RESEARCH ARTICLE

Trustful ad hoc cross-organizational data exchanges based on the Hyperledger Fabric framework

Laurens Van Hoye* | Tim Wauters | Filip De Turck | Bruno Volckaert

[1]IDLab, Ghent University - imec, Belgium

Correspondence
*Laurens Van Hoye, Technologiepark-Zwijnaarde 126, 9052, Ghent, Belgium.
Email: laurens.vanhoye@ugent.be

Abstract

Organizations share data in a cross-organizational context when they have the goal to derive additional knowledge by aggregating different data sources. The collaborations considered in this article are short-lived and ad hoc, i.e. they should be set up in a few minutes at most (e.g. in emergency scenarios). The data sources are located in different domains and are not publicly accessible. When a collaboration is finished, it is however unclear which exchanges happened. This could lead to possible disputes when dishonest organizations are present. The receipt of requests / responses could be falsely denied or their content could be point of discussion. In order to prevent such disputes afterwards, a logging mechanism is needed which generates a replicated irrefutable proof of which exchanges have happened during a single collaboration. Distributed database solutions can be taken from third parties to store the generated logs, but it can be difficult to find a party which is trusted by all participating organizations. Permissioned blockchains provide a solution for this as each organization can act as a consensus participant. Although the consensus mechanism of the permissioned blockchain Hyperledger Fabric (version 1.0-1.4) is not fully decentralized, which clashes with the fundamental principle of blockchain, the framework is used in this article as an enabler to set up a distributed database and a proposal for a logging mechanism is presented which does not require the third party to be fully trusted. A proof of concept is implemented which can be used to experiment with different data exchange setups. It makes use of generic web APIs and behaves according to a Markov chain in order to create a fully automated data exchange scenario where the participants explore their APIs dynamically. The resulting mechanism allows a data delivering organization to detect missing logs and to take action, e.g. (temporarily) suspend collaboration. Furthermore, each organization is incentivized to follow the steps of the logging mechanism as it may lose access to data of others otherwise. The created proof of concept is scaled to ten organizations, which autonomously exchange different data types for ten minutes, and evaluation results are presented accordingly.

KEYWORDS:
blockchain, collaborations, cross-organizational, distributed, logging

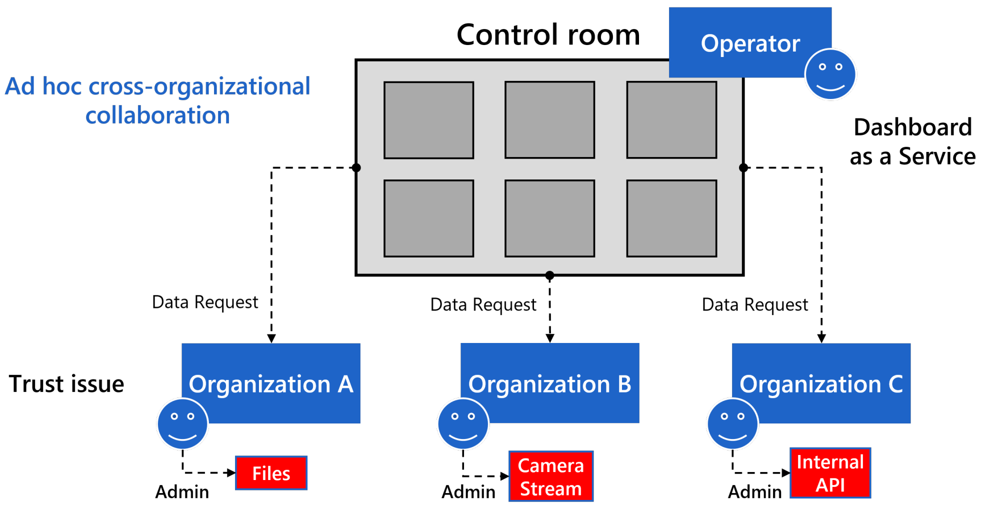


FIGURE 1 Sample ad hoc cross-organizational collaboration where multiple organizations work on a common project: a control room operator requests data from other organizations in order to offer a dashboard as a service to speed up a decision-making process.

1 | AD HOC CROSS-ORGANIZATIONAL COLLABORATIONS

The context of this research is the exploration of technical enablers for allowing the rapid creation and configuration of ad hoc cross-organizational collaborations aiming to share knowledge (e.g. data, services) among the different participants. Although any type of collaboration between different organizations can be considered a valid use case for this article, a prominent one is that of an emergency situation[1] or a smart city scenario[2]. An example collaboration scenario is illustrated in Figure 1 where a control room operator wants to access data sources from different participating organizations. The aggregated data stream will be shown on the control room dashboard, such that the operator can offer this service to speed up a decision-making process. Persons are involved in enabling and disabling a stream of data exchanges, more specifically in granting and revoking authorization to internal data sources.

Although it might be necessary to collaborate, there may be trust issues between the involved organizations. Trust in this context can be defined as an assumption of belief in the honest and truthful operation of another organization at a particular point in time. When an organization shares data with the operator as shown in Figure 1, it wants to be sure that it has a proof of each data exchange, such that it can be compensated for each unique and valuable piece of data it delivers, and that the operator only requests data relevant to the problem at hand. In general, it should not be possible for any data requesting organization to falsely deny (the integrity of) a request / response in order to avoid any responsibility or obligation towards the sending party. Most companies will therefore not trust others, especially in the case of ad hoc collaborations, for which there is no time to negotiate data exchange contracts or service level agreements upfront. Not only the time is an issue, but it may also be difficult to determine upfront which data needs to be exchanged in an ad hoc situation. For the remainder of this article, it is thus assumed that there is a lack of full trust between the collaborating organizations.

The core problem is that, after a collaboration is finished, it is unclear which organization performed which action when no logging mechanism is used. This means that disputes between the partners are possible when dishonest organizations are present. Two types of possible disputes are (1) disputes concerning the (integrity of) a request / response and (2) disputes concerning potentially wrongful acts. The latter could be the case when there are no fine-grained access control policies in place due to the collaboration being set up ad hoc. A solution to this problem is to provide the different partners with a pair of cryptographic keys and to let them execute the following steps:

1. Organization Y (Operator) sends a signed request to Organization X
2. Organization X sends a signed notification confirming it received the request of Organization Y (Operator)
3. Organization X sends a signed response to Organization Y (Operator)

4. Organization Y (Operator) sends a signed notification confirming it received the response of Organization X

This mechanism provides both Org X and Y with the logs it would need to prove which data exchanges happened in case of a possible dispute in the future. It is important to note that these logs only capture the data streams that took place, not whether the data was relevant. For example, when a camera stream is requested and only empty frames are sent, compensation for sending this stream is questionable. The logging mechanism thus provides a way to observe the exchanges that have happened and to impose consequences after the collaboration is finished.

Although this solution is sufficient to solve the proposed problem, it also needs to be examined how the logs need to be stored. Org X and Y could store their exchanged logs separately, but this has two drawbacks. First, when one of them loses its logs, disputes are again possible. Data replication is thus needed instead. Second, there is no central overview of all exchanges that happened during a single collaboration. As multiple organizations will join a collaboration to achieve a common goal, it would be better to aggregate all the different logs, e.g. to allow participating organizations to create a live overview of the current status of the collaboration, to execute data exchanges according to a predefined regulated process (e.g. when data needs to be exchanged and processed according to consecutive steps), to easily provide all logs to an organization higher-up (e.g. to the headquarters in case different business units of a company collaborate), to determine the exact contribution of each organization, etc. Finding an appropriate storage solution is examined in this article.

The remainder of the article is structured as follows. Section two presents recent related work integrating or extending the Hyperledger Fabric framework[3], which is a so called permissioned blockchain. Before elaborating on the proposed logging mechanism, section three presents the integration of generic web APIs to allow collaborating participants to explore their APIs dynamically. Section four then outlines possible topologies to store the logs and explains how the Hyperledger Fabric framework fits within this use case. Missing verification mechanisms are identified and further detailed in section five, after which the implemented proof of concept is explained and evaluation results for a specific setup is shown in section six. Finally, conclusions are drawn for the presented work and future research directions are described.

2 | RELATED WORK

This section presents related work building further on the Hyperledger Fabric framework[3 4]. The uptake of the Fabric framework is clearly visible, both in industry and academia. Applications have been proposed for various use cases, e.g. banking[5], transparency for personal data handling[6], global trading[7], Internet of Things[8], Community Mesh Networks[9], etc. Furthermore, there are research papers on the performance of the framework[10], possible performance improvements[11 12 13], risks related to the implementation of a smart contract[14], the mitigation of attacks by a malicious peer[15], etc. This enterprise framework provides a so called permissioned blockchain, which is a shared ledger where the writers of the system, i.e. the consensus participants, are formed by a restricted set of members. The transactions (TXs) are stored in a chain of cryptographically linked blocks and define the changes applied to key-value pairs (KVPs) stored in a separate database. Its execute-order-validate architecture, referred to as the consensus mechanism, is novel compared to other permissioned blockchains as TX execution, i.e. passing an input through the business logic of the application, and TX validation are separated from the consensus protocol for ordering[4]. This allows TX endorsement policies to be created in which it is defined which endorsing peers (EPs) need to execute and sign a TX before it is considered valid. The framework is also open-source, i.e. it can be extended in any way, and modular, e.g. new consensus protocols can easily be integrated. Furthermore, it allows general-purpose programming languages (GPPLs) to be used to code the business logic of the application into so called smart contracts without the need for cryptocurrencies, and provides identity management tools. Why Fabric is chosen for this problem, is further detailed in section 4.

