David Steiman, 338864

Design, Development, and Evaluation of a blockchain-based Decentralized Access Control Protocol for Microservices

Bachelor thesis expose

Version 5

I. Context

In a microservice / federated architecture, there are different demands on security. The focus of today's microservice sources is mostly for the question: how to ensure that at request time security policies can be decided. In details, the topics cover authentication (e.g., OpenID, IAM, SAML) to identify the requesting subject and authorization (e.g., OAuth 2, JWT) to gather all information about the authorization of the subject. The next question is about, how to make a decision. For this purpose, there are different approaches to access control enforcement, like role-based access control (RBAC) or attribute-based access control (ABAC). These concepts are simple, but lack on expressibility in complex access control requirements. For fine-grained access control, a well-established way is using "access control lists" (ACL) for a detailed definition of policies, which often occur in monolithic applications. While tools like JWT make it unnecessary to store a state of a session, ACLs have to be persisted in a way accessible to all microservices. Someone also has to manage the ACL configuration. In particular, in federated systems, where services belong to a certain party, every member has to trust this policy administrator. Consequently, the core idea of ACLs lacks in two points: decentralization and trust.

This concept relies on the model of domain-based distributed systems. It describes a domain as "the realm of knowledge [..] of the software system being designed" [1] which "can be used to group objects in a hierarchical structure, to apply a common security policy [..] of large distributed systems." [2] Based on the concept of Yialelis, which is close to a system this thesis aims for, access control is designed with ACLs. An entry of the list contains the requesting subject, requested target and access rules defined for those. A reference monitor enforces policies. Although his idea has a focus on distributed systems, the concept assumes the presence of a policy administrator, which is performing the overall security configuration. So this access control mechanism requires trust in a central authority.

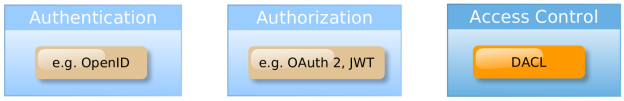


Figure 1. DACL for access control in microservices.

This thesis introduces a "decentralized access control list" (DACL) protocol. Assuming the presence of an identity management and authentication, DACL describes how to enforce authorization policies, being fully decentral and eliminating the need for a central authority (trustlessness).

In favor of decentralization, DACL does not depend on a central storage, by moving the responsibility for storing policies to IPFS. In this case, the policies are retrievable as long at least one node is healthy. For trustlessness, DACL supposes all state changing operations are coordinated by a smart contract, deployed on the Ethereum blockchain.

In favor of decentralized governance of microservices, the protocol itself does not cover technical details, to leave the implementation open to the service. The mechanism will be applied to an exemplary scenario to show, how these protocols should be used and implemented.

II. Example scenario

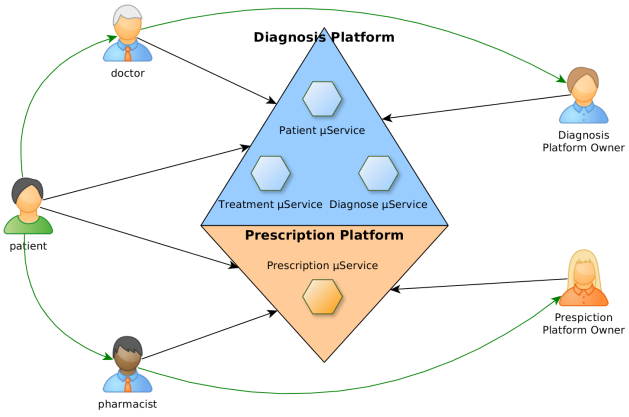


Figure 2. architecture of an eHealth Platform.

The scenario describes a federated eHealth application platform for diagnoses and prescriptions. During treatment, a doctor declares several diagnoses and prescribes prescriptions for his patients. In this scenario, two parties are maintaining the

eHealth platform, which consists of a diagnosis and a prescription platform. As medical data of a patient is highly sensible data, the access control policies are complex. On the one side, the patient should always be able to supervise the permission on his data, but allow exceptions for his doctor, e.g., to open access to diagnoses to collaborating doctors. On the other side, both doctors and pharmacists might become patients as well, so approaches like RBAC do not fit well. The green arrows in figure 2 show the permission grant direction, while the black show the common requests done by the different users. As data relating to a user should only be under the user's control, maintaining a central policy authority is not acceptable. Using centralized ACL mechanism also means, one node (or a centrally managed group of nodes) is responsible for storing the policies and allows this central authority to bypass the policies and manipulate the data.

