CapBAC in Hyperledger Sawtooth

Stefano Bistarelli, Claudio Pannacci, Francesco Santini

CapBAC in Hyperledger Sawtooth

Stefano Bistarelli[1], Claudio Pannacci[2] and Francesco Santini[1]

[1] Department of Mathematics and Computer Science
University of Perugia, Italy
[stefano.bistarelli,francesco.santini]@unipg.it
[2] Faculty of Technology
Linnæus University (Växjö), Sweden
cp222kr@student.lnu.se

Abstract. In the *Internet of Things* (*IoT*) context, the number of connected devices can be too large for a centralised server. This paper focuses on how to enforce authorisation in such a distributed and dynamic environment. The key idea is to use a blockchain-based technology both as a way to maintain a common distributed ledger to store and use access control information, and as a way to enforce Access Control policies in the form of smart contracts. An implementation of an access-control system is presented as a proof of concept: it corresponds to an adaptation of the *Capability-based Access Control Model* (*CapBAC*) in the form of a transaction family in *Hyperledger Sawtooth*. The main claim is that the features and simplicity of CapBAC magnify the usefulness of a blockchain to control the access in the IoT.

1 Introduction

Regarding decentralised systems, recent years have seen the success of *Bitcoin* [13], based on the blockchain technology and the first crypto-currency that solved the double spending problem, which consequently granted trust in a trustless peer to peer network. Bitcoin opened the door for a new era of decentralisation touching every sector, not only the economical one, with improvements, variants and generalisations of the protocol enabling for a variety of services, like messaging platforms and distributed file storage, to be secure without a grantee.

Many researchers started to work on/with the blockchain, some of them seeing in this new technology also a solution for some problems related to the security in the *Internet of Things* (*IoT*). This paper follows in this belief: it is focused on the research of an *Access Control Model* (simply, *ACM*), among the ones already designed specifically for the IoT, which addresses all or most of the features required, such as *scalability*, *lightweightness*, and *privacy* [3], while being compatible with the blockchain architecture.

In this IoT scenario, the goal of this paper is "go back" and propose a "simple" ACM, as the *Capability-based Access Control Model* (*CapBAC*) [7], adapted to an underlying *Hyperledger Sawtooth* blockchain, with the purpose to manage capabilities. We implemented all the components of the proposed ACM in

Python (code released in a public repository, link in the following) and using Sawtooth. Sawtooth is also scalable and secure, two features needed by ACM for the IoT world (see Sect. 2.1). Among all Hyperledger blockchains, Sawtooth also comes with an SDK that allows its integration into Android applications. The "simplicity" of CapBAC and its focus on capability tokens lead to a high usability level in this kind of apps, particularly if compared to more complex ACMs (see Sect. 2.2).

However, we present this work as a proof-of-concept, mapping from the abstract model to the actual components, and paving the way to future ad-hoc blockchains. In fact, we believe (and explain why in the following of the paper) that CapBAC perfectly and easily integrates with the blockchain principles, but an ad-hoc blockchain developed to support CapBAC would overcome the limitations of Sawtooth and additional current ledgers. For instance, allowing untraceability and using "light" cryptography, in order to use constrained devices (even more constrained than mobile phones).

The paper is organised as follows: in Sect. 2 we introduce blockchains and ACMs. In Sect. 3 we describe CapBAC and we motivate why the use of a capability-based blockchain reinforces CapBAC as an ACM for the IoT. Section 4 describes our case-study by implementing such a model with Sawtooth. Section 5 reports related work, while Sect. 6 finally wraps up the paper with conclusive thoughts and future work.

2 Background

We divide background information into two different subsections, about blockchains (Sect. 2.1) and ACMs for the IoT (Sect. 2.2) respectively.

2.1 Blockchain

A *distributed ledger* (also *Distributed Ledger Technology* or *DLT*) is a consensus of replicated, shared, and synchronised digital data distributed across multiple sites. A peer-to-peer network and a consensus algorithms among such peers are required to ensure consistent replication across the nodes. One of the possible forms of DLT implementation is the blockchain system.

In 2009, Bitcoin [13] was proposed by Satoshi Nakamoto as a new cryptocurrency: approximately every 10 minutes, the nodes on the network come to consensus on a set of unspent coins, and the conditions required to spend them. This data set, known as the "unspent transaction output" (*UTXO*), can be modified by submitting transactions to the network that replace one or more UTXOs with a new set of unspent transaction outputs. In order to ensure that all nodes come to consensus, the Bitcoin protocol leverages a set of transaction validation rules and a consensus mechanism known as *Proof-of-Work* (*PoW*) [13]; this allows a permissionless[3] and anonymous participation in the consensus protocol.

[3] In permissioned blockchains, the network owners decide who can join the network, and in general any action as block verification, transaction issuing, or just consulting the blockchain, can be allowed or not by some *administrator*.

After Bitcoin, many other blockchain systems with different characteristics have been proposed, mainly with the purpose to improve its limited scalability, limited scripting language, and intermediate-statelessness design. However, most of them share the following characteristics:

- *Distributed*: The blockchain database is shared among potentially untrusted participants and it is demonstrably identical on all nodes in the network (at least in case of permissionless blockchains).
- *Immutable*: The blockchain database is an unalterable history of all transactions that uses block hashes to make it easy to detect and prevent attempts to alter the history.
- *Secure*: All changes are performed by transactions that are signed by known identities. These features and agreed-upon consensus mechanisms work to provide "adversarial trust" among all participants in a blockchain network.

