DIGITAL MANUFACTURING PLATFORMS FOR CONNECTED SMART FACTORIES

D3.12 Permissioned Blockchain for ZDM

Deliverable Id :	**QU4LITY-D3.12**
Deliverable Name :	**Permissioned Blockchain for ZDM**
Status :	**Final**
Dissemination Level :	**PU**
Due date of deliverable :	**31/03/2021**
Actual submission date :	**08/06/2021**
Work Package :	**WP3**
Organization name of lead contractor for this deliverable :	**Engineering Ingegneria Informatica S.p.A. (ENG)**
Author(s):	**Mauro Isaja (ENG)**
Reviewer(s):	**Marcel van der Kraan (TNO)**
Partner(s) contributing :	**ENG, ATOS, FHG, AIT**

Abstract: This deliverable is devoted to the presentation of blockchain-based digital enablers, also known as Distributed Applications (DApp in brief), that implement the secure data sharing and state synchronization across multiple systems in the manufacturing value chain.

Co-funded by the Horizon 2020 Programme of the European Union

Contents

List of figures

List of Abbreviations

Abbreviation	Explanation
API	Application Programming Interface
B2B	Business-to-business
CA	Certificate Authority
ERP	Enterprise Resource Planning
DApp	Decentralized Application
DID	Decentralized Identifier
DLT	Distributed Ledger Technology
HLF	Hyperledger Fabric
HTTP	HyperText Transfer Protocol
IIoT	Industrial Internet-of-Things
MSP	Membership Service Provider
OEM	Original Equipment Manufacturer
PL	Private Ledger
QADM	Quality Assessment Data Model
QAR	Quality Assessment Report
QCH	Quality Clearing House
RA	Reference architecture
REST	Representational State Transfer
RWE	Real World Entity
SID	Secure Identity Directory
SMB	Secure Messaging Board
SSI	Self-sovereign identity
SUM	Shipping Unit Manifest
VCL	Value Chain Ledger
WP	Work package
ZDM	Zero Defect Manufacturing

HISTORY

Version	Date	Modification reason	Modified by
0.1	26/03/2021	First draft: structure	Mauro Isaja
0.2	02/04/2021	Sections 2 & 3: recap of D3.11	Mauro Isaja
0.3	22/04/2021	Section 4: DApp documentation	Mauro Isaja
0.9	06/05/2021	First version ready for review	Mauro Isaja
1.0	19/05/2021	Final version after review	Mauro Isaja

Executive Summary

Deliverable D3.12 "Permissioned Blockchain for ZDM – final version" reports on the work performed and the final results of Task 3.6 "Blockchains for Secure Decentralized State Management". The objective of Task 3.6 is the development of a Distributed Ledger Technology (DLT) infrastructure that will enable secure state sharing and synchronization of distributed industrial processes related to AQ/ZDM. Most importantly, such infrastructure will support the traceability of data and ensure its integrity, thus significantly improving trust in decentralized processes. This double role of the infrastructure will be possible thanks to the capability of the underlying blockchain platform to keep confidential data secure within separate environments, enforcing specific access policies.

This document is a major update over the previously released D3.11 report. While the initial version was focused on more general topics like the inner workings of Blockchains and Smart Contracts, the selection of Hyperledger Fabric as the baseline DLT platform for QU4LITY and the project's approach to development and deployment of the infrastructure, the main part of this current version is a detailed description of the actual Distributed Applications (DApp*s*) released. These are the following:

- **Quality Clearing House (QCH)**. QCH enables a decentralized workflow that targets a typical Quality Management / ZDM process in a non-hierarchical supply chain scenario.
- **Secure Message Board (SMB)**. SMB enables the publishing of sensitive content (e.g., software, firmware, algorithms, parameters, data, documents) to subscriber channels, with the added value of strong guarantees on provenance, integrity and confidentiality.
- **Secure Identity Directory (SID)**. SID enables the decentralized management of digital identities and authentication credentials for both human and machine actors. SID is also used by both QCH and SMB as their identity management infrastructure.

All the above mentioned DApp*s* are now available on QU4LITY's Value Chain Ledger (i.e., the Hyperledger Fabric permissioned Blockchain network specifically created for QU4LITY and hosted by ENG on a Cloud facility): any user application that has been properly integrated and configured can access their functionality by connecting through the public Internet. Moreover, the source code of all DApp*s* has been released under the terms of the Apache v2.0 open-source license.

1 Introduction

Work package 3 (WP3) is about developing and integrating a range of digital enablers, based on background technological developments of the partners and properly customized to the needs of ZDM, to support the QU4LITY Autonomous Quality paradigm as described in deliverable D2.4. Furthermore, these digital enablers will be integrated as part of ZDM solutions so as to emphasizes interoperability and flexibility and will be further exploited (even outside the project boundaries) through the QU4LITY marketplace (as part of the WP8 activities). Last but not least, the term "digital enablers" implies that each digital component will be reusable and accessible via an Open API, which will facilitate their use in ZDM processes and applications.

The challenge faced by QUALITY is the requirement of interoperability among ZDM digital enablers which may deeply differ from each other. QU4LITY-based systems should, in fact, rely on a layer of abstraction, which, to the extent possible, obscures the system from the underlying implementation.

In this context, task T3.6 has developed a permissioned Distributed Ledger Technology (DLT) infrastructure for quality management processes and related supply chain interactions. Such infrastructure enables secure state sharing and synchronization of distributed industrial processes, and autonomous code execution by the means of smart contracts. This opens a new world of possibilities in terms of agreement management across manufacturers, customers and other stakeholders. Furthermore, DLT can be exploited to address today's challenge in complex and often internationally spanning supply chains; for example, granular evaluation of provenance of physical goods, smart diagnostics and self-service application for machines and cost avoidance impact on supply chain transactions. Many other challenges can be found and solved with distributed ledger technologies, thus, enriching end-users' experience inside QU4LITY platform, contrary to monolithic non-distributed systems.

This D3.12 deliverable, representing the final outcome of task T3.6, is a major update over the previously-released D3.11 and also includes the documentation of the three digital enablers developed as Distributed Applications (DApps) on top of the above-mentioned DLT infrastructure.

The methodology followed for the development and the prototyping of the three DApps was a three-step process. The first two were completed during the first part of the task schedule, which culminated with the release of the D3.11 report (M12).

- **Phase 1 – Requirement analysis and preliminary design**. The requirements for the use of DLT in the QU4LITY context have been identified, with the active involvement of prospect users. A first hypothesis of the to-be-developed DApps was presented in deliverable D3.11.
- **Phase 2 – Platform selection and infrastructure deployment**. The state-of-the-art of DLT was investigated, resulting into the selection of an open-

source "permissioned" Blockchain platform – i.e., Hyperledger Fabric (HLF) – as the baseline for the development of DApps. Consequently, a dedicated HLF network was deployed and configured as the QU4LITY Value Chain Ledger (VCL).

- **Phase 3 – DApp design and implementation**: Finally, the preliminary designs from Phase 1 were reviewed in light of the feedback received within the project, resulting into the specification of the three DApps to be implemented by the task. A prototype of each DApp was developed and deployed on the VCL.

This document is focused on the third phase, and its structure is as follows:

- **Section #1 Introduction** – This present section.
- **Section #2 DLT for QU4LITY** – A recap of the general approach to DLT and smart contracts, in the context of the QU4LITY Reference Architecture. This section is an update to what already discussed in deliverable D3.11.
- **Section #3 QU4LITY's DLT infrastructure** – A recap of the preliminary work done in the scope of task T3.6 during the first part of the project – basically, the selection of the baseline blockchain software and the set up of the QU4LITY Value chain Ledger.
- **Section #4 QU4LITY's DLT digital enablers** – This is the core of the present document, where the main results of the second part of the T3.6 schedule are reported. It contains a thorough description of the three digital enablers that were designed, prototyped and deployed as part of the QU4LITY's DLT infrastructure.
- **Section #5 Conclusion** – Final considerations.

2 DLT for QU4LITY

In the previous version of this deliverable, we have used the "Context and Requirements" section for positioning the Value Chain Ledger (VCL) and the Private Ledger (PL) components in the general context of ZDM and, more specifically, of the QU4LITY project. To facilitate understanding of this final version, here we briefly recap our reasoning.

2.1 Context

Figure 1 below shows the VCL and PL components framed inside the Digital Infrastructures layer of QU4LITY's Reference Architecture (RA). Their role with respect to the other components is to ensure data integrity and non-repudiation. Their novelty is that they enable such functionality in *distributed* environments, where documents must travel across organizations who do not share a common Enterprise Resource Planning (ERP) system.