The thesis claims to show, how DACL helps to construct a decentralized access control mechanism, to ensure patients keep control over their data by implicitly changing the security policies of the eHealth platform.

III. Problems

A. Trustlessness

The first core problem of DACL is trustlessness. Trustlessness means, no one has special privileges over the state of the policies. In other words: there is no need to trust someone. So the subsequent problems here are:

1) Network members and proposal handling: The DACL smart contract allows a set of participants (Ethereum wallets) to be part of a network, where every member is permitted to interact with the smart contract. As there is no network administrator, the protocol must define how new members can join or how to remove malicious participants.

As there is no central policy authority as well, which can perform changes on its behalf, the protocol must define a proposal handling for changes. The basic idea says every participant can read the policies, propose changes and vote on them. By default, all participants are forced to vote on any proposal. However, there are use-cases, where a participant is not required to participate in the voting process for some reason. The protocol must define the proper conditions when change proposals are accepted.

In the case when multiple policy change proposals are pending the network must end up in a state, where all valid policy change proposals are checked in.

2) Access policy enforcement: Comparing to the reference monitor from Yialelis' domain security model, the DACL protocol should define, how policies are enforced. Assuming technical requirements violates the microservice approach of decentralized governance the protocol should specify, how the policies are enforced. Optionally it could be discussed, how other members can prove the policy enforcement.

3) After-the-fact mandate changes: Although only participants known by the smart contract are permitted to interact with the policies, there still are business use-cases to make exceptions on these mandates. To cover similar exceptions on a "consortium blockchain", the Hyperledger requirements work group released a set of 8 requirements [4], where the following are relevant to the DACL design:

- a member(anchor) shall be able to sign over its authorization to another member (guardian) after the fact. This requirement accords to the following two scenarios:
- anchor loses its authorization
- anchor maintains its authorization over the smart contract
- the network shall be able (e.g., via a blockchain consortium) to sign over the authorization over a smart contract from anchor to another member
- it should be possible to re-assign the authorization
- all assignments shall not require handling over private keys

As the DACL does not define a consortium, the other requirements do not fit well here.

4) Policy storage: Agnostic to the storage design, the owner of a node should not be able to change the data. Manipulating the value of the stored data should make the data invalid.

B. Decentralization

The second problem of DACL is reducing the responsibility of a single participant and coordinate all tasks around communication and persistence completely distributed. In particular:

1) Costs of operation: As the core component of DACL is an Ethereum smart contract, every writing operation costs some fees in Ether (the Ethereum cryptocurrency) to be paid by a participant. Also, the complexity of the contracts final implementation is relevant for the running costs, as these depend on the number of instructions processed in Ethereum. When a participant runs out of gas, he becomes unable to communicate with the network, so the contract must regulate this costs with methods like auto-refilling and define how the network responds to frozen participants. For example, implementing a health check, where every participant writes the current timestamp on the blockchain can prevent this block on proposals, but increase the running costs.

Putting data on the blockchain is expensive, so it must be clarified, which part of the data is “on-chain“ and which “off-chain“. In particular, how to refer to the off-chain-stored data.

2) Performance: The performance of DACL can be narrowed down to the response time for a request, which involves an access decision, and the duration for new proposals getting committed and propagated to all participants. The entire state can be stored in a single document, containing the full policy configuration. Consequently, a proposal consists of a whole state, too. Handling these policies unstructured may heavily affect the time a participant needs to iterate over the proposal. In a different approach a proposal consists only of a set of changes, so the current state is the result of all accepted changed applied after each other. Without caching, accumulating the changes may cause many requests, what affects the response time.