Hyperledger[4] is an umbrella project of open source blockchains and related tools. All the frameworks part of the project (as *Fabric*, *Sawtooth* and *Iroha*, *Burrow* and *Indy*) share the common goal of improving performance of blockchains with the purpose to support global business transactions by technological, financial, and supply chain companies. The code is open-source and the same standards are followed in order to achieve inter-operability.

To implement the access control scheme we propose in this paper, we took advantage of *Hyperledger Sawtooth* because of the following features.

Scalability: Sawtooth was originally designed to overcome scalability challenges of a typical blockchain, such as Bitcoin. For this reason, the lightweight consensus algorithm *PoET* (*Proof-of-Elapsed-Time*) is adopted. PoET is a form of random leader election, wherein each validator node waits a random amount of time before trying to claim a block. In other random leader election algorithms like PoW, that randomness is enforced by searching for partial hash collisions.[5] A benchmark that measures the scalability of different blockchains (Hyperledger included) is presented in [5].

Security: in order to narrow the attack surface, Sawtooth has a contract logic which is termed as transaction families (more details in the following).

There are five core components in Sawtooth:

- A peer-to-peer network for passing messages and transactions between nodes;
- A distributed log which contains an ordered list of transactions;
- A state machine / smart contract logic layer for processing the content of those transactions;
- A distributed state storage for storing the resulting state after processing transactions;
- A consensus algorithm for achieving consensus across the network on the ordering of transactions and the resulting state.

[4] Hyperledger project: `https://www.hyperledger.org`.

[5] PoET replaces that work (needed by PoW) with trusted computing.

In Sawtooth, the data model and transaction language are implemented in a *transaction family*. A transaction family is a group of operations or transaction types that a programmer allows on the ledger. The users are expected to build custom transaction families that reflect the unique requirements of their ledgers.

The Sawtooth framework elaborates on the concept of smart contracts by viewing them as a state machine, or *transaction processor*. After passing through a strictly-ordered distributed log (i.e., a blockchain), transactions are routed to the appropriate processor. These processors than ingest the payload of transactions, as well as any relevant state, before processing the transaction (updating the state). Sawtooth is capable of supporting both the stateless (UTXO) and stateful (Ethereum-style[6]) models. Smart contracts can be written in different languages (e.g., Python or Solidity). By creating a domain specific transaction processor, it is much easier to limit the types of actions that can be performed on a blockchain network, which can improve security and performance.

Sawtooth is a configurable blockchain with a focus on security and designed for IoT scenarios, however, as already stated in Sect. 1, the model we propose in this paper is not meant to be blockchain-specific. To develop an "ad hoc" blockchain could be the only way to include constraint devices in the secure peer-to-peer network.

2.2 IoT and ACMs

The term "Internet of Things" was used for the first time in 1999 to describe a scenario in which computers, and so the Internet as a whole, can gather real world data without human intervention [2]. In recent years the term became widespread under the acronym "IoT" and usually refers to a network of constrained devices, as embedded systems with sensors and actuators, connected to the Internet.

Concerning how the security of the IoT can be enforced, in this paper we focus our attention on *authorisation*. It refers to the specification of access rights or privileges to specific resources, which in our case is the information gathered and managed by IoT devices. The OM-AM[7] reference model proposed in [15] gives a better view of what authorisation means in the IoT world. Among its required objectives, authorisation has to be *decentralised*, *anonymous* or *pseudonymous* (*unlinkability* prevents from connecting pseudonyms to real ids), *scalable*, *lightweight* (low use of resources such power and memory), it needs to offer revocation and delegation of rights, the response time has to be low, and access information needs to last for a long time.

In the following, we will briefly discuss some ACMs frequently used in the IoT; for their connection to related work, see Sect. 5. CapBAC-based schemes, access rights are granted to subjects based on the concept of capability, which is a transferable and unforgeable token of authority (e.g, a key), and describes a set of access rights for each subject (in Sect. 3 we will provide more details).

[6] Sawtooth also supports Ethereum smart contracts via *Seth*, a Sawtooth transaction processor integrating the *Hyperledger Burrow Ethereum Virtual Machine*.

[7] Objective-Model Architecture-Mechanism layers (OM-AM).

DCapBAC [9] is a distributed version of CapBAC developed with the specific purpose to control the access in the IoT. The framework conceived in this paper moves the distribution-related features of DCapBAC to the blockchain level.

In the *RBAC*-based schemes [6], access control is based on the roles (e.g., administer or guest) of subjects (i.e., entities that access to resources) within an organisation. By associating roles with access rights (e.g., read, write, execute) and assigning roles to subjects, the RBAC-based schemes can establish a many-to-many relationship between access rights and subjects.

The *ABAC*-based schemes [10] exploit policies that combine various types of attributes, such as subject attributes, object (i.e., the entity that holds resources) attributes and environment attributes, in order to define a set of rules expressing under what conditions access rights can be granted to subjects.

OrBAC [11] focuses on the concept of organisation: a security policy that applies to a given organisation is defined as a collection of permissions, prohibitions, obligations, and recommendations.

Finally, the work in [3] proposes a Trust-based ACM (*TBAC*), which adds a trust evaluation on top of the decentralised architecture of DCapBAC.

3 CapBAC and Blockchains

In Sect. 3.1 we introduce the basic components of CapBAC and their interaction, while in Sect. 3.2 we suggest how it smoothly integrates with a blockchain system.

