://www.isst.fraunhofer.de/Broker>;
<http://industrialdataspace.org/2016/10/ids/core#transferCreatedAt> "2018-
02-19T10:49:43.797Z"^^<xsd:dateTime>;

<http://jrdfb.iais.fraunhofer.de/vocab/classMapping>
        [<http://industrialdataspace.org/2016/10/ids/core#DataTransfer>
"de.fraunhofer.iais.eis.DataTransferImpl";
        <http://industrialdataspace.org/2016/10/ids/core#authToken>
"de.fraunhofer.iais.eis.AuthTokenImpl";
```

[1] See https://www.w3.org/Protocols/rfc1341/7_2_Multipart.html

[2] https://www.w3.org/RDF/

[3] https://www.w3.org/TR/turtle/

```
        <http://industrialdataspace.org/2016/10/ids/core#hashFunction>
"de.fraunhofer.iais.eis.HashFunction";
        <http://industrialdataspace.org/2016/10/ids/core#payloadDigest>
"[B";
        <http://industrialdataspace.org/2016/10/ids/core#receiver>
"java.net.URL";
        <http://industrialdataspace.org/2016/10/ids/core#sender>
"java.net.URL";
        <http://industrialdataspace.org/2016/10/ids/core#transferCreatedAt>
"org.apache.xerces.jaxp.datatype.XMLGregorianCalendarImpl";
        <http://jrdfb.iais.fraunhofer.de/vocab/isMappingRoot>
"true"
        ] .
<http://industrialdataspace.org/authToken/2360d56f-6f35-4a83-a568-
5d64c18d202e>      a
<http://industrialdataspace.org/2016/10/ids/core#AuthToken>;
<http://industrialdataspace.org/2016/10/ids/core#tokenValue>
"
eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJQU3hueS1yQ01jX0xlQjZEOGV6
WVNXVWV5X0ExMEZZcFg1aktaRUlPUTlNIn0.eyJqdGkiOiJjOTZiNTkxNS1jMWZhLTRmYTAtYjk3
ZC1mNmRlNWNjYTI2Y2MiLCJleHAiOjE1MTkwMzc2ODEsIm5iZiI6MCwiaWF0IjoxNTE5MDM3Mzgx
LCJpc3MiOiJodHRwOi8vMTkyLjE2OC4yMTEuMTExOjgwODAvYXV0aC9yZWFsbXMvQnJva2VyIiwi
YXVkIjoiQnJva2VyX0ludGVybmFsIiwic3ViIjoiZDViN2VkY2MtNjNkMC00ZDIyLTgxNTUtZjJi
MWExNzRlNmY2IiwidHlwIjoiQmVhcmVyIiwiYXpwIjoiQnJva2VyX0ludGVybmFsIiwiYXV0aF90
aW1lIjowLCJzZXNzaW9uX3N0YXRlIjoiZTlmZjViNmEtNGZmNi00YzBkLTliNjEtN2VhMTRlZmM4
ZGY4IiwiYWNyIjoiMSIsImNsaWVudF9zZXNzaW9uIjoiNDM1ZDllYTgtMjA0NS00MzUzLWE0OTkt
YTdlODQ0ZDkwYzBiIiwiYWxsb3dlZC1vcmlnaW5zIjpbIiJdLCJyZWFsbV9hY2Nlc3MiOnsicm9s
ZXMiOlsidW1hX2F1dGhvcml6YXRpb24iLCJicm9rZXIiLCJicm9rZXItY2xpZW50Il19LCJyZXNv
dXJjZV9hY2Nlc3MiOnsiYWNjb3VudCI6eyJyb2xlcyI6WyJtYW5hZ2UtYWNjb3VudCIsIm1hbmFn
ZS1hY2NvdW50LWxpbmtzIiwidmlldy1wcm9maWxlIl19fSwiY2xpZW50SG9zdCI6IjE3Mi4xOC4w
LjEiLCJjbGllbnRJZCI6Ikyb2tlcl9JbnRlcm5hbCIsIm5hbWUiOiIiLCJwcmVmZXJyZWRfdXNl
cm5hbWUiOiJzZXJ2aWNlLWFjY291bnQtYnJva2VyIGludGVybmFsIiwiY2xpZW50QWRkcmVzcyI6
IjE3Mi4xOC4wLjEiLCJlbWFpbCI6InNlcnZpY2UtYWNjb3VudC1icm9rZXIgaW50ZXJuYWxAcGxh
Y2Vob2xkZXIub3JnIn0.u1VBh2AYIVvyt8SxyDnDZHEZL8Xqx4ArCkt58cr4kTXBXlAXlJhCs40i
V1_QxcLAemBwFM3KqUNDIPNRqKMqzM7aJut37uZ8jjs_2A8xtBKwfTQbh8JgH9EaiQFEa2NzMg2v
5C9xJSoMG9y_B9mk9MJ_OrwfustXg3QTnh2nWIoJEEwiOvBiLHSVTgXKwkD61tQw3M7iHibO8Z0w
yFYjMIMBWIc1AcgtnNxTcoNo7U00y2PvguRzq_wgZJXQARYaFNwlcSPti3rOP9dvldeUBehk1pW9
8DHrZldAk2xkOWWKL1hXppssp6T4NKbc1lo4QdOEOTi3XwH7ULWh5hiA
";
<http://jrdfb.iais.fraunhofer.de/vocab/classMapping>
        [<http://industrialdataspace.org/2016/10/ids/core#AuthToken>
"de.fraunhofer.iais.eis.AuthTokenImpl";
        <http://industrialdataspace.org/2016/10/ids/core#tokenValue>
"java.lang.String"
        ] .
```

Figure 6 Sample Industrial Data Space data transfer header in RDF

The sample data transfer header has the following structure:

- The first double line contains the message ID "fa2049be-a089-4b8a-b9e7-744456031f7b", which is a generated UUID.
- The second double line contains the reference to the authentication token "2360d56f-6f35-4a83-a568-5d64c18d202e". The token follows later.
- The next double line provides the name of the hash function, applied to the payload. Here the value is "SHA-512".