The cross-organizational data exchange applications built on Fabric that can be found in literature serve different purposes, but it is possible to divide them into two categories. The applications in both categories have in common that they require the data sender and receiver to communicate via both on-chain and off-chain communication channels. However, applications in the first category cause the receiver to wait for off-chain data to be consumed until certain interaction with the blockchain is successfully completed[16 17]. This is what we refer to as a synchronous data exchange approach. A specific case of this situation is when physical instead of digital goods are exchanged, as the sender and receiver need to meet in person and acknowledge the handover on-chain before the actual package transfer happens[18]. The earlier proposed logging mechanism by the authors[19] is in a second category, as the data receiver can immediately consume the received data without having to wait for on-chain information to be received. This is referred to as an asynchronous data exchange approach. The goal behind this design choice

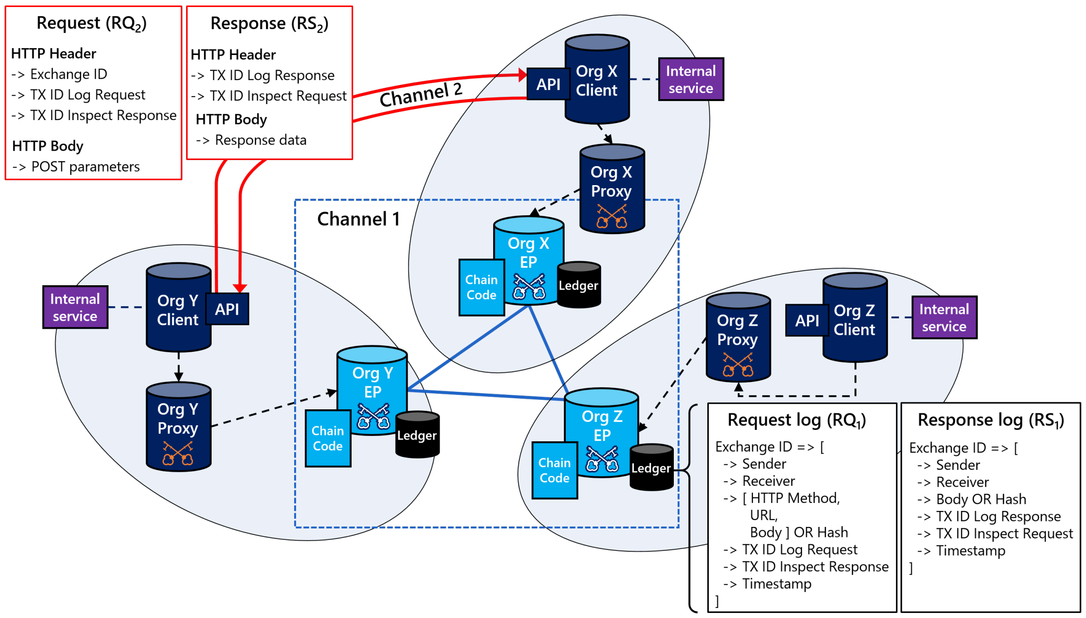


FIGURE 2 Recap of the components needed for the proposed logging mechanism.[19]

is to minimize the additional latency imposed by the logging process on the actual data exchange, at the cost of less logging certainty.

The goal of the proposed mechanism is to allow a data delivering organization to assess the integrity and correctness of the entire collaboration setup such that it can decide whether it wants to share its data with the other participants in a trustful way. Trustful means that every data exchange is logged, both request and response, in order to prevent possible disputes afterwards. Both the sending and receiving organization are incentivized to log properly: the sending organization may miss its compensation otherwise, while the receiving organization may lose access to the required data stream. There are thus only two direct stakeholders in providing appropriate data exchange logs. Furthermore, it is not an issue if any couple of organizations would agree on some fake logs to try to fool other organizations, as the actions of these indirect stakeholders are not influenced by these logs in a, for the dishonest couple, positive way.

Figure 2 shows the components of the architecture together with the content of the messages that are exchanged. The client components each expose an API to the partners in the collaboration and invoke other APIs. The proxy components receive logging requests from their corresponding client and invoke the EP to execute a function of the smart contract (=chaincode). Before Org Y sends an HTTP data request (RQ_2) via channel 2, it logs the request with the chaincode (CC) function `logRequest` (RQ_1) via channel 1. Note that TX IDs are sent along in the HTTP header to allow the counterparty to search for the expected TXs. When Org X receives the request, it queries its internal service, and sends the corresponding HTTP response (RS_2) via channel 2 and executes the CC functions `inspectRequest` and `logResponse` (RS_1) via channel 1. Finally, Org Y executes the CC function `inspectResponse` via channel 1 based on the received response. Note that the actual data request and response are replaced by hashes in RQ_1 and RS_1 respectively and that all interactions with channel 1 are executed asynchronously to prevent channel 2 from blocking. This flow thus generates four logging TXs for each data exchange in order to obtain an irrefutable proof of its existence and its integrity. This article explains the properties of the mechanism more extensively, applies the concept of generic web APIs as found in literature to the cross-organizational problem, and shows additional evaluation results.

3 | DATA AND SERVICE SHARING THROUGH EXPOSED WEB API FEATURES

Collaborating organizations make use of client-server relationships, i.e. Org Y sends a request to Org X, X processes the request, and sends a response back to Y. Each organization, willing to share services, exposes a web API to the partners in the collaboration to allow them to request data from internal resources. Authentication and authorization, although crucial in this scenario, are not further detailed here but will be treated separately in future work. The focus in this section is on the design of these APIs.

Each organization could define its own API using e.g. OpenAPI[20] to automatically generate server stub code, client library code and documentation. However, this should not be left to the organizations themselves, as it would result in a number of heterogeneous APIs being tightly coupled to different backend systems, imposing two severe drawbacks. First, an organization would need to study the documentation of each API to be able to use it. This is not possible when an ad hoc collaboration is initiated in case of an emergency situation. Second, even minor changes in the backend system of an organization could lead to API changes, resulting in many software updates to solve incompatibilities. It would instead be more useful to let connecting organizations, also called clients, couple to one or more reusable features. This feature-based vision, enabling automated clients, is extensively described by its founders Verborgh and Dumontier[21]. It is important to note that their vision is aimed at the entire web API ecosystem and that the selection and granularity of the different features needs to be agreed upon by the community. Their vision is applied here on a much smaller subset of this ecosystem, i.e. on cross-organizational collaborations. They propose five principles describing how a feature-based web could be realized:

- *"Web APIs should consist of features that implement a common interface."* The advantage this provides is that clients consuming the API can be generalized, i.e. an organization can connect or switch to (a subset of) features offered by different organizations without changing any code. Also, server-side interface code can be reused by multiple organizations.
- *"Web APIs should partition their interface to maximize feature reuse."* An organization should only define a new feature when there is no feature available which already provides the required functionality. As no common repository of features is available yet, a specific repository for cross-organizational collaborations can be defined to illustrate the concept.
- *"Web API responses should advertise the presence of each relevant feature."* The advantage this provides is that the offered API functionality can be discovered at runtime by any organization, which is necessary in case of a time-critical collaboration.
- *"Each feature should describe its own invocation mechanics and functionality."* Feature identification is one thing, using them is equally important. Communicated feature URLs can be dereferenced to find new API routes, request body formats in case of HTTP POST request, etc. This hypermedia constraint is important for the cross-collaboration scenario, as it allows organizations to explore how features can be used at runtime.
- *"The impact of a feature on a Web API should be measured across implementations."* This principle is not further discussed here, as the idea of using generic clients for cross-organizational collaborations is more important for this article than the exact composition of the feature set.

An illustrative setup is proposed in Figure 3. An Org X has three features available to share data of employees, image files and text files. Implementing a feature per resource type is a reasonable mapping as organizations will most likely share similar types of data and code can be reused. Furthermore, this separation allows an organization to easily start, stop and scale the sharing of individual resource types at runtime, as features are implemented as independent services. The labeled links in the figure are now discussed:

I The only information Org Y knows in advance about Org X is an entry point defined by a fixed URL, e.g. `example.org/collab`. The client component of Org Y uses this URL to send an initial HTTP GET request to the hypermedia-driven API exposed by the client of Org X. The JSON-LD[22] data format, being compatible with the Resource Description Framework (RDF)[23], is used to define the semantics of an organization's resources in plain JSON. The Hydra vocabulary, proposed by Markus Lanthaler, is furthermore used to allow dereferenceable Internationalized Resource Identifiers (IRIs) to be described[24]. This vocabulary consists of several classes and properties which are commonly found in web APIs. The base class of Hydra is the Resource class, from which all other classes inherit, and indicates that an IRI is dereferenceable and thus further information can be retrieved[24]. The code snippets below show how the proposed API can be dressed up with keywords from this vocabulary. This vocabulary provides a useful addition to RDF, as the latter uses IRIs for identification without any description related to their dereferenceability, forcing clients to blindly dereference each IRI in order to discover whether further information can be retrieved[24].

II When features are deployed, they first register themselves with the entry point of the API, i.e. the collaboration feature. This way, it is possible for the collaboration feature to have an up-to-date overview of all enabled features. When queried, an HTTP response is sent to the client of Org Y which contains an overview of all enabled features for which it is authorized.

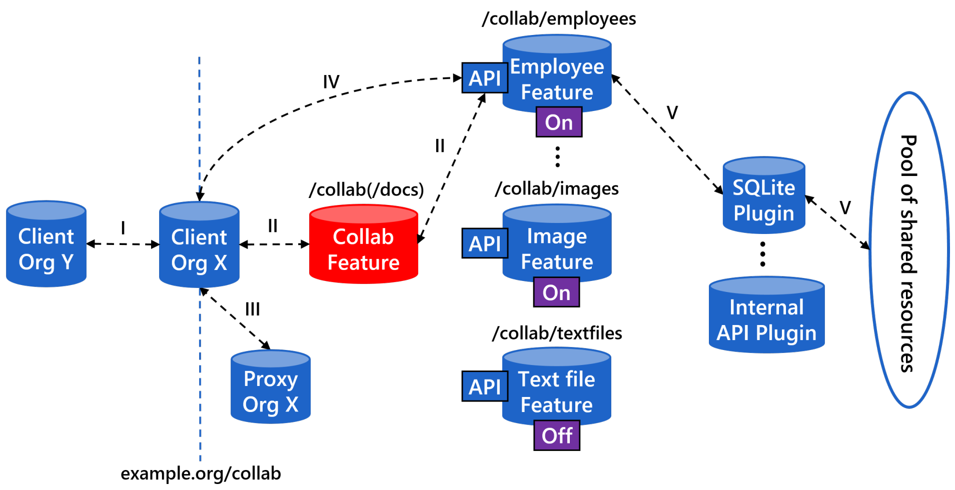


FIGURE 3 The proposed web API consisting of features which can be reused by multiple organizations.

