FACULTY OF ENGINEERING
AND ARCHITECTURE

Mitigating Potential Trust Issues in Ad Hoc Collaborations

Laurens Van Hoye

Doctoral dissertation submitted to obtain the academic degree of
Doctor of Computer Science Engineering

Supervisors
Prof. Filip De Turck, PhD - Prof. Bruno Volckaert, PhD
Department of Information Technology
Faculty of Engineering and Architecture, Ghent University

December 2022

GHENT
UNIVERSITY

ISBN 978-94-6355-667-5
NUR 986
Wettelijk depot: D/2022/10.500/108

Preface

"But, dear readers, all this well-intentioned advice is just a run-up to my tenth and final golden rule which transcends all others.

Don't listen to advice."

– Robbert Dijkgraaf, Ten rules for success, NRC, May 30th 2009

This quote is extracted from an article written by the renowned mathematical physicist Robbert Dijkgraaf. It describes ten action points people (considering) working in an academic environment should do to become successful. The aim of inserting this citation is twofold.

Although I do not know dr. Dijkgraaf personally, he is a truly inspiring person, as became clear to me in recent years when watching episodes of DWDD University, a Dutch television show popularizing science among a wide audience. He is able to fascinate people like no one else, really someone to look up to, and therefore a short tribute is included here. The fundamental research topics he discusses, driven by mathematics and physics, are of the most complex category and therefore in no way comparable with the more practical topics discussed here. Taking into account this wider picture makes me feel humble, but also proud at the same time, as I have been able to experience and contribute to academia myself during a period of my life.

On the other hand, the quote itself is quite funny as the article has only one goal, to provide advice, while you should thus not listen to advice, which is an advice itself. I guess this setup was intended this way. Apart from that, it definitely is something to think about. When you ask people their opinion about doing research or obtaining a PhD, you get a diverse set of responses. Listening to other opinions is definitely important, but in the end, I learned that it is always important to listen to yourself, in this case to grab the opportunity. Although it was not always easy, especially during the COVID-19 period, I really enjoyed the PhD trajectory. I improved my research skills, writing skills, speaking skills and last but not least I got to know myself better. If people would ask me whether doing a PhD brings value, I would strongly confirm, but remind you, this is only an advice.

First and foremost, I would like to thank my promotors prof. dr. Filip De Turck and prof. dr. Bruno Volckaert. For sure, you gave me the opportunity to start this journey at IDLab, but your contribution entails way more than this prerequisite alone. Bruno, thank you for the interesting discussions we had, either in person or online. You have the same earlier mentioned talent to inspire people. Each time I felt to be stuck at some point, you provided new pointers to start from, a bit of fresh air you could say. Even at home they knew after a while whether I had talked to you lately, based on my mood regarding the PhD. I feel lucky to have had the opportunity to work with you for five years. Every person can identify a few people who have had a major influence on their career, and you are definitely part of my selection. Filip, you breath teaching, engaging students, fostering collaboration, managing people, everything related to being a professor. Your feedback, either on articles or cover letters, was of incredible value, because you are able to understand issues quickly and provide constructive answers which are spot-on. We furthermore communicated quite a lot to discuss practicalities related to the C/C++ project for the Programming (PGM) course you are teaching. I am happy to be one of the few people who can say that they were able to send, but also receive, emoticons when communicating with a professor. Furthermore, I will never forget that both of us received chocolates from a student for solving an urgent issue. Both of you, a massive thanks, and we will of course keep in touch.

Other exceptional colleagues have been part of my journey. The gentleman of office 200.012, more specifically Sander Borny, Vincent Bracke, Tom Goethals, Jerico Moeyersons, Leandro Ordonez Ante, Stefano Petrangeli, Merlijn Sebrechts, Wim Van de Meerssche and Thijs Walcarius, you are both very friendly and smart guys. Coming to the office, in the pre-pandemic period, was never boring due to

the unique blend of personalities and backgrounds present. I am grateful for all enjoyable conversations we had. Also thanks to Sarah Kerkhove who, at the time, could be considered the Swiss Army knife for software development. Both Pieter-Jan Maenhaut and Gregory Van Seghbroeck need to be acknowledged for their help at the start of the research process, especially for constructing the FWO proposal. Furthermore, I would like to thank Laurens D'hooge, Bart Moons and Jeroen van der Hooft for their contribution to PGM and the relatively short yet pleasant interactions we had. A special thanks goes to dr. Tim Wauters for thoroughly reviewing our articles. The quality of all presented publications improved significantly due to your constructive feedback. Finally, I would like to thank Mathias De Brouwer for discussing particularities of PhD live and the further continuation of our friendship.

Of course, my family needs to be credited as well for the immense support they have given me throughout the years. My father Georges, my mother Marianne, and my brother Vincent. I guess this preface is not needed anymore to make clear how much I appreciate you. You are irreplaceable.

This year marks my tenth anniversary at UGent, from 2012 starting the bachelor to 2022 finishing the PhD. It has been a great time so far and I would therefore like to thank the university UGent, the research group IDLab, and Piet Demeester in particular as head of the lab. I am really proud to have been a part of the institute. Last but not least, enjoy reading the dissertation, and feel free to contact me at any point in time.

Ghent, October 2022
Laurens Van Hoye

Table of Contents

List of Figures

List of Tables

List of Acronyms

A

ABAC	Attribute-Based Access Control
AI	Artificial Intelligence
API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
AVC	Advanced Video Coding

B

BFT	Byzantine Fault Tolerant
BST	Binary Search Tree

C

CA	Certificate Authority
CC	Chaincode
CCTV	Closed-Circuit Television
CFT	Crash Fault Tolerant
CIDR	Classless Inter-Domain Routing
CNI	Container Networking Interface
CPU	Central Processing Unit
CRI	Container Runtime Interface

D

DB	Database
DNS	Domain Name System

E

EP	Endorsing Peer

F

FPS	Frames Per Second
FUSE	Flexible federated Unified Service Environment

G

GCC	GNU Compiler Collection
GPPL	General-Purpose Programming Language
GPU	Graphics Processing Unit
gRPC	gRPC Remote Procedure Calls

H

HEVC	High Efficiency Video Coding
HTTP	Hypertext Transfer Protocol

I

IaaS	Infrastructure as a Service
ID	Identifier
IP	Internet Protocol
IQR	Interquartile range
IRI	Internationalized Resource Identifier

J

JSON	JavaScript Object Notation
JWT	JSON Web Token

K

KVP	Key-Value Pair

L

LD	Linked Data
LoRa	Long Range
LTS	Long Term Support

M

MPEG	Moving Picture Experts Group

O

OCI	Open Container Initiative
OOM	Out of Memory
OPA	Open Policy Agent
OSN	Ordering Service Node

P

PaaS	Platform as a Service
PAT	Protection API Token
PBFT	Practical Byzantine Fault Tolerant
PID	Process Identifier
PLEG	Pod Lifecycle Event Generator
PSNR	Peak Signal-to-Noise Ratio

Q

QR	Quick Response

R

RAM	Random-Access Memory
RBAC	Role-Based Access Control
RBI	Received Block Indication
RDF	Resource Description Framework
REST	Representational State Transfer
RPT	Requesting Party Token
RTT	Round-Trip Time

S

SaaS	Software as a Service
SBI	Sent Block Indication
SDK	Software Development Kit
SGX	Software Guard Extensions
SHA	Secure Hash Algorithm
SLA	Service Level Agreement
SQL	Structured Query Language
SSD	Solid-State Drive

T

TCP	Transmission Control Protocol
TLS	Transport Layer Security
TTP	Trusted Third Party
TX	Transaction

U

UMA	User-Managed Access
URL	Uniform Resource Locator

V

VM	Virtual Machine
VPN	Virtual Private Network
VXLAN	Virtual Extensible Local Area Network

Y

YAML	YAML Ain't Markup Language

Samenvatting – Summary in Dutch –

Organisaties kunnen een dringende reden hebben om een samenwerking op te starten. Wanneer levens in gevaar zijn of ernstige financiële schade dreigt, zijn spoedig oplossingen vereist. Een terroristische aanval is een evidente aanleiding voor ad-hoc-kennisdeling. Veelal is het zo dat relevante databronnen (bijvoorbeeld lijsten van personen aanwezig in het gebouw, camerabeelden van de omgeving, tracking-gegevens van mobiele telefoons, etc.) enkel beschikbaar zijn door afzonderlijke applicaties te raadplegen die verspreid zijn over verschillende organisaties. Vanwege de urgentie is het daarom vereist om een duidelijk georganiseerde procedure te volgen die toelaat om deze interne databronnen binnen enkele minuten samen te brengen. Hoewel de samenstelling van een dergelijke ad-hoc-federatie realiseerbaar is vanuit een technisch oogpunt, moet aandacht besteed worden aan specifieke uitdagingen die deze urgente samenwerkingen tussen organisaties met zich meebrengen. Meer specifiek kan niet zomaar verondersteld worden dat de deelnemers aan de samenwerking, een groep die gedurende de samenwerkingsperiode kan wijzigen, elkaar kennen en ***vertrouwen***, aangezien deze ongepland tot stand wordt gebracht. Dit betekent dat, in tegenstelling tot een langdurige samenwerking waarbij contracten op voorhand kunnen worden opgemaakt, beoordelingen en overeenkomsten terstond moeten worden samengesteld. ***Vertrouwen*** is dus het sleutelwoord in een aantal belangrijke lastigheden die veroorzaakt worden door dit type van tijdelijke samenwerking en die besproken worden in dit proefschrift.

De eerste kwestie die wordt behandeld, heeft betrekking op het delen van interne databronnen. De ad-hoc-samenwerking maakt dat het voor organisaties ondoenlijk is om de handleiding van elke beschikbare databron te bestuderen en dus is het vereist dat deze vlot navigeerbaar zijn. Dit realiseren is onmogelijk zonder een bepaald niveau van standaardisatie te introduceren waarmee vastgelegd wordt hoe interfaces op generieke wijze door clients kunnen worden aangesproken. Hoofdstuk 2 en Appendix A presenteren daarom een prototype dat laat zien hoe een volledig geautomatiseerd scenario voor gegevensuitwisseling geïmplementeerd kan worden. Een belangrijk onderdeel hiervan is een mechanisme dat zorgt voor het genereren van onomstotelijke bewijzen van alle gegevensuitwisselingen die tijdens een samenwerking plaatsvinden. Deze bewijzen worden opgeslagen in een daarvoor passende gedistribueerde databank die voor alle belanghebbenden inzichtelijk maakt welke (types) uitwisselingen onderling plaatsvinden. Een dergelijk mechanisme is noodzakelijk, aangezien deelnemende organisaties mogelijk geen volledig ***vertrouwen*** hebben in alle entiteiten die onderdeel uitmaken van de samenwerking. Onenigheden moeten ten alle tijden vermeden worden, aangezien deze ertoe kunnen leiden dat organisaties (i) deelname aan gegevensuitwisselingen met anderen weigeren of (ii) zich terugtrekken uit lopende uitwisselingen. Evaluatie toont aan dat de voorgestelde oplossing, die toelaat om inbreuken in verband met logging te detecteren, asynchroon dient uitgevoerd te worden ten opzichte van het proces voor gegevensuitwisseling, wanneer de performantie van dit proces een factor van belang is.

De tweede kwestie die wordt behandeld, heeft betrekking op het veilig plaatsen van software in de organisatieoverschrijdende clusteromgeving. Wanneer geen controlemechanismes aanwezig zijn, is het mogelijk voor een deelnemende organisatie om geconfronteerd te worden met een kwaadwillende externe partij die kwaadaardige software in zijn domein installeert. Deze potentiële kwetsbaarheid dient opgelost te worden om bijbehorend ***vertrouwen***sprobleem weg te werken. In Hoofdstuk 3 wordt daarom een gedetailleerd stappenplan voorgesteld waarmee relatief kleine deployments van hoogstens tien containers op veilige wijze kunnen worden geplaatst. Aangezien de veelgebruikte orchestrator Kubernetes gekozen is doorheen dit proefschrift om containers te beheren, wordt de werking van de Kubelet, een computerprogramma dat op elke machine zorg draagt voor het daadwerkelijk opstarten van containers, nauwkeurig bestudeerd en beschreven. De uitbreidingsmogelijkheid die hiermee wordt blootgelegd, wordt vervolgens gebruikt om een integratie van het autorisatieprotocol UMA 2.0 te bespreken. De resulterende architectuur laat lokale administrators toe om voorstellen tot uitrol van software op asynchrone wijze te verifiëren en te autoriseren. Evaluatie toont aan dat het mogelijk is om met het geïmplementeerde prototype binnen een seconde containers te plaatsen wanneer verificatie volledig geautomatiseerd is. Deze tijdsduur zal significant toenemen wanneer manuele verificatie door de hosting administrators wenselijk is.

De derde kwestie die wordt behandeld, heeft betrekking op het plaatsen van containers in een organisatieoverschrijdende clusteromgeving die typisch sterk heterogeen is. Het kan lastig zijn voor de cluster operator, de verantwoordelijke partij voor clusterbeheer, om workloads te (her)verdelen aangezien de machines en de indeling van het cluster volledig onbekend zijn. Het stap voor stap verkennen van de beschikbaarheid van resources in het cluster is dus een essentiële voorwaarde om inschattingsfouten ten aanzien van de (her)verdeling van containers, die zorgen voor onwenselijke performantieproblemen, te voorkomen. Hoofdstuk 4 bespreekt daarom verschillende types probes. Dit zijn metingen die lokaal worden uitgevoerd om inzicht te verwerven in bijvoorbeeld de computationele kracht van een machine. Door het inzetten van een zwerm van probes kan op verschillende tijdstippen snel een overzicht bekomen worden van de technische mogelijkheden van machines, hun onderlinge verbindingen, en eventuele verborgen performantierestricties die zijn ingeschakeld door hosting administrators vanwege een gebrek aan volledig ***vertrouwen***. Een integratie van dit probing concept in de Kubernetes scheduler wordt daarom voorgesteld. Evaluatie van een illustratief scenario waarbij workloads dienen te worden verplaatst, toont aan dat de performantie van een applicatie significant zou kunnen stijgen: met een factor vijf, tien of honderd in vergelijking met wanneer deze probes niet gebruikt zouden worden.

De vierde kwestie die wordt behandeld, heeft eveneens betrekking op het verdelen van workloads. Hierbij wordt gekeken naar (her)verdelingsopdrachten waarbij het eindresultaat organisatieoverschrijdende verwerking van data mogelijk moet maken. Het verwerven van een technisch overzicht van het cluster is slechts één deel van de oplossing voor dergelijke gevallen. Onbekende niet-technische vereisten kunnen beslissingen over (her)verdeling van containers ook beïnvloeden. Deze vereisten moeten dus ontdekt worden tijdens het verdelingsproces om zinloze pogingen tot plaatsing te vermijden. Hoofdstuk 5 presenteert daarom rangschikkingen voor bijdrage en beloning die dienen als uitbreiding op het in vorig hoofdstuk voorgestelde verdelingsproces. Door middel van deze rangschikkingen is het mogelijk voor organisaties om zowel technische (hints die evaluatie door probes sturen) als niet-technische vereisten op een toegankelijke wijze te communiceren. Vereisten van het laatste type laten toe om limieten in bijdrage te specifiëren en garanties over de erkenning van een contributie te eisen. Op deze manier kunnen organisaties, die deelnemen aan een proces waarbij data organisatieoverschrijdend verwerkt wordt, verdelingsbeslissingen sturen, hetgeen ervoor zorgt dat hun ***vertrouwen*** in de samenwerking toeneemt. Het mogelijk maken voor betrokken belanghebbenden om voorkeuren onder elkaar te uiten, laat een onderhandelingsprocedure toe. Evaluatie van een illustratief scenario waarbij workloads dienen te worden verplaatst, laat zien dat deze vereisten een stevige impact kunnen hebben op de performantie van een applicatie, maar ook dat onderhandeling het mogelijk maakt om dit (deels) te verhelpen.

De voorgestelde oplossingen, of delen daarvan, kunnen bediscussieerd en in vraag gesteld worden, zoals gebruikelijk is in wetenschap. Dit reflectieproces kan ervoor zorgen dat extra inzicht verworven wordt. In Hoofdstuk 6 wordt daarom voor elk voorstel een aantal noemenswaardige overwegingen besproken. Bovendien worden pistes voor verder onderzoek geïdentificeerd, aangezien ad-hoc-samenwerkingen tussen organisaties aanleiding geven voor een reeks interessante uitdagingen die opgelost dienen te worden.

Summary

Organizations may have an urgent reason to initiate a collaboration. Scenarios in which lives are at risk or severe financial damage is at stake require solutions quickly. A terrorist attack is an obvious cause for ad hoc knowledge sharing. Typically, relevant data sources (e.g. lists of persons present in building, camera feeds of vicinity, cellphone tracking information, etc.) are only available through isolated applications scattered over different organizations. Due to the urgency, it is therefore required to follow a streamlined operation allowing these internal data sources to be merged in a matter of minutes at most. Although the composition of such an ad hoc federation is feasible from a purely technical point of view, attention needs to be paid to specific challenges imposed by these urgent cross-organizational collaborations. More specifically, the collaboration participants, a group which could possibly change over time, may not necessarily know and ***trust*** each other as the collaboration needs to be enabled ad hoc. This means that, contrary to a long-lasting collaboration in which contracts can be arranged upfront, evaluations and agreements need to be constructed on the fly. A number of significant ***trust*** difficulties introduced by this type of temporary collaboration are addressed throughout this dissertation.

The first difficulty is related to the sharing of internal data sources. The ad hoc nature of the collaboration requires these data sources to be easily navigable, i.e. without organizations having to study the manual of each available endpoint. This is impossible without a certain level of standardization dictating how interfaces can be consumed by clients in a generic way. Chapter 2 and Appendix A therefore present a prototype showing how a fully automated data exchange scenario could be implemented. A crucial part of this setup is a logging mechanism generating irrefutable proofs of all data exchanges that happen during a collaboration. As these proofs are stored in

a tailored distributed database, it is clear for all stakeholders which (types of) exchanges occur between them. Such a mechanism is required, as participating organizations may not fully ***trust*** all entities present in the collaboration. Disputes should be prevented in any case as they may cause organizations (i) to refuse to participate in data exchanges with others or (ii) to withdraw from ongoing ones. Evaluation shows that the presented logging solution, allowing to detect infringements, should be executed asynchronously with respect to the data exchange process if performance is considered relevant.

The second difficulty is related to the secure deployment of software in the cross-organizational cluster environment. When no verification mechanisms are present, it is possible for a participating organization to be faced with a malicious external entity willing to host malicious software in its domain. Tackling this potential vulnerability is required for the associated ***trust*** issue to be resolved. Chapter 3 therefore proposes a detailed container deployment flow for relatively small deployments of at most ten containers. The widely used container orchestrator Kubernetes is chosen throughout this dissertation and therefore the operation of the Kubelet, which is a node agent responsible for the actual deployment of containers, is closely examined and described. Using the extension point identified in this analysis, an integration of the authorization protocol UMA 2.0 is discussed. The resulting architecture allows local administrators to verify and authorize deployments asynchronously. Evaluation shows that the implemented prototype allows containers to be deployed with a subsecond overhead in case verification is fully automated. This duration will increase significantly in case manual verification by the hosting administrators is desirable.

The third difficulty is related to the deployment of containers in a, typically highly heterogeneous, cross-organizational cluster environment. The cluster operator, being responsible for cluster management, may have a challenging time (re)scheduling workloads as the nodes and the cluster layout will be completely unknown. Gradually exploring resource availability in the cluster is thus an essential prerequisite to prevent scheduling mistakes leading to costly performance issues. Chapter 4 discusses types of probes, which are locally executed code fragments, that can be used to obtain insight in, for example, the computational power of a node. The deployment of a

probe swarm can then quickly provide an overview, at different points in time, of the technical capabilities of nodes, their interconnections, and possible hidden performance restrictions enabled by the hosting administrators due to a lack of full ***trust***. An integration of this probing concept into the Kubernetes scheduler is proposed. Evaluation of an illustrative rescheduling scenario shows that application performance could increase significantly: a factor five, ten or hundred compared to the scenario in which no probes are used.

The fourth difficulty is also related to the scheduling of workloads. It considers (re)scheduling assignments in which the end result should become a cross-organizational data pipeline. Obtaining a technical overview of the cluster is only one part of the solution for these cases. Unknown non-technical requirements may also influence (re)scheduling decisions. These requirements need to be discovered during the scheduling process to prevent useless deployment attempts. Chapter 5 therefore extends the scheduling process proposed in the previous Chapter by introducing a load and reward ranking. Using these rankings, it is possible for organizations to communicate both technical (probe evaluation hints) and non-technical requirements in an accessible way. Requirements of the latter type allow to specify contribution limits and to demand guarantees about the recognition of a contribution. This way, organizations participating in a cross-organizational data pipeline are able to steer scheduling decisions, causing their ***trust*** level to increase. Allowing involved stakeholders to express preferences among each other paves the way for a negotiation procedure. Evaluation of an illustrative rescheduling scenario shows that these requirements could severely impact application performance, but also that negotiation allows for mitigation.

The proposed solutions, or parts of them, can be discussed and questioned, as is common in science. This reflection process can cause additional insight to come to mind. Chapter 6 therefore discusses, for each of the proposals, a few considerations that are worth mentioning. Furthermore, avenues for future research are identified, as the cross-organizational collaboration topic allows for an abundance of interesting challenges to be solved.

1 Introduction

1.1 Urgent Collaborations

A cross-organizational collaboration is the key subject of the research discussed throughout this dissertation. The paragraphs below provide more details on the type of collaboration that is exactly envisioned.

1.1.1 Types of Federations

The coupling of otherwise isolated experimentation testbeds available in research institutes is a well-researched topic. For example, the federation framework proposed by Wauters et al. [1] provides an overview of the Fed4FIRE federation architecture. It covers all required lifecycle management functionality and integrates existing tools, allowing to conduct research experiments over very heterogeneous experimentation facilities worldwide (covering clouds, advanced networking, big data, 5G, smart cities, etc.). Special concern has been given to strong access controls so that industry parties can maintain the confidentiality of their commercially-sensitive experiments. It is possible for new participants to integrate with this federation on two levels: (1) advanced federation, where all testbed

APIs must be implemented so the complete lifecycle of all infrastructural resources can be managed, or (2) light federation, where only service-level APIs must be made available, without full access to the individual infrastructural resources [2]. Typical implementation times for option (1) are in the order of several months, while for (2) this is reduced to several weeks. Although these federation solutions are useful for mid- to long-term collaborations, they do not fit the emergency situations described below. These cases require short-term federations to be set up instantly, i.e. in a matter of minutes at most. Enabling such ad hoc cloud environments is however not straightforward as additional management concerns pop up, especially those dealing with security threats are of primary importance [3]. This dissertation further elaborates on this topic.

1.1.2 Envisioned Emergency Situation

An emergency situation should be mitigated urgently. This sense of urgency may be easily explainable, for example, there are cases in which lives need to be saved, environmental damage needs to limited or the amount of money in lost revenues needs to be reduced. Finding a decent solution under time pressure may however be hard for decision-makers if not all relevant data sources are available. The goal should therefore be to link these data consumers to the correct data producers, preferably immediately. This topic is discussed in the FUSE research project [4]. As the relevant data sources will likely be located and exclusively available in their own isolated domain, there will undoubtedly be service linking issues when they need to be made available to external parties. It is therefore required to federate them using a well-prepared Flexible federated Unified Service Environment (FUSE) providing an automated solution to link heterogeneous domains. The goal of such a platform is thus to allow for smoother integration and to enable the desired successful sharing of data. The research project aims to solve two specific use cases:

Use case in manufacturing: manufacturing pipelines may stall due to machine interruptions, causing a major decline of production capacity. Equipment builders, possessing the knowledge to debug these issues, should therefore be able to quickly manage any of the machines, located in heterogeneous manufacturing environments, in the remote fleet it is supervising. It is important to note that this supervision is only due to a temporary intervention as manufacturers are not willing to grant permanent access to external entities. A

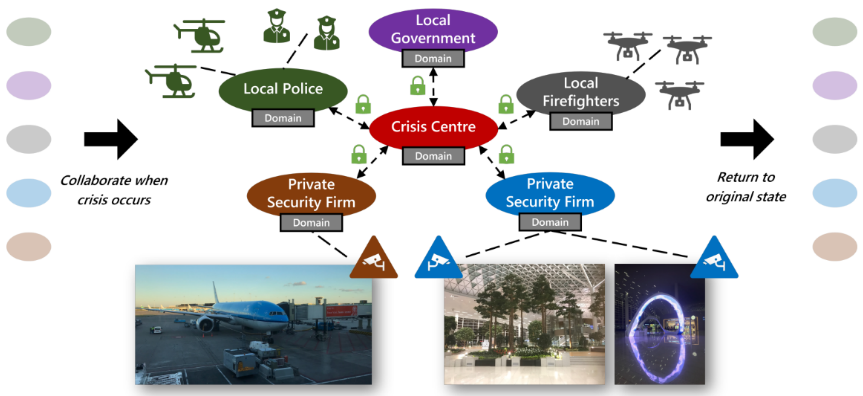


Figure 1.1: A terrorist attack at an airport is an example of an emergency situation requiring the temporary installation of a crisis centre being connected to both public safety instances and private security firms.

solution allowing machines to be monitored and managed remotely may eventually create a backdoor which could be exploited, a risk which increases significantly in case multiple vendors require access.

Use case in control room setup: all kinds of data, especially video streams, need to be merged into a single dashboard solution. Control room operators should therefore be able, at least, to access the data. The possibility to manipulate the data source remotely may also be relevant, for example to allow a camera sensor to be controlled, to lock/unlock electronic doors, etc. A detailed example illustrating a control room scenario is shown in Figure 1.1. Imagine an urgent event, for example a terrorist attack, occurring at an airport. This requires a crisis centre enabling data sharing to be in place, preferably immediately, until the crisis is resolved. The goal of such a centre is to provide local government bodies with an overview of relevant data sources to steer the operation. These data sources are located in different organizational domains, in which they are deployed either locally or in a cloud. Multiple relevant data sources can be identified in this case: video streams from body cams or helicopters produced by the police, video streams from monitoring drones deployed by firefighters, video streams from both indoor and outdoor networks of security cameras possibly maintained by external entities, etc.

These use cases clearly show the need for an urgent cross-organizational collaboration. In general, there are three important characteristics for each of the discussed collaborations in this dissertation:

- The collaboration needs to be set up *ad hoc*, i.e. in a few minutes at most. The level of preparation is limited, as it is not possible to make any detailed cross-organizational agreements upfront.
- The collaboration is *dynamic*, i.e. nodes or even entire organizations may join at any time. In a realistic scenario, the size of the collaboration will likely increase gradually, as each of the organizations will require a different setup time. Likewise, organizations may leave at any time, for example due to disruption of (a part of) the network.
- The collaboration period is *short*, i.e. the collaboration is only needed for a limited amount of time and organizations therefore return to their normal state after a while. The exact duration depends on the specific use case, but will likely be in the order of minutes or hours at most.

The relevance of these types of collaborations is further motivated by Figure 1.2. What is crucial to consider in any kind of collaboration is the level of preparation that can be achieved. When pre-planned scenarios and contingency plans are available, it is possible to increase the level of trust between organizations as they are less vulnerable to misbehavior by other participants. Generally speaking, this can be neatly captured as follows: *"If we were blessed with an unlimited computational ability to map out all possible contingencies in enforceable contracts, trust would not be a problem"* [5]. Achieving a decent level of preparation thus seems to be important to enable fruitful collaboration. However, for the case discussed here, there may arise a lack of flexibility introduced by these plans and agreements. For example, in case a video stream of a camera needs to be shared to display a chemical plant fire, there may be situations which are not covered by pre-set policies as listed in the figure, causing access to required data to be refused. This example illustrates that preparedness may dramatically impact the way an emergency situation will be handled. As data is required immediately as every second counts, it is needed to bypass procedures which would normally be in place. As organizations may therefore doubt whether they should join the collaboration, it is needed to investigate generally applicable enablers which allow them to find an acceptable trade-off between flexibility and vulnerability.

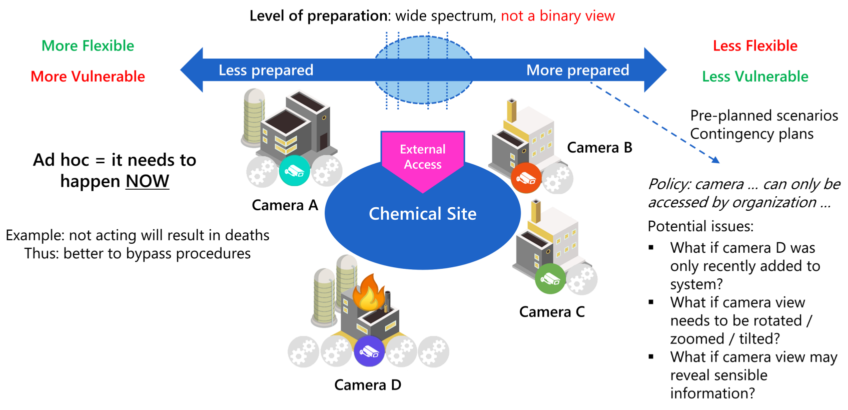

Level of preparation: wide spectrum, not a binary view
More Flexible
More Vulnerable
Less prepared
More prepared
Less Flexible
Less Vulnerable
Pre-planned scenarios
Contingency plans
Ad hoc = it needs to happen NOW
Example: not acting will result in deaths
Thus: better to bypass procedures
Camera A
External Access
Chemical Site
Camera B
Camera C
Camera D
Policy: camera ... can only be accessed by organization ...
Potential issues:
What if camera D was only recently added to system?
What if camera view needs to be rotated / zoomed / tilted?
What if camera view may reveal sensible information?

Figure 1.2: The trade-off which plays a crucial role in the presented use case.

1.1.3 Centralized Operator Setup

A crucial element is the way organizations communicate to initiate such a collaboration. When there is a lack of synchronization, it will take too much time to set up. Therefore it is assumed that a central operator will take the lead during the setup and deployment phase. Orchestration overhead should be kept away from the other organizations, which are possibly inexperienced end-users, as much as possible. The operator may be one of the participating organizations, but it may equally be another organization which acts as an enabling third party offering ”Collaboration as a Service”. The idea behind the central operator is thus purely based on practical reasoning, i.e. to have a single point of contact allowing for smoother communication and orchestration and thus shorter setup times. Note that this does not mean that the contributions discussed throughout this dissertation are fully committed to this centralized setup. Contrary, each of the contributions tries to shift or decentralize power towards the participating organizations to motivate them to join.

Joining the collaboration will be a hybrid process, i.e. it will consist of both manual and automated steps. Initially, an organization should contact the others, for example by joining an online call or chat. It needs to provide details on its online reachability and authenticate through the exchange of certificates. After downloading a FUSE client, it is then possible for the organization to join a VPN network and become a participant of a cross-organizational deploy-

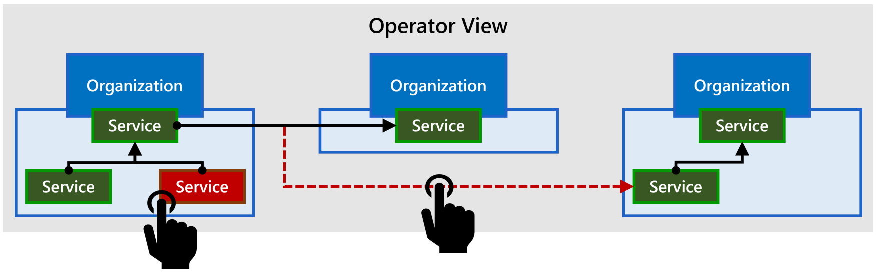


Figure 1.3: The operator is able to deploy, move and delete services across the collaborating organizations. Furthermore, services can be linked together graphically.

ment cluster. A detailed setup, which keeps in mind that (1) the orchestrator environment should be fully isolated from the host, and (2) an organization should be able to join in a few minutes, is outlined by Goethals et al. [6]. It is thus not mandatory for participants to run a specific stack or to comply with specific software as the entire service orchestration environment will be installed at runtime. Given this platform, it is then possible to automate the orchestration of containerized services across the different organizations, for example the software enablers outlined in the contributions. Figure 1.3 shows how the operator is able to manage services via a central overview. Services needed for the collaboration can easily be deployed, moved, deleted and linked. Clearly, manual steps should be avoided as much as possible, but they are mentioned here because they might be present due to the limited level of preparation that is assumed.

As this deployment platform allows write-once-run-anywhere services to be deployed in a cross-organizational setup, it is assumed throughout this dissertation that the networking part required to set up these collaborations is already available. Therefore, no further analysis is available discussing topics related to networking. Evidently, there are relevant venues which need further research. A VPN network is currently needed to provide communication channels for the components of the orchestration platform. These components need to communicate continuously, for example to reconcile desired and actual container state. Further automating the composition of the cluster through software-defined networking is one such venue. Programmability of the network would for instance allow to quickly open and close firewall ports required by the orchestration platform. A completely different venue can be found in the development of zero trust architectures [7]. These architectures tend to shift businesses away from the traditional network view in which the perimeter-based topology is considered trusted. VPN is typically part of this setup, as the trusted network is simply extended through encrypted tunneling. Due to issues with security and scalability, amongst others, it might be more beneficial to trust nothing and to verify everything. The impact of this trend on cross-organizational collaborations and especially on the deployment of the orchestration platform needs further research. What seems to be a risk, is that too fine-grained access control, meaning an extensive set of attributes is required for access to be granted, may lead to the issue of being overprepared as discussed above. The absence of a single attribute, for example a behavioral attribute, may have a disproportionate consequence.