- The following triple line provides the calculated hash value (i.e. digest) "ko0+TuFXfpHwkVfo6s1CVITGAv7txskPYyHrYMx0ueZVGWzEisMO5BnC9wW9LUNHMq9HXL4ZvuHlBWL51cwcfg==" in Base64 encoding.
- The next two lines contain the URLs of the receiver and of the sender.
- The next entry is the timestamp for the creation of the message.

- The next block identifies the mappings that are used for the different header parameters.
- The header closes with the reference to the authentication token, the token value as a string (here 1502 characters) and the mapping information for the token string.

The size of our sample data transfer header is 3,605 bytes. It has been generated by means of a Java library from the Industrial Data Space Information Model[1].

Note that the sample header does not contain customAttributes.

The following figure shows the decoded authentication token[2].

```
HEADER: ALGORITHM & TOKEN TYPE
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "PSxny-rCMc_LeB6D8ezYSWUey_A10FYpX5jKZEIOQ9M"
}
PAYLOAD: DATA
{
  "jti": "c96b5915-c1fa-4fa0-b97d-f6de5cca26cc",
  "exp": 1519037681,
  "nbf": 0,
  "iat": 1519037381,
  "iss": "http://192.168.211.111:8080/auth/realms/Broker",
  "aud": "Broker_Internal",
  "sub": "d5b7edcc-63d0-4d22-8155-f2b1a174e6f6",
  "typ": "Bearer",
  "azp": "Broker_Internal",
  "auth_time": 0,
  "session_state": "e9ff5b6a-4ff6-4c0d-9b61-7ea14efc8df8",
  "acr": "1",
  "client_session": "435d9ea8-2045-4353-a499-a7e844d90c0b",
  "allowed-origins": [

    ""
  ],
  "realm_access": {
    "roles": [
      "uma_authorization",
      "broker",
      "broker-client"
    ]
  },
  "resource_access": {
    "account": {
      "roles": [
        "manage-account",
        "manage-account-links",
        "view-profile"
      ]
    }
  },
  "clientHost": "172.18.0.1",
  "clientId": "Broker_Internal",
  "name": "",
  "preferred_username": "service-account-broker internal",
  "clientAddress": "172.18.0.1",
  "email": "service-account-broker internal@placeholder.org"
}
```

Figure 7 Decoded sample authentication token

[1] https://github.com/IndustrialDataSpace/InformationModel

[2] https://jwt.io/

The next part of the multipart message contains the payload, e.g. as an octet string.

The third part contains the signature of the data transfer header. The signature is generated by means of a public key.

```
SHA256withRSA:6dbb89f43efbdc2e8c81233fee681fc45b55e6cab231bffffbb329f5c62cc3
20fb0f6c1268a1cd9c05f3df5f5c3741dfaee87d2913a50db2566f2fbb33704b79b67a9f32da
b334994591705f8f406eabb117b4f92e1d9d1bf9aaba31805aeab744739eeebea164399507b4
dc8de7248d81381474c1d417e77feae7b7df6b04a4163ceff0ac42c359d893fc7f1cfcb3f76a
b445075afefebc3b42a57ee23b0c3f02b348029df7d04792489f8c24f495955b863cfbfd432a
131992217fa93ce8e9342d7d0bff0fe45f5e02c9c4bd091a7cd71967d8adde4c7c88393d2d91
80577fcf08257cd84b487f707e07a91749b5bea14c1e74b8ee7ea0170a21fed70eb6e7
```

Figure 8 Sample signature

The composed multipart message is then sent to the receiver.

2.4.2 Receiver

The receiver performs the following steps:

1. Parsing of the multipart components of the message to identify data transfer header, payload and checksum
2. Check authentication token by sending the token to the identification provider for validation
3. Check if the right receiver has received the message
4. Check signature to verify that the message has not been modified (see code example below).
5. Check digest by calculating the checksum for the payload with the function denoted in hashFunction

The following code snippets illustrate the validation of the checksum by the receiver.

```
private static final String ALGORITHM = "SHA256withRSA";
   /*
         * Validated signature for data transfer with given public key.
         *
         * @param dataTransfer the signed object
         * @param signature the signature data which will be checked
         * @param key the public key used for verification
         */
public boolean validateSignature(DataTransfer dataTransfer, SignatureData
signature, PublicKey key) {
        Signature sig;
        try {
            sig = Signature.getInstance(ALGORITHM);
            sig.initVerify(key);
            update(sig, dataTransfer);
            return
sig.verify(StringUtil.hexToBytes(signature.getSignature()));
        } catch (NoSuchAlgorithmException |
InvalidKeyException|SignatureException ex) {
            throw new RuntimeException(ex);
        }
    }
        /**
     * Loads public key from X.509 certificate string
     *
     * @param publicKey certificate data
     * @return corresponding public key, or none if an error occurs
     */
    public static PublicKey stringToPublicKey(String publicKey) {
        InputStream inputStream;
```

```
        CertificateFactory certificateFactory;
        try {
            inputStream = new ByteArrayInputStream(publicKey.getBytes());
            certificateFactory = CertificateFactory.getInstance("X.509");
            // read certificate
X509Certificate certificate = (X509Certificate)
certificateFactory.generateCertificate(inputStream);
            return certificate.getPublicKey();
        } catch (CertificateException e) {
            logger.error("Error during creation of certificate
factory/parsing data.", e);
        }
        // in case of error
        return null;
    }
    /**
     * Prefix of private key, which needs to be removed prior to parsing
     */
    private static final String privateKeyPrefix = "-----BEGIN PRIVATE KEY--
---";
    /**
     * Suffix of private key, which needs to be removed prior to parsing
     */
    private static final String privateKeySuffix = "-----END PRIVATE KEY----
-";
        /**
     * Loads private key from PEM[1] file.