An example definition of the entry point is shown in Listing 1. Note that the listings presented in this section are strongly based on the demo code of the generic client Hydra Console[25].

```
{
    "@context": {
        "prefix": "http://example.org/collab",
        "docs": "prefix:/docs#",
        "EntryPoint": "docs:EntryPoint",
        "employees": {
            "@id": "EntryPoint:/employees",
            "@type": "@id"
            => Property value is an IRI
        },
        => Analogous for other features
    },
    "@id": "prefix:",
    "@type": "EntryPoint",
    "employees": "prefix:/employees",
    => Analogous for other features
}
```

Listing 1: Entry point of the Web API

Each HTTP response furthermore contains a URL `/collab/docs`, stored in the HTTP Link header[24], allowing the receiving organization to send a web API documentation request *D*. When such a request is received by the collaboration feature of the sending organization, a documentation is constructed by querying all active features and merging their documentations into one response. The merged documentation used in this work is presented in Listing 2. The merge in this approach boils down to the aggregation of the supported classes and supported properties from the different features as indicated in the comments in the code below.

```
{
    "@context": {
        "prefix": "http://example.org/collab",
        "docs": "prefix:/docs#",
        "hydra": "http://www.w3.org/ns/hydra/core#",
        "ApiDocumentation": "hydra:ApiDocumentation", ...
    },
    "@id": "prefix:/docs",
    "@type": "ApiDocumentation",
    "supportedClass" : [
        {
            "@id": "docs:EntryPoint",
                "@type": "Class",
                "title": "The main entry point of the API",
                "supportedOperation": [
                    {
                        "@type": "Operation",
                        "method": "GET",
                        "description": "Retrieves the entry point of the API.",
                        "returns": "docs:EntryPoint",
                        "possibleStatus": [
                            {
                                "description": "The entry point was retrieved successfully.",
                                "statusCode": 200
                            }
                        ]
                    }
                ],
                "supportedProperty": [
                    {
                        // Define a new property
                        "property": {
                            "@id": "docs:EntryPoint/employees",
                            "@type": "Link",
                            => Property value is a dereferenceable IRI
                            "domain": "docs:EntryPoint",
                            => Property is found in Entrypoint instances
                            "range": "docs:EmployeeCollection"
                            => IRI points to an EmployeeCollection instance
                        },
                        "title": "employees",
                        "description": "Link to the collection of employees of Org X"
                    },
                    => Analogous properties for other features (aggregation)
                ]
            }, {
                "@id": "docs:Employee",
                "@type": "Class",
                "subClassOf": "schema:Person",
                "title": "An employee of Org X",
                "supportedOperation": [ => Similar as above
                ],
                "supportedProperty": [ => e.g. name, employer, department, IP address, etc.
                ]
            }, {
                "@id": "docs:EmployeeCollection",
                "@type": "Class",
                "subClassOf": "Collection",
                "title": "The collection of employees of Org X",
                "supportedOperation": [ => Similar as above
                ],
                "supportedProperty": [ => e.g. number of items, members, etc.
                ]
            },
            => Analogous classes for other features (aggregation)
        ]
    }
```

Listing 2: Documentation of the Web API

III The client component instructs the proxy component to execute the CC functions according to the steps of the proposed logging mechanism.

IV The requests sent by the client of Org Y are authenticated and authorized by the client of Org X after which they are forwarded to the correct feature. As the resulting web API is Hydra-compliant, it can be consumed by the aforementioned Hydra Console[25]. When a collaboration is set up, this console can be used by every organization to explore such an API in an interactive way, i.e. by displaying dynamically constructed HTTP request forms, by showing HTTP responses and the corresponding documentation.

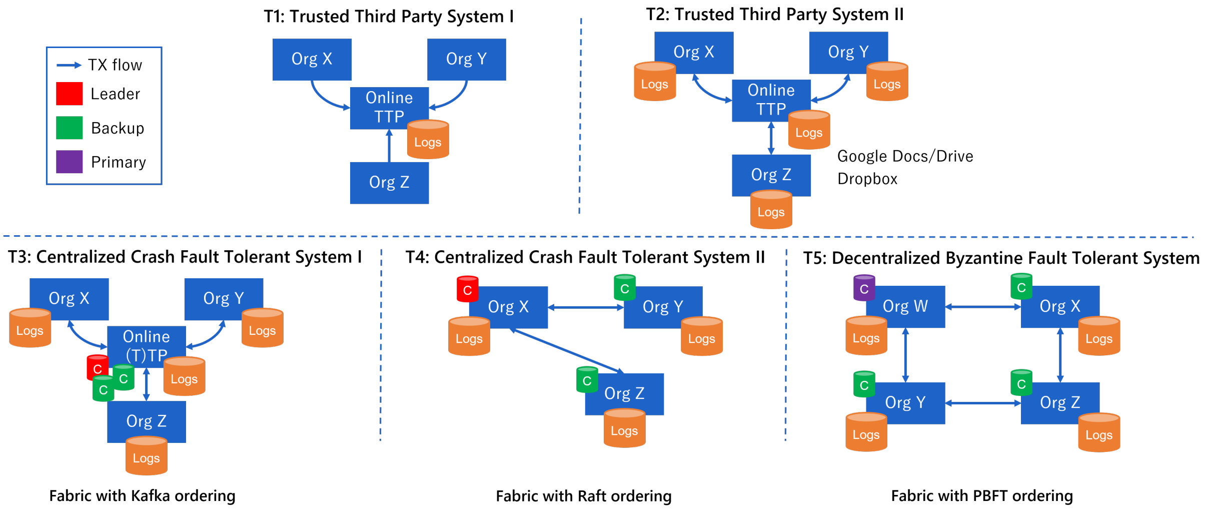

TX flow
Leader
Backup
Primary
T1: Trusted Third Party System I
Org X
Org Y
Online TTP
Logs
Org Z
T2: Trusted Third Party System II
Google Docs/Drive
Dropbox
T3: Centralized Crash Fault Tolerant System I
Online (T)TP
C
Fabric with Kafka ordering
T4: Centralized Crash Fault Tolerant System II
Fabric with Raft ordering
T5: Decentralized Byzantine Fault Tolerant System
Org W
Fabric with PBFT ordering

FIGURE 4 Overview of possible topologies discussed in this article.

V The constructed web API provides a level of abstraction as the client of Org Y does not know whether it is actually communicating with an internal API, database, etc. Writing code, called plugins, to map the functionality of the generic features to the company-specific internal services is the responsibility of Org X. This requires coding effort upfront, e.g. the connection with an SQL database, before a collaboration can be initiated. Once these plugins are implemented, an organization only has to provide configuration settings such as file path, URL, port, type of database, credentials, etc. An additional advantage of the feature based approach could be the reuse of plugin implementations amongst organizations. On the one hand, an organization could copy a full plugin implementation and use the same initial internal API / database structure. On the other hand, plugins could be implemented in a more generic way, e.g. by searching for matching tables, columns and/or records in metadata based on feature related attributes.

4 | STORING LOGS OF DATA EXCHANGES

As discussed in section 1, a database with shared write access is needed in order to store the logs generated by the collaborating organizations. An overview of possible topologies is given in Figure 4. First, topology T1 and T2 are discussed, which both rely on a trusted third party. The applicability of a blockchain solution is then investigated, after which the topologies T3, T4 and T5 integrating the permissioned blockchain Hyperledger Fabric are discussed.

4.1 | Trusted third party

The first solution, visualised by topology T1, is to delegate the logs to an online trusted third party storage solution provided by companies like Amazon, Google, Microsoft, IBM, etc. As all logs are stored at that single organization, there are two requirements for this topology to work. First, a third party offering a Platform as a Service (PaaS) / Software as a Service (SaaS) solution needs to be found which is trusted by all collaborating organizations. Although this might be difficult, it is not unlikely. Second, the organization operating as the admin of the cloud solution also needs to be trusted as it has the power e.g. to provide different views of the database to the different organizations. The second solution, visualised by topology T2, is to use a master-slave topology, i.e. where one organization takes the lead in processing updates and distributing them to the others. Obvious solutions in this context are e.g. Google Docs / Drive or Dropbox to share a file or folder among authorized organizations. Without any countermeasures, this solution also requires all organizations to agree on a trusted third party and on a trusted share owner, as both have the power to e.g. delete the share and corresponding file activity at any point in time. Trust is the key requirement for both topologies and only when this requirement is fulfilled, a suitable storage solution is found. For the general collaboration case, it is however not possible to make assumptions about trust, so alternative topologies are to be examined.

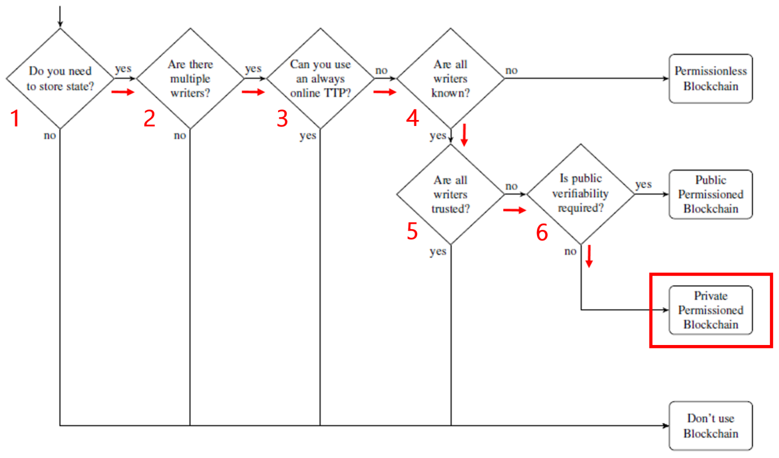

Do you need to store state?
yes
no
1
Are there multiple writers?
yes
no
2
Can you use an always online TTP?
no
yes
3
Are all writers known?
no
yes
4
Permissionless Blockchain
Are all writers trusted?
no
yes
5
Is public verifiability required?
yes
no
6
Public Permissioned Blockchain
Private Permissioned Blockchain
Don't use Blockchain

FIGURE 5 Flow chart to assess whether a blockchain solution may add value to an application.[26]

4.2 | Blockchain applicability

Due to the potential impossibility of finding a trusted third party, a blockchain solution can be applied to this problem. This observation can be verified using the commonly used flow chart in Figure 5. The steps of the chosen path are discussed below:

1. The organizations need to store state, i.e. the different logs, in a structured way.
2. As each organization wants to store its logs, multiple writers are present.
3. As is clear from section 4.1, it might be difficult to find an online trusted third party to delegate the logs to. An offline trusted third party acting as a certificate authority could potentially be found.
4. The set of writers, i.e. the collaborating organizations, is known at each time. The on-boarding process is guided by a set of admins which grant or revoke permissions for writers.
5. As explained in section 1, the trust assumption between the collaborating organizations might not hold. It is therefore needed to identify each of the known writers using a public key infrastructure to be able to guarantee authentication and data integrity for each action taken by any of the writers. This way, malicious behavior can easily be traced.
6. The set of readers is also restricted, as the collaborating organizations are the only involved stakeholders.

The result of the analysis is that a private permissioned blockchain solution may be utilised to solve the issue at hand. Note that this type of blockchain does not meet the criteria of a public blockchain[27]: a blockchain is a digital ledger which is (1) decentralized, i.e. independent entities are involved in the consensus process, (2) immutable or permanent, i.e. it is impossible to revert previous state transitions and (3) transparent, i.e. anyone can verify the correctness of the ledger. A permissioned blockchain should be fully decentralized, but this is not the case for Fabric (version 1.0-1.4) as will be explained in the next section. The immutability guarantee always needs to be evaluated against the probability of a certain scenario happening and is not a binary guarantee, as extensively described by Greenspan[28]. Assume that at least $\geq 50\%$ of the organizations unanimously create a fork because they do not agree with the current version, e.g. two out of four organizations do this, it is unknown which ledger should be seen as the correct one, as there exists no majority for any chain. As the number of copies of the chain is rather low in a permissioned enterprise setup, it is more prone to such an attack. Finally, a private permissioned blockchain is not transparent as the set of readers is restricted.

4.3 | Hyperledger Fabric

A key property of each distributed database is the chosen consensus mechanism in order to keep the different copies of the ledger in-sync. Blockchain architectures in particular should be able to cope with Byzantine faults[11]. Byzantine Fault Tolerant

TABLE 1 Overview of ordering service implementations for Fabric version 1.0-1.4.

