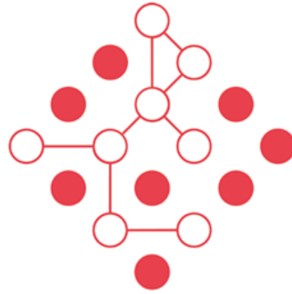

DATA MARKET
AUSTRIA

www.datamarket.at

DMA Blockchain Design

Deliverable number	*D5.2*
Dissemination level	*Public*
Delivery date	*30 June 2017*
Status	*Final (Version 1.0)*
Author(s)	*Ross King, Sven Schlarb*

The Data Market Austria Project has received funding from the programme "ICT of the Future" of the Austrian Research Promotion Agency (FFG) and the Austrian Ministry for Transport, Innovation and Technology (Project 855404)

Executive Summary

This Deliverable defines the architecture and design of the DMA Blockchain component in the context of the DMA technology foundation, based on the DMA proposal, the Community-driven "Data-Services Ecosystem Requirements (1st version)" (D2.2), and the "Data Technology Specification and Development Roadmap" (D5.1).

In the first section, we justify our selection of Ethereum as the basis for a private blockchain within the DMA ecosystem, and review some of the basic Ethereum concepts (Externally Owned Addresses, Ether/Gas, and Contracts).

In the second section, we identify specific components within the DMA technical foundation that must exist and interact with the Blockchain component.

In the next section, we review the DMA Blockchain data model and its relationship to the more general metadata models for Users, Data Assets, and Services. We also present an overview of the DMA Blockchain architecture.

In the fourth section, we present class definitions (describing attributes and methods) for the so-called "State Contracts" - Membership, Data Asset, and Service Contracts - that provide a guide for the implementation of said contracts using Solidity or another language supported by the Ethereum Virtual Machine.

In the final section, we define how the Blockchain layer (the combination of the Blockchain API defined in D5.1 and the Blockchain component defined in this deliverable) interacts with other DMA components in order to implement selected DMA use cases.

Table of Contents

1 Introduction 5
2 Blockchain Technology 5
2.1 Private Ethereum Network 5
2.2 Ethereum Basics 6
2.2.1 Ethereum EOAs 6
2.2.2 Ethereum Contracts 6
2.2.3 Ethereum Ether (or Gas) 7
3 Blockchain Architecture 8
3.1 Ethereum Addresses 8
3.2 Membership Management 9
3.3 Identity Management 9
3.4 Secret Management 10
3.5 Smart Contracts 10
4 Data Model 11
4.1 DMA Entities 11
4.2 Relation to DMA Metadata 12
4.3 Architecture Overview 13
5 State Contracts 14
5.1 Data Asset Contract 14
5.2 Service Contract 16
5.3 Membership Contract 17
6 Use Cases 18
6.1 User Creation 18
6.2 User Login 19
6.3 Create Data Asset 19
6.4 Version Data Asset 19
6.5 Fork Data Asset 20
6.6 Download Data Asset 20
6.7 Create Service 20

List of Abbreviations

API	Application Programming Interface
DMA	Data Market Austria
ECDSA	Elliptic Curve Digital Signature Algorithm - the algorithm employed by Ethereum to generate a public key associated with a given private key
EOA	Externally Owned Account - Addresses (which are also Ether accounts) linked to an agent (organization or user) in an Ethereum Blockchain
EVM	Ethereum Virtual Machine - a software interpreter that runs on each Ethereum node, executing the code defined in Ethereum Contracts.
GPL	Gnu Public License - a type of open source software license that requires any derivatives of the code to also be published under a GPL licence
Oauth	The OAuth 2.0 “open authentication” authorization framework
PID	Persistent Identifier
SLA	Service Level Agreement
UTXO	In Bitcoin, an Unspent Transaction Output

1 Introduction

In any scientific endeavor, the question of the provenance of research data should be foremost in the researcher's priorities. What is the source of the data? How has the data been altered over time? In Big Data terminology, this priority addresses the **veracity** of data. Not only is the question central to the rigorousness of scientific results, it is also becoming a central pillar of data protection regulations: even if personal data is public, it can only be used if its provenance is documented and understood.

From the point of view of archiving and long-term preservation, the **integrity** and authenticity of the data are also paramount, not only from the point of view of changes introduced intentionally over time, but also due to degradation of the storage media ("bit-rot") that can alter the meaning or readability of the Data Asset.

Regarding non-technical aspects of a data ecosystem, business models that rely on centralized infrastructures and administrations have inherent single-points of failure that endanger long-term **sustainability** of such endeavors.

As a result, a primary research goal of the DMA proposal was to investigate the application of Blockchain technologies to the questions of decentralized curation, preservation, provenance and security of Data Assets. Or in other words, how can Blockchain technology contribute to the veracity, integrity, and sustainability of a data and service ecosystem?

In this Deliverable, we define the architecture and design of the DMA Blockchain Security & Provenance component (or more succinctly, the **DMA Blockchain component**) in the context of the DMA technology foundation, based on the DMA proposal, the Community-driven "Data-Services Ecosystem Requirements (1st version)" (D2.2), and the "Data Technology Specification and Development Roadmap" (D5.1).

