Logging mechanism for cross-organizational collaborations using Hyperledger Fabric

Laurens Van Hoye, Pieter-Jan Maenhaut, Tim Wauters, Bruno Volckaert, Filip De Turck
Department of Information Technology
Ghent University - IDLab
Technologiepark-Zwijnaarde 126, 9052 Ghent, Belgium
{Laurens.VanHoye, PieterJan.Maenhaut, Tim.Wauters, Bruno.Volckaert, Filip.DeTurck}@UGent.be

***Abstract*—Organizations nowadays are largely computerized, with a mixture of internal and external services providing them with on-demand functionality. In some situations (e.g. emergency situations), cross-organizational collaboration is needed, providing external users access to internal services. Trust between partners in such a collaboration can however be an issue. Although (federated) access control policies may be in place, it is unclear which data was requested and delivered after a collaboration has finished. This may lead to disputes between participating organizations. The open-source permissioned blockchain Hyperledger Fabric is utilized to create a logging mechanism for the actions performed by the participants in such a collaboration. This paper presents the architecture needed for such a logging mechanism and provides details on its operation. A prototype was designed in order to evaluate the performance of an asynchronous logging approach. Measurements show that the proposed logging mechanism enables organizations to create a log of service interactions with limited delay imposed on the data exchange process.**

***Index Terms*—blockchain, collaborations, cross-organizational, distributed, logging**

I. Introduction

A. Context

Cross-organizational collaborations should allow participants to share in-house services across administrative domains in a secure way, i.e. without making them publicly accessible. The added value of this is that it allows to share knowledge among the partners and as such to derive more intelligence. Figure 1 shows an example of a possible use case. Manufacturers need machines for product creation, e.g. a robotic arm, and order them from equipment builders. These equipment builders could fix the operation of their machines using the data they produce, but they have no direct access to these machines once installed. The investigated scenario is always the same, i.e. there is a data exchange between Org X and Org Y and the latter requests the data. As plenty of such cross-organizational collaborations are possible, there is both a scientific interest and market potential for research focusing on interconnecting cross-organizational systems in a secure way.

B. Goal of logging mechanism

In most cross-organizational collaboration scenarios, participants will have access control policies in place which define what data can be accessed. After a collaboration has ended, it is however not clear what specific data has been requested and what data has been delivered if no logging mechanism is used, which may lead to disputes between organizations. The proposed logging mechanism has two characteristics:

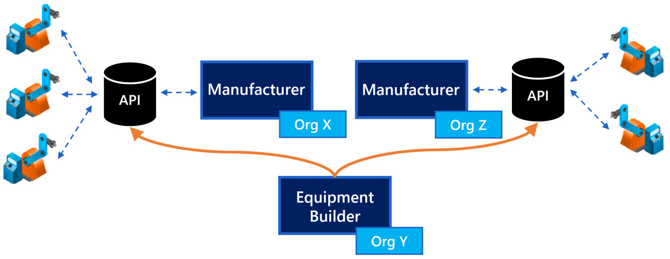

Fig. 1. Sample collaboration scenario showcasing an equipment builder gaining access to internal APIs of machines it installed at different manufacturers.

- Make it impossible for an organization to deny (the integrity of) a request/response which it received during the collaboration, leaving no option for a dispute afterwards.
- Allow an honest organization to detect a lack of logging information either due to dishonest organizations, due to malfunctioning caused by e.g. a network failure or due to an attack on the operation of the mechanism.

The goal of the logging mechanism is that, when it is executed correctly, no disputes are possible afterwards. As will become clear in section III-A, the mechanism itself cannot enforce correct execution, meaning disputes are still possible. However, an honest organization can detect incorrect behavior and decide to immediately stop collaborating with the participants in the collaboration. An honest organization thus continuously assesses whether the logging procedure is correctly executed and takes action when this is not the case.

In order to realize this goal, it is necessary to produce cryptographically signed logs which describe the exchange in an unforgeable way. An appropriate solution could be to communicate, for each data exchange, four signed messages between Org Y and Org X: signed request and response messages and also signed request and response confirmation messages. This approach is used in this paper. The only remaining problem is that, in case of data loss, an honest organization loses all its logging data. The solution therefore needs to enforce a more fail-safe data storage. The first option is to store the logs in a crash fault-tolerant storage solution, managed by a third party, which can be read and written by all organizations. However, in the case under investigation, it might be difficult to find a third party which is trusted by all involved organizations. This party has the power to manipulate logging state, even when organizations execute periodic checks on it, as it can still be manipulated after the collaboration has finished. The second option is to replicate state over different nodes which is investigated in the next section.

II. Related work

The main advantage of replicating state over using a third party is that each organization has its own replica of the state stored in its trusted domain. This means that it can execute checks on this copy without having to rely on an external entity. An honest organization can execute two checks in order to detect a lack of logging information:

1) For each data exchange, it will check whether its state contains all the signed logs it expects there to be.
2) It can compare its state with those of other organizations to verify whether data is correctly replicated.

