Journal of Data Intelligence, Vol. 2, No. 3 (2021) 301–325

A PENALTY-AWARE CLOUD MONITORING SYSTEM BASED ON BLOCKCHAINS

CHRISTIAN DIENBAUER
Faculty of Computer Science, University of Vienna
Waehringerstrasse 29, Vienna, Austria
a1026759@univie.ac.at

BENEDIKT PITTL
Faculty of Computer Science, University of Vienna
Waehringerstrasse 29, Vienna, Austria
benedikt.pittl@univie.ac.at

ERICH SCHIKUTA
Faculty of Computer Science, University of Vienna
Waehringerstrasse 29, Vienna, Austria
erich.schikuta@univie.ac.at

Today, traded cloud services are described by service level agreements that specify the obligations of providers such as availability or reliability. Violations of service level agreements lead to penalty payments. The recent development of prominent cloud platforms such as the re-design of Amazon's spot marketspace underpins a trend towards dynamic cloud markets where consumers migrate their services continuously to different marketspaces and providers to reach a cost-optimum. This leads to a heterogeneous IT infrastructure and consequently aggravates the monitoring of the delivered service quality. Hence, there is a need for a transparent penalty management system, which ensures that consumers automatically get penalty payments from providers in case of service violations.

In the paper at hand, we present a cloud monitoring system that is able to execute penalty payments autonomously. In this regard, we apply smart contracts hosted on blockchains, which continuously monitor cloud services and trigger penalty payments to consumers in case of service violations. For justification and evaluation we implement our approach by the IBM Hyperledger Fabric framework and create a use case with Amazon's cloud services as well as Azures cloud services to illustrate the universal design of the presented mechanism.

Keywords: Cloud Comuting, Smart Contract, Service Monitoring, SLA Management

1. Introduction

Cloud providers such as Amazon and Azure sell their datacenter resources in form of services. For the description of services, so-called Service Level Agreements (SLAs) are used [1]. They are specifications that define, inter alia, details about the offered service such as availability and reliability. Violations of SLAs can lead to penalty payments. Thus, consumers are interested in monitoring the service performance. Cloud providers such as Amazons offer monitoring platforms like Amazon Cloudwatch that provide APIs to query the performance metrics of the service. Besides, several third-party cloud monitoring platforms are existing,

such as dynatrace[a], datadog[b]or AppDynamics[c]. The vision of a monitoring platform that also manages penalty payments is still unrivaled [2]. For the establishment of such a platform in industry not only machine-readable SLAs such as described in [2] are necessary, but also the transparent management of penalty payments has to be ensured. Hence, in the last years, neutral third parties were introduced that are responsible for confirming service violations and transferring penalty payments to consumers [3, 4]. Even such neutral third party approaches have structural weaknesses as Robert Sams summarizes with three sins [5]: sin of commission, sin of deletion and sin of omission. Nowadays, the scientific community reverts to neutral consensus finding approaches among untrusted participants which are technically realized by blockchain technology. For example, the authors of [6] introduced an approach for managing and modifying SLA terms with blockchains. However, penalty management for SLA violations was neglected.

The paper at hand focuses on the development of a blockchain-based, penalty-aware cloud monitoring platform. Thereby we apply so-called smart contracts, which are programs executed on the blockchain. A first step towards such systems was introduced in [7], where a pure provider-centric smart contract was introduced. In contrast to that, the approach introduced in our paper treats a smart contract as a bilateral agreement between consumers and providers. Therefore, consumers and providers have to register their SLAs of traded cloud services in a smart contract, including corresponding commonly trusted cloud monitoring authorities. The smart contract uses them to monitor the performance of the cloud services and *automatically* transfers penalties to the consumer in case of SLA violations. This ensures that consumers get penalties immediately for service violations, while providers can profit from a clear and transparent monitoring approach. The technical feasibility of our approach is demonstrated by a smart contract deployed on the Hyperledger Fabric Blockchain[d] that uses arbitrary monitoring services for detecting SLA violations and consequently executing penalty payments. Within this paper, we use cloud services as well as monitoring APIs both from Amazon and from Azure.

The remainder of the paper is structured as follows: First, we focus on foundations and related work. Then, we present the concept of the penalty-aware cloud monitoring platform, which is followed by an introduction of our implementation using the IBM Hyperledger Fabric Blockchain for Amazon EC2 and Azure. A discussion about the findings of the presented concept and future work is given before we close the paper with a summary and conclusion.

2. Foundations and Related Work

This section is structured into two parts. The first part describes foundations of the blockchain technology and the second part presents related cloud monitoring approaches.

With the rising popularity of the Bitcoin in mid-2017, the underlying blockchain technology gained attraction. This technology is not limited to cryptocurrencies and so organizations from various domains started to identify reliable business models enabled by the blockchain [8]. A typology of emerging blockchain applications with regards to the domains where they are

[a]https://www.dynatrace.com
[b]https://www.datadoghq.com/
[c]https://www.appdynamics.com/
[d]https://www.ibm.com/blockchain/hyperledger

applied is presented in [9, 10]. Due to the decentralized nature with no need for intermediaries, the blockchain technology can handle transactions in different markets and consequently reduces friction costs [11, 8]. The Economist article *The Trust Machine* considers the blockchain technology as a solution for establishing trust and its antecedents, as confidence, integrity, reliability, responsibility and predictability [8]. While sometimes blockchains are described as *trust-less* [12], other authors [13] state that there is a shift of trust from current intermediaries to the technical implementation of blockchain systems. Limitations of the blockchain such as scalability, energy usage, and growing complexity are summarized in [14]. While data once stored within a blockchain is immutable, it does not mean that this data is always valid. Therefore, current research investigates in incentive-based blockchains to provide data quality [12]. When a blockchain needs to query data from the *outside world*, it uses so-called *oracles*. However, the capabilities of oracles are sometimes limited: For example, Ethereum[e] prohibits that smart contracts can query external sources, and so the data needs to be stored within a smart contract [15] first. The smart contract with the injected data can then be used as oracle and queried by other smart contracts. Hyperledger Fabric, as a contrary example, uses general-purpose programming languages and allows that data can be retrieved from anywhere. Important hereby is that, no matter by whom this data is queried and regardless of time, the results need to be the same to ensure a deterministic behavior of the blockchain.

Today, most of the monitoring platforms are cloud provider-specific limiting the monitoring of heterogeneous IT infrastructures, as found in smart city architectures and industry 4.0 [16]. This enables data-driven applications that require reliable communication models to operate more efficiently [17]. Complex cloud service compositions are currently hard to monitor by existing monitoring platforms. In [18] a generic framework to monitor the performance of services deployed on different providers is presented. To collect data from the services, the framework proposes a *client-server approach*, where agents are deployed together with the service that should be monitored: Those agents can be accessed by a centralized monitoring platform to retrieve data of monitored services. A similar approach in the context of edge computing is presented in [16]. There, an architecture divided into management and worker layer is presented. The management layer provides functionalities to measure the activities of the edge nodes in the network to distribute workloads based on different algorithms.