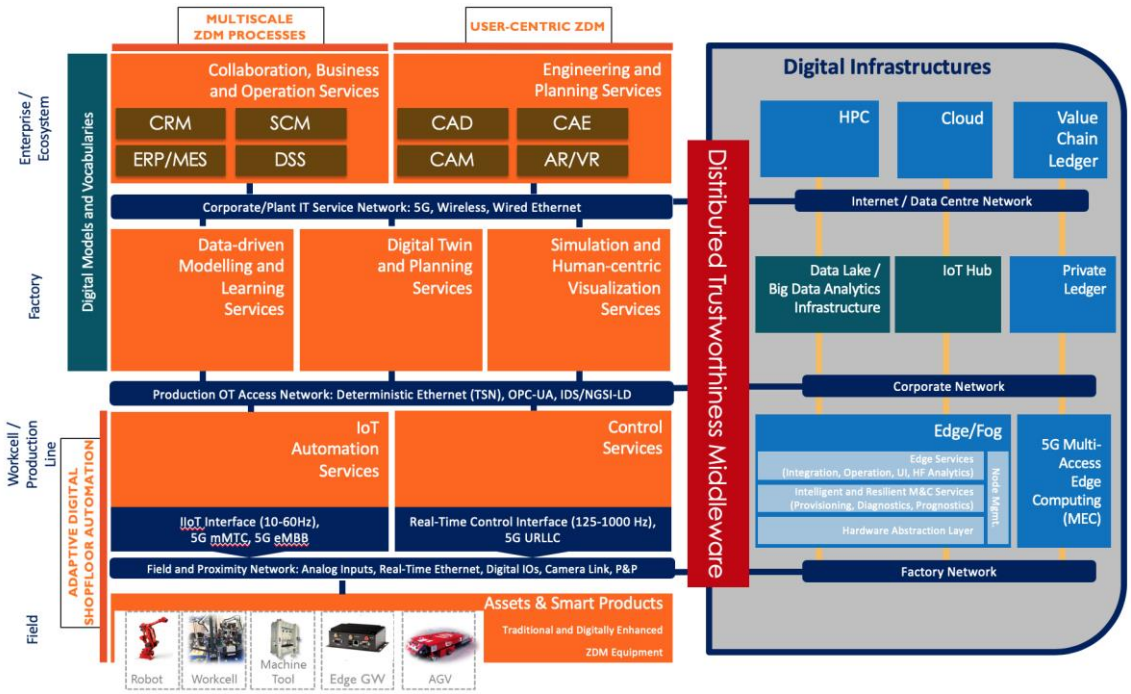

Figure 1 - QU4LITY's Reference Architecture

In DLT, every participant – often referred to as a "node" of the system – maintains a copy of all transactions. This provides an audit trail of every transaction and/or event that has occurred, so that any participant can detect if individual records or the trail itself have been altered by someone else – on purpose or by mistake. Overall, the DLT approach to recording transactions and events offers some business benefits over conventional techniques:

- A transparent, validated single version of the truth, which is ensured by the anti-tampering properties and the distributed consensus mechanisms of the blockchain network.

- Real-time visibility on the entire network, as the blockchain records all transactions and breaks the information silos across the various trading parties.

2.2 Potential use cases

The following paragraphs outline some manufacturing use cases that, according to research literature [Andrews17], [Wang17], can be best addressed through DLT.

2.2.1 Supply chain tracing

DLT can be used to offer highly secure and immutable access to supply chain data [Kim18]. For raw materials and product parts, they are able to provide a digital platform enabling their physical traceability. In combination with IoT-enabled sensors, they can monitor the journey of a good from raw material to finished product. First, the sensors capture finely granular real-time data about environment characteristics as well as location and timestamps throughout the supply chain. Then the DLT platforms manage the chain of custody for said goods, enabling ownership to be transferred and traded on a network using smart contracts. What is more, governance and validation of the logs for those activities is equally distributed between the peer nodes of the system, making their tampering an expensive therefore uneconomical effort for malevolent actors.

2.2.2 Equipment identity management

DLT technologies are well-suited to provide an effective mechanism that enables equipment management, and more specifically identity authentication and authorization. Quality control requires strict supervision over which equipment has clearance to modify which subsets of data collections. What is more, the logs assembled during the procedure ought to be immutable. By assigning a digital identifier to each piece of equipment, allowing it to univocally "sign" its interactions with data, transparency and, therefore, an uncontested single source of truth are formulated step by step. In practice, DLT provides the possibility of creating unique accounts, fitted with a pair of cryptographic keys; a public one to be universally authenticated and a private one to "sign" transactions. In such a configuration, any interaction with data is signed with the equipment's private key and can be verified by anyone who has access to the latter's public key. This verification proves that the equipment had access to the private key, and therefore is likely to be the one associated with the public key. This also ensures that the digital signature has not been tampered with, as it is mathematically bound to the key it originally was made with. From their part, smart contracts can be employed for both handling authorization requests and translating authorization policies into machine-readable self-executing code.

2.2.3 Smart machine's diagnostics and pay-per-use

DLT can be used for developing diagnostics and self-service applications for "smart" machines, where the machines themselves will be able to monitor their state, diagnose problems, and autonomously place service, consumables replenishment, or

part replacement requests to the machine maintenance vendors. DLT technologies further expand the possibilities for using an innovative pay-per-use model for those services. The rationale is that a "ledger service" is responsible of evaluating the service received (to the satisfaction of both the owner and the user), while the ledger itself ensures that the relevant log recordings cannot be falsified in any way.

2.2.4 Monitoring conformity to SLAs, standards and regulation

The advantages of translating Service Level Agreements to self-imposed smart contracts are noteworthy [Uriarte18]. First and foremost, this process automates their lifecycle stages: discovery and negotiation, deployment, monitoring, billing/penalty and termination. Furthermore, it introduces clarity, since all rules are univocally defined, and transparency, since all interactions between physical and non-physical parties are recorded in a definitive manner. In a real-world application, a "master" smart contract can be designed to enforce legal standards and agreements of any kind. By cross-examining the data uploaded by different stakeholders all parties can verify to what extent the process meets the predefined regulatory conditions. Once all the requirements are met, the regulatory approval may be automatically granted through a smart contract with no further need for on-site inspections or in-person verification.

2.2.5 Circular economy

The term "circular economy" refers to an economic system aimed at employing reuse, sharing, repair, refurbishment, remanufacturing and recycling to minimize the use of resource inputs and the creation of waste. From a manufacturers' perspective it is an attractive option for social, financial, environmental and legal reasons. However, a key success factor in any such a production chain is the condition and quality status of the returned items to be pushed into the remanufacturing process. A distributed ledger that constantly monitors the lifecycle of a product arms a remanufacturer with substantial advantages towards this direction. First, it provides an accurate and unchangeable account of the item's provenance and journey, filtering out counterfeit and misused products. Second, it asserts quite transparently that the item enters remanufacturing being on a status that complies with some minimum quality criteria and desirable specifications. And, finally, it provides a guarantee of the item's status after the finalization of the remanufacturing process, useful, for instance, to compose warranty documents.

3 QU4LITY's DLT infrastructure

As explained in the previous version of this deliverable, the baseline blockchain platform we have selected for QU4LITY is Hyperledger Fabric (HLF), now at release v2.2. Besides meeting all the requirements for QU4LITY's DLT infrastructure, HLF supports multi-tenancy through *channels*, a feature that is used to isolate groups of participants within private encrypted ledgers. When a channel is defined, only those nodes that are allowed to *join* it will be able to access its ledger and participate to its transactions. Moreover, each node can join multiple channels at the same time. This extreme flexibility allows one single HLF infrastructure – i.e., a set of physical nodes – to serve multiple logical networks, each dedicated to a specific business ecosystem.

This was indeed our approach in QU4LITY: one HLF deployment providing the VCL and supporting any number of PLs. To facilitate QU4LITY partners in running their experiments, we choose to run all the VCL nodes on the same Cloud facility, operated by ENG as part of their Hyperlab initiative[1]. This means that a naturally decentralized system is actually managed as a centralized one (a model that is often referred to as "Blockchain-as-a-Service"). With this architecture we don't exploit at all the decentralization concept that is at the heart of the DLT paradigm, but on the other hand we gain a significant advantage in terms of accessibility, as QU4LITY partners that wish to experiment with DLT do not need to set up their own node. This compromise is acceptable in the context of a project that doesn't aim at validating a blockchain platform, but rather at exploring its potential in real-world business cases.

The baseline QU4LITY DLT infrastructure thus consists of the following elements:

- Certificate Authority (CA) – Standard set of tools for releasing identity / access credentials (actually, self-signed X509 certificates) to participants.
- Membership Service Provider (MSP) – Manages identity authentication and authorization, enforcing access control rules on the blockchain network.
- Ordering Service – Defines the system-wide temporal sequence by which transactions, executed concurrently on individual nodes, are applied to the ledger.
- Peer Nodes – Four nodes, all belonging to the same organization (i.e., ENG).

This deployment provides the "minimum viable product" in terms of infrastructure: it implements a basic VCL, with one common ledger that all QU4LITY partners can operate on – once assigned the proper credentials and access clearance. This VCL can be extended with more Peer Nodes if required, and additional VCLs can also be created on demand just by configuring new channels.

Each time there is the need for the deployment of a PL, the following will be added:

- Configuration of a dedicated channel.
- One private Peer Node for each PL participant (at least two), all of them attached to the dedicated channel. It is possible – but not required – for these private nodes to be deployed by the participant organization(s) on their own premises.