For this approach to work, it is important that each organization has an append-only log of state transitions, as otherwise check one could evaluate to true at inspection time but to false after state rewrites. A technology providing this finality is Hyperledger Fabric, a prominent permissioned blockchain architecture, in which each peer has a ledger consisting of a world state database with key-value pairs (KVPs) and an append-only chain of transactions (TXs) capturing the corresponding state transitions [1]. Storing TXs in a blockchain data structure is also interesting for check two, as the latest hash provides a summary of all TXs that happened before. Two organizations comparing state then only need to compare their hash value at a certain block height, which is an efficient operation. It is important to stress that, in this use case, chaining blocks of TXs is not used to enforce immutability, like this is done in e.g. Bitcoin where mining blocks is a costly operation due to the Proof-of-Work consensus mechanism, but rather to have an efficient way to compare state. The choice for a private permissioned blockchain is supported by the flow chart in Figure 2 which is commonly used in literature:

1) Data needs to be stored in a structured way, introducing the need for a database.
2) There are multiple writers as each organization will be allowed to store its logs.
3) As already mentioned, delegating logs to an always online trusted third party (TTP) is not possible because all organizations would need to trust it for processing the logs correctly, an assumption which may not be true for all collaborations. Instead, an offline TTP can be used as a certificate authority for a permissioned blockchain.
4) All writers are known, namely the participants in the collaboration.
5) If all writers would mutually trust each other, each organization could simply maintain its own log file. If organizations have a history of trustful collaboration, this could be the case, but it cannot be assumed for an ad hoc collaboration between unknown participants.
6) Public verifiability is not required as the participants involved in the collaboration are the only stakeholders in the data exchange process.

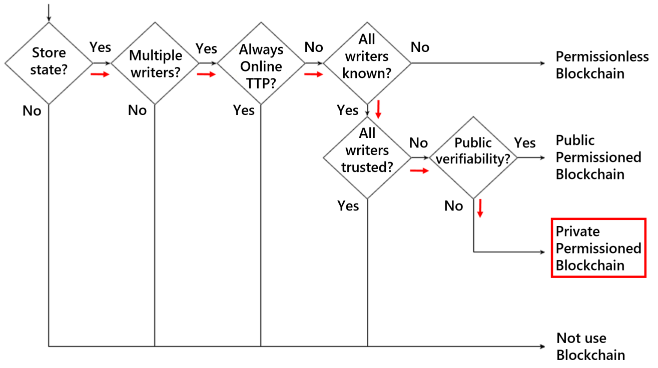

Fig. 2. Which architectural blockchain model is most appropriate for an application? [2]

The paper written by E. Androulaki et al. [3] describes the fundamentals of Hyperledger Fabric. Although this subsection does not reproduce its entire internal operation, it is important to address some fundamental concepts. The main innovation of Fabric is that it uses a three-phase model as consensus mechanism. For each TX, there are three separate phases, more specifically TX execution, ordering and validation. As explained in the paper, this model solves a number of limitations which are commonly found in permissioned blockchains which use an order-execute model. One of the advantages it brings is that the ordering step is decoupled, meaning pluggable consensus can be used for this phase, i.e. for agreeing upon a total order of TXs. Currently, Fabric provides only one out of the box production-ready ordering service which is based on a Kafka cluster. This distributed messaging platform can withstand crash faults, but can not cope with malicious brokers introducing Byzantine faults. The paper written by J. Sousa et al. [4] proposes the first Byzantine Fault Tolerant (BFT) ordering service for Fabric. The Kafka cluster is replaced with a set of frontend nodes, which the peers can connect to, and a set of ordering nodes, which the frontend nodes connect to. A Practical Byzantine Fault Tolerant (PBFT) scheme, based on the BFT-SMaRT library [5], is used between the ordering nodes as consensus mechanism. This way, it is possible to withstand f malicious ordering nodes from a set of size n as long as $f < \frac{n}{3}$. Assuming an organization is only allowed to deploy at most one ordering node in its own domain to prevent a Sybil attack, 4-6 organizations can cope with 1 malicious organization, 7-9 with 2 malicious organizations, 10-12 with 3 malicious organizations, etc. This also means that, when the Byzantine ordering service is used, a collaboration between three organizations does not seem to be possible in a fully distributed setting.

The integrity of Fabric thus lies in the operation of the ordering service. For the cross-organizational collaborations researched in this paper, the Kafka ordering service is used. It cannot cope with Byzantine faults, but as organizations execute the checks mentioned above, they can detect any malicious behavior. The conclusion is that an improved trust model for the ordering service could be used, as it makes malicious behavior of this part of the architecture harder, but it is not necessary for this application due to the proposed checks. It is important to note that deploying a Kafka cluster at one organization is not the same scenario as using a TTP, because each organization has its own ledger for which it can execute checks. A malicious ordering service could never invent TXs as it is not capable of creating a valid signature. Furthermore, it could never remove TXs as organizations would find out by executing the checks. The only thing it could do is reorder TXs, but it is only important that there is a strict order of TXs in order to obtain the same chain of TXs for each organization, not what that specific order is [6]. Finally, the TXs only contain hash values as will become clear in section III-A, meaning it is impossible to leak information.

There are already multiple research papers examining the Hyperledger Fabric technology. On the one hand, there are papers which present use cases different than the one described in this paper, e.g. banking [7], voting [8], managing access to an electronic health record [9], managing configuration of IoT devices [10], managing inter-organizational user authentication in a distributed manner [11], decentralizing service ecosystems [12] and executing know-your-customer validation [13]. The use case presented in the paper written by S. Kiyomoto et al. [14] comes close to the use case examined in this paper. Encrypted anonymized data is sent between a data broker and data receiver and fingerprints of this exchange are stored in the ledger by the data broker. Only when the data broker has received a confirmation message of the TX coming from the blockchain, it sends the key to the data receiver to decrypt the data. This approach is a synchronous one, i.e. the data can only be used when the blockchain operation is completed. An asynchronous approach will be proposed and evaluated in this paper. On the other hand, there are also papers which focus more on Fabric itself, e.g. on how the deployment life cycle should be managed [15] and on how the blockchain could be queried in an efficient way [16]. There are also papers available which address the issue of privacy, e.g. when only a subset of the peers is allowed to see the exchanged data, e.g. using secure multiparty computation whereby data is encrypted using a shared secret key or using the public key of each allowed organization [17], or for executing smart contracts with secrets, e.g. in trusted execution environments like Intel SGX enclaves [18].