Ordering service	CFT	BFT	Max. malicious consensus nodes
Solo (only one node)	-	-	0
Kafka cluster (centralized consensus in a single domain)	X	-	0
Raft[32] (centralized consensus over multiple domains)	X	-	0
PBFT proposal[33] (decentralized consensus)	X	X	x when #nodes $\geq 3x + 1$

(BFT) systems are able to reach consensus on the state of the ledger among the honest nodes in the presence of faulty nodes. The general purpose of this class of algorithms is to provide safety and liveness guarantees by masking Byzantine faults, e.g. a lost TX due to a network error or due to a malicious node. The safety guarantee implies that a distributed system behaves like a centralized one from the viewpoint of a client, while the liveness guarantee implies that clients of the system will eventually receive a reply to their requests[29]. These guarantees only hold under certain assumptions. The first assumption is a (partially) synchronous network, as the FLP theorem[30] proves that it is impossible to reach consensus in a fully asynchronous network when there is only one faulty process. Second, for e.g. the Practical Byzantine Fault Tolerance (PBFT) protocol, consensus is possible as long as there are at most f number of faulty nodes in a set n of size $3f + 1$. This means e.g. that when two out of four consensus nodes are faulty, the ordering system already malfunctions. Facebook's LibraBFT consensus mechanism[31] is a recent example of a BFT protocol deployed on a larger scale, i.e. $n = 100$ and thus $f = 33$. It is clear that, the more consensus participants there are in the ordering system, the more fault tolerant it becomes.

The consensus mechanism used in Fabric consists of three phases: simulation of TX execution, TX ordering and TX validation. As already mentioned, the separation of TX execution and validation from the consensus protocol for ordering allows an application-specific endorsement policy to be specified defining which trust assumptions hold, i.e. which EPs need to endorse a TX in order for it to be considered valid. This policy allows to prevent Byzantine behavior on the application level, at least when the policy requires endorsements from multiple EPs to be available when the TX is validated. The ordering of TXs in Fabric is done by the so called ordering service which basically collects TXs and packages them into blocks. At the time of writing this article, Fabric has not officially released a BFT ordering service as is visible in the overview of Table 1. This thus means that this phase is not decentralized and consequently that the consensus mechanism is not fully decentralized. Fabric advertises itself as a permissioned blockchain solution, but the versions 1.0-1.4 do not fully meet the expected requirements yet.

Topology T5 displayed in Figure 4 is thus not possible yet, and a master-slave topology using a crash fault tolerant (CFT) cluster of Kafka brokers, which is topology T3 in Figure 4, needs to be used instead. A second option is to use topology T4 which uses an implementation of the Raft protocol[32]. As this algorithm also uses a leader node to dictate the commands to its followers, the only difference is that it is more crash fault tolerant as the consensus nodes can be deployed at different organizations in the network. Topology T3, which is examined in the remainder of this article, thus has no decentralized consensus mechanism for ordering TXs. Given this constraint, it is needed to find a solution in which the third party, which hosts the crash fault tolerant ordering service, does not need to be fully trusted. This third party could be an external organization, e.g. one which provides a SaaS solution for Fabric, or one of the participating organizations in case one of them is the initiator of the collaboration, e.g. the operator in a control room scenario as show in Figure 1. An important observation is that masking faulty behavior for the ordering phase is not necessarily required for this application. The goal is to allow each organization to detect faulty behavior, as will be described in section 4.4.3, such that it can pause its data sharing with a specific organization or even its participation in the collaboration and it can assess whether it still wants to be part of the current cooperation. Although faulty behavior will only be detected once the faults wrongly influence the state of the ledger, i.e. safety is not guaranteed at all, it is not a crucial issue for this application as it should only provide a way to verify whether the collaboration setup can be trusted. The requirements for this application are thus less strict compared to these of critical BFT systems for which any propagation of faults may be disastrous.

One could argue why a blockchain framework like Fabric, given the restriction of topology T3, would be used over already existing distributed database technologies. The differences between both systems and the reasons why Fabric is used for this use case are listed below:

- Fabric provides identity management out of the box, allowing TXs to be cryptographically signed and to be tamper proof. Every state update can thus be linked to an individual organization. Furthermore, as the authentication and integrity of each block is guaranteed by a digital signature of one of the ordering service nodes (OSNs), it is impossible for malicious entities to tamper blocks and to convince others of having the correct version of the ledger, as they do not have corresponding block headers signed by an OSN.

- A traditional CFT system, using leader-backup replication, requires a single entity to be responsible for the execution of all TXs, which is not desirable due to the earlier mentioned trust issue. A traditional BFT system, e.g. using PBFT consensus, requires all organizations to execute all TXs sequentially, which causes an unnecessary performance penalty and a problem with confidentiality[4]. Fabric provides a hybrid approach through the endorsement policy, which is further detailed in section 4.4.2, allowing the execution and validation of TXs to be parallelized.

- The chain data structure in blockchain systems like Fabric and the recovery log as used for traditional databases have similar properties, i.e. they are both append-only and describe state updates[34]. However, there are also four differences, as identified by Mohan[35]: read operations are often also recorded in Fabric, a recovery log could be truncated when databases are backed up, log records of multiple TXs are interspersed in a recovery log whereas all information related to a single TX is packed together in a blockchain, and a hash chain is not used in a recovery log as Byzantine faults are not considered.

- The purpose of chaining blocks is completely different in the case of Bitcoin, being the first generation blockchain, compared to Hyperledger Fabric. Assume an attacker in Bitcoin wants to tamper history, i.e. wants to take its previously spent money back[36], and tries to mine this adapted block. The expected number of hash calculations C that needs to be executed to mine a block with difficulty D in Bitcoin is[37 38]:

  $$C = \frac{1}{P(validhash)} = \frac{D \cdot 2^{256}}{0xFFFF \cdot 2^{208}} = \frac{D \cdot 2^{48}}{2^{16} - 1}$$

  The expected time T in seconds to mine a block, when hardware with a hashrate H is used, therefore equals:

  $$T = \frac{C}{H} \approx \frac{D \cdot 2^{32}}{H}$$

  At the time of writing, $D \approx 7.93 \cdot 10^{12}$. When an attacker would use one GPU, e.g. the Nvidia GTX680 with 120 Mhash/s[39], it would take about 9 million years to mine a valid block. This shows that an attacker would need a significant amount of hashrate to do the proof-of-work for one block in a reasonable amount of time. Assume the attacker can control such an amount ($\leq$ 50% of the network hashrate) or is lucky and mines a block much faster. This is not a problem as, due to the longest chain principle, all succeeding blocks x after the tampered block need to be re-mined in order to become the longest chain. The original paper[40] shows that the probability for such an attack to occur decreases exponentially when x increases, assuming the hashrate of the attacker is not more than half the network hashrate. Note that this assumption is crucial, as otherwise a private fork can be generated faster than the longest chain. Although this assumption seems to be valid due to the size of the Bitcoin network, it may not be true as mining is dominated by a few large pools[41]. Nevertheless, the goal of the chain of blocks is to lower the probability of double spending and to increase the probability of immutability.

  In Fabric however, forks are not possible assuming a correct operation of the ordering service, as the consensus mechanism is deterministic, and double spending is solved by the read-write conflict check at validation phase[4]. Thus, the presence of the chain does not change the difficulty with which the consensus protocol for ordering could be attacked. However, the purpose of chaining blocks is only to allow the peers to audit the integrity of the sequence of blocks more efficiently[4]. Another efficiency advantage of the chain is discussed in section 4.4.3.

- Smart contracts have similar properties as stored procedures in traditional databases[42]. However, Fabric allows the business logic defined in smart contracts to be written in GPPLs such as Go, while stored procedures are often written in a SQL dialect. This allows the database for the KVPs to be freely chosen in Fabric.

- When a scalable BFT ordering service is provided by Fabric in the future, it can easily be swapped in, as ordering is totally separated from other parts of the architecture. Switching to a decentralized ordering service will make malicious behavior within the ordering phase harder.

- Fabric is open source and the components can easily and rapidly be deployed using Docker containers.

4.4 | Validation mechanisms

This section investigates which validation mechanisms are required for the discussed use case. First, validation mechanisms which are already present in the code of the EPs are identified. Second, the validation of TXs according to the endorsement policy is discussed. Finally, two additional mechanisms, which are not yet present in Fabric, are proposed.

4.4.1 | Block validation

At a certain point in time, the ordering service creates a block of TXs and attaches a block header which contains the block number `number`, the hash of the header of the previous block `previous_hash`, and the hash of the data included in the current block `data_hash`[43]. This block header is signed with the private key of one of the OSNs and the corresponding signature, public key and certificate are attached to the metadata section of the block. When a block is received by a Fabric peer, authentication and data integrity are verified using the digital signature and the `data_hash` field is checked for correspondence with the TXs in the block[44]. This last check is important as it allows a peer to detect whether a malicious OSN sends a chain of block headers which does not match with the actual content of the blocks. When these checks are validated, the peer performs two other checks before it adds an incoming block to its ledger:

1. The code shown in Listing 3 only processes a new block when its number corresponds with the current height of the local chain. This check prevents TXs from being executed multiple times, e.g. when a block with number x is replayed, it guarantees that blocks are processed in the correct order, and it prevents already received blocks from being overwritten by a tampered version.

```
func (mgr *blockfileMgr) addBlock(block *common.Block) error {
    bcInfo := mgr.getBlockchainInfo()

    if block.Header.Number != bcInfo.Height {
        return errors.Errorf(
            "block number should have been %d but was %d",
            mgr.getBlockchainInfo().Height,
            block.Header.Number,)
    } ...
}
```

Listing 3: Check one of block storage functionality[45] in Fabric version 1.3.

This check should go one step further. Each received block with height x needs to be exactly the same due to the finality property[46] of Fabric. This property guarantees that forks are not possible when the ordering service is honest as it operates deterministically. This means that, when a block with number x is received for which a different block with number x was already received, an attacker controlling one ore more OSNs might try to tamper the history of the logs. Instead of only reporting an error in that case, an organization may decide to (temporarily) withdraw itself from the collaboration.

2. The code shown in Listing 4 focuses on the integrity of the chain: the hash, set by an OSN in the header of an incoming block, at height x (`PreviousHash`) needs to be the same as the hash of the header of the latest received block at height $x-1$, as calculated by the EP of Org X (`CurrentBlockHash`). Both hash values are calculated over the ASN.1 encoded block header[47]. This check thus verifies that a valid chain is received and that no operational mistakes are made by an OSN.

```
func (mgr *blockfileMgr) addBlock(block *common.Block) error {
    ... if !bytes.Equal(block.Header.PreviousHash, bcInfo.CurrentBlockHash) {
        return errors.Errorf(
            "unexpected Previous block hash. Expected PreviousHash = [%x], PreviousHash
referred in the latest block= [%x]",
            bcInfo.CurrentBlockHash,
            block.Header.PreviousHash,)
    }
    blockBytes, info, err := serializeBlock(block) ...
}
```

Listing 4: Check two of block storage functionality[45] in Fabric version 1.3.