     *
     * @param filename filename of private key (located in resources)
     * @param algorithm the algorithm used for this key
     * @return private key in usable form
     */
    private PrivateKey filenameToPrivateKey(String filename, String
algorithm) {
        Resource resource = new ClassPathResource(filename);
        InputStream inputStream;
        try {
            inputStream = resource.getInputStream();
            String privateKeyData =
org.apache.commons.io.IOUtils.toString(inputStream);
            // remove pre-/suffix & decode
            privateKeyData = privateKeyData.replace(privateKeyPrefix, "")
                    .replace(privateKeySuffix, "")
                    .replace("\r\n", "")
                    .replace("\r", "")
                    .replace("\n", "");
            byte[] decodedPrivateKey =
Base64.getDecoder().decode(privateKeyData);
        PKCS8EncodedKeySpec keySpec = new
        PKCS8EncodedKeySpec(decodedPrivateKey);
            KeyFactory keyFactory = KeyFactory.getInstance(algorithm);
            // obtain & return private key
            return keyFactory.generatePrivate(keySpec);
        } catch (IOException e) {
            logger.error("Error loading private key file.", e);
        } catch (NoSuchAlgorithmException e) {
            logger.error("Unknown algorithm: " + algorithm + ".", e);
        } catch (InvalidKeySpecException e) {
            logger.error("Could not obtain private key for given keySpec",
e);
        }
        // in case of error
        return null;
    }
```

Figure 9 Code sample for checksum validation

If the digest is the same, the payload can be processed.

[1] PEM: Privacy Enhanced Mail

3
A Use Case for an Industrial Data Space Sensor Connector

3.1 Business Models and the "Shareconomy"

Sharing industrial resources empowered by digital means is a multi-billion euro business that boosts flexibility, cost efficiency and division of work. Companies do not necessarily own assets or processes, but are able to flexibly provide resources or create value-creating networks ad hoc. For example in the field of logistics, already today fourth-party logistics providers (4PL) do not possess trucks, but serve as a broker between inquirer (e.g. manufacturer) and providers (hauler).

The economy of sharing ("shareconomy" or "platform economy") is expected to grow rapidly [10]. This megatrend is commonly known from the consumer business where eBay, AirBnB, Uber and others help customers sharing the respective resources and services. In the IT sector, computing performance, storing capacities and algorithms are provided as cloud services like Amazon Web Services or Microsoft Azure.

The mechanism of shareconomy can be also observed in the industrial sector: material and technology companies (e.g. thyssenkrupp) launch printing farms for additive manufacturing that can be flexibly integrated into to value creation process, for example to mitigate demand peaks. Manufacturing and material handling equipment can be rent for the duration of demand. Manufacturers of machines and equipment typically offer rental services, e.g. manufacturers like Crown, Jungheinrich, Linde or Still offer thousands of forklifts for rent.

Digitization acts as a catalyst in two ways: first, digitally supported business processes with suitable interfaces on the physical and on the IT level are faster and easier to exchange and to be flexibly integrated with alternative suppliers and customers. This increases the demand of shared resources. Second, the availability of data within and across companies provides the foundation for new business models. Manufacturers of turbojets for aircrafts do not necessarily sell the jet, but its operating hours. Automotive manufacturers prepare to become a mobility service provider. Any machine or tool provided to a customer can be offered as a service with defined uptime.

In a well-designed platform-driven business model, both the provider and the customer can benefit from a higher resource capacity usage. The resource provider receives compensation for management and service provisioning, the customer gains flexibility and pays only per use to, consequently, save money.

Obviously, the availability of data is a crucial prerequisite for these business models, because providers have to anticipate demand, predict the need for maintenance and offer usage-dependent pricing models. This also requires data sharing between at least the manufacturer and the operator, however other instances like maintenance providers can be involved. The use case example in the next section describes how these parties will cooperate and why data sovereignty is required.

3.2 Use Case: Forklift Rental in the Digitized Shareconomy

Forklifts or any other equipment like loaders, scaffolds or even drilling machines are typically leased to operators. Fees mostly depend on operating hours and the

occupation of the resource. Linde, for example, charges a daily fee per 8 hour shift for a maximum of 100 operating hours per month.[1] This conventional pricing scheme is quite rigid and, thus, limits the flexibility of the operator and restrains potential turnover of the provider.

The application of sensors to the forklift provide further parameters that can be involved in a pricing and maintenance model adapting to the actual need for resource maintenance or renewal. In our use case, relevant parameters influencing the stress on material, maintenance cycles and the lifetime of the forklift are the average and maximum temperature of the environment and the engine as well as strong acceleration and breaking. As forklifts are operating in a closed system (i.e. logistics site), the sensor could be easily integrated into an existing or a specifically setup Wi-Fi network. In the simple use case presented in this paper, the available data needs to be shared with the following stakeholders:

1. Forklift lessee (operator, data creator)
2. Forklift lessor (manufacturer)
3. Maintenance prediction service provider
4. Maintenance service provider

This ecosystems could be easily enhanced, e.g. by a financing service provider, a fuel provider, a forklift pool operator different from the manufacturer and any digital "smart" service.

The required information flow between the three actors is depicted in *Figure 10*.

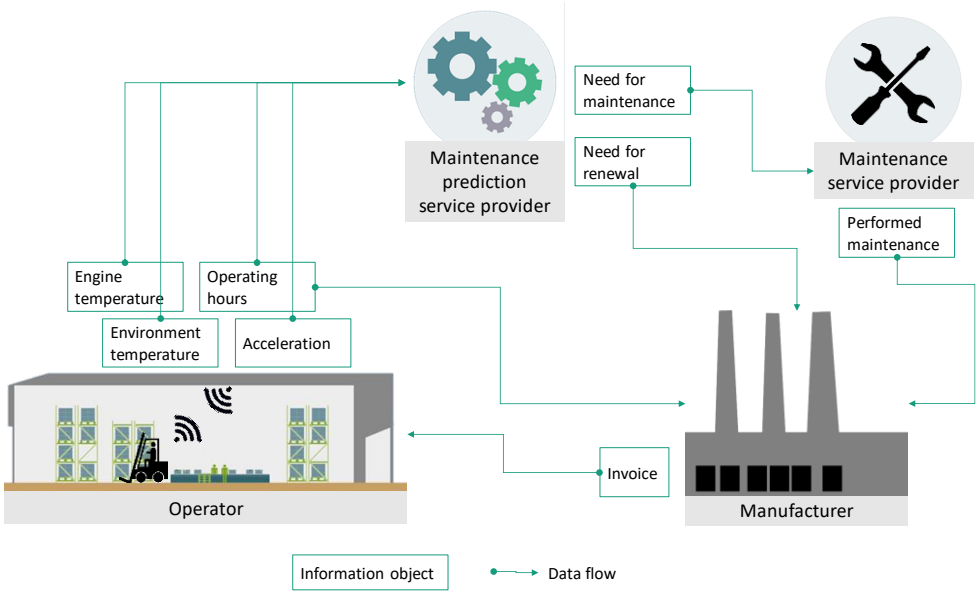

Figure 10 Forklift leasing scenario based on the Industrial Data Space in the digitized shareconomy

Even in this basic ecosystem, data generated by the operator is shared between many parties. This business model finds the operator's acceptance if his data sovereignty is ensured. Therefore, first, access restrictions and control according to the exemplary scheme in *Figure 11* are needed.