2 Blockchain Technology

2.1 Private Ethereum Network

In DMA Deliverable 5.1, we defined four principle criteria for the evaluation of Blockchain technologies:

1. Licence Type
2. Maturity
3. Community Support
4. Smart Contract Support

Based on these criteria, we have decided to base the DMA Blockchain implementation on the Ethereum[1] project. In terms of maturity, Ethereum is the second-oldest publicly deployed Blockchain technology, behind Bitcoin, and has a very large development community supporting it. Ethereum also provides much richer Smart Contract support than Bitcoin, and indeed was developed specifically with Smart Contracts in mind. Ethereum is available under the open source GPLv3[2] license, which means it cannot be bundled with other license types, either proprietary or

[1] https://www.ethereum.org/

[2] https://www.gnu.org/licenses/gpl-3.0.en.html

open (such as Apache V2 or the MIT license). GPL also requires that any changes to the codebase to be made available under the GPL license terms; however, we intend to use the Ethereum core “as is” without modification and to build our application logic on top of this.

The main reason for deploying a private instance of Ethereum, rather than participating in the global Ethereum network is that we wish to remain de-coupled from the economics of Ether from real-world currency evaluation. We do not wish to incentivize mining in DMA in this way; DMA mining should be carried out by those nodes that wish to invoke contracts (i.e. access data sets and services), rather than by those wishing to achieve monetary gain (through the collection of Ether). Furthermore, we wish to research the question of decentralized control of private network membership during the project, which would not be possible in the Ethereum public network.

Throughout the rest of this document, we will refer to the instantiation of the DMA project-specific private Ethereum network as the **DMA Blockchain**.

2.2 Ethereum Basics

The Ethereum blockchain can be alternately described as a blockchain with a built-in programming language, or as a consensus-based globally executed virtual machine. The part of the protocol that actually handles internal state and computation is referred to as the Ethereum Virtual Machine (EVM). The EVM is the engine in which transaction code gets executed, and is the core differentiating feature between Ethereum and other systems. From a practical standpoint, the EVM can be thought of as a large decentralized computer containing millions of objects (accounts) which have the ability to maintain an internal database, execute code and talk to each other.

There are two types of accounts:

1. Externally owned accounts (EOAs): these are accounts controlled by a private key, and if you own the private key associated with an EOA you have the ability to send Ether and messages from it.
2. Contracts: contracts are accounts that store their own code, and are controlled by code.

2.2.1 Ethereum EOAs

The long-established Blockchain-based virtual currency Bitcoin stores data about users' balances in a structure based on unspent transaction outputs (UTXOs): the entire state of the system consists of a set of "unspent outputs." A user's "balance" in the system is thus the total value of the set of unspent output transactions for which the user has a private key capable of producing a valid signature. In contrast, Ethereum implements a much simpler approach: the state stores a list of Externally Owned Accounts (EOAs) where each account has a balance, as well as Ethereum-specific data (code and internal storage), and a transaction is valid if the sending account has enough balance to pay for it, in which case the sending account is debited and the receiving account is credited with the value.

2.2.2 Ethereum Contracts

By default, the Ethereum execution environment is static; nothing happens and the state of every account remains the same. However, any user can trigger an action by initiating a transaction from an EOA, thus triggering a chain of transactions. If the destination of the transaction is another EOA, then the transaction may transfer some Ether but otherwise does nothing. However, if the destination is a contract, then the contract in turn activates, and automatically runs its code.

Ethereum provides an internal Turing-complete scripting language for the definition and development of smart contracts, which one can use to construct any contract or transaction type that can be formally defined. This code (representing an Ethereum Smart Contract) has the ability

to read/write to its own internal storage (a database mapping 32-byte keys to 32-byte values), read the storage of the received message, and send messages to other contracts, triggering their execution in turn. Once execution stops, and all sub-executions triggered by a message sent by a contract stop (this all happens in a deterministic and synchronous order, ie. a sub-call completes fully before the parent call goes any further), the execution environment halts once again, until woken by the next transaction.

2.2.3 Ethereum Ether (or Gas)

The Ethereum programming language is Turing-complete, and as a result transactions may be programmed to use bandwidth, storage and computation in arbitrary quantities. The latter in particular may end up being used in quantities that due to the halting problem cannot be reliably predicted ahead of time. Preventing denial-of-service attacks via infinite loops is an additional key objective of the Ether-based transaction model.

In order to address this problem, the Ethereum protocol includes the concept of **Ether**. Ether is a virtual currency used by clients in an Ethereum network to pay transaction fees to the machines in the network that execute the requested operations and contracts. The amount of Ether dedicated to a specific transaction is referred to as the **gas** for that transaction. As explained in the Ethereum design rationale here[1]:

"The basic mechanism behind transaction fees is as follows:

- Every transaction must specify a quantity of "gas" that it is willing to consume (called `startgas`), and the fee (in Ether) that it is willing to pay per unit gas (`gasprice`). At the start of execution, `startgas*gasprice` Ether are removed from the transaction sender's account.
- All operations during transaction execution, including database reads and writes, messages, and every computational step taken by the virtual machine consumes a certain quantity of gas.
- If a transaction execution processes fully, consuming less gas than its specified limit, say with `gas_rem` gas remaining, then the transaction executes normally, and at the end of the execution the transaction sender receives a refund of `gas_rem*gasprice` and the miner of the block receives a reward of `(startgas-gas_rem)*gasprice`.
- If a transaction "runs out of gas" mid-execution, then all execution reverts, but the transaction is nevertheless valid, and the only effect of the transaction is to transfer the entire sum `startgas*gasprice` to the miner.
- When a contract sends a message to the other contract, it also has the option to set a gas limit specifically on the sub-execution arising out of that message. If the sub-execution runs out of gas, then the sub-execution is reverted, but the gas is nevertheless consumed."