1.2 Problem Statement

From a technological point of view, a cross-organizational collaboration may thus be set up in an ad hoc way making extensive usage of container technology. Services can be orchestrated and connected to local data sources on demand. Apart from a few technological deviations, there does not seem to be much difference between an orchestration environment spanning a single organization and an environment spanning multiple organizations like the one discussed here. There is however a single important difference: potential trust issues due to the required level of flexibility discussed in Section 1.1.2. Three types of situations may be distinguished:

- There is mutual trust between all participating organizations. In this case, the operator has a set of nodes at its disposal, which it can use to distribute any required service. A participating organization contributes resources as much as possible and shares data to support the collaboration as much as needed. Clearly, there is no need to further complicate a collaboration of this type.
- An organization does not trust any or a subset of the organizations. In this case of no trust, it is difficult to imagine that an organization is willing to collaborate at all. For example, internal data may be confidential, and might be abused by others to gain competitive advantage. This case thus illustrates that in order to collaborate, there at least needs to be a certain level

of trust.

- This automatically leads to a more hybrid situation in which an organization has some trust in the participants but no full trust. It is willing to share data, but the actions of external entities need to be supervised to a certain extent.

It is this last type of collaboration that is further studied in this dissertation. Doubts should be eliminated as the goal is to enable and maintain fruitful collaborations that are of utmost importance. As the cross-organizational container orchestration platform mentioned in Section 1.1.3 is used to facilitate collaboration, it is needed to evaluate which additional software components are needed to mitigate potential trust issues within this cluster environment. As *"trust is a function of user perceptions of technical trustworthiness characteristics"* [8], it is thus the challenge to identify and address the most significant concerns organizations willing to contribute may have based on how they might perceive the collaboration framework. Four significant concerns are addressed throughout this dissertation as shown in the overview diagram in Figure 1.4. Each of them focuses on the protection of an honest organization willing to participate in the presence of potentially malicious participants. Ultimately, the goal is to allow organizations to switch on/off the presented enablers based on their desired level of protection. This way, it is possible to increase the confidence level of organizations, which could be inexperienced end-users interpreting the enablers as black boxes, and thus to convince them to join the collaboration. Note that the overall trustworthiness of the entire collaboration system is a multifaceted problem which may definitely be further researched. For example, in terms of resilience, it is needed to prepare the central operator to achieve a production-level grade, possibly through replicated control plane components executing algorithms for leader election and/or through handover of the operator role in case of failure. Given the discussed positioning and purpose of the contributions, it is possible to introduce the four non-trivial research questions, which are discussed below.

Due to the ad hoc and urgent nature of the collaboration it is required to quickly open up internal services providing data relevant for the collaboration. A scenario similar to a break-the-glass procedure, as is typically studied in the context of healthcare [9], needs to be supported. This means that data sources should be available in a short amount of time, preferably immediately, and this can only be realized

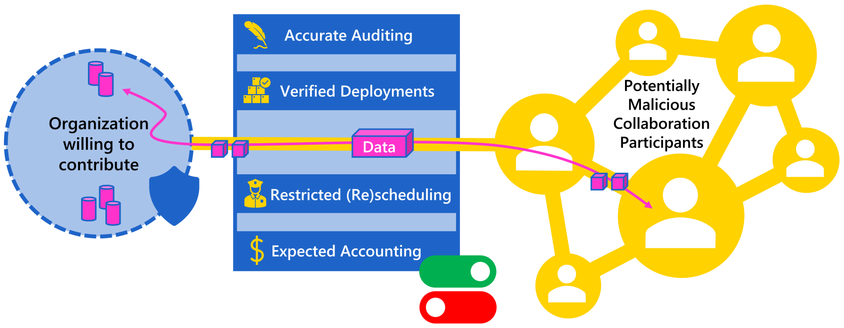

Accurate Auditing
Verified Deployments
Data
Restricted (Re)scheduling
Expected Accounting
Organization willing to contribute
Potentially Malicious Collaboration Participants

Figure 1.4: An organization willing to contribute is able to protect itself by enabling (some of) the extensions proposed throughout the dissertation.

by bypassing protection mechanisms that are in place. Fine-grained access control is thus not possible in such a scenario. Opening up internal data sources to external parties without any additional precautions thus leaves possibilities for disputes. Both parties, both the data producer and consumer, may be faced with malicious behavior. Data access could be denied by the consumer, falsely, or data not relevant for the collaboration could be accessed. Data consumption could be claimed by the producer, falsely, potentially leading to an unjustified claim. In either case, such disputes could lead to reputational and financial damage and should therefore be avoided. This potential trust issue leads to the first research question addressed in this dissertation:

Research Question 1: How can potential disputes, related to the sharing of data between organizations, be prevented in case of an urgent collaboration?

To enable organizations to quickly share data at all, it is required for the cluster operator to deploy software components into the different network domains. This is only possible if a participating organization contributes one or more nodes, otherwise it is impossible to run any process supporting the collaboration. Although containerized applications are isolated from local processes, assuming no bugs in the container runtime or orchestrator are present, it is not desirable to allow external entities to deploy any kind of software into your network domain. Resources could be allocated for malicious purposes, for example to contribute to distributed bot nets attacking vital network infrastructure. In such a severe case, the damage may be permanent, when network domains of the hosting organization get blacklisted. Such an infringement could for example be caused by the deployment of a container using a slightly modified container image. This does not necessarily need to happen on purpose. The origin of the container image might not be verified, or the deployment of an outdated version is suggested. In any case, it is required to prevent these kinds of trust issues related to deployment. This leads to the second research question:

Research Question 2: How can the deployment of containers, proposed by a potentially malicious external entity, be verified by the hosting organization?

When an organization refuses to deploy a container, or workload in general, there are two options: (i) the functionality enabled by the deployment of the container cannot be realized outside that specific domain or (ii) the container can be moved to any other node in the cluster. The second case allows for further steps in the scheduling procedure as a set of candidate nodes is available. The selection of a node, however, may not be straightforward for the cluster operator. The layout of the cluster and the diverging capabilities of the nodes contributed by different organizations will likely be unknown. These conditions may even vary over time. Furthermore, local administrators may decide to limit the capabilities available to external workloads, which may again be due to a lack of trust. Nodes could be tried on a trial and error basis, but this might take a considerable amount of time in case rescheduling operations are required to fix workload underperformance. Clearly, an operator has to deal with much uncertainty, and should therefore be assisted to lower the chance of scheduling mistakes. This leads to the third research question dealt with in this dissertation:

Research Question 3: How can the scheduling of containers, a task for which the cluster operator is responsible, be fitted to an unknown heterogeneous cluster environment?

Gaining insight in the cluster based on technical assessments clearly is a crucial requirement. However, it may not be sufficient to match any cross-organizational setup. Organizations may have additional, hidden, requirements which may not immediately be clear to the operator. Most likely, these non-technical requirements act as protective measures to increase the trust level of the hosting organization. Finance is an important reason to activate such preferences. Each organization may have its own vision of how the collaboration should be organized, or its own interpretation of how the collaboration went. To prevent any ambiguity, organizations should be able to express such requirements. Scheduling should therefore take into account these concerns as no suitable workload placement may be realized otherwise. Ultimately, the negotiation of requirements between organizations is desirable, as scheduling issues may be solved quickly. This gives rise to the fourth and final research question discussed in this dissertation:

Research Question 4: How can the scheduling of containers, a task for which the cluster operator is responsible and which is fitted to an unknown heterogeneous cluster environment, be extended to allow for negotiation between the participating organizations?

1.3 Research Contributions

A summary of the research conducted to solve the aforementioned research questions is presented in the paragraphs below. The parts of the dissertation which can be consulted to find more detailed answers for each research question are also mentioned. The outline of the dissertation is thus structured as follows:

1.3.1 Logging mechanism to generate irrefutable proofs

Chapter 2, corresponding with publication I listed in Section 1.4.1, and Appendix A, corresponding with publication II listed in Section 1.4.2, address the first research question. Preventing disputes requires actions to be registered and verified, a concept which is commonly referred to as traceability, meaning that each exchange of data needs to be fully captured. Three interesting topics need to be researched to allow this. Firstly, organizations need to be able to exchange data in the first place. Each of them will have exposed APIs available, but exploring these becomes a time-consuming task if an organization needs to do this manually. This clearly is an issue in case of an urgent collaboration. It should therefore be examined how API exploration can be unified. Secondly, proofs of the data exchange need to be securely stored. As these proofs, or logs, are of interest to any stakeholder in the collaboration, it is needed to set up a distributed database. The architecture of this database needs to be closely examined though, as possibly no trusted third party could be found, especially in this ad hoc scenario. Dealing with potentially malicious entities should thus definitely be supported and the concepts brought by recent blockchain solutions therefore need to be studied. Finally, it is needed to bring together the proposed data exchange and logging solutions. Their interaction is of specific importance, as in this urgent case, it is questionable to what extent the exchange of data should be interrupted by a supportive mechanism.

1.3.2 Framework to authorize container deployments

Chapter 3, corresponding with publication II listed in Section 1.4.1, addresses the second research question. Enabling an organization to stay in control over the software it hosts, requires the introduction of an intermediate step between the scheduling and deployment phase of a container. Three parts need to be researched in order for this verification phase to be constituted. Firstly, it needs to be examined how the cluster orchestration platform, in this case Kubernetes [10], should be unravelled to be able to intervene in the container deployment process. The proposed integration should ideally be activated transparently, i.e. it should be possible to turn on the intervention extension without having to alter any code in the vanilla Kubernetes implementation. Secondly, an appropriate state of art authorization protocol needs to be matched against the scenario in which a hosting organization can assess, asynchronously, whether it wants to grant access to an external organization. Identifying an open source implementation of this protocol needs to occur as well. Finally, an integration of the protocol into the orchestrator should be found based on the earlier researched manner, providing the local administrators with a dashboard overview of open container deployment requests ready for inspection and review.

1.3.3 Probe swarm to explore unknown clusters

Chapter 4, corresponding with publication III listed in Section 1.4.1, addresses the third research question. Assisting the operator in the scheduling of containers in an unknown heterogeneous cluster environment, requires supportive tooling to be integrated in the process. Again, three steps need to be researched in order for a solution to be proposed. Firstly, the steps followed by the default scheduling process in Kubernetes need to be examined. As this process should be extended with a custom phase, it is needed to identify which extension points are available. A particular aspect to investigate is whether both type of deployments, verified and unverified, could coexist. Secondly, it needs to be detailed why this default process is technically not able to meet the conditions of a heterogeneous environment as considered here. Based on this identification of missing elements, it is then possible to discuss how scheduling should advance, i.e. how further insight in the capabilities of unknown nodes can be gained. The usage of compile-once-run-anywhere tooling needs to be discussed to realize this. Finally, a proposal for integration of this tooling should be presented based on the earlier research findings. This proposal needs to present a flow of steps enabling a hybrid solution consisting of both manual decisions by the operator and automatic evaluations provided by the suggested components.

1.3.4 Rankings to express and negotiate preferences

Chapter 5, corresponding with publication IV listed in Section 1.4.1, addresses the fourth research question. Enabling negotiation between participating organizations to schedule a cross-organizational data pipeline, requires the previously discussed scheduling adaption to be further extended. The first research task is to determine which guarantees are likely to be specified by contributing organizations. Although the set of guarantees may be infinite, theoretically, it is needed to align negotiable requirements among organizations. This does not mean that this set is static by definition, the only demand is uniformity, as otherwise no structured way of finding agreement is possible. The second research task is to examine how the administrators of organizations, either manually or automatically, should be given the opportunity to assess scheduling proposals, and to communicate or discuss requirements among them. The interaction between the newly proposed and previously discussed components is of particular interest here. The presented result should be an extended flow of steps allowing for the composition and scheduling of a data pipeline based on the input of multiple stakeholders.

1.3.5 Analysis of the presented contributions

Finally, Chapter 6 concludes the research contribution. This means that the research questions are revisited and, given the content of the different parts discussed above, further analysed. This analysis entails a short summary of the main research findings, a critical view on the presented content and a perspective of future work.

1.4 Publications

The results of the research during this PhD study have been published in scientific journals and presented at different international conferences. This section provides an overview of these publications.

1.4.1 Publications in International Journals

I. **L. Van Hoye**, T. Wauters, F. De Turck, and B. Volckaert, **Trustful ad hoc cross-organizational data exchanges based on the Hyperledger Fabric framework**, *International Journal of Network Management*, vol. 30, no. 6, p. e2131, 2020, doi: 10.1002/nem.2131.

II. **L. Van Hoye**, T. Wauters, F. De Turck, and B. Volckaert, **A secure cross-organizational container deployment approach to enable ad hoc collaborations**, *International Journal of Network Management*, vol. 32, no. 4, p. e2194, 2022, doi: 10.1002/nem.2194.

III. **L. Van Hoye**, T. Wauters, F. De Turck, and B. Volckaert, **Enabling the Rescheduling of Containerized Workloads in an Ad Hoc Cross-Organizational Collaboration**, *Journal of Network and Systems Management*, vol. 31, no. 1, article no. 10, 2023, doi: 10.1007/s10922-022-09699-9.

IV. **L. Van Hoye**, T. Wauters, F. De Turck, and B. Volckaert, **Enabling organizations to participate in the ad hoc scheduling of a cross-organizational data pipeline**, *Journal of Network and Systems Management*, *Submitted for review*, 2022.

1.4.2 Publications in International Conferences

I. T. Goethals, S. Kerkhove, **L. Van Hoye**, M. Sebrechts, F. De Turck, and B. Volckaert, **FUSE: A Microservice Approach to Cross-domain Federation using Docker Containers**, *in the 9th International Conference on Cloud Computing and Services Science (CLOSER)*, 2019. doi: 10.5220/0007706000900099.

II. **L. Van Hoye**, P-J. Maenhaut, T. Wauters, B. Volckaert, and F. De Turck, **Logging mechanism for cross-organizational collaborations using Hyperledger Fabric**, *in the 1st IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2019. doi: 10.1109/BLOC.2019.8751380.

1.4.3 Code Repositories

Source code is available as well. Due to the cooperation agreement in force in the FUSE research project, the decision was made to prevent disclosure of the different repositories. However, the most important parts of the code, which is a relatively small set for the topics discussed throughout this dissertation, are presented in the different chapters. The authors can, of course, be contacted to request more information in case interest is expressed.

Bibliography

[1] T. Wauters, B. Vermeulen, W. Vandenberghe, P. Demeester, S. Taylor, L. Baron, M. Smirnov, Y. Al-Hazmi, A. Willner, M. Sawyer, D. Margery, T. Rakotoarivelo, F. Lobillo Vilela, D. Stavropoulos, C. Papagianni, F. Francois, C. Bermudo, A. Gavras, D. Davies, J. Lanza, and S.-Y. Park, "Federation of Internet experimentation facilities: architecture and implementation," in *Proceedings - European Conference on Networks and Communications (EuCNC)*, (Bologna, Italy), pp. 1–5, 2014.

[2] T. Vanhove, G. Van Seghbroeck, T. Wauters, F. De Turck, B. Vermeulen, and P. Demeester, "Tengu: An Experimentation Platform for Big Data Applications," in *Proceedings - IEEE 35th International Conference on Distributed Computing Systems (ICDCS) Workshops*, (Columbus, OH, USA), pp. 42–47, 2015. https://doi.org/10.1109/ICDCSW.2015.19.

[3] D. M. Shila, W. Shen, Y. Cheng, X. Tian, Shen, and X. Sherman, "AMCloud: Toward a Secure Autonomic Mobile Ad Hoc Cloud Computing System," *IEEE Wireless Communications*, vol. 24, no. 2, pp. 74–81, 2017. https://doi.org/10.1109/MWC.2016.1500119RP.

[4] "FUSE: Flexible federated Unified Service Environment." https://www.imec-int.com/en/what-we-offer/research-portfolio/fuse.

[5] D. Gambetta, "Can we trust trust?," in *Trust: Making and Breaking Cooperative Relations*, pp. 213–237, Blackwell, 1988.

[6] T. Goethals, S. Kerkhove, L. Van Hoye, M. Sebrechts, F. De Turck, and B. Volckaert, "FUSE: A Microservice Approach to Cross-domain Federation using Docker Containers," in *Proceedings - 9th International Conference on Cloud Computing and Services Science (CLOSER)*, (Heraklion, Greece), pp. 90–99, SciTePress, 2019. https://doi.org/10.5220/0007706000900099.

[7] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, "Zero Trust Architecture," tech. rep., National Institute of Standards and Technology (NIST), 2020. https://doi.org/10.6028/NIST.SP.800-207.

[8] B. Stanton and T. Jensen, "Trust and Artificial Intelligence," tech. rep., National Institute of Standards and Technology

(NIST), 2021. https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=931087.

[9] Y. Yang, X. Liu, and R. H. Deng, "Lightweight Break-Glass Access Control System for Healthcare Internet-of-Things," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 8, pp. 3610–3617, 2018. https://doi.org/10.1109/TII.2017.2751640.

[10] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade," *ACM Queue*, vol. 14, no. 1, p. 70–93, 2016. https://doi.org/10.1145/2898442.2898444.

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

This chapter presents a published research article tackling the first research question: ***how can potential disputes, related to the sharing of data between organizations, be prevented in case of an urgent collaboration?*** *The first part of the chapter deals with generic clients, which are required to allow distinct APIs to be consumed in a standardized way. These pave the way for a fully automated data exchange evaluation. Furthermore, logging solutions are discussed to prevent data exchange disputes. The position of Hyperledger Fabric (version 1.0-1.4) within the distributed database space is described, its architecture is analyzed, and shortcomings are discussed. Given the level of decentralization required here, a verification mechanism is presented allowing for logging mistakes, either on purpose or not, to be detected by participating organizations. Most noticeable is the focus on the asynchronous interaction between the logging process on the one hand and the data exchange process on the other, to reduce delay overhead for these urgent collaboration cases.*

⋆⋆⋆

L. Van Hoye, T. Wauters, F. De Turck, and B. Volckaert

Published in International Journal of Network Management, August 2020

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 chapter 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 chapter 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.

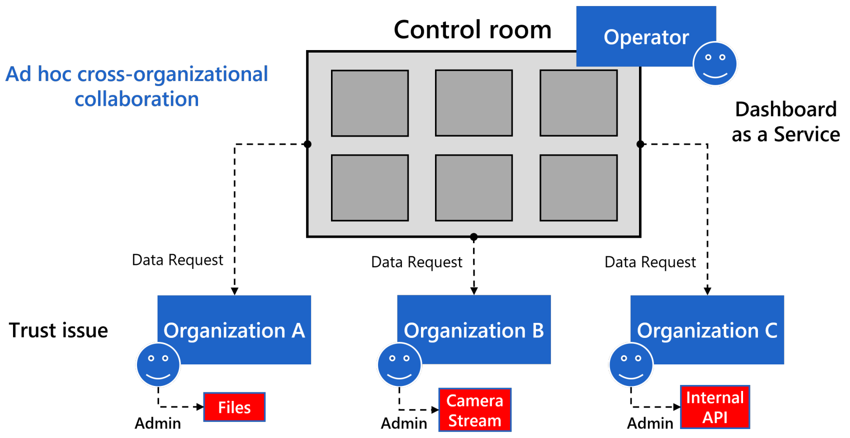


Figure 2.1: Sample ad hoc cross-organizational collaboration where multiple organizations work on a common project: a control room operator requests data from other organizations in order to offer a dashboard as a service to speed up a decision-making process.

2.1 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 chapter, a prominent one is that of an emergency situation [1] or a smart city scenario [2]. An example collaboration scenario is illustrated in Figure 2.1 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 2.1, 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 chapter, 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 chapter.

The remainder of the chapter is structured as follows. Section 2.2 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 2.3 presents the integration of generic web APIs to allow collaborating participants to explore their APIs dynamically. Section 2.4 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 2.5, after which the implemented proof of concept is explained and evaluation results for a specific setup is shown in Section 2.6. Finally, conclusions are drawn for the presented work and future research directions are described.

2.2 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 2.4.

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 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 2.2 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 chapter 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.

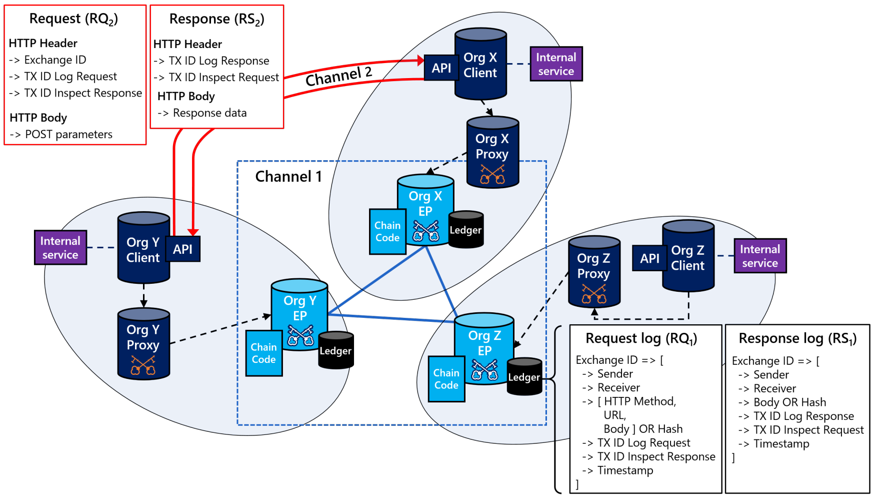


Figure 2.2: Recap of the components needed for the proposed logging mechanism. [19]

2.3 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 chapter than the exact composition of the feature set.

An illustrative setup is proposed in Figure 2.3. An Org X has three

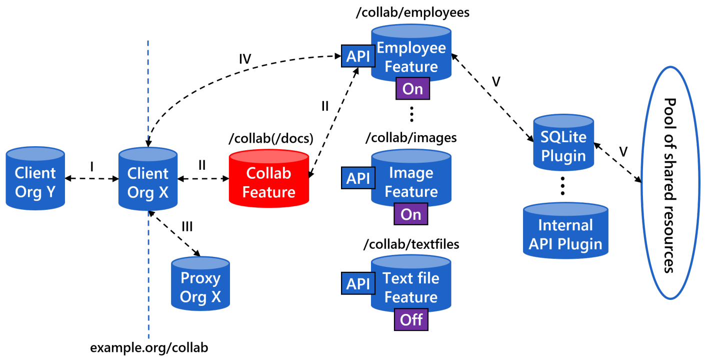


Figure 2.3: The proposed web API consisting of features which can be reused by multiple organizations.

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. An example definition of the entry point is shown in Listing 2.1. 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 2.1: 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 2.2. 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 2.2: 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.

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

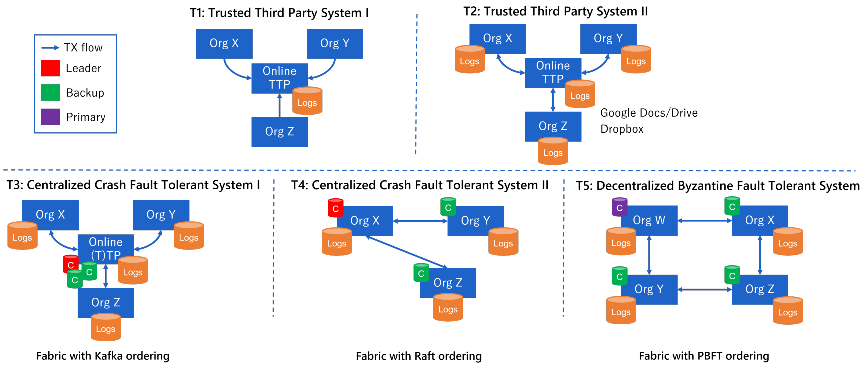

TX flow
Leader
Backup
Primary
T1: Trusted Third Party System I
Org X
Org Y
Online TTP
Logs
Org Z
T2: Trusted Third Party System II
Google Docs/Drive
Dropbox
T3: Centralized Crash Fault Tolerant System I
Online (T)TP
C
Fabric with Kafka ordering
T4: Centralized Crash Fault Tolerant System II
Fabric with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

Figure 2.4: Overview of possible topologies discussed in this chapter.

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.

2.4 Storing logs of data exchanges

As discussed in Section 2.1, 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 2.4. 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.

2.4.1 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.

2.4.2 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 2.5. 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 2.4.1, 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 2.1, 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

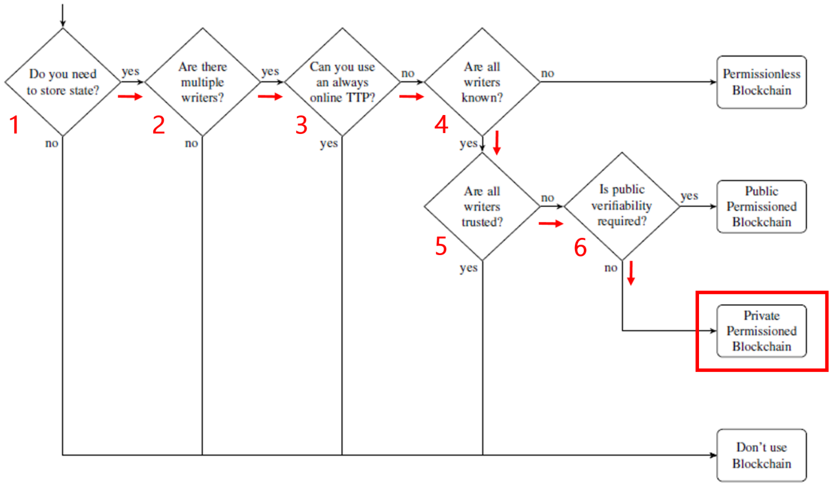

Do you need to store state?
yes
no
Are there multiple writers?
yes
no
Can you use an always online TTP?
no
yes
Are all writers known?
no
yes
Are all writers trusted?
no
yes
Is public verifiability required?
yes
no
Permissionless Blockchain
Public Permissioned Blockchain
Private Permissioned Blockchain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

Figure 2.5: Flow chart to assess whether a blockchain solution may add value to an application. [26]

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.

2.4.3 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 (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 chapter, Fabric has not officially released a BFT ordering service as is visible in the overview of Table 2.1. 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 2.4 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 2.4, 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 chapter, 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 2.1. 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 2.4.4.3, 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:

Table 2.1: 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 $\geq 3x + 1$

- 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 2.4.4.2, 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 2.4.4.3.

- 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.

2.4.4 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.

2.4.4.1 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 2.3 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.

```go
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 2.3: 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 or 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 2.4 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 2.4: Check two of block storage functionality [45] in Fabric version 1.3.

2.4.4.2 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 2.2.

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 2.5.

- 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 2.4.3, the main purpose of chaining blocks is to allow for efficient auditing techniques.

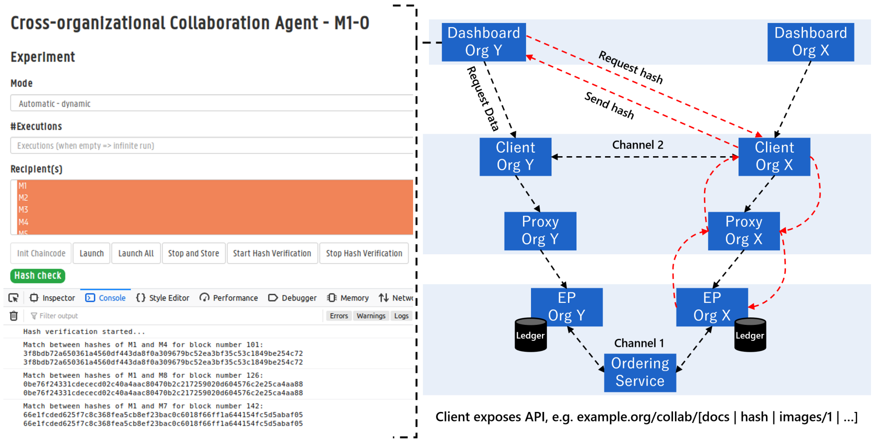
```go
func (t *Collab) inspectResponse(stub shim.ChaincodeStubInterface,
    ↪ exchangeId string, data string) pb.Response {
    // (Step I) Unpack string by Org Y to Response object and hash
    ↪ it
    response := Response{}
    err := json.Unmarshal([]byte(data), &response)
    if err != nil {
        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

    // (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 2.5: Snippet of the proposed smart contract.

2.5 Extension of the logging mechanism

The implementation of the second check outlined in Section 2.4.4.3 has not been discussed before. Figure 2.6 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



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

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 2.6. 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 2.2 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 2.7 and 2.8 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 2.6: 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.

2.6 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.

2.6.1 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 2.7 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:

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

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

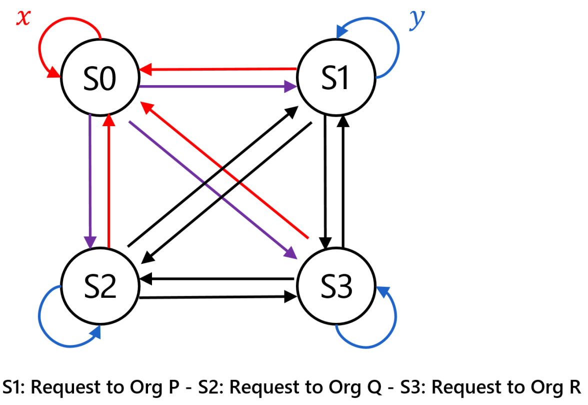

x
y
S0
S1
S2
S3
S0: Sleep Org Y - S1: Request to Org P - S2: Request to Org Q - S3: Request to Org R

Figure 2.7: Markov chain for an Org Y participating in a collaboration with $O = 4$.

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
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 2.3: 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.

2.6.2 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 2.7 and 2.8 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 2.6.1)

    // Dynamically construct a request (Section 2.6.1)

    // 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 2.7: Client code to periodically generate cross-organizational data requests.

```
// Receive requests from Org Y
app.receive('/collab/*') {
    // Protection I (Section 2.5): Check for pending hash
    ↪ verification requests with Org Y

    // Protection II (Section 2.5): 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 2.8: 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 2.6.1, 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 2.5 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 2.9 and 2.10 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 2.9: Proxy code to execute a chaincode function.

```
// 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 2.10: Proxy code to inspect the log of a counterparty.

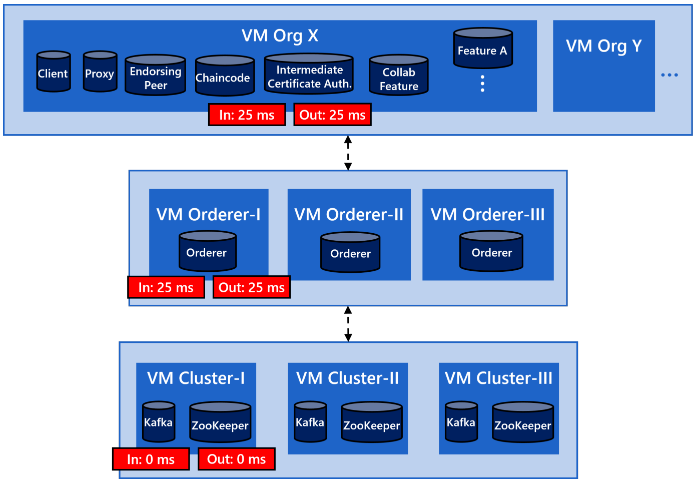


Figure 2.8: Overview of the different VMs used in the evaluation setup.

2.6.3 Evaluation results

The experiment setup, as summarized in Figure 2.8, 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.

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 2.2. Data is exchanged in a fully automated way as described in Section 2.6.1. A part of the data requests will be refused due to the protections discussed in Section 2.5, 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 2.3 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 2.2, 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

Table 2.2: 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

used when the `sendTransactionProposal` function (shown in Listing 2.9) 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 2.4 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.

2.7 Conclusions

This chapter 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

Table 2.3: Overview of both cross-organizational interactions and generated logging data when the parameter set defined in Table 2.2 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 2.4: Overview of cross-organizational interactions when the parameter set defined in Table 2.2 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

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.

Acknowledgments

The work described in this chapter, 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).

Bibliography

[1] J. Moeyersons, B. Farkiani, T. Wauters, B. Volckaert, and F. De Turck, “Towards Distributed Emergency Flow Prioritization in SDN Networks,” *Int J Netw Manag*, vol. e2127, 2020. https://doi.org/10.1002/nem.2127.

[2] J. Dos Santos, T. Wauters, B. Volckaert, and F. De Turck, “Fog computing : enabling the management and orchestration of smart city applications in 5G networks,” *Entropy*, vol. 20, no. 1, pp. 1–26, 2018. https://doi.org/10.3390/e20010004.

[3] “Hyperledger Fabric.” https://www.hyperledger.org/projects/fabric. Accessed December 1, 2019.

[4] 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 - 13th EuroSys Conference*, pp. 1–15, ACM, 2018. https://doi.org/10.1145/3190508.3190538.

[5] “we.trade | The Platform.” https://we-trade.com/the-platform. Accessed December 1, 2019.

[6] C. Schaefer and C. Edman, “Transparent Logging with Hyperledger Fabric,” in *Proceedings - 1st IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pp. 65–69, IEEE, 2019. https://doi.org/10.1109/bloc.2019.8751339.

[7] “TradeLens.” https://www.tradelens.com. Accessed December 1, 2019.

[8] R. Han, V. Gramoli, and X. Xu, “Evaluating Blockchains for IoT,” in *Proceedings - 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pp. 1–5, IEEE, 2018. https://doi.org/10.1109/NTMS.2018.8328736.

[9] M. Selimi, A. R. Kabbinale, A. Ali, L. Navarro, and A. Sathiaseelan, “Towards Blockchain-enabled Wireless Mesh Networks,” in *Proceedings - 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems (CryBlock)*, pp. 13–18, ACM, 2018. https://doi.org/10.1145/3211933.3211936.

[10] P. Thakkar, S. Nathan, and B. Viswanathan, “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)*, pp. 264–276, IEEE, 2018. https://doi.org/10.1109/MASCOTS.2018.00034.

[11] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, “How to Databasify a Blockchain: the Case of Hyperledger Fabric,” tech. rep., Saarland Informatics Campus, 2018. http://arxiv.org/abs/1810.13177.

[12] C. Gorenflo, S. Lee, L. Golab, and S. Keshav, “FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second,” in *Proceedings - 1st IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pp. 455–463, IEEE, 2019. https://doi.org/10.1109/bloc.2019.8751452.