3) Policy storage: No node should form a single point of failure. As of the CAP theorem, there is no way to guarantee consistency, availability, and partition. An access control decision must be possible at any time. There should be a tradeoff between consistency, (ensure new policies are always available right after their commit), or partition (avoid storing a complete policy configuration state on every node).

4) Horizontal scalability of the microservices: Microservices are often designed according to the twelve-factor app specification [3]. Processes and Disposability are the key factors for horizontal scalability, as microservices run replicated and might be terminated at any time. Considering DACL is causing a time-intensive overhead for building a valid state, this would harm these two factors and consequently impact the horizontal scaling abilities of DACL secured microservices. A vivid example would be if every microservice must first synchronize with the entire Ethereum blockchain from scratch, what would take hours for a new replica to be running.

IV. Proposed solution approach

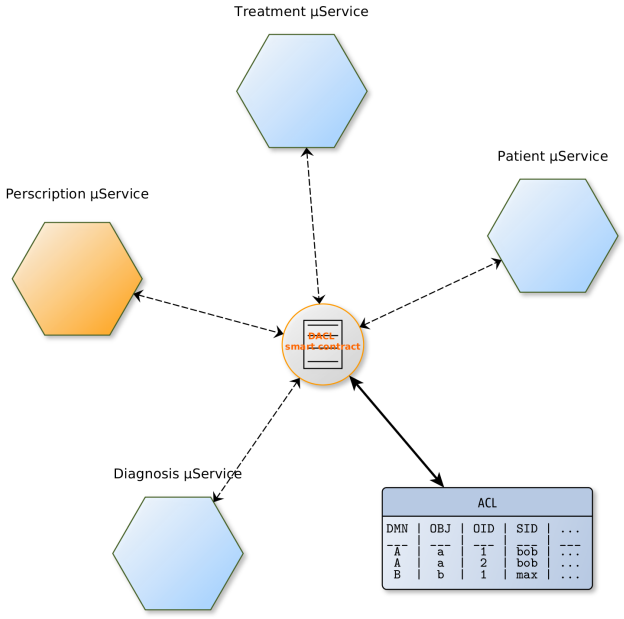


Figure 3. Basic DACL architecture.

The proposed distributed access control list protocol (DACL) claims to give a solution to the problems. At the core, all members of the DACL defined network interact with a single smart contract, which provides the actual state of the ACL and coordinates changes on this state.

A. the DACL protocols

DACL is designed using the following sub-protocols:

1) access decision protocol: The protocol first defines an access control entity $ace \in \mathbb{ACL}$, which is in an access control list $\mathbb{ACL}$, defined by: $\mathbb{ACL} = \{(domain, oid, uid, sid, permission, order) : domain, oid, uid, sid \in \Sigma^*, permission \in \{0,1\}^4, order \in \mathbb{N}\}$ with:

$domain$ = the domain that owns the object

oid = the object identifier of target object

uid = the objects unique identifier

sid = the unique identifier of the requesting subject

$permission$ = 4 bits for create, read, update and delete permission

$order$ = the order of this entry in a set of matched rows

This infinite discrete set can be represented as the following table:

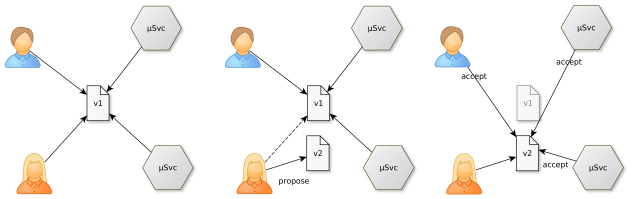
domain	OID	UID	SID	permission	order
papers	paper	123	Alan	0-1-1-1	1
papers	paper	123	@any	0-1-0-0	2
papers	paper	@any	@any	1-0-0-0	3

Every microservice acts as a participant for a domain (by owning a private key known to the smart contract).

To evaluate this ACL for a given object and subject, a participant iterates over all access control entries for its matching domain in the defined order. The first match is providing the decision if access is granted. A participant, is free to use any convention for identifying a single principal, a group of users or a role.