The necessity of gas is a complication associated with the Ethereum implementation, but it is a necessity given the expressibility of the smart contract language, which is in turn a requirement for the complex types of contracts envisioned by the DMA project. Our goal is to hide this complexity from the DMA end user to whatever extent possible.

[1] https://github.com/ethereum/wiki/wiki/Design-Rationale

3 Blockchain Architecture

3.1 Ethereum Addresses

An Ethereum address represents an "externally owned account," or EOA. For a given EOA, the address is derived from the public key associated with the private key that controls the account. As a string, an example account looks like 'cd2a3d9f938e13cd947ec05abc7fe734df8dd826'. This is a hexadecimal format (base 16 notation), which is often indicated explicitly by prepending 0x to the address. Even though users commonly refer to this address as the "public key," this is actually not the case. The public key is generated in an intermediate step that the user never sees.

EOAs are generated in three steps:

1. Generate a Private Key

The private key consists of 256 random bits, which can in turn be represented by a string of 64 randomly generated hexadecimal characters. Any algorithm that generates a new private key should be seeded with a proper random number generator.

2. Calculate the Public Key from the Private Key

Apply the Elliptic Curve Digital Signature Algorithm (ECDSA) to the private key in order to calculate the public key that is 64 bytes (512 bits) long.

3. Create the Address from the Public key

Apply the Keccak-256[1] hash algorithm to the 64-byte public key, which results in a string with 32 bytes. (Note: SHA3-256[2] is now an established standard, but Ethereum presently uses the slightly outdated, less standard Keccak-256 algorithm.)

The final 20 bytes (40 characters) of this (Keccak-256) hash (dropping the first 12 bytes) is the public address, or the EOA. When prefixed with the hexadecimal indicator "0x" the address becomes 42 characters long.

Note that a typical Ethereum user never directly generates the private key! The three steps above are accomplished using Ethereum command line tools (e.g., `eth` or `geth`):

```
$ geth account new
Your new account is locked with a password. Please give a password. Do not
forget this password.
Passphrase:
Repeat Passphrase:
Address: {168bc315a2ee09042d83d7c5811b533620531f67}
```

This tool automatically creates a random private key in the background, which is then stored on the local Ethereum node, encrypted by the user's Passphrase. This fact has implications for the DMA design, because it means that a user can only authenticate her identity (and thereby access the private key necessary to initiate transactions) on the Ethereum node where that identity was established.

Note that the only metadata associated with an EOA is the balance of Ether attributed to that account. As a result, it is not feasible to represent more complex objects, such as data assets or services, as simple EOAs in the DMA Blockchain. Instead, these entities must be represented by

[1] http://keccak.noekeon.org/

[2] https://en.wikipedia.org/wiki/SHA-3

Contracts. This point is elaborated in the Sections "Smart Contracts" and "State Contracts" below. Ethereum Contracts are like classes in object-oriented programming: they store attributes (state) and offer methods (functions).

3.2 Membership Management

In his description of the Stellar consensus protocol[1], David Mazières notes that the drawback of private systems is that the organization or organizations that control the membership (by defining the "starter list") effectively have more power within the network than other members. One of the research questions that DMA should address with respect to the Blockchain is therefore, how can one manage the membership in a private Blockchain network in a decentralized manner?

Our approach will be to deploy a special contract, the DMA Membership Contract, which maintains a list of all member organizations (or more specifically, a list of the EOAs of the member organizations). This contract also implements methods that allow proposing new members (allowed by any existing member), and also adding new members to the membership list, based on a majority (or if so desired, unanimous) vote by all existing members. DMA Member organisations are then empowered to add Actors that act on their behalf (refer to the "DMA Entities" Section below).

This contract could also maintain some special metadata about member organizations; namely, the IP addresses of the server nodes that should participate in the DMA Blockchain network. These nodes could then be synchronized with the Bootstrap Node - a special node in the Blockchain network that allows peers to find other peers.

3.3 Identity Management

We also require a method by which Actors in the DMA metadata store (Organizations, Employees) are linked to and represented by Externally Owned Accounts (EOAs) in the DMA Blockchain. We suggest the address generated when creating an EOA could be used as a unique identifier (PID) in the DMA Metadata Management component. In other words, these identifiers would have the form:

```
{member-organization-namespace-string}:{data-asset-id}
```

For example, with the string "`ait`" representing the node of the member organization AIT Austrian Institute of Technology GmbH and "`168bc315a2ee09042d83d7c5811b533620531f67`" as the hash string, the PID would be as follows:

```
ait/168bc315a2ee09042d83d7c5811b533620531f67
```

Or, as a PID URI:

```
info:dma/ait/168bc315a2ee09042d83d7c5811b533620531f67
```

In this manner, the address is a bridge that allows the system to identify any Blockchain agent based on a PID or find the metadata record of any actor given the Blockchain address. This does however require that an actor is created and registered simultaneously in the DMA Metadata Management component and in the DMA Blockchain component; that is, the same application logic must interact with both of these components in a single process.