III. Logging mechanism

A. Design decision

The baseline architecture to start from consists of multiple client-server relationships. This situation is shown in Figure 3, whereby three organizations want to share APIs among them. Two conceptual channels are defined, i.e. putting content in the common ledger is called channel 1 and direct communication between a pair of organizations is called channel 2. There are two possible design strategies:

1) Only channel 1 is used, i.e. both request and response are communicated via this channel and are thus stored in the common ledger. In the case of e.g. video data, TXs become large and storage could become a problem as the ledger grows quickly: when a 1080p 24fps video stream is encoded with an H.264/MPEG4-AVC encoder, a stream with a bit rate of approximately 1000 kbps is obtained with a Y-PSNR of around 35 [19]. When only 10 minutes of video data is shared, this leads to 75 MB of data per camera that needs to be stored at each peer, which does not scale very well. Another example is transferring files of a few MB or more between organizations. An advantage of this approach is that it is secure, i.e. no disputes are possible about the actual request/response that was sent, as all participants can query the ledger.
2) Only a fingerprint of the request and response is stored in the ledger, i.e. the data of each request and response is hashed. The actual data is then exchanged via a communication channel different than the ledger. Each organization is responsible to store the data corresponding with the hashes it puts in the ledger, as it should be able to reveal its data in case of a dispute. The drawback of this approach is that organizations other than the two involved in the data exchange cannot determine whether the request/response sent via channel 2 matched with the one logged via channel 1. This means that e.g. dishonest Org Y could falsely deny to have received a response from honest Org X. In general, it is impossible to verify whether an organization did not confirm a request/response on purpose, i.e. allowing a possibility for a dispute, or whether it did not receive an actual request/response at all. This means that disputes are still possible until a signed confirmation message is stored.

Due to the possible scalability issue of strategy one, the second strategy is further examined in this paper. Although it is less secure as strategy one, it achieves the goals mentioned in section I-B.

B. Architecture of logging mechanism

Figure 3 shows the components that are used for the proposed logging mechanism. Each organization has an EP, client, and a proxy. EP stands for endorsing peer as used in the Hyperledger Fabric architecture. The set of endorsing peers is typically a subset of the entire peer set. Their role is to simulate TX proposals originating from the proxies, i.e. they execute the chaincode (CC), also called smart contracts, with the given input parameters and send back their simulated response and read/write sets of the ledger's key-value pairs [20]. As each organization should be able to sign its own TX proposals, they all need at least one such endorsing peer. Each EP runs the CC in a separate Docker container. As Docker prevents a container from accessing data and processes running in the host system and also prevents it from

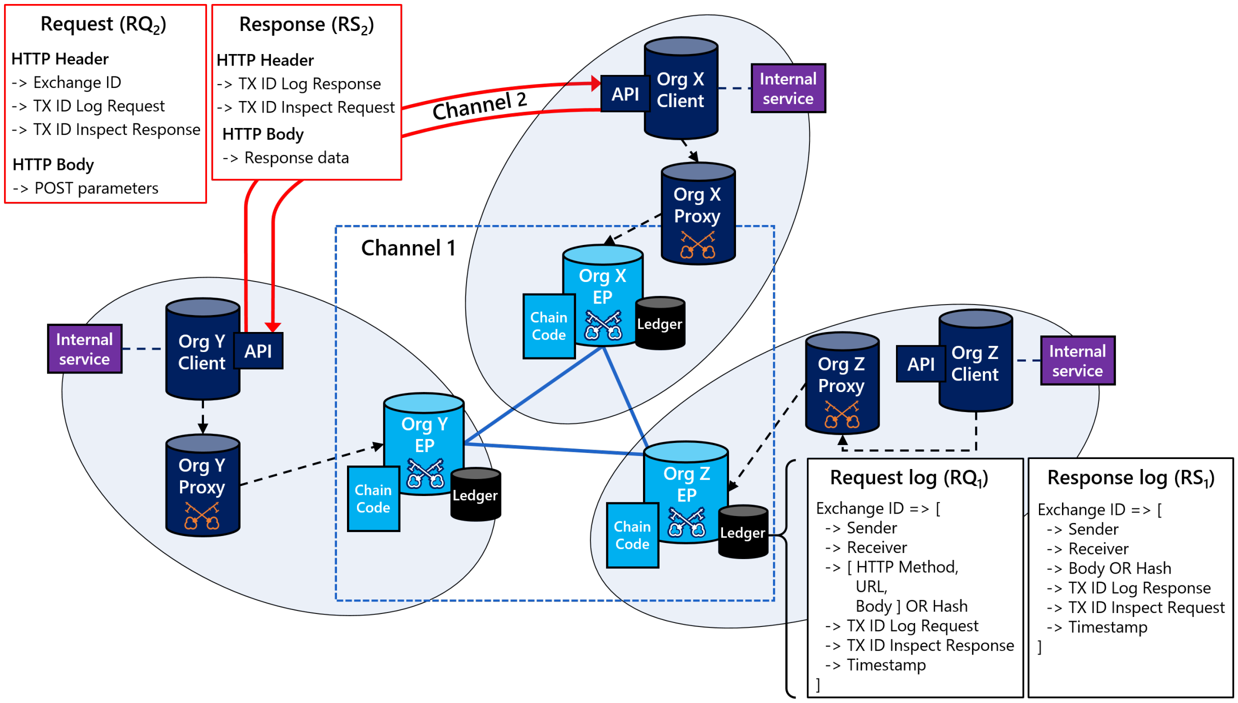

Fig. 3. Components needed for the proposed logging mechanism.