[1] http://www.linde-mietstapler.de/de/de_mietstapler/mietsortiment/mietsortiment_1.html

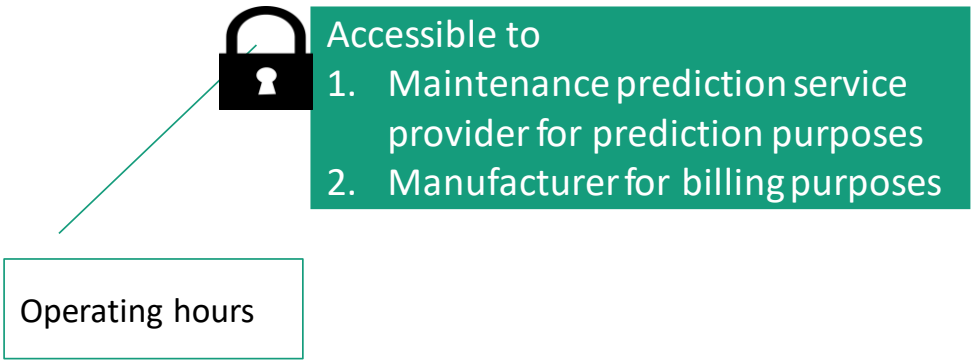

Figure 11 Exemplary data access restriction rules to guarantee data sovereignty

Second, the data owner needs a certain guarantee that these "terms of use" are respected on the side of the data receiver.

3.3 Requirements from the Use Case Perspective

In summary, from the application perspective, the fulfillment of the following requirements enable the data-driven shareconomy model outlined above:

1. Economic advantage (higher flexibility, higher reliability, less costs, …) for each ecosystem participant
2. Near-real-time perception of relevant process parameters
3. Availability in a digital ecosystem
4. Mechanisms of data control by terms of data use and their remote enforcement

The first requirement is in particular a matter of contract design, whereas the other requirements can be fulfilled by an appropriate lightweight sensor connector based on the Industrial Data Space and presented in the following section.

4
An Industrial Data Space Connector for IoT Devices

We start with a look at different options for integrating IoT with the Industrial Data Space. Based on the results and Industrial Data Space requirements on connectors and their security we describe an architecture for an Industrial Data Space Sensor Connector. Then we describe the implementation and first experiences.

4.1
IoT Integration Patterns for the Industrial Data Space

IoT has been defined as "a global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies"[4]. The ISO / IEC IoT Reference Model (IoT RM) [5] shows two different ways of transferring sensor data from a physical entity to a backend application: either the device communicates directly with the application or it uses an IoT gateway as intermediary.

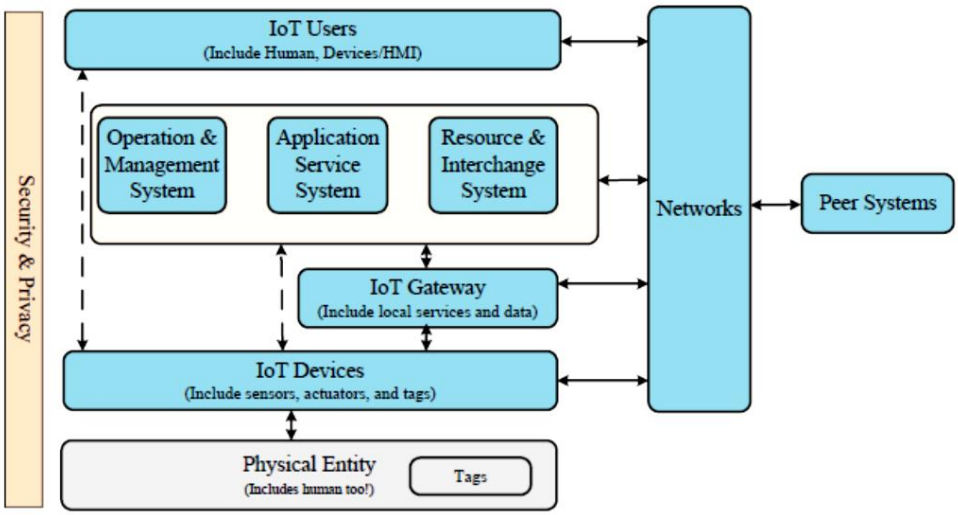

Figure 12 ISO / IEC IoT Reference Model (IoT RM)

Transferring this view to the Industrial Data Space, we see three different integration patterns.

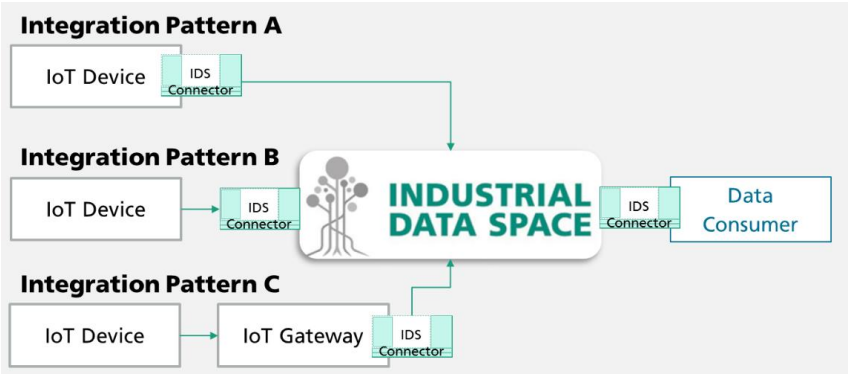

Figure 13 Industrial Data Space IoT systems integration patterns

Pattern A shows an IoT device with an Industrial Data Space Connector on board communicating with a peer connector at the data consumer site. Pattern B refers to an IoT device that sends its data to a local Industrial Data Space Connector, which communicates with the Industrial Data Space Connector at the receiving site. In pattern C, the IoT device communicates with an IoT gateway that has an Industrial Data Space Connector.