[13] C. Gorenflo, L. Golab, and S. Keshav, “XOX Fabric: A hybrid approach to transaction execution,” tech. rep., University of Waterloo, 2019. https://arxiv.org/abs/1906.11229.

[14] K. Yamashita, Y. Nomura, E. Zhou, B. Pi, and S. Jun, “Potential Risks of Hyperledger Fabric Smart Contracts,” in *Proceedings - IEEE 2nd International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pp. 1–10, IEEE, 2019. https://doi.org/10.1109/IWBOSE.2019.8666486.

[15] N. Andola, Raghav, M. Gogoi, S. Venkatesan, and S. Verma, “Vulnerabilities on Hyperledger Fabric,” *Pervasive and Mobile Computing*, vol. 59, p. 101050, 2019. https://doi.org/10.1016/j.pmcj.2019.101050.

[16] Z. Xiao, Z. Li, Y. Liu, L. Feng, W. Zhang, T. Lertwuthikarn, and R. S. M. Goh, “EMRShare: A Cross-Organizational Medical Data Sharing and Management Framework Using Permissioned Blockchain,” in *Proceedings - The International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 998–1003, IEEE, 2018. https://doi.org/10.1109/PADSW.2018.8645049.

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

pp. 85–92, IEEE, 2017. https://doi.org/10.1109/SERA.2017.7965711.

[18] M. Muller, S. R. Garzon, M. Westerkamp, and Z. A. Lux, "HI-DALS: 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)*, pp. 802–808, IEEE, 2019. https://doi.org/10.1109/IEMCON.2019.8936305.

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

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

[21] R. Verborgh and M. Dumontier, "A Web API Ecosystem through Feature-Based Reuse," *IEEE Internet Computing*, vol. 22, no. 3, pp. 29–37, 2018. 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] M. Lanthaler, *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] M. Lanthaler, "Hydra Console." http://www.markus-lanthaler.com/hydra/console. Published March, 2014. Accessed December 1, 2019.

[26] K. Wüst and A. Gervais, "Do you need a Blockchain?," in *Proceedings - 1st Crypto Valley Conference on Blockchain Tech-*

nology (CVCBT), pp. 45 – 54, IEEE, 2018. https://doi.org/10.1109/CVCBT.2018.00011.

[27] B. Rodrigues, E. Scheid, R. Blum, T. Bocek, and B. Stiller, "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] G. Greenspan, "The Blockchain Immutability Myth." https://www.multichain.com/blog/2017/05/blockchain-immutability-myth. Published May 4, 2017. Accessed December 1, 2019.

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

[30] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J Assoc Comput Machin*, vol. 32, no. 2, pp. 374–382, 1985. https://doi.org/10.1145/3149.214121.

[31] Z. Amsden, R. Arora, S. Bano, M. Baudet, S. Blackshear, A. Bothra, and G. Cabrera, "The Libra Blockchain," tech. rep., Facebook, Calibra, 2019. https://www.semanticscholar.org/paper/The-Libra-Blockchain-Amsden-Arora/59df4bdd67ed1cde5191447bcc80fba2d70bee71.

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

[33] 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)*, pp. 51–58, IEEE, 2018. https://doi.org/10.1109/DSN.2018.00018.

[34] C. Mohan, "State of Public and Private Blockchains: Myths and Reality," in *Proceedings - ACM SIGMOD/PODS International*

Conference on Management of Data, pp. 404–411, ACM, 2019. http://doi.acm.org/10.1145/3299869.3314116.

[35] C. Mohan, “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] A. Bogomolny, “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] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” tech. rep., bitcoin.org, 2008. https://bitcoin.org/bitcoin.pdf.

[41] S. Micali, “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] G. Greenspan, “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.

3
A secure cross-organizational container deployment approach to enable ad hoc collaborations

This chapter presents a published research article tackling the second research question: ***how can the deployment of containers, proposed by a potentially malicious external entity, be verified by the hosting organization?*** *The first part of the chapter presents an analysis of what the pipeline of the container orchestrator, in this case Kubernetes, looks like at each of the worker nodes. This analysis is needed to allow organizations to intervene locally and thus to prevent any trust assumptions. Most noticeable is the integration of the authorization protocol UMA 2.0 into this Kubelet container deployment flow, resulting in a procedure consisting of three phases. The steps of each phase are detailed including the implementation of the UMA protocol in the Identity and Access Management solution Keycloak. The procedure successfully allows organizations to verify, asynchronously, potentially malicious deployments. Either manual or automatic verification is possible, a choice based on the desired level of supervision balanced against the level of deployment urgency.*

⋆⋆⋆

L. Van Hoye, T. Wauters, F. De Turck, and B. Volckaert

Published in International Journal of Network Management, December 2021

Abstract When organizations need to collaborate urgently, for example in the case of an emergency situation, it is needed to deploy software components into the different domains in order to allow crucial data to be exchanged. The ad hoc aspect is important as it does not allow the participating organizations to negotiate entire workflows and/or contracts upfront. To enable these ad hoc cross-organizational collaborations, a container orchestration platform, like Kubernetes, can be used to quickly deploy pods of containers in a cross-organizational overlay network, even fully automated. Although this is technically feasible, there may be a trust issue from the perspective of a participating organization when an external organization is capable of deploying any software inside its network domain. This concern is examined and resolved in this chapter, by proposing an extension to the existing deployment scheme used in vanilla Kubernetes. It allows the participating organizations to assess whether a suggested deployment conforms to the goal of the project and to maintain an overview of all activities related to a single collaboration. This intermediate step prevents an honest organization against potentially malicious behaviour of external entities, either the orchestrator and/or the other organizations, solving the aforementioned trust issue. Evaluation of the implemented prototype shows that a secure collaboration, which requires at most tens of containers, can be attained with sub-second deployment overheads per container, apart from the required manual interventions for trust management purposes.

3.1 Ad hoc cross-organizational collaborations

Organizations that want to share in-house data sources urgently, need to set up an ad hoc cross-organizational collaboration. The common goal of such a collaboration could be the exchange of data, e.g. to allow intervention services retrieving names of people that are badged in at a building where an emergency situation is unfolding, or the

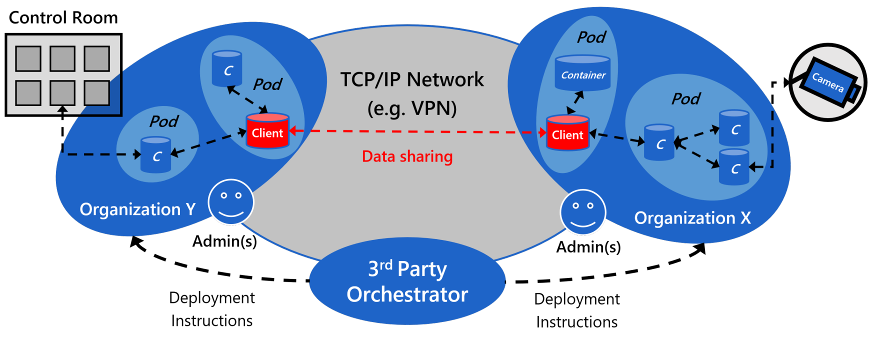


Figure 3.1: An example cross-organizational deployment scenario with the goal to share access to a private camera owned by Org X with the control room dashboard hosted by Org Y.

sharing of access to internal applications / services, e.g. a building security system. An example use case is shown in Figure 3.1: Organization X wants to share access to a private camera in order to allow Organization Y to remotely control the point of view and as such to constitute a more comprehensive operational picture. In order for this to work, software components need to be deployed at the different sites. An example of a required technical enabler is a transcoder, which is responsible for encoding the video stream before it is sent. Other examples are the client components which are needed to perform cross-organizational data sharing. As the collaboration needs to be set up ad hoc, the deployment of such components needs to be done as quickly as possible. The remainder of this chapter therefore assumes that a container orchestration platform is used, in this case Kubernetes [1], which allows containers to be deployed in a cluster of network domains which are connected via a layer-3 network.

Each cluster has one or more Kubernetes master nodes, managed by a third party orchestrator, while the participating organizations join the cluster by hosting a labeled worker node. This way, it is possible to deploy the required containers, in the form of pods, in the different domains in a minimum amount of time, without the individual organizations having to deploy all software themselves. Although this solution works, there may be a trust issue from the perspective of a participating organization, as it depends on an external entity to decide which software may be deployed inside its domain. The cluster administrator plays an important role in this matter of trust, as (s)he has the power to decide if / which authorization policies are set for the Kubernetes API. When vanilla Kubernetes is used, each organization could therefore be susceptible to the deployment of malicious configurations. As this chapter studies ad-hoc cross-organizational collaborations, the probability of misuse of resources is even higher, as deployment needs to be executed ad hoc without thorough negotiation and investigation upfront. For this specific use case, it is instead needed to integrate an intermediate step, allowing the admins of each of the participating organizations to check whether any proposed deployment configuration matches with the collaboration goal of a certain project.

A major potential issue with running external software is its ability to connect to the outside world. The resources of another organization could be used to take part in a botnet, to perform hash calculations for cryptocurrency mining, to send SPAM mails, etc. These kinds of issues could however be solved by inspecting the Kubernetes network configuration: the VXLAN backend specified by Flannel [2] is used as an enabler to create a layer-3 network for inter-host communication. This mechanism, together with the Flannel Container Networking Interface (CNI) plugin for Kubernetes [3], realizes inter-container communication. Figure 3.2 shows the (virtual) network interfaces and their interactions when this network setup is used. It also indicates which network traffic should be dropped to block communication between a container and the outside world. Note that the details shown are not necessary to understand the remainder of this chapter. This network level restriction already solves a lot of potential problems. However, it is still possible for a malicious organization to misuse the CPU/memory/disk/network resources of others, as the result of malicious computations could simply be sent back to the malicious organization. This problem can not be solved as data sharing between the organizations is the goal of a cross-organizational collaboration. Some data thus always needs to cross the perimeter of an organization in order for it to be used by others.

The research question answered in this chapter is thus to reduce this potential vulnerability. The goal is to allow a niche use case as described above to be realised in a more trustful way. The decentralization of authorization decisions in a common container orchestration platform like Kubernetes needs to be examined to enable the different organizations to decide whether they want to execute deployment configurations proposed by external entities. The remainder of this chapter is structured as follows. Section 3.2 highlights work related to

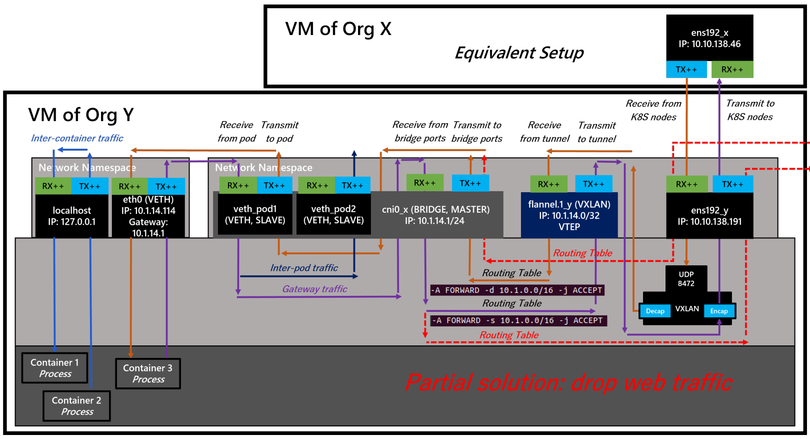


Figure 3.2: Kubernetes networking when the Flannel VXLAN backend is used for inter-host networking.

cross-organizational deployments and the protocols and frameworks used in this chapter. Section 3.3 then highlights the internals of the Kubelet component, which is a binary that runs at each worker node in a Kubernetes cluster, and discusses how additional software components could be integrated into the existing pod deployment flow (first contribution). Section 3.4 presents the proposed software components needed to solve the aforementioned trust issue and introduces the steps of the User-Managed Access (UMA) 2.0 protocol [4] [5] which are executed by these components (second contribution). Section 3.5 then discusses the evaluation of a prototype with the goal to measure the overhead introduced by the proposed deployment flow (third contribution). Finally, Section 3.6 provides a conclusion and discusses directions for future work.

3.2 Related Work

The starting point of an ad hoc cross-organizational collaboration is the realization of a cross-domain deployment environment. Goethals et al. propose a setup for microservice-based applications using a single Kubernetes cluster [6], allowing organizations to quickly setup, join and tear down a federation. The proposed environment does not consider the trust issue mentioned in Section 3.1. Using this environment, it is possible for an orchestrator to deploy any kind of application in the federated cluster with only minimal installation and configuration input needed from the participating organizations. One such application is proposed in a previous article by the authors [7], where a logging mechanism using a decentralized database setup is suggested in order to persist logs describing cross-organizational data exchanges. This mechanism allows an honest organization to protect itself against the potentially malicious behaviour of other participating organizations. Such a mechanism is needed as organizations need to collaborate ad hoc, meaning there is no time to negotiate any contract or service-level agreement upfront. The ultimate goal of the proposed logging mechanism is the same as for the solution presented in this chapter: reduce a potential trust issue between collaborating organizations. The difference is that a trust issue related to cross-organizational deployment is researched in this chapter as opposed to a trust issue related to cross-organizational data sharing. Preuveneers et al. discuss an access control solution for microservice APIs in the presence of multiple (cross-domain) stakeholders and resource owners [8]. It proposes a separate microservice enabling multi-party delegated authorization, which can be used in combination with the UMA 2.0 protocol in a fully transparent way, in order to allow a resource to be protected by different UMA 2.0 authorization servers. As the proposed access control solution is generally applicable, the paper does not narrowly focus on secure container deployment. Note that this UMA 2.0 protocol, as outlined by Schwartz and Machulak [9], is also used in this chapter as an enabler for secure cross-organizational container deployments as will become clear in Section 3.4. Finally, there are papers available in which the aforementioned trust issue is addressed, thus coming closest to our goal. Wild et al. propose a workflow for decentralized automated application deployment in a cross-organizational context [10]. The decentralization aspect of their deployment solution is crucial as it enables participating organizations to retain control over their infrastructure. Our chapter however, although it shares the same trust issue related to the provisioning of local resources in a cross-organizational context, does assume the need for a central orchestrator due to the ad hoc nature of the collaborations studied. After all, a central orchestrator will need less time to coordinate and find agreement between participating organizations. A kind of hybrid solution is thus examined here allowing a central operator to be used to set up an ad hoc intervention, while participating organizations should not fully trust it. The SLATE architecture [11] [12] then considers a set of Kubernetes clusters and federates them through their custom central API server with the goal to simplify multi-institution scientific collaborations. The administrators of the individual clusters retain control of their cluster and are able to decide who is allowed to deploy applications from a reviewed catalog. For our use case, in which ad hoc collaborations are examined, it is more efficient to let each organization spin up a lightweight Kubelet process and join the collaboration as a worker node. In summary, to the best of our knowledge, no other papers discuss a solution which allows the secure deployment of containerized services in the context of ad hoc cross-organizational collaborations.

This chapter gives special attention to Kubernetes, which "is an open-source system for automating deployment, scaling, and management of containerized applications" [1]. This orchestrator deploys containers in so called pods. These units, each with their own unique IP address, consist of one or more containers which share an interprocess communication, network and (optionally) process ID namespace and which belong to the same pod control group [13]. An important aspect of Kubernetes is its centralized API: it allows people, internal Kubernetes components, and external automation tools to control deployments [14]. In Kubernetes, access control is possible through the Kubernetes API and through the Kubelet API. For the Kubernetes API, authentication, authorization and admission control are integrated in the API server [15]. When the server is configured accordingly, it is possible to enable attribute-based access control (ABAC), role-based access control (RBAC) or webhooks [16] in order to authorize requests. For the Kubelet API, (bearer) authentication and authorization decisions are delegated to the API server [17]. The API server thus plays a crucial role in access control decisions. These off-the-shelf authorization mechanisms are however not sufficient for our use case, as the cluster administrator is fully responsible for enabling and configuring the corresponding modules. As long as participating organizations are not given the power to decide on authorization decisions themselves, there might be a trust issue from their perspective, as addressed in Section 3.1. Finally, as already mentioned, it is more efficient for our use case to focus on a single Kubernetes cluster, stretching different domains, for the cross-organizational deployment environment. The out-of-the-box Kubernetes solution for the federation of multiple Kubernetes clusters [18], which consists of multiple guest clusters and a single host cluster, thus provides no solution either. Furthermore, like a single Kubernetes cluster, the idea behind it is that a single organization is responsible for managing these clusters, as they can be managed using a single federated control plane. It could therefore not be mapped to the use case presented here without further modifications.

3.3 Breaking down the Kubelet

To be able to identify which additional software components are required to realize the goal mentioned in Section 3.1, it is first needed to explain the operation of the Kubernetes Kubelet component. Section 3.3.1 highlights the first part of the trajectory of a pod update when it is processed by a Kubelet. This part is included to obtain a complete overview of the container deployment process, but is not crucial to understand the remainder of the chapter. Section 3.3.2 describes the pipeline between the Kubelet runtime abstractions and the specific container runtime. Crucial is the existence of the container runtime interface (CRI) which allows the container deployment process to be intervened in a transparent way. Finally, Section 3.3.3 discusses the different CRI functions by means of their different functionalities and the possible security threats they may pose to participating organizations.

3.3.1 The Kubelet loop

The core function executed by the Kubelet binary is `syncLoop`, which executes an infinite loop with the goal to converge the current state of the locally deployed pods to their desired state. This desired state, which is also referred to as `PodSpec`, reaches the Kubelet via so called configuration updates. These updates can come from multiple sources [19], but the ones retrieved from the Kubernetes master node are of main interest in this chapter. The Kubelet pulls these configuration updates from the API server in a specific manner. It uses a mechanism of listing and watching. It first pulls, from the API server, a list of pods allocated to its node, by performing an HTTP GET request for `/pods`, and filters the JSON response for `spec.nodeName = X`. It then starts watching pod updates given the `resourceVersion` of the received list. The `PodUpdate` objects from the different sources are merged into a single configuration update channel using a mux, which is passed to the `syncLoop` function. Starting from this `syncLoop` function, it is possible to explain numerous different components, like all the different managers. As these are not immediately important for the remainder of this chapter, only a summary of the Kubelet core functions is given below.

1. `syncLoopIteration`: This function executes a single iteration of the `syncLoop` function. It fetches events from five distinct Go channels, among which `configCh` and `syncCh`. The `configCh` channel is the aforementioned configuration update channel. When a new configuration update is received, the `PodUpdate` object is inspected for its operation, e.g. addition of a new pod, and the pods specified in this `PodUpdate` object are passed to the appropriate handler, like the `HandlePodAdditions` function below.

2. `HandlePodAdditions`: The specified pods are further processed in this function. The pod specification of each pod is added to the pod manager, as it is responsible for storing the desired state of all pods belonging to the node. An admission process is then started to verify whether the pod can safely be deployed, e.g. whether the node is not under disk pressure. When admitted, the `dispatchWork` function is invoked, which in turn invokes the `UpdatePod` function below.

3. `UpdatePod`: Each pod has a corresponding pod worker. This is a Goroutine which executes the `managePodLoop` function below. A channel is passed to this Goroutine, in order to inject synchronization updates. This update function first checks whether a channel belonging to the specified pod already exists. If so, the update is passed to it, initiating the asynchronous synchronization of the pod. If no channel is present yet, one is created after which it is passed to a new pod worker.

4. `managePodLoop`: This function is thus executed by each pod worker. Every time a synchronization update is received via the channel, the current state of the pod is retrieved via the pod cache, a component which is available to all pod workers. This cache is filled by the Pod Lifecycle Event Generator (PLEG), a component which periodically invokes the container runtime to retrieve the state of all containers. The retrieved current state from the cache, together with the desired state of the pod, as specified in the synchronization update, are then passed to the `syncPod` function below. After a successful synchronization update is executed, a subsequent synchronization is scheduled with a re-sync interval of one minute, causing the `HandlePodSyncs` handler to be triggered.

5. `syncPod`: This thread-safe function executes a preparatory workflow related to the synchronization of a single pod. The code

can be inspected to discover every single step. At the end of the workflow, the current state and the desired state of the pod are passed to the `SyncPod` function of the container runtime manager, which is further highlighted in the next section.

Given the `PodUpdate` objects received from the API server, one may suggest to allow the administrator of the hosting organization to filter them in an early stage, before they are passed to the configuration update channel. Although possible, it might be a time-consuming task for the administrator to find out which containers in the pod are already running, which containers cannot be admitted, etc. It is instead more efficient to allow the Kubelet to perform these checks and computations with the code it already possesses. The proposed extension will then allow the administrator to intervene after the Kubelet has computed which actions need to be performed.

3.3.2 Interaction with the container runtime

The Kubelet is designed in such a way that the container runtime is pluggable meaning that different runtimes can be used. This is realized using the structs, and the noticeable interfaces each of them implements, as shown as blue and purple components in Figure 3.3 respectively. What is created, since Kubernetes release 1.5, is a so called CRI [20]. The CRI API, more specifically the version reviewed in this chapter [21], consists of two APIs: a so called client API and a server API. The client API, of which the implementation acts as a client sending container runtime requests, provides an interface to other Kubelet components, allowing them to invoke container runtime functions. The `RuntimeServiceClient` and `ImageServiceClient` interfaces constitute this client API. The server API, of which the implementation acts as a server handling container runtime requests and sending responses, provides an interface to the gRPC [22] server, allowing it to handle requests. The `RuntimeServiceServer` and `ImageServiceServer` interfaces constitute this server API. As is clear, the server part, which is also called a shim in the case of Docker, can easily be changed without having to recompile the Kubelet code. It is only necessary to pass the endpoint of a gRPC server to the Kubelet, and this server should have a component registered which implements both server interfaces. An example of such a component is the `dockerService` struct, which provides communication with the Docker daemon in order to manage containers. A `customService` struct will be defined in Section 3.4.

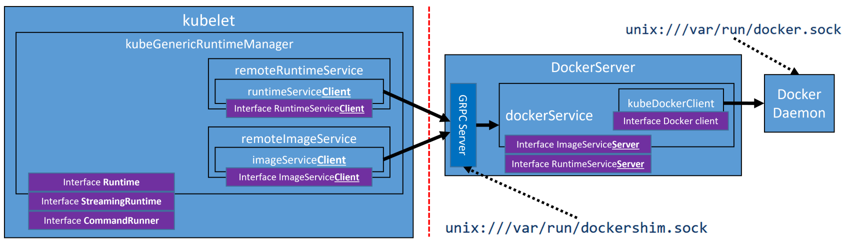

kubelet
kubeGenericRuntimeManager
remoteRuntimeService
runtimeServiceClient
Interface RuntimeServiceClient
remoteImageService
imageServiceClient
Interface ImageServiceClient
Interface Runtime
Interface StreamingRuntime
Interface CommandRunner
GRPC Server
DockerServer
dockerService
kubeDockerClient
Interface Docker client
Interface ImageServiceServer
Interface RuntimeServiceServer
Docker Daemon
unix:///var/run/docker.sock
unix:///var/run/dockershim.sock

Figure 3.3: The Kubelet components which constitute the container runtime interface.

The client API is consumed by the `kubeGenericRuntimeManager`, a struct which implements the `Runtime` interface. The `SyncPod` function, mentioned in the previous section, belongs to this interface. This function is largely based on a helper function, `computePodActions`, which compares the given desired and current state of the pod and determines whether there are actions, such as pods or containers that need to be (re)created or killed, that need to be performed. For example, when `kubectl replace` is used to change the image of a running container, this function is able to detect this change by comparing the hash values of both the old and updated container specifications. This causes the running container to be killed and the updated container to be restarted, again resulting in the invocation of the `startContainer` function implemented by the same `kubeGenericRuntimeManager`. The latter function causes the container image to be pulled, the container to be created and started, and any post start lifecycle hooks to be executed. This means that client API functions, such as `CreateContainer`, are invoked due to this function, resulting in container runtime updates to be executed. The set of CRI API functions is discussed below.

3.3.3 Opportunities to enhance control

The gRPC client-server architecture, discussed in the previous section, thus allows for an additional software component to be integrated without having to recompile the existing Kubelet code. The red dotted line in Figure 3.3 indicates this integration visually. In this chapter, such a component is introduced, which acts as a middleman between the client and the server. It has the goal to intercept and relay container runtime requests in order for them to be reviewed and rejected / accepted by the administrator(s) of an organization. This way, it is possible for an organization running the Kubelet to gain control over the software that is deployed in their domain. Before discussing the architecture of this component in the next section, it is needed to identify which CRI API functions should be intercepted and evaluated before they are executed. Note that only the functions of the `RuntimeServiceServer` interface are discussed, as the functions of the `ImageServiceServer` for image management are not considered to be potentially harmful from the perspective of the hosting organization.

- **CRI functions that pose no security threat**

    - `Version`, `Status`, `PodSandboxStatus`, `ListPodSandbox`, `ContainerStatus`, `ListContainers`, `ContainerStats`, `ListContainerStats`, `ReopenContainerLog`: These functions are all allowed without further evaluation as they only try to collect information. Their goal is not to change the state of the hosting node in any way.

    - `UpdateRuntimeConfig`: This function, which currently is only able to update the pod CIDR attached to a node, is invoked when the Kubelet is initialized. It may also be invoked by an internal Kubelet Goroutine which synchronizes node status with the master. These Kubelet internals do no expose any security risk and thus the function is allowed without further evaluation.

    - `RunPodSandbox`: This function creates and starts a pod sandbox. For the Docker container runtime, this sandbox boils down to the creation of a container with the required pod namespaces [20]. The `ContainerCreate` function of the Docker Engine API is invoked with the image `k8s.gcr.io/pause:<version>`, after which it is started with the function `ContainerStart` of the same API. The required networking configuration for the pod is, given the new network namespace, then set up as the chosen CNI plugin, such as Flannel [3] or Canal [23], gets executed eventually. This function is allowed without further evaluation, as it only prepares a sandbox environment for regular containers which will be spawned later on.

    - `StartContainer`: This function causes the container process, as specified by the `ENTRYPOINT` or `CMD` command, to run in the earlier created container environment. This function is allowed without further evaluation as the administrators of the hosting organization already verified whether the container is allowed to be deployed, including the entrypoint or command as specified by the deployment file, either explicitly or implicitly.

- ☐ `StopPodSandbox`, `RemovePodSandbox`, `StopContainer`, `RemoveContainer`: These functions are all allowed without further evaluation as stopping and removing containers, either with a forced `SIGKILL` or gracefully `SIGTERM` signal, does not necessarily harm the hosting organization. The decision could however be made, depending on the use case, that killing containers should also be protected, which is possible. Note however that an internal Kubelet garbage collector may also invoke these functions, e.g. when a pod has no corresponding regular containers. This garbage collector does not remove a container which is already created but not yet running. This is important for the intervention of the `CreateContainer` function discussed later on, as an arbitrary delay between container creation and startup will be introduced by the proposed extension.

- ☐ `UpdateContainerResources`: This function may be invoked in the CPU manager of the Kubelet depending on configuration. It enables one to match CPU cores to a container in a way that they are assigned exclusively, i.e. such that other containers are coupled to different cores, in order to prevent many CPU context switches [24]. There are two execution paths which lead to the invocation of this function. First, when the function `PreStartContainer` is executed, which happens after a container is created with the function `CreateContainer`. This is allowed as the creation of the container will already be verified by the administrator(s). Second, the function executes due to an internal Kubelet Goroutine which periodically reconciles the CPU sets as known by the CPU manager with the container runtime, which does not cause any risk. This function is thus allowed without further evaluation.

■ **CRI functions that need to be evaluated**

- ☐ `CreateContainer`: This function causes a new container environment to be prepared in the container runtime, given its full configuration (image, command, arguments, working directory, environment variables, mounting directories, etc.) and the configuration of the pod sandbox to which it belongs. Note that, based on annotations attached to the pod specification, it is possible for the middleman component to figure out which of the sources mentioned in Section 3.3.1 caused the container to be created. This is valuable information for the middleman component, as it allows it to decide whether function evaluation could be skipped, when the update source is internal like a file local to the Kubelet, or whether function evaluation is necessary, when the update source is external like the API server.

- ☐ `PortForward`: The goal of port forwarding is to allow a client of the Kubernetes cluster to connect to a containerized application running in a pod. When the Kubelet receives a port forwarding request via its Kubelet API, this function is invoked. It returns a URL `/portforward/<token>`, like `http://127.0.0.1:33793/portforward/KT874aVP`, which points to an internal port forwarding streaming server. An internal reverse proxy uses this URL to proxy this stream between the internal server and an external entity. An invocation of this function should be evaluated, as otherwise it could be possible for any organization in the cluster to get networking access to a certain pod in the cluster.

- ☐ `Exec`: The goal of the exec operation is to allow a client of the Kubernetes cluster to run a command in a container and as such to create a new process. The `stdout` and `stderr` emerging from executing the command are displayed to the client. Passing `stdin` to the newly created process is possible, and could even be upgraded to a shell when the `sh` or `bin/bash` command is used and a TTY is allocated. When the Kubelet receives an exec request via its Kubelet API, a similar streaming approach is used as for the `PortForward` function. An invocation of this function should be evaluated, as otherwise it could be possible for any organization in the cluster to deploy new processes inside other operational domains.

- ☐ `Attach`: The goal of the attach operation is to allow a client of the Kubernetes cluster to receive `stdout` and `stderr` of an existing container process, which has ID 1 in its own isolated PID namespace, and all its child processes and to send `stdin` to this process if required. Note that `stdin` can only be a TTY when one was already allocated by the container process. When the Kubelet receives such an attach request via its Kubelet API, a similar streaming approach is used as for the `PortForward` function. An invocation of this function should be evaluated, as it could be possible for any organization to manipulate the running container process using instructions sent from `stdin`.

- ☐ `ExecSync`: The goal of this function is to execute a command in a container synchronously, without the intervention of a separate streaming server as discussed above. The results of `stdout`, `stderr` and the exit code of the executed process constitute the response. This function is invoked for both (1) internal and (2) external requests: (1) when the container YAML file has `exec` defined as `postStart` and/or `preStart` container lifecycle hook, and/or when the YAML file has `exec` defined as `livenessProbe` and/or `readinessProbe` ; (2) when the Kubelet API receives a request to execute a command in a container. For the same reason as with `Exec`, this function should be evaluated, but only for the external requests. To allow our middleman component to know whether an invocation of this function is the result of an internal or external request, it is needed to inspect the container YAML file at creation time to find any of the aforementioned hooks and/or probes and to store the corresponding commands that are thus allowed according to its definition. When a command is received that does not match with one of these stored commands, it must be the result of an external request.

The analysis of the API functions above shows that five functions qualify for further verification by the administrator(s). Note that there is a difference between the `CreateContainer` function and the other four debugging functions. It is assumed that the Kubelet guarantees unique identification of each container deployment / update request using the triple (1) pod sandbox ID (2) container name and (3) attempt number. These values are included in the `CreateContainerRequest` parameter of the `CreateContainer` function. The `AttachRequest` parameter of the `Attach` function, as similar to the other

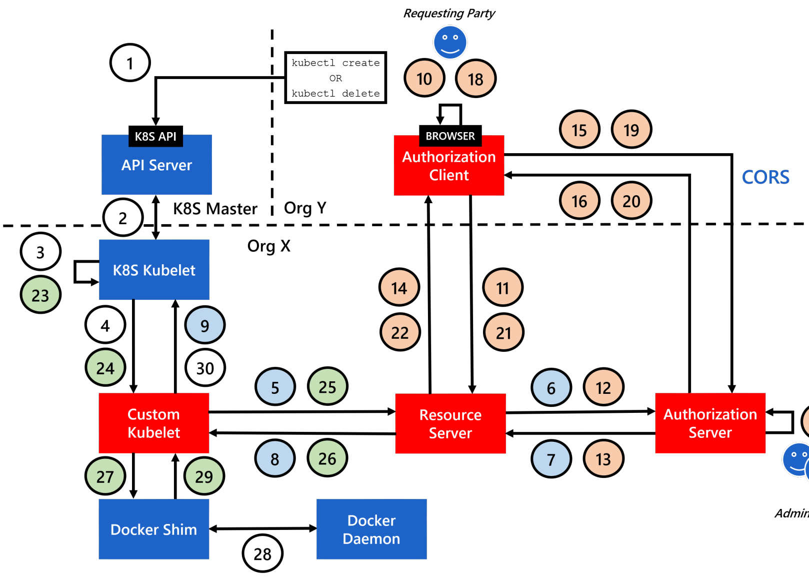


Figure 3.4: Overview of the proposed container deployment process needed to enable each organization to decide on cross-organizational container deployments. The blue components and white indexed interactions represent software present in the vanilla Kubernetes code base. The red components and blue / orange / green indexed interactions represent the proposed integration. The three colors of these newly added interactions each match a deployment phase as will be explained in Section 3.5.

three debugging functions, does not provide such a unique combination, which could lead to two equal Kubelet API invocations from two different cluster clients to be processed as being the same request. Furthermore, this could cause the debugging function response, such as a stream, to be claimed by the wrong client. An additional mechanism is needed in order to solve this problem, e.g. using a token per client session. The flow presented in the next section, which shows the details for the `CreateContainer` function, should thus be extended for the debugging functions.