exhausting resources [21], the host system cannot suffer from malicious code. The client components each expose a web API within the collaboration to share internal services. Requests for data and corresponding responses are exchanged between these clients. When the exchange is ongoing, the proxies are asynchronously called by the clients to execute steps of the logging mechanism. These proxies then communicate with the EPs of the organization to sign TX proposals before they are sent to the ordering service.

As mentioned in section I-B, four signed messages are needed per data exchange. The core of the logging mechanism therefore consists of four functions defined in the CC. Every organization executes the same CC, i.e. they update the ledger in the same predefined way, and only one channel is needed as each organization is allowed to see all TXs. There are two logging functions and two inspect functions. The logging functions are needed to log the request coming from Org Y and the response coming from Org X:

- `LogRequest`: Org Y creates a TX proposal for putting the data of RQ_1, as shown in Figure 3, into the ledger. It signs the proposal and sends the TX to the ordering service.
- `LogResponse`: Org X creates a TX proposal for putting the data of RS_1, as shown in Figure 3, into the ledger. It signs the proposal and sends the TX to the ordering service.

A hash is calculated in the implementation of these functions. For RQ_1, the hash is calculated over the HTTP method, URL and body. For RS_1, the hash is calculated over the response body. Currently, the SHA-256 hashing function is used, meaning data of any length is compressed to 32 bytes. Afterwards, the data is stored in the ledger using Fabric's function `PutState`, i.e. they update the ledger's KVPs and cause state transitions. These transitions are then logged as different TXs in the chain of blocks. It is important to note that Hyperledger Fabric throws an error when two TXs in the same block try to update the same KVP [22]. To prevent this, a unique key in the ledger is constructed by appending _request or _response to the exchange ID value.

As the content set in these KVPs can be anything, i.e. an organization can log whatever it wants, it needs to be examined whether the logged requests/responses correspond with the actual requests/responses sent via channel 2. Only when this is true, the log can be seen as a correct reflection of what has happened during a collaboration. Two more functions are thus required:

- `InspectRequest`: Org X needs to confirm whether the received request RQ_2 matches with the one logged RQ_1 by Org Y.
- `InspectResponse`: Org Y needs to confirm whether the received response RS_2 matches with the one logged RS_1 by Org X.

Both functions use Fabric's function `GetState`, i.e. they read data from the ledger. A read operation does not cause state transitions, meaning no evidence of this check is stored in the ledger. However, the goal is to obtain a log file showing the exchanges that have happened, implying that when an organization agrees with a log, it should confirm this. The organization inspecting a request/response should therefore do the same as with `LogRequest` and `LogResponse`, i.e. put its confirmation in the ledger by creating a TX proposal, signing it and sending it to the ordering service.

As already mentioned in section II, an asynchronous flow is proposed in this paper. This means that the speed of the data exchange process does not heavily depend on the speed of the logging mechanism, i.e. channel 2 is almost not delayed by channel 1. The more synchronous the approach is, i.e. the more blocking behavior is present, the slower the data exchange process becomes. If this logging mechanism would then be used to log calls received by an API being faced with a high load, it could become the bottleneck of the system. The goal is therefore to minimize the overhead caused by the logging mechanism and to evaluate the performance of an asynchronous approach. To show the complete cycle of a secure data exchange, a sequence diagram is presented in Figure 4. It shows the asynchronous approach with its two interaction schemes each operating at their own pace. The first scheme enables a fast exchange of data, the second scheme enables a slower logging of all the actions. The exact order of execution can differ a bit, depending on how long an asynchronous operation takes to execute.

Figure 3 shows the HTTP headers of RQ_2 and RS_2. As Org Y logs the request for data, it needs to send the TX ID Log Request along with RQ_2. This enables Org X to wait for the TX to be committed to its local ledger and to execute its inspect function (arrow 12 and 13). This works the same for the response, i.e. Org X sends the TX ID Log Response along with RS_2, allowing Org Y to execute its inspect function at the appropriate moment (arrow 14 and 15). Finally, each organization also wants to verify whether its partner executed the inspect function. Each organization therefore sends along the TX ID which it will use to register its inspection. Sending the TX ID Inspect Response in RQ_2 allows Org X to check whether Org Y inspected its response, while sending the TX ID Inspect Request in RS_2 allows Org Y to check whether Org X inspected its request. It is important to note that when waiting for a TX to be committed to the local ledger, a period of two seconds is used between two consecutive inspects of the ledger and a timeout value is used to determine when a TX should be committed at the latest. This timeout is needed to enable check one mentioned in section II.