[1] https://www.stellar.org/papers/stellar-consensus-protocol.pdf

3.4 Secret Management

When a new user is created, this user will presumably also define a password that unlocks the user account on the DMA application layer. At the same time, a Passphrase must also be generated that unlocks the private key associated with the EOA.

One possibility would be that the same password unlocks both the user account and the private key. This would be the simplest approach and could be recommended for the year one implementation roadmap.

However, it may be both desirable and necessary that DMA entities have multiple private keys or other credentials, in which case it will be useful to add a secrets management application to the DMA application stack, such as the OpenStack Barbican[1] component.

This will in particular be necessary in for those (non-open) Data Assets that are stored as encrypted containers in the DMA network. In this case, each Data Asset will require an encryption key, which can be allocated to a user in the case that the conditions of the License Contract are fulfilled. Such keys should in turn be managed through a Secret Management component.

Given the necessity of a Secret Management component, it then becomes possible to separate a user's DMA access password from the user's DMA Blockchain passphrase. The passphrase could be stored in the Secret Management component, unlocked by a token that in turn is authorized by the user's password. But this additional complication, if required at all, could be implemented as part of the year two roadmap.

3.5 Smart Contracts

Data Assets and Services are additional entities that must be modelled in the DMA ecosystem. But from the point of view of the DMA Blockchains, these are not agents. A Data Asset does not initiate a transaction in the Blockchain; rather, an agent (User or Organization) initiates a transaction related to a Data Asset, e.g., access the Data Asset, fork the Data Asset, enrich the Data Asset. Similarly, agents may stage a Service (apply to make the service available on a chose computational cluster), access a Service, or apply a Service to a Data Asset. From this point of view it should be clear that Data Assets and Services cannot be modelled as EOAs, but rather as Contracts.

As Contracts, these entities can maintain stateful information in the DMA Blockchain, and this information can be modified over time through Blockchain transactions. As transactions are also permanently and irrevocably preserved in the Blockchain, this provides a perfect provenance history for the entities in question.

Furthermore, Ethereum Contracts can invoke other Contracts. This means it is both possible and desirable to model other contractual entities in particular the License Contracts that govern the terms of service (SLAs) and terms of use (costs, conditions. etc.)l. By separately modelling the License Contracts, the same type of model contract[2] (e.g., "access this entity by paying money to this actor") can be referenced by multiple entities (Data Assets). This approach also creates a clear separation of interest between DMA infrastructure developers, who would develop the State Contracts, and DMA legal experts, who would develop the License Contracts in the context of WP3.

[1] https://wiki.openstack.org/wiki/Barbican

[2] http://solidity.readthedocs.io/en/develop/contracts.html%23inheritance

4 Data Model

4.1 DMA Entities

The hierarchy of DMA entities to be modelled in the context of the DMA Blockchain is illustrated in Figure 1 on page 12.

Entities have two main classes: Agents and Contracts

Agents are those entities that have agency in the Blockchain, that is, those entities that can initiate transactions. In the Blockchain world, there are only Agents, and these Agents are represented by Externally Owned Accounts (EOAs).

In the world of the DMA metadata model, there are two types of Agents: Organizations and Actors. Organisations are those legal entities that can become members of the DMA Ecosystem, and Actors are those entities that act on behalf of organisations (e.g., creators, employees, contractors). Therefore a 1:N relationship between Organisations and Actors should be modeled there; this relationship will however not be apparent in the DMA Blockchain and Blockchain transactions can be initiated by either type of Agent.

Contracts represent either those DMA entities whose state (and historical provenance of that state) should be stored in the DMA Blockchain (the “State” contract type), or the conditions under which those entities may be accessed (the “License Contract” type). State Contracts represent DMA Data Assets or DMA Services. Data Assets refer specifically to static data sources, as opposed to dynamic data sets or streaming data. It is assumed that dynamic and streaming data will be offered as a service, hence the sub-class “Data Services,” which may or may not be necessary in terms of implementation, but is useful in order to clarify the difference between static and dynamic data asset contracts.

A static Data Asset (represented by a Data Asset Contract) has potential features that cannot be applied to dynamic data assets (as previously addressed in Deliverable D5.1); for example, a static Data Asset may have an associated hash value (used to verify the integrity of the content), it may be encrypted (in order to have a higher level of security), and it may be forked (that is, one could create a new Data Asset that is clearly derived from a pre-existing Data Asset).