3.4 Integrating the UMA 2.0 protocol into the Kubelet

What is needed is an authorization framework which allows the resource owner, in this case the administrators of a hosting organiza-

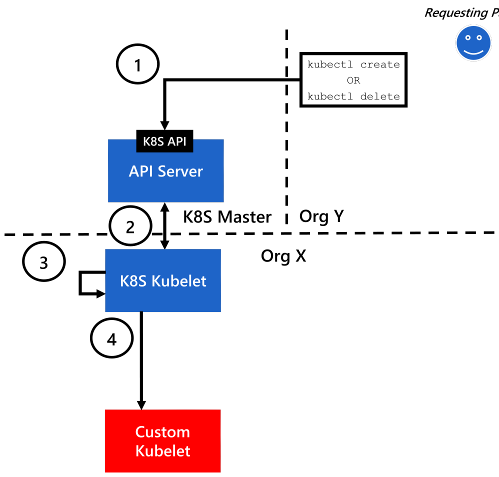


Figure 3.5: A subset of the steps extracted from Figure 3.4.

tion, to grant access to its internal infrastructure in order to allow containers to be created on request of an external entity. The UMA 2.0 protocol, which is an extension to the OAuth 2.0 authorization framework [25], provides authorization functionalities that can be used to enable this, as it allows a resource owner to authorize data requests from requesting parties in an asynchronous manner. For this use case, it is sufficient to define an analogous scenario based on the standard flow, suited to deploy a pod with one or more containers. The result of this analysis is displayed in Figure 3.4. The remainder of this section will focus on the implementation of the different steps. Parts of Figure 3.4 are replicated in Figures 3.5-3.9 for convenience. Note that, although the example given is ented on the Docker container runtime, it is possible to integrate this extension for every runtime that supports the CRI. This means that it is equally possible to support Open Container Initiative (OCI) compliant runtimes, such as `runc` and `Kata Containers`, through the CRI-O runtime [13].

The steps 1-4, highlighted in Figure 3.5, are discussed in more detail below.

1. The requesting party, being one of the participating organizations or the orchestrator, uses the Kubernetes API server to communicate deployment instructions. For example, a command like `kubectl create` or `kubectl delete` is executed together with a YAML deployment file, either automatically via a script

or manually, to propose the deployment of a pod.

2. The pod scheduling logic present in the Kubernetes master node reads the deployment instruction and based on its content, e.g. whether a `nodeSelector` is present or whether memory or CPU requests are specified, it assigns the pod to an appropriate node. The pod specification is then pulled by the corresponding Kubelet.

3. The Kubelet loop processes the pod configuration update as described in Section 3.3.1.

4. This step marks the start of the `startContainer` function mentioned in Section 3.3.2. This function causes the CRI API function `CreateContainer` to be invoked eventually. In order to introduce the middleman component suggested in Section 3.3.3, which is referred to as the custom Kubelet from now on, it is needed to alter the existing gRPC setup. First, the gRPC server code used by the Docker shim is duplicated for the custom Kubelet and the endpoint `unix:///var/run/custom-server.sock` is passed as a parameter to the Kubelet binary. This creates a gRPC connection between the Kubelet and the custom Kubelet. Second, the gRPC client code used by the Kubelet is duplicated for the custom Kubelet and the endpoint `unix:///var/run/dockershim.sock` is passed as a parameter to it, such that a similar gRPC connection is created between the custom Kubelet and the Docker shim. It is thus possible to intercept container runtime requests only by duplicating existing Kubelet code and setting endpoint parameters. Additional steps need to be integrated once container requests are intercepted by the gRPC server of the custom Kubelet. To enable this, a `customService` struct is foreseen which implements custom request handlers for the five CRI API functions discussed earlier.

The steps 5-9, highlighted in Figure 3.6, are discussed in more detail below.

5. Inside the custom handler of the `CreateContainer` function, an HTTP POST request to the endpoint `http://<resource-server-ip>:<resource-server-port>/containers` is made, a path which is part of the REST API exposed by the resource server. This resource server is a confidential client of the authorization server as it is able to securely keep a shared secret which it needs

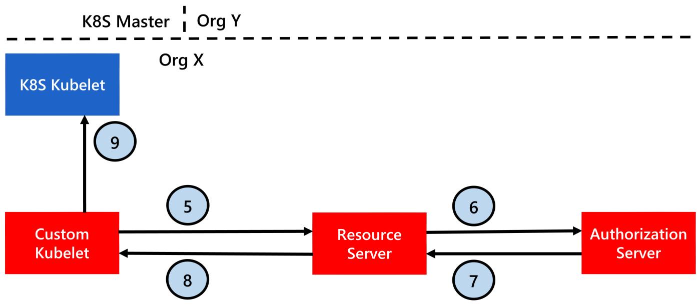


Figure 3.6: A subset of the steps extracted from Figure 3.4.

for further communication. The authorization server used in this research is provided by the open source identity and access management solution Keycloak [26]. Keycloak Authorization Services [27] is enabled for the resource server, meaning that the authorization server can be used to enforce policies associated with the paths exposed by the resource server in order to protect them from unauthorized access. The resource server implementation consists of a Spring Boot application, which is secured by Spring Security, and for which Keycloak provides an out-of-the-box adapter which hooks into the existing Spring Security authentication filter chain. This adapter adds four additional filters as defined in the `KeycloakWebSecurityConfigurerAdapter` class, of which two of them are most important: a `KeycloakAuthenticationProcessingFilter` to handle multiple authentication strategies, like bearer token authentication and OAuth authentication, and a `KeycloakAuthenticatedActionsFilter` to intercept authenticated requests and check for authorization. Note that the Keycloak adapter can be configured to fit for different use cases. The configuration used for this research is displayed in Listing 3.1. The bearer-only setting indicates that the resource server should act as a REST service rather than as a web application. This means that it should only verify bearer tokens sent along by public clients and not try to login users of these clients by redirecting their user agent according to the OAuth 2.0 Authorization Code Grant [25].

```
{
  "realm": "secure-container-deployment",
  "auth-server-url": "http://<auth-server-ip>:<auth-server-
    ↪ port>/auth/",
  "resource": "resource-server",
  "bearer-only" : true,
  "credentials": {
    "secret": "XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX"
  },
  "policy-enforcer": {
    "user-managed-access": {},
    "lazy-load-paths": true,
    "paths": [
      {
        "path" : "/containers",
        "enforcement-mode": "DISABLED"
      },
      {
        "path" : "/containers/*",
        "methods" : [
          {
            "method": "POST",
            "scopes" : ["CREATE"]
          },
          {
            "method": "DELETE",
            "scopes" : ["DELETE"]
          }
        ]
      }
    ]
  }
}
```

Listing 3.1: Configuration of the Keycloak adapter

The resource server API has both an internal and external part. It is consumed by a component inside its own domain, the custom Kubelet, and by a component outside its domain, the authorization client respectively. Authentication and authorization for both parts are different. The following policy is chosen for the internal part: it is only allowed to invoke `/containers` when a bearer access token, which is a JSON Web Token (JWT) [28], is sent with the HTTP authorization header of the request which contains the role `ROLE_ADMIN`. This RBAC policy is hard-coded in the resource server code in the following way:

```
http.authorizeRequests().antMatchers("/containers").
    hasRole("ADMIN")
```

Listing 3.2: Hard-coded role-based policy for internal API requests

The last filter of the Spring Security authentication filter chain, the `FilterSecurityInterceptor`, will then check whether the authenticated security context has the required role. As this policy is hard-coded, it is not needed to execute the steps of the Keycloak authorization filter. More specifically, each path of the internal API needs to have a disabled enforcement mode, as shown in Listing 3.1, in order to skip the creation of a Keycloak authorization challenge, as will be the case for the external part of the API, discussed from step 11.

The custom Kubelet thus needs to obtain an access token (JWT) from the authorization server with the admin role. Therefore, a second confidential client is registered at the authorization server, of which the secret is shared with the custom Kubelet. As a service account is associated with this confidential client, and the `ROLE_ADMIN` role is assigned to it, the custom Kubelet is able to obtain an access token conform the OAuth 2.0 Client Credentials Grant [25]. Access to POST `/containers` path is now granted.

6. The resource server handler associated with the POST `/containers` path is then executed. It extracts the received container specifications, adds it to a `ResourceRepresentation` object, and sends it to the resource registration endpoint of the authorization server. Note that a protection API token (PAT) is needed in order to register the resource [5]. This PAT, which is an access token with the scope `uma_protection`, can be obtained using the service account associated with the confidential client of the resource server, again conform the OAuth 2.0 Client Credentials Grant [25]. The following specifications are set:

   - ☐ The path is set to `/containers/<container-hash>`. The container hash placeholder is equal to the SHA-256 hash of the triple pod sandbox ID, container name, and the attempt number. This triple is chosen as it is assumed to be unique per Kubelet specification.

   - ☐ The resource owner is set to be the `admin` user. Furthermore, owner-managed access is turned on, allowing the administrator to evaluate permission requests.

- ☐ Several container specifications such as the required image and the command to be executed are set as resource attributes. These specifications are extracted from the `CreateContainerRequest` parameter by the custom Kubelet. A lot of other attributes could be added here. These attributes allow the administrator to make a decision on container deployment.

- ☐ Scopes are also associated with each resource. Both the CREATE and DELETE scope are added, such that killing a container could easily be evaluated too using the same resource, in case it would be needed at a later stage.

7. Once resource registration is finished successfully, the resource server handler completes the request with the storage of the container hash together with a `WAITING` tag in a thread-safe hash map.

8. A 201 HTTP response is sent back to the custom Kubelet. The body of this response contains the URL of the resource server `http://<resource-server-ip>:<resource-server-port>/containers`.

9. When the custom Kubelet receives a 201 HTTP response, it creates a message indicating that the container deployment request is started, together with the received resource server URL, and sends this message back to the Kubelet as an error. A container creation error is returned purposely, as it causes, via the aforementioned `syncCh` channel of the Kubelet loop, a new iteration of the `SyncPod` function to be executed after at least `backOffPeriod` seconds. This way, synchronization of a pod is rescheduled automatically, trying to deploy all containers for which no authorization was granted yet. This thus means that the steps 3-5 and 8-9 are repeated as long as the Kubelet determines an error for a container deployment. The steps 6 and 7 are skipped as long as the `WAITING` tag is associated with the corresponding container hash. Further progress is described from step 23 onwards.

The steps 10-16, highlighted in Figure 3.7, are discussed in more detail below.

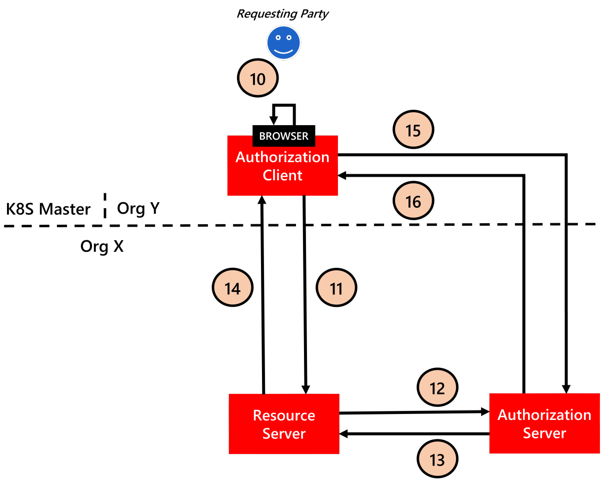


Figure 3.7: A subset of the steps extracted from Figure 3.4.

10. The requesting party, the entity which triggered the pod deployment, then tries to set up an HTTP connection using its browser with the resource server of the hosting organization. There are two scenarios for the requesting party to know which resource server URL to use. First, the pod deployment was targeted, meaning that the pod specification file contained a `nodeSelector` element. The requesting party will thus, most likely, know the URL of the resource server of the targeted organization. Second, the pod deployment was not targeted, meaning that the Kubernetes scheduler decided which organization to use for deployment. The requesting party is then able to extract the required URL of the resource server, either manually or automatically, by observing the pod state via the Kubernetes API and interpreting the `CreateContainerError` message reported by the associated Kubelet.

    The browser of the requesting party then downloads the required authorization client code from the resource server. The authorization client is a public client of the authorization server which acts on behalf of the requesting party. A login page is shown to the requesting party in order to supply its user credentials and to obtain an access token according to the OAuth 2.0 Authorization Code Grant [25]. When logged in, the requesting party is presented with a dashboard showing an overview of all pods at the hosting organization which are waiting to be claimed. This provides the requesting party the opportunity to claim its pod deployment request and, if applicable, accept associated conditions such as estimated operational costs. The manual intervention at this stage thus allows a requesting party to assess the proposed deployment scenario and give a final decision. This intervention is especially useful for this research as we are dealing with ad hoc cross-organizational collaborations, for which it is nearly impossible to exchange deployment constraints and/or preferences upfront.

11. When the requesting party claims a pod deployment, a button is clicked, causing an HTTP GET request to be sent to `http://<resource-server-ip>:<resource-server-port>/containers/<container-hash>`. This request is unauthenticated, meaning that authentication fails in the `KeycloakAuthenticationProcessingFilter`. As policy enforcement is enabled for the paths `/containers/*`, as is visible in Listing 3.1, the Keycloak adapter does not immediately send an access denied response. The local path configuration defines which scopes should be granted to allow a certain HTTP method. This local configuration is further extended by the adapter in the following way: as the path contains an asterisk and lazy loading of paths is enabled, the adapter causes the authorization server to be queried at `/auth/realms/secure-container-deployment/authz/protection/resource_set` in order to find a path configuration which matches exactly with the target URL. Note that a PAT is also required for this request. This configuration is always found, as it is created in step 6. The local and received path configurations are merged, more specifically the local HTTP-method-to-scope mapping is added to the received configuration. The Keycloak adapter then follows the same procedure as when an authenticated request would be processed, but without the required permissions: the UMA response is prepared.

12. In order to prepare this response, it is first needed to obtain a permission ticket from the authorization server, summarizing to which resources and scopes access is requested. To do so, the resource server first inspects its path configurations, as mentioned in previous step, to determine which resource and scope(s) are required, and constructs an HTTP POST request to the permission endpoint of the authorization server at `/auth/realms/secure`

`-container-deployment/authz/protection/permission`. Note that a PAT is also required for this request.

13. The received permission ticket is a JWT. Its internal structure is defined by the authorization server and thus completely opaque to other entities.

14. A 401 HTTP response is returned to the authorization client. The permission ticket as well as the URL of the authorization server are included in the `WWW-authenticate` response header.

15. The authorization client uses the received permission ticket to construct an authorization request, which is sent to the token endpoint of the authorization server at `/auth/realms/secure-container-deployment/protocol/openid-connect/token`. The goal of the authorization client is to request a Requesting Party Token (RPT), which is a grant of type `urn:ietf:params:oauth:grant-type:uma-ticket`. This is an access token with a separate `permissions` section stating the resource IDs and scopes to which access is granted. Note that the access token obtained in step 10 is added to the HTTP authorization header of the authorization request in order to authenticate the authorization client and the requesting party on which behalf the client is requesting access.

16. The response to the authorization request is a 403 HTTP access denied response together with the message `request_submitted`. This message indicates that a manual intervention of the resource owner, in this case the administrator(s), is needed in order to evaluate the permission request. The reason for an intervention is because no permissions are pre-set by the resource owner, as it is not able to know which containers will ever be deployed in the future. Note that a discrepancy exists between the authorization server as implemented by Keycloak and the formal UMA protocol. More specifically, the protocol requires permission tickets to be single-use, i.e. once they are sent to the token endpoint, they should be invalidated, and a new permission ticket should be returned to the authorization client in order to check for access at a later point in time. Keycloak does not return a new permission ticket and allows the same permission ticket to be reused until it is not active anymore, i.e. once its expiration time is passed.

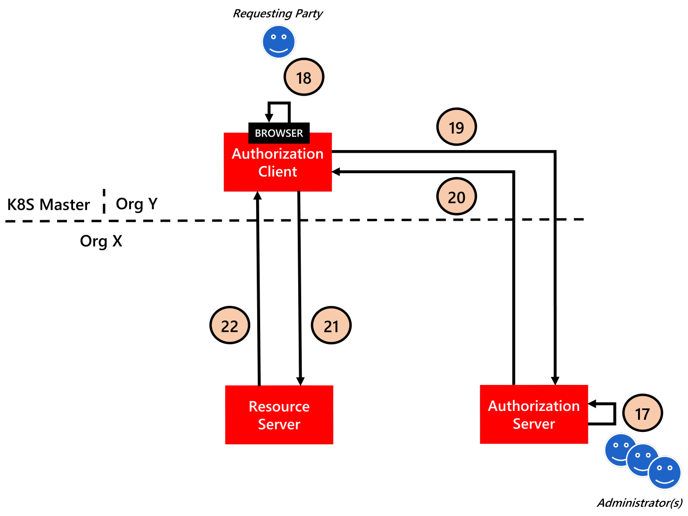


Figure 3.8: A subset of the steps extracted from Figure 3.4.

The steps 17-22, highlighted in Figure 3.8, are discussed in more detail below.

17. The administrator then uses its browser to navigate to `/auth/realms/secure-container-deployment/account` in order to view its Keycloak account web page. Again, the steps of the OAuth 2.0 Authorization Code Grant [25] are followed for login. The administrator uses its user agent to interact with the Keycloak account web application, configured as a confidential client, in order to allow this application to retrieve an access token with the user-role mappings `manage-account` and `view-profile`. The account page provides a dashboard displaying an overview of all resources owned by the administrator. For this scenario, this boils down to all resources associated with the paths `/containers/*`. The open permission request from the requesting party is visible here. When the administrator agrees with the container deployment request after studying the corresponding attributes as highlighted in step 6, the permission grant is stored. The container deployment request is, from now on, associated with the requesting party, meaning that, once a collaboration is finished, the costs of deployment can easily be audited and invoiced, if necessary. Note that multiple administrators can be considered, according to the extension proposed in literature as explained

in Section 3.2, and e.g. a majority vote could be required in order to grant permission.

18. The requesting party waits some time before it attempts the next authorization request via the same dashboard as mentioned in step 10. As it is unknown for the requesting party how long it will take for the administrator(s) to grant access, it could be that multiple authorization attempts are necessary. This step highlights the asynchronous characteristic of the UMA flow, i.e. authorization is not immediately granted or denied after successful authentication.

19. This step involves the same authorization request as sent in step 15. Note that the same permission ticket is used, due to the reason mentioned in step 16.

20. When the administrator(s) has/have granted permission, the RPT is finally obtained.

21. The authorization client now has the RPT it needs. It executes the same HTTP GET request to `http://<resource-server-ip>:<resource-server-port>/containers/<container-hash>` as described in step 11, but now with the RPT included in the HTTP authorization header. As the required resource ID and scope are included in the token, it is now allowed to execute the corresponding handler. This handler changes the `WAITING` tag, as described in step 7, to the `GRANTED` tag.

22. A 200 HTTP response is sent back to the authorization client. The requesting party now knows that its container deployment request will be executed soon.

The steps 23-30, highlighted in Figure 3.9, are discussed in more detail below.

23. Step 9 highlights how a new iteration of the `SyncPod` function is scheduled due to the container creation error. This step represents the overhead caused by the last iteration.

24. This step is analogous to step 4.

25. This step is analogous to step 5.

26. The resource server again checks which tag is associated with the specific container hash. At this stage, this will be the `GRANTED` tag, and a 200 HTTP response is sent back to the custom

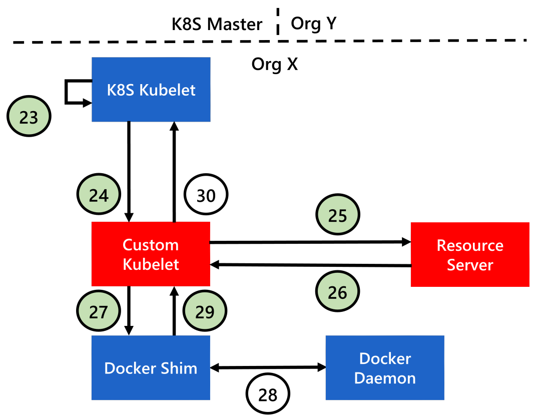


Figure 3.9: A subset of the steps extracted from Figure 3.4.

Kubelet.

27. When the 200 HTTP response is retrieved, it is finally allowed for the custom Kubelet to instruct the Docker shim to create the container by invoking the original `CreateContainer` function. As explained in step 4, the custom Kubelet uses the same gRPC client code as used by the Kubelet, but now to communicate with the gRPC server of the Docker shim.

28. The Docker shim uses the Go client for the Docker Engine API to communicate with the Docker daemon. The interaction with the Go client is already provided by Kubernetes, so no additional step is integrated here.

29. The Docker shim returns a `CreateContainerResponse` struct back to the custom Kubelet.

30. The custom Kubelet forwards this struct to the Kubelet, finishing the container deployment flow.

It is important to note that the proposed architecture is constructed with two fundamental design choices in mind. First, it should not be required to alter existing Kubelet code to allow the extension to work. The proposed extension is fully transparent from the perspective of the Kubelet, allowing future Kubelet releases and container runtimes supporting the CRI API to be easily extended as well. Second, the proposed architecture should not only handle the static scenario in which the requesting party couples a container to a specific node via a so called `nodeSelector` field in the YAML file, but also the dynamic scenario in which the Kubernetes scheduler decides on container placement. Both design choices are satisfied when the proposed extension is used.

3.5 Evaluation

The main purpose of the evaluation section is to show that a proof-of-concept works and to characterize the overhead introduced by the proposed extension presented in Section 3.4. What is clear for the case discussed here is that we focus on ad hoc cross-organizational collaborations, involving typically less than ten organizations. The reasoning is that collaborations we envision are of the urgent kind, e.g. emergency scenarios, and as such a much larger number of organizations would not make sense as they would not offer the kind of agility we have in mind when urgent collaborations are needed. This means that only a handful of pods will be placed in a cross-organizational setting. Otherwise, it would not be realistic for an administrator to validate container deployments manually. In this context, we believe it is more valuable to evaluate a single iteration of the proposed flow rather than its performance in a large-cluster setup.

3.5.1 Setup

It is possible to separate the additional steps of the proposed container deployment process into three phases. These phases are also shown in Figure 3.4 by means of the blue / orange / green indexed interactions. The white indexed interactions are attached to steps which should not be taken into account when evaluating the additional overhead caused by the proposed integration. The reason is straightforward: these steps are executed by vanilla Kubernetes. Note that Kubernetes version 1.19.15 is considered for the experiment, more specifically, the blue components in Figure 3.4 belong to this version.

1. During the first phase, which is also called the preparation phase and which comprises the blue indexed interactions 5-9, the pod specification is processed by the custom Kubelet and the representations of the containers which need to be started are set up in the authorization server. Both the authorization

server, Keycloak version 8.0.1, and the resource server, Spring Boot version 2.2.2 with Spring Security version 5.2.1 and with the corresponding Keycloak adapter [29], are containerized applications which are running on a single virtual machine (VM) with following specifications: Ubuntu 18.04 LTS equipped with four vCPUs of an Intel Xeon E5645 2.4 GHz processor and 4 GiB of RAM. The custom Kubelet, being a Go binary, is also started on this machine. An artificial network latency is set for the packet scheduler of this VM, being applied to both the ingress and egress traffic. As the average ping round-trip time (RTT) to Amazon servers in Western Europe varies around 30 ms [30], this latency is set to 15 ms. All interactions belonging to this phase are performed between components in the same network domain, meaning that these will not be impacted by this additional latency. The measurement starts when the `CreateContainer` function of the custom Kubelet is invoked and ends when the container creation error is received by the Kubelet.

2. During the second phase, which is also called the approval phase and which comprises the orange indexed interactions 10-22, the requesting party claims certain container deployments and awaits approval for these from the administrator of the hosting organization. The cross-organizational interactions considered in this phase are initiated in the browser of the authorization client. The JavaScript code written for this prototype makes use of the Keycloak JavaScript libraries `keycloak.js` and `keycloak-authz.js`, both also of version 8.0.1. As the browser of the authorization client could be running anywhere in the world, time measurements related to the different API invocations could vary per location and over time. Instead of recording measurements in the browser, the command line tool `curl` will be executed by a separate VM, representing Org Y, in the same cluster as the VM of Org X, in order to obtain more stable measurements.

   As this phase consists of at least four cross-domain interactions between the authorization client and the components of Org X, it is heavily influenced by network latency. Assuming the one-way latency between the domains of Org X and Y is in the order of tens of milliseconds, the total latency of these interactions will take at least in the order of hundreds of milliseconds per container. The exact latencies of these cross-domain interactions are however negligible when two other interactions are taken into account. Both authorization client and authorization server introduce a manual intervention in interaction 10, 18 and 17 respectively. Such a manual intervention means that decisions are taken sequentially, as the different containers are analyzed one by one, both for requesting a deployment and for deciding on deployment. As a human is part of this sequential process, the overhead for these steps will be in the order of tens of seconds at least. To be able to measure the duration of the second phase, these manual interventions are skipped. The evaluation of this phase thus boils down to the time taken by steps 11-16 and 19-22, which are labeled as phase II.A and II.B respectively.

3. During the third phase, which is also called the execution phase and which comprises the green indexed interactions 23-27 and 29, the custom Kubelet finally gets permission to perform the actual container deployment. As already explained in step 9, the duration of step 23 depends on the `backOffPeriod` parameter, which is set to be ten seconds. The actual overhead introduced by this step will depend on when permission for container deployment is granted. In the worst case, the overhead is in the order of seconds: when permission is granted immediately after a container creation attempt failed. The evaluation of this phase thus boils down to the time taken by steps 24-27 and 29. The measurement starts when the `startContainer` function is invoked in the Kubelet and ends when a `CreateContainerResponse` is received by the custom Kubelet. The time the Docker shim needs to create the container is subtracted from this duration as this is a vanilla Kubernetes step.

3.5.2 Overhead in time

The analysis in the previous section shows that the manual interventions of both the requesting party and the administrator of the hosting organization are dominating factors when overhead in time is studied. It is however possible to measure the overhead in time of the other steps, which are fully automated, and have a look at a best case scenario: a scenario for which it is assumed that the manual interventions occur instantly, that the last `SyncPod` iteration is executed right after permission is granted, that only a single container per pod needs to be deployed, and that Org X is known upfront. This experiment, which deploys an Alpine-based Node.js container, is repeated ten times and the results are presented in Table 3.1. The first run is a dry run, i.e. it is executed in order to allow things to be initialized, for example: during the first phase, the public key used by the authorization server to sign tokens (JWTs) is retrieved by the resource server to be able to verify signatures. Note that both the requesting party and administrator are already logged in and that their single sign-on session and corresponding tokens remain valid throughout the experiment.

The measurements show that a sub-second overhead should be expected when the proposed container deployment process is used. The overhead introduced by both the first and third phase is small as it only counts for around ten percent of the total measured overhead. The second phase, which is split into two sub-phases each consisting of two API interactions, clearly suffers from the aforementioned artificial network latency. The time measurements presented for this phase include twice the RTT latency of around 30 ms: once to perform the TCP handshake and once to perform the API invocation. This thus means that around eighty percent of the total duration of this phase is due to this network latency, and thus that the overhead to prepare the different API responses is rather small. As already mentioned, the presented overheads are based on the aforementioned assumptions and a higher latency should be expected for practical use cases due to the integrated manual interventions.

The total time overhead for a pod, consisting of multiple containers, is not necessarily equal to the sum of the total overheads of the individual containers. The reason for this is that the flow of each container deployment process is completely independent of others (except for small steps, e.g. due to concurrent hash maps). This allows (1) the authorization client, the resource server and the authorization server to work on them in parallel on different machines, and (2) each of these components to exhibit concurrent behaviour due to use of multiple threads or a single thread with asynchronous event handling. However, it is important to note that this level of concurrency is only interesting to consider for an automated evaluation scenario, as in practice it will almost be nullified by the dominant, time consuming manual interventions which exhibit a sequential character. Increasing the number of independent administrators validating container deployment suggestions is a possible solution to (partially) break this

Table 3.1: Overhead in time introduced by the different phases of the proposed container deployment process.

	Phase I (ms)	**Phase II.A (ms)**		**Phase II.B (ms)**		**Phase III (ms)**	
	Steps 5-9	*Steps 11-14*	*Steps 15-16*	*Steps 19-20*	*Steps 21-22*	*Steps 24-27 & 29*	
Run 0	571	89	80	78	82	11	Dry run
1	25	79	76	79	78	16	
2	23	82	80	76	75	13	
3	28	80	75	79	76	21	
4	25	80	74	78	74	11	
5	24	79	75	81	83	12	
6	22	77	77	79	72	12	
7	25	82	76	78	75	14	
8	19	81	74	77	74	10	
9	24	79	75	78	77	15	
10	20	77	77	78	73	13	
Median	24	80	76	78	75	13	
							Total: 346 ms

sequentiality.

3.6 Conclusion

This chapter presents a software framework allowing for cross-organizational container deployments to be executed in a secure manner. It integrates the UMA 2.0 protocol into the existing Kubernetes workflow to create a way for hosting organizations to validate container deployments suggested by other organizations in the same Kubernetes cluster. This framework can be used for ad hoc cross-organizational collaborations in which deployments should be realized quickly, but also securely, assuming no time is available to negotiate contracts upfront. Evaluation of a prototype shows that a sub-second overhead should be expected for an individual container deployment, but that, due to manual interventions of which the exact duration is unknown upfront, a higher impact should be expected. A trade-off between security and overhead in time should be made in order to decide whether the proposed extension is appropriate for the use case at hand. When organizations mutually trust each other, there is no reason to deploy the proposed extension. When organizations do not fully trust each other, this extension provides the perfect solution to allow containers to be deployed securely, assuming no huge amounts of containers need to be deployed. Future work should investigate how the administrator(s) of the hosting organizations could be supported in their decision-making process in order to speed it up. Part of the validation could for example be automated using pre-defined policies. The policy engine OPA (Open Policy Agent [31]) is a perfect enabler for this. Container image scanning tools could be integrated too. This way, a combination of manual and automated decisions could be realised, leading to a more secure cross-organizational collaboration.

Acknowledgments

The work described in this chapter, was partly funded by the FUSE research project [32], in which a Flexible federated Unified Service Environment was investigated. The project was realized in collaboration with imec. Project partners were Barco, Axians and e-BO Enterprises, with project support from VLAIO (Flanders Innovation & Entrepreneurship).

Bibliography

[1] "Kubernetes." https://kubernetes.io.

[2] "Flannel." https://github.com/coreos/flannel. Accessed March 31, 2021.

[3] "CNI - Flannel plugin." https://github.com/flannel-io/cni-plugin. Accessed March 31, 2021.

[4] M. Machulak, J. Richer, and E. Maler, "User-Managed Access (UMA) 2.0 Grant for OAuth 2.0 Authorization." https://docs.kantarainitiative.org/uma/wg/rec-oauth-uma-grant-2.0.html. Published January 7, 2018. Accessed March 31, 2021.

[5] M. Machulak, J. Richer, and E. Maler, "Federated Authorization for User-Managed Access (UMA) 2.0." https://docs.kantarainitiative.org/uma/wg/rec-oauth-uma-federated-authz-2.0.html. Published January 7, 2018. Accessed March 31, 2021.

[6] T. Goethals, S. Kerkhove, L. Van Hoye, M. Sebrechts, F. De Turck, and B. Volckaert, "FUSE: A Microservice Approach to Cross-domain Federation using Docker Containers," in *Proceedings - 9th International Conference on Cloud Computing and Services Science (CLOSER)*, (Heraklion, Greece), pp. 90–99, SciTePress, 2019. https://doi.org/10.5220/0007706000900099.

[7] L. Van Hoye, T. Wauters, F. De Turck, and B. Volckaert, "Trustful ad hoc cross-organizational data exchanges based on the Hyperledger Fabric framework," *Int J Network Mgmt*, vol. 30, no. 6, p. e2131, 2020. https://doi.org/10.1002/nem.2131.

[8] D. Preuveneers and W. Joosen, "Towards Multi-party Policy-based Access Control in Federations of Cloud and Edge Microservices," in *Proceedings - IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, pp. 29–38, IEEE, 2019. https://doi.org/10.1109/EuroSPW.2019.00010.

[9] M. Schwartz and M. Machulak, "User-Managed Access," in *Securing the Perimeter*, pp. 267–299, Apress, Berkeley, CA, 2018. https://doi.org/10.1007/978-1-4842-2601-8_8.

[10] K. Wild, U. Breitenbücher, K. Képes, F. Leymann, and B. Weder, "Decentralized Cross-organizational Application De-

ployment Automation: An Approach for Generating Deployment Choreographies Based on Declarative Deployment Models," in *Proceedings - 33rd International Conference on Advanced Information Systems Engineering (CAiSE)*, pp. 20–35, Springer International Publishing, 2020. https://doi.org/10.1007/978-3-030-49435-3_2.

[11] J. Breen, L. Bryant, J. Chen, E. Ford, R. W. Gardner, G. Glupker, S. Griffith, B. Kulbertis, S. McKee, R. Pierce, B. Riedel, M. Steinman, J. Stidd, L. Truong, J. Van, I. Vukotic, and C. Weaver, "Managing Privilege and Access on Federated Edge Platforms," in *Proceedings - Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning) (PEARC)*, pp. 1–5, Association for Computing Machinery, 2019. https://doi.org/10.1145/3332186.3332234.

[12] G. Carcassi, J. Breen, L. Bryant, R. W. Gardner, S. Mckee, and C. Weaver, "SLATE: Monitoring Distributed Kubernetes Clusters," in *Proceedings - Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning) (PEARC)*, p. 19–25, Association for Computing Machinery, 2020. https://doi.org/10.1145/3311790.3401777.

[13] "cri-o." https://cri-o.io. Accessed March 31, 2021.

[14] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade," *ACM Queue*, vol. 14, no. 1, p. 70–93, 2016. https://doi.org/10.1145/2898442.2898444.

[15] "Controlling Access to the Kubernetes API." https://kubernetes.io/docs/concepts/security/controlling-access. Accessed March 31, 2021.

[16] M. S. Islam Shamim, F. Ahamed Bhuiyan, and A. Rahman, "XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices," in *Proceedings - IEEE Secure Development (SecDev)*, pp. 58–64, IEEE, 2020. https://doi.org/10.1109/SecDev45635.2020.00025.

[17] "Kubelet authorization." https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet-authentication-authorization. Accessed March 31, 2021.

[18] "Kubernetes Cluster Federation." https://github.com/kubernetes-sigs/kubefed. Accessed March 31, 2021.

[19] "Kubernetes Documentation - kubelet." https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet. Accessed March 31, 2021.