Given that an IoT gateway serves multiple IoT devices, we can assume that the connector underlying platforms in patterns B and C are more powerful than that of pattern A. For patterns B and C, we can assume that the connectors run on a fixed/residential platform implying fewer restrictions on hardware and software capabilities.

Pattern A, in contrast, mainly refers to mobile applications, i.e. the IoT device is attached to a moving "thing". In that case, generally more or less strict constraints occur regarding e.g. form factor and weight of the IoT device. Consequently, small scale embedded systems are used as IoT devices that have very limited hardware resources (i.e. low power CPU, small memory) with minimal power consumption. To run an Industrial Data Space Connector on such a platform, a connector with minimal footprint is required.

4.2 A Lightweight Industrial Data Space Sensor Connector Architecture

From the use case scenario described above we can derive that the Industrial Data Space Connector serves as a unidirectional sender of sensor data to an identified, predetermined receiver in the backend. The IoT device has a fixed set of sensors attached to it. The device is exclusively used for transferring sensor data from the physical entity it is attached to into the backend for further processing. It uses built-in WLAN for network access.

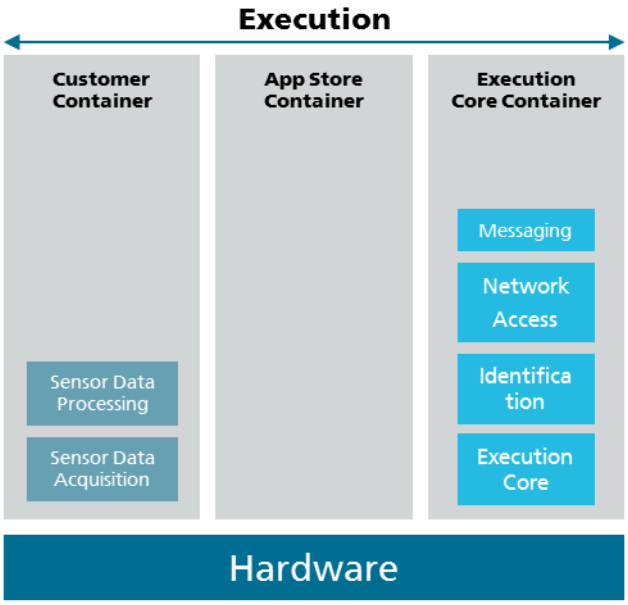

Figure 14 Lightweight Industrial Data Space Sensor Connector Architecture

As an operating system is not mandatory for an Industrial Data Space Connector (see [3] p.8]: "… generally an operating system is necessary."), the lightweight sensor connector is implemented on bare hardware.

The separation of functionalities into different containers reflects only a logical view, as container management is optional (see figure 3).

In accordance with the connector reference architecture, we use data services in the custom container part for sensor data acquisition and pre-processing. The resulting sensor data are inserted into a predefined, Industrial Data Space compliant MQTT message type. An MQTT client, as messaging module in the execution core container, pushes these sensor messages to the predetermined receiving connector.

For self-declaration, i.e. identification, the connector uses a webserver that publishes the connector metadata as RDF file.

4.3 Lightweight Industrial Data Space Sensor Connector Implementation

The prototype sensor connector runs on an ESP8266[1] hardware platform. It is a low-cost 32-bit microcontroller (processor running at 80 MHz) with a Wi-Fi microchip (IEEE 802.11 b/g/n) with full TCP/IP stack, 512 kB flash memory and 16 GPIO pins. Network protocols IPv4, TCP/UDP/HTTP/FTP are supported.

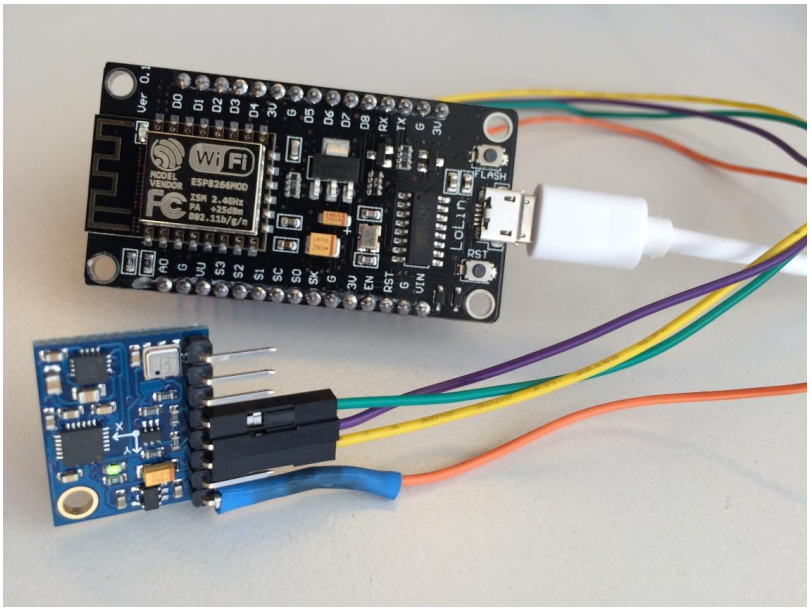

Figure 15 Sensor connector on ESP8266 with GY-87 10DOF sensor module

To enable the sensing of object movements (e.g. forklift in a warehouse) as described in the use case above we connect a GY-87 10DOF sensor module[2] to the ESP8266 that contains a three-axis gyroscope, a tri-axial accelerometer, a three-axis magnetic field sensor, a temperature sensor and a pressure sensor.