The ‘Fabric‘ lifeline is further detailed in Figure 5, showing the integration of Fabric’s TX flow into the logging mechanism. This interaction is executed for each CC function, e.g. `LogRequest` as shown in the figure. For this use case, an organization only needs to send TX proposals to its own endorsing peer (arrow 1-3). As each organization is responsible for its own actions, no other organizations need to endorse the proposal, which benefits the scalability of this mechanism. This does however imply that each organization can update any KVP it wants. However, as each update is signed, backtracking dishonest behavior is simple. The rest of the diagram shows the normal TX flow as used in Hyperledger Fabric.

Finally, it is important to focus on Fabric’s finality aspect as already mentioned in section II. Fabric can provide finality due to the use of a deterministic ordering service. This service is responsible for ordering incoming TXs from the organizations’ proxies, creating blocks, signing them for data integrity and authentication, and finally delivering them to all the peers in the network. Once delivered, blocks can never be changed, as a Fabric’s peer always checks whether an incoming block’s sequence number succeeds the height of its chain. This means that, even when an ordering service is malicious, it can never rewrite history of an honest organization.

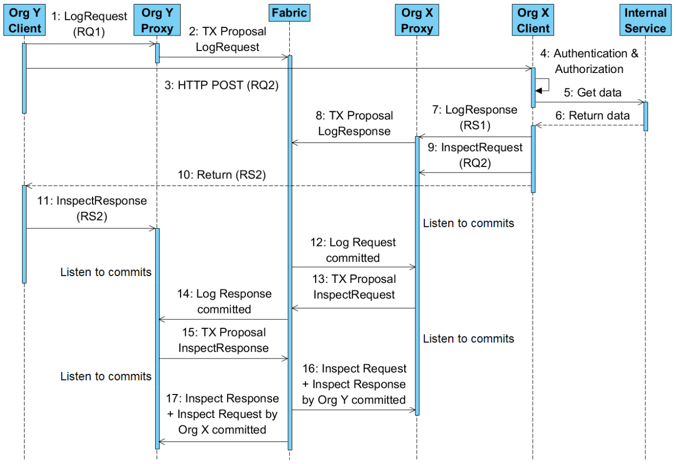
Org Y Client
Org Y Proxy
Fabric
Org X Proxy
Org X Client
Internal Service
1: LogRequest (RQ1)
2: TX Proposal LogRequest
3: HTTP POST (RQ2)
4: Authentication & Authorization
5: Get data
6: Return data
7: LogResponse (RS1)
8: TX Proposal LogResponse
9: InspectRequest (RQ2)
10: Return (RS2)
11: InspectResponse (RS2)
Listen to commits
12: Log Request committed
13: TX Proposal InspectRequest
14: Log Response committed
15: TX Proposal InspectResponse
Listen to commits
Listen to commits
16: Inspect Request + Inspect Response by Org Y committed
Listen to commits
17: Inspect Response + Inspect Request by Org X committed

Fig. 4. Sequence diagram showing the asynchronous execution flow of one data exchange between two organizations.

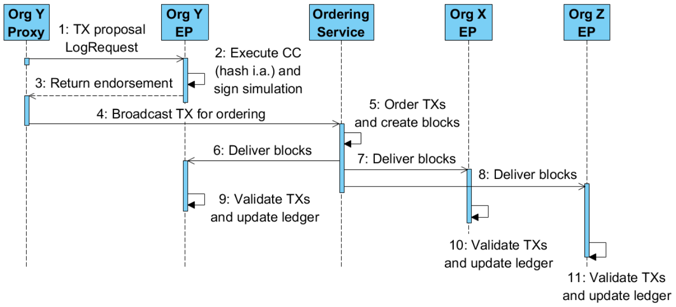
Org Y Proxy
Org Y EP
Ordering Service
Org X EP
Org Z EP
1: TX proposal LogRequest
2: Execute CC (hash i.a.) and sign simulation
3: Return endorsement
4: Broadcast TX for ordering
5: Order TXs and create blocks
6: Deliver blocks
7: Deliver blocks
8: Deliver blocks
9: Validate TXs and update ledger
10: Validate TXs and update ledger
11: Validate TXs and update ledger

Fig. 5. Sequence diagram showing the integration of Hyperledger Fabric.

IV. Performance evaluation

A. Setup

A prototype is designed in order to evaluate the performance of the proposed logging mechanism. Hyperledger Fabric v1.3 is used together with the Go programming language to write CCs. On top of Fabric’s components, which are setup using containers and command line instructions, a Node.js client and proxy process are written incorporating Fabric’s Node SDK [23]. The result is a fully containerized application which is deployed in a Kubernetes cluster. Within this cluster, all pods belonging to one organization are deployed on the same machine. It is important to note that a Kubernetes cluster is only used to ease the evaluation process, i.e. to rapidly scale replicas, but that this setup could not be used for real collaboration scenarios as the Kubernetes master nodes could remove crucial container instances, e.g. the EP of an organization. Figure 6 shows an overview of the containers that