An approach for monitoring data streams in distributed systems with the focus on communication efficiency and data privacy can be found in [19]. The system collects information about different components and sets the granularity based on a given threshold by a centralized monitoring unit. Such centralized monitoring systems suffer from a single point of failure. Therefore, a distributed agent-based monitoring system to handle multi-tenant service-based systems is used [20]. A challenge of service monitoring is the system overhead of monitoring tools that the collection of data can be done in a resource-effective manner [21]. The authors of [7] introduce a concept of a blockchain based cloud monitoring system without a concrete implementation. Thereby, the provider creates a smart contract that manages a list of consumers. In case of a service violation *service coins* are transferred to consumers. Motivated by the vision of billing consumers with smart contracts, the role of the consumer is neglected: it neither has to acknowledge the smart contract nor does it acknowledge the used monitoring

[e] https://ethereum.org/

services.

To foster the quality of services and the reliability of business processes that are based on these, Service Level Agreements (SLAs) between a service provider and consumer are defined [22]. As SLAs are legally binding contracts, they have to be audited by an instance that is trusted by all participants. While conventional SLA trust models are centrally configured, deployed and maintained, new approaches to define SLAs using smart contracts can be found in [23, 24, 25, 26, 22, 27]. A conceptual blockchain-based framework for SLA management is presented in [23]. It gives a general overview of this topic and addresses the challenge of trust between all participants arguing that no single party should have control over the SLA lifecycle. In traditional SLA approaches service consumers need to detect abnormalities and check SLA violations, which leads to an extensive manual process including personnel and capital resources [23, 25]. An approach for automated compensation of SLAs based on smart contracts can be found in [24]. It compares its approach to existing solutions with regards to human-computer interaction.

A formal description of a SLA lifecycle based on a smart contract is presented in [26] that includes 5 steps: 1. Discovery of service and Negotiation of SLAs, 2. Deployment of the SLA using smart contracts, 3. Monitoring of the service, 4. Billing and Penalty Enforcement, and 5. Termination of the smart contract. A system architecture in the context of fog computing can be found in [22] that provides IoT client devices with the means to explore the best-suited fog nodes via smart contracts by considering the reputation and credibility based on SLA requirements.

Current approaches try to address today's requirements of heterogeneous and decentralized infrastructures by agent-based systems with either a centralized unit or a wholly distributed approach. They identify the challenges of data privacy and communication efficiency. These approaches are usually isolated systems as the data is subject to a single entity with lack of transparency. Thus, for organizations it is hard to get reliable information about the current and past state of their IT infrastructure [28]. Consequently, consumers are forced to implement systems to gather information This results in additional costs and the burden of proofing the lack of quality [23, 25].

While approaches exist to define Service Level Agreements using smart contracts without the need to rely on a single party and automate the way to identify SLA violations, they are either conceptual or lack the implementation of a penalty mechanism. Approaches mentioned in [23, 24] use Ethereum as the underlying blockchain technology but miss the part on how to inject service metrics into smart contracts to trigger events while preserving a deterministic behavior of the blockchain. Therefore, we focused on the implementation of a monitoring system for a service provider where SLA can be defined. Compared to existing approaches we implemented a mechanism to define SLAs and trigger penalties in case of their violation.

Thus, for our research endeavor we pursue the following requirements:

1. It is mandatory that cloud services from multiple service provider can be monitored.
2. The monitoring tool needs to be distributed with no single point of failure.
3. Data within the system will be treated confidentially.
4. There is no need to trust in a single participant and transparency of the system.

5. Implementation uses effective communication models for scalability.

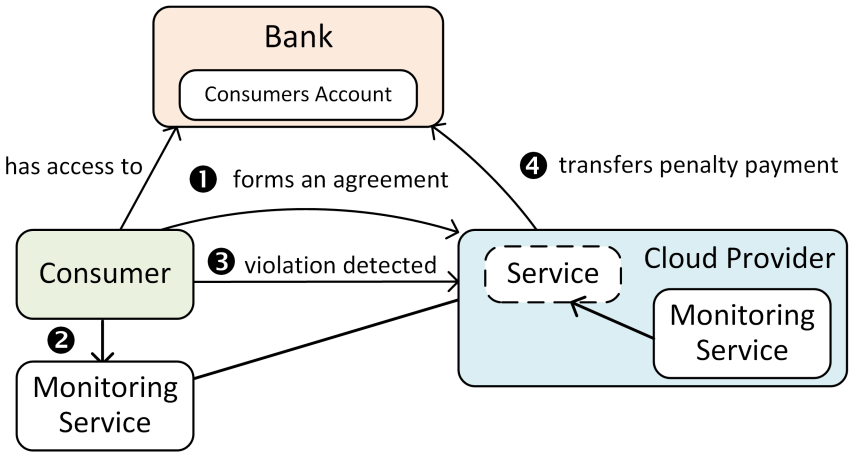

Fig. 1. Traditional cloud monitoring scenario

3. Cloud Monitoring with Smart Contract

A traditional cloud monitoring scenario is presented in Figure 1. First, the consumer purchases a service from a provider (1) then both, consumer and provider use services to monitor the performance of traded service (2). As soon as the consumer observes a service violation, it claims a penalty payment from the provider (3). Finally, the provider transfers the penalty payments to the consumers' bank (4). The traditional monitoring scenario comes with two drawbacks: First, the identification of a service violation can lead to contradictory results as both, the consumer and the provider could use separate monitoring services. Second, the scenario requires a trusted third party such as the bank which is responsible for transferring penalty payments.

To tackle these issue,s the paper introduces a cloud monitoring system based on blockchains. It uses smart contracts that are responsible for processing data from monitoring services and executing penalty payments in case of identified cloud service violations. The following two components are essential for such a smart contract:

- **Monitoring Service.**

  A smart contract cannot monitor the cloud service directly. Therefore, it has to make use of a monitoring service that returns performance metrics. Such services are offered by cloud providers such as Amazon,[f] but also by independent monitoring service providers such as datadoghq[g]. Both, the consumer and the provider have to agree on the selected monitoring service. Even multiple monitoring services could be used in parallel, whereby the smart contract has to decide which value is used in the case of contradictory results. In the rest of the paper, the monitoring services are explicitly referred to as *monitoring services* to distinguish them from the traded cloud service, which is termed *service*.

[f] https://aws.amazon.com/de/cloudwatch/
[g] https://www.datadoghq.com

- **Wallet.**

  The smart contract is a neutral entity that is trusted by both, consumer and provider. To guarantee transparent and fast penalty payments, the smart contract needs a wallet. Instead of directly transferring the service fee to the provider, the consumer has to pay it by transferring tokens[h] to the smart contract. Those tokens are stored in the wallet of the smart contract. In the case of service violations, tokens are transferred back to the consumer. The remaining tokens are transferred to the provider after the contract period expired.