[1] See https://www.eng.it/en/enabling-technologies/blockchain

4 QU4LITY's DLT digital enablers

4.1 General view of Decentralized Applications in QU4LITY

The QU4LITY DLT infrastructure described in the preceding section is just what the names suggests: a hosting facility for decentralized applications. The QU4LITY project, however, also provides a basic portfolio of such applications, which are meant to enable decentralized quality management use cases. Each enabler is implemented as a *Decentralized Application (DApp)*. As a DApp provides a coordination workspace for a collaborative process, all participants – generally referred to as *user applications* – need to be integrated into the system. To facilitate this system integration task, the structure of a "standard" DApp – and of its distribution package – consists of three separate elements:

1. A smart contract implementation, which is actually an HLF *chaincode* program. The chaincode exposes an access-controlled API that is meant to be invoked by client software. Such API is *not* documented, because user applications will never invoke it directly; instead, they will use the *local proxy* included in the distribution package (see point #2).
2. A client library for Java environments, packaged as a single JAR file that also contains the software documentation in JavaDoc format. The library provides an in-process API to user applications. The in-process API is actually a higher-level abstraction of the chaincode's API. User applications will invoke the remote API through the local proxy in a simple and straightforward way: the technical details of how the client library connects to and communicates with the chaincode (low-level gRPC protocols, authentication / authorization) are kept hidden from the caller. Moreover, all data structures defined by the in-process API are implemented in the client library as Java classes, so that user application coders don't need to deal with text-based representations like XML or JSON.
3. The general documentation of the DApp. This is not the same as the API documentation, because it explains the use case workflow and goes into details of the data model – i.e., the structure of the "application state" that is managed by the chaincode.

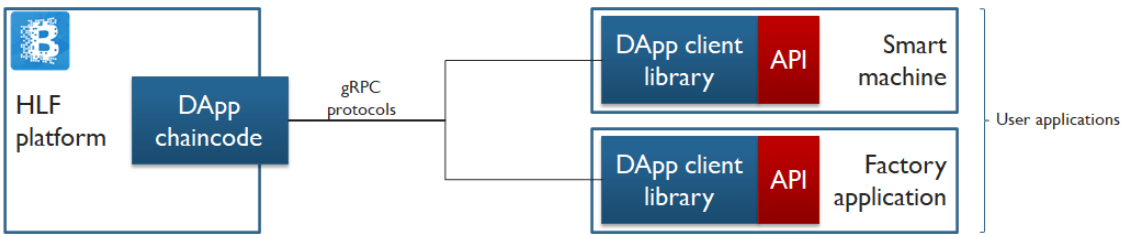

Figure 2 - DApp structure

The integration of a user application with a standard DApp starts by "importing" the client library. The choice of the Java language for the development of the client library is a constraint: although it is theoretically possible to call Java libraries from other

environments, the most straightforward way to do it is that both the user application and the client library are executed within the same JVM. The minimum supported version of the Java environment is 1.8.

As a side note, it is worth pointing out that other ways of packaging and integrating a DApp do exist and have actually been followed in some specific cases: in the sections that follow (see §4.2.2 and §4.2.3), these cases will be explained in some detail. However, the chaincode + client library approach is considered a the "golden standard" for QU4LITY's VLC and PL(s), and DApp developers are encouraged to adopt it.

Once the integration layer has been coded into the user application, proper configuration is needed in order for the integrated system to work. Two things are required on the user application's side: correct network parameters and valid credentials for access. The former are pointers to a specific chaincode instance deployed on a given channel of a given HLF network; the latter is a digital certificate, released by the Certificate Authority, that identifies an authorized user. Both are provided by Hyperlab's admins on request and delivered as a single ZIP file that must be copied to the local filesystem of the device hosting the user application.

Once everything is set up, using a DApp through its local client is not much different than making in-process calls. One thing that developers of user applications need to be aware of is that calls, despite being local, are actually triggering a transaction on a remote distributed system. Although chaincode transactions are asynchronous, the client library will simulate a synchronous call by blocking the caller until the remote transaction is confirmed on all the nodes of the system. This introduces significant latency: the time elapsed until control is returned to the caller is typically some orders of magnitude longer than an equivalent local call. Mileage may vary, so developers are strongly encouraged to verify that the latency they experience is well-tolerated by their AQ/ZDM process.

4.2 QU4LITY's Decentralized Application catalogue

Three DApp enablers have been developed in the scope of the QU4LITY project, one of which plays an additional "support" role for the other two:

- **Quality Clearing House (QCH)**. QCH enables a decentralized workflow that targets a typical Quality Management / ZDM process in a non-hierarchical supply chain scenario.
- **Secure Message Board (SMB)**. SMB enables the publishing of sensitive content (e.g., software, firmware, algorithms, parameters, data, documents) to subscriber channels, with the added value of strong guarantees on provenance, integrity and confidentiality.
- **Secure Identity Directory (SID)**. SID enables the decentralized management of digital identities and authentication credentials for both human and machine actors. SID is also used by both QCH and SMB as their identity management infrastructure.

The chaincode of all these DApps is currently deployed on the QU4LITY VCL infrastructure: any user application that has been properly integrated and configured (see previous section) can access their functionality by connecting through the public Internet. The source code of both the chaincode and of the client library has been released under the terms of the Apache v2.0 open-source license.

4.2.1 Quality Clearing House DApp

4.2.1.1 QCH general description

QCH enables a decentralized workflow that targets a typical Quality Management / ZDM process in a non-hierarchical supply chain scenario. It provides a common "system of record" for a manufacturing ecosystem where actors need to continuously assess the quality of raw material, parts and final products and match the results against contractual standards that may change frequently. Thanks to distributed ledger technology, QCH records are secure and trustworthy: they are timestamped, immutable and non-repudiable. Data storage and business logic are replicated on all the nodes of the system, which are operated equally by all participants, so that no single "owner" of the system exists who may introduce bias in the process.

4.2.1.2 QCH data model and workflow

The supply chain processes supported by QCH follow a simple pattern, the workflow of which is described below. To exemplify the pattern and for the sake of simplicity, we have identified distinct "actors" playing the three roles embodied in the system (Quality Master, Supplier, Quality Verifier); however, in real-world supply chain processes it is likely that multiple organizations will play the Supplier role, and/or that one single organization will play the remaining ones.

Ledger records

All ledger records have their own unique identifier, which is used internally for cross-reference, and are owned by the party that creates them.

- Quality Assessment Data Model (QADM): a structured digital document that defines the standard of quality that applies to a given material, part or product, as stipulated by a commercial agreement (which is out of this scope). The standard is expressed in terms of a list of measurements, each consisting of a qualitative definition[2] and a quantitative range. One QADM document may exist for the entire duration of a contract, or new versions may be created that override previous ones in order to follow along the evolution of quality requirements.
- Shipping Unit Manifest (SUM): a digital record that identifies a shipped batch of materials, parts or products as subject to a given quality agreement. It consists of a pointer to an existing QADM and of a list of IDs, each associated to a physical item in the batch.
- Quality Assessment Report (QAR): a digital record that reports the quality measurements taken on a received batch of materials, parts or products, along the guidelines of their agreed standard. It consists of pointers to an

[2] The vocabulary used to identify measurements must be in common between all parties involved: the meaning of each measurement declaration, which includes not only the "what" but also the "how" and possibly the "when", must be unambiguous for everyone. To this goal, a formal ontology may be defined.

existing QADM and SUM, plus the actual values of all the measurements taken. Depending on the quality agreement in place, measurements may be reported per-batch (average values) or per-item.

Process roles

- Quality Master: it's the company that manages the process. It creates the QADM document(s).
- Supplier: it's a member of the supply chain that manufactures / provides materials, parts or products. It creates SUM records.
- Quality Verifier: it is responsible of measuring the quality parameters on physical items with respect to the standard. It may be the same entity as the Quality Master or a different one – i.e., a third party in charge of independent assessment. In the latter case, it should be trusted by all the involved parties. The Quality Verifier creates QAR documents.

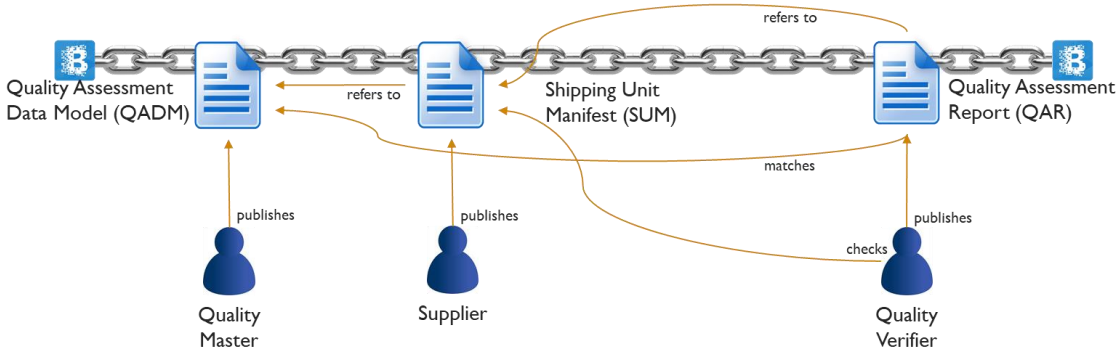

Figure 3 – QCH DApp: ledger records and process roles

Example

Factory A plays the role of Quality Master.
Factory B plays the role of Supplier.
Company C plays the role of Quality Verifier.