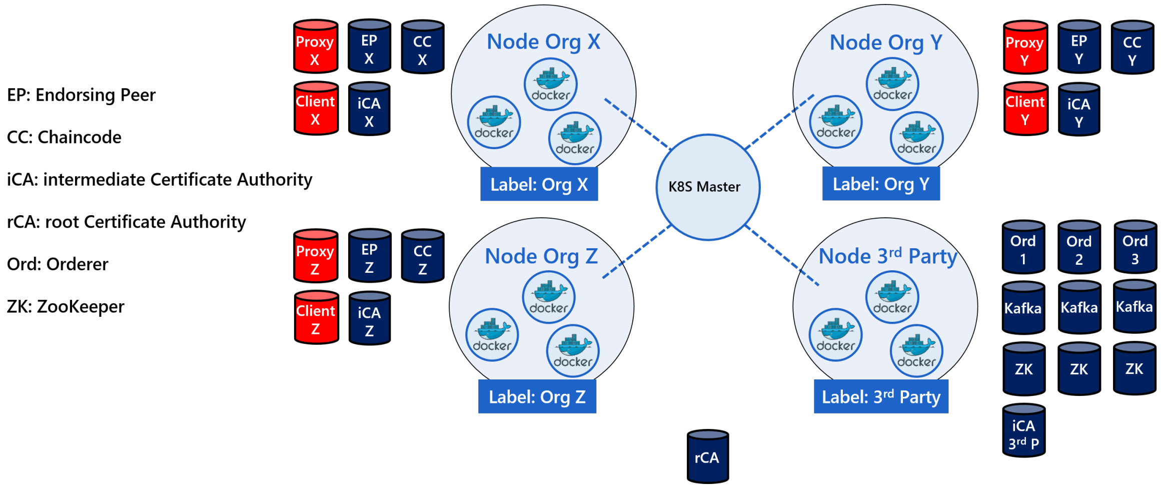

Fig. 6. An overview of the evaluation setup when the Kafka ordering service is used.

are required to run experiments for a collaboration consisting of three organizations. The blue containers are required for setting up Fabric while the red containers are required to share an organization's internal services. Transport Layer Security (TLS) is not enabled for these experiments as there is no research challenge in setting this up. Fabric's default key-value store LevelDB [24] is used as database in the peers' memory.

The minimum number of organizations required is three based on the following reasoning. Multiple organizations can collude to tamper one or more blocks in their local copy of the chain and recalculate all the hash values of the subsequent blocks. This hack only works when at least 50% of the organizations do this, because the chain contained by the majority of the organizations will be seen as the correct one. This is under the assumption that the Kafka cluster does not exist anymore, as otherwise the original chain could be regenerated. Although theoretically possible, this scenario is assumed to be unlikely, because it requires different organizations to agree on corruption. A scenario with two organizations is thus not allowed, because one organization then has a 50% share of the network, meaning it can easily recalculate its local chain. A possible solution for this could be that a third party is added to supervise the collaboration, i.e. a non-endorsing peer in Fabric's terminology. Such a peer would only keep a copy of the ledger and store the TXs without interacting with the network. There is no maximum number of organizations, but a collaboration between ten organizations already seems to be a lot from a practical point of view.

According to the documentation, at least four Kafka brokers and three, five or seven ZooKeeper nodes need to be available [25]. For this use case however, a minimum of three Kafka brokers is sufficient, based on following reasoning. The minimum number of in-sync replicas needs to be two in order to avoid a single point of failure. The number of replicas needs to be three in order to retain the minimum number of in-sync replicas, i.e. keep the channel readable and writable, when one Kafka broker fails. However, when there is such a failure, no channel can be created as Kafka topic creation requires all replicas to be alive. For this use case, channel creation is not necessary anymore once there is a channel available. Requiring at least four Kafka brokers is therefore not strictly necessary here.

Each organization's EP needs to have a signing identity, i.e. private key, to sign TX proposals. To allow the network to verify its digital signature, the EP also needs a certificate. A certificate chain consisting of an organization's intermediate certificate authority and a root certificate authority is used in this setup, both deployed using Fabric's CA server implementation. As signature creation and verification takes time, they will certainly have an impact on time measurements. It is important to note that in a real collaboration scenario, the root certificate authority shown in Figure 6 will not be there as the world's largest certificate authorities will be used as root of trust. After all, organizations could simply use the certificate coupled to their domain to issue certificates to their peers, while the certificates for the ordering service could be granted to a third party hosting the ordering service.

B. Measurements

The same example collaboration shown in Figure 1 is used to perform measurements, i.e. Org Y wants to pull data from Org X and Org Z. Each organization runs an Ubuntu 18.04 VM on a 2.4GHz machine with four VMware vCPUs, 4GiB of RAM and a hard disk partition of at least 16GiB. The VM for the third party is given 8GiB of RAM. Network delay is furthermore emulated using the *tc* command. As the average *ping* round-trip time to Amazon servers in Western Europe varies around 30 ms [26], the artificial delay of the egress packet scheduler is set to 15ms.

In fact, a lot of parameters can be tweaked for these experiments, not only latency L, but also block size BS, block creation timeout BT, number of organizations O, number of data requests from the equipment builder to the manufacturers

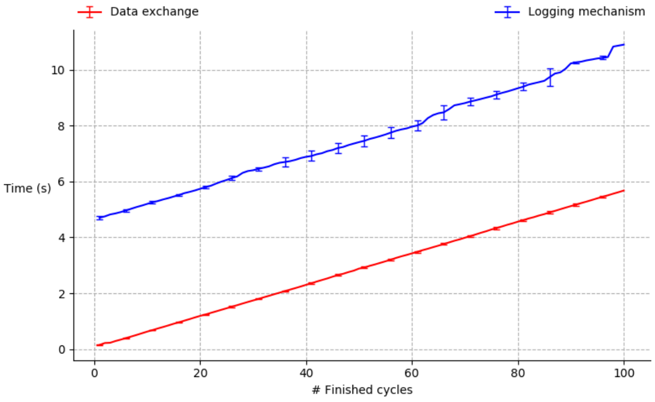

Fig. 7. The asynchronous approach has an average throughput of 17.6 data exchanges per second. The logging mechanism is able to keep up with the speed of the data exchange process.