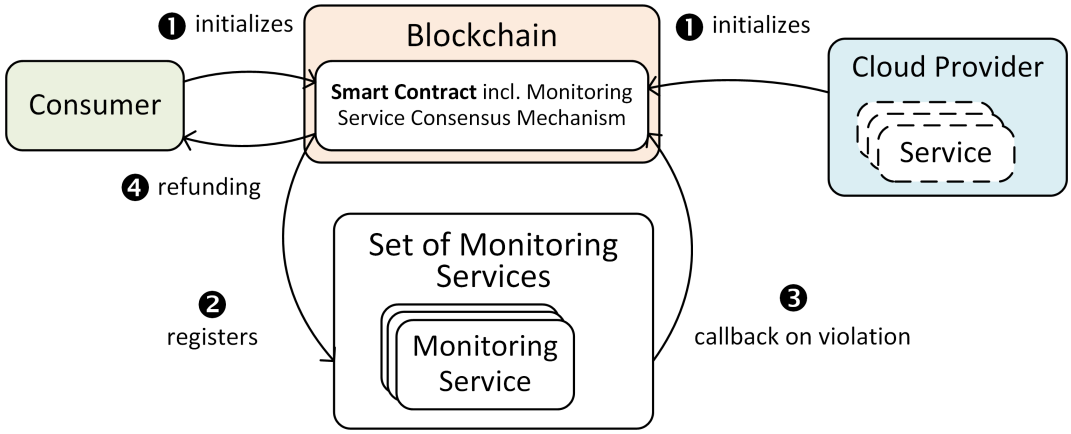

Fig. 2. Blockchain-based cloud monitoring scenario

Figure 2 introduces an exemplary scenario in which a *cloud provider* hosts the service used by the *consumer*. The numbers in the following enumeration refer to the numbers in the figure.

1. A consumer decided to use a cloud service from a provider for a certain period of time. For the autonomous penalty management, the consumer and the provider create a smart contract. The provider adds endpoints of the used monitoring services to it. They are used by the smart contract for monitoring the service performance. The consumer agrees to use the service by transferring the service fee in form of tokens to the smart contract. They are stored in the smart contract's wallet.

2. The consumer uses the service. The service is monitored by the monitoring services defined in the smart contract.

3. Service violations detected by the monitoring service are accessible for the smart contract. If multiple monitoring services are used, the smart contract has to resolve contradictory results to determine if a service violation occurred. This conflict resolution is part of the smart contract, so both the consumer and the provider are aware of it.

[h]Tokens can be money but also other trade-able entities.

4. If the smart contract detects a service violation, it automatically transfers penalty payments from its wallet to the consumer. After the contract period between consumers and providers expired, the remaining tokens in the wallet of the smart contract are transferred to the provider. If no service violations occur, all tokens representing the service fee are transferred to the provider.

The introduced approach has several benefits over approaches where non-blockchain based software solutions are employed for penalty management. (i) First of all, smart contracts are deployed and executed on the blockchain and consequently on a neutral platform while typical software solutions have to be hosted on a server. There, the availability, as well as the execution, are not guaranteed. (ii) Further, smart contracts are participants of the blockchains, which means that they can send and receive tokens such as money. For the realization of non-blockchain based software, a trusted third party such as a bank would be necessary. (iii) Trusted third parties charge fees and require time for executing payments. On the blockchain, these fees for trusted third parties are not relevant. The consumer gets penalty payments automatically and immediately on detected service violations. (iv) Smart contracts are immutable, which means that the code cannot be modified after it is deployed and so the approach is appropriate for untrusted consumers and providers.

The consumers have to transfer the service fee upfront to the smart contract. This implies that the consumer has to pre-pay the complete service fee while the provider gets the remaining service fee at the end of the contract period. Especially for high service fees or long contract periods, this might cause financial embarrassment. In such cases, the consumer and the provider could agree to transfer only parts of the payments to the smart contract.

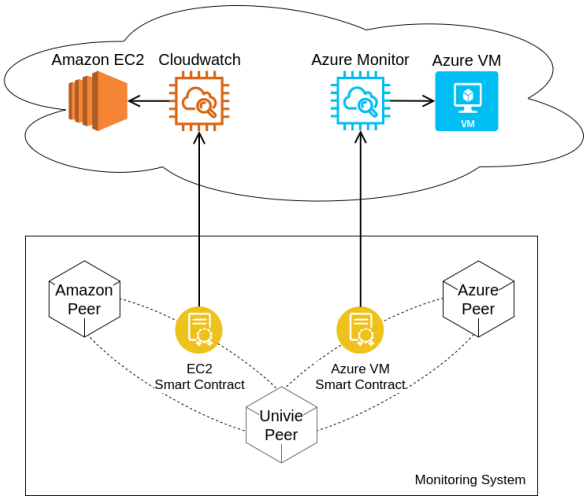

Fig. 3. Scenario of the cloud monitoring system

3.1. *Forming an Agreement*

We assume that the deployment of the Smart Contract takes place during the subscription process for a service. Customers lookup the desired service via the product catalog of a

service provider and choose available subscription options. In our example of subscribing to a virtual machine, customers can select the number of CPUs, the available RAM, or the available network bandwidth as well as metrics for the availability. The service properties and selected options are then used to shape the characteristics of the smart contract to enforce SLA violations. Important hereby is that metrics stated in an SLA, need to be requestable by a smart contract using the monitoring tools of the service provider as an oracle.

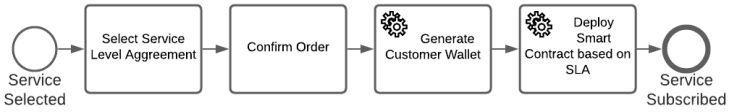

Fig. 4. Defining SLAs in a Smart Contract

After the customer accepted the Service Level Agreements, a wallet holding the cryptographic key-pair will be generated for the customer. The smart contract with the defined SLAs and information about the wallet addresses of the provider and the client will be deployed to the blockchain network (see Figure 4). All blockchain-related steps are performed automatically in the background without any interaction needed by the customer. At the end of the subscription process, the customer gets a summary of the deployed smart contract and information on how to access the wallet. Executing the contract verifies that the service operates as stated in the SLAs. If it fails to deliver the characteristics as defined during the subscription process, a penalty payment is transferred to the wallet of the customer. This corresponds to step two, three, and four of the SLA lifecycle described in [26].

3.2. *Retrieving Data Off-Chain*

Sometimes blockchain-based systems need to obtain information outside the boundaries of the blockchain network. This is necessary, if volatile information, e.g. data from the stock market or metrics from a web service, are used as a basis for decisions. While data once stored in the blockchain is immutable and therefore considered as trustworthy, using data from the *outer-world* is a critical process to preserve trust and transparency. For example, an approach could mitigate the strength of a blockchain, where the client application sends information as a payload to a smart contract. No other participant of the network could be able to verify if the information a user committed is correct.

This challenge is addressed by so-called *oracles*. An oracle is a data source queried by smart contracts influencing their action. Depending on the underlying blockchain technology an oracle can either be:

1. Smart contract itself, or
2. Trusted third party service

Important for oracles is their deterministic behavior. Thus, regardless of time, the participants of the network and who is querying the smart contract, the oracle will always deliver the same result.