1. When a commercial agreement is first defined, Factory A defines the quality standard and creates a new QADM, which is published on the QCH. Factory A also sets up and configures its quality assessment process and tools in collaboration with Company C, which provides the metrology equipment that is deployed on Factory A's premises.
2. Factory B prepares a batch of goods under the aforementioned agreement. The physical items in the batch are tagged with individual IDs. When the batch is shipped, Factory B publishes a new SUM record on the QHC that points to the reference QADM and lists all the IDs contained in the shipment.
3. Factory A receives the shipment. As the individual items herein contained are unloaded, they are sent to a quality assessment facility where the equipment provided by Company C is in use. The metrology tool read the tag, identifies the item and execute the appropriate measurements.
4. When the shipment has been entirely processed, a QAR is generated by the metrology tool and published on the QCH on behalf of Company C.
5. The process can now iterate any number of times, starting from step #2.

6. When payment to Factory B is due, Factory A will apply any penalties and discounts defined in the agreement for missed quality targets documented in the QCH.

4.2.1.3 QCH API

As already explained in §4.1, the QCH API is exposed by a Java library. The Java classes that are available to the embedding application are the following:

- `QualityModel`: a Java "data transfer object" (DTO) that represents a QADM entity; contains 1-n QualityParameter DTOs.
- `Shipment`: a DTO that represents a SUM entity; contains 1-n Item DTOs.
- `QualityAssessment`: represents a QAR entity, contains 1-n ItemAssessment DTOs.
- `QualityClearingHouse`: an interface that represents the smart contract's local proxy, through which all read and write operations on the distributed ledger can be executed. The concrete implementation of the interface is hidden by the factory class (see next point).
- `QualityClearingHouseFactory`: a factory class exposing a single static method, which creates an instance of the smart contract's local proxy (see previous point). The method needs a pointer to a local configuration directory, the contents of which determine the actual implementation class that will be used as the proxy.

To invoke QCH operations, the embedding application must first obtain an instance of the proxy class from its factory:

```
QualityClearingHouse proxy =
      QualityClearingHouseFactory.getProxy(configPath)
```

The factory method receives a single parameter, which must be the absolute path of the local configuration directory. In order for this to work at runtime, a local configuration directory must be created as part of the application's deployment procedure. The directory is created by expanding the `wallet.zip` file that is provided – on demand – by the administrator of the VCL. The ZIP file contains not only the correct network pointers to the QCH smart contract, but also the authentication credentials that the client library will use to establish the connection. For this reason, the `wallet.zip` is strictly personal for each authorized user and should never be shared with anyone else.

Once the embedding application has obtained the proxy object, this can be used for all the operations documented in the UML class diagram below – see Figure 4 below.

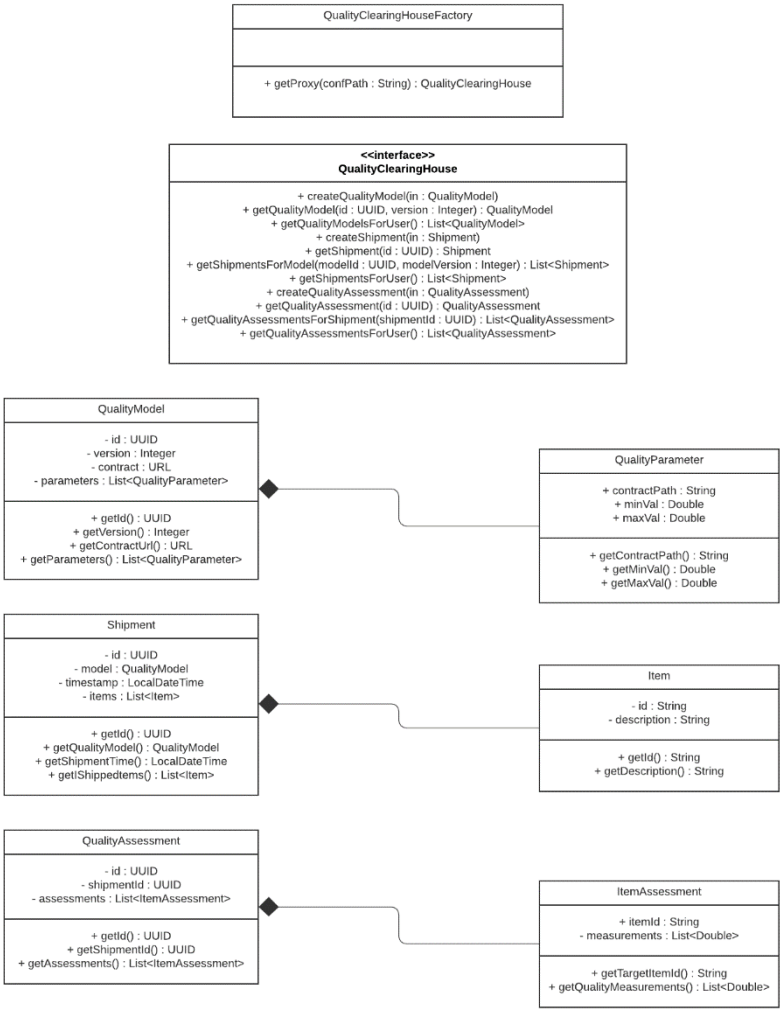

Figure 4 - QCH API documentation

4.2.1.4 QCH availability

The QCH smart contract is installed on QU4LITY's VCL. In order to receive the personal `wallet.zip` file that enables client access, developers and integrators should contact the ENG team that is responsible for QU4LITY's task T3.6. Any QU4LITY partner is allowed to experiment online with the DApp, but with the caveat that the VCL is just a "sandbox" environment: no guarantees are provided by the VCL administrator that the system will be up 24/7, that data will be persisted for long periods of time and that security will be properly managed. *Under no circumstances should this shared instance of the DApp be used to store valuable or confidential information or to manage critical processes.* If this is indeed the case, we recommend a dedicated installation of the QCH smart contract on a privately-owned PL.

The source code of the QCH DApp – including the smart contract – is publicly available on GitHub: https://github.com/Engineering-Research-and-Development/qu4lity-dapps/tree/master/chq.

4.2.2 Secure Message Board DApp

4.2.2.1 SMB general description

SMB enables the publishing of sensitive content (e.g., software, firmware, algorithms, parameters, data, documents) to subscriber channels, with the added value of strong guarantees on provenance, integrity and confidentiality. For instance, an OEM may safely send customized software updates to any specific smart machine installed at a client site, or to all machines of a certain type. While the actual content is stored as a *binary object* on any cloud facility, an immutable and timestamped record on the ledger provides a "digital seal" that guarantees its authenticity: the record includes a hash value calculated on the content, so that any alteration to the latter can be easily detected. Moreover, the SMB smart contract – in cooperation with SID (see §4.2.3) – ensures that the publisher is always correctly identified and cannot be impersonated by a malicious actor.

This version of the DApp is distributed together with a command-line client tool that offers a simple interactive interface. The client tool provides the basic read / write functions: a POST command for publishing new messages or replacing exiting ones with an updated version, a GET command for retrieving the latest version of given message and a VERSION command that queries for the current version number of a given message.

It is worth noting that the current implementation only supports Google Drive as the cloud-based persistent storage for binary objects. Future versions will remove this limitation.

4.2.2.2 SMB data model and workflow

The SMB workflow is straightforward. The Publisher, who must already own a registered SID identity, publishes a content item by executing a command-line tool. The tool must receive as the execution arguments at least the *name* assigned to the item and a local filesystem path that points to its *content*.

The item name can be specified within the namespace defined that is composed of up to three nested components: the *domain*, the *environment* and the *process* (in decreasing order of scope). So, for example, a content item can have the plain "myNewItem" name, which implies that "myNewItem" is a unique identifier in the top-level scope; while another may be identified as "myDomain/myEnvironment/myProcess/myNewItem": although this shares the same name with the previous item, it lives in the "myDomain/myEnvironment/myProcess" scope, so that the two homonymous items are actually distinct and unrelated. It is important to understand that **each namespace, including the anonymous top-level one, defines a publishing Channel**: Subscribers can "listen" on one or more specific Channels and will receive a notification when a new item is published there. Another important point is that published items can be updated: when an item is published on a Channel and has the same name of a previously-published one on the same Channel, the new item replaces the previous one as the "current version". All versions are kept in permanent

storage, but the latest is the one that is retrieved when no version number is specified in the request.

Once the tool is launched for a POST operation, as described above, it first uploads the item contents to the cloud storage, which will store it as an opaque binary object identified by a unique key. The tool then calculates the *hash value* of the content, determines its size in bytes and finally invokes the SMB smart contract, impersonating the Publisher's identity (to this goal, the private wallet containing the Publisher's credentials must be installed together with the tool). Finally, the smart contract (after verifying the caller's identity) creates a new record on the ledger: the Message Manifest. This record is basically a public announcement – to whoever has access to the ledger – that the new item has been published. Moreover, the record contains a pointer to the binary object in the permanent storage and, most importantly, the SID identity of the Publisher and the hash value of the content.

When instead the tool is launched for a GET operation – receiving the name of the items as an argument – it first reads the corresponding Message Manifest record on the ledger, from where it retrieves the information described before. It then proceeds to download the item's content from the cloud storage, using the pointer. The content is saved as a file on the local filesystem. Before confirming to the caller that the retrieve operation was successful, the tool checks that the hash value stored in the Message Manifest is actually matching the hash value calculated on the saved copy. This ensures that no tampering of the content has happened after the item was published.