3.1 CapBAC

In a typical use-case of CapBAC, after the owner of a *resource*, hosted by a *device*, or someone delegated to do so, referred to as the *issuer*, issues a *capability token* in name of a *subject* for that particular resource, *subject* is then able to access to a resource by sending an *access request* to *device* in which the capability token is attached. The authorisation is granted if the token passes the *authorisation procedure*. We now briefly introduce all the main actors in such a scenario.

A **resource** is a univocally identifiable and actable-upon object (e.g. a RESTful resource). A **capability token** is a communicable object (e.g. in *JSON*) digitally signed by the issuer, in which the subject is represented by its public key. It is unequivocally identifiable by its id, it has a time-stamp and a validity time interval. Moreover, it has a field to store the issuer's capability that is used to specify from which parent capability it is derived from, in case of delegation. A capability with no parent is said to be a *root capability*. In a delegation every token is chained to the one it is derived from up to the root. A **revocation token** contains the issuer's capability as authorisation. The revoked token is specified along with the revocation type. An **access request** for a service/operation on a resource also includes (or refer to) a capability token. A **policy decision-point** (*PDP*) is a resource-agnostic service in charge of managing the validation of the access rights granted in the received capabilities against local policies, and it updates the capability database. A **resource manager** manages the requests

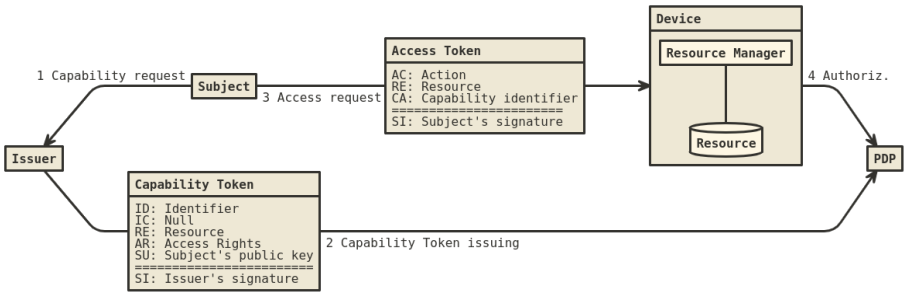

Fig. 1. Token issuing and the access request procedure in CapBAC.

for a resource. It checks the acceptability of the capability token shipped with the service request, as well as the validity and congruence of the requested service/operation against the presented capability, by taking also the validation outcome from the PDP into account. The **revocation service** validates incoming capability revocations and it updates the capability and revocation databases (storing the revocation tokens).

When an access request for a resource is received by a device, the **authorisation procedure** is executed: the resource manager first *i)* checks the *formal validity* of all the capabilities in the authorisation chain, then *ii)* it checks the *logical validity* of the request by evaluating the congruence of the operation granted by the capability with the operation reported in the initial request; finally, *iii)* it forwards the request to the PDP. The PDP checks the applicability of the operation by *i)* verifying the logical validity of each capability in the delegation chain, and *ii)* inquiring the revocation service. The revocation service checks in the revocation database if the capability wasd. After this step succeeds, the authorisation is granted. Figure 1 and Fig. 2 summarise the issuing procedure of a token, the access request and revocation procedures in the CapBAC model.

3.2 Blockchains with CapBAC: Characteristics

In this section we survey the advantages in adopting CapBAC supported by a blockchain model, highlighting the compatibility between the two, and commenting about related work.

Capability and revocation database. The main advantage of a blockchain is that it offers a trusted distributed database, which can be used in an ACM to address revocation and delegation in an effective way. In DCapBAC [9], both revocation and delegation are abandoned in favour of pure distribution (we instead keep both these two phases in our proposal): the *PDP* is integrated into the device itself and the capability tokens, once issued, are stored by their subject, which attaches the whole token to its requests. On the other hand, a ledger can be used to store both the capability and revocation tokens granting revocation and

delegation even in a distributed scenario: this feature is almost always required when dealing with IoT security. A single database for both the capability and revocation tokens was adopted in the implementation proposed in [7].

PDP and revocation service. A blockchain featuring smart contracts can apply any rule-set to the validation of transactions; this means that the work done by the PDP in [7] and [3] can be distributed to all the blockchain nodes in the form of transaction validation. The same also holds with the controls performed by the revocation service. Their computation is thus distributed.

Tokens. In our proposal, tokens are represented on the ledger state of Sawtooth. In this case, the CapBAC model has the great advantage, over other ACMs, of using tokens, i.e. the capability tokens, that behave in a similar way to transactions in a token-based crypto-currency. The authorisation/ownership is granted by the issuer/sender with the inclusion of the subject/receiver's public key in the token/transaction; the issuer/sender has to digitally sign the token/transaction; the whole delegation chain/transaction history is required in order to prove the validity/ownership of the capability/token. In such a CapBAC blockchain, a transaction is considered valid only if it contains all the fields of a capability token and the hash of the parent transaction (owned by the issuer), which has to allow the delegation on all the stated access-rights. The revocation concerns a different kind of transaction (without owners), also chained to a pre-existing capability transaction in order to spend/revoke it and/or its descendants. Once added to the ledger, a capability transaction is valid until it (or a capability in its chain of transactions) expires or it is revoked.