[1] See https://www.espressif.com/en/products/hardware/esp8266ex/overview
[2] GY-87 10DOF MPU6050 BMP180 HMC5883L

4.3.1 Connector modules

In the following, we briefly describe the modules of the sensor connector according to the reference architecture depicted in figure 14.

Sensor data acquisition

The device acquires sensor data in configurable intervals, e.g. every 100 milliseconds.

Sensor data pre-processing

In the pre-processing phase, sensor data are aggregated. The mean value and standard deviation for temperature data of one second are determined to discard outliers. The revised value is then transmitted to the backend.

Secure message transfer via WLAN and TCP/IP (TLS)

Network access for the sensor connector is established via the ESP8266 built-in WLAN chip and TCP/IP (IPv4). The credentials for WLAN access can be configured. For security reasons a secure Wi-Fi driver[1] is deployed that applies TLS 1.2[2] for message transfer.

MQTT client

The sensor connector uses the MQTT protocol version 3.1.1 [6] to establish a connection with the receiving connector. We use the pubsub MQTT client[3] on the sensor connector. The client establishes a clean session, i.e. the clean session flag in the CONNECT message is set to TRUE and the Quality of Service flag is set to zero. That means that the MQTT broker on the receiving connector will not store anything for the sending client. Accordingly, in PUBLISH messages the Quality of Service Level (QoS) is set to zero and the RetainFlag is set to FALSE (fire-and-forget mode). The topic "Industrial Data Space/Sensor_Connector_4711/Company_A/forklift/movement" denotes the sensor data stream that the sensor connector pushes to the receiving connector.

MQTT messaging

The connector pushes sensor data in SenML format [7]. The MQTT PUBLISH messages embed these sensor data as payload.

```
{"n":"isst:esp:temp:esp01",
"u":"Cel",
"id":"1000000",
"v":"23.5"}
```

Figure 16 Sample SenML payload for temperature sensor data

"n" denotes the identification of the sensor and device; "u" denotes the unit of measured data, which is "Celsius" in our case; "id" gives the identification of the

[1] https://github.com/tzapu/WiFiManager
[2] Transport Layer Security, a cryptographic protocol between MQTT and TCP/IP to secure all communications between the sensor connector and the receiving connector
[3] https://github.com/knolleary/pubsubclient

measurement, i.e. a sequential number starting from a base value; "v" denotes the measured value.

Webserver

The webserver[1] installed on the ESP8266 is used for self-declaration and for configuration management of the device and the connector.

Self-declaration

Industrial Data Space requires the self-declaration of a connector. Either a connector pushes its self-declaration to a broker when registering there or it provides an interface where the self-declaration can be retrieved.

The lightweight connector uses the installed webserver for providing its self-declaration upon request. The self-declaration is provided as RDF file in Turtle notation in accordance with the Industrial Data Space Information Model.

4.3.2 Industrial Data Space Data Transfer Header for IoT

In addition to the sensor data, every Industrial Data Space compliant message transfer requires Industrial Data Space metadata as part of a message. An Industrial Data Space message consists of three parts: a data transfer header, the payload and a checksum.

For real-time IoT data streams from low-cost IoT devices, such a data transfer header as depicted in figure 6 above is a heavy burden. Protocols for machine-to-machine (M2M) communication like MQTT are designed for high-speed transfer of telemetry data, using messages of short size. Typical message sizes are in the range of several hundred bytes to 1 kB. IBM Watson IoT platform accepts MQTT messages with a maximum payload size of 128 kB[2].

From the perspective of the M2M protocol (e.g. MQTT), the three parts of an Industrial Data Space message represent the payload. Hence, the Industrial Data Space data transfer header should be as small as possible to leave enough room for the real payload, i.e. sensor data.

Under the assumption that the hash function that processes the payload does not change for a sender, the hash function to be applied on the receiving site can be retrieved from the self-declaration of the sender. Hence, we can drop it. CustomAttributes are considered as optional as a concrete description of usage is still missing. From the perspective of the sending site, the receiver is fix so there is no need to transfer the link to the receiver's self-declaration. The message id might be needed to identify lost messages or to guarantee the order of messages if it is not part of the protocol. An MQTT message, for instance, contains a packetID as part of its message header anyway. That would serve the same purpose. Therefore, we consider it as optional. The hash value of the payload (header attribute *digest*) assures that the payload is not manipulated. From the security perspective, we keep it as well as the authentication token. Given that we have a high frequency stream of IoT data and microcontrollers often do not have a clock to create a timestamp, we consider the transferCreatedAt attribute as optional. As the sender is already identified through the MQTT message header, we drop this parameter.

[1] https://github.com/esp8266/Arduino

[2] https://console.bluemix.net/docs/services/IoT/reference/mqtt/index.html#ref-mqtt, retrieved on 02-13-2018

Table 2 shows the minimal data transfer header for Industrial Data Space IoT messages from our perspective.

Attribute	Type	Description
authToken	AuthToken	Authentication token
digest	byte [ ]	Hash value of the payload

Table 2 Minimal data transfer header of an Industrial Data Space IoT message

If an application needs more attributes in the data transfer header, e.g. some of those we consider as optional, the description of the header structure should be a part of the self-declaration.

The sample data transfer header depicted in figure 3 consists of 3,605 bytes. That is a heavy overhead for the transfer of a few bytes of sensor data. The size results from a very redundant description of header attributes and their values, each containing the entire path name of the Industrial Data Space Information Model and from an authentication token that contains a lot of metadata as payload (see figure 7). The authentication token in the example consists of 1,500 bytes.

The structure of an authentication can be defined. If we limit the token header to type and algorithm attributes and include in the payload only the issuer, issuing time and expiration time, we end up with an authentication token in the range of 200 bytes[1] as the example below shows.