Ledger record

The structure of the Message Manifest record can be seen in Figure 5 below:

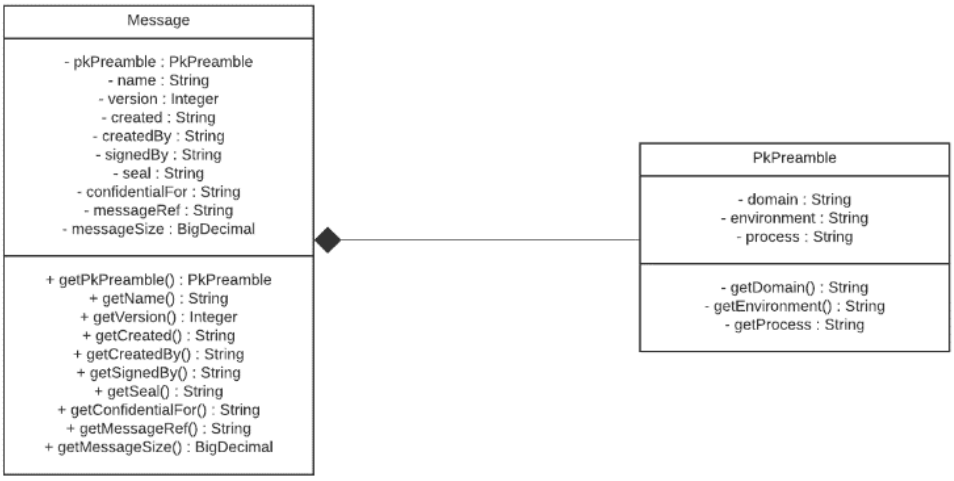

Figure 5 – SMB DApp: data model

Note that the `signedBy` and `confidentalFor` fields are unused in the current version of the DApp. In the future, they may be used to verify the digital signature of the Author (who may be a different entity than the Publisher) and for implementing one-

to-one confidentiality (content is encrypted by the Published using the public key of the Subscriber).

Process roles

- Publisher: sender of messages. It needs to send messages "securely" (i.e., ensuring that the original message cannot be tampered with) on a Channel, possibly without knowing in advance the recipients.
- Subscriber: receiver of messages. It needs to receive messages sent on any Channel it is interested to, and also to verify that the received messages are "authentic" (i.e., they come from a well-identified Publisher and have not been altered).

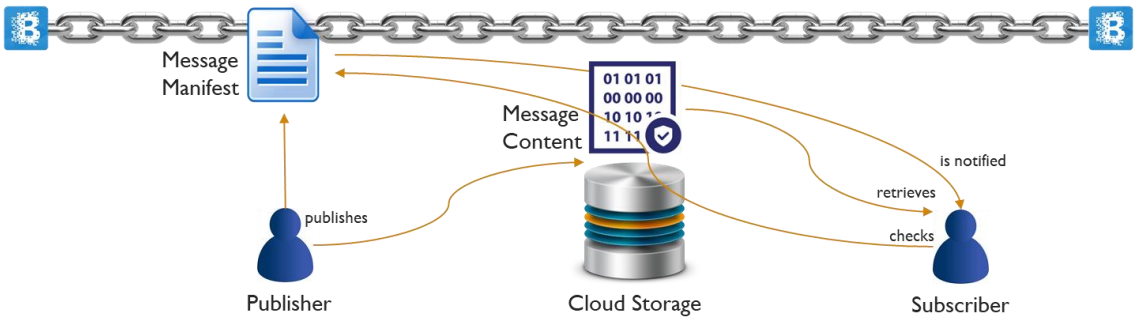

Figure 6 - SMB DApp: ledger records and process roles

Example

OEM plays the role of Publisher.
Smart Tool A and *Smart Tool B* play the role of Subscriber.

Context: Smart Tool A and Smart Tool B are two intelligent manufacturing machines of the same type, manufactured by OEM, and are installed in two different factories. OEM has a remote management agreement with its customers: the output quality of its tools is measured on-site and reported to the OEM, the indicators reported are analyzed, possible optimizations to the tool's parameters and logic are implemented by the OEM as firmware updates, and finally the updates are pushed back to the running tools, without any disruption in shopfloor operations or even human intervention. The last part of the process is the one covered by the SMB DApp in the use case that follows. Smart Tool A & B both embed an SMB client.

1. When Smart Tool A is deployed, it is configured such that the embedded SMB client will operate under its own unique SID identity (a local wallet containing the identity's private key is created) and will listen to two SMB Channels that are named as follows:
   Channel Main (model-specific updates)
   ```
   domain=OEMCompanyName-SmartToolModelName
   environment=(void)
   process=(void)
   ```
   Channel A (machine-specific updates)
   ```
   domain=OEMCompanyName-SmartToolModelName
   environment=RemoteQualityManagementAgreementABC
   ```

`process=SmartToolSerial0015302`

2. When Smart Tool B is deployed, it goes through the same installation process as Smart Tool A, but the second configured Channel is different:
   Channel B (machine-specific updates)
   `domain=OEMCompanyName-SmartToolModelName`
   `environment=RemoteQualityManagementAgreement`**`XYZ`**
   `process=SmartToolSerial0015`**`677`**
   This configuration reflects the fact that the remote management agreement between OEM and the factory that operates Smart Tool B is different, and of course that the Smart Tool B is a different physical machine. Note that this same factory may deploy additional machines of the same model, which will share the same "environment" but have their distinct "process" – meaning that they will all listen to different machine-specific Channels.
3. During the operational lifetime of Smart Tool A, the continuous quality assessment process identifies a specific fix that will improve an indicator. Consequently, OEM prepares a patch to be applied *only to* Smart Tool A. This patch will not affect the basic logic but only some site-specific parameters.
4. OEM publishes the patch on SMB, setting the Channel arguments as seen at step #1, Channel A. The "name" property of the published item is set to "parameter-patch".
5. Smart Tool A, which listens to both Channel Main and Channel A, detects that a new item has been published on channel A. It then retrieves the item, checking that the Publisher is actually OEM and that the content matches the hash signature. As the item name is "parameter-patch", Smart Tool A identifies the content as a patch to be applied on its own site parameters.
6. Smart Tool A automatically applies the patch as instructed, thus improving the quality of its output.
7. OEM designs an improvement to the logic of the product model that both Smart Tool A and Smart Tool B belong to. Such improvement is implemented as a firmware update. Consequently, OEM prepares a patch to be applied to all machines of that type. This patch will affect the basic logic but will leave any site-specific parameter unchanged.
8. OEM publishes the patch on SMB, setting the Channel arguments as seen at step #1, Channel Main. The "name" property of the published item is set to "firmware-patch".
9. Smart Tool A and Smart Tool B, both listening to both Channel Main, detect that a new item has been published on the channel. They then retrieve the item, check that the Publisher is actually OEM and that the content matches the hash signature. As the item name is "firmware-patch", both Smart Tool A and Smart Tool B identify the content as a patch to be applied on their own firmware.
10. Smart Tool A and Smart Tool B signal to their local operation environment that a firmware path is ready to be installed. When the operators deem that this is possible with minimal disruption of the shopfloor processes, they manually activate the update.

Steps 3 to 6 can be repeated any number of times, for both Smart Tool A and Smart Tool B. Same for steps 7-10. The two sequences may be intertwined in any order.

Note that firmware updates may also include a different configuration of the SMB client, thus changing the Channels that machines are listened to.

4.2.2.3 SMB command-line tool

The CL tool is not interactive: to perform a single operation, it must be launched from the system with a specific command. The command is structured as follows (it is assumed that the Java JRE has been configured correctly):

`java - jar smb-ledger-<version>.jar`

- `-w` *(mandatory)*: path to a local directory containing the wallet files
- `-o` *(mandatory)*: operation requested [`POST`, `GET` or `VERSION`]
- `-f` *(mandatory for* `POST` *and* `GET` *operations)*: path of the local file that either holds the content of the new item to be published (`POST`) or will be created to hold the content of the existing item to be retrieved (`GET`)
- `-d` *(optional)*: "domain" component of the Channel identifier
- `-e` *(optional)*: "environment" component of the Channel identifier
- `-p` *(optional)*: "process" component of the Channel identifier
- `-n` *(mandatory)*: name of the content item
- `-v` *(optional, supported only for* `GET` *operations)*: version number

At each execution, the related log (activity and any errors) is added to the end of the smb-ledger.log, located in the logs sub-folder.

Examples for Windows systems (mandatory arguments in bold):

- **java -jar smb-ledger-1.0.0.jar**
  -w C:\Users\mywindowsuser\smb\smbuser
  -o POST
  -d mydomain -e myenv -p myprocess **-n myobject**
  -f C:\Users\mywindowsuser\Documents\mycontent-myversion.bin

- ***java -jar smb-ledger-1.0.0.jar***
  -w C:\Users\mywindowsuser\smb\smbuser
  -o GET
  -d mydomain -e myenv -p myprocess -v targetversion ***-n myobject***
  -f C:\Users\mywindowsuser\MyFolder\mycontent.bin

- ***java -jar smb-ledger-1.0.0.jar***
  -w C:\Users\mywindowsuser\smb\smbuser
  -o VERSION
  -d mydomain -e myenv -p myprocess ***-n myobject***

4.2.2.4 SMB availability

The SMB smart contract is installed on QU4LITY's VCL. In order to receive the personal `wallet.zip` file that enables client access, developers and integrators should contact the ENG team that is responsible for QU4LITY's task T3.6. Any QU4LITY partner is allowed to experiment online with the DApp, but with the caveat that the VCL is just a "sandbox" environment: no guarantees are provided by the

VCL administrator that the system will be up 24/7, that data will be persisted for long periods of time and that security will be properly managed. *Under no circumstances should this shared instance of the DApp be used to store valuable or confidential information or to manage critical processes.* If this is indeed the case, we recommend a dedicated installation of the SMB smart contract on a privately-owned PL.