Privacy. Blockchain privacy is controversial. In our proposal, transparency is granted by default since anyone in the network can access to the ledger. The capability system is user-driven by design, since the owner of a resource is always in full control of the rights granted: owning the root capability, she can easily revoke any derived access right. The challenge is represented by anonymity: while *pseudonymity* is obtained by using signature validation as identification and authentication means, blockchains usually do not easily grant *unlinkability* or *untraceability*, e.g. the identity of the participants can be inferred by looking at the transaction history [16]. A capability token-based blockchain, however, could achieve the complete detachment between identities and capabilities by exploiting the delegation system as described in [7, Sect. 6].[8] Moreover, *unobservability* of accesses is granted, since the request validation can be performed without broadcasting any information, and *decentralisation* is the key feature offered by both capability-based models and blockchains.

Confidentiality and Integrity. As previously stated, the selling points of the proposed system are the *granularity* of access control rights and an effective *delegation* and *revocation* procedures.

[8] This issue in not addressed in this paper, but ad-hoc blockchains can be designed following the example of untraceable tokens as in *Monero* crypto-currency.

Reliability and Availability. The *offline mode* is not available with a blockchain: the communication among devices is required during the authentication procedure. The *short-term availability* of information depends on the implementation: e.g., Sawtooth provides a low latency [5]. Finally, *long-term availability* is a key feature of the blockchain, since all the stored information is unalterable.

Social and Economic aspects. Given the diffusion of inter-operating private blockchains, such as the *Hyperledger* project, collaboration is to be expected between different implementations following the same standards: we propose CapBAC as an ACM to be supported by different blockchains. To achieve *context-awareness*, additional considerations need clarifications, as the use of a level of trust, either in access control rules or in underling consensus algorithm.

Technological Constraints. While the proposed system provides high *flexibility* and *scalability*, the main question is whether or not it would be able to run on constrained devices, i.e. if it provides the *lighweightness* required by IoT. At first the idea of having every device storing a copy of the whole ledger could seem too memory intensive for it to be deployed on low-hardware devices, and it certainly is at the current state of technology. However, as for the PDP in [3] and [7], it is not necessary for every device to be a blockchain node, i.e. storing the ledger and providing validation, but the ones without enough resources could rely on a validator network composed of more powerful devices. We can also refer to [1] as a proof of the feasibility of this approach. Finally, *heterogeneity* would require some standardisation efforts, but it is achievable because nothing about the proposed system is hardware-specific.

Usability. The presented capability system is "simple", as well as the development of highly accessible applications that can mange rights through tokens (as crypto-currency *wallets* for mobile devices already do); this makes the proposed solution *user-friendly*. A high-level interface could allow everyone to manage the access-control rules of her own device.

4 Implementation

We dedicate this section to the presentation of our proof-of-concept, by using Hyperledger Sawtooth as the blockchain where to implement a CapBAC ACM. Ad-hoc blockchains could be envisioned in the future, for instance achieving untraceability (see Sect. 3.2), or fully exploiting cryptographic primitives of resource-constrained devices, typical of the IoT. However, the following implementation paves the way to such solutions, proving their feasibility.

We use the ledger state to represent the issued capability tokens, while the policies are enforced during blocks validations. The implementation runs in a *Docker*[9] using *Ubuntu* images on a 64bit architecture, thus it is not directly

[9] Docker.com: `https://www.docker.com/`.

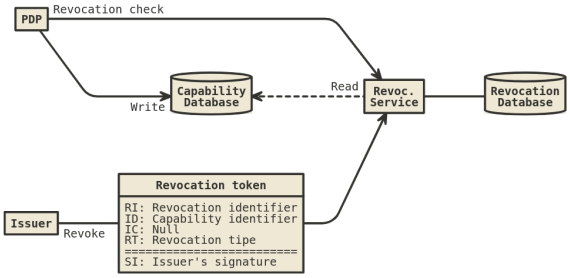

Fig. 2. CapBAC model: a visual description of the capability and revocation databases, and revocation service.

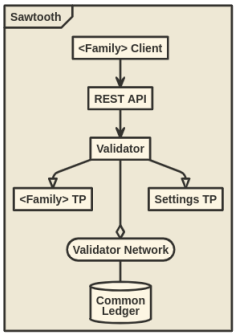

Fig. 3. A Sawtooth node with a custom transaction family.

meant for constrained devices, but it is anyhow useful for a proof-of-concept. All the developed source code was committed to a *GitHub* repository[10].

Transaction Families. In Sawtooth it is possible to build a custom service on top of the pre-built blockchain architecture by defining a so called *Transaction Family* (*TF*). This is possible because the core system is detached from the application layer. The main component of a TF is the *Transaction Processor* (*TP*) which, as the name suggests, is where the evaluation of incoming transactions is performed, according to the built-in rule-set (in a smart contract) written in one of the supported programming languages. More TPs can be attached to the same *validator*, which is a node in the peer-to-peer validator network responsible for maintaining the common ledger. For example, Sawtooth provides a TF for managing the settings saved in-state (including the ones used by the consensus module); its TP is always required on the side of all the others. Each TP has an addressing space in the ledger state for storing its information usually obtained from the family name itself. Between the validator and the client resides the *REST API*, also offered by Sawtooth, which manages the incoming requests. To be noted is the fact that Sawtooth allows the transmission of any kind of data format and encoding; the only requirement is that every participant to the network has to sign in with an *RSA* key pair in order to send transactions. Multiple transaction requests are encapsulated in a transaction batch (also signed), which is atomically validated (a single faulty transaction will cause the batch to be rejected). When a batch is accepted, the ledger state is modified accordingly to the transactions it contains. It is also possible to generate custom events that spread through the network and can be listened by any client aware of the specific transaction family from which they are generated. Figure 3 shows the relations between the various component: the validator is the main component, since it evaluates transaction requests forwarded from the REST API. It has a

[10] Proof-of-concept project: `https://github.com/kappanneo/sawtooth-capbac`.