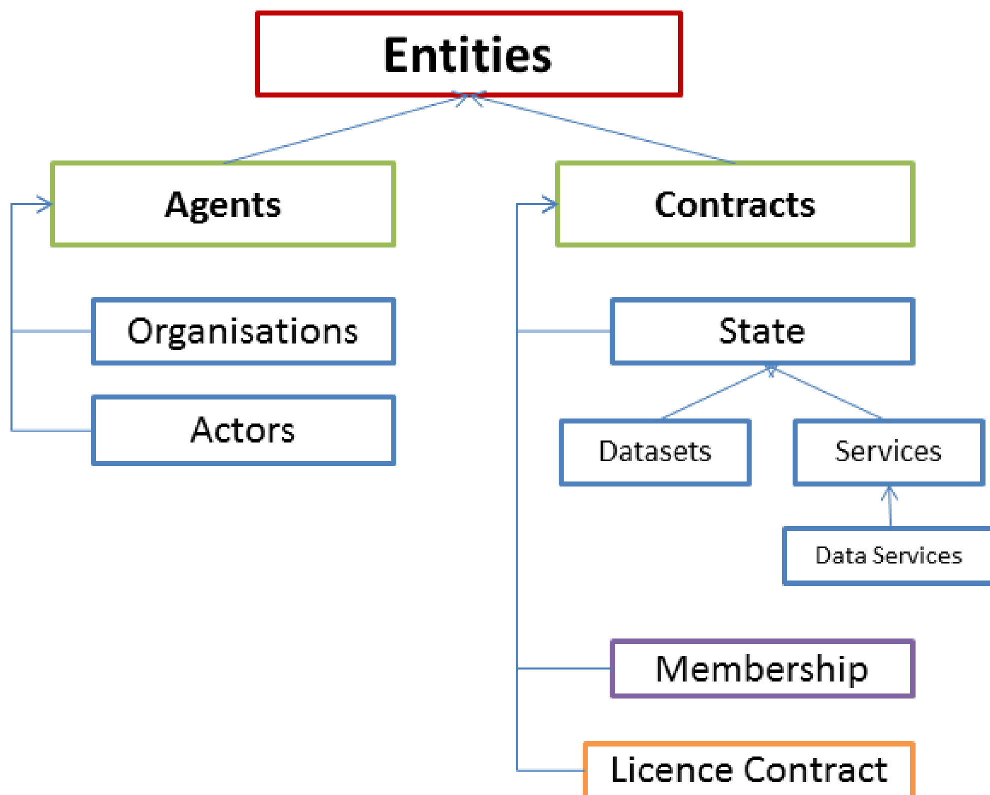

Figure 1: DMA Blockchain Entities

None of these concepts can be applied to dynamic Data Assets, nor indeed to Services in general. A Service on the other hand may have other attributes, such as the "Developer" (an Actor that is responsible for the software associated with the Service), whereas Data Assets would have "Curators" instead.

The State-based Contracts for Data Assets and Services will be developed in the context of DMA WP5 (Task 5.2).

License Contracts (in German *Lizenzverträge*) are traditional Smart Contracts that define the obligations of the Agent that offers a Data Asset or Service and the obligations of the Agent that wishes to make use of said Data Asset or Service (such as agreement to license terms, payment for use, etc.). License Contracts should be developed in the context of DMA WP3 (Task 3.3).

4.2 Relation to DMA Metadata

The design of the DMA Blockchain data model requires a synchronization of key attributes with the DMA metadata storage component. As noted above ("Identity Management"), sharing identifying entities in both systems through their unique EOA is the backbone of this approach. The concept is elaborated in Figure 2, which illustrates how these entity IDs are references across metadata records, agents, and Contracts.