The source code of the SMB DApp, including its command-line client tool, is publicly available on GitHub: https://github.com/Engineering-Research-and-Development/qu4lity-dapps/tree/master/smb.

4.2.3 Secure Identity Directory DApp

4.2.3.1 SID general description

SID is a decentralized infrastructure for identity management. It answers the problem of how a decentralized application may depend on common, trustworthy information about user identities without introducing any centralization bottleneck. Therefore, SID is centered on a *smart contract* that can be deployed either on the QU4LITY VCL or on any PL. User and machine identities are registered on the ledger in a format that is compatible with the DID standard (see next section) but also contains a rich (and extensible) set of information. This decentralized registry of identity descriptors allows any DApp that supports SID (including SID itself, as will be explained later) to assess the identity of their users in a secure way and, optionally, to enforce access restrictions according to their own policies.

4.2.3.2 SID data model and workflow

Every identity in the SID registry is associated to an alphanumeric ID, which in SID jargon is also called an *address*, because it's actually a globally unique name that points to the same identity on any network and on any system. Together with the address, every identity descriptor in the SID registry also contains a *public key*, for the purpose of validating ownership claims: it is assumed that only the legitimate owner of an identity holds – in private storage – a copy of the matching *private key* (this rule must be enforced by the identity creation process, as we will see later). On this assumption, applications and online services can authenticate any SID-registered user by obtaining a cryptographic proof of the possession of the private key. This is done by means of a challenge-response protocol:

- the user claims to be the legitimate owner of a given SID address
- the verifiers look up the corresponding public key in the SID registry
- the verifier creates a string of random text and sends it to the user as a "challenge"
- the user digitally signs the challenge with its private key, and sends the result to the verifier as the "response"
- finally, the verifier checks that the signature in the response actually matches the challenge.

In this way, the identity claim is verified without any disclosure of the private key – as opposed to what happens with a password, which must be a "shared secret" known to both the user and the verifier.

The above-described mechanism is commonly used in scenarios where users are purposely kept anonymous – like cryptocurrencies and self-sovereign identity (SSI) schemes. Indeed, SID is compatible with SSI systems because identity descriptors are easily translated into a W3C standard interoperability format called *DID Document* (see https://www.w3.org/TR/did-core/). However, SID is focused on B2B and IIoT interactions, where anonymity is not an option. Thus, identity descriptors in the SID registry are enriched with additional information that points, unambiguously, to the *real-world entity* (RWE) that qualifies as the legitimate owner.

Ledger records

The structure of the identity descriptor can be seen in Figure 7 below. It is actually composed of one "Identity" record and one or more related "IdentityStatus" sub-records. The latter are used not only to define the current status of an identity (`active`, `suspended`, `revoked`) but also to track the status history back in time, so that it would be possible to assess – for instance – is a given identity was active at a given past date/time.

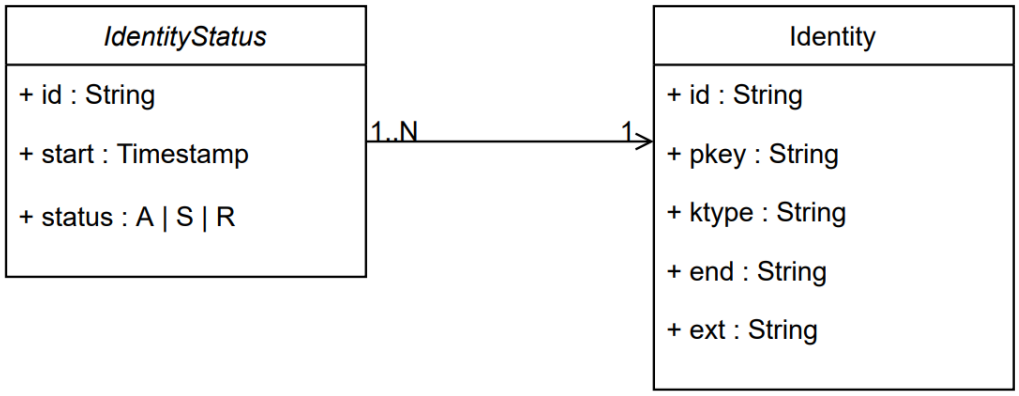

Figure 7 - SID DApp: data model

The `pkey` field contains the value of the identity's public key, encoded as a string using the common Base64 algorithm. The public key is the actual unique identifier, because the `id` field – the identity's address – is computed from the public key's binary value with an algorithm similar to that used in the Bitcoin system for "Bitcoin addresses". The `end` field, when present, is a reference to the `id` of the identity's controller – i.e., the SID identity that is responsible for management, if any (for more info, see the coming section on the identity management workflow). Finally, the `ext` field is used to extend the identity descriptor with additional information. In the SID DApp, this is actually the contact information of the RWE that owns the identity. This is represented with the following JSON structure:

```
{
  "DAMP:RWE" : {
    "LegalName" : "..",
    "LegalAddress" : {
      "AddressLine1" : "..",
      "AddressLine2" : "..",
      "AddressLine3" : "..",
      "City" : "..",
      "Region" : "..",
      "Postcode" : "..",
      "Country" : ".."
    },
    "DUN" : "..", // Data Universal Numbering System
    "BIC" : "..", // Business Identifier Codes
    "TIN" : "..", // Taxpayer Identification Number
    "LEI" : ".."  // Legal Entity Identifier
  }
}
```

Identity management workflow

In the SID world, every user can read the public registry without restrictions, executing a "read" API operation. On the other hand, creating or modifying an identity descriptor – executing a "write" API operation – can only be done under certain conditions: basically, the user must qualify either as a platform-level administrator or – more commonly – as the "controller" of the target identity. Here is where things get a bit complicated, and some understanding of how SID authentication and access control work is required.

Authentication in SID happens on two levels: the *native* Hyperledger Fabric (HLF) protocol layer and, on top of it, the SID challenge/response protocol previously described. The lowest level ensures that the calling client has been authorized by the HLF platform's administrator to connect to the HLF network and invoke HLF smart contracts in general – and the SID smart contract in particular. This is a very broad rule, but is required by the HLF system, as HLF is a "permissioned" blockchain. To pass this check, an X509 digital certificate signed by the platform's certificate authority (usually, the same entity that has the role of platform's administrator), must be issued to the user and installed on the user's client software. Once this pre-requisite is met, the user is able to invoke both "read" and "write" SID API operations. However, when the SID smart contract is invoked for a "write" operation, it will *also* make an additional check: the caller must either have a "special" role as an HLF platform user ("native" authentication + authorization) or, if that is not the case, it must be the legitimate owner of a pre-existing SID identity (SID authentication); moreover, the SID identity of the caller must be the *controller* of the SID identity that is the target of the operation. For example, if User A is the owner of Identity A and wants to create a new identity for User B – i.e., Identity B – the descriptor if the new identity must hold a reference to Identity A as the controller (actually, the `end` field of the record – see the data model in the previous section). One the Identity B

record has been created and registered in SID, only User A will be allowed to make changes to that record – for instance, setting its status to Suspended or Revoked.

The practical impact of this authorization scheme is that "regular" users (in other words, those who do not have any special privilege on the platform, which is the typical case) wishing to have their own identity registered on SID will have to ask another SID user to endorse them. This endorsement should not be given lightly, because the controller of a SID identity is to all effects responsible for ensuring A) that the information registered on the ledger (e.g., the RWE contact information – see the data model section) is correct, and B) that only the identity owner has the matching private key. While the first condition depends on factors that are outside the scope of the SID DApp, the second is easily achieved by the proper use of a software component that is also part of the SID distribution: the SID Wallet command-line tool – see §0.

4.2.3.3 SID API

The SID API is *not* exposed directly by the smart contract. Instead, a REST-over-HTTP interface is provided by a "SID API server" that acts as a mediator between the caller and the smart contract. This architectural choice makes life much easier on the client side, because standard HTTP clients (e.g., web browsers) are supported, but has a profound impact on how the SID DApp is deployed and used. Clearly enough, standard HTTP clients are not capable of handling the SID challenge-response protocol, so the SID API server must *impersonate* the caller and forward every call to the smart contract, taking full responsibility for authentication. To do that, the SID API server needs full access to the private credentials of the user, which are kept locally in an encrypted vault: the *SID Wallet*. Consequently, **the SID API server must be installed and run as a *personal gateway* for a specific user, and embed its SID Wallet** (see Figure 8). This should not be a big an issue on most devices, because the server is implemented as a lightweight Jetty container (see https://www.eclipse.org/jetty/). The usage of the server as a "regular" shared network service is *not* supported.