4.4.2 | TX validation

Individual TXs are validated against the endorsement policy. This policy is coupled to the CC and defines which EPs, one deployed at each organization, need to simulate and sign a TX proposal by evaluating the CC deployed in their local Docker container. The policy chosen for this application is that only one organization needs to sign the proposal in order to be valid. The reason for this is twofold. On the one hand, each ledger update an organization triggers is signed, meaning it can be held responsible for this update. This means that, when an organization removes the logging information entered by another organization, it is immediately clear which organization is responsible for this undesirable behavior. On the other hand, the data exchange and logging process are separated, in order to avoid latency for the exchange[19]. Conceptually, two layers can be identified, the application layer and the blockchain layer. Due to this, it is impossible to choose an alternative policy requiring the two organizations involved in a data exchange to endorse a single TX, as the actual data exchange can only be known by the EPs when it is communicated top-down, i.e. from the application layer to the blockchain layer. Creating a bottom-up dependency, i.e. from the blockchain layer to the application layer, is not desirable as the logging process should be abstracted as much as possible. A consent by the counterparty to confirm the correctness of a log proposal is thus not directly provided via an endorsement in the same TX, but through a separate TX generated using the inspect CC functions. This approach thus leads to four TXs per data exchange as mentioned in section 2.

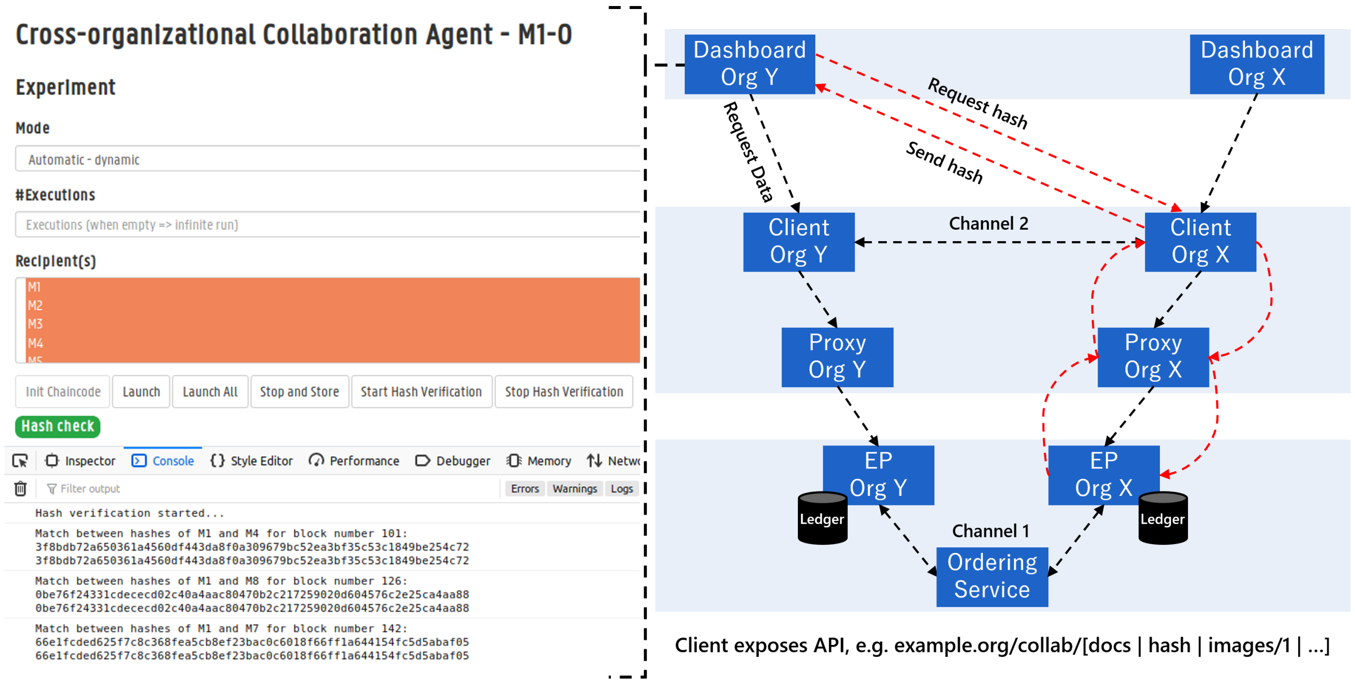
4.4.3 | Validation collaboration setup

The validation mechanisms yet present in Fabric are not enough to check for any faulty behavior in this application. Two additional checks are required to be able to verify the correct operation of the entire collaboration setup[19].

- As each organization can store the logs it wants and as TXs can get lost, intentionally (e.g. malicious ordering service) or not (e.g. network failure), it is needed for each organization to verify the presence and integrity of all expected logs. It is important to note that the involved organizations should obtain complete logging cycles, i.e. both for request and response, as external parties are not able to know what exactly happened during their private communication without those logs. This is a consequence of the asynchronous approach, implemented using two channels, to limit the delay on the data exchange caused by the integration of the logging mechanism. The implementation of the first check was already done, as the CC functions `inspectRequest` and `inspectResponse` were defined for this purpose. These functions allow an organization to inspect the logs, which are stored by the counterparty using the CC functions `logRequest` and `logResponse`, in order to verify whether they match with the actual data exchange. The implementation of the smart contract is straightforward, i.e. is mainly based on getters and setters, as is partially illustrated in Listing 5.
- As a malicious ordering service may create a separate fork for each organization satisfying check one, it is needed to verify whether this service replicates the data correctly. As the hash `previous_hash` stored in the block header at height x provides a summary of all TXs that modified the database from the genesis block up to and including the TXs of block $x-1$, it is possible for the organizations to perform an efficient check among each other. They only have to exchange the `previous_hash` value together with the corresponding block height to know whether they share the same state. As already mentioned in section 4.3, the main purpose of chaining blocks is to allow for efficient auditing techniques.

```go
func (t *Collab) inspectResponse(stub shim.ChaincodeStubInterface, exchangeId string, data string)
    pb.Response {
    // (Step I) Unpack string by Org Y to Response object and hash it
    response := Response{}
    err := json.Unmarshal([]byte(data), &response)
    if err != nil {
        return shim.Error(err.Error())
    }
    responseString := fmt.Sprintf("%v", response)
    sum := sha256.Sum256([]byte(responseString))
    hashResponse := hex.EncodeToString(sum[:])

    // (Step II) Get log stored by Org X
    logByte, err := stub.GetState(exchangeId + "_response")
    if err != nil {
        return shim.Error(err.Error())
    }
    // ... and unpack string by Org X to Response object => loggedResponse
```



```
Hash verification started...
Match between hashes of M1 and M4 for block number 101:
3f8bdb72a650361a4560df443da8f0a309679bc52ea3bf35c53c1849be254c72
3f8bdb72a650361a4560df443da8f0a309679bc52ea3bf35c53c1849be254c72
Match between hashes of M1 and M8 for block number 126:
0be76f24331cdececd02c40a4aac80470b2c217259020d604576c2e25ca4aa88
0be76f24331cdececd02c40a4aac80470b2c217259020d604576c2e25ca4aa88
Match between hashes of M1 and M7 for block number 142:
66e1fcded625f7c8c368fea5cb8ef23bac0c6018f66ff1a644154fc5d5abaf05
66e1fcded625f7c8c368fea5cb8ef23bac0c6018f66ff1a644154fc5d5abaf05
```

FIGURE 6 Cross-organizational hash verification request which allows organizations to assess whether they share the same state up to some point.

```
    // (Step III) Compare hash of log reported by Org X with hash of response observed by Org Y
    if loggedResponse.Hash == hashResponse {
        return shim.Success(nil)
    } else {
        return shim.Error("POSSIBLE FRAUD: Hashes of responses Org X and Org Y did not match!")
    }

}
```

Listing 5: Snippet of the proposed smart contract.

5 | EXTENSION OF THE LOGGING MECHANISM

The implementation of the second check outlined in section 4.4.3 has not been discussed before. Figure 6 shows a high level overview of how it is integrated in the architecture. The idea is to extend the API of each organization with a `/hash[/blocknumber]` endpoint, such that the hash of the header of a specific block x can be requested by any of the collaborators. Each Org Y then randomly picks another Org X each I seconds and sends a hash verification request H to it. As such, it is possible to verify whether the same chain of block headers is received by both organizations up to that block number.

The `queryBlock` function of the Node.js Software Development Kit (SDK)[48] of Fabric can be used to retrieve the full block located at a certain height. The hash of the header of a block x can either be found as `previous_hash` in block $x+1$, but as it is not sure whether block $x+1$ exists, the hash value for block x is calculated separately in the same way as mentioned previously, i.e. using the ASN.1 encoded block header. It is important to note that this hash verification check can be turned on or off, depending on the needs for a specific application. When for example a BFT ordering service is used instead of a CFT ordering service, an organization might find it sufficient to trust the assumption(s) under which the service operates. In our proof of concept, this code is written in JavaScript and executed periodically using the `setInterval` function, and it runs in the browser when the user clicks the verification button in the dashboard of the collaboration.

As the goal is to allow Org Y to specify a specific block number B for which Org X needs to send its hash, it is needed for Org Y to know which blocks are already received by Org X. Finding a general solution to solve this problem is difficult, as there may be a(n) (accumulating) difference in block processing latency, e.g. due to a difference in network latency between the ordering service and each of the organizations. A more specific solution can however be found when the interaction of the different logging functions is used. To be able to explain this, it is needed to show the code in Listing 6. The `listenToEvent` function receives a set of TX IDs and returns a promise which only resolves when the local ledger of the EP contains all corresponding TXs. The different TXs are looked up concurrently, by scheduling the execution of the `lookUpTx` function each two seconds. In order to search more efficiently, block indications are exchanged between Org Y and X, i.e. the RQ_2 and RS_2 messages as shown in Figure 2 are extended with block indication headers. This means that, for each data exchange, an organization receives a block indication (`receivedBlockIndication (RBI)`) and sends a block indication (`sentBlockIndication (SBI)`). These values are only communicated to make searching for TXs more efficient, i.e. instead of traversing all blocks starting from genesis until the TX is found, these lower limits can be used to skip blocks for which it is certain that the looked for TX is not in there. The values of these indicators are determined in the client code, just before the `logRequest` and `logResponse` functions are invoked as shown in Listing 7 and 8 respectively, and correspond with the number of the latest inspected incoming block + 1. The client receives updates on this block number from the proxy, as each time a logging cycle is completed, a block number is communicated back from the proxy to the client, as is visible in the `waitForCycle` function.

```
// Map filled with blocks, based on block events sent by EP
var blocks = new Map();

// FUNCTION I
function lookUpTx(txId, index, resolve, reject) {
    while (index <= latestReceivedBlockNumber) {
        var block = blocks.get(index);

        if (block) {
            var txs = block['filtered_transactions'];
            var result = txs.findIndex(tx => tx['txid'] === txId);
            if (result > -1) {
                return resolve(index); // Return number of block in which TX is found
            }
        }

        index++;
    }

    setTimeout(function() { lookUpTx(txId, index, resolve, reject); }, 2000);
}

// FUNCTION II
function listenToEvent(txIds, RBI) {
    var promises = [];

    for (var i = 0; i < txIds.length; i++) {
        promises.push(new Promise(function(resolve, reject) {
            var txId = txIds[i];
            setTimeout(function() {
                lookUpTx(txId, RBI, resolve, reject); // FUNCTION I
            }, 2000);
        }));
    }

    return Promise.all(promises);
}

// FUNCTION III
function waitForCycle(txIds, RBI, SBI, ...) {
    // Wait until inspect TXs are received (i.e. the logging cycle is completed)
    listenToEvent(txIds, RBI) // FUNCTION II
    .then((blockNumbers) => {
        // Reply to client max(blockNumbers)

        // Store SBI for future hash verification requests with counterparty
    });
}
```

Listing 6: Proxy code which is needed to wait for logging cycles to be completed.