In the case of Ethereum, an oracle is a smart contract, as Solidity, the language used to implement smart contracts, has no concept to query external services. So off-chain data needs to put into a smart contract first, which can afterward be queried by other smart contracts. The correctness of the data put into the smart contract makes it reproduce- and traceable. The smart contract that acts as oracle can implement some logic, that the input of multiple parties will be aggregated. For example, if an escrow service is implemented, an oracle will consider the input of the buyer, the seller, and a trusted third party. If the seller and buyer do not agree, the input of a trusted third party will be required to either send the funds to the seller or give them back to the buyer.

Other blockchain technologies, e.g. Hyperledger, use general-purpose programming languages like JavaScript, Python, or Go to request services. In this case, an oracle can be a trusted third party service that is queried in runtime. The logic is implemented in a smart contract on which the participants of a network previously agreed. Thereby, it is important to preserve the deterministic behavior of the blockchain again. In figure 5 the concept is shown, where multiple organizations execute the same smart contract to verify that the data stored in the blockchain is indeed the data from the trusted third party.

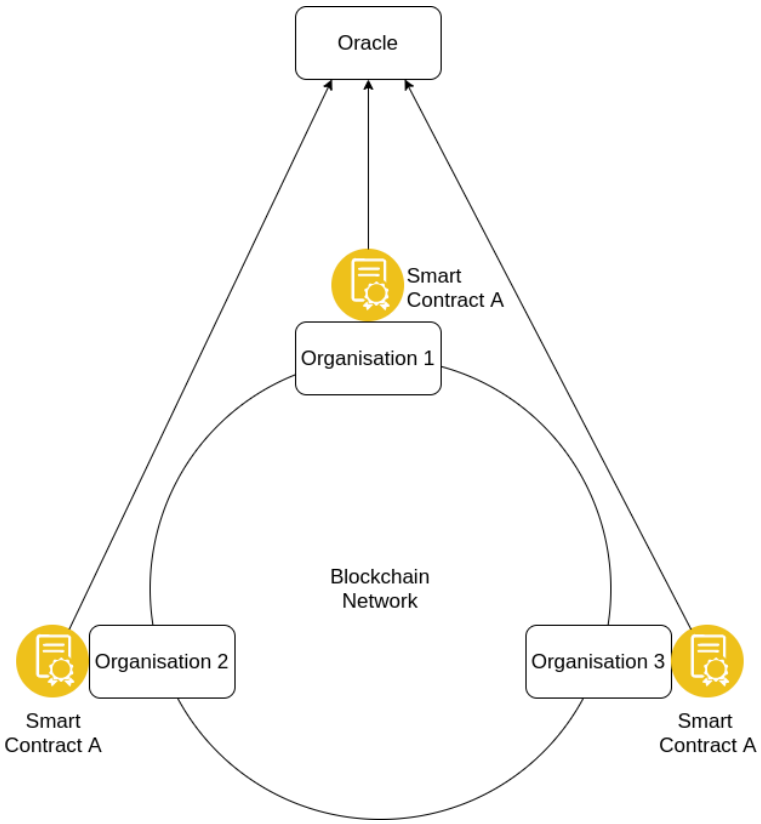

Fig. 5. Participants validate data

4. Implementation

This section introduces the implementation of the scenario shown in Figure 3. It distinguishes between three stakeholders: two cloud service providers and one consumer. In the scenario, one service provider is Amazon, the other service provider is Microsoft Azure. The consumer uses a virtual machine, which represents the traded cloud service from both providers. For each cloud service, a smart contract is instantiated. It has access to monitoring services hosted by the corresponding provider respectively. Those monitoring services act as oracles for the smart contract and allow for a deterministic behavior of the states in the blockchain.

Different blockchain technologies can be used for the implementation of the scenario. For the selection of appropriate blockchain technology implementing the envisioned penalty-aware cloud monitoring platform, we follow the blockchain-selection guideline [29]. As the participants' identities will be established before purchasing a service and data of services is not publicly verified, we decided to use a private permissioned blockchain. A private permissioned blockchain will guarantee that service data will only be accessible for the users specified in an endorsement policy: the service provider and the consumer. Transaction fee for processing data is not relevant for private blockchains. For the presented use-case we used the Hyperledger Fabric framework[i], which supports the usage of a general-purpose programming language for writing smart contracts, chaincode as they call them, to implement any conceivable logic.

Figure 6 shows the technical architecture of the implemented scenario. It consists of the smart contracts, monitoring service integration code and the core environment.

Peer Nodes, or short peers, are a fundamental part of a blockchain network and inherit a cryptographic identity given by a certificate authority. With this identity, peers can be authenticated in the network and, with regards to the endorsement policy, join certain channels. A channel is a sub-net of the blockchain network and enables private communication between peers. Members of the channel share a distributed ledger to store confidential transactions. Smart contracts are deployed on channels and can be executed by its peers to interact with the ledger. In our example, the service consumer shares a separate ledger with each provider. Each participant deploys a graphical user interface and an API via a docker-compose file in its environment. Communication with the participants of the network is only done via smart contracts.

- **Smart Contract.**

  For the implementation of the smart contracts we decided to use JavaScript, as Hyperledger Fabric officially supports an SDK for *Node.js*. If a consumer consumes multiple services from a provider, one smart contract is sufficient. It can handle multiple services of the same type by linking the measurements to a single instance via a unique service key. In the code snippet below an excerpt of a smart contract is shown for retrieving measurements. The credentials are stored in the ledger and will be used for querying the Amazon EC2 Oracle.

```
async updateService(ctx, serviceKey, startTime, endTime) {
    const serviceAsBytes = await ctx.stub.getState(serviceKey);
```