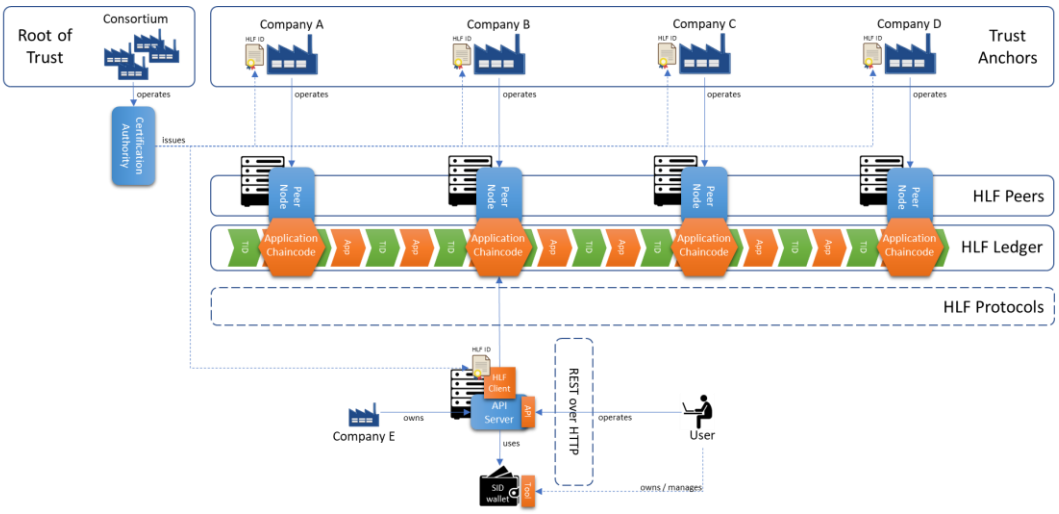

Figure 8 - Deployment of a SID system

The REST-over-HTTP interface exposed by the API Server is very simple, as it only provides the means for registering new identities, setting their status (active, suspended or revoked) and any application-specific additional data (the "`ext`" field of the identity record), plus obviously reading back any registered identity record. The four API calls supported are listed below:

`POST /ten/identity` – Registers a new identity

`GET /ten/identity/{id}` – Retrieves the identity corresponding to the "`id`" resource

`PUT /ten/identity/{id}` – Sets the status of the "`id`" identity

`PUT /ten/identity/{id}/ext` – Sets the "`ext`" field of the "`id`" identity

As mentioned above, the API Server is a "personal gateway" that must impersonate the user when interacting with the SID API on the DLT infrastructure; for this reason, all the REST calls that require user authentication (basically, `POST` and `PUT` methods) need the caller to provide as additional arguments the password to unlock the SID Wallet embedded in the API Server *and* the user identity to be impersonated. This means that if the API Server is deployed on a remote system (with respect to the caller), it is of paramount importance that only encrypted HTTPS channels are used.

The four REST API calls are documented in the following figures, which are actually screenshots of the Swagger web tool installed on an experimental API Server:

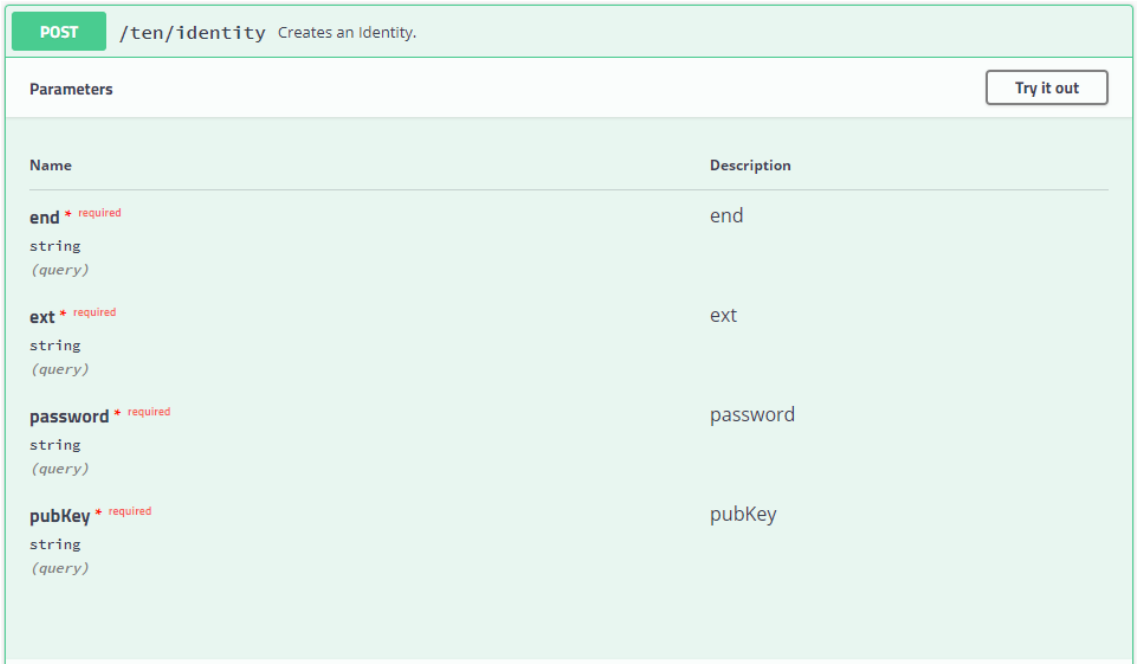

Figure 9 - SID REST API: detail of the POST method for registering a new identity

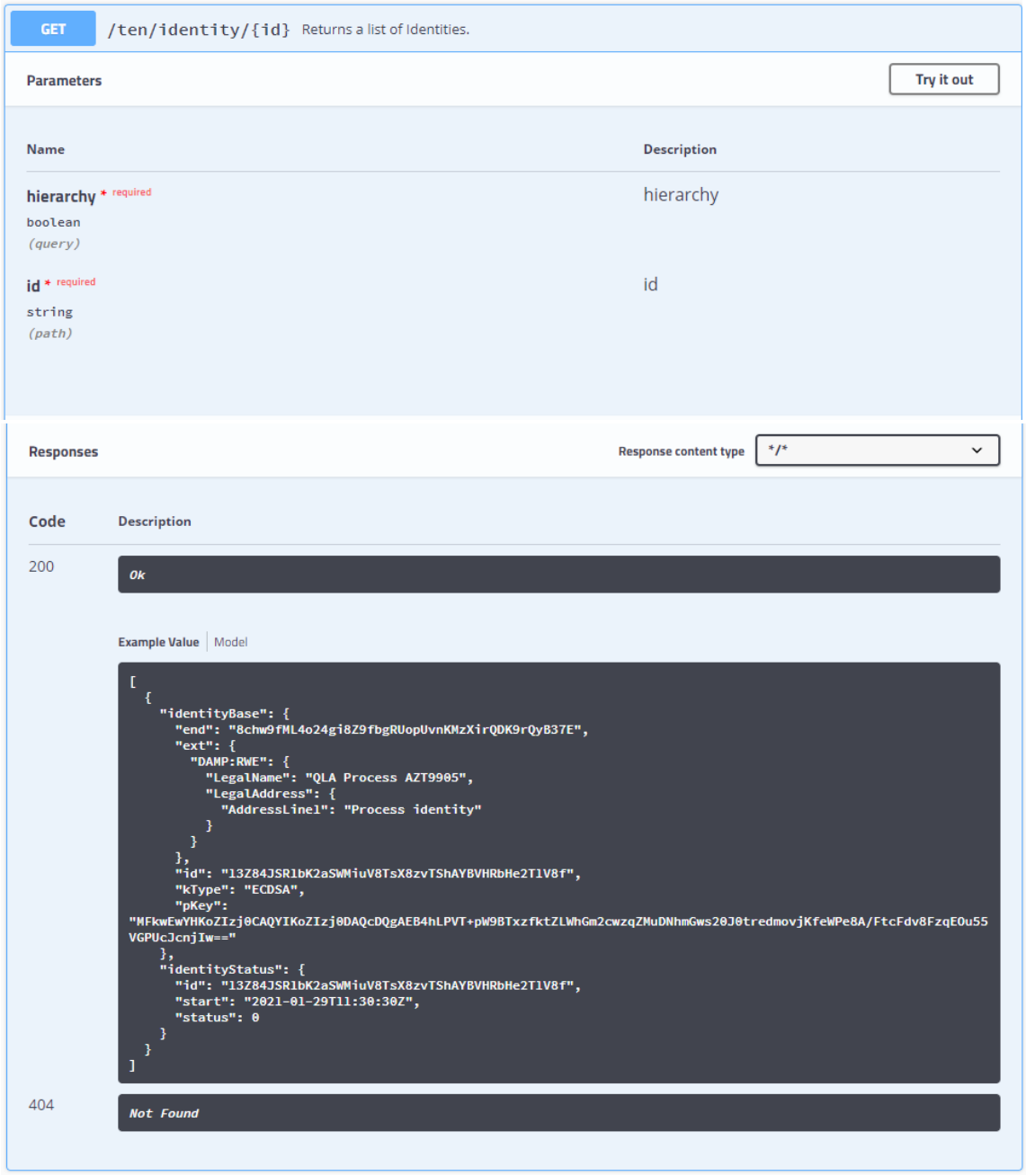

Figure 10 - SID REST API: detail of the GET method for retrieving one specific identity