TP as additional module, and it behaves as a peer in the peer-to-peer validator network, which shares the same *Common Ledger*.

4.1 A New Transaction Familiy for CapBAC

We now describe the operations implemented by the new TF developed in Sawtooth. These operations can be performed by any client, where "client" is a software module that receives and executes commands in any peer of the network. The CapBAC transaction family allows users to `issue` and `revoke` capability tokens in which access rights for a particular resource are given to a specific subject. Since the tokens are stored in the ledger state, both issuing and revoking steps require a transaction request, sent to the validator and processed by the TP. A CapBAC client can also **list** the tokens stored in the current state for a specific device, `sign` an access token composed of a resource request and a capability id, and, finally, `validate` it over the ledger state. The client can be used from the command line with the following syntax:

```
list <device URI>
issue [--root] <not-signed capability token as JSON string>
revoke <not-signed revocation token as JSON string>
sign <not-signed access token as JSON string>
validate <access token as JSON string>
```

State. Capability tokens are stored in-state in a *Concise Binary Object Representation* (*CBOR*) [4] encoded dictionary. The format in which they are stored is different from the one used for the issuing and as transaction payload, since the processor removes the fields that are not necessary anymore after validation, and it reformats the access rights to let the validation of access tokens be more efficient. In this scenario, the information contained in the ledger is already tamper-proof, thus there is no need for the token to be stored as it is: the state is obtained as a consequence of the whole transaction history and transactions are the ones to be stored as-they-are for validation purposes, so the signature for the capability tokens is no longer required; this is one of the main differences between an implementation of the model on top of the state of a new-generation blockchain, versus the use of ad-hoc blockchains where transactions themselves would represent capability tokens, hence without the need of a ledger state. For the same reason, the revocation tokens are not saved, but the state is changed according their content instead, just by removing revoked tokens. Since the global state is obtained as the result of the whole transaction history, it is always possible to inspect it in order to recover previously revoked tokens.

Addressing. CapBAC data is stored in the state dictionary using addresses that are generated from *i)* the CapBAC name-space prefix, and *ii)* the unique *URI* of the device. The latter is also the parameter passed to the `list` command, and one of the attributes of the JSON entry of the other commands ("DE" in the examples shown in Fig.6), so that they all refer to the same address in the

ledger state. The choice of a *per-device* addressing has many advantages: the whole capability token tree for a device is under the same address: neither a request has to include a reference to all the capabilities in the delegation chain, nor the validator has to search for them across the whole state. The address for the whole tree is the only input and output of the transaction, hence meaning that is by-design impossible to change access data for a device different from the one reported in a transaction. The URI of devices does not need to be explicitly saved in the state, thus it is removed to increase privacy. URIs are by definition unique: they will not cause name conflicts, while token identifiers can be reused for different devices. Addresses adhere to the following format:

- an address is a 70 character hexadecimal-string;
- the first 6 characters of the address are the first 6 characters of an *SHA512* hash of the CapBAC name-space prefix: `capbac`;
- the following 64 characters of the address are the last 64 characters of an `SHA512` hash of the device URI.

Transaction Payload. The payload is an object encoded in CBOR with two fields: the name of the action performed, and the corresponding token as object. As already said, the commands `capbac issue` and `capbac revoke` are the only ones that can be used to create and send transactions. Therefore, the object of the payload has only two possible formats: one for the capability, and one for the revocation token. The parameters of the transaction header are : the *inputs and outputs* (the address generated using the device URI), the *dependencies* (in our case `None` since our transaction family does not depend on any other transaction family), the *family name* (i.e., `capbac` version: 1.0), and the *encoding* (the encoding field needs to be set to `application/cbor`).

Execution. As shown in Fig. 4, when the TP receives a transaction it first checks the formal validity of the payload, including the specific token format in relation to the proposed action. In case of the `issue` command, also the logical validity of the time interval is checked. Finally, the signature is verified and the state is retrieved.

If the transaction is created for issuing a token, first the TP checks if a token with the same ID is not already in the state, and if the issuer is the owner of the parent capability; then, the token is reformatted to match the in-state representation and its validity is tested over the delegation chain, by checking at each step that all the access rights are included in the parent token, the delegation for each is allowed and the token is not expired. If all the delegation chain is valid up to the root capability, the token is added to the state.

If the transaction is due to the revocation of a token, then the authorisation of the requester is checked by also verifying that the revoker's capability is an ancestor of the revoked one. If the operation is valid, then the tokens are removed according to the revocation type. Finally, the signature of the token is verified against the requester public key, and the state is updated.