[i] https://www.hyperledger.org/use/fabric

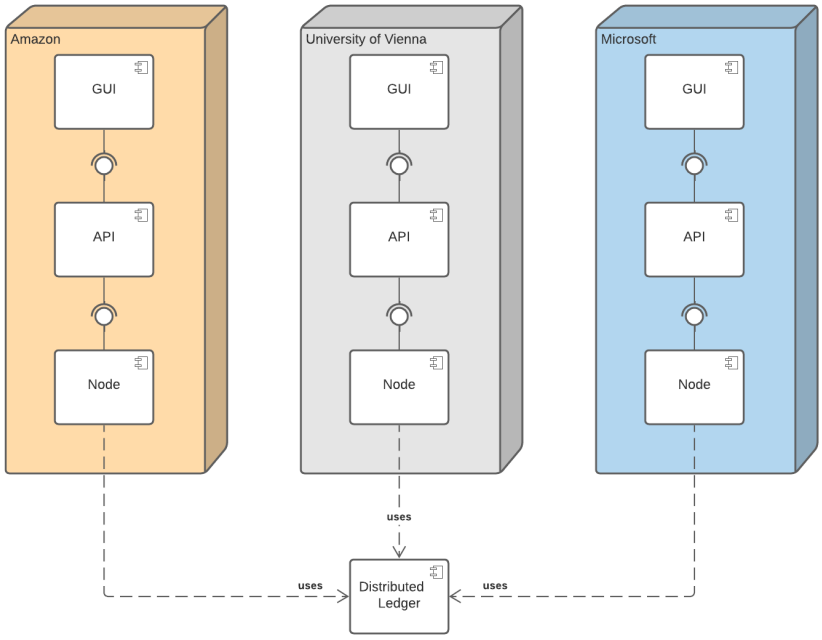

Fig. 6. Components of the cloud monitoring system

```
if (!serviceAsBytes || serviceAsBytes.length === 0) {
    throw new Error(`${serviceKey} does not exist`);
}
let service = JSON.parse(serviceAsBytes.toString());
...
// query the oracle
service.data = await this.getMetricsDataEC2(
    service.accessKeyId,
    service.secretAccessKey,
    service.region,
    service.dimensionName,
    service.dimensionValue,
    startTime,
    endTime
);
// calculate availability
if (service.cpuUtilization >= 0) {
    service.timesAvailable += 1;
    console.log("Connection good");
} else {
    service.timesNotAvailable += 1;
    console.log("No connection");
```

```
}
...
// give credit if availability is to low
if (currentAvailability < service.promisedAvailability) {
    service.credit += PENALTY_AMOUNT;
}
await ctx.stub.putState(
    service.name,
    Buffer.from(JSON.stringify(service)));
```

- **Monitoring Service Integration.**

  Smart contracts require access to the monitoring services. In blockchain-environments, accessing such external services is technically challenging: The *Node.js* SDK provided by Amazon is used to retrieve metrics of the service from their monitoring tool Cloudwatch. When a service is registered to the monitoring system, an unique service key with all the necessary information to retrieve metrics from Cloudwatch is stored within the ledger. This enables retrieving information from Cloudwatch and storing the information into the ledger running purely inside a smart contract by just passing the service key for identification.

  An update function is used to retrieve the information about the service metrics of the latest time interval. During registration of a service in our system, the availability promised by the service provider is passed as a parameter and stored together with the credentials and information of the service in the ledger. As the update function queries the information about the service from Cloudwatch, we see from the response if a service is available or not. Within the smart contract we mark this requested interval as either available or unavailable and compare it to the total run-time of the service. If a service's downtime exceeds the promised availability of a service provider, the customer will receive a certain amount of credit for this service. This credit can then be used as an offset during settlement. As smart contracts are not self-executable programs, the update function needs to be requested from outside of the network. To prevent a user from requesting the service's status for the same time interval twice, smart contracts check at each request of the update function if the time of the last state in the ledger will not overlap with the time interval in the request.

  The same logic is used for the implementation of the smart contract for Azure VM services. However, instead of using an SDK to retrieve information from the Azure monitoring tool, we decided to use their provided REST API.

- **Core Environment.**

  To provide users of the system a convenient way to register services and explore their current and past state together with actual availability and possible credits, we created a graphical user interface with React. As React is based on JavaScript and the Hyplerdeger also provides a JavaScript SDK for application to interact with smart contracts deployed in the network, our first attempt was to use this SDK within the React

application. Unfortunately this was not possible as the version we used of React is based on the ES6 JavaScript specification, whereby the SDK for Fabric 1.4 follows the specification of ES5. This mismatch of versions of the specifications lead to import errors and crashes of the React application. Thus, we developed a REST API which uses the SDK to interact with the blockchain network and provides endpoints that can be requested by the React application. This API has as further advantage that it can be used by a cron job to automatically query the latest states of a registered service.

A reference to the complete script to reproduce the created setup can be found on GitHub[j]. The main components of the script are:

1. Generate x.509 certificates,
2. Configuration of the channels,
3. Creation of the peer nodes,
4. Joining the peers to the channels, and
5. Instantiating the smart contracts in the channels

The current implementation of the monitoring service is capable of retrieving information about two service types of two different service providers. To prove the feasibility of integrating service level agreements we started with the implementation of the *promised availability* that is passed to the system during registration process. Though, as this logic is deployed via smart contracts, the system can be extended in a modular manner to support additional service types as well, as a variety of service metrics that are provided by the oracles. All this information can then be used to define SLAs via the monitoring system. If an SLA agreement is violated, the current approach will trigger a penalty by increasing the credit of variables within the ledger that is shared between the service provider and the consumer. To change the value of this variable, a predefined logic within the smart contract is used that considers all the past states of service as well as the promised availability passed during registration of the service in the system. As smart contracts are not self-executable programs, the process of retrieving new states of a service needs to be started by the user. Currently, this is done via a button in the user interface but can be automated using a cron job. For this purpose, we created a RESt API as it was necessary to implement the JavaScript SDK to interact with the smart contracts. As first attempt we used the SDK within the React application, which led to importing errors, as the JavaScript standard used by React (ES6) did not match the standard used by the SDK (ES5). The code listing shows the endpoint of the API that is used by the frontend to query the smart contract to retrieve new states of service. The user interface is deployed locally without any direct connection to the internet. All the information that is shown is retrieved from the blockchain network by using smart contracts. The code snippet shows how new services are registered to the system. Our scenario uses services from Amazon and Microsoft and can choose between two registration forms. This form contains information for authentication to the oracles as well as information about the SLA. If the

[j] https://github.com/dieni/blockchain-based-cloud-monitoring

entered information is correct, the credentials will be stored in a smart contract. On the bottom of the form, a list of all registered services of a provider can be seen. When clicking on the services, the details page of the service 8 opens: Here a user can see information about the retrieved measurements of the service. By clicking on the button, new measurements can be retrieved. If the service is more often unreachable than promised, the customer will receive a credit.

```
app.put('/aws/services/:serviceKey', async (req, res) => {
  console.log(req.body)
  let serviceKey = JSON.parse(req.params.serviceKey)

  const gateway = new Gateway()

  try {
    console.log('CONNECT TO NETWORK')
    await gateway.connect(connectionProfile, connectionOptions)
    const network = await gateway.getNetwork('channel-aws')
    const contract = network.getContract('ec2contract')

    const endTime = new Date()
    const startTime = new Date(endTime.getTime() - 30 * 60000)

    await contract.submitTransaction(
      'updateService',
      serviceKey,
      startTime.toISOString(),
      endTime.toISOString()
    )

    console.log('SERVICE UPDATED')
  } finally {
    // Disconnect from the gateway
    console.log('Disconnect from Fabric gateway.')
    gateway.disconnect()
  }
})
```

In the appendix we documented the system setup as well as the available services that we implemented.