[20] Y.-J. Hong, "Introducing Container Runtime Interface (CRI) in Kubernetes," 2016. https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes. Accessed March 31, 2021.

[21] "Github Kubernetes Release 1.19.15 - api.pb.go." https://github.com/kubernetes/cri-api/blob/kubernetes-1.19.15/pkg/apis/runtime/v1alpha2/api.pb.go. Accessed March 31, 2021.

[22] "gRPC." https://grpc.io. Accessed March 31, 2021.

[23] "Canal." https://docs.projectcalico.org/master/getting-started/kubernetes/flannel/flannel. Accessed March 31, 2021.

[24] B. Subramaniam and C. Doyle, "Feature Highlight: CPU Manager," 2018. https://kubernetes.io/blog/2018/07/24/feature-highlight-cpu-manager. Accessed March 31, 2021.

[25] D. Hardt, "The OAuth 2.0 Authorization Framework." https://tools.ietf.org/html/rfc6749. Published October, 2012. Accessed March 31, 2021.

[26] "Keycloak." https://github.com/keycloak/keycloak. Accessed March 31, 2021.

[27] "Keycloak - Authorization Services Guide." https://www.keycloak.org/docs/8.0/authorization_services. Accessed March 31, 2021.

[28] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)." https://tools.ietf.org/html/rfc7519. Published May, 2015. Accessed March 31, 2021.

[29] "Keycloak Spring Security adapter 8.0.1." https://github.com/keycloak/keycloak/tree/8.0.1/adapters/oidc/spring-security/src/main/java/org/keycloak/adapters/springsecurity. Accessed March 31, 2021.

[30] "CloudPing." https://www.cloudping.info. Accessed March 31, 2021.

[31] "Open Policy Agent - Policy-based control for cloud native environments." https://www.openpolicyagent.org. Accessed March 31, 2021.

[32] "FUSE: Flexible federated Unified Service Environment." https://www.imec-int.com/en/what-we-offer/research-portfolio/fuse.

4
Enabling the rescheduling of containerized workloads in an ad hoc cross-organizational collaboration

This chapter presents a published research article tackling the third research question: ***how can the scheduling of containers, a task for which the cluster operator is responsible, be fitted to an unknown heterogeneous cluster environment?*** *The first part of the chapter discusses the scheduling process available in Kubernetes and identifies shortcomings when it is applied to an unknown heterogeneous cluster environment spanning multiple domains. Dealing with these kinds of technical deployment uncertainties can be solved using a probe swarm architecture allowing scheduling calamities to be solved by quickly comparing node performance levels. Most noticeable is making sure hidden technical restrictions are identified. Organizations may wish to provide a restricted view of their technical capabilities and these artificial restrictions may even change over time. To obtain an overview of cluster layout, it is thus needed to take a more active approach compared to a rather passive one sufficient for regular cluster setups in which nodes are managed by a single entity.*

⋆ ⋆ ⋆

L. Van Hoye, T. Wauters, F. De Turck, and B. Volckaert

Published in Journal of Network and Systems Management, October 2022.

Abstract A group of organizations wishing to collaborate urgently, for example in case of a crisis, needs to have a way to quickly deploy applications which enable them to speed up a potentially crisis-resolving decision-making process. A cross-organizational Kubernetes cluster, which is orchestrated by a central operator, allows to initiate these deployments in an ad hoc way. Performance issues may however arise at runtime, for example, a video pipeline belonging to a CCTV camera may produce a too low number of frames per second. The ad hoc cross-organizational collaboration case is especially prone to such issues as the set of candidate nodes and the environment in which they run may not be fully known to the operator. This chapter therefore motivates and describes the usage of a probe swarm architecture, which allows the operator to quickly generate an overview of the resource capabilities of a set of nodes, by executing code fragments locally. The obtained measurements can then enable the operator to decide on rescheduling operations. Evaluation of an illustrative probe swarm intervention shows that the performance of an example application could improve with factor five, ten or hundred when the pod would be rescheduled. This indicates that the proposed probe swarm architecture may complement other performance bottleneck detection techniques to improve performance of applications that need to be deployed urgently.

4.1 Ad hoc pod rescheduling in a cross-organizational cluster

The case studied in this chapter is that of an ad hoc cross-organizational collaboration, more specifically a set of organizations need to collaborate urgently in the case of a time-critical situation. For example, in case of an explosion on a chemical site, the company, local government, police and firefighters need to share information. Another case is that of an equipment builder connecting to the pipelines of manufacturers to quickly analyze and solve machine interruptions. In all cases, a central operator has the control over a cross-organizational cluster, as shown in Figure 4.1, which allows the deployment of software components into the different network domains to be orchestrated. The central operator thus has the role of cluster administrator. It is responsible for deploying data pipelines in the cross-organizational cluster and is thus fully aware of the data flows and dependencies that exist. It may or may not be part of a data pipeline itself, i.e. it is either one of the collaborating organizations or a facilitating third party respectively, depending on the use case discussed. One particular use case, which is practically relevant, is that of an emergency control room, in which the operator is an experienced crisis manager, managing a dashboard which is the endpoint of each data pipeline in the cluster. The chapter is written with this scenario in mind. The cluster itself, being a Kubernetes cluster [1], uses the concept of pods to distinguish groups of containerized workloads. An important observation is that, contrary to a regular Kubernetes cluster, the operator does not have a comprehensive overview of the types of nodes that are part of the cluster in such a cross-organizational setup. This set of heterogeneous nodes is composed of different hardware specifications, different network interconnections and different container runtime configurations. Furthermore, unknown background loads may be present on the worker nodes of the different organizations involved in the collaboration. The usefulness of labels indicating static hardware properties (e.g. a node having a high speed storage system like SSDs instead of slower speed physical hard disk drives) may be gone, as different organizations will likely use different key-value pairs as labels. This uncertainty is especially true when the set of nodes may change over time, due to (nodes of) organizations which join or leave the collaboration. An unknowing operator is more likely to select scheduling decisions that lead to performance issues, something which should be avoided, especially in case of an urgent problem-solving process. A few examples of cases in which a performance degradation may be noticed are:

- A pod running a data-intensive job which suffers from a low-capacity or saturated network link.
- A pod running a storage-intensive job which suffers from slow storage mount options.
- A pod running a job which suffers from the absence of GPU-acceleration.

These kinds of performance issues are all due to scheduling decisions

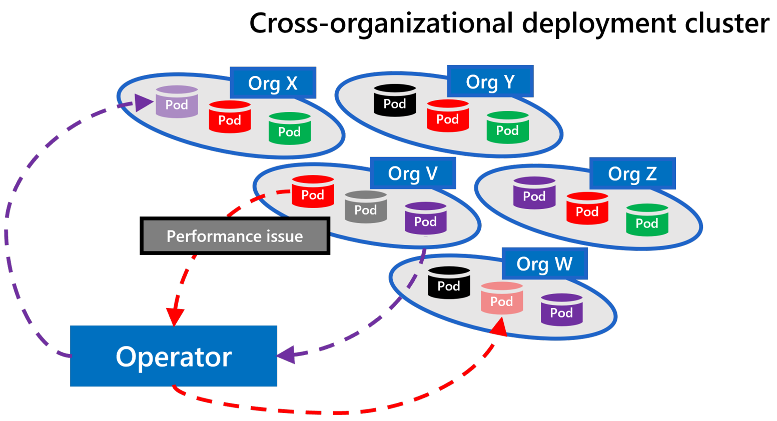


Figure 4.1: The operator, having a central overview of all cross-organizational workloads, needs assistance in the rescheduling of misplaced pods.

based on limited context information from the environment on which to schedule. The probe swarm architecture presented in this chapter enables the rollback of these situations by providing insight in the performance capabilities of the different nodes, allowing the operator to decide on urgent pod rescheduling decisions. Note that providing only node-related insight to support a rescheduling review by the operator may prove to be insufficient, as performance issues may have other causes as well, like badly written software, deadlocks, input / output delay, slow human-software interactions, etc. The ultimate goal of the rescheduling of pods is to shorten the execution time required to finish several jobs. Good scheduling decisions may impact the way a critical case is solved. The time dimension is thus of uttermost important in these ad hoc collaboration cases. In this regard, it is also important to note that the goal is not to optimize scheduling decisions, as this would require quite a lot of pod movements, causing unnecessary delays. Providing support for scheduling decisions in a (partially) unknown resource environment, based on time measurements, is the goal of the research presented in this chapter. The outline is as follows, Section 4.2 presents work related to the scheduling of workloads and advances for Kubernetes, Section 4.3 then presents shortcomings in Kubernetes for the scheduling of pods in a cross-organizational setup (first contribution), while Section 4.4 discusses the composition of a probe swarm to solve the aforementioned problem (second contribution). The evaluation of an illustrative probe swarm intervention is discussed in Section 4.5 (third contribution), after which the chapter is finalized with a conclusion and directions for future work in Section 4.6.

4.2 Related Work

The contribution presented in this chapter is part of the FUSE research project [2]. The goal of this project is to enable organizations to collaborate in an ad hoc way by constructing a cross-organizational service mesh. Goethals et al. [3] show which software components are needed to initiate such a cross-domain federation in an ad hoc way. The proposed federation allows cross-organizational deployments to be realized in a few minutes at most. More extensive research is available on the federation of scientific computing environments. Although these types of federations have a less ad hoc character, there are similar challenges to overcome, like the enforcement of diverse local organizational policies. Wickboldt et al. [4] discuss a platform which shortens the time an experienced operator and inexperienced end-users, such as companies, need to provision ad hoc cross-domain network circuits, mainly through the application of easy-to-use visual editors. This scenario closely resembles the case discussed in this chapter, especially because of the mix of manual and automatic decisions that is inevitably present within ad hoc processes.

Two other topics are already addressed by the authors in previous work. Both consider trust issues that arise when different organizations need to collaborate and thus share data. First, a logging mechanism is needed to allow an honest organization to protect itself against potentially malicious partners and to gain trust in the collaboration at hand [5]. Second, it should be possible for an organization to perform checks and balances with respect to container deployments that are suggested for its domain by a potentially malicious external operator [6]. There was still a need to further research solutions to enable these ad hoc cross-organizational collaborations as they have some important characteristics: they are applied in critical situations and should allow the involved organizations to quickly find a solution for an urgency. The scheduling of workloads in such a context is another topic that exhibits specific properties and is therefore discussed in this chapter.

A multitude of papers present ideas on how the default Kubernetes scheduler should advance. It is possible to build further on the concept of resource requests, either manually by allowing an operator to classify an application based on its resource usage [7], or automatically by means of extending the Kubernetes Vertical Pod Autoscaling feature [8]. An improved scheduler is often required when the heterogeneity of a cluster, for example in the case of fog computing, plays an important role in the performance of an application. Most papers propose solutions which try to make the scheduler aware of a distinct aspect. The focus could be on the minimization of the overall cost of a Kubernetes deployment in a cloud environment [9]. Similarly, the focus could be on reducing the end-to-end latency between applications while maintaining bandwidth requirements [10] [11], and their placement in geo-distributed environments [12]. Another emerging aspect is that of energy efficiency [13] [14], a strategy applied by a specific set of schedulers among the wider group of topology-aware [15] and hardware-aware schedulers. Examples of the latter are a GPU-aware scheduler making use of historic pod executions to speed up calculations [16] and an Intel SGX-aware scheduler [17]. Another crucial aspect focuses on the real-time utilization of node resources to schedule workloads [18]. This load-awareness is especially important in multi-tenancy cases [19], as interference effects such as cache misses and CPU context switches may lead to performance degradation. There are also papers which try to combine several of those aspects and propose a weighted multi-criteria decision strategy with the goal to optimize workload placement [20] [21]. The scale at which a scheduling algorithm needs to operate is another distinctive characteristic. A category of papers focuses on scheduling algorithms which are backed by queuing theory fundamentals [22]. Those are crucial in very large dedicated data center setups, but are thus less applicable in this ad hoc case. Finally, there is a paper which proposes an architecture that applies measurement probes at the different worker nodes [23], a similar approach as is presented in this chapter. Bayer et al. suggest the usage of both resource monitoring probes and application-specific probes, the latter to perform security checks or to monitor energy consumption. This scheduling strategy, focusing on local observations, comes closest to the one presented here. However, as with other cited related work, no other research takes into account the cross-organizational aspect and its potential consequences. The fact that the management of nodes is distributed among different organizations is specific for the Kubernetes clusters deployed in the cross-organizational collaboration case. Furthermore, where other scheduling approaches try to steer decisions based on generically applicable metrics, which are perfect for automated scheduling decisions, it is required here to gather higher-level metrics. These should indicate consequences for the collaboration in a more easy to interpret way for the operator such that a performance issue can be solved quickly. To the best of our knowledge, no other work has been conducted addressing specific concerns related to ad hoc workload scheduling, which are discussed in the next section, introduced by a cross-organizational setup.

4.3 Necessity of probes in a cross-organizational context

The probing concept presented in this chapter assists an operator in making rescheduling decisions when the cluster layout is largely unknown. The goal of this assistance is to allow collaboration applications to operate efficiently, thus without severe bottlenecks increasing execution time and downgrading quality of service. Before the probes are detailed and discussed, it is necessary to identify why they should be used in the first place. The remainder of this section therefore illustrates why a vanilla Kubernetes cluster is not sufficiently capable to achieve the proposed goal. The necessity of probes may be explained from different perspectives:

- **Requirements from the perspective of the operator**
  - The collaborations considered in this chapter are of the ad hoc type. The corresponding Kubernetes clusters are thus composed dynamically. This means that the operator is either not or only in a limited way able to reuse any previous knowledge related to cluster layout. For example, a new heterogeneous cross-organizational hardware setup does not have any Kubernetes labels attached that are meaningful for the operator. Furthermore, the operator does not have the time to thoroughly study which resources are offered by the different organizations. Ad hoc cluster management is thus needed for this case, which brings additional difficulties to making proper rescheduling decisions.
  - The implementation of the default Kubernetes pod scheduler uses a node filter function, implemented in the `NodeResourcesFit` plugin, which checks pod resource requests against the availability of resources on worker nodes

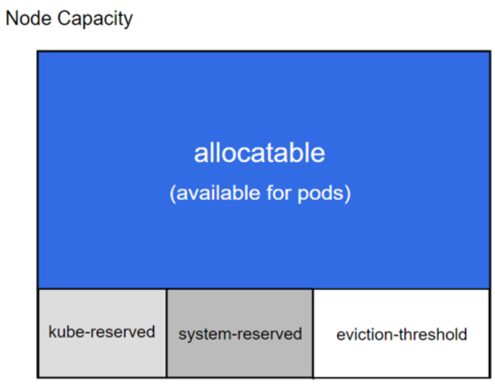


Figure 4.2: Overview of node capacity as interpreted by vanilla Kubernetes [26].

[24]. For example, when a pod requests a CPU time allocation of 1.5 / 4 physical or virtual CPUs, the scheduler will consider 2.5 CPUs to be allocatable for future pods. Figure 4.2 shows how capacities are represented: except for strictly required daemons, the remaining capacity is considered to be available for pods. The resource availability checks present in the plugin thus only consider the resource requests of pods, not their actual consumption, and most importantly, the load of processes which are not under the supervision of the Kubelet are neglected in the scheduling process [25]. This predictability in container resource consumption and consequently performance, something which is an important aspect in a container orchestrator like Kubernetes, may thus be broken in an ad hoc cross-organizational setup due to the possible presence of severe external background loads and their corresponding unknown effects.

☐ Only a very limited set of static node properties is available. It is limited to general properties such as the instruction set architecture, kernel version, operating system, Kubernetes version and container runtime version [27]. This set could easily be extended with other properties such as CPU manufacturer, clock speed, hypervisor if the worker node runs as a VM, disk manufacturer, etc. Furthermore, the metrics endpoint of each Kubelet, `/metrics/resource`, could be used to gather resources metrics of a node over time. A combination of these static and dynamic data could already provide more insight to the operator and solve some ad hoc scheduling questions. There are however two major concerns. The first concern is that it may remain difficult to derive, in an ad hoc way, node performance differences from such an extensive data set. For example, quickly comparing the performance of CPUs from different manufacturers and from different release years, each with their own set of cores, caches, multithreading settings, clock speed values, current workload, physical vs. virtual cores, etc. is almost impossible. The same observation holds for other hardware components such as memory and disk. Furthermore, organizations should be able to dynamically join or leave the collaboration at any point in time, increasing the complexity even further. The second concern is that, due to the fact that the nodes are under management of different organizations, additional node information may not always be available. Organizations may not always approve the sharing of potentially sensitive node information with external entities. Organizations may also decide to only share a restricted view on local node settings. Finally, the sharing of metrics may be limited keeping in mind the associated bandwidth cost. In all cases, it cannot merely be assumed that the operator is able to query every wished-for data.

- ☐ When pods need to communicate according to a cross-organizational flow, e.g. when they constitute a service but are deployed using an anti-affinity rule on an organizational level, it may also be needed to evaluate the performance of the links between the hosting nodes. Severe network bottlenecks may be prevented or discovered this way, allowing the operator to pick another scheduling approach.

- ■ **Requirements from the perspective of the hosting organization**

  - ☐ A hosting organization may impose additional and/or further restricting resource constraints on the containers received from the operator. These local constraints should definitely be taken into consideration for this particular use case, as hosting organizations will highly likely want to protect themselves against potentially malfunctioning or malicious operators claiming most of the available resources. It is important to note that these possible additional resource restrictions are completely unknown to the operator. The theoretically infinite set of static and dynamic data, as suggested above, thus not necessarily forms a sufficient base anymore to allow the operator to schedule properly.

☐ These locally enabled resource constraints may be the same for each container that is proposed by the operator, but they may also be different. The Docker container runtime is one example of a runtime which allows resource settings to be configured per container [28]. This dynamic aspect makes it impossible for the operator to know under which conditions the container will run. Examples of such dynamic configurations are: (1) enabling/disabling swap memory when a container process reaches the memory limit, or (2) configuring that three external containers are allowed to consume half of CPU capacity at maximum, according to a specific partitioning like $\frac{1}{4}$, $\frac{1}{6}$ and $\frac{1}{12}$. It is clear that these examples negatively affect the predictability of container performance, an important reason why Kubernetes did not support swapping to disk until version 1.23, the most recent one at the time of writing [29].

These observations lead to the conclusion that it is necessary for the ad hoc cross-collaboration case to evaluate constraints locally. Probes, which are discussed in the next section, pave the way for an operator to gather more insights supporting a rescheduling decision under these circumstances. Note that probing may not necessarily be required during the entire collaboration duration. When the availability of node resources is identified, and when the cluster operates in a more or less steady and predictable way, most uncertainties are solved.

4.4 Probe swarms enabling pod rescheduling

This section first discusses probes in general and what they could look like in a few examples. Afterwards, a possible probe swarm architecture with corresponding steps is proposed, providing an idea how probing could be integrated in vanilla Kubernetes.

4.4.1 Probes as performance indicators

A probe, in its most general definition, is a software function or a collection of functions, thus consuming (a combination of) resources such as CPU, memory, disk, GPU, network bandwidth etc. They are short-lived and finite as there are only a few tens of seconds at maximum between the probe pod starting up and tearing down. It is thus a code fragment which needs to be processed by a selection of worker nodes, allowing an operator to obtain an overview of execution times and thus relative performance differences between nodes. As is clear from this definition, a probe can be selected from an infinite pool of possible functions. Two types of probes can be identified at both ends of the spectrum. The first type of probes, the generic probes, could be applied independently of the workload that needs to be scheduled. These probes allow to dynamically pressurize target resources, as there may be a CPU-intensive probe, a memory-intensive probe, a disk-intensive probe, etc. The output produced by the probe execution is useless for the collaboration, i.e. only the corresponding execution time is important. An example of such a generic probe can be found in the class of algorithms calculating the n-th digit of Pi. One well-known use case of these algorithms is to benchmark compute infrastructure. A relevant implementation is for example the calculator TachusPi [30], which is able to calculate billions of digits using only commodity hardware. The second type of probes are exact copies of the considered workload. The containers of a pod could simply be duplicated to other nodes in the cluster. Both types of probes have clear disadvantages. The generic probes will likely provide relative performance differences between nodes if the operator can find an appropriate parameter set. However, it may remain difficult for the operator to interpret these results with regard to their impact to the collaboration at hand. A more insightful measurement may thus be handy. The copy probes on the other hand, do resemble the original pods, but their initialization may not be ad hoc and their execution may also severely impact the probed nodes and the processes that are running there. A third option is to consider application-specific probes which are somewhere in the spectrum discussed above. They could respectively allow for a more insightful and more efficient probing solution when the two discussed types of probes are not considered appropriate. Put generally, it would be possible to use derived probes, i.e. representative functions, based on the considered workload. Each workload boils down to an application, being a main function, which could be further decomposed into (much) smaller functions, each which could be used as a probe. Decomposing an application on code level seems impossible for this case, as both reverse engineering a binary and analysing the different functions takes time. It is however possible to select probes, which conceptually match several parts of the workload, from a well-prepared cross-organizational probe catalogue. The granularity of decomposition may differ per case. Theoretically, one could create a service mesh of configurable and linkable probes reflecting a multitude of applications. In reality however, due to time limitations caused by the ad hoc character of the collaboration, a line needs to be drawn between a portfolio of either more general or more specific probes. A balance between reusability and efficiency in time needs to be found.

Two application-specific probes are proposed in this chapter. Other examples are possible, but these serve to illustrate the idea behind the deployment of reusable probes, that is allowing the evaluation of a certain algorithm which may resemble a pool of possible workloads. This way, different resource consumption patterns can easily be tested on the different nodes.

- **Video processing probe**: The processing of video streams is a frequently reoccurring application in a cross-organizational collaboration. Different camera feeds may be shared in a cross-organizational collaboration, for example to allow the monitoring of an industrial site via static cameras and drones in case of an emergency situation. A screen sharing session is another application which may be used to produce a video stream. For these use cases, different video probes could be defined, e.g. a probe which pre-processes a data source and encodes it according to a video coding standard, and a probe which decodes the stream and does post-processing. These probes could then be parametrized in a such a way that different codecs could be applied, such as H.264/AVC and H.265/HEVC. As these video probes will be CPU-intensive, it might also be possible to evaluate whether a node supports parallelization through multithreading. Even more, it might be possible to shift some calculations to the GPU and check which performance improvements may be observed from GPU-enabled nodes in the cluster. Note that, contrary to the generic probes, the measured execution time of the probe provides additional insight in resource capabilities. For example, it might be interesting to know how long it takes to encode a video stream of ten seconds, using following command:

```
./encoder.exe --source cam01 --codec h.264 --time 10 --width
          640 --height 480 --output encoded.h264
```

- **Data structure probe**: Another frequently reoccurring application is a data storage solution which allows a collection of data to be stored. This data could for example be generated by a video source, a case which would allow for a probing pipeline connecting a video probe with a data structure probe. The collection may be a simple data dump, but mostly a more efficient processing solution is needed. When the collection needs to be stored according to a a specific structure, a data structure needs to be used. Well-studied data structures are for example arrays, linked lists, (binary) search trees and (binary) heaps. They only differ in their implementation of data operations, such as an insertion, deletion, lookup, traversal, sorting, etc. and corresponding asymptotic behaviors. This means that one data structure could easily be swapped for another as long as an interface of functions is implemented. Which structure needs to be chosen depends on which requirements need to be fulfilled, for example the performance of a lookup operation may be more important to that of an addition operation. The performance of these base operations may indicate how suitable a node is to assign and deploy a data storage solution. Again, the absolute execution time of the probe may be of interest here, for example to know how much data could be processed by a single node. Note that both the volatile storage in memory and the persistent storage on disk could be analysed by this specific probe. For the latter case, a tree could for example be written to a file, as is illustrated for a binary search tree (BST) by the C code in Listing 4.1.

```
// Depth-first traversal to iterate tree with non-empty root
void traversalTree(node* root, funcptr func, meta* meta) {
    if(root->left) {
        traversalTree(root->left, func, meta);
    }
    func(root, meta);
    if(root->right) {
        traversalTree(root->right, func, meta);
    }
}

// Probe function: write tree to disk
void storeTree(node* root, meta* meta) {
    meta->output = fopen("tree.txt", "w");
    ...
    // displayNode serializes a tree node
    traversalTree(root, displayNode, meta);
    fclose(meta->output);
}
```

Listing 4.1: A data structure probe should allow to evaluate disk performance of a node as illustrated in this sample.

These probes focus specifically on the resource availability of nodes, but they could also be extended to allow for the evaluation of specific network links. In general, a network probe should allow for communication between two pods hosted at different nodes in the cluster. The integration of such a network probe would allow for more end-to-end based probing tests, for example a video stream could first be pre-processed and encoded at Node X, after which it is sent to Node Y, which then decodes and post-processes the data. This way, it is possible to test whether the throughput of a network link supports the bit rate of a corresponding video stream when it is encoded using a certain parameter set, and whether certain quality of service metrics such as the number of frames per second or signal-to-noise ratio can attain certain wished-for levels.

4.4.2 Probe swarm architecture

This section will present the flow of a probe swarm architecture, more specifically the steps needed to integrate a probing solution into vanilla Kubernetes. Figure 4.3 shows the different components. Of particular interest are the purple colored components, because they represent the additional elements that are needed to achieve the proposed goal. The cross-organizational collaboration shown in the figure, illustrates the processing of three camera streams into two

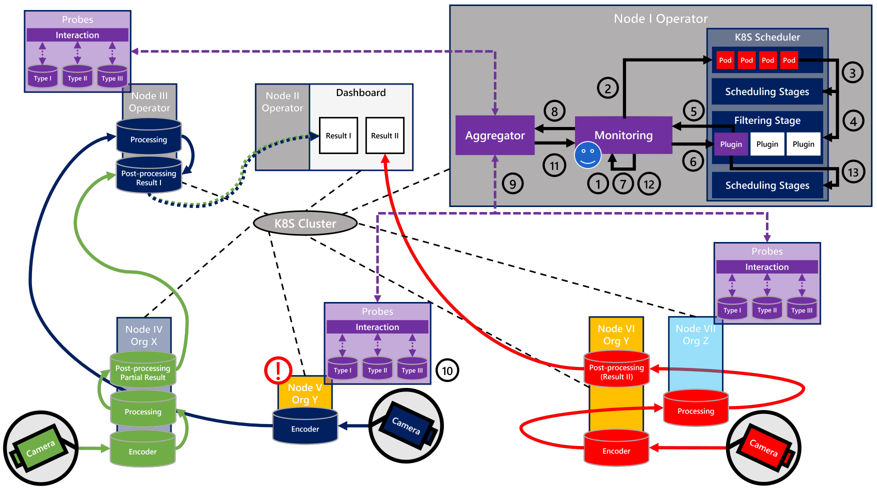


Figure 4.3: The architecture needed to deploy the proposed probe swarm.

relevant results for the case at hand, which are displayed on a dashboard under supervision of the central operator. The steps discussed below assume the case of a pod rescheduling situation caused by performance issues of a node in the cluster.

1. The operator runs a monitoring pod in its own domain, which allows to obtain a complete overview of the cross-organizational collaboration and the corresponding Kubernetes cluster. This boils down to an overview of all nodes and deployed pods, corresponding resource consumption metrics, data flows between the individual organizations based on a logging solution (e.g. like the one proposed by the authors [5]), and alerts by an alerting system. When an application needed for the cross-organizational collaboration shows issues related to quality of service, an operator may decide to reschedule the corresponding pod. This manual decision may further be supported by automated bottleneck detection techniques, but the exact rescheduling trigger does not immediately matter. In this example, a scheduling issue is present at Node V, being one of the yellow nodes of Organization Y, which will serve as an illustration in the remainder of this section.

2. The operator thus needs to find a solution to move the load of Node V. The Processing pod, responsible for analysing the video stream, may put a too heavy load on this particular node.

The operator therefore marks this pod as a candidate to be rescheduled, causing the corresponding YAML file to be sent to the Kubernetes scheduler to be inserted in the pod scheduling queue. Note that there is a dependency between the Processing pod and the Post-processing pod. The operator will first wait for an alternative node to be found for the Processing pod, after which the Post-processing pod can easily be moved to this new destination node. Although this rescheduling may not be necessary from a load perspective, it may be desirable to prevent cross-domain interactions due to accompanying network latencies.

3. Each pending pod is then processed according to a scheduling profile [31]. Such a profile consists of a number of scheduling stages which each have their extension point. Plugins implement either a single or multiple of these extension points. This step represents the pass of the Processing pod through the plugins belonging to the stages before the filtering stage.

4. The filtering stage is constituted of plugins which check for either soft or hard requirements, for example affinity/anti-affinity rules, pod spreading rules, pod resource requests, etc. The suggested probing solution is in fact an additional filtering step, as only nodes with suited performance capabilities should be considered for pod placement. A new scheduling profile should thus be defined consisting of the stages and plugins as used in the default scheduling profile, extended with a custom plugin as the last filtering step. This custom scheduling profile is then available for those pods that require a probe swarm intervention. The Kubernetes Scheduling Framework [32] allows custom plugins, implementing an extension point interface, to be compiled into the scheduler.

5. The custom plugin needs to invoke the monitoring backend of the operator using a webhook. This causes the considered pod and corresponding filtered set of nodes to be registered in the monitoring system and consequently to be presented to the operator. As the plugins in the filtering stage may evaluate nodes concurrently [32], multiple invocations per pod may be expected.

6. The response to the webhook invocation may be either a (1) request registered, response pending notification or (2) a fur-

ther, by the probing solution, filtered set of nodes. In the first case, the pod is marked as unschedulable by the custom plugin. This causes the scheduling cycle to be aborted, after which the pod is returned to the scheduling queue waiting for a consecutive cycle to be initiated [32]. Steps 3-6 are thus repeated by the scheduler as long as required. The final attempt is when the second case occurs, i.e. when the scheduling process is able to continue to step 13 with the nodes that passed the probing selection.

7. The operator then investigates the set of proposed nodes. A manual assessment of the filtered set of nodes takes place. There are two possibilities for the pod to be rescheduled:

    (a) The operator tries to reschedule the pod within an organization, so called intra-organization rescheduling. This approach may have advantages. The pod was already allowed to be deployed in the domain, meaning that a switch between nodes in the same domain would not take additional verification time. Furthermore, nodes of the same organization may be most nearby in the network, guaranteeing a more predictable continuation of operation of the pod. Applying this to the discussed example, the operator may first consider Node VI of the same Organization Y. As this node has already offloaded the Processing pod to Node VII of Organization Z, it is clear that this node should be skipped from probe evaluation.

    (b) The operator tries to reschedule across organizations, so called inter-organization rescheduling. This means, again applied to the example, that the operator should select Node III of the operator and Node VII of Organization Z, assuming these nodes were indeed part of the filtered set up to this point, together with the reference Node V, to be evaluated by the probes.

    This manual intervention may thus cause the set of potential nodes to become smaller. It is important to note that node selection is focused on finding an appropriate pod as quickly as possible. The goal is not to search any further for a better scheduling decision, as it would become an optimisation case for the entire cluster, which does not fit the ad hoc and rapidly changing scenario discussed here. This also means that it is not needed to run probes at each node, only at the selected nodes.

8. The types of probes and parameter sets selected by the operator need to be pushed to the aggregator component. A series of commands which need to be executed by the selected nodes are thus communicated in an asynchronous way. Multiple different configurations may be tested over time, enabling the operator to do some live probing.

9. The aggregator is responsible for the deployment of the probes, which are pods themselves. The `kubectl create` command thus needs to be executed. The default scheduling profile is applied to these pods, as probes should be deployed without the intervention of a probe swarm. A probe pod consists of a container which has all binaries required to probe available, and has a process running which keeps the container alive. The aggregator is then able to push commands, representing the probe executions, to such a container and to obtain time measurements. The `kubectl exec` command could for example be used by the aggregator to enable this. It is of uttermost importance to note that a probe, once selected by the operator via step 8, needs to be executed multiple times with a specified frequency, like every ten seconds. This is needed to filter outliers from the time measurements. This is the reason to keep the probe alive, as otherwise it would be needed to deploy it multiple times, causing an unnecessary overhead for the hosting node. The Processing pod in the example, may represent any video processing step. Which kind of probe is selected by the operator, will thus differ per case. It may range from the deployment of a generic probe to the deployment of a more specific probe executing a computer vision algorithm. The latter is a perfect example of a class of algorithms which could easily be prepared in a probe catalogue. For example, when the processing of the encoded video stream focuses on QR code detection, a probe may easily simulate this as follows:

```
./cv.exe --lib opencv --module objdetect --class qrcodedetector
              --function detect --input encoded.h264
```

10. It is possible to deploy multiple probe instances at a node. These probes may be of the same type, for example multiple video encoders. Such a probing intervention would allow to evaluate how many video streams could concurrently be en-

coded on a single node. The probes may also be of a different type, for example a video probe and a data structure probe. These different types may even be linked dynamically, for example when it is needed to process video first and to store it afterwards. These interactions allow for more complex probing solutions. Such dynamic links should be prepared as well, to allow the operator to quickly link different probes together.

11. The time measurements are collected by the aggregator, and aggregated for each probe. Aggregation takes place on a rolling basis, i.e. every measurement cycle, the x-th percentile may for example be calculated and passed to the operator. It should be possible for the operator to specify any custom aggregation.

12. Based on both absolute measurements, in case more specific probes are used, and relative differences in probe executions, the operator is able to obtain on overview of the resource capabilities of the different nodes. A weighted evaluation of probe results may be part of this assessment. The operator then manually selects one or multiple nodes, which constitute the new filtered set of nodes. This set of nodes is then passed as a response to the webhook discussed in step 5-6, to allow the scheduling process to progress.

13. The nodes are then further processed by the remaining scheduling stages. Ranking the nodes based on different scoring criteria is the main goal of this final evaluation. For example, nodes which already have the required container image, may be favored. Finally, the most favored node is chosen and binding between pod and node takes place. This flow thus allows an operator to manually intervene a scheduling operation, based on the deployment of probes, which inform the operator about the potentially unknown underlying deployment cluster.