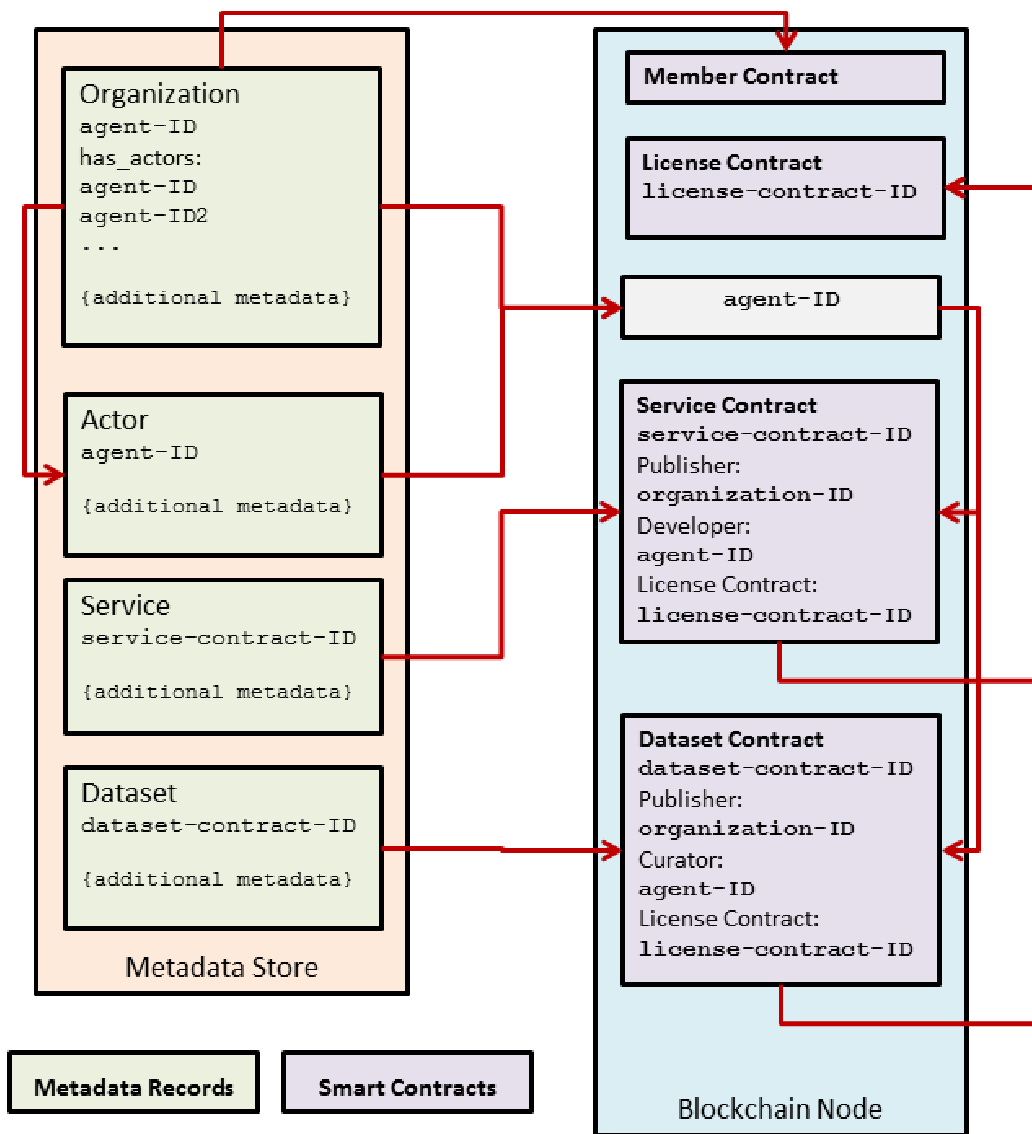

Figure 2: Relationships between the DMA Metadata Store and the DMA Blockchain

4.3 Architecture Overview

An overview of the architectural concepts described in this section is shown in Figure 3. Note that this view is a mixture of physical architecture (depicting actual users and server nodes) as well as logical architecture (depicting Contracts and their relationships). The nexus at the center of the diagram represents the DMA Blockchain itself, which is of course replicated on all physical nodes.

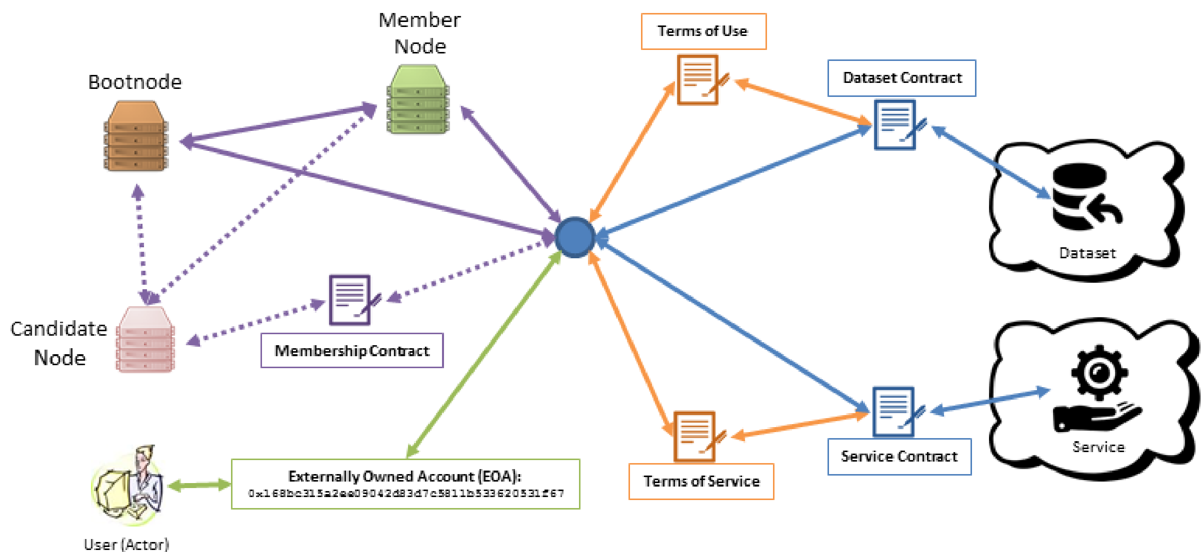

Figure 3: Overview of DMA Blockchain Architecture

5 State Contracts

This section is meant to specify the attributes and functionality of the DMA State Contracts (Membership, Data Asset and Service Contract). As part of Task 5.2, this functionality will be implemented in one of the Ethereum-supported Smart Contract languages, for example Solidity.

5.1 Data Asset Contract

This section specifies the attributes and functionality of the DMA Data Asset Contract.

Attributes
creator: EOA of the agent that owns the Data Asset
publisher: EOA of the agent that publishes the Data Asset
curator: EOA of the agent that curates the Data Asset
version: the current version of Data Asset
hash: the map of hash values calculated from the datasets included in the current version of the Data Asset
uri: the unique identifier of the Data Asset
predecessor: the Address of the Data Asset from which this Data Asset was

forked (empty by default)
license: the Address of the License Contract associated with this Data Asset
Methods
change_creator (new_creator): changes the creator of the Data Asset to the agent designated by the EOA **new_creator**. Can only be invoked by the present creator of the Data Asset. Returns: **true** (if successful) or **false** (if failed)
change_publisher (new_publisher): changes the publisher of the Data Asset to the agent designated by the EOA **new_publisher**. Can only be invoked by the present creator of the Data Asset. Returns: **true** (if successful) or **false** (if failed)
change_curator (new_curator): changes the curator of the Data Asset to the agent designated by the EOA **new_curator**. Can only be invoked by the present creator or the present publisher of the Data Asset. Returns: **true** (if successful) or **false** (if failed)
update_dataasset (uri): replaces the current Data Asset with the Data Asset identified by the **uri**. The attributes uri, version, and hash are updated. Can only be invoked by the present creator, publisher, or curator of the Data Asset. Returns: map of **hash** values of datasets contained in updated Data Asset
fork_dataasset (uri): creates a new Data Asset Contract with the Data Asset identified by the **uri**. The attribute uri is set to **uri**, the attribute **predecessor** is set to the Address of the parent Data Asset. The attribute **version** is set to 1.0, and the **hash** value is calculated based on the bitstream available at **uri**. Can only be invoked by the present creator, publisher, or curator of the Data Asset. Returns: **Address** of new Data Asset Contract
download_dataasset(): Can be called by any Agent in the system. Invokes the **license** Contract for the Data Asset on behalf of said Agent. Returns: **true** if **license** Contract conditions for download are fulfilled, **false** if not.
stage_dataasset(): Can be called by any Agent in the system. Invokes the **license** Contract for the Data Asset on behalf of said Agent. Returns: **true** if **license** Contract conditions for staging are fulfilled, **false** if not.