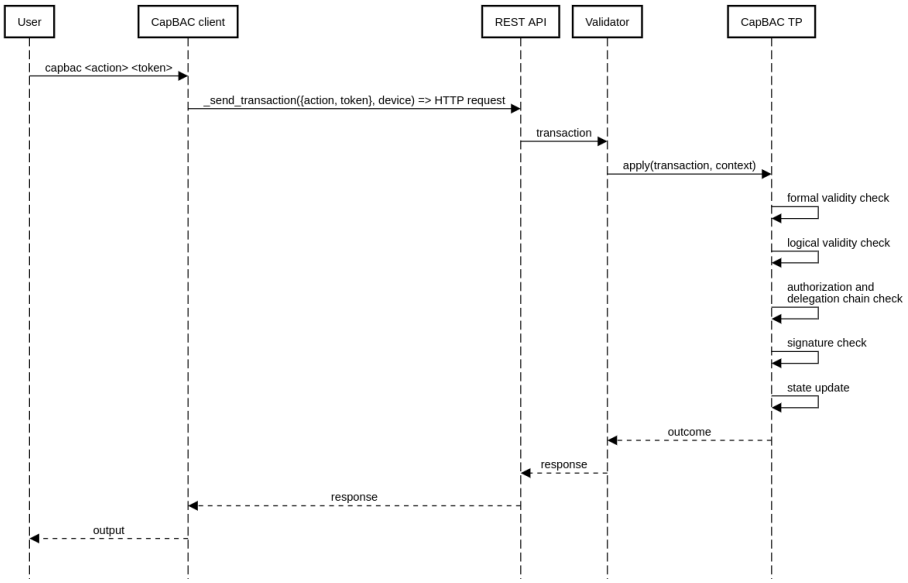

Fig. 4. The sequence diagram for the issuing or revocation of capabilities: the two possible values for *action* are `issue` and `revoke`, while *token* can respectively be a capability or a revocation token.

4.2 Testing Environment

Our implementation features a testing environment that can be assembled by using *Docker Compose*, a tool for defining and running multi-container Docker applications: we build a container for each actor in the scenario shown in Fig. 5. The REST API and validator both run in two separate containers assembled from the images offered by Sawtooth. This set-up can be used to test the functioning the introduced TF within a single node. Indeed, the same Docker images can be used to create a more complex network composed of multiple nodes. Scalability tests will be part of future work, aslo because we primarily believe in the development of an ad-hoc blockchain, and Sawtooth is used as a proof-of-concept towards this ultimate goal.

Issuing of a capability token. Tokens are issued using the `issue` command with an incomplete capability token (as the ones shown in Fig.6) as parameter. An example of capability issuing is the one automatically performed by *device* before starting the *CoAP* server[11]. For testing purposes, the two resources *device* will open to requests are *i)* `time` (the actual time on the machine) and *ii)* `resource` (a re-writable string with no meaning). Since it is the first token to be issued under the address space for the URI `coap://device`, the token will be a root

[11] The *Constrained Application Protocol* [18], is the IoT standard transfer-protocol at the application layer, and it is based on the same RESTful principles as HTTP.

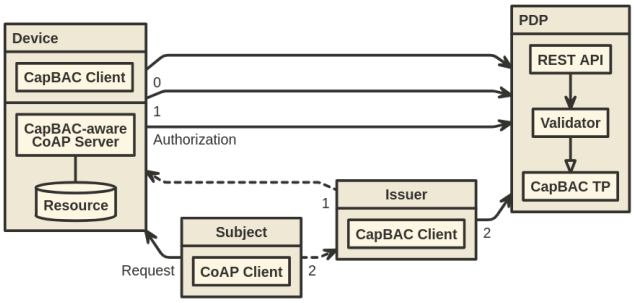

Fig. 5. A testing scenario of a CapBAC TF. Each one of the four entities is executed in a different Docker container. The numbered relations represent the issuing of a capability token and, when dashed, the corresponding delegation hierarchy. A JSON representation of the capability tokens is shown in Fig. 6.

token. A root token does not have a parent and, as architectural constraint desirable in this implementation, its subject is always the issuer of the token itself: hence, in this specific case it is *device*. Since the CapBAC client has access to the public key of the issuer of the token, the subject field is automatically filled in before signing the token, and the parent capability is set to `null`. The signing procedure also adds the TF version (1.0) and a timestamp, required for a signature to be unique. If a given token is not a root token, then also the parent capability needs to be specified. After a token is assembled, it is set as the object of a transaction payload, with action set to `issue`; then, it is sent to the validator through the REST API.

Delegation. The owner of a capability token can delegate to someone else any of the access rights it is granted, only if their *delegation depth* (`DD`) is greater than zero. This is done by issuing a new capability token for the same device that refers to her capability, also known as *issuer's capability* (`IC`) or *parent capability*, and listing a subset of those access rights. Moreover, in the access rights of the new token, the `DD` of each resource has to be strictly less than the one in the parent token. If this condition is not satisfied, then a token is invalidated by the TP, and the corresponding transaction is discarded by the validator. If the *device* now wants to delegate all its access rights to *issuer*, *device* can do so by using `issue` with a capability token formatted as the second one shown in Fig.6. In the same way, if *issuer* wants to give to *subject* some access rights from the ones she also has, but without granting the possibility to delegate them any further, *issuer* can issue a capability token as the third one from the right in Fig. 6.

Accessing to a resource. Once in possession of a capability token, *subject* can access to a resource on *device* by sending a CoAP request with an access token as a prefix to its payload. When *device* receives this request, first it checks that the access token matches the request itself, then it passes the token to `validate`. It searches for the referred capability in the ledger state, also *climbing*

Fig. 6. An example of a delegation chain showing the JSONs used during the issuing of the respective capability tokens. The leftmost one is for a root capability. Following the same delegation relations shown in Fig.1, both 1st and 2nd capabilities (from the right) are issued by the *device*, while the 3rd one by the *issuer*.

the delegation chain up to the root token. If the access token refers to a valid capability, then the operation requested by *subject* is performed. The whole access procedure is summarised in Fig. 7.