4.5 Evaluation

The case discussed in previous section can be simplified a bit, obtaining a situation as shown in Figure 4.4. It defines a cross-organizational collaboration between organizations X, Y, Z which is orchestrated by a central operator. An evaluation is presented in this section to illustrate the potential of the proposed probe mesh and to show to which performance improvements it could lead. The goal for

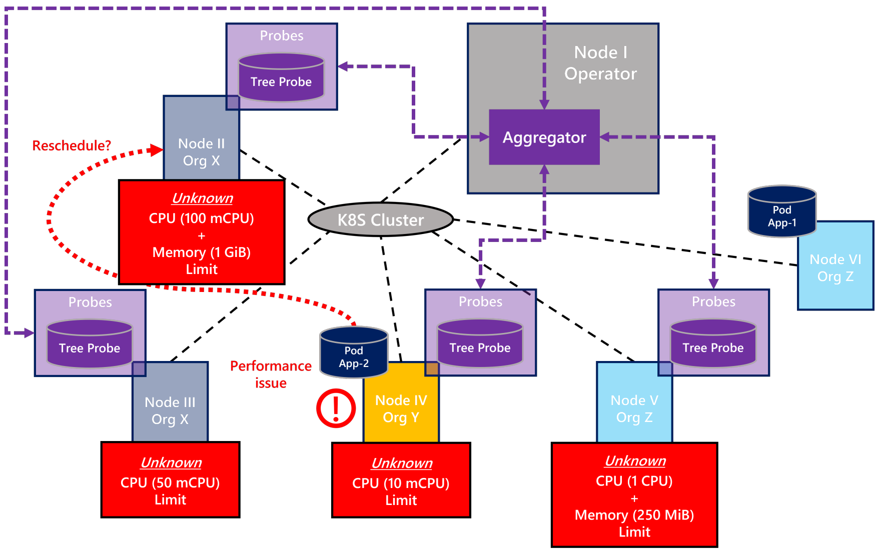


Figure 4.4: An example use case to which the proposed probe swarm is applied.

the operator is to reschedule one of the dark blue collaboration pods. More specifically, the `App-2` pod, which is deployed at Organization Y initially, needs to be moved to another node, due to the reason explained in the next paragraph.

The Kubernetes cluster used for this evaluation consists of homogeneous nodes: each node is a virtual machine (VM) running Ubuntu 18.04 LTS and is equipped with four vCPUs of an Intel Xeon E5645 2.4 GHz processor and 4 GiB of RAM. There are no hardware differences between the nodes, and no additional background loads are deployed. To realize performance differences between nodes, local resource limits are set, as explained in Section 4.3. These limits, for now focusing on CPU and memory usage, are enforced by a hosting organization, and are thus unknown to the operator. This way, a heterogeneous cluster is realized, from the perspective of the operator. The local limits used in this example are shown in the figure. Organization Y, hosting Node IV, applies a restrictive CPU limit to the containers in the collaboration pod, as only one hundredth of a core may be consumed. This hurts performance significantly, implying the need for a rescheduling. The set of candidate nodes consists of Node II, III and V. Node VI is excluded from this set, as it already hosts the `App-1` pod, and for example an anti-affinity rule may require to put both pods on different nodes to achieve certain crash fault-tolerance.

It is assumed that the collaboration pods considered here represent a storage solution. A data structure probe, as suggested in Section 4.4, is thus selected by the operator for this evaluation scenario. Assume further that the performance of data lookups is crucial for the considered application. Therefore, a balanced BST tree may be selected as data structure type, as it is known for its relatively short lookup times. To obtain an idea of the relative performance differences between nodes and to obtain an idea of the time it takes to store a certain amount of data given the chosen data structure, it is sufficient to create, and to free afterwards, such a balanced BST tree and to perform time measurements. The following code fragment, written in C, shows how such a probing solution could be defined.

```c
// Node representation
typedef struct _node {
    int id; char* data; struct _node* left, right;
} node;

// Probe part I
//   Create a balanced BST with 'max' nodes, each with 'bytes' data
void createTree(node** root, int min, int max, int bytes) {
    int id = (min + max) / 2;
    // Insert a leaf by following path from the root
    insertNode(root, id, bytes);
    if (min == max) { return; }
    if (id > min) { createTree(root, min, id - 1, bytes); }
    createTree(root, id + 1, max, bytes);
}

// Probe part II
//   Free dynamic memory occupied by all tree nodes
void freeTree(node* root);
```

Listing 4.2: A tree probe to capture the performance differences between nodes in the cluster.

The size of a tree node is equal to 32 bytes when the GCC compiler is used on a 64 bit machine: 4 bytes for the ID integer plus 4 padding bytes, 8 bytes for the data pointer, and 8 bytes for both the left and right child pointer. The ID field ranges from 1 to N and is used to sort the items in the BST, while the `data` field contains a string of D random alphanumerical characters. Using this code, the operator could launch multiple short-lived tree probes and experiment with different tree sizes, to obtain an overview of the performance differences between the nodes in the candidate set. A strategy could for example be to gradually intensify the probing experiment, by tuning the dominant parameters N and D, to prevent a probe from having a too significant impact on the performance of a node. The results of such an evaluation are presented in Table 4.1. The experiments are executed using containers with Ubuntu 18.04 image running a `sleep` process to keep the container alive. The probe executable file and corresponding time measurements are then initiated using the following command:

```
time -f %e ./tree.exe --type bst --balanced true --nodes N --data D
```

The table shows the sorted execution times of five probe cycles of the tree probe at the different candidate nodes for six configurations of N and the size of a single tree node $S = 32 + D$ in bytes. Only the configurations for which at least 100 MB needs to be allocated, based on the analysis of dynamic memory allocations, are evaluated. The reason for this is that smaller cases tend to only show negligible performance differences, while larger cases are too large for a 4 GiB RAM capacity. The obtained time measurements clearly match with the ratios of the local CPU limits: compared with reference Node IV, Node II executes about ten times quicker, Node III executes about five times quicker, and Node V executes about hundred times quicker. This result is to be expected from the evaluation setup presented here, but such an insight is important for the operator in an unknown setup. Significant performance increases may thus be gained when the `App-2` pod would be rescheduled. The deployment of the probe swarm furthermore shows that different candidate nodes may be recommended for the different probe configurations. This thus proves the need for probes, as it may be completely unclear for the operator which nodes will perform well under an unknown set of local settings. It becomes clear that some probe pods are not able to finish their execution, as their corresponding probe containers exceed the memory limits as defined by the local administrator of the hosting organization. This causes them to be terminated, more specifically to be out of memory (OOM) killed. The evaluation also confirms that parameter settings should be increased gradually, as suggested above. The probe executions for memory sizes larger than 100 MB may already take around one minute or more for the probe to finish. A possibility to solve this issue partially, is to stop a running probe at a Node X when its execution time significantly surpasses the one of an already finished probe at a Node Y. When no probe finishes quickly, it is needed to specify an upper limit on execution time. Finally, it is clear that when N increases, more overhead is present. The $N = 10^6$, $S = 32$ B + 1 kB configuration needs more time to execute and results in an extra OOM killed container, although the data structure allocates roughly the same amount of dynamic memory as in the $N = 10^4$, $S = 32$ B + 100 kB case.

The five probe measurements executed at each node, as presented in Table 4.1, are close to each other. This means that additional probe executions or cycles would not lead to significantly different conclusions, as no varying deployment conditions are considered. In a real scenario however, as already mentioned in step 9 of Section 4.4.2, it may be that dynamic factors such as background loads are present, causing probing results to vary and insights in the performance of nodes to change. It is therefore crucial that multiple probe cycles are evaluated over time. It is possible to illustrate this using a hypothetical scenario in which a probe swarm is deployed on four nodes and the results of ten probe cycles are gathered. The probe swarm measurements could and should be analyzed in multiple ways depending on the situation at hand. Figures 4.5, 4.6 and 4.7 show a possible evaluation under the assumption that it is needed to find the node which is most likely to perform best. This results in a binary evaluation per probe cycle, i.e. a node has the lowest probe execution time or not. Based on this evaluation, it is possible to calculate for each node its share in fastest probe executions. The three illustrations should be interpreted as follows:

- Figure 4.5: The default scenario is that of a weighted evaluation of all cycles, for example using equal weights as shown here. Node C is not a candidate node as it never has the fastest probe execution. A reason could be that this node is lagging behind clearly, for example when a single probe execution takes longer than the time between two consecutive probe cycles, causing it to be excluded from further evaluation to speed up the evaluation process. Given the observations after ten cycles, the operator may decide to choose Node B as it has the highest chance of being the most performing one. This result depends on a number of parameters such as the duration of probing, the number of probing cycles and the weight distribution. The number of required probing cycles could depend on the expected duration of the application to be scheduled. When a longer-running job

Table 4.1: Time measurements in seconds of tree probe executions at Nodes II, III, IV and V for different configurations of N and S, which represent the number of tree nodes and size of a tree node in orders of bytes respectively.

	S: (32 + 10) B	S: 32 B + 1 kB	S: 32 B + 100 kB	S: 32 B + 1 MB
N: 10^2	4.2 kB $\ll$ RAM capacity	103.2 kB $\ll$ RAM capacity	10.0 MB $\ll$ RAM capacity	100.0 MB II: 3.9 4.1 4.1 4.2 4.3 s III: 8.0 8.1 8.2 8.2 8.3 s *IV: 41.1 41.4 41.5 41.7 42.1 s* **→ V: 0.4 0.4 0.4 0.4 0.4 s**
N: 10^3	42 kB $\ll$ RAM capacity	1.0 MB $\ll$ RAM capacity	100.0 MB II: 4.3 4.3 4.4 4.6 4.7 s III: 8.4 8.4 8.5 8.5 8.7 s *IV: 42.5 42.7 42.8 43.1 44.5 s* **→ V: 0.4 0.4 0.4 0.4 0.5 s**	1.0 GB **→ II: 39.9 40.0 40.7 41.6 42.4 s** III: 80.4 80.8 81.8 82.1 84.2 s *IV: 414.4 417.2 418.6 428.6 429.7 s* V: OOM
N: 10^4	420 kB $\ll$ RAM capacity	10.3 MB $\ll$ RAM capacity	1.0 GB **→ II: 42.3 42.4 42.6 42.9 43.3 s** III: 85.5 85.6 87.4 87.4 87.6 s *IV: 443.0 444.8 444.8 445.4 446.7 s* V: OOM	10.0 GB $\gg$ RAM capacity
N: 10^5	4.2 MB $\ll$ RAM capacity	103.2 MB II: 5.8 5.8 5.9 6.6 6.6 s III: 11.8 12.2 12.2 12.5 12.7 s *IV: 61.6 62.0 62.2 62.5 62.6 s* **→ V: 0.6 0.6 0.6 0.6 0.7 s**	10.0 GB $\gg$ RAM capacity	100.0 GB $\gg$ RAM capacity
N: 10^6	42 MB $\ll$ RAM capacity	1.0 GB II: OOM **→ III: 135.5 136.4 136.8 136.8 137.0 s** *IV: 728.9 735.0 737.6 739.2 740.2 s* V: OOM	100.0 GB $\gg$ RAM capacity	1.0 TB $\gg$ RAM capacity

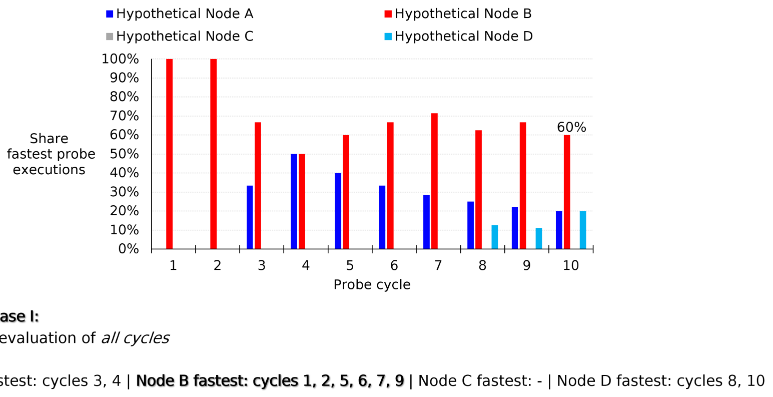


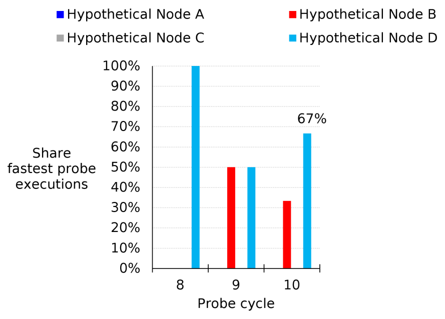
Figure 4.5: The analysis of all probe cycles, suggesting the selection of Node B.

is considered, it would be better to consider more probe cycles, as it provides a more reliable historic view on node performance.

- Figure 4.6: Contrary, for shorter-running jobs, it is less valuable to take older probe cycles into account, as the more recent node performance measurements are more relevant. Only evaluating the latest three cycles is thus another possibility. The operator would then select Node D in this example, as this one shows promising results during the latest probe cycles.

- Figure 4.7: Additional criteria could be applied to the evaluation of the probe cycles. It could be chosen, for example, that a node at cycle x is considered fastest only if it lowers execution time with $> 30\%$ compared with the fastest node up until cycle $x - 1$. This way it is prevented that marginal performance changes have a significant impact on the decision of the operator. This would mean for the example that, although Node D suddenly shows promising results during the latest probe cycles, its performance results are only slightly better than those of Node B. It is therefore conceivable to just ignore them, causing the operator to be more confident about the selection of Node B.

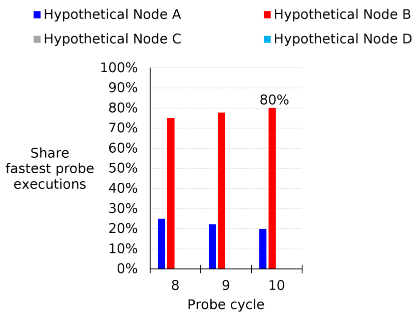
This example makes clear that different data analyses are possible and that there is not necessarily a single correct solution. Which



Example case II:
Weighted evaluation of *latest three cycles*

Node A fastest: - | Node B fastest: cycle 9 | Node C fastest: - | **Node D fastest: cycles 8, 10**

Figure 4.6: The analysis of the latest three probe cycles, suggesting the selection of Node D.



Example case III:
Weighted evaluation of *all cycles*, but with the extra condition that a node at cycle x is considered fastest only if it lowers execution time with >30% compared with the fastest node up until cycle x-1

Node A fastest: cycles 3, 4 | **Node B fastest: cycles 1, 2, 5, 6, 7, 8, 9, 10** | Node C fastest: - | Node D fastest: -

Figure 4.7: The analysis of all probe cycles, but with an extra condition, again suggesting the selection of Node B.

scheduling decision should be taken and which result it will bring depends on many factors. The operator could for instance follow the strategy to pick the node which is most likely to perform best based on probe input. However, this may be an expensive node to schedule, in case costs are considered relevant in the collaboration. Furthermore, it may not necessarily be needed to pick the top-performing node. Imagine a video pipeline, for which the video processing probe is used. If it is only required to be able to process 25 frames per second, it is unnecessary to select any node for which probe evaluation shows a higher potential capacity, assuming the nodes considered show a comparable stability over time. The key message of this chapter is that probes are needed in a cross-organizational setup to fuel these types of analyses. Without them, an operator would only be able to perform limited analyses, and as such only gain limited insight in the performance differences between nodes in the cluster.

4.6 Conclusion

Probes are needed in a cross-organizational cluster to allow an operator to make ad hoc decisions on the placement of pods. There are simply too many uncertain factors from the perspective of this operator, as the worker nodes are managed by other organizations. Background loads may thus be present, the state of underlying hardware may be (partially) unknown, and local performance limits may be applied. These unique conditions in a dynamically composed cluster require the intervention of a probe swarm. As discussed, different types of probes are possible, ranging from generic probes to copy probes, and application-specific probes. The latter are perfectly suited to simulate performance differences between nodes when typical cross-organizational applications are considered. A possible integration of these probes into the vanilla Kubernetes scheduling pipeline is presented, allowing for proper rescheduling of misplaced pods. This rescheduling is especially important for the case at hand, as an ad hoc collaboration should not be delayed due to resource bottlenecks. Finally, a set of probes based on a BST are deployed and evaluated when local resource limits are applied to the container of the probe pod. It shows that these limits, which are unknown to the scheduling operator, can have a significant impact on the execution time and thus performance of the proposed application-specific probe: rescheduling the pod may improve performance with a factor five, ten or even hundred. Future work should focus on bottleneck discovery of microservice applications, being it cross-organizational services or not. There may be plenty of reasons for an application to underperform. Automatic mechanisms are thus required to trace application state, for example using a (distributed) tracing framework like OpenTelemetry [33]. This may be a complex task to solve when a multitude of services interact with each other. Based on gathered observations, it may be easier for an operator to pinpoint reasons behind an application stall. One of these reasons may be the misplacement of an application within the cluster, triggering the deployment of a probe swarm in the case of a cross-organizational setup.

Acknowledgments

The work described in this chapter, was partly funded by the FUSE research project [2], in which a Flexible federated Unified Service Environment was investigated. The project was realized in collaboration with imec. Industry project partners were Barco, Axians and e-BO Enterprises, with project support from VLAIO (Flanders Innovation & Entrepreneurship).

Bibliography

[1] "Kubernetes." https://kubernetes.io.

[2] "FUSE: Flexible federated Unified Service Environment." https://www.imec-int.com/en/what-we-offer/research-portfolio/fuse.

[3] T. Goethals, S. Kerkhove, L. Van Hoye, M. Sebrechts, F. De Turck, and B. Volckaert, "FUSE: A Microservice Approach to Cross-domain Federation using Docker Containers," in *Proceedings - 9th International Conference on Cloud Computing and Services Science (CLOSER)*, (Heraklion, Greece), pp. 90–99, SciTePress, 2019. https://doi.org/10.5220/0007706000900099.

[4] J. A. Wickboldt, M. Q. Guerreiro, L. Z. Granville, L. P. Gaspary, M. F. Schwarz, C. Guok, V. Chaniotakis, A. Lake, and J. MacAuley, "MEICAN: Simplifying DCN Life-Cycle Management from End-User and Operator Perspectives in Inter-Domain Environments," *IEEE Communications Magazine*, vol. 56, no. 1, pp. 179–187, 2018. https://doi.org/10.1109/MCOM.2017.1601205.

[5] L. Van Hoye, T. Wauters, F. De Turck, and B. Volckaert, "Trustful ad hoc cross-organizational data exchanges based on the Hyperledger Fabric framework," *Int J Network Mgmt*, vol. 30, no. 6, p. e2131, 2020. https://doi.org/10.1002/nem.2131.

[6] L. Van Hoye, T. Wauters, F. De Turck, and B. Volckaert, "A secure cross-organizational container deployment approach to enable ad hoc collaborations," *Int J Network Mgmt*, vol. 32, no. 4, p. e2194, 2022. https://doi.org/10.1002/nem.2194.

[7] V. Medel, C. Tolón, U. Arronategui, R. Tolosana-Calasanz, J. Á. Bañares, and O. F. Rana, "Client-Side Scheduling Based on Application Characterization on Kubernetes," in *Proceedings - 14th International Conference on Economics of Grids, Clouds, Systems and Services (GECON)*, (Biarritz, France), pp. 162–176, Springer International Publishing, 2017. https://doi.org/10.1007/978-3-319-68066-8_13.

[8] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring Potential for Non-Disruptive Vertical Auto Scaling and Resource Estimation in Kubernetes," in *Proceedings - IEEE 12th International Conference on Cloud Computing (CLOUD)*, (Mi-

lan, Italy), pp. 33–40, IEEE, 2019. https://doi.org/10.1109/CLOUD.2019.00018.

[9] Z. Zhong and R. Buyya, "A Cost-Efficient Container Orchestration Strategy in Kubernetes-Based Cloud Computing Infrastructures with Heterogeneous Resources," *ACM Trans. Internet Technol.*, vol. 20, no. 2, pp. 1–24, 2020. https://doi.org/10.1145/3378447.

[10] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Resource provisioning in Fog computing: From theory to practice," *Sensors*, vol. 19, no. 10, p. 2238, 2019. https://doi.org/10.3390/s19102238.

[11] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards delay-aware container-based Service Function Chaining in Fog Computing," in *Proceedings - IEEE/IFIP Network Operations and Management Symposium (NOMS)*, (Budapest, Hungary), pp. 1–9, IEEE, 2020. https://doi.org/10.1109/NOMS47738.2020.9110376.

[12] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with Kubernetes," *Computer Communications*, vol. 159, pp. 161–174, 2020. https://doi.org/10.1016/j.comcom.2020.04.061.

[13] P. Townend, S. Clement, D. Burdett, R. Yang, J. Shaw, B. Slater, and J. Xu, "Invited Paper: Improving Data Center Efficiency Through Holistic Scheduling In Kubernetes," in *Proceedings - IEEE International Conference on Service-Oriented System Engineering (SOSE)*, (San Francisco, CA, USA), pp. 156–15610, IEEE, 2019. https://doi.org/10.1109/SOSE.2019.00030.

[14] I. Rocha, C. Göttel, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni, "Heats: Heterogeneity-and Energy-Aware Task-Based Scheduling," in *Proceedings - 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, (Pavia, Italy), pp. 400–405, IEEE, 2019. https://doi.org/10.1109/EMPDP.2019.8671554.

[15] "Topology Aware Scheduling." https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/kep/119-node-resource-topology-aware-scheduling. Accessed March 1 2022.

[16] G. El Haj Ahmed, F. Gil-Castiñeira, and E. Costa-Montenegro, "KubCG: A dynamic Kubernetes scheduler for heterogeneous clusters," *Software: Practice and Experience*, vol. 51, no. 2, pp. 213–234, 2021. https://doi.org/10.1002/spe.2898.

[17] S. Vaucher, R. Pires, P. Felber, M. Pasin, V. Schiavoni, and C. Fetzer, "SGX-Aware Container Orchestration for Heterogeneous Clusters," in *Proceedings - IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, (Vienna, Austria), pp. 730–741, IEEE, 2018. https://doi.org/10.1109/ICDCS.2018.00076.

[18] "KEP - Trimaran: Real Load Aware Scheduling." https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/kep/61-Trimaran-real-load-aware-scheduling. Accessed March 1 2022.

[19] A. Tzenetopoulos, D. Masouros, S. Xydis, and D. Soudris, "Interference-Aware Orchestration in Kubernetes," in *Proceedings - International Conference on High Performance Computing*, (Frankfurt, Germany), pp. 321–330, Springer International Publishing, 2020. https://doi.org/10.1007/978-3-030-59851-8_21.

[20] T. Menouer, "KCSS: Kubernetes container scheduling strategy," *J Supercomput*, vol. 77, no. 5, p. 4267–4293, 2021. https://doi.org/10.1007/s11227-020-03427-3.

[21] T. Rausch, A. Rashed, and S. Dustdar, "Optimized container scheduling for data-intensive serverless edge computing," *Future Generation Computer Systems*, vol. 114, pp. 259–271, 2021. https://doi.org/10.1016/j.future.2020.07.017.

[22] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel, "Job-Aware Scheduling in Eagle: Divide and Stick to Your Probes," in *Proceedings - Seventh ACM Symposium on Cloud Computing*, (Santa Clara, CA, USA), p. 497–509, Association for Computing Machinery, 2016. https://doi.org/10.1145/2987550.2987563.

[23] T. Bayer, L. Moedel, and C. Reich, "A Fog-Cloud Computing Infrastructure for Condition Monitoring and Distributing Industry 4.0 Services," in *Proceedings - 9th International Conference on Cloud Computing and Services Science (CLOSER)*,

(Heraklion, Greece), pp. 233–240, SciTePress, 2019. https://doi.org/10.5220/0007584802330240.

[24] "Kubernetes 1.19.16 - noderesources/fit.go." https://github.com/kubernetes/kubernetes/blob/v1.19.16/pkg/scheduler/framework/plugins/noderesources/fit.go#L230. Accessed March 1 2022.

[25] "Nodes - Resource capacity tracking." https://kubernetes.io/docs/concepts/architecture/nodes/#node-capacity. Accessed March 1 2022.

[26] "Reserve Compute Resources for System Daemons - Node Allocatable." https://kubernetes.io/docs/tasks/administer-cluster/reserve-compute-resources/#node-allocatable. Accessed March 1 2022.

[27] "Nodes - Info." https://kubernetes.io/docs/concepts/architecture/nodes/#info. Accessed March 1 2022.

[28] "Docker - Runtime options with Memory, CPUs, and GPUs." https://docs.docker.com/config/containers/resource_constraints. Accessed March 1 2022.

[29] E. Hashman, "New in Kubernetes v1.22: alpha support for using swap memory." https://kubernetes.io/blog/2021/08/09/run-nodes-with-swap-alpha (2021). Accessed March 1 2022.

[30] F. Bellard, "TachusPI Documentation." https://bellard.org/pi/pi2700e9/readme.html (2009). Accessed March 1 2022.

[31] "Scheduler Configuration." https://kubernetes.io/docs/reference/scheduling/config. Accessed March 1 2022.

[32] "Scheduling Framework." https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework. Accessed March 1 2022.

[33] "OpenTelemetry." https://opentelemetry.io. Accessed March 1 2022.

5 Enabling organizations to participate in the ad hoc scheduling of a cross-organizational data pipeline

Finally, this chapter presents a submitted research article tackling the fourth research question: ***how can the scheduling of containers, a task for which the cluster operator is responsible and which is fitted to an unknown heterogeneous cluster environment, be extended to allow for negotiation between the participating organizations?*** *In addition to data pipelines being deployed in a single domain, it is possible to deploy data pipelines crossing multiple domains. For the latter, multiple stakeholders can be identified which may all have a vision on how this pipeline should be deployed among them. Proactively taking into account their preferences is necessary to limit the number of time-consuming scheduling attempts. Most noticeable is thus the consideration of hidden non-technical scheduling requirements during the scheduling process. Together with the earlier discussed hidden technical limitations, resolved by means of the probing concept, they constitute the set of scheduling requirements. This set may change through negotiation as discussed in the chapter.*

⋆⋆⋆

L. Van Hoye, T. Wauters, F. De Turck, and B. Volckaert

Submitted for review to the Journal of Network and Systems Management, October 2022.

Abstract In case of an emergency situation, it is required to come up with solutions quickly. The supporting decision-making process should therefore be based on relevant data sources which are fed to data processing pipelines. These data sources may however be located in different domains of distinct organizations. Although the technical realisation of cross-organizational data pipelines is possible, for example using a federated Kubernetes cluster managed by a central operator, it is unclear how those pipelines should be scheduled among the participating organizations. Firstly, the cross-organizational infrastructure is unknown and highly heterogeneous, and secondly, there may be undiscovered scheduling preferences present. The first issue can be solved using software probes, while the second issue will be solved through an extension of this probing concept. During the proposed inter-domain scheduling process, organizations may specify monetary reward requirements and requirements on the maximum load they wish to bear. The former allows administrators to specify a lower limit with respect to payment they request for their contribution, the latter allows the specification of an upper limit on the workload an organization wishes to process. This way, mismatching expectations on contribution level, which may potentially cause harm from the hosting organization perspective, are prevented, increasing the trust level of contributors. These kinds of requirements have nothing to do with the technical assessment of a node, but they do impact scheduling decisions and performance of an application, as shown in our evaluation. The proposed scheduling flow not only allows organizations to steer scheduling decisions, but also to negotiate requirements among each other, giving rise to ad hoc conflict resolution all the while collaborating in solving the emergency situation.

5.1 Introduction

Ad hoc decision-making is crucial in case of an emergency situation. An explosion at a chemical site, in case lives are at risk, or a shutdown of a production plant, in case loss of revenue is at stake, are

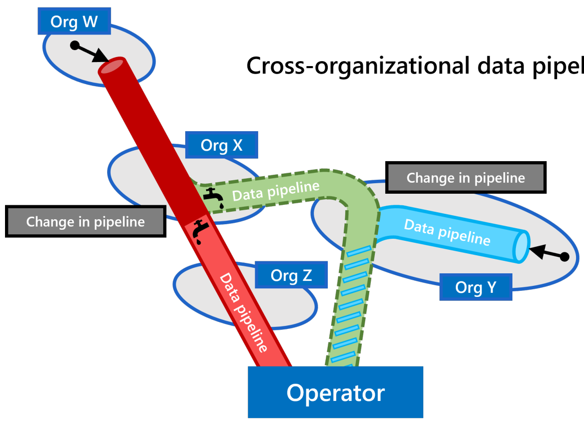


Figure 5.1: The set of stakeholders contributing to a cross-organizational data pipeline may change dynamically during the collaboration period.

just two examples. Organizations should therefore have technical enablers to allow them to quickly setup an ad hoc cross-organizational collaboration between them. Such a collaboration consists of multiple data pipelines in which data is gathered, processed and presented to the relevant stakeholders. The orchestrator Kubernetes [1], allowing a central operator to manage containerized workloads, can be used to federate different domains and create this desired setup [2]. However, the orchestration of workloads in a cross-organizational setup is more complex than the regular cluster scheduling process managed by a single entity. The environment is highly heterogeneous, meaning that nodes will have different performance levels due to variations in hardware, container runtime, network connections, background loads, artificial resource limits set by the local administrator, etc. Furthermore, data pipelines may cross organizational boundaries, as shown in Figure 5.1. These cross-organizational data pipelines require a certain contribution from different organizations. Finding an agreement on the level of contribution by each of the stakeholders is the challenge studied in this chapter. The figure illustrates the following scenarios:

- The initial situation is that of a red-colored data pipeline which crosses organisations W, X, and Z. It terminates at the operator where it may be used to feed a dashboard. At a certain point in time, it may become clear that an organization, imagine Org Z, is responsible for the performance degradation of the pipeline. When this organization is unable to reorganize resources locally to mitigate this, it is required to reschedule the workload to an external location. Assume this is the case and that a technical assessment indicates that Org Y is capable to solve this performance issue. Although Org Y hosts suitable nodes from a technical perspective, it is unclear whether this organization is willing to contribute to this green-colored pipeline. Furthermore, it may want to obtain guarantees about its contribution to the pipeline and corresponding reward, as it in fact pre-finances associated costs. This trust issue related to compensation is not present in a regular deployment cluster spanning a single organization.

- In case of a divisible workload, it is possible to examine a more generic solution compared to a full rescheduling. Balancing the load between Org Z and Org Y is an alternative in that case. These kinds of workloads allow for both parallel and concurrent execution in the cross-organizational layout. Workload changes may then even occur more frequently. Again, the organizations involved in a data pipeline, here W, X and Y, need to have a say in this workload division. The configuration of a load-balancing component is thus not merely a decision by the operator.

- The data pipeline may be extended, as shown by the blue-colored pipeline to which Org Y contributes, as a mix of data inputs may result in a more insightful dashboard overview. This causes not only Org W, but also Org Y, to become a data source owner. These owners may have specific requirements regarding their contribution level, which are unknown upfront by the operator. Due to their crucial role in the functioning of the data pipeline, it is key to understand their desired participation conditions.

It is with these types of cross-organizational dynamics in mind that the remainder of this chapter is written. Section 5.2 discusses how this chapter relates to other research papers which already address the need for negotiation during the scheduling of workloads. The different components and accompanying steps required to fit a cross-organizational scheduling process are then presented in Section 5.3. An illustrative evaluation is presented in Section 5.4, indicating what performance differences may be observed when custom requirements are negotiated upfront. Finally, Section 5.5 provides a conclusion on the topic and a possible direction for future work.

5.2 Related Work

This chapter is written in the context of the FUSE research project [3], which aims to create an ad hoc federated unified service environment among organizations involved in the collaboration. Earlier publications have discussed other trust issues related to this setup. Research has been published on how a cross-organizational deployment cluster based on Kubernetes could be deployed in a matter of minutes [2], how data exchange disputes could be prevented using a suitable logging mechanism [4], how container deployments proposed by a potentially malicious external entity could be verified pre-deployment [5], and finally how the Kubernetes scheduling process should be adapted to take into account the technical uncertainties related to the cross-organizational setup [6]. The latter topic is further extended in this chapter and Section 5.3.1 therefore revisits its contribution. In summary, what is still missing, is a negotiation procedure allowing hosting organizations and the operator to communicate hidden requirements among each other. These requirements, set out of a lack of full trust, may either be technical or non-technical and hosting organizations are able to manipulate them.

When searching for related work addressing negotiation needs within the cloud domain, it becomes clear that the management and especially negotiation of service level agreements (SLAs) is a widely studied topic [7]. These SLAs are legal agreements between resource consumers and cloud providers defining expectations of typical parameters such as average response time of a service, the price model that is applied, the level of availability, etc. Fundamental to this negotiation process is the observation that providers want to maximize their profit, while resource consumers want to have desired quality of service at the lowest cost possible. The participants in an SLA negotiation thus try to optimize their objectives. Dedicated papers therefore exist which focus on the maximization of some objectives, for example, in case of a cloud provider, it is possible to differentiate SLA negotiation according to different business-level objectives, one of them being revenue maximization [8]. The negotiation process itself is extensively studied as well. Multiple approaches are available, for example to automate SLA negotiation in case a composite service, i.e. a service which consists of a set of logically connected services, needs to be negotiated [9] [10]. A recent paper by Omezzine et al. even proposes a generic framework regarding automated negotiation, identifying which types of agents and which interactions are needed to allow negotiation between multiple layers: the users layer, SaaS / PaaS layer, and IaaS layer [11]. The purpose of these more technical papers is to identify which negotiation protocol is needed, how negotiation should be coordinated, which negotiation strategy should be used (e.g. based on game theory), which metrics need to be evaluated to assess proposals, how counterproposals should be generated, etc. Although these automated procedures could equally be used to negotiate parameters relevant to a cross-organizational collaboration, at least from a theoretical point of view, there is an important practical consideration. The participating organizations in the ad hoc collaboration discussed here, may not be prepared to initiate a complex SLA negotiation among them, meaning that the agents and strategies required for this process will likely be missing. Even worse, organizations may not have any prior experience with SLA negotiation. To force the setup of an advanced negotiation infrastructure in this case, it would be necessary for (a subset of) organizations to trust an external entity for the proper configuration of its agent. This assumption may not hold for any collaboration. Due to the urgency and temporariness, a negotiation procedure is therefore needed in which rules are more basic, to allow for the ad hoc nature of decisions. Automation could still be considered relevant, in case agents are present, but rather in a supportive role. The incorporation of manual decisions should be investigated instead as it reduces the burden to participate and as it allows decision-making to happen instantly. The focus in this chapter is thus not necessarily on the procedure itself, but rather on which elements need to be negotiated to increase the trust level of organizations enabling a cross-organizational data pipeline. To the best of our knowledge, no previous research is available addressing the urgent negotiation of both technical and non-technical limitations for each participant in an accessible manner.