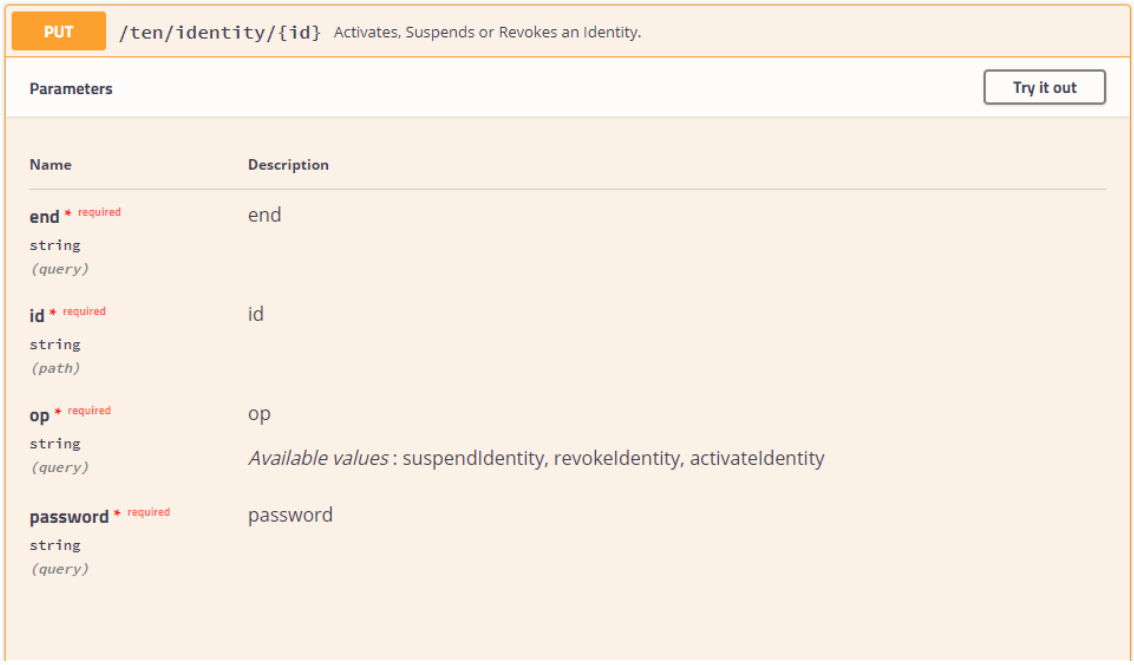

Figure 11 - SID REST API: detail of the PUT method for setting the status of an identity

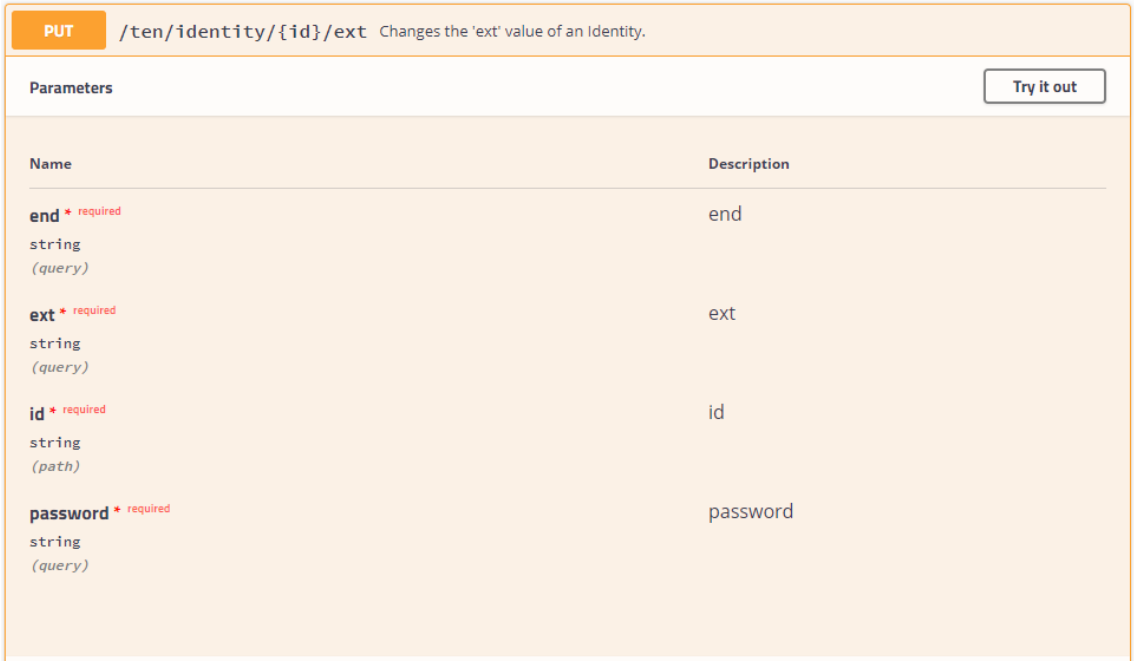

Figure 12 - SID REST API: detail of the PUT method for setting the "ext" field of an identity

4.2.3.4 SID Wallet command-line tool

To allow users to manage the identity wallet embedded in their personal SID API server, the SID DApp includes a simple command-line tool. By issuing commands, the user can insert new identities (which will then need to be registered on the SID ledger by some controller) in its wallet and browse through its contents.

The CL tool is not interactive: to perform a single operation, it must be launched from the system with a specific command. The command is structured as follows (it is assumed that the Java JRE has been configured correctly):

```
java - jar <path to the executable file>
```

- `-c` *(mandatory)* command: `gen`, `show`, `list`
  - `gen` generate a new identity and store it in the wallet
  - `show` show the details of a given identity (use with `-a`)
  - `list` list all identities contained in the wallet
- `-p` *(mandatory)* password that unlocks the wallet
- `-a` *(mandatory for* `show` *command)* address of the identity to display
- `-h` help: list available commands

4.2.3.5 SID availability

The SID smart contract is installed on QU4LITY's VCL. In order to receive the personal `wallet.zip` file that enables client access, developers and integrators should contact the ENG team that is responsible for QU4LITY's task T3.6. Any QU4LITY partner is allowed to experiment online with the DApp, but with the caveat that the VCL is just a "sandbox" environment: no guarantees are provided by the VCL administrator that the system will be up 24/7, that data will be persisted for long periods of time and that security will be properly managed. *Under no circumstances should this shared instance of the DApp be used to store valuable or confidential information or to manage critical processes.* If this is indeed the case, we recommend a dedicated installation of the SID smart contract on a privately-owned PL.

The source code of the SID DApp, including its REST API, is publicly available on GitHub: https://github.com/Engineering-Research-and-Development/qu4lity-dapps/tree/master/sid

5 Conclusion

This deliverable described the final results of tasks T3.6, providing a summary of the work done in the initial part of the schedule (up to M12) and the documentation of the three digital enablers developed during its second part (M13-27).

Task T3.6 delivered a fully-functional DLT infrastructure to support quality management processes across multi-stakeholder value chains. The decentralized nature of DApps makes it possible for all participants to collaborate as *peers* without any central authority in the role of coordinator and rule enforcer. Such infrastructure is now available to all QU4LITY partners as a "virtualized" Hyperledger Fabric network (i.e., a full network where all peer nodes are running on the same Cloud facility) hosted by ENG and accessible through the public Internet.

All the three DApps released as part of the DLT infrastructure are, generally speaking, digital enablers of secure state sharing and synchronization of distributed industrial processes, supporting the traceability of the data and improving access across different autonomous organizations. That said, one of them – namely, SID – is actually an "enabler of enablers": it provides the other two with the capability of controlling access in a secure way, introducing decentralized identity management and a solid authentication / authorization mechanism that can be used by Hyperledger Fabric's smart contracts instead of the native platform-level security, which is centralized and much less flexible. SMB is a generic tool for the secure exchange of trustworthy messages and documents across a business ecosystem. In the context of the QU4LITY project, SMB will be used in a pilot site to distribute *corrective software updates* to a network manufacturing machines, following the results of quality analysis of finished products. Finally, QCH is a tool for collaborative quality management – actually, the only one that is designed according to a specific business process.

In the overall perspective of the QU4LITY project, the QU4LITY DLT infrastructure is a unique asset, because it opens up a wealth of new possibilities: it is not meant to optimize existing "centralized" quality management processes, but rather to disrupt and innovate by introducing *trustworthy decentralization*.

References

[Andrews17] Andrews, Colin, et al. "Utilising financial blockchain technologies in advanced manufacturing." (2017).

[Kim18] Kim, Henry M., and Marek Laskowski. "Toward an ontology-driven blockchain design for supply-chain provenance." Intelligent Systems in Accounting, Finance and Management 25.1 (2018): 18-27.

[Ott19] B. Otto et al, "IDS REFERENCE ARCHITECTURE MODEL - INDUSTRIAL DATA SPACE, version 3.0" 2019. [Online]. Available: https://www.internationaldataspaces.org/en/ressource-hub/publications-ids

[Uriarte18] Uriarte, Rafael Brundo, Rocco De Nicola, and Kyriakos Kritikos. "Towards distributed SLA management with smart contracts and blockchain." 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, 2018.

[Wang17] Wang, Jun, et al. "The outlook of blockchain technology for construction engineering management." Frontiers of engineering management (2017): 67-75.