```
HEADER:ALGORITHM & TOKEN TYPE
{
  {

  "typ": "JWT",
  "alg": "HS256"
}
PAYLOAD:DATA
{
  "iss": "www.industrialdataspace.org/authtoken/",
  "iat": 1459448119,
  "exp": 1459454519
}
RESULTING TOKEN
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJ3d3cuaW5kdXN0cmlhbGRhdGFzcGF
jZS5vcmcvYXV0aHRva2VuLyIsImlhdCI6MTQ1OTQ0ODExOSwiZXhwIjoxNDU5NDU0NTE5fQ.OCYX
jXXfqT5_vZ3JfQJrqU7rmd70l3GxWbaBbEnShKY
```

Figure 17 Sample authentication token for Industrial Data Space IoT messages

If we apply a notation like SenML to the reduced data transfer header with the minimized authentication token, we can significantly reduce the header size as the example shows.

[1] https://jwt.io/ with secret L3@RNJWT

{
„Bn“: http://industrialdataspace.org/2016/10/ids/core#

„at“:“eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJ3d3cuaW5kdXN0cmlhbGRhdGFzcGFjZS5vcmcvYXV0aHRva2VuLyIsImlhdCI6MTQ1OTQ0ODExOSwiZXhwIjoxNDU5NDU0NTE5fQ.OCYXjXXfqT5_vZ3JfQJrqU7rmd70l3GxWbaBbEnShKY“

„dg“:“69db910dc7a0298839de25072dc8615146ac4ce07d0dc5338bbf81974be06721“
}

Figure 18 Sample Industrial Data Space IoT Data Transfer Header

The attribute *bn* corresponds with the base name in SenML and refers to the URL of the Industrial Data Space Information Model. The attribute *at* denotes the authentication token and the attribute *dg* represents the digest attribute. In our example, we used the SHA-256 algorithm[1] for hashing the payload depicted in figure 17.

The reduced header consists of 324 bytes for a payload of 63 bytes, which is a factor of 5 compared to the original header with a size of 3,605 bytes that causes overhead of a factor of 57. In this calculation, the overhead of the signature in the third message part is still ignored.

4.3.3 Industrial Data Space IoT Message Signature

The signature of Industrial Data Space messages is the checksum of the data transfer header (see section 2.3.3). As the “standard” data transfer header of these messages is fairly long (3,600 bytes in our example, see section 2.4.1), the resulting signature has a considerable length as well. In our example it takes 500 bytes.

The algorithm for the original signature is SHA256withRSA. The algorithm first generates a hash code for the data transfer header, using SHA256. In a second step, the hash code is encrypted with RSA, using the private key of the connector.

If we take our reduced data transfer header for IoT messages as described in the preceding section as a basis and use only SHA256 without RSA encryption[2], the signature is also significantly shorter.

"581011AFA2F8EFC326D74FC5FFE728467D0C2EA2DAEA9230333167213D23E7D6"

Figure 19 Signature of the sample IoT data transfer header

The signature for our sample data transfer header has a size of 64 bytes, which is only a fraction of the “standard” signature.

As a result the entire sample IoT message consists of a 324 bytes data transfer header, 63 bytes of payload and 64 bytes of signature, totalling to 451 bytes.

4.3.4 Summary

In this section we have described an approach for an Industrial Data Space message format that applies the basic principles and concepts of the Industrial Data Space and is still applicable for the transfer of IoT data by means of standard IoT communication protocols like MQTT. We preserve the general message structure, consisting of a data transfer header, payload and signature, as well as the key security features, namely an

[1] http://www.xorbin.com/tools/sha256-hash-calculator
[2] We could not find a lib for the ESP8266 that provides SHA256withRSA.

authentication token and the checksum of the payload as part of the data transfer header and the checksum of the data transfer header. To keep the message as short as possible we have reduced the amount of metadata.

For a sample IoT message that transfers temperature data from a sensor device, we end up with a message size of 442 bytes (see figure 20). The payload takes 43 bytes, 399 bytes are required for metadata and/or security data, respectively. This means an overhead factor of 9.3.

In contrast to our approach a "standard message would take 3,605 bytes for the data transfer header, 63 bytes of payload and 500 bytes for the signature, summing up to 4,168 bytes. That is an overhead of a factor 65.2.

5
First Experiences

We have implemented the Industrial Data Space Sensor Connector as described above. We have chosen a low-cost hardware platform with a small form factor that is applicable for mobile applications.

The connector sends acceleration and gyro data in 200 millisecond intervals to an MQTT broker, running on a Raspberry Pi 3. For simplicity reasons we use a fix authentication token instead of renewing it periodically. The Raspberry Pi displays the decoded messages in a console window. Further processing or storing of data has been omitted as our focus is on the connector and not on the receiving side.


```
header

Bn          "http://industrialdataspace.org/2016/10/ids/core"

at
            "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJ3d3cuaW5kdXN0cmlhbGRhdGFzcGFjZS5
vcmcvYXV0aHRva2VuLyIsImlhdCI6MTUyMDI1Nzk0NywiZXhwIjoxNTIwMjU3OTc3fQ.zp4_NvCZY4brdB9V
Mw6c2-EGuefe62vXklUUnSbHpZA"

dg          "E1E34C85EDA76FC1402F316DA68FF989980246D1F9E88FF9C84753BE701AC9EB"
payload
bn          "hmi:esp:0:tmp"
bu          "Cel"
t           0
v           25
checksum    "581011AFA2F8EFC326D74FC5FFE728467D0C2EA2DAEA9230333167213D23E7D6"
```


Figure 20 Sample MQTT message body as JSON object

The sample message body as JSON object to transfer temperature sensor data consists of 442 bytes, including blank characters. Sensor data require 43 bytes.

Given an average throughput of 22 Mbps for a WLAN connection (IEEE 802.11g)[1], message transfer without TLS would take in the range of 0.2 milliseconds. In contrast, the transfer using the "standard" message format would take 1.4 milliseconds without TLS.

The performance penalties caused by using TLS 1.2 and by running hash functions on the payload and on the data transfer header limit sending intervals. Our experiments show that we can transfer temperature data at 50 millisecond intervals. If we increase the size of sensor data, e.g. by transferring acceleration data with values for three axis we have to extend the interval to 170 milliseconds or more to transfer data without losses.