What is particularly interesting to note in a significant subset of the available SLA negotiation research, is the presence of a third party broker, also commonly referred to as auctioneer, within the negotiation process. This is typically the case in a double auction approach in which the goal is to satisfy both sides, i.e. the resource consumer and provider. The papers published by Samimi et al. [12] and Mao et al. [13] are two such examples implementing a centralized auction-based approach. There is however a security risk associated with this setup [14] [15]. Negotiation participants need to share potentially sensitive information with this third party and it needs to be trusted for that reason. Furthermore, it could become a performance bottleneck. The cluster operator present in the setup discussed here, plays a similar role as central coordinator. It does not act as a broker, but it is responsible for the technical support of the negotiation process. The suggested concerns are however not immediately present here. As the collaboration duration is short and as only a relatively small portion of infrastructure is going to be offered by a hosting organization, the security risks associated with the communication of either technical or non-technical limitations seems to be minimal. Furthermore, this leakage of information does not bring any advantages to the other participants, as they will likely not be considered as an opponent. Quite the opposite, openly expressing requirements across participants is considered to be an asset in this case, as organizations gain maximum insight in each others behavior and can adapt their own based on this. Maximum visibility of these otherwise hidden requirements should thus be supported to ensure participants have enough confidence to join and thus allow the deployment of the cross-organizational data pipeline to succeed. Fostering this openness through easy to interpret monitoring and dashboarding solutions seems to be the best way to progress. This latter proposal is supported by the observation that intuitive mechanisms need to be accommodated, especially to inexperienced end-users which may be omnipresent in this generic collaboration case, to ease collaboration in inter-domain environments [16].

5.3 Enabling organizations to agree on a reward scheme

As explained in Section 5.1, a cross-organizational deployment cluster is characterized by the presence of cross-organizational data pipelines. The use of such pipelines raises questions on the reward schemes that need to be used in order to properly recognize the input brought by each of the stakeholders and incentivize partners in participating. This section highlights, using the discussed load-balancing scenario, which components are required to turn scheduling into a collaborative project, i.e. to construct a data pipeline which is the result of decisions taken by multiple entities.

5.3.1 Extending the probe rescheduling mechanism

This chapter extends a rescheduling mechanism proposed by the authors [6]. This mechanism uses probes, which are short-running software programs, to assess the technical capabilities of individual nodes and, ultimately, to determine how the availability of cluster resources is distributed. Three types of probes may be present, either generic, application-specific or copy probes. A generic probe may for example calculate the n-th digit of Pi, an application-specific probe gets selected from a well-prepared cross-organizational application catalogue, and a copy probe is a duplicate of the original application to be (re)scheduled. These probes are required to understand the heterogeneous cross-organizational cluster layout with which the operator is faced. Initially, this layout of nodes and their associated properties, are fully unknown to the operator. Local administrators have the ability to limit the resources available to external workloads, like the amount they are allowed to consume. An important reason to put this into operation is to ensure that local background loads have priority over external processes in case those are essential for business continuity. The granularity of these limiting constraints may even be on the level of individual containers. The concept of probing, i.e. executing local code fragments implemented on a compile-once-run-anywhere basis, is thus crucial for ad hoc scheduling of pods. In addition to a purely technical evaluation, it is equally important to evaluate whether there are unknown non-technical requirements, specified by the organizations individually. Organizations may wish to overrule the technical assessment of their nodes or may demand a certain level of compensation. After all, there may be a lack of mutual trust between them as organizations might have conflicting interests, compelling them to protect themselves. Consequently, it is required to allow involved organizations to negotiate both technical and non-technical requirements, which is further discussed in the next section. Figure 5.2 shows the Kubernetes scheduler as a black box, figuratively, as its operation and shortcomings are already addressed in [6]. The deviation from the default Kubernetes scheduling process marks both the beginning and the ending of the flow explained below.

5.3.2 Components required to negotiate contribution

Three phases can be identified in the proposed scheduling mechanism. Figure 5.2 shows the discussed components and steps. Note that the purple components are new and detailed in this chapter.

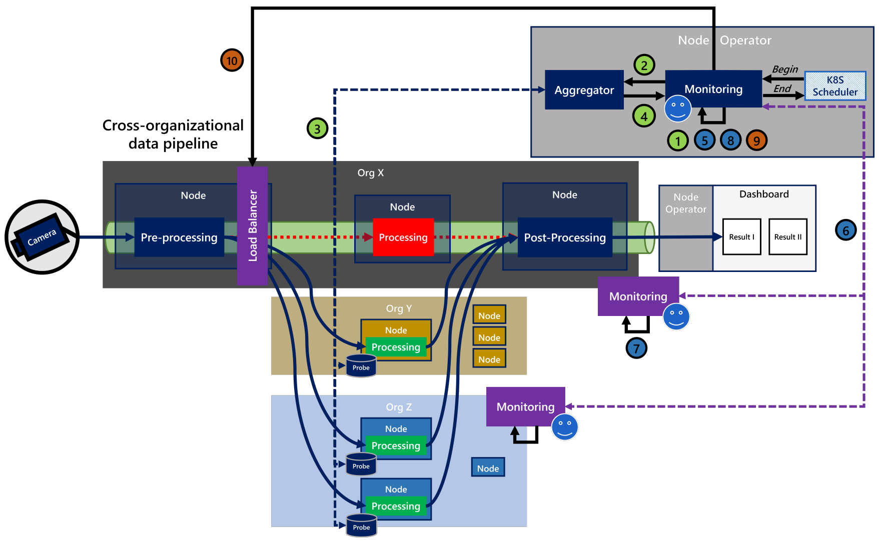


Figure 5.2: The additional software components and their operations in chronological order that can be identified during the adaptation of a cross-organizational data pipeline.

The steps belonging to the first phase are colored in green and correspond with the following actions. These four steps are identical to the ones explained in previous work [6].

1. The operator gets confronted with a scheduling assignment via its monitoring solution. This enables the operator to pick nodes, which should be probed, from the proposed selection. The selection may thus become smaller due to this manual intervention. For now it is assumed that interaction with the Kubernetes scheduler happens at the beginning and ending of the process. More details on the integration of this custom scheduling strategy can be found in [6].

2. The operator selects a suitable type of probe: either a copy probe, generic probe or application-specific probe. The deployment of these probes on the selected nodes is then initiated by the aggregator component.

3. After deployment is finished, the operator is able to manipulate the job the probe is executing. Commands can be pushed dynamically, allowing the operator to perform live probing.

4. The aggregator component is responsible for the collection of

the obtained performance measurements. Multiple probe executions need to be gathered for each of the considered nodes, to obtain a robust overview of the capabilities of the different nodes.

The steps belonging to the second phase are colored in blue and correspond with the following actions:

5. The result of the first phase is a ranking of nodes based on their probing performance. As explained in Section 5.3.1, unexpected results may be obtained. Using this ranking, the operator is then able to decide which of the probed nodes need to be selected for the current scheduling cycle. Again, this will likely be a manual assessment due to the ad hoc nature of the collaboration. At this stage, the operator may also decide that probing needs to be re-executed using an adapted set of nodes. Steps 1-4 are repeated as long as this scenario happens.

6. Based on this performance ranking, the operator is able to define an initial reward scheme. For the remainder of this chapter, it is assumed that this reward is credited on a pro rata basis, i.e. each organization receives a share in the total reward expressed in percents. What remains to be evaluated is whether there are hidden reward requirements set by the organizations involved in the specific data pipeline. The operator therefore pushes both the ranking and reward scheme to them. Three types of organizations may be distinguished based on their evaluation approach:

    - Organizations which check the reward scheme manually. These organizations have a monitoring solution available, showing all active pipelines in the cross-organizational collaboration. This dashboard could for example be based on the logging info produced by the blockchain-based mechanism proposed by the authors [4]. The negotiation of contribution is then an additional plugin within the multipurpose dashboard, which is a frontend enabling bi-directional communication with the monitoring solution of the operator. The administrator will be notified when a new proposal is received.

    - Organizations which check the proposed reward scheme automatically. The evaluation is then based on pre-defined reward boundaries.

- Organizations which do not care about rewards for their contribution. They approve the reward scheme anyway. These could be non-profit organizations like governmental institutions wishing to contribute without any restrictions, possibly due to legal regulations.

Organizations wishing to change the reward scheme, send their counter proposal to the operator. This is an important insight in this chapter, more specifically, organisations may have, for the operator unknown, requirements related to their part of the contribution to the overall pipeline. Such requirements are not necessarily known upfront, because organizations may adapt them based on the data pipeline proposal.

7. The administrator of the organization is able to change the proposal in four ways as shown in Figure 5.3:

   Increase probe ranking: the probe ranking allows each of the administrators to gain insight in the performance of the involved nodes. Based on this evaluation, an administrator could decide to change the local configuration, i.e. to grant more resources to an external container. For example, to grant more CPU and/or memory resources or to upgrade a (virtual) node. Another probe evaluation could therefore be requested by the administrator. The probes will then evaluate whether the claimed performance increase is noticeable, i.e. whether the organization is capable of providing the correct response to an unknown workload request. The response time should then be decreased, with a certain factor, compared to latest probe evaluation.

   Decrease probe ranking: the administrator could also decide, based on the probe ranking, to grant less local resources to an external container, lowering the chance of being selected in future probe evaluations. A second possibility for the administrator is to specify an upper limit on the amount of processing capacity a node has to offer, either in absolute or relative terms. This way, it is possible for the administrator to protect powerful nodes against repetitive selection, preventing a skewed distribution of workload among the partners. Clearly, such an upper limit is artificial, as it is not related to the technical assessment

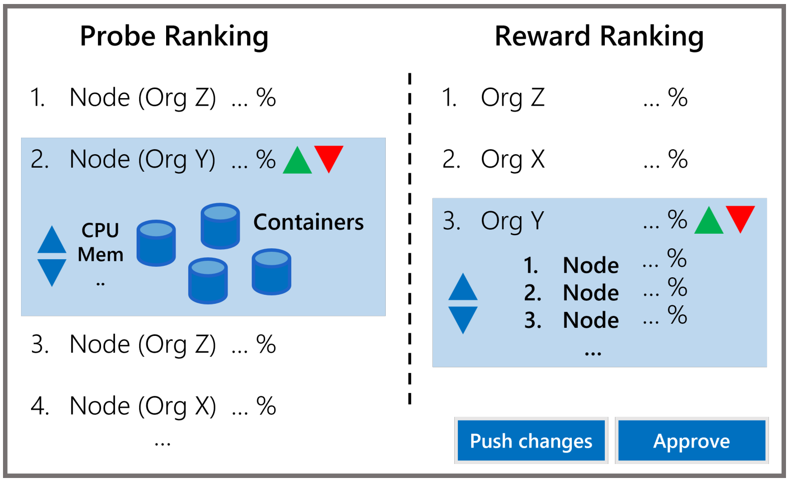


Figure 5.3: An organization, in this case Org Y, has the power to influence scheduling decisions related to a data pipeline in which it is involved.

of a node.

Increase individual reward: the reward ranking represents the recognition of the individual contributions per organization, or even split up per node. In case of a financial reward, it could represent the part of the budget each organization receives. By default, the reward ranking could be derived from the probe ranking, i.e. the workload capacity should be reimbursed proportionally. However, it could be that administrators demand a minimum reward, a lower limit, for their contribution. A data source owner, for example, could demand a certain level of reward for inserting its data into the pipeline. These hidden requirements should become clear.

Decrease individual reward: a less probable case is that of an administrator reducing the reward that is coupled to its organisation. This may happen when an organization does not prioritize the reward it receives, for whatever reason, like the organization being non-profit or possessing a huge data center with plenty of available resources. The required budget to instantiate the data pipeline, as managed by the operator, may then decrease or the freed up budget may for example be used to schedule an additional node.

These rankings thus play a significant role in the selection of the ultimate scheduling decision. It is even possible to go one step further. Local administrators may use this monitoring solution permanently to tweak requirements associated with one or more cross-organizational data pipelines over time. This step can then be considered as the entrypoint for a separate negotiation procedure, i.e. not related to any scheduling procedure. A typical cause for this would be an organization willing to tighten its requirements. There is no immediate reason to shutdown the pipeline from its side, as otherwise a more time-consuming full rescheduling would be necessary. Instead, the organization may specify a deadline by which an updated agreement honoring its requirements should be composed. Such an approach could be described as a graceful rescheduling. Contrary, fundamental pipeline changes like the variations discussed in Section 5.1, in fact result in completely new data pipelines for which a separate scheduling cycle is deemed necessary.

8. The operator evaluates the rankings in a number of rounds. Each consecutive round applies the scheduling hints sent by the organisations. This process continues until all involved organisations agree on both rankings. It may become clear that no agreement can be found or that the negotiation process takes too long, for example if a predefined maximum number of rounds is exceeded. In that case, the operator should abort the scheduling of this unschedulable configuration, adapt the set of nodes, compose new rankings, and initiate the back and forth gathering process. In a worst case scenario, no suitable scheduling result can be found after several such restarts.

The steps belonging to the third phase are colored in orange and correspond with the following actions:

9. From this point, an agreement between the organizations has been found. To secure the probe and reward ranking, the operator should put these in a distributed database spanning all organizations involved in the cross-organizational collaboration. Again, the logging mechanism cited in step six provides fundamentals to realize this. A new type of transaction should be introduced, recording the agreements for a certain data pipeline. The digital signature applied by the pipeline initiator, in this case the operator, guarantees that the reward ranking is distributed in a secure way. In addition, the digital signatures

of the involved organizations could be asked for, in case the corresponding transaction validation policy is configured this way.

10. Based on the final probe ranking, it is needed to enforce the workload division agreements that were made. A cross-organizational load balancer, being a separate pod, is therefore required. The most important property of this component is that it should be externally configurable, as the operator needs to be able to push balancing parameters. Network parameters should not necessarily be passed as the load balancer can reach the backing processing pods via their individual DNS name `processing-<ID>.processing-svc` as is feasible using a Kubernetes headless service. The `ID` is an integer ranging from 1 to N, with N being the number of service replications. The implementation of the load balancer thus has the responsibility to split up a workload according to a certain scheme, but should also allow the data to be put together in a later phase if required. In case of a video processing application, as shown here, this would mean that metadata needs to be sent along with the split up video data specifying the original camera ID and frame index or interval.

5.4 Evaluation

The possibility for each of the participating organizations to steer scheduling decisions may have a significant impact on the performance of an application. To illustrate this, it is possible to study the scenario shown in Figure 5.4. A cross-organizational collaboration of six organizations is presented. Org X has three camera streams available, each operating at 25 frames per second (FPS), and preprocesses these. The further processing of these streams needs to be executed by different organizations either for performance or analysis reasons. Org X therefore selects Org Y and Org Z as candidates for this job. These organizations provide a pool of ten nodes. Initially, it is assumed that the operator of the cluster does not have any upfront knowledge related to the capabilities of the nodes and/or requirements specified by the administrator(s) of the organizations. All these hidden properties are labeled with a purple colour in the figure. Again, this is the key distinctive element of the deployment clusters studied in this chapter.

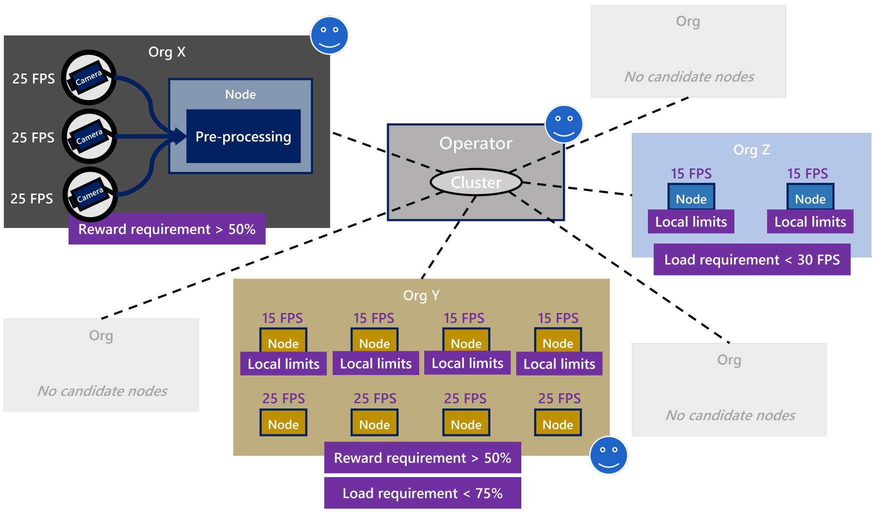


Figure 5.4: An illustrative scenario showing three out of six organizations wishing to construct multiple video processing pipelines. The purple-colored information is unknown, from the perspective of the operator, and thus needs to be uncovered during the scheduling process.

As shown, the scenario presents three custom requirements for load and reward allocation. To obtain an overview of resulting performance differences, it is possible to examine eight cases, in which each of the requirements is either turned on or off. For this case, only requirements on the level of an organization are specified, but it is equally possible for them to be specified on the level of (a set of) nodes.

- **No custom requirements**: the operator initiates a few probing rounds and finds out that a processing capacity of 75 FPS can be found. It is interesting to note that when a node is selected in Org Y, it will either be a low-capacity or high-capacity one, as Org Y has two categories of nodes available. Probing could thus provide a performance of either 15 or 25 FPS. Using the suggested probe ranking, it is possible for the administrator of Org Y to assist the operator. It could suggest the deployment of a new probing round on the same node, when its local limits have been changed, or on another node with different capacity. Such intervention, providing guidance based on scheduling hints, is especially useful in this ad hoc collaboration case, as it allows the operator to gain insight in the cluster more quickly.

- **Requirement by Org X**: when a deployment proposal is submitted to the involved stakeholders, it is possible to communicate custom reward requirements via the suggested reward ranking. Org X, in this case being the data source owner and thus playing a significant role, may want to obtain a guarantee related to its contribution. For example, it may demand more than half of the reward available for the realisation of this data pipeline. As the other organizations have no additional reward requirements, in this case, the data pipeline will be realised in the same way as in the previous one, but this time with a different reward distribution. The use of the reward ranking thus provides a certain level of trust to an organization, eventually leading to the setup of the collaboration. Note that absolute reward requirements could exist as well. In case both types exist, a total budget (per time interval) should be communicated to allow reward requests to be compared.

- **Requirement by Org Y**: Org Y specifies a load requirement, via the suggested probe ranking, on top of a reward requirement. Although it has sufficient resources available in its data center, it requires other organizations to be responsible, i.e. to pick up a part of the urgent processing job (see Figure 5.4: load requirement $< 75\%$). Note that absolute load requirements can also be specified when a set of agreed metrics exists. This is different from absolute reward requirements focusing on the amount of money solely. The reward requirement is certainly not an issue, in this case, as no other such requirements are specified. The load requirement specifies that less than three quarter of the workload should be deployed at Org Y, meaning that Org Z should contribute as well, in order to increase capacity. An exact capacity fit of 75 FPS can be found by selecting three low-capacity nodes provided by Org Y and two low-capacity nodes provided by Org Z. For this case, it would thus not be needed to consume the resources of a high-capacity node.

- **Requirement by Org Z**: similarly, Org Z is able to specify a load requirement via the suggested probe ranking. As Org Z is only able to contribute two low-capacity nodes, it limits the amount of capacity available for this specific pipeline. Only a single node is available for contribution. For this case, there is no noticeable impact, as Org Y has all requested capacity available.

- **Requirements by Org X, Y**: the reward requirements specified by Org X and Y conflict. It is impossible to grant both organizations more than half of the reward. The data provided by Org X is vital for the pipeline, meaning that its requirement should definitely be fulfilled. The result is that Org Y should be neglected in the further scheduling of workload. The only remaining solution is to pick the two nodes provided by Org Z, resulting in a capacity of only 30 FPS. Clearly, imposing strict requirements will have an impact on cross-organizational scheduling, and thus on the performance output of an application. Again, these requirements are hidden for the operator, and should be discovered in a dynamic way via the suggested probe and reward rankings.

- **Requirements by Org X, Z**: both the reward requirement by Org X and the load requirement by Org Z will be honoured, as Org Y will be solely responsible for providing the required capacity.

- **Requirements by Org Y, Z**: the additional load requirement by Org Z makes it impossible to use the previously discussed solution of five low-capacity nodes. To approximate the wished-for capacity of 75 FPS, it is therefore needed to select a high-capacity node. The maximum capacity available at Org Y is 55 FPS: one high-capacity node in combination with two low-capacity nodes. An additional low-capacity node at Org Z brings the total capacity to 70 FPS.

- **Requirements by Org X, Y, Z**: the additional load requirement by Org Z does not provide a solution to the conflicting reward requirements specified by Org X and Y. Org Y is therefore not able to provide candidate nodes. Only a single node in Org Z may be selected, due to the additional load requirement, leading to a resulting capacity of only 15 FPS.

The scenario above, which is only one from an infinite set, illustrates which kinds of requirements could pop up during scheduling and what impact they could have on the performance of an application. For this requirement configuration, two problematic cases can be identified, both being caused by the conflicting reward requirements of Org X and Y. The advantage of the use of both suggested rank-

Table 5.1: The proposed scheduling flow, allowing organizations to contribute to scheduling decisions controlled by a central operator, enables the resolution of potentially conflicting requirements.

	Org X	Org Y	Org Z
Round I	X Reward req.: > 50% ***Tentative***	Load req.: < 75% Reward req.: > 50% ***Detached***	Load req.: < 30 FPS X ***Approved (15 FPS)***
Round II	X Reward req.: > **33%** ***Approved***	Load req.: < 75% Reward req.: > 50% ***Approved (55 FPS)***	Load req.: < 30 FPS X ***Approved (15 FPS)***
Round III	X Reward req.: > 33% ***Tentative***	**X** Reward req.: > **66%** ***Approved (75 FPS)***	Load req.: < 30 FPS X ***Detached***
Round IV	X Reward req.: > **41%** ***Approved***	X Reward req.: > **58%** ***Approved (75 FPS)***	Load req.: < 30 FPS X ***Detached***

ings is that they allow for negotiation between the different partners. A problematic situation could trigger one or more of the organizations to loosen requirements. The additional insight this brings for the operator but also for all the involved organizations is the main advantage of the proposed scheduling approach. Note that instant adjustment of the requirements is only possible when they are not fixed, i.e. when they are managed by either an administrator or an algorithm. As indicated in the figure through the face icons, Org X and Y have managed requirements, while Org Z has an unmanaged requirement.

The case in which all requirements are present can be further studied given the possibility to negotiate. Table 5.1 provides an overview of a possible set of negotiation rounds which eventually causes the performance issue to be solved. The first evaluation round uncovered all custom requirements set by the organizations and a resulting capacity of only 15 FPS. While Org Y already detached due to its unfulfilled requirements and Org Z is unmanaged, Org X moved in the tentative state, i.e. it currently considers an adaptation of its requirement. During evaluation round two, Org X takes the initiative to force a breakthrough. It lowers its reward requirement to around one third, based on the idea that all three companies involved in this pipeline should receive a fair share, for example 34%, 51% and 15% for Org X, Y and Z respectively. The result is that a deployment can be found in which all requirements are fulfilled and 70 FPS can be obtained. All organizations approve this deployment option.

However, the operator is not yet satisfied with this solution, as the resulting performance is less than 75 FPS. Evaluation round three is therefore started, in which the operator proposes Org Y to loosen its load requirement in exchange for a higher reward. Org Y has enough capacity to process all frames, meaning that Org Z is not necessarily needed to enable this pipeline. The proposal for Org Y therefore suggests to drop the load requirement, given an increase in reward to around two third of the total budget. Org Y approves this proposal and its contribution of 75 FPS. However, Org X, which initially decreased its reward requirement to force a breakthrough, is not satisfied with this full shift of reward towards Org Y. It submits a counter proposal in which it takes back halve of its initial concession. Both Org X and Y approve this deployment and the negotiation process comes to an end. The approval of Org Z is not required anymore, as it is not part of the final pipeline.

5.5 Conclusion

Scheduling in a cross-organizational deployment cluster is less obvious than in a regular cluster controlled by a single organization. Each of the organizations has its own local node setup, local limits set by an administrator, interests, and ideas related to deployment. For the central operator, managing the cluster, it is required to bring these hidden requirements to the table to allow for decent scheduling decisions. As some of those requirements may even change during the procedure, it is required to evaluate them periodically. The scheduling flow presented in this chapter extends the probing solution proposed in previous work, by allowing organizations to steer scheduling decisions and communicate their intentions. This way, all organizations involved in a data pipeline have an overview of the capabilities of other nodes, allowing them to evaluate and possibly change their own contribution. Furthermore, custom requirements related to reward and load may be communicated via two separate rankings. As shown using an illustrative evaluation, it may become clear that a deployment, although nodes are technically capable to run a certain workload, is not allowed by a hosting organization due to its requirements. Negotiation between the organizations may however resolve conflicts after several rounds of interaction. In summary, the proposed scheduling flow in this chapter has two important properties: organizations are guaranteed deployment preferences, solving potential trust issues obstructing collaboration, and those preferences can be coordinated quickly, fitting the ad hoc nature of the collaborations studied. Future work should investigate how finding agreement between organizations should evolve, from being only a single step during scheduling, to an ongoing procedure which runs in parallel to the execution of a workload. This way, it is possible to obtain a sliding agreement, i.e. an agreement which consists of a series of subsequent sub-agreements over time. This approach would also allow organizations to steer their requirements based on the real-time allocation of resources and their associated costs calculated using their individual pricing model. In the end, maintaining agreement between different organizations is crucial to keep the cross-organizational collaboration alive.

Acknowledgments

The work described in this chapter, was partly funded by the FUSE research project [3], in which a Flexible federated Unified Service Environment was investigated. The project was realized in collaboration with imec. Industry project partners were Barco, Axians and e-BO Enterprises, with project support from VLAIO (Flanders Innovation & Entrepreneurship).

Bibliography

[1] “Kubernetes.” https://kubernetes.io.

[2] T. Goethals, S. Kerkhove, L. Van Hoye, M. Sebrechts, F. De Turck, and B. Volckaert, “FUSE: A Microservice Approach to Cross-domain Federation using Docker Containers,” in *Proceedings - 9th International Conference on Cloud Computing and Services Science (CLOSER)*, (Heraklion, Greece), pp. 90–99, SciTePress, 2019. https://doi.org/10.5220/0007706000900099.

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

[4] L. Van Hoye, T. Wauters, F. De Turck, and B. Volckaert, “Trustful ad hoc cross-organizational data exchanges based on the Hyperledger Fabric framework,” *Int J Network Mgmt*, vol. 30, no. 6, p. e2131, 2020. https://doi.org/10.1002/nem.2131.

[5] L. Van Hoye, T. Wauters, F. De Turck, and B. Volckaert, “A secure cross-organizational container deployment approach to enable ad hoc collaborations,” *Int J Network Mgmt*, vol. 32, no. 4, p. e2194, 2022. https://doi.org/10.1002/nem.2194.

[6] L. Van Hoye, T. Wauters, F. De Turck, and B. Volckaert, “Enabling the Rescheduling of Containerized Workloads in an Ad Hoc Cross-Organizational Collaboration,” *J Netw Syst Manage*, vol. 31, no. 1, 2023. https://doi.org/10.1007/s10922-022-09699-9.

[7] S. Mubeen, S. A. Asadollah, A. V. Papadopoulos, M. Ashjaei, H. Pei-Breivold, and M. Behnam, “Management of Service Level Agreements for Cloud Services in IoT: A Systematic Mapping Study,” *IEEE Access*, vol. 6, pp. 30184–30207, 2018. https://doi.org/10.1109/ACCESS.2017.2744677.

[8] M. Macías and J. Guitart, “SLA negotiation and enforcement policies for revenue maximization and client classification in cloud providers,” *Future Generation Computer Systems*, vol. 41, pp. 19–31, 2014. https://doi.org/10.1016/j.future.2014.03.004.

[9] B. Shojaiemehr, A. M. Rahmani, and N. N. Qader, “A three-phase process for SLA negotiation of composite cloud services,” *Computer Standards & Interfaces*, vol. 64, pp. 85–95, 2019. https://doi.org/10.1016/j.csi.2019.01.001.

[10] J. Yan, R. Kowalczyk, J. Lin, M. B. Chhetri, S. K. Goh, and J. Zhang, "Autonomous service level agreement negotiation for service composition provision," *Future Generation Computer Systems*, vol. 23, no. 6, pp. 748–759, 2007. https://doi.org/10.1016/j.future.2007.02.004.

[11] A. Omezzine, N. Bellamine Ben Saoud, S. Tazi, and G. Cooperman, "Towards a generic multilayer negotiation framework for efficient application provisioning in the cloud," *Concurrency Computat: Pract Exper*, vol. 32, no. 1, p. e4182, 2020. https://doi.org/10.1002/cpe.4182.

[12] P. Samimi, Y. Teimouri, and M. Mukhtar, "A combinatorial double auction resource allocation model in cloud computing," *Information Sciences*, vol. 357, pp. 201–216, 2016. https://doi.org/10.1016/j.ins.2014.02.008.

[13] Y. Mao, X. Xu, L. Wang, and P. Ping, "Priority Combinatorial Double Auction Based Resource Allocation in the Cloud," in *Proceedings - IEEE Sixth International Conference on Big Data Computing Service and Applications (BigDataService)*, (Oxford, UK), pp. 224–228, IEEE, 2020. https://doi.org/10.1109/BigDataService49289.2020.00043.

[14] L. Li, L. Liu, S. Huang, S. Lv, K. Lin, and S. Zhu, "Agent-based multi-tier SLA negotiation for intercloud," *J Cloud Comp*, vol. 11, no. 16, 2022. https://doi.org/10.1186/s13677-022-00286-6.

[15] L. Li, C. S. Yeo, C.-Y. Hsu, L.-C. Yu, and K. R. Lai, "Agent-based fuzzy constraint-directed negotiation for service level agreements in cloud computing," *Cluster Comput*, vol. 21, no. 2, pp. 1349–1363, 2018. https://doi.org/10.1007/s10586-017-1248-y.

[16] J. A. Wickboldt, M. Q. Guerreiro, L. Z. Granville, L. P. Gaspary, M. F. Schwarz, C. Guok, V. Chaniotakis, A. Lake, and J. MacAuley, "MEICAN: Simplifying DCN Life-Cycle Management from End-User and Operator Perspectives in Inter-Domain Environments," *IEEE Communications Magazine*, vol. 56, no. 1, pp. 179–187, 2018. https://doi.org/10.1109/MCOM.2017.1601205.

6
Conclusions and Perspectives

This concluding chapter reflects on the proposed solutions in the broader context of this dissertation and identifies interesting open challenges.

6.1 Reflecting on the Research Questions

The solutions proposed throughout the dissertation try to give an appropriate answer to the posed research questions. For each of these parts, it is however possible to present some additional thoughts that came to mind during the writing process or after publication. The research questions presented in Chapter 1 are revisited for convenience.

Research Question 1: How can potential disputes, related to the sharing of data between organizations, be prevented in case of an urgent collaboration?

This research question is answered by proposing a fully automated data exchange setup, making use of generic clients, extended with an appropriate logging mechanism. The asynchronous aspect of the logging mechanism, meaning that a data exchange does not need to wait for logs to be created fitting the ad hoc nature of the collaboration, is crucial and different levels of this asynchronous behaviour are compared. There are a few considerations worth mentioning:

- The presented contributions in Chapter 2 and Appendix A provide an in-depth step-by-step description of how the trust issue is exactly addressed, given the requirements of the case discussed and the limitations identified in the used technology. In summary, the proposed result couples a logging mechanism, asynchronously, to a data exchange process. The connection between those two conversations is crucial: it allows a participating organization to continuously assess whether the intentions of counterparties are right, i.e. whether they are prepared to audit ongoing exchanges to prevent disputes afterwards. As stated, the logging mechanism does not claim to prevent disputes by any means, but it allows an organization to quickly detect potentially malicious behavior. The solution thus enables an organization to perform an ongoing assessment of the obtained level of traceability, to pause the sharing of data with specific counterparties when issues are detected, and to resume collaboration only when remaining logging needs are fulfilled.

- Taking the viewpoint of an attacker, it is possible to analyze the attack surface. A side-channel attack is possible, where organizations part of the collaboration are involved in unauthorized sharing of data with other parties through other (non-approved) back-channels. As the sharing of data between participants is at the core of a collaboration, this is difficult to mitigate. Highly sensitive data should not be shared anyway: for example, in the case of a video stream, it would be possible to blur or hide sensitive parts of the image, at the start of the data exchange process. Furthermore, literature is available discussing how unique watermarks could be integrated to identify which entity is responsible for the leakage of data [1]. The possibility for a Sybil attack to occur is already discussed in Chapter 2. This may happen in case an attacker is able to obtain multiple ordering service node (OSN) signing identities of the deterministic ordering service. However, when this happens, a participating organization will detect malicious behavior in case its expected logs are missing and pause the sharing of data, as discussed above. Given that detectability of potentially malicious behavior is sufficient for this case, it is not necessarily an issue that, at the time of writing the chapter, no BFT consensus mechanism is provided by Hyperledger Fabric. Obviously, it then becomes easier for an attacker to violate the integrity of the system, and thus to thwart fruitful collaboration. Finally, as discussed in Chapter 2 as well, there is a non-technical risk due to the relatively low number of participants. The ultimate selection of database (chain) may be point of discussion: when multiple organizations agree on a tampered fork afterwards, the untampered version may lose its majority vote.