Revoking a capability token. At any time, an issued capability token can be revoked by its issuer or the issuer of an ancestor capability, i.e. a capability that is "higher" in the delegation chain of the revoked one. In our case this operation is performed via a `revoke` transaction including a revocation token in its payload. This transaction, as for the `issue` one, is built from an incomplete token, and it is completed by the CapBAC client. The revocation type (i.e., `RT`) field specifies the type of revocation, and it can be one of the following:

- `ICO` (Identified Capability Only): it revokes only the capability identified in the revocation;
- `DCO` (Descendant Capabilities Only): it revokes all the descendants of the identified capability;
- `ALL`: it revokes the identified capability and all its descendants.

5 Related Work

In order to overcome the problems related to ABAC and RBAC concerning scalability and flexibility, a CapBAC model is proposed in [7] with a focus on IoT. It offers revocation, delegation support, and granularity of the access rights. It is meant to be easy to understand and to use, removing the burden represented by the management of identities. Also, by design, it enforces the *Least Privilege* principle, i.e., only the rights strictly necessary are granted, so that abuses are

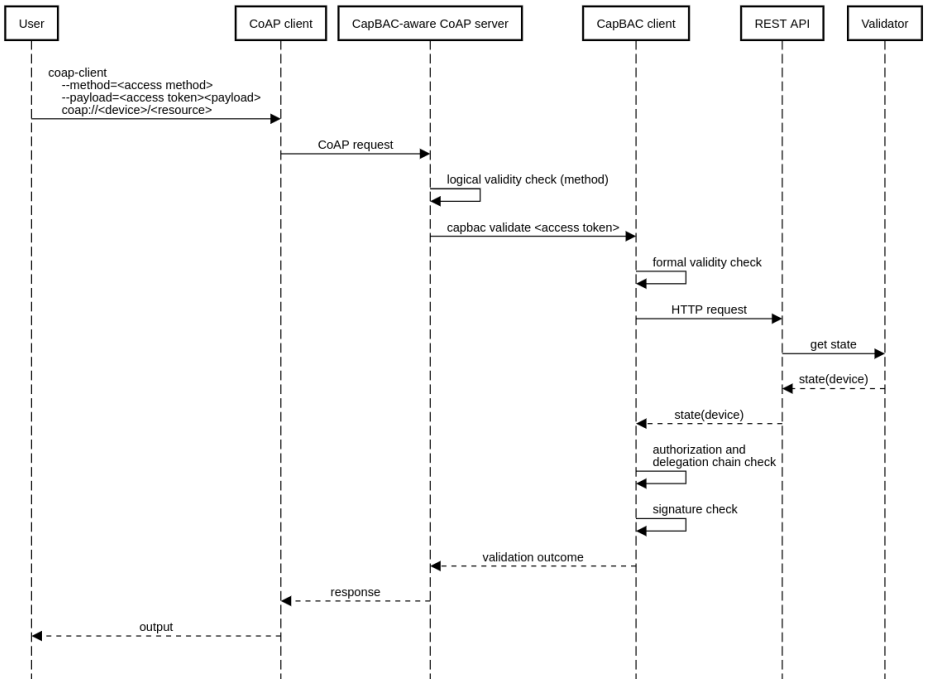

Fig. 7. A sequence diagram of scenario where a user want to access to a resource.

not possible. However, it does not specify how the issuing of tokens could be enforced and it is based on the *RSA* encryption scheme [17] for authentication, which is not supported by constrained devices, typical of the IoT world.

In [9], a distributed CapBAC is presented as DCapBAC. Compared to [7] it does not offer neither delegation nor revocation since the focus of the paper is the authentication through "Elliptic Curve Cryptography" (*ECC*), i.e. a public-key crypto-system compatible with constrained devices. Other improvements are the use of *JSON* as data format for the tokens, and the use of the *Constrained Application Protocol* (*CoAP*) [18], recently announced as the standard transfer protocol at the application layer for IoT, and based on the same RESTful principles as HTTP. As in [7], the generation of tokens is not discussed.

In [12] the authors propose a blockchain technology to publish the policies expressing the right to access a resource and to allow the distributed transfer of such rights among users. They take advantage of Attribute-Based Access Control (ABAC) policies. However, their proof-of-concept implementation is based on the Bitcoin blockchain (quite limited on the application side), which cannot handle smart contracts (as Sawtooth can instead do), and thus the actual authorisation system is external to the blockchain, which only stores tokens. Indeed, not having the authorisation logic on the blockchain is a critical point.

In [19] a blockchain access control ecosystem is implemented with *Hyperledger Fabric*; the Hyperledger composer modelling tool is used to implement the smart contracts or transaction processing functions that run on the blockchain network. The authors adopt a RBAC-based scheme: users are assigned roles and roles are assigned privileges controlled by asset owners. A smart contract is triggered to pull the roles that have access to that asset. As advanced in Sect. 3, we believe that a simple scheme (as CapBAC) with no super-entity keeps the model more secure and scalable, particularly if the architecture is IoT-oriented.

In [1] an implementation of a token-based ACM on top of a private Ethereum blockchain network is showcased, featuring a *Proof of Possession* (*PoP*) consensus protocol, to bind the client's identity to an access token. The focus is on the feasibility of the blockchain architecture: neither the token format nor the access control model are described in details. However, this is not a limitation since the Ethereum's smart contract language allows for any access control rule to be described so that the proposed solution can potentially express any ACM.

Two other IoT-related ACMs are presented in [14] and [20]. The former proposes a private blockchain for managing *Identity and Access Management*, while the latter advances a proof-of-concept prototype implemented on both resources-constrained devices, tested on a local private blockchain network.