2) policy coordination protocol: The policy coordination protocol describes how exactly parties can change the ACL. Given the fact, all communication is done via the DACL smart contract, the following figure describes how a successful reconciliation should proceed

When a participant wants to change policy rules, he defines a new version of the ACL and reports a proposal to the network. Each participant can either accept the new version, deny with a user defined reason or abstain from giving an answer. If no of the participants denies the proposal, the smart contract provides the new ACL list.



(a) initial state (b) state change proposal (c) accepting new policy

Figure 4. test

As an alternative, participants could send deltas instead of full versions of the ACL.

B. implementation and on-chain/off-chain design

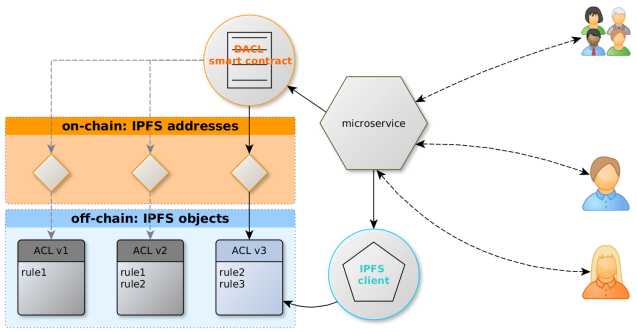


Figure 5. DACL architecture

After defining that protocol formally, a reference prototype must be implemented using a common language for building microservices. The example will be implemented using NodeJS and Solidity (for developing smart contracts).

To keep as little as possible data on the blockchain, the smart contract will store IPFS addresses of objects. In the most basic scenario every microservice implements the policy coordination protocol directly when interacting with IPFS and ethereum.

In order to standardize the coordination, a DACL client can be developed that runs parallel to each microservice and makes the protocol processes available via a well-defined API. It interacts with the smart contract, reads the address from the blockchain and opens the content using an IPFS client.

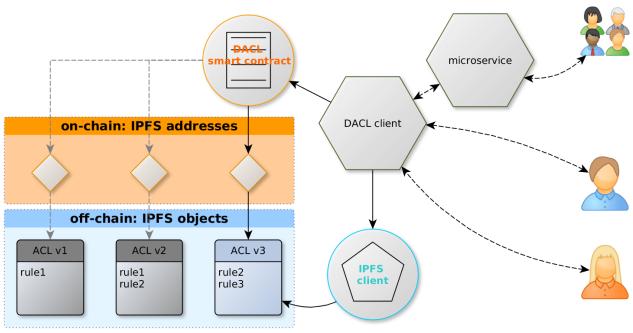


Figure 6. DACL architecture with client

V. Evaluation

A. comparing results with problems

In the first chapters of my thesis, I figure out important properties/objectives DACL claims to satisfy. After implementing the example platform, it can be compared to these objectives. For example the after-the-fact change, where there is a set of requirements to check.

B. attack vectors

The prototype will consist of various design decisions. Like the question, how to act in a situation, where no clear policy is present (e.g. for e newly created entity and a still pending policy). Furthermore a single participant might block any further changes to the ACL by always denying any new proposals. As there is a currency circulating, which acts as fuel for the systems, it should be checked what happens, when DACL run out of fuel, what the consequences are and how this can be prevented. From the point of view of a single malicious participant, there different ways of breaking the intended behavior, which will be tried out and summarized.

VI. Timeline

- formally design the DACL protocol
- (possibly) starting implementation of DACL client
- Building a prototype of a microservice based application, implementing the protocol and using the DACL policies
- Evaluating the prototype as a proof-of-concept, referring to defined claims and new problems, found during the modeling and/or implementation process

Related Work

- after-the-fact mandate changes (Hyperledger requirement WG)
- domain-based security in distributed systems (Yialelis)
- ethereum whitepaper (Buterin)
-

More papers and references are pending.

References

[1] E. Evans, *Domain Driven Design*, 2002.
[2] N. Yialelis, *Domain-Based Security forDistributed Object Systems*, 1996.
[3] The twelve-factor app. [Online]. Available: https://12factor.net/
[4] M. O. van Deventer and R. Joosten, “Hyperledge - after-the-fact mandate changes.”