[1] https://en.wikipedia.org/wiki/IEEE_802.11#802.11g

6 Conclusion

From the Industrial Data Space Reference Architecture model and related documentation, we have derived an architecture for a lightweight sensor connector and a lightweight message format that comply with Industrial Data Space requirements on architecture and security. The lightweight connector aims at integrating IOT devices as data sources into Industrial Data Space ecosystems. The connector has been implemented on a low-cost device and transfers sensor data via Industrial Data Space compliant MQTT messages over WLAN at a speed of 5 to 20 messages per second, depending on sensor data size.

For MQTT the reduced Industrial Data Space message format is acceptable. The message size in our experimental setting is in the range of 400 to 500 bytes and can easily be handled by MQTT clients and brokers. For other machine-2-machine protocols like e.g. CoAP [8] the situation is different. The recommendation for CoAP is to restrict message size to fit within a single IP packet to avoid IP fragmentation; that leaves 1 kB for payload. This would still work for our application scenario. When using CoAP on 6LoWPAN networks, however, the limit shrinks to 60-80 bytes[1]. In that case, even our reduced message format has too much overhead from security means like authentication token and signature. Here, only a data transfer header without authentication token and some payload would fit into these limits. This is a trade-off between security and size.

Security of IoT data transfers is an issue that needs to be discussed in more detail. In a typical IoT application scenario, like e.g. monitoring of movements (see use case above) or predictive maintenance of machines, multiple devices stream their data to a single backend. For each sending IoT device, it is a point-to-point connection with a known receiver. From the perspective of the receiving site, it is multiple connections with maybe a changing number of senders. A sender wants to be sure that the message is tamper-proof sent to the correct receiver. A receiver wants to be sure that the messages he receives originate from valid senders and that the message has not been corrupted. The Industrial Data Space hence establishes security on multiple levels. Data transfer is protected through message encryption by means of TLS. Authenticity of messages is guaranteed through authentication tokens with short expiration times, provided by a trusted instance within the Industrial Data Space ecosystem, and by the signing of messages with a private key of the sender. The receiver then is able to decode the signature by means of a public key. To protect against the tampering of payload data Industrial Data Space messages contain a hash code of the payload in their data transfer header and the header is an input parameter for the generation of the signature.

These security provisions are reasonable and valuable for the transfer of documents or sensitive data in general. If they are meaningful for IoT applications depends on the application scenario.

Let us consider the sensor data stream in our sample use case. For the sending side, the receiver address is typically a configuration parameter that is set at system setup time. Thus, it is not critical at message transfer time. Message transfer is secured with TLS

[1] https://stackoverflow.com/questions/32007807/what-is-the-size-of-coap-packet, retrieved on 02-13-2018

and certificates in a first step. Under the assumption that the Industrial Data Space ecosystem establishes a trusted site for the issuing of certificates the risk of man-in-the-middle-attacks is minimized. Hashing of the payload and a signature for the Industrial Data Space data transfer header are further means to enhance security. However, that implies that an IoT device is powerful enough to run the hash algorithm and the signature fast enough to send messages in an interval required by the application. Besides that, the underlying communication infrastructure that connects sender and receiver must be able to cope with the resulting message sizes. As the CoAP example shows, this is not necessarily the case.

Our prototype development shows that it is possible to provide a lightweight Industrial Data Space connector for IoT applications that fulfils security requirements to a considerable extent. It is able to cover a broad range of IoT applications at low-costs.

Acknowledgements

We would like to thank Dr. Ralf Nagel for his support regarding the Industrial Data Space message format and Fabian Bruckner for patiently providing us with information on implementation details and for delivering the code snippets (figures 6 to 9).

7
References

[1] Gartner (2017). Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016. Press release, Egham, U.K., February 7, 2017. Retrieved January 31, 2018, from https://www.gartner.com/newsroom/id/3598917.

[2] Otto, B. et al. (2017). Reference Architecture Model for the Industrial Data Space. Fraunhofer-Gesellschaft zur Förderung der angewandten Forschung e.V. Munich, Germany. Retrieved September 06, 2017, from http://www.industrialdataspace.org/publications/industrial-data-space-reference-architecture-model-2017

[3] Nagel, R. (2017). Industrial Data Space - Connector in a nutshell. Fraunhofer ISST, November 2017.

[4] ITU-T (2012). Overview of the Internet of things. SERIES Y: GLOBAL INFORMATION INFRASTRUCTURE, INTERNET PROTOCOL ASPECTS AND NEXT-GENERATION NETWORKS, Next Generation Networks – Frameworks and functional architecture models, Recommendation ITU-T Y.2060, 06/2012. Retrieved January 30, 2018 from https://www.itu.int/ITU-T/recommendations/rec.aspx?rec=y.2060.

[5] ISO/IEC (2016). Information technology – Internet of Things Reference Architecture (IoT RA). ISO/IEC CD 30141:20160910(E), ISO/IEC JTC 1/WG 10. Retrieved July 3, 2017, as 10N0536_CD_text_of_ISO_IEC_30141 from https://www.w3.org.

[6] OASIS (2015). MQTT Version 3.1.1 Plus Errata 01. OASIS, 10 December 2015. Retrieved January 31, 2018, from http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.

[7] Jennings, C. et al. (2017). Media Types for Sensor Measurement Lists (SenML). IETF Network Working Group, Internet Draft 12, December 14, 2017. Retrieved January 31, 2018, from https://tools.ietf.org/html/draft-ietf-core-senml-12.

[8] Shelby, Z., Hartke, K. and Bormann, C.: The Constrained Application Protocol (CoAP). Internet Engineering Task Force (IETF), RFC 7252, June 2014. Retrieved February 13, 2018, from https://tools.ietf.org/html/rfc7252#page-25.

[9] Otto, B.: Data sovereignty. The Industrial Data Space. Published by Fraunhofer-Gesellschaft eV., Munich 2016. https://www.isst.fraunhofer.de/content/dam/isst/en/documents/Publications/Digitization-in-Service-Industries/fhg_207-DATA-SOVEREIGNTY-www.pdf

[10] Kenney, M., Zysman, J.: The Rise of the Platform Economy. Issues in Science and Technology; Washington Vol. 32, Iss. 3, (Spring 2016): 61-69. http://issues.org/32-3/the-rise-of-the-platform-economy/