- The proposed logging mechanism leaves open how the actual exchanged data needs to be stored as this will differ per collaboration scenario. This can either be fully on-chain, fully off-chain and a corresponding hash on-chain, or a hybrid scenario in which only part of the data is stored on-chain. Selecting one of these options is merely a trade-off between the amount of storage a distributed database should be allowed to occupy and the level of reconstructability needed for the collaboration. This topic is shortly touched in the publication presented in Appendix A. Each approach has its advantages and disadvantages. The presented publications elaborate on the second option, using hashing, as the most generic collaboration scenario does not allow to make any assumptions related to the type of data, and to guarantee a level of reproducibility for the evaluation section. Clearly, this solution comes with a cost, as the data corresponding with the hash needs to be kept to know what was exactly exchanged. To be clear, any storage solution is compatible with the logging mechanism discussed, there is no single right solution. Finally, it is interesting to share following ideas. A hashing solution could prove to be sufficient in case compact metadata describing the exchange is stored on-chain. For example, in case a video stream is shared, it may be sufficient to know which objects were detected based on the analysis of a computer vision algorithm. Also, in case of a hybrid scenario, it may be sufficient to store I-frames on-chain, and P-frames / B-frames off-chain, if the latter is required.

- As a potential trust issue regarding storage could be identified in this ad hoc collaboration case, it seemed logical to explore state of art blockchain solutions. After a while, it became clear that the set of available concepts, which are all branded under the same blockchain label, had really diverse use cases. The topic seems to cause confusion and even controversy. Clearly, there is a huge difference between permissioned and permissionless blockchains. The former could be seen as an evolution of distributed databases, while the latter brings revolution due to the newly proposed consensus mechanisms. Strong proponents of permissionless blockchains do not consider permissioned ones to be valid blockchains. Although this basically is a discussion about terminology, it may never hinder innovative concepts to improve the resilience of distributed database against malicious entities. Due to the limited number of participants in the case discussed here, the set of writers will always be known and does not need to change dynamically over time, and therefore the complexity of the consensus mechanisms brought by permissioned blockchains seems to be sufficient. The conclusion drawn from all experiences, including a scientific conference on this topic, is that one should carefully match wished-for properties against assumptions exhibited by each solution.

- It is hard to find a comprehensive evaluation case for ad hoc collaborations in general, 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, the setup of the Kafka cluster, etc. all depend on a specific situation at hand. The parameter set that is currently used and documented, focuses on a short-term collaboration between a limited set of at most ten organizations. This is a typical case, as the proposed application is intended for small scale urgent cross-organizational collaborations. The main purpose of the experiments is to show that a proof-of-concept works and to compare the situation where logging is turned on to the situation where logging is turned off. Finally, the Markov chain allows to play with different collaboration scenarios quickly.

- Chapter 2 compares the purpose of the chain data structure in the proof-of-work consensus mechanism as used by Bitcoin and the purpose of the chain as used here. Although the internals of Bitcoin are only slightly touched, it is interesting to share a technical report by Ozisik and Levine [2], which was read at a later point in time. It discusses the analysis of a successful double-spend attack as presented in the original Bitcoin paper. The detailed derivations of the equations to calculate this probability of attacker success are shown. Of particular interest are the conflicting assumptions regarding the budget of the attacker in this analysis, as pointed out by the authors. They aptly summarize this as follows: "*It's like saying you have infinite money for gas for your car, but can't spend any of those funds on a faster car, even though faster cars are available.*"

- The data exchange mechanism currently focuses on the typical request / response interaction scheme. Extending this solution to support eventing or pushing of data is a possibility for future work. In both cases, a slightly adapted pattern consisting of the identified logging functions will be obtained, which basically means that new implementations making use of the same fundamentals need to be foreseen.

Research Question 2: How can the deployment of containers, proposed by a potentially malicious external entity, be verified by the hosting organization?

The answer to this research question consists of the integration of the authorization protocol UMA 2.0, of which an implementation is brought by Keycloak, into the container orchestrator Kubernetes. This way, it is possible for one or more local administrators to verify deployments asynchronously. There are a few considerations worth mentioning:

- The verification of external deployments would not be needed in case interactions between organizations were limited to the consumption of APIs in another domain and the processing of tasks by means of containers deployed in own domain. This could be considered as the default scenario covering most use cases. However, there may be motivations to deploy software remotely. From a network point of view, there may be bandwidth restrictions between the participants or latency issues due to the encryption of significant volumes of data in transit as executed by protocols like TLS. Furthermore, confidentiality may be an issue, both for data from local sources and for algorithms (e.g. a patented computer vision algorithm). Last but not least, as inexperienced end-users could be part of an ad hoc collaboration, it is needed to deploy external software anyway.

- The aim of the presented solution is to bring together technology available in the state of art: (I) Kubernetes, as a widely used container orchestrator and (II) binary / image / container verification techniques targeted at finding exploits and/or vulnerabilities like suspicious network patterns, indications of compromise, etc. The main question answered in this contribution is thus how to integrate verification techniques, which are already out there and new techniques which will be researched in future work, into Kubernetes in a transparent way. By no means safety is claimed through the suggested solution. Having a way to implement strong access control through UMA and possibly policy-based authorization is in fact only the beginning of an entire verification pipeline through which external deployments need to pass. Having a way to verify external deployments is considered key here, even to prevent against benign organizations going rogue, regardless of what the exact verification steps look like.

- Although a prototype showed that this architecture allows to realize verified deployments, it is worth mentioning a trade-off. The presented solution requires three additional components to be deployed in the domains of each of the participating organizations: a custom Kubelet, a resource server and an authorization server. The reason for this is a lack of trust, as the organizations only trust themselves. Clearly, this level of supervision comes at the cost of a more decentralized setup which is highly likely to take more time to prepare and manage (e.g. to prevent the authorization component against attacks like denial-of-service at layer 3 or 7, brute force credentials guessing, etc). Furthermore, the replication of these components will also have an impact on the overall cost associated with the collaboration. Outsourcing authentication and/or authorization is an alternative for each hosting organization, but thus means that a trusted party needs to be found. Chapter 3 considers a generic collaboration and therefore does not make any assumption regarding the availability of a (partially) trusted third party. The conclusion is that the presented architecture is not the single correct choice of deployment and that a practical setup depends on the collaboration at hand. Each of the organizations should thus be able to express its trust preferences at the start of a collaboration, for example by selecting a trust category from a pre-compiled list. This is further discussed later on.

Research Question 3: How can the scheduling of containers, a task for which the cluster operator is responsible, be fitted to an unknown heterogeneous cluster environment?

The provided answer to this research question consists of the integration of a probe swarm constituting the so called probe swarm architecture. There are a few considerations worth mentioning:

- Usually, papers discussing the scheduling topic present algorithms, in fact heuristics, and compare them with existing ones based on evaluation scenarios. The focus of Chapter 4 is not necessarily on the selection of the algorithm, but rather on the input that needs to be considered in case of an ad hoc cross-organizational collaboration. As the cluster layout is way more unknown and dynamic compared to a regular cluster setup, it is needed to actively gather accurate input data through the usage of probes. Clearly, the performance of any algorithm depends on the quality of the input data, thus this is a crucial aspect to lower the chance of costly scheduling mistakes. The processing of this input data may be done in different ways. Well-known heuristics could definitely be applied to this data, but likely in a supportive role. Due to the ad hoc and dynamic nature of the collaboration, it seems that a mix of manual and automatic decisions is inevitably present to steer the scheduling process. Furthermore, the goal is not to squeeze node performance, as is the case for scheduling in data centers. The conclusion is that the presented and motivated probe swarm architecture matches these criteria.

- The proposed contribution does not try to replace existing distributed monitoring frameworks like Prometheus or in-band telemetry setups. As explained in Chapter 4, the assessment of metrics or node properties is definitely possible, but may exhibit a few complications. It may be difficult to quickly compare gathered data, for example diverging hardware characteristics. Furthermore, these frameworks need to be configured, which requires a level of preparation amongst the participants. Measurements need to be collected continuously, which allocates bandwidth and may leak internal data, and they can easily be forged by participants. Therefore, an additional, likely complementary, solution is needed allowing rapid evaluation of nodes and their different potential in light of the collaboration at hand. By posing a challenge, it is possible to discover the capabilities and (artificial) restrictions of nodes over time. Clearly, significant performance differences are mainly targeted here, for example to identify restricted edge devices.

- Regarding the possibility to exploit the probing concept, two risks can be identified. First, an attacker could figure out when exactly a node is probed and at that time maliciously adapt the software behavior to consume at least more resources than actually needed for the purpose of the application. Although this attack is possible, external containers will likely be made subject to performance limits by hosting organizations. Bursts are therefore able to happen, but their effect on competing processes will only be limited. Second, an attacker could trigger continuous reschedulings negatively impacting the scheduling progress in case no countermeasures are taken. Although this issue is less prevalent in this context, as the rescheduling procedure would only be triggered in case of a (severe) performance issue, it is needed to incorporate a possible countermeasure. An appropriate solution would be to introduce a cooling down period, i.e. when a node appeared to underperform, it cannot be part of the set of candidate nodes evaluated during several consecutive scheduling rounds. Multiple performance issues with a node could then cause this period to increase progressively, thus preventing the risk of a (re)scheduling denial-of-service attack.

Research Question 4: How can the scheduling of containers, a task for which the cluster operator is responsible and which is fitted to an unknown heterogeneous cluster environment, be extended to allow for negotiation between the participating organizations?

The answer to this question is mainly driven by the introduction of a load and reward ranking which allow to deal with highly dynamic collaborations. There are a few considerations worth mentioning:

- The main aspect to consider when answering this question is how the negotiation between participating organizations should be organized. The requirements identified in Chapter 5, a set which could possibly be extended, need to be communicated. It is possible to define a negotiation protocol detailing the fixed steps organizations should take to come to agreement. It is however questionable whether this would match the flexibility of the case discussed here. Organizations will likely change requirements frequently, for example to react to requirements specified by others, or to try different sets of requirements and wait for the reactions of the partners. Furthermore, organizations could join and leave the collaboration at any point in time, causing the negotiation protocol to be adapted and possibly organizations to be informed. The presented solution takes an approach focused on freedom. Organizations have rankings, presented visually via a monitoring solution, at their disposal which allow them to specify requirements at any point in the scheduling process. For sure, this would allow a scheduling process to continue indefinitely. Due to the urgency of the collaboration, scheduling is time-critical, meaning that if the process would take longer than a pre-defined upper limit, it would be needed for an operator to gradually decrease the level of freedom, for example by specifying a maximum number of remaining editing rounds. When the deadline has passed and no agreement has been produced, it is needed to restart the scheduling procedure. The conclusion is that the chance for a collaboration to succeed should be maximized, and that in case of urgency, top-down rules can be imposed to force a final decision.

- As negotiation may remain ongoing after initial scheduling and deployment, it is required to explore techniques which are capable to cope with this behavior. As already mentioned in Chapter 5, graceful rescheduling is one such technique. In addition to do this, it would be interesting to investigate to which extent it is possible to suspend and resume jobs (e.g. as allowed by Kubernetes) during this negotiation procedure. Allowing workloads to be executed dynamically would also enable organizations to negotiate priorities and to transform to a scheduling procedure based on the agreed importance of tasks.

- An important reason why the concept of the load and reward ranking is proposed, is that organizations may not be familiar with the configuration of fully automated negotiating agents typically used in SLA negotiation. In this case, it should trust another entity for configuring the agent properly, which may not be desirable. Ideally, automated negotiation would still be the main driver behind the negotiation process, but the configuration by the local organization should be made more easily accessible. Although this suggestion is a moonshot, an interesting concept was recently proposed by Meta AI through the Cicero agent [3], which combines strategic reasoning and natural language processing to understand and negotiate the intentions of human players and to find shared objectives. Although targeted at a specific board game, there clearly is a similar trend, in this case in the domain of AI, to achieve more natural interaction between humans and agents. It would be interesting to see how this idea could be transferred to the ad hoc collaboration case as discussed here. This could definitely be a major step forward in the negotiation of SLAs.

6.2 Future Perspectives

To conclude this dissertation, it is possible to identify interesting directions for future work. In general, the cross-organizational aspect of the setup, introducing the discussed potential trust issues, allows many aspects of a collaboration environment to be checked against these new circumstances. The challenges these bring on the level of networking are just one of the many not yet discussed elements. To keep things focused, only directions are discussed below which elaborate on the earlier discussed work.

Supporting a multi-cluster environment

Each of the discussed solutions assumes the existence of a single federated orchestration cluster being operated by a cluster operator. The idea is that organizations can join the collaboration at any time by contributing worker nodes. It is however possible that an organization is willing to contribute a significant amount of nodes. As connecting a node to the federated cluster requires quite some configuration [4], for example to enable networking and to guarantee no changes are left on the host, it seems illogical to replicate this overhead when nodes are already part of an orchestration environment. Future work should therefore investigate how organizations could contribute an entire cluster of nodes under its supervision, either located on-premise or in a cloud. Projects like Open Cluster Management [5] and Cluster API [6] should definitely be examined in this regard. Although this setup simplifies orchestrator deployment, it complicates the management of containers. For example, a multitude of schedulers becomes present, a situation which may significantly change the roles of the operator and the local administrators. As the potential trust issues can still be identified in this new environment, it appears that all proposed methodologies are still relevant.

Researching the edge

As assumed until now, and as is also clear from previous paragraph, organizations contribute worker nodes and only these are further analyzed. Implicitly, this means that those nodes are capable of running components needed to participate in the orchestrator environment. Clearly, an organization should have at least one node in a preferably stable environment, to ensure a gateway is permanently available to communicate with the cluster. The actual data sources however, that are of real interest, may be much deeper down in the network on edge computing nodes. Such nodes are typically placed in sub-optimal network environments, for example in a production plant offering a diverse set of network connection technologies (e.g. WiFi, LoRa, 3G/4G). Consuming their data may bring additional challenges in terms of reliability and availability. Furthermore, the microservices connecting them to the top-level nodes should be resilient against this heterogeneous and resource constrained environment. Fault tolerance of these services should thus be examined. Finally, it should be evaluated in which cases it is interesting to enable edge nodes to join the collaboration straight away, for example through the usage of lightweight Kubernetes solutions [7].

Expressing trust relationships

A key characteristic of the collaborations discussed throughout this dissertation is a lack of full trust between the participating organizations. As discussed earlier, this lack of trust needs to be remediated by introducing additional mechanisms, and configuring associated software components costs time and money. Until now, a binary approach was followed, either there is full trust or no full trust. It is however possible to take a more granular approach regarding trust. Organizations should be able to express trust relationships. A subset of the collaborating entities may for example have a common trusted third party (preferred partner), or an organization may have a trusted third party responsible for authentication, or business units of the same organization may fully trust each other. These are all semi-trust examples in which trust conditions can be loosened and deployment be simplified. Clarifying these exceptions should happen

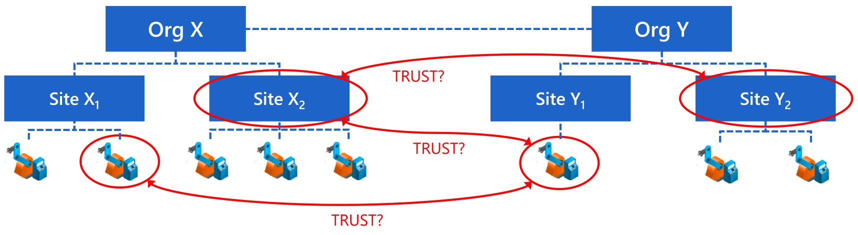


Figure 6.1: A more fine-grained evaluation of trust relationships.

at the start of a collaboration, but changing them should also be allowed when the collaboration is ongoing. This way, it is possible to define a cross-organizational collaboration taking into account trust tailored to a specific case. Future work should investigate adequate solutions to enable this.

Shifting trust and negotiation mechanisms downwards

The presented contributions focus on trust at the level of organizations. However, it would be possible to consider more fine-grained trust assessments by shifting the proposed solutions towards lower levels in the hierarchy if they are present. For example, as illustrated by Figure 6.1, it is possible to evaluate trust relationships between different sites or business units of organizations. Equally, it would be possible to do this at the level of edge devices. The advantage this would bring is that (temporary) trust violations, which can potentially be caused by relatively small issues, do not cause organizations to suspend their entire collaboration. A trade-off to be made is that management complexity will increase, as multiple trust enabling software components need to be duplicated. Furthermore, future work should investigate research consumption of these components, as this will become important once devices with constrained resources are going to play a role in the proposed mechanisms.

Bibliography

[1] H. Mareen, *Pirates of the film industry : the curse of the forensic watermark*. PhD thesis, Ghent University, 2021. https://bit.ly/PhDThesisHannes.

[2] A. P. Ozisik and B. N. Levine, "An Explanation of Nakamoto's Analysis of Double-spend Attacks," tech. rep., 2017. https://doi.org/10.48550/arXiv.1701.03977.

[3] "Meta AI presents Cicero," 2022. https://ai.facebook.com/research/cicero.

[4] T. Goethals, S. Kerkhove, L. Van Hoye, M. Sebrechts, F. De Turck, and B. Volckaert, "FUSE: A Microservice Approach to Cross-domain Federation using Docker Containers," in *Proceedings - 9th International Conference on Cloud Computing and Services Science (CLOSER)*, (Heraklion, Greece), pp. 90–99, SciTePress, 2019. https://doi.org/10.5220/0007706000900099.

[5] "Open Cluster Management." https://open-cluster-management.io.

[6] "Cluster API." https://github.com/kubernetes-sigs/cluster-api.

[7] "K3s: Lightweight Kubernetes." https://k3s.io.

A

Logging mechanism for cross-organizational collaborations using Hyperledger Fabric

⋆ ⋆ ⋆

L. Van Hoye, P-J. Maenhaut, T. Wauters, B. Volckaert and F. De Turck

Published in IEEE International Conference on Blockchain and Cryptocurrency (ICBC), March 2019.

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 appendix 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.

A.1 Introduction

A.1.1 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 A.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.

A.1.2 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:

- Make it impossible for an organization to deny (the integrity of)

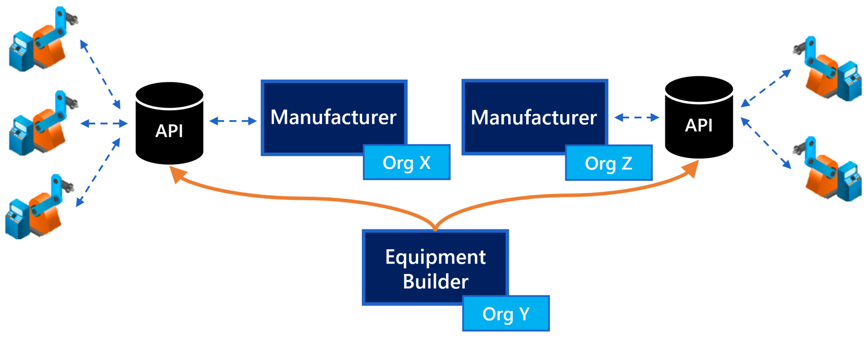


Figure A.1: Sample collaboration scenario showcasing an equipment builder gaining access to internal APIs of machines it installed at different manufacturers.

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 A.3.1, 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 appendix. 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.

A.2 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 A.2 which is commonly used in literature:

1. Data needs to be stored in a structured way, introducing the need for a database.

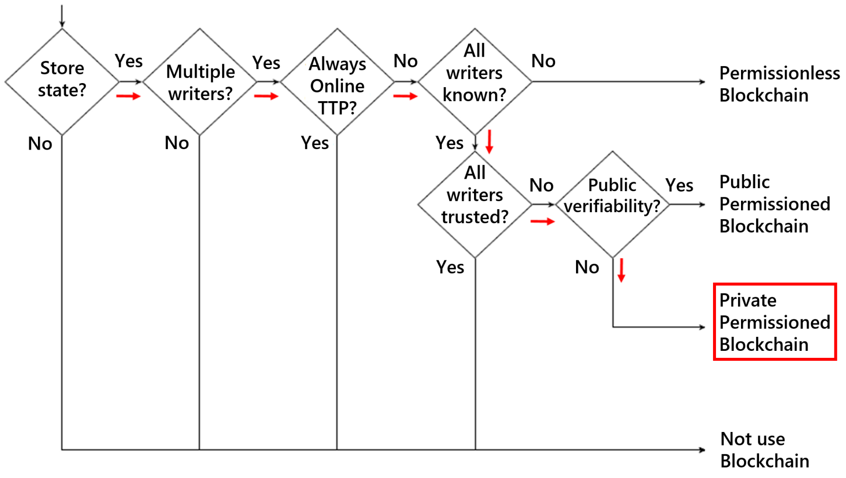


Figure A.2: Which architectural blockchain model is most appropriate for an application? [2]

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.

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 appendix, 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 A.3.1, 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 appendix, 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 appendix. 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 appendix. 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 multi-party 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].

A.3 Logging mechanism

A.3.1 Design decision

The baseline architecture to start from consists of multiple client-server relationships. This situation is shown in Figure A.3, whereby three organizations want to share APIs among them. Two conceptual channels are defined, i.e. putting content in the common ledger

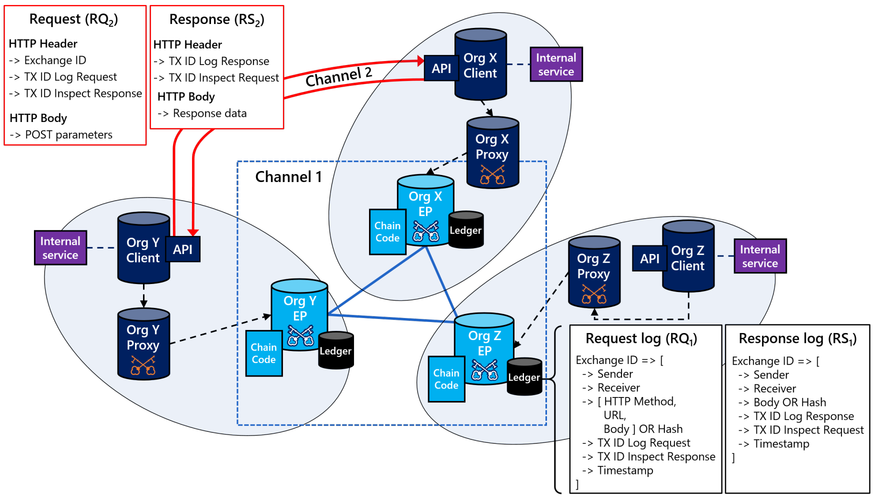


Figure A.3: Components needed for the proposed logging mechanism.

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 appendix. Although it is less secure as strategy one, it achieves the goals mentioned in Section A.1.2.

A.3.2 Architecture of logging mechanism

Figure A.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 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 A.1.2, 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 A.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 A.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 A.2, an asynchronous flow is proposed in this appendix. 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 A.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 A.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 A.2.

The 'Fabric' lifeline is further detailed in Figure A.5, showing the

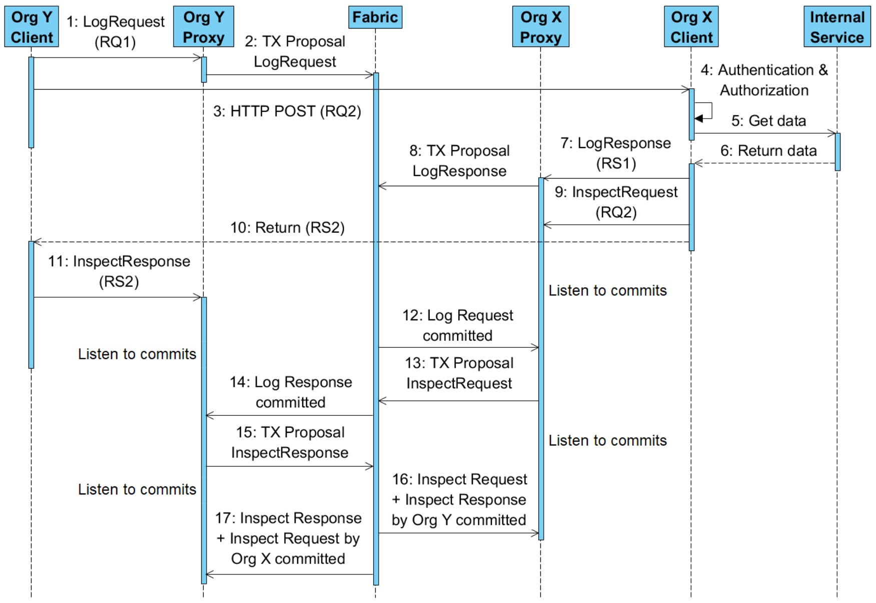

Org Y Client
Org Y Proxy
Fabric
Org X Proxy
Org X Client
Internal Service
1: LogRequest (RQ1)
2: TX Proposal LogRequest
3: HTTP POST (RQ2)
4: Authentication & Authorization
5: Get data
6: Return data
7: LogResponse (RS1)
8: TX Proposal LogResponse
9: InspectRequest (RQ2)
10: Return (RS2)
11: InspectResponse (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

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

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 A.2. 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

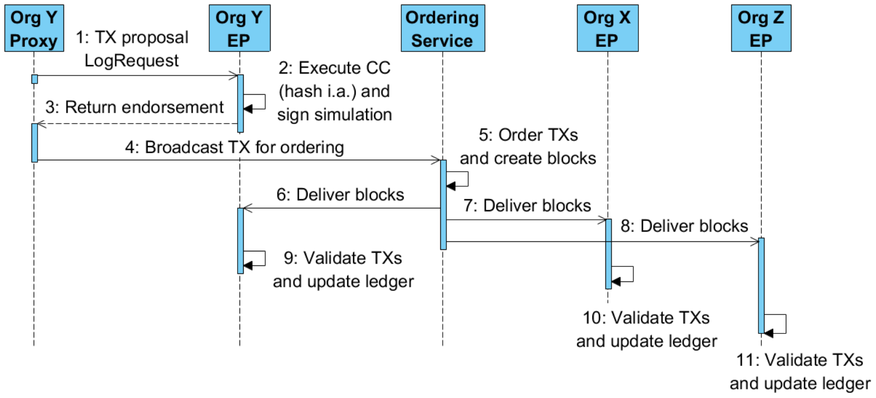

Org Y Proxy
Org Y EP
Ordering Service
Org X EP
Org Z EP
1: TX proposal LogRequest
2: Execute CC (hash i.a.) and sign simulation
3: Return endorsement
4: Broadcast TX for ordering
5: Order TXs and create blocks
6: Deliver blocks
7: Deliver blocks
8: Deliver blocks
9: Validate TXs and update ledger
10: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

Figure A.5: Sequence diagram showing the integration of Hyperledger Fabric.

never rewrite history of an honest organization.

A.4 Performance evaluation

A.4.1 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 A.6 shows an overview of the containers that 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.

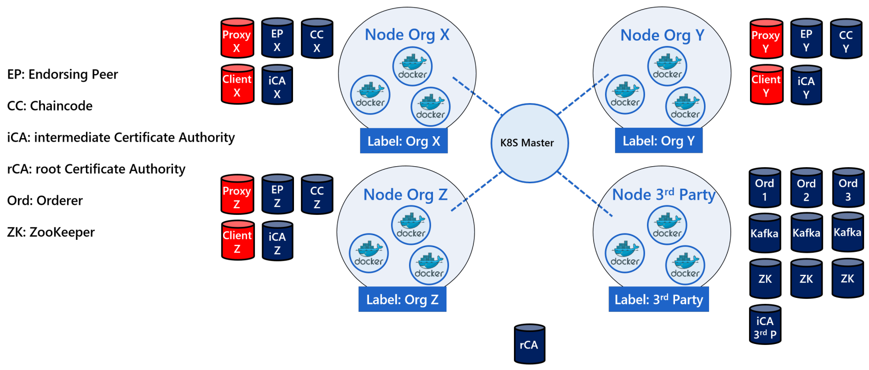


Figure A.6: An overview of the evaluation setup when the Kafka ordering service is used.

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 A.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.

A.4.2 Measurements

The same example collaboration shown in Figure A.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 per second E and size of the data S. The following parameter setting is determined for the use case examined in this appendix:

- L is set to 15ms as explained above.
- BS is limited to 512 KiB. The maximum size of an individ-

ual 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.

Figure A.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

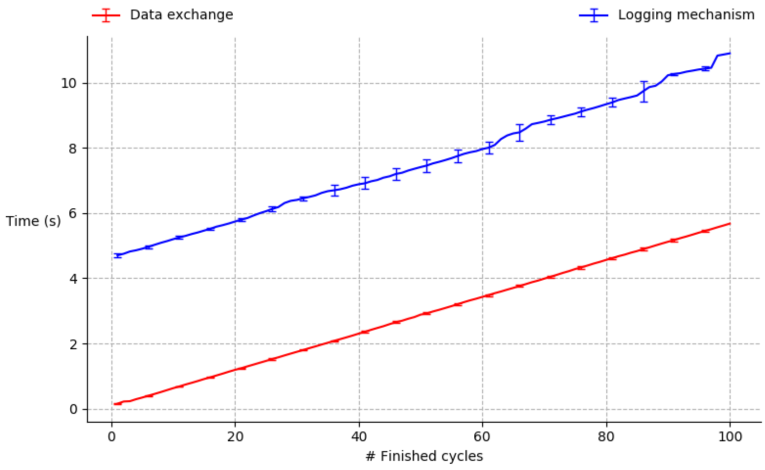


Figure A.7: The asynchronous approach has an average throughput of 17.6 data exchanges per second. The logging mechanism is able to keep up with the speed of the data exchange process.

Table A.1: 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

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 A.8 presents the obtained results. The experiment is repeated five times for each value of O and the corresponding average values

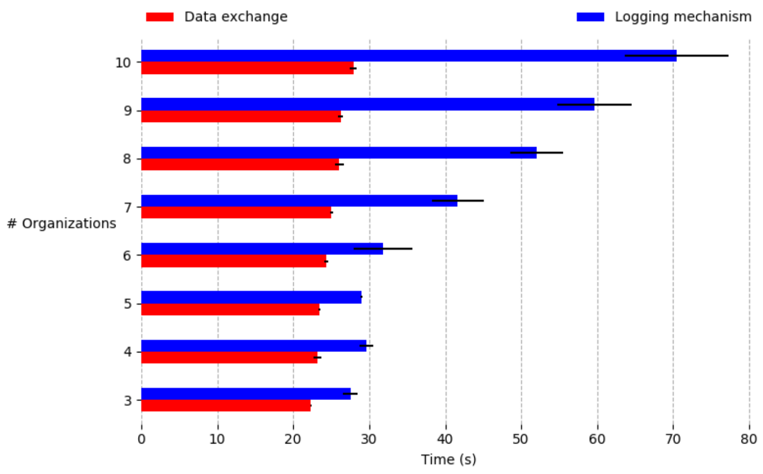


Figure A.8: The decoupling of the data exchange and logging mechanism processes emerges when the number of organizations is scaled.

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 A.1 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 A.1 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.

A.5 Conclusions and future work

In this appendix, 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.

Acknowledgments

This appendix 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).

Bibliography

[1] “Hyperledger Fabric.” https://www.hyperledger.org/projects/fabric.

[2] K. Wüst and A. Gervais, “Do you need a blockchain?,” in *Proceedings - 2018 Crypto Valley Conference on Blockchain Technology, CVCBT 2018*, pp. 45–54, IEEE, 2018. https://doi.org/10.1109/CVCBT.2018.00011.

[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*, pp. 1–15, Association for Computing Machinery, 2018. https://doi.org/10.1145/3190508.3190538.

[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*, pp. 51–58, IEEE, 2018. https://doi.org/10.1109/DSN.2018.00018.

[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*, pp. 355–362, IEEE, 2014. https://doi.org/10.1109/DSN.2014.43.

[6] “Peers.” 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)*, pp. 614–621, IEEE, 2018. https://doi.org/10.1109/CLOUD.2018.00085.

[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), pp. 401–408, IEEE, 2018. https://doi.org/10.1109/CLOUD.2018.00057.

[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*, pp. 699–706, IEEE, 2018. https://doi.org/10.1109/DSD.2018.00008.

[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*, pp. 1–5, IEEE, 2018. https://doi.org/10.1109/NOMS.2018.8406324.

[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*, pp. 1–6, IEEE, 2018. https://doi.org/10.1109/NOMS.2018.8406310.

[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)*, pp. 25–32, IEEE, 2018. https://doi.org/10.1109/CLOUD.2018.00011.

[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*, pp. 385–391, IEEE, 2018. https://doi.org/10.1109/IC2E.2018.00073.

[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*, pp. 85–92, IEEE, 2017. https://doi.org/10.1109/SERA.2017.7965711.

[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)*, pp. 310–317, IEEE, 2018. https://doi.org/10.1109/CLOUD.2018.00046.

[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)*, pp. 1489–1494, IEEE, 2018. https://doi.org/10.1109/ICDE.2018.00167.

[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*, pp. 357–363, IEEE, 2018. https://doi.org/10.1109/IC2E.2018.00069.

[18] M. Brandenburger, C. Cachin, R. Kapitza, and A. Sorniotti, "Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric," tech. rep., IBM Research, 2018. http://arxiv.org/abs/1805.08541.

[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. https://doi.org/10.1109/MCOM.2006.1678121.

[20] "Transaction Flow." https://hyperledger-fabric.readthedocs.io/en/release-1.3/txflow.html.

[21] "Docker security." https://docs.docker.com/engine/security/security.

[22] "Ledger." https://hyperledger-fabric.readthedocs.io/en/release-1.3/ledger.html.

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

[24] "LevelDB." https://github.com/google/leveldb.

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

[26] "CloudPing.info." https://www.cloudping.info/.

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