This `waitForCycle` function lets both Org Y and X wait until a logging cycle is completed, i.e. when their copy of the ledger contains the two inspect TXs, of which the IDs are passed via the `txIds` parameter. The solution to know which block numbers are already received by the counterparty, can be found in this function. When a logging cycle is completed, the inspect TX generated by the counterparty is received, meaning that the counterparty should have received the block containing the original logging TX. The block indication sent to the counterparty (`SBI`) thus provides a trustful sharp lower limit for the block numbers in which this original logging TX can be stored. When the time has come to execute a hash verification check, this stored block number can then be used in the query to the selected organization to report its hash value corresponding with that number. The only assumption for this to work properly is that couples of organizations exchange data. As long as this is true, they will learn the number of blocks received by other organizations, allowing it to perform hash verification checks and to gain confidence in the correct replication of the logging data.

For the logging mechanism to work properly, it is needed to enforce organizations to follow the proposed steps. The data delivering organization X sets two protections for this purpose, i.e. it refuses requests R from Org Y when (Protection I) there is an unanswered hash verification request, as explained above, sent to Org Y and (Protection II) the difference between the number of data responses sent to Org Y and the number of corresponding logging cycles by Org Y exceeds M. These protections incentivize Org Y to behave as expected by Org X, as it could lose access to the data of Org X otherwise. The first protection prevents Org X from not being able to compare the state of its local ledger with those of others and to be fooled by a malicious ordering service. Org X should repeat this hash verification request until it is fulfilled. The second protection prevents Org X from sending its data without receiving any corresponding log from Org Y. In case this scenario occurs, Org X should repeatedly send the exchange IDs which it expects to be re-executed by Org Y, in order to obtain complete logging cycles for these exchanges. Each Org X can set its own value $M \geq 0$, even per Org Y. The selection of this value is a trade-off between performance and security, i.e. a high value allows more data exchanges per time interval but provides less logging guarantees and vice versa.

6 | EVALUATION PROOF OF CONCEPT

A proof of concept of this logging mechanism is developed, deployed and evaluated. The details of the evaluation phase are given in the following sections.

6.1 | Data exchange model

In order to improve upon the fixed evaluation scenario where only one organization requests data from the other $O-1$ organizations[19], a Markov chain can be used to simulate generic pseudorandom collaboration scenarios. Figure 7 shows the four states in which each organization can reside when the collaboration consists of four participants. The red arrows indicate the probability x of moving to the sleep state in which an organization does not request any data. The blue arrows indicate the probability y of remaining in the same state. In general, an organization will likely try to query the same organization multiple times in a row, meaning that this probability needs to be different than the one of the black arrows. The state transition matrix $P_{i,j}$, defining the probabilities of moving from state i to state j, is a right stochastic matrix and equals for a collaboration of $O > 2$ organizations:

$$P_{i,j} = \begin{bmatrix} x & \frac{1-x}{O-1} & \frac{1-x}{O-1} & \dots & \dots \\ x & y & \frac{1-x-y}{O-2} & \frac{1-x-y}{O-2} & \dots \\ x & \frac{1-x-y}{O-2} & y & \frac{1-x-y}{O-2} & \dots \\ \dots & \dots & \dots & \dots & \dots \end{bmatrix}$$

The Markov chain determines the model that is used for each organization deciding on the next organization to send a request for data to. It is however also required to define which data needs to be requested. The experiments consider two different data sources: (1) an image feature mapping to a REST API written in Python and (2) an employee feature mapping to an SQLite database. The former data source provides three base64 encoded images with an original size of 75, 21 and 45 KiB, while the latter data source is a database file with size 88 KiB consisting of one thousand records of dummy employees. Both data sources are spawned at each organization such that they can be used in the data exchange process. In order to realize a fully autonomous collaboration scenario, the implemented generic client needs to execute a number of steps automatically based on the response sent by Org X. Although multiple approaches are possible, the following steps are sufficient for our purpose:

1. The JSON-LD expansion algorithm is applied to the response of Org X

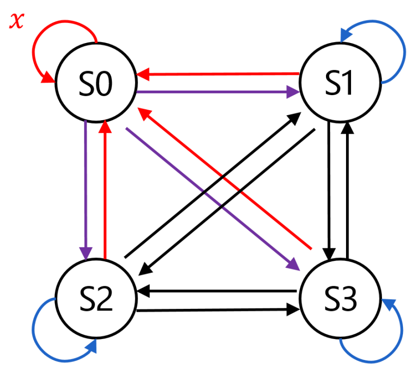

x
y
S0
S1
S2
S3
S0: Sleep Org Y - S1: Request to Org P - S2: Request to Org Q - S3: Request to Org R

FIGURE 7 Markov chain for an Org Y participating in a collaboration with $O = 4$.

2. The type of the resource is inspected and looked up in the supported class section of the API documentation
3. One of the supported class properties of type Hydra Link is randomly selected, as the values of these properties are known to be dereferenceable IRIs
4. Org Y executes an HTTP GET request using the IRI of the selected property

This way, the execution trace of a client will be extended indefinitely. Note that this sequence of requests is constructed step by step per organization, meaning that Org Y postpones new requests P until it has received a response from Org X. The steps can be illustrated using the code listings shown in section 3: Org Y retrieves the entry point of Org X and discovers, using the documentation, that the value of the `employees` property is a dereferenceable IRI which points to an instance of the class `EmployeeCollection`. Detailed information on this GET request and any other supported operations are found in the `supportedOperation` property of that class. Note that for this evaluation resources are only retrieved and not created, updated or deleted, which covers most cross-organizational collaborations. When the `EmployeeCollection` is received, the documentation can again be used to discover its supported properties. Only the `hydra:member` property is eligible as it is an instance of the Hydra Link class. Finally, when an `Employee` is received, no further dereferencing is possible, and the process is repeated. Changing functionality can be dealt with at runtime as requests and responses are constructed dynamically. The speed at which changes can be incorporated depends on the refresh rate of the API documentation, which is set to one minute for this evaluation. Note that documentation exchanges are the only exchanges which do not get logged in the experiments.

6.2 | Measurement setup

Additional code fragments are presented in this section to show how data exchange and logging cycles are processed and how the measurements, as shown in the next section, are exactly obtained. The code fragments in Listings 7 and 8 highlight the setup as used in the client code.

```
    var repeat = setInterval(function() {
        if ((Date.now() - startTime > timeLimit)) {
            return clearInterval(repeat);
        }
        // Select recipient using matrix P (section 6.1)

        // Dynamically construct a request (section 6.1)

        // Determine latest inspected block number + 1 (= SBI)

        // FUNCTION IV: Log Request asynchronously (RQ_1)

        // Request data from Org X asynchronously (RQ_2, SBI)
        var call = http.request(options, response => {
            // FUNCTION V: Inspect Response asynchronously (args: SBI, and RBI from Org X)
            var  call = http.request(options , response => {
                // Update latest inspected block number
            });

            // Process received data
        });
    }, delay);
```

Listing 7: Client code to periodically generate cross-organizational data requests.

```
    // Receive requests from Org Y
    app.receive('/collab/*') {
        // Protection I (section 5): Check for pending hash verification requests with Org Y

        // Protection II (section 5): dataResponsesToOrgY - loggingCyclesByOrgY <= M

        // Retrieve requested data from internal feature
        var call = http.request(options, response => {
            // Determine latest inspected block number + 1 (= SBI)

            // FUNCTION IV: Log Response asynchronously (RS_1)

            // FUNCTION V: Inspect Request asynchronously (args: SBI, and RBI from Org Y)
            var call = http.request(options, response => {
                loggingCyclesByOrgY++;
            });

            dataResponsesToOrgY++;
        });

        // Send requested data (RS_2, SBI)
    });
```

Listing 8: Client code to process incoming data requests.

The core of the client code is the periodically executed function `getData` which allows data requests to be sent concurrently to different organizations. The `delay` in milliseconds with which the function is executed can be expressed in terms of S, which represents the number of state transitions per second: $\frac{1000}{E}$ ms. Each time it executes, it selects a recipient according to the process described in section 6.1, logs the request asynchronously, executes the request asynchronously, inspects the response asynchronously when the request is resolved and processes the received data. The periodic execution of the function stops when a preset deadline, e.g. ten minutes, is exceeded. When a request is received, the two protections as described in section 5 are evaluated. Only when both are passed, the requested data is retrieved from the internal feature, the response is logged asynchronously, the request is inspected asynchronously and the data is sent. Listings 9 and 10 highlight the setups as used in the proxy code. The core of this code is the `inspect` function. It first waits until the request/response to be inspected is received. When this is the case, it then invokes the `callCc` function to do the inspection. This latter function returns a promise in which it first sends a TX proposal to the EP, which sends back its endorsement, after which the endorsement is sent to the ordering service in order to be integrated in a block. When this is done, the previously discussed `waitForCycle` function is invoked as a final step.

```
    // FUNCTION IV
    function callCc(ccFunction, txId, args) {
        return new Promise((resolve, reject) => {
            var phase_one = // Struct: CC ID, CC function, TX ID, transient map, targeted EP

            // Let EP execute TX proposal
            channel.sendTransactionProposal(phase_one)
            .then(endorsement => {
                var phase_two = // Struct: endorsement, original proposal and TX ID

                // Send TX to the ordering service
                channel.sendTransaction(phase_two)
            })
            .then(() => { resolve(); });
        });
    }
```

Listing 9: Proxy code to execute a chaincode function.