5. Findings and Future Work

This paper is part of a research project aiming on using blockchain technology for monitoring services of different service providers. Thereby, we aim at a decentralized architecture to enable penalty payments if service level agreements are violated. This concept is realized by

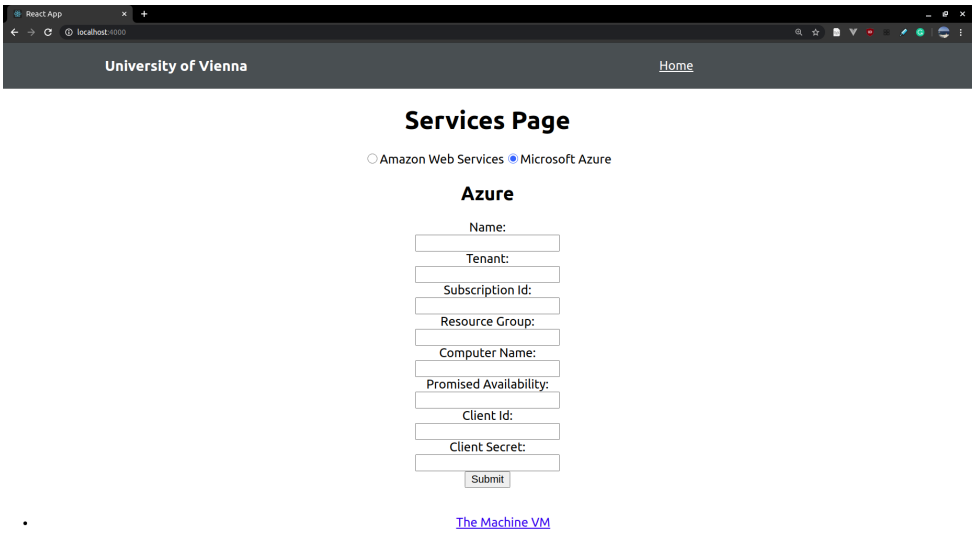

Fig. 7. Registration page of a cloud service that should be monitored by the smart contract deployed on the Hyperledger Fabric Network

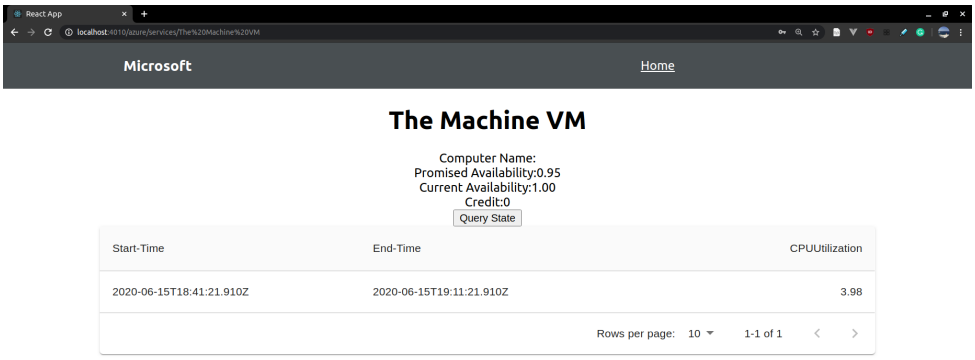

Fig. 8. Service details page that show the current status of the monitored service - the same information is retrieved by the smart contract

the presented implementation, where for each service type a smart contract is deployed in a self hosted Hyplerledger Fabric network and shared between service provider and consumer via private channels. Even more, the design of the implemented approach will work for any other provider as long as an API is available that returns the required service performance metrics.

The following taxative list sums up the findings of the presented work and gives impetus for future work:

- **Blockchain as a Service**

  The network was created using the CLI tools provided by Hyperledger and was deployed locally running the components within docker containers. Hyperledger provides the containers via docker hub for all necessary components to run the network. Though, the configuration of the network can be quite difficult as documentation about different settings and operations is quite rare. A recommended approach for a future project is to use Blockchain as a service. Amazon [k] as well as Microsoft [l] provide managed blockchains services to create Hyperledger Fabric networks as well as Ethereum. They claim that this can be done with minimal time for configuration. Another platform to be considered for using Hyperledger Fabric as a service is from IBM [m].

- **Smart Contracts.**

  With the use of a general-purpose programming language, the development of smart contracts is nothing out of the ordinary. Hyperledger Fabric supports a chaincode SDK for JavaScript that enables a convenient interaction with the ledger using a so-called *transaction context*. Other languages are supported too. Packages can be used without any restrictions. One thing to keep in mind when developing smart contracts is, that each operation needs to follow a deterministic behavior. This is necessary that nodes of the network reach consensus and the states in the ledger are reproducible.

- **Oracles.**

  For the deterministic behavior of the system, we used the monitoring tools of the service provider as our oracles. Amazon as well as Microsoft provides a sophisticated API to query current and past states of services based on different metrics. For AWS we used a JavaScript SDK and for Azure a Rest API. Both methods work like expected and can be used to query information in granularity of one minute. The execution of a request will be done by a single node and the results distributed through the network. To prevent the owner of a node making changes to the request, the oracle might sign the reply with its private key so that other nodes can verify the validity of the data.

- **Multiple service types.**

  We decided to deploy one smart contract between a service provider and consumer for each service type. This enables a consumer to monitor multiple services from the

[k] https://aws.amazon.com/marketplace/pp/B0797GK9YY
[l] https://azuremarketplace.microsoft.com/en-us/marketplace/apps/microsoft-azure-blockchain.azure-blockchain-hyperledger-fabric-aks-based
[m] https://www.ibm.com/blockchain

same provider using a single smart contract. It has to be investigated how to enable the monitoring of multiple services from different types between a service provider and consumer. One approach could use one smart contract for each additional service type that has to be monitored. However, This results in a very static design and a large number of smart contracts when supporting several service providers each delivering a variety of services. A generic approach of registering different service types is worthwhile to be investigated, to limit the number of smart contracts and to have a more flexible way of on-boarding new service providers with there services.

- **Service Level Agreements.**

  For this project, we used a single metric to determine the availability of the service and compared it to the promised availability. With each request to an oracle, the state of the service was compared with all past events. As smart contracts are not self-executable, we had to care that requests are not made twice for the same time interval. We solved this by checking the last time interval stored in the ledger before retrieving new measurements from the oracle. For more sophisticated SLA's we will need to consider more metrics and let the users define the appropriate levels for those. The best case would be to combine this task with a generic approach to registering services.

- **Penalty Payments.**

  As already said, if there is a violation of the SLA's, a predefined amount is credited to the consumer. This is currently done via a variable stored in the ledger that can only be modified based on the retrieved metrics from the oracles. While the credit is defined when creating the invoice, we aim for an independent process of transferring funds between the users. One approach is to define a penalty token that will be transferred between the users, if SLA's are violated. Therefore we need to replace the logic of increasing the variable with the mechanism to trigger the transfer. The tokens need to be transferred between wallets dedicated to the users to give them the possibility to have them at one's disposal later.

- **Tokens.**

  Usually, in blockchain networks tokens Bitcoins are transferred instead of money. They represent compensation of money and can be transferred and traced in a tamper-resistant manner. Especially the ability of smart contracts to transfer tokens *in the program code* enables the implementation of the introduced monitoring system. On the contrary, if consumers and provider do not execute any other transactions in that blockchain network, they have to exchange them for global currencies such as Dollars or Euros. Especially for industrial use-cases this comes with currency risks as the exchange rate is volatile. State-supported Blockchain networks could help to tackle that issue.