5.2 Service Contract

This section specifies the attributes and functionality of the DMA Service Contract.

Attributes
creator: EOA of the agent that owns the Service
publisher: EOA of the agent that publishes the Service
developer: EOA of the agent that developed the Service
version: the current version of Service
uri: the locator of the Service
license: the Address of the License Contract (or SLA) associated with this Service
license: the Address of the License Contract associated with this Service
Methods
change_creator (new_creator): changes the creator of the Service to the agent designated by the EOA **new_creator**. Can only be invoked by the present creator of the Service. Returns: **true** (if successful) or **false** (if failed)
change_publisher (new_publisher): changes the publisher of the Service to the agent designated by the EOA **new_publisher**. Can only be invoked by the present creator of the Service. Returns: **true** (if successful) or **false** (if failed)
change_developer (new_developer): changes the Developer of the Service to the agent designated by the EOA **new_developer**. Can only be invoked by the present creator or the present publisher of the Service. Returns: **true** (if successful) or **false** (if failed)
update_service (uri): replaces the current Service with the Service specified by the **uri**. The attributes uri and version are updated. Can only be invoked by the present creator, publisher, or developer of the Service. Returns: **version** of updated Data Asset

invoke_service (): Can be called by any Agent in the system. Invokes the **license** Contract for the Service for invocation on behalf of said Agent. Returns: **true** if **license** Contract conditions for invocation are fulfilled, **false** if not.
stage_service (): Can be called by any Agent in the system. Invokes the **license** Contract for the Service for staging on behalf of said Agent. Returns: **true** if **license** Contract conditions for staging are fulfilled, **false** if not.
invoke_license (): Can be called by any Agent in the system. Invokes the **license** Contract for the Service on behalf of said Agent. Returns: **true** if **license** Contract conditions are fulfilled, **false** if not.

5.3 Membership Contract

This section specifies the attributes and functionality of the DMA Membership Contract.

Attributes
members: array of EOAs of each DMA member organisation
votes: map of
Methods
propose_member (candidate_eoa): Creates a new entry in the **votes** map keyed to the **candidate_eoa**. Can be invoked by any DMA member, but only once for any given candidate. Returns: **true** (if successful) or **false** (if failed)
vote (candidate_eoa, approve): Registers the value of **approve** (true or false) for a given **candidate_eoa** and the member organization executing the function in the **vote** map. Can be invoked by any DMA member, but only once for any given candidate. Returns: **true** (if successful) or **false** (if failed)
add_member (candidate_eoa): Checks the status of the **votes** map for a given **candidate_eoa**. If a majority of the existing members have voted in favor of the new candidate, the **candidate_eoa** is added to the members array and the vote is removed from the **votes** map. If a majority of the existing members have voted against the new candidate, the vote is

```
removed from the votes map (and the proposed membership is rejected). If
member votes are still pending, the function returns false but no other
action is taken. Can be invoked by any DMA member.

Returns: true (if successful) or false (if failed)
```

6 Use Cases

Use cases cannot be implemented through the DMA Blockchain component alone; they must be executed in coordination between the layers indicated in Figure 4 below:

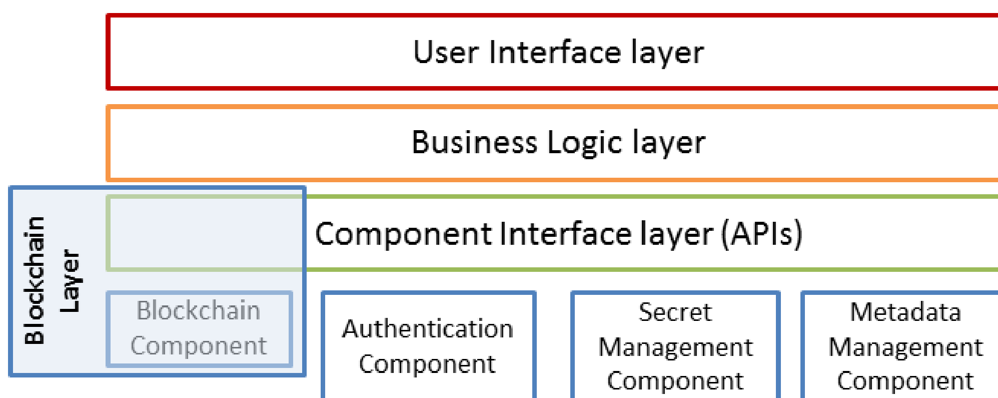

Figure 4: DMA Application Layers

The User Interface layer represents the (presumably) web-based user interface with which an end user directly interacts. The Business Logic Layer is the programmatic layer that implements the use cases and business processes required by the Ecosystem. The Component Interface layer represents the (presumably) RESTful APIs of the various DMA components, including Authentication, Secrets, and Metadata Management components, as well as the Blockchain API.