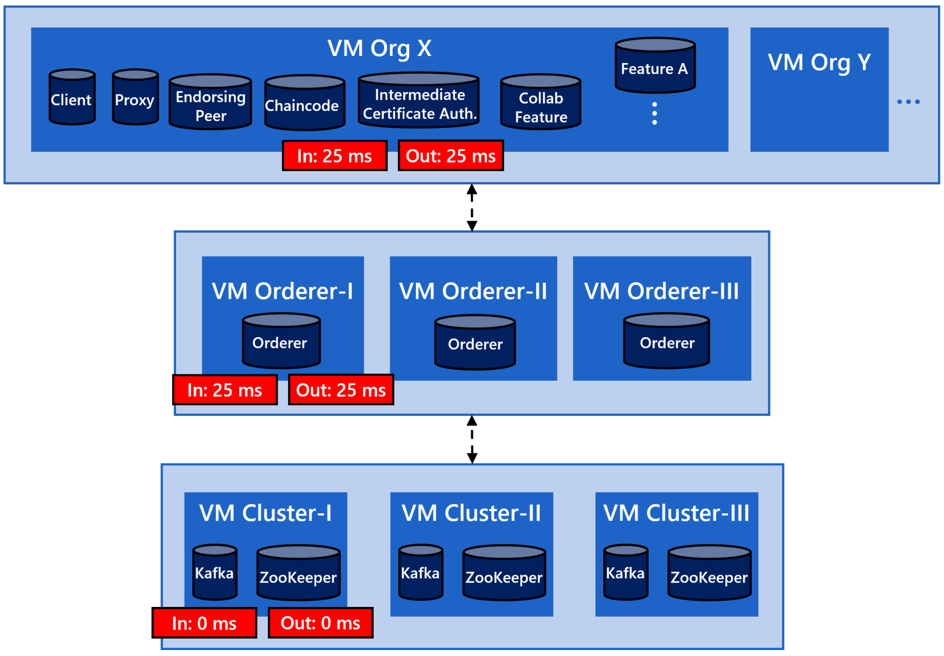


FIGURE 8 Overview of the different VMs used in the evaluation setup.

```
    // FUNCTION V
    function inspect('request' / 'response', SBI, RBI, ...) {
        var txIdLog = RQ_2[TX_ID_LOG_REQUEST] / RS_2[TX_ID_LOG_RESPONSE];

        // Listen for log of counterparty to come in
        listenToEvent(txIdLog, RBI) // FUNCTION II
        .then(() => {
            var txIdInspect = RS_2[TX_ID_INSPECT_REQUEST] / RQ_2 [TX_ID_INSPECT_RESPONSE];

            // Inspect log of counterparty
            callCc('inspect_request' / 'inspect_response', txIdInspect, ...); // FUNCTION IV
        })
        .then(() => {
            // txIds = [txIdInspect, RQ_2[TX_ID_INSPECT_RESPONSE] / RS_2[TX_ID_INSPECT_REQUEST]]
            waitForCycle(txIds, SBI, RBI, ...); // FUNCTION III
        });
    }
```

Listing 10: Proxy code to inspect the log of a counterparty.

6.3 | Evaluation results

The experiment setup, as summarized in Figure 8, consists of sixteen virtual machines (VMs). Each of these VMs runs Ubuntu 18.04 LTS and is equipped with four vCPUs of an Intel Xeon E5645 2.4 GHz processor and 4 GiB of RAM. The different VMs act as worker nodes in the Kubernetes v1.13 cluster. The Docker images of Fabric version 1.3 are used for the EPs and the OSNs and the default key-value store LevelDB[49] is used as database in the peers' secondary memory. Furthermore, three Kafka[50] and three ZooKeeper[51] instances are deployed to set up a Kafka cluster with replication factor three and with at least two in-sync replicas. The *tc* command is used to add an equal latency of 25 ms to both the incoming and outgoing packets for the VMs of the organizations and the VMs of the OSNs, leading to a round trip time of 100 ms. The VMs of the Kafka cluster do not impose an artificial delay on packets, as it is assumed that the different brokers are running in the same data center. This latter assumption is important, as a serious drop in logging cycle completion can be observed when an artificial delay between these brokers is set. Finally, the Node.js codes of the client, proxy and feature components use HTTP agents which reuse existing TCP connections in order to heavily lower the different number of sockets that need to be used.

TABLE 2 The parameter set used for the reported experiments.

Parameter	**Value**
# Organizations (O)	10
# State changes per second per org (S)	25
Collaboration duration	10 minutes
Probability of moving to sleep state (x)	$\frac{1}{5}$
Probability of remaining in same state (y)	$\frac{3}{10}$
Time between cross-organizational hash checks (I)	5 seconds
Max(#data responses to counterparty - #logging cycles by counterparty) (M)	20
Fabric's max. block size (BS)	512 KiB
Fabric's block creation timeout (BT)	2 seconds

The goal of the evaluation section is not to present an exhaustive performance overview of a data exchange service implementing the proposed logging mechanism. The reason for this is that performance results are influenced by many use case specific parameters, such as the number of organizations, the number of data streams per second, the direction of data streams, the type of data, the collaboration duration, the number of clients and proxies per organization, the latencies between the virtual machines, etc. Therefore, a typical case is selected, i.e. a short-term collaboration of ten minutes between a limited set of ten organizations, and the performance impact of the logging mechanism is evaluated. The complete set of parameters used to evaluate the software is shown in Table 2. Data is exchanged in a fully automated way as described in section 6.1. A part of the data requests will be refused due to the protections discussed in section 5, as these have the goal to protect a data delivering organization from a potentially malicious collaboration setup. As no truly malicious entities are present in this controlled experimental setup, both protections will always be satisfied eventually (depending on the different loads of the involved organizations), causing subsequent data requests to be processed on an ongoing basis.

Table 3 compares the performance results when the logging mechanism is turned on and off. Both experiments are repeated ten times and the median values together with the corresponding interquartile ranges (IQRs) are shown (percentiles are calculated using the nearest-rank method). The number of completed exchanges reduces with around 5% when the mechanism is turned on, but still more than ten data exchanges per second per organization are finished. The logging mechanism thus has an impact, but it certainly does not drastically interrupt the data exchange processes under this configuration. The ten minute collaboration is, thanks to the logging mechanism, fully captured and stored in a replicated directory of size 1.1 GiB. Note that this, by Fabric created, directory includes the chain of blocks and the LevelDB database, but not the actual request and response data which are needed to reconstruct the stored hash values. The number of hash verification requests agrees with the setting of parameter I as displayed in Table 2, as each organization does 120 chain synchronization communications in 10 minutes. The number of API documentation requests is also expected, as each organization refreshes the API documentation of the other nine organizations every minute. Finally, the average TX size is 3.5 KiB. This average TX size is a bit lower than the earlier reported 4.8 KiB [19]. The reason for this is that the `transientMap` field instead of the `args` field is now used when the `sendTransactionProposal` function (shown in Listing 9) is called. This prevents the arguments passed to each CC function to be logged in a TX, i.e. it prevents the actual request and response data to be logged unintentionally.

Table 4 shows the performance results when the value of M is further lowered, i.e. when the data exchange processes between organizations are more tightly coupled to their corresponding logging processes. The experiments are repeated ten times and the median values together with the corresponding interquartile ranges (IQRs) are shown (percentiles are calculated using the nearest-rank method). The results show that setting low M values seriously impacts the performance of the data exchange processes. When M is decreased from twenty to zero, the number of completed data exchanges drops significantly. Compared to the situation when the logging mechanism is disabled, it leads to a performance reduction of more than 80%. The selection of the parameter M is thus a crucial decision. A trade-off has to be made by each data delivering organization, i.e. they have to decide whether they prefer more logging certainty or whether they are willing to contribute to a higher performance of the exchange processes. This value could possibly be changed according to the level of trust and the required performance at a specific moment in time.

TABLE 3 Overview of both cross-organizational interactions and generated logging data when the parameter set defined in Table 2 is used, and when the logging mechanism is turned on/off.

	Logging disabled		**Logging enabled**	
# Completed Exchanges (E_{total})	70941	IQR: 317	67785	IQR: 230
# `/collab`	23673		22620	
# `/collab/images`	11812		11310	
# `/collab/employees`	11805		11274	
# `/collab/images/[0-9]`	11796		11295	
# `/collab/employees/[0-9]`	11788		11256	
# Postponed Requests (P_{total})	47528	IQR: 156	45982	IQR: 214
# Refused Requests (R_{total})	0	IQR: 0	672	IQR: 14
# Hash Verification Requests (H_{total})	0	IQR: 0	1200	IQR: 1
# API Documentation Requests (D_{total})	900	IQR: 0	900	IQR: 0
# Blocks received by each Org	0	IQR: 0	1974	IQR: 2
# TXs received by each Org	0	IQR: 0	271810	IQR: 972
Size ledger data at each Org (GiB) `/var/hyperledger/production/ledgersData`	0	IQR: 0	1.1	IQR: 0

TABLE 4 Overview of cross-organizational interactions when the parameter set defined in Table 2 is used, when the logging mechanism is turned on, and when the parameter M is lowered.

	$M = 20$		$M = 10$		$M = 5$		$M = 1$		$M = 0$	
# E_{total}	67785	IQR: 230	67641	IQR: 240	56579	IQR: 181	22183	IQR: 25	11517	IQR: 28
# P_{total}	45982	IQR: 214	45965	IQR: 426	44512	IQR: 49	39500	IQR: 179	37761	IQR: 233
# R_{total}										
Protection I: Pending hash verification	672	IQR: 14	687	IQR: 32	679	IQR: 63	666	IQR: 29	695	IQR: 27
Protection II: M exceeded	0	IQR: 0	97	IQR: 3	13181	IQR: 74	54590	IQR: 59	67009	IQR: 249

7 | CONCLUSIONS

This article focuses on a logging mechanism for temporary and ad hoc cross-organizational collaborations using the Hyperledger Fabric framework which implements a so called permissioned blockchain. In the normal case, i.e. when a complete logging cycle is generated, the logging mechanism provides proof of what has happened during a collaboration in order to prevent possible disputes. When faulty behavior is introduced, intentionally or not, a data delivering organization will be able to detect this, and pause its data sharing with a specific organization or even its participation in the collaboration until the problem is solved. Although the cause of the fault cannot be proven, it provides a way for data delivering organizations to share data in an untrusted setup. Important is that organizations are incentivized to execute the logging functions properly, as they may lose access to the data of other organizations otherwise. The contributions compared to a previously published paper by the authors[19], are a more extensive explanation of the properties of the proposed mechanism, the idea of applying generic web APIs as found in literature to this use case and a more detailed evaluation of the designed proof of concept. Future work will need to investigate how this work can be combined with existing access control solutions which allow person-to-person data sharing and how container orchestration could be applied to cross-organizational scenarios.

8 | ACKNOWLEDGEMENTS

The work described in this paper, is partly funded by the FUSE project[52], in which a Flexible federated Unified Service Environment is investigated. The project is realized in collaboration with imec. Project partners are Barco, Axians and e-BO Enterprises, with project support from VLAIO (Flanders Innovation & Entrepreneurship).