per second E and size of the data S. The following parameter setting is determined for the use case examined in this paper:

- L is set to 15ms as explained above.
- BS is limited to 512 KiB. The maximum size of an individual TX is a few kilobytes at most as the TXs' payload does not contain raw request/response data, resulting in blocks with around 100 TXs. The allowed maximum number of TXs per block is set to a larger value in order for it to be no separate block-cutting trigger.
- BT is set to 2 seconds, i.e. a partially filled block will be cut 2 seconds after the first TX of the interval arrived.
- O is set to 3 for the example collaboration.
- E is set to 20. This parameter limits the number of requests to the different manufacturers on channel 2. The goal is to send 10 calls per second to each manufacturer.
- S is set to 1500 bytes, i.e. the internal service returns 1500 random hexadecimal characters in each JSON response.

During the experiments, data requests are sent to manufacturer A and manufacturer B in an alternated way. The duration of each data exchange cycle is measured at the client of the equipment builder, while the duration of each logging cycle is measured at its proxy. It is important to note that the alternation between the different manufacturers is non-blocking, i.e. data requests are sent periodically with rate E using Node.js its `setInterval` function. Five runs are executed for the experiments, each time with a clean deployment, and the average of these measurements is stored. The creation of the different CC containers is not included in the measurements as they are started beforehand. Finally, it is important to mention that I/O operations are kept to a minimum, i.e. no console messages in the Node.js processes appear and measurements are written to disk when the experiment finishes. However, Fabric's logging information for the peer and orderer container is set to DEBUG. This is needed for our implementation as debug information is required for coordinating the start of Fabric's network.

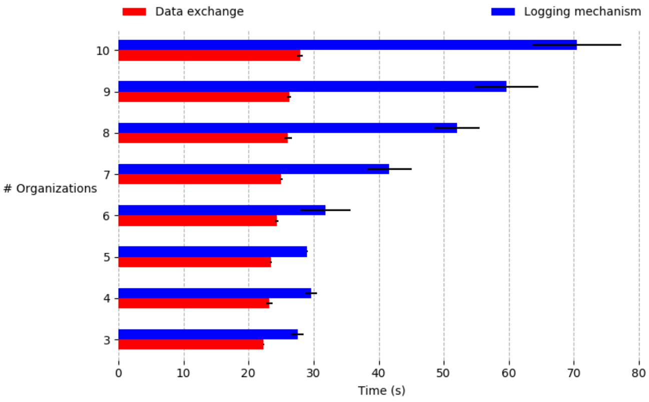

Fig. 8. The decoupling of the data exchange and logging mechanism processes emerges when the number of organizations is scaled.

TABLE I
THROUGHPUT VALUES FOR AN INCREASING NUMBER OF ORGANIZATIONS

	Organizations							
	3	4	5	6	7	8	9	10
T_o **(Exch./s)**	17.9	25.8	34.1	41.1	47.9	53.7	61.0	64.4
T_r **(%)**	89.6	86.1	85.3	82.2	79.8	76.8	76.2	71.6

Figure 7 shows the results of the asynchronous approach. The average values are drawn together with error bars at each 5-th data point showing the standard deviation. The advantage of this approach is immediately clear: 100 data exchanges occur in about 5.68 seconds, resulting in an average throughput of 17.6 exchanges per second approximately. The performance is further evaluated by scaling the number of organizations O. The same experiment is executed here, i.e. there is one equipment builder which sends data requests to the different manufacturers in an alternated way. Based on the reasoning in the previous section, O is scaled from three to ten, i.e. the number of manufacturers ranges from two to nine. The number of data exchanges and the rate with which they are sent E are adapted in order to obtain equivalent scenarios where each manufacturer needs to send 200 responses. This means that E ranges from 20 to 90 calls per second and the number of exchanges from 400 to 1800. Other parameters are kept constant and the same number of Kafka, ZooKeeper and orderer nodes are used, i.e. 3 replicas of each type are deployed.

Figure 8 presents the obtained results. The experiment is repeated five times for each value of O and the corresponding average values and standard deviations are drawn. It shows that the data exchange process only takes a little bit more time to complete for larger collaborations, while the time needed for the logging mechanism increases significantly. The advantage of the asynchronous approach is clear as the data exchange process scales very well. The equipment builder, processing the data, observes almost no additional delay. Table I shows the average throughput T_o observed at the equipment builder and the throughput rate T_r. The latter is the rate between T_o and E, whereby E can be seen as the theoretical maximum throughput value as data requests are periodically sent with this rate. The results show that T_o increases significantly because more and more manufacturers will send their responses in the same time interval. The obtained values show that tens of data exchanges per second can be completed. Table I also shows that T_r decreases. This can be expected as the equipment builder has to execute an increasing number of operations for the logging mechanism, which means that the data exchange process gets delayed. Finally, the size of the chain is around 34 MiB when $O = 10$, meaning the average size of a TX in the system is $\frac{34}{1800 \cdot 4} = 4.8$ KiB.

V. Conclusions and future work

In this paper, a logging mechanism for cross-organizational collaborations is proposed, which enables organizations to create an irrefutable log file. When the logging mechanism is correctly executed, no disputes are possible about which data was exchanged. When an honest organization detects that something is wrong with the logging procedure, either due to the presence of a dishonest organization, due to malfunctioning or due to an attack, it can assess whether it is still useful to be part of an unreliable collaboration setup. The logging mechanism does not heavily interrupt the data exchange process as all logging operations are executed asynchronously, allowing to reach tens of data exchanges per second, even when the number of organizations is increased.