The Blockchain Component itself represents the Ethereum web.js endpoint, which is called by the Blockchain API and operates directly on the EOAs and Contracts as described in the previous sections. In the use cases discussed below, the combination of the Blockchain Component and the Blockchain API will be referred to as the **Blockchain layer**. Functions in the Blockchain layer should never be directly invoked by an end user, but only through the Business Logic layer.

The use cases discussed in the following subsections are examples that have been extracted from the DMA WP4 User Stories[1] that are relevant for the DMA Blockchain component. These descriptions are not intended to be comprehensive, but rather to convey how the different layers will have to interoperate within the system.

6.1 User Creation

User Interface layer: The user starts at a DMA Web Portal instance (either the central node or a locally deployed node, if such exists). The user first invokes a “New User” function, the successful return of which triggers the request for additional user metadata.

[1] https://docs.google.com/document/d/1m1dPHYlPimeRrIEzHQjEVj_Iz6DpyTA_zvuPlHcZz2k/edit

Business Logic layer: This layer must either retrieve a passphrase from the user or generate a random passphrase. This passphrase may or may not be stored in the DMA Secret Management Component. This layer first calls the Blockchain layer, in order to establish a new EOA and to retrieve the address of this EOA, which becomes the PID for the user in the DMA Metadata Management component as well as for the user account in the DMA Authentication component. It then calls the Metadata Management component in order to store additional information about the user.

Blockchain layer: We must implement an API method that allows the creation of new EOAs. At present we are considering making use of the Web3.py[1] library.

6.2 User Login

User Interface layer: The user submits her authentication credentials through a web interface

Business Logic layer: This layer authenticates the user against the DMA Authentication component and returns an OAuth token. At the same time, it uses the passphrase (or retrieves the passphrase from the Secret Management component with the user credentials) and authenticates the user's EOA on the Blockchain node.

Blockchain layer: This layer makes use of the Web3.py library to authenticate the user account on the Blockchain node in question.

6.3 Create Data Asset

User Interface layer: The user starts at a DMA Web Portal instance (either the central node or a locally deployed node, if such exists). The user first invokes a "New Data Asset" function, which includes a form for all required (and optional) Data Asset metadata. This must include references to any related License Contract addresses.

Business Logic layer: Based on the submitted metadata, this layer first calls the Blockchain layer in order to initialize the Data Asset Contract. With the returned address of the new Contract, the layer then triggers the Ingest component, which also calls the Metadata Management component in order to store and index the discovery metadata for the Data Asset. This layer may also need to generate a random encryption key (to be stored in the Secret Management component) should the Data Asset be stored encrypted.

Blockchain layer: This layer receives a request and a set of metadata necessary to instantiate a new instance of a Data Asset Contract, and returns the address of the new Contract.

6.4 Version Data Asset

User Interface layer: An authenticated user selects a Data Asset for which she is either the creator, Publisher, or Curator. She then modifies the metadata record for the Data Asset and submits.

Business Logic layer: This layer updates the Data Asset metadata record through the Metadata Management component, and invokes the update method of the Blockchain layer API.

Blockchain layer: This layer invokes the `update_dataasset` method of the Data Asset Contract.

[1] http://web3py.readthedocs.io/en/latest/

6.5 Fork Data Asset

User Interface layer: An authenticated user selects a Data Asset for which she would like to fork. She creates a new metadata record to go along with the new forked Data Asset and submits.

Business Logic layer: This layer first invokes the fork method of the Blockchain layer API; if this method returns "true" then it creates a new Data Asset metadata record through the Metadata Management component.

Blockchain layer: This layer invokes the `fork_dataasset` method of the Data Asset Contract.

6.6 Download Data Asset

User Interface layer: The already-authenticated user invokes the search functionality of the DMA Portal, which in turn submits queries to the Metadata Management component's indexes, in order to find a specific Data Asset. The user then clicks the download option for the data asset.

Business Logic layer: This layer invokes the selected Data Asset's License Contract. If the terms of use are fulfilled (for example, if the Data Asset has been declared as "open", this Contract will always be fulfilled for any authenticated user in the system), the Blockchain layer returns "true" and the download can proceed. This may or may not involved retrieving the Data Asset's encryption key in order to provide an unencrypted bytestream to the user.

Blockchain layer: This layer first invokes the `download_dataasset` method of the associated Data Asset Contract with the user as the Agent in question. This in turn invokes the License Contract specified by the Data Asset Contract. If the terms of use specified in the contract are fulfilled, the Blockchain layer returns "true"; otherwise it returns "false."

6.7 Create Service

User Interface layer: The user starts at a DMA Web Portal instance (either the central node or a locally deployed node, if such exists). The user first invokes a "New Service" function, which includes a form for all required (and optional) Service metadata. This must include references to any related License Contract addresses.

Business Logic layer: Based on the submitted metadata, this layer first calls the Blockchain layer in order to initialize the Service Contract. With the returned address of the new Contract, the layer then triggers the Metadata Management component in order to store and index the discovery metadata for the Data Asset.

Blockchain layer: This layer receives a request and a set of metadata necessary to instantiate a new instance of a Service Contract, and returns the address of the new Contract.