References

1. Moeyersons J, Farkiani B, Wauters T, Volckaert B, De Turck F. Towards Distributed Emergency Flow Prioritization in SDN Networks. *Int J Netw Manag* 2020; e2127. https://doi.org/10.1002/nem.2127.
2. Dos Santos J, Wauters T, Volckaert B, De Turck F. Fog computing : enabling the management and orchestration of smart city applications in 5G networks. *Entropy* 2018; 20(1): 1–26. https://doi.org/10.3390/e20010004.
3. Hyperledger Fabric. https://www.hyperledger.org/projects/fabric. Accessed December 1, 2019.
4. Androulaki E, Manevich Y, Muralidharan S, et al. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In: Proceedings - 13th EuroSys Conference. ACM; 2018: 1–15. https://doi.org/10.1145/3190508.3190538.
5. we.trade | The Platform. https://we-trade.com/the-platform. Accessed December 1, 2019.
6. Schaefer C, Edman C. Transparent Logging with Hyperledger Fabric. In: Proceedings - 1st IEEE International Conference on Blockchain and Cryptocurrency (ICBC). IEEE; 2019: 65–69. https://doi.org/10.1109/bloc.2019.8751339.
7. TradeLens. https://www.tradelens.com. Accessed December 1, 2019.
8. Han R, Gramoli V, Xu X. Evaluating Blockchains for IoT. In: Proceedings - 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS). IEEE; 2018: 1–5. https://doi.org/10.1109/NTMS.2018.8328736.
9. Selimi M, Kabbinale AR, Ali A, Navarro L, Sathiaseelan A. Towards Blockchain-enabled Wireless Mesh Networks. In: Proceedings - 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems (CryBlock). ACM; 2018: 13–18. https://doi.org/10.1145/3211933.3211936.
10. Thakkar P, Nathan S, Viswanathan B. Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform. In: Proceedings - 26th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS). IEEE; 2018: 264–276. https://doi.org/10.1109/MASCOTS.2018.00034.
11. Sharma A, Schuhknecht FM, Agrawal D, Dittrich J. How to Databasify a Blockchain: the Case of Hyperledger Fabric. tech. rep., Saarland Informatics Campus; 2018. http://arxiv.org/abs/1810.13177.
12. Gorenflo C, Lee S, Golab L, Keshav S. FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second. In: Proceedings - 1st IEEE International Conference on Blockchain and Cryptocurrency (ICBC). IEEE; 2019: 455–463. https://doi.org/10.1109/bloc.2019.8751452.
13. Gorenflo C, Golab L, Keshav S. XOX Fabric: A hybrid approach to transaction execution. tech. rep., University of Waterloo; 2019. https://arxiv.org/abs/1906.11229.
14. Yamashita K, Nomura Y, Zhou E, Pi B, Jun S. Potential Risks of Hyperledger Fabric Smart Contracts. In: Proceedings - IEEE 2nd International Workshop on Blockchain Oriented Software Engineering (IWBOSE). IEEE; 2019: 1–10. https://doi.org/10.1109/IWBOSE.2019.8666486.
15. Andola N, Raghav , Gogoi M, Venkatesan S, Verma S. Vulnerabilities on Hyperledger Fabric. *Pervasive and Mobile Computing* 2019; 59: 101050. https://doi.org/10.1016/j.pmcj.2019.101050.
16. Xiao Z, Li Z, Liu Y, et al. EMRShare: A Cross-Organizational Medical Data Sharing and Management Framework Using Permissioned Blockchain. In: Proceedings - The International Conference on Parallel and Distributed Systems (ICPADS). IEEE; 2018: 998–1003. https://doi.org/10.1109/PADSW.2018.8645049.

17. Kiyomoto S, Rahman MS, Basu A. On Blockchain-Based Anonymized Dataset Distribution Platform. In: Proceedings - 15th IEEE/ACIS International Conference on Software Engineering Research, Management and Applications (SERA). IEEE; 2017: 85–92. https://doi.org/10.1109/SERA.2017.7965711.

18. Muller M, Garzon SR, Westerkamp M, Lux ZA. HIDALS: A Hybrid IoT-based Decentralized Application for Logistics and Supply Chain Management. In: Proceedings - IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON). IEEE; 2019: 802–808. https://doi.org/10.1109/IEMCON.2019.8936305.

19. Van Hoye L, Maenhaut PJ, Wauters T, Volckaert B, De Turck F. Logging mechanism for cross-organizational collaborations using Hyperledger Fabric. In: Proceedings - 1st IEEE International Conference on Blockchain and Cryptocurrency (ICBC). IEEE; 2019: 352–359. https://doi.org/10.1109/BLOC.2019.8751380.

20. What Is OpenAPI?. https://swagger.io/docs/specification/about. Accessed December 1, 2019.

21. Verborgh R, Dumontier M. A Web API Ecosystem through Feature-Based Reuse. *IEEE Internet Computing* 2018; 22(3): 29–37. https://doi.org/10.1109/MIC.2018.032501515.

22. JSON-LD 1.1. https://www.w3.org/TR/2019/WD-json-ld11-20191112/. Updated November 12, 2019. Accessed December 1, 2019.

23. RDF 1.1 Concepts and Abstract Syntax. 2014. https://www.w3.org/TR/rdf11-concepts. Updated February 25, 2014. Accessed December 1, 2019.

24. Lanthaler M. *Third Generation Web APIs - Bridging the Gap between REST and Linked Data*. PhD thesis. TU Graz, 2014. http://www.markus-lanthaler.com/research/third-generation-web-apis-bridging-the-gap-between-rest-and-linked-data.pdf.

25. Lanthaler M. Hydra Console. http://www.markus-lanthaler.com/hydra/console. Published March, 2014. Accessed December 1, 2019.

26. Wüst K, Gervais A. Do you need a Blockchain?. In: Proceedings - 1st Crypto Valley Conference on Blockchain Technology (CVCBT). IEEE; 2018: 45 - 54. https://doi.org/10.1109/CVCBT.2018.00011.

27. Rodrigues B, Scheid E, Blum R, Bocek T, Stiller B. Blockchain and Smart Contracts - From Theory to Practice. http://icbc2019.ieee-icbc.org/files/2019/05/ICBC-2019-Tutorial-1-Blockchain-and-Smart-Contracts.pdf. Published May 14, 2019. Accessed December 1, 2019.

28. Greenspan G. The Blockchain Immutability Myth. https://www.multichain.com/blog/2017/05/blockchain-immutability-myth. Published May 4, 2017. Accessed December 1, 2019.

29. Castro M, Liskov B. Practical Byzantine Fault Tolerance. In: Proceedings - 3rd Symposium on Operating Systems Design and Implementation (OSDI). ACM; 1999: 173–186. https://dl.acm.org/doi/10.5555/296806.296824.

30. Fischer MJ, Lynch NA, Paterson MS. Impossibility of Distributed Consensus with One Faulty Process. *J Assoc Comput Machin* 1985; 32(2): 374–382. https://doi.org/10.1145/3149.214121.

31. Amsden Z, Arora R, Bano S, et al. The Libra Blockchain. tech. rep., Facebook, Calibra; 2019. https://www.semanticscholar.org/paper/The-Libra-Blockchain-Amsden-Arora/59df4bdd67ed1cde5191447bcc80fba2d70bee71.

32. Ongaro D, Ousterhout J. In Search of an Understandable Consensus Algorithm. In: Proceedings - USENIX Annual Technical Conference. USENIX Association; 2014: 305–320. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

33. Sousa J, Bessani A, Vukolic M. A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform. In: Proceedings - 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE; 2018: 51–58. https://doi.org/10.1109/DSN.2018.00018.

34. Mohan C. State of Public and Private Blockchains: Myths and Reality. In: Proceedings - ACM SIGMOD/PODS International Conference on Management of Data. ACM; 2019: 404–411. http://doi.acm.org/10.1145/3299869.3314116.

35. Mohan C. Slides accompanying [34]. https://drive.google.com/file/d/1wJm4K7_7CkvyzmyuySmqDGH0N-mwICSX. Published July 3, 2019. Accessed December 1, 2019.

36. Bitcoin is not ruled by miners. https://en.bitcoin.it/wiki/Bitcoin_is_not_ruled_by_miners. Updated August 17, 2017. Accessed December 1, 2019.

37. Difficulty. https://en.bitcoin.it/wiki/Difficulty. Updated November 25, 2019. Accessed December 1, 2019.

38. Bogomolny A. Number of Trials to First Success. https://www.cut-the-knot.org/Probability/LengthToFirstSuccess.shtml. Accessed December 1, 2019.

39. Non-specialized hardware comparison. https://en.bitcoin.it/wiki/Non-specialized_hardware_comparison. Updated June 10, 2019. Accessed December 1, 2019.

40. Nakamoto S. Bitcoin: A Peer-to-Peer Electronic Cash System. tech. rep., bitcoin.org; 2008. https://bitcoin.org/bitcoin.pdf.

41. Micali S. Algorand's Core Technology (in a nutshell). https://www.algorand.com/resources/blog/algorands-core-technology-in-a-nutshell. Published April 4, 2019. Accessed December 1, 2019.

42. Greenspan G. Blockchains vs centralized databases. https://www.multichain.com/blog/2016/03/blockchains-vs-centralized-databases. Published March 17, 2016. Accessed December 1, 2019.

43. Hyperledger Fabric - Blocks. https://hyperledger-fabric.readthedocs.io/en/release-1.3/ledger/ledger.html#blocks. Updated April 19, 2018. Accessed December 1, 2019.

44. Github Fabric Release 1.3 - mcs.go. https://github.com/hyperledger/fabric/blob/release-1.3/peer/gossip/mcs.go#L120. Updated October 10, 2017. Accessed December 1, 2019.

45. Github Fabric Release 1.3 - blockfile_mgr.go. https://github.com/hyperledger/fabric/blob/release-1.3/common/ledger/blkstorage/fsblkstorage/blockfile_mgr.go#L240. Updated September 17, 2018. Accessed December 1, 2019.

46. Hyperledger Fabric - Peers. https://hyperledger-fabric.readthedocs.io/en/release-1.3/peers/peers.html. Updated May 17, 2018. Accessed December 1, 2019.

47. Github Fabric Release 1.3 - block.go. https://github.com/hyperledger/fabric/blob/release-1.3/protos/common/block.go#L51. Updated February 19, 2017. Accessed December 1, 2019.

48. Hyperledger Fabric SDK for node.js. https://github.com/hyperledger/fabric-sdk-node/tree/release-1.3. Updated January 21, 2019. Accessed December 1, 2019.

49. LevelDB. https://github.com/google/leveldb. Accessed December 1, 2019.

50. Kubernetes Kafka. https://github.com/kubernetes-retired/contrib/blob/master/statefulsets/kafka/kafka.yaml. Updated April 17, 2017. Accessed December 1, 2019.

51. Kubernetes ZooKeeper. https://github.com/kubernetes-retired/contrib/blob/master/statefulsets/zookeeper/zookeeper.yaml. Updated October 20, 2017. Accessed December 1, 2019.

52. FUSE: Flexible federated Unified Service Environment. https://www.imec-int.com/en/what-we-offer/research-portfolio/fuse. Accessed December 1, 2019.