6 Conclusion and Future Work

The aim of this paper was to propose the use of CapBAC revitalised by managing capabilities for heterogeneous and light devices with a blockchain. The ultimate purpose is to secure the IoT with a scalable and decentralised implementation that takes advantage of an underlying blockchain-based architecture. Most features of CapBAC, as for instance its simplicity and fine-graineess, smoothly adapt to a distributed ledger, which on the other hand enforces trust in a naturally unsafe environment as the IoT. We have discussed the features of the proposed system along with its problems (and possible solutions to them). Finally, we showcased an implementation realised by exploiting the transaction families of Hyperledger Sawtooth, in order to show that the mapping from the proposed model to a real blockchain-based architecture is possible.

A possible continuation of this work could be the implementation of a capability-based access control solution using a new, memory-efficient and multi-purpose private blockchain as proposed in [1], and featuring a lightweight public key encryption system (like what used also in [9]). Actually, different blockchains could implement this model, at different levels and for different scenarios, interoperating to form a wide system in which authorisation decisions are taken thanks to a distributed effort. One of these technologies could even be a blockchain with the only purpose of managing accesses, in which the object of transactions are capability tokens and the validation operation is light enough to reach even the most constrained devices.

OAuth [8] is the standard currently proposed for authorisation frameworks, and it is also based on delegable tokens. However, it inherits the centralised

approach from its previous version, and so it requires a trusted third-party in order to work. We leave a possible integration of OAuth with the model proposed in this paper as future work.

Acknowledgment

This research is supported by project "REMIX" (funded by Banca d'Italia and Fondazione Cassa di Risparmio di Perugia).

References

1. Alphand, O., Amoretti, M., Claeys, T., DallAsta, S., Duda, A., Ferrari, G., Rousseau, F., Tourancheau, B., Veltri, L., Zanichelli, F.: Iotchain: A blockchain security architecture for the internet of things. Tech. rep., IEEE Wireless Communications and Networking Conference, Murcia 30100, Spain (2018)
2. Ashton, K., et al.: That internet of things thing. RFID journal **22**(7), 97–114 (2009)
3. Bernabe, J., Ramos, J.H., Gomez, A.S.: TACIoT: multidimensional trust-aware access control system for the internet of things. Soft Computing (2015)
4. Bormann, C., Hoffman, P.: Concise binary object representation (CBOR). IETF RFC 7049 (2013)
5. Dinh, T.T.A., Liu, R., Zhang, M., Chen, G., Ooi, B.C., Wang, J.: Untangling blockchain: A data processing view of blockchain systems. IEEE Trans. Knowl. Data Eng. **30**(7), 1366–1385 (2018)
6. Ferraiolo, D., Kuhn, D.: Role-based access control. 15th National Computer Security Conference p. 554563 (1992)
7. Gusmeroli, S., Piccione, S., Rotondi, D.: A capability-based security approach to manage access control in the internet of things **58**, 1189–1205 (2013)
8. Hardt, D.: The oauth 2.0 authorization framework. RFC 6749, RFC Editor (October 2012), `http://www.rfc-editor.org/rfc/rfc6749.txt`, `http://www.rfc-editor.org/rfc/rfc6749.txt`
9. Hernàndez-Ramos, J., Jara, A., Marín, L., Gómez, A.S.: DCapBAC: embedding authorization logic into smart things through ECC optimizations (2016)
10. Hu, V.C., Ferraiolo, D., Kuhn, R., Friedman, A.R., Lang, A.J., Cogdell, M.M., Schnitzer, A., Sandlin, K., Miller, R., Scarfone, K., et al.: Guide to attribute based access control (ABAC) definition and considerations (draft). NIST special publication **800**(162) (2013)
11. Kalam, A.A.E., Benferhat, S., Miège, A., Baida, R.E., Cuppens, F., Saurel, C., Balbiani, P., Deswarte, Y., Trouessin, G.: Organization based access contro. In: 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY. p. 120. IEEE Computer Society (2003)
12. Maesa, D.D.F., Mori, P., Ricci, L.: Blockchain based access control. In: Distributed Applications and Interoperable Systems. LNCS, vol. 10320, pp. 206–220. Springer (2017)
13. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2009)
14. Nuss, M., Puchta, A., Kunz, M.: Towards blockchain-based identity and access management for internet of things in enterprises. In: Trust, Privacy and Security in Digital Business - 15th International Conference TrustBus. LNCS, vol. 11033, pp. 167–181. Springer (2018)

15. Ouaddah, A., Mousannif, H., Elkalam, A., Ouahman, A.: Access control in the internet of things: Big challenges and new opportunities **112**, 237262 (2016)
16. Pfitzmann, A., Köhntopp, M.: Anonymity, unobservability, and pseudonymity - A proposal for terminology. In: Designing Privacy Enhancing Technologies, International Workshop on Design Issues in Anonymity and Unobservability. LNCS, vol. 2009, pp. 1–9. Springer (2000)
17. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978)
18. Shelby, Z., Hartke, K., Bormann, C.: The constrained application protocol (coap). IETF RFC 7252 10 (2014)
19. Uchibeke, U.U., Kassani, S.H., Schneider, K.A., Deters, R.: Blockchain access control ecosystem for big data security. CoRR **abs/1810.04607** (2018)
20. Xu, R., Chen, Y., Blasch, E., Chen, G.: Blendcac: A smart contract enabled decentralized capability-based access control mechanism for the iot. Computers **7**(3), 39 (2018)