6. Summary and Conclusion

The trend towards dynamic cloud markets, where consumers purchase cloud services from various providers in an ad-hoc manner, leads to heterogeneous IT-infrastructures. Consumers

have to ensure that the service performance conforms to the published service specification as violations might lead to penalty payments. With the increasing number of potentially unknown service providers the penalty management becomes a relevant issue for consumers, which raises the need of an autonomous penalty-aware cloud monitoring system that ensures transparent penalty management in case of service violations.

In the presented paper, we applied blockchain technology for the realization of such a monitoring system: Traded cloud service are registered in a smart contract, which continuously monitors the performance of the traded cloud service. As soon as a service violation is identified, the smart contract transfers penalty payments to the consumer. Consumers profit from an autonomous transfer of penalty payments while providers profit from a transparent and reasonable assessment of services. Due to the management of the penalty payments in the smart contracts, correct payoffs are ensured for both, the provider and the consumer.

To prove the technical feasibility of the approach, a penalty-aware cloud monitoring system was implemented using the IBM Hyperledger Fabric framework. The implemented scenario, monitoring the availability of the virtual machines from Amazon and Azure, illustrates the generic applicability of our introduced approach. For immediate adoption in industry a unification of service performance metrics is necessary as well as a predictable exchange rate for tokens to global currencies.

References

[1] B. Pittl, W. Mach, and E. Schikuta, "A classification of autonomous bilateral cloud SLA negotiation strategies," in *Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services, iiWAS 2016, Singapore, November 28-30, 2016*, 2016, pp. 379–388.

[2] H. Nakashima and M. Aoyama, "An automation method of SLA contract of web apis and its platform based on blockchain concept," in *IEEE International Conference on Cognitive Computing, ICCC 2017, Honolulu, HI, USA, June 25-30, 2017*, 2017, pp. 32–39. [Online]. Available: https://doi.org/10.1109/IEEE.ICCC.2017.12

[3] A. Maarouf, Y. Mifrah, A. Marzouk, and A. Haqiq, "An autonomic SLA monitoring framework managed by trusted third party in the cloud computing," *IJCAC*, vol. 8, no. 2, pp. 66–95, 2018.

[4] Y. Zhang, X. Li, and Z. Han, "Third party auditing for service assurance in cloud computing," in *2017 IEEE Global Communications Conference, GLOBECOM 2017, Singapore, December 4-8, 2017*, 2017, pp. 1–6.

[5] M. Mainelli, M. Smith *et al.*, "Sharing ledgers for sharing economies: an exploration of mutual distributed ledgers (aka blockchain technology)," *The Journal of Financial Perspectives*, vol. 3, no. 3, pp. 38–69, 2015.

[6] R. B. Uriarte, R. D. Nicola, and K. Kritikos, "Towards distributed SLA management with smart contracts and blockchain," in *2018 IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2018, Nicosia,*

Cyprus, December 10-13, 2018, 2018, pp. 266–271. [Online]. Available: https://doi.org/10.1109/CloudCom2018.2018.00059

[7] N. Neidhardt, C. Köhler, and M. Nüttgens, "Cloud service billing and service level agreement monitoring based on blockchain," in *Proceedings of the 9th International Workshop on Enterprise Modeling and Information Systems Architectures, Rostock, Germany, May 24th - 25th, 2018*, 2018, pp. 65–69. [Online]. Available: http://ceur-ws.org/Vol-2097/paper11.pdf

[8] R. Beck, "Beyond bitcoin: The rise of blockchain world," *Computer*, vol. 51, no. 2, pp. 54–58, 2018.

[9] C. Elsden, A. Manohar, J. Briggs, M. Harding, C. Speed, and J. Vines, "Making sense of blockchain applications: A typology for HCI," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI 2018, Montreal, QC, Canada, April 21-26, 2018*, 2018, p. 458.

[10] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: A survey," *International Journal of Web and Grid Services*, vol. 14, no. 4, pp. 352–375, 2018.

[11] M. Crosby, P. Pattanayak, S. Verma, V. Kalyanaraman *et al.*, "Blockchain technology: Beyond bitcoin," *Applied Innovation*, vol. 2, no. 6-10, p. 71, 2016.

[12] A. Auinger and R. Riedl, "Blockchain and trust: Refuting some widely-held misconceptions," in *Proceedings of the International Conference on Information Systems - Bridging the Internet of People, Data, and Things, ICIS 2018, San Francisco, CA, USA, December 13-16, 2018*, 2018.

[13] O. Labazova, T. Dehling, and A. Sunyaev, "From hype to reality: A taxonomy of blockchain applications," in *52nd Hawaii International Conference on System Sciences, HICSS 2019, Grand Wailea, Maui, Hawaii, USA, January 8-11, 2019*, 2019, pp. 1–10.

[14] A. Welfare, *Commercializing Blockchain: Strategic Applications in the Real World*. Wiley, 2019.

[15] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: platforms, applications, and design patterns," in *International conference on financial cryptography and data security*. Springer, 2017, pp. 494–509.

[16] T. Bayer, L. Moedel, and C. Reich, "A fog-cloud computing infrastructure for condition monitoring and distributing industry 4.0 services," in *Proceedings of the 9th International Conference on Cloud Computing and Services Science, CLOSER 2019, Heraklion, Crete, Greece, May 2-4, 2019*, V. M. Muñoz, D. Ferguson, M. Helfert, and C. Pahl, Eds. SciTePress, 2019, pp. 233–240.

[17] X. Jiang, C. Fischione, and Z. Pang, "Poster: Low latency networking for industry 4.0," in *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks, EWSN 2017, Uppsala, Sweden, February 20-22, 2017*, 2017, pp. 212–213.

[18] A. Noor, D. N. Jha, K. Mitra, P. P. Jayaraman, A. Souza, R. Ranjan, and S. Dustdar, "A framework for monitoring microservice-oriented cloud applications in heterogeneous virtualization environments," in *12th IEEE International Conference on Cloud Computing, CLOUD 2019, Milan, Italy, July 8-13, 2019*, 2019, pp. 156–163.

[19] J. Sun, R. Zhang, J. Zhang, and Y. Zhang, "Pristream: Privacy-preserving distributed stream monitoring of thresholded PERCENTILE statistics," in *35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-14, 2016*, 2016, pp. 1–9.

[20] D. Ye, Q. He, Y. Wang, and Y. Yang, "An agent-based decentralised service monitoring approach in multi-tenant service-based systems," in *2017 IEEE International Conference on Web Services, ICWS 2017, Honolulu, HI, USA, June 25-30, 2017*, 2017, pp. 204–211.

[21] Y. Wang, Q. He, D. Ye, and Y. Yang, "Formulating criticality-based cost-effective monitoring strategies for multi-tenant service-based systems," in *2017 IEEE International Conference on Web Services, ICWS 2017, Honolulu, HI, USA, June 25-30, 2017*, 2017, pp. 325–332.

[22] M. Debe, K. Salah, M. H. U. Rehman, and D. Svetinovic, "Iot public fog nodes reputation system: A decentralized solution using ethereum blockchain," *IEEE Access*, vol. 7, pp. 178 082–178 093, 2019.