The proposed architecture will be further investigated in future work. The deployment, i.e. setup and tear down, of this logging mechanism in a rapid, ad hoc way will be researched as well as the associated cost in terms of time and money. Finally, a dynamic scenario, where organizations can join and leave the collaboration when needed, must be investigated.

Acknowledgment

This paper is written in the context of the FUSE project [27], 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] “Hyperledger Fabric.” [Online]. Available: https://www.hyperledger.org/projects/fabric

[2] K. Wust and A. Gervais, “Do you need a blockchain?” in *Proceedings - 2018 Crypto Valley Conference on Blockchain Technology, CVCBT 2018*, no. i, 2018, pp. 45–54.

[3] E. Androulaki, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, A. Barger, S. W. Cocco, J. Yellick, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, and G. Laventman, “Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains,” in *Proceedings of the Thirteenth EuroSys Conference on - EuroSys ’18*, 2018, pp. 1–15.

[4] J. Sousa, A. Bessani, and M. Vukolic, “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 2018*, 2018, pp. 51–58.

[5] A. Bessani, J. Sousa, and E. E. Alchieri, “State Machine Replication for the Masses with BFT-SMaRT,” in *Proceedings - 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014*, 2014, pp. 355–362.

[6] “Peers.” [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.3/peers/peers.html

[7] X. Wang, X. Xu, L. Feagan, S. Huang, L. Jiao, and W. Zhao, “Inter-Bank Payment System on Enterprise Blockchain Platform,” in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 614–621.

[8] W. Zhang, Y. Yuan, Y. Hu, S. Huang, S. Cao, A. Chopra, and S. Huang, “A Privacy-Preserving Voting Protocol on Blockchain,” in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, no. April, 2018, pp. 401–408.

[9] T. Mikula and R. H. Jacobsen, “Identity and Access Management with Blockchain in Electronic Healthcare Records,” in *Proceedings - 21st Euromicro Conference on Digital System Design, DSD 2018*, 2018, pp. 699–706.

[10] H. Kinkelin, V. Hauner, H. Niedermayer, and G. Carle, “Trustworthy Configuration Management for Networked Devices using Distributed Ledgers,” in *IEEE/IFIP Network Operations and Management Symposium: Cognitive Management in a Cyber World, NOMS 2018*, 2018, pp. 1–5.

[11] M. Grabatin and W. Hommel, “Reliability and Scalability Improvements to Identity Federations by managing SAML Metadata with Distributed Ledger Technology,” in *IEEE/IFIP Network Operations and Management Symposium: Cognitive Management in a Cyber World, NOMS 2018*, 2018, pp. 1–6.

[12] Z. Gao, Y. Fan, C. Wu, J. Zhang, and C. Chen, “DSES: A Blockchain-Powered Decentralized Service Eco-System,” in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 25–32.

[13] K. Bhaskaran, P. Ilfrich, D. Liffman, C. Vecchiola, P. Jayachandran, A. Kumar, F. Lim, K. Nandakumar, Z. Qin, V. Ramakrishna, E. G. Teo, and C. H. Suen, “Double-Blind Consent-Driven Data Sharing on Blockchain,” in *Proceedings - 2018 IEEE International Conference on Cloud Engineering, IC2E 2018*, 2018, pp. 385–391.

[14] S. Kiyomoto, M. S. Rahman, and A. Basu, “On Blockchain-Based Anonymized Dataset Distribution Platform,” in *Proceedings - 2017 15th IEEE/ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2017*, 2017, pp. 85–92.

[15] J. Duan, A. Karve, V. Sreedhar, and S. Zeng, “Service Management of Blockchain Networks,” in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 310–317.

[16] H. Gupta, S. Hans, K. Aggarwal, S. Mehta, B. Chatterjee, and P. Jayachandran, “Efficiently Processing Temporal Queries on Hyperledger Fabric,” in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 1489–1494.

[17] F. Benhamouda, S. Halevi, and T. Halevi, “Supporting Private Data on Hyperledger Fabric with Secure Multiparty Computation,” in *Proceedings - 2018 IEEE International Conference on Cloud Engineering, IC2E 2018*, 2018, pp. 357–363.

[18] M. Brandenburger, C. Cachin, R. Kapitza, and A. Sorniotti, “Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric,” IBM Research, Zurich, Tech. Rep., 2018.

[19] D. Marpe, T. Wiegand, and G. J. Sullivan, “The H.264/MPEG4 advanced video coding standard and its applications,” *IEEE Communications Magazine*, vol. 44, no. 8, pp. 134–142, 2006.

[20] “Transaction Flow.” [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.3/txflow.html

[21] “Docker security.” [Online]. Available: https://docs.docker.com/engine/security/security

[22] “Ledger.” [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.3/ledger.html

[23] “Hyperledger Fabric SDK for Node.js.” [Online]. Available: https://fabric-sdk-node.github.io/release-1.3

[24] “LevelDB.” [Online]. Available: https://github.com/google/leveldb

[25] “Bringing up a Kafka-based Ordering Service.” [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.3/kafka.html

[26] “CloudPing.info.” [Online]. Available: https://www.cloudping.info/

[27] “FUSE: Flexible federated Unified Service Environment.” [Online]. Available: https://www.imec-int.com/en/what-we-offer/research-portfolio/fuse