[23] A. Alzubaidi, E. Solaiman, P. Patel, and K. Mitra, "Blockchain-based sla management in the context of iot," *IT Professional*, vol. 21, no. 4, pp. 33–40, 2019.

[24] E. J. Scheid and B. Stiller, "Automatic sla compensation based on smart contracts," Technical Report No. IFI-2018.02, April, Tech. Rep., 2018.

[25] H. Nakashima and M. Aoyama, "An automation method of sla contract of web apis and its platform based on blockchain concept," in *2017 IEEE International Conference on Cognitive Computing (ICCC)*. IEEE, 2017, pp. 32–39.

[26] R. B. Uriarte, R. De Nicola, and K. Kritikos, "Towards distributed sla management with smart contracts and blockchain," in *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2018, pp. 266–271.

[27] A. T. Wonjiga, S. Peisert, L. Rilling, and C. Morin, "Blockchain as a trusted component in cloud sla verification," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, 2019, pp. 93–100.

[28] G. Vossen, T. Haselmann, and T. Hoeren, "Cloud computing für unternehmen," *Technische, wirtschaftliche, rechtliche und organisatorische Aspekte. dpunkt, Heidelberg*, 2012.

[29] K. Wüst and A. Gervais, "Do you need a blockchain?" in *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*. IEEE, 2018, pp. 45–54.

Appendix A Setup of the System

The implementation of the system can be found on GitHub[n].

[n] https://github.com/dieni/blockchain-based-cloud-monitoring

The creation of the blockchain network is done with some command-line tools provided by Hyperledger. All the commands to spin up the network including peer nodes for the users are composed in *sms.sh* within the network folder. This script file takes one of the three options: up, down, and generate.

1. Install prerequisites for Hyperledger Fabric as described in its documentation
2. Navigate into the network folder and execute the command *./sms.sh up*
   This will execute multiple operations:
   - If they do not already exist, this will generate certificates for each of the organizations and stores them into the *crypto-config* folder
   - Based on the endorsement policy stated in *configtx.yaml* the artifacts for the channel creation will be generated and stored in the folder *channel-artifacts*. Note: When generating the artifacts, the folder needs to exist already.
   - Starts docker service containers described in the *docker-compose.yaml* file which are an ordering service, peer nodes for each organization and another peer node which acts as a helper for further configuration of the network
   - Within the cli peer the *utils.sh* the script will be executed which makes use of the crypto files of the different organizations and performs following operations:
     – Based on the previously generated channel artifacts two channels will be created and registered to the ordering service: *channel-aws* and *channel-azure*
     – The AWS peer together with the peer of University are joined to the channel-aws
     – Analog to the previous step the Azure peer together with the peer of the University are joined to the channel-azure
     – The smart contract for the EC2 services gets installed on the Univie and AWS peer and instantiated on the channel-aws
     – The smart contract for the Azure VMs gets installed on the Univie and Azure peer and instantiated on the channel-azure
   - When you see in the terminal following output, the creation of the network was successful:
     '========= SMS Up! ==========='
3. Generate wallets for each of the organizations
   This is necessary for the SDK to interact with the ledger
   (a) Go into the *crypto-config* folder and copy the name of the .pem file of the organization you would like to interact with the network.
   `/network/crypto-config/peerOrganization/<organization>/users/<user>/msp/keystore/<...9a08817cd53f6_sk>`

   Beware that the different users of an organization have different permissions. Those were stated in the endorsement policy on the creation process of the channel artifacts. For the reading and writing *User1*.

(b) Within the folder of the organization open the file:
`organization/{organization}/application/addToWallet.js`

- Past the name of the .pem file and make sure that the path to the .pem file matches the actual location.
- Define the path to the MSP certificate of the identity form the *crypto-config*
- Define a label for the identity and save the file

(c) Execute the *addToWallet.js*
This will create a wallet within the folder of the organization which can be used for an application to authenticate to the peer and manage transactions

(d) Repeat this procedure for all three organizations

4. Execute the *docker-compose* file with the organization folder
This will create the container for the User Interface and the API of each organization. The user interfaces for the organizations can be found under:

- localhost:4000 (univie)
- localhost:4010 (azure)
- localhost:4020 (aws)

API of the System

The implementation is located in */organizations/org/server* and uses the Nodejs library 'express' for building a REST API. Depending on the organization this interface consists either of the routes for the AWS service, for the Azure service, or both in the case of the University of Vienna. The APIs can be accessed under the following addresses:

- localhost:4001 (University of Vienna)
- localhost:4011 (Azure)
- localhost:4021 (Amazon)

The endpoints for EC2 instances of the API are as following:

/aws/services	
Method	GET
Returns the keys of all EC2 instances registered in the ledger of the AWS channel. It uses the function *getallKeys* of the AWS EC2 contract. This operation does not change the state of the ledger.	
Request	-
Response	HEADER: Content-Type: application/json BODY: List of service keys

/aws/services	
Method	POST
Register a EC2 instance in the ledger of the AWS channel. Uses the function *createService* of the AWS EC2 contract. This operation changes the state of the ledger.	
Request	HEADER: Content-Type: application/json BODY: { name: < *string* >, dimensionName: < *string* >, dimensionValue: < *string* >, region: < *string* >, accessKeyId: < *string* >, secretAccessKey: < *string* >, promisedAvailability: < *string* > {
Response	-

/aws/services/< *serviceKey* >	
Method	GET
By passing the service key in the url the ledger will be queried for all measurements of a service. It uses the function *getStateHistory* of the AWS EC2 contract. This operation does not change the state of the ledger.	
Request	-
Response	HEADER: Content-Type: application/json BODY: List of measurements

/aws/services/< *serviceKey* >	
Method	PUT
By passing the service key in the url information about a service will be retrieved from the service provider and written to the ledger. It uses the function *updateService* of the AWS EC2 contract. This operation changes the state of the ledger.	
Request	-
Response	-

The endpoints for Azure VM of the API are as following:

/azure/services	
Method	GET
Returns the keys of all Azure VM instances registered in the ledger of the Azure channel. It uses the function *getallKeys* of the Azure VM contract. This operation does not change the state of the ledger.	
Request	-
Response	HEADER: Content-Type: application/json BODY: List of service keys

/azure/services	
Method	POST
Register a Azure VM instance in the ledger of the Azure channel. Uses the function *createService* of the Azure VM contract. This operation changes the state of the ledger.	
Request	HEADER: Content-Type: application/json BODY: { name: < *string* >, tenant: < *string* >, clientId: < *string* >, clientSecret: < *string* >, subscriptionId: < *string* >, resourceGroup: < *string* >, computerName: < *string* >, promisedAvailability: < *string* > {
Response	-

/azure/services/< *serviceKey* >	
Method	GET
By passing the service key in the url the ledger will be queried for all measurements of a service. It uses the function *getStateHistory* of the Azure VM contract. This operation does not change the state of the ledger.	
Request	-
Response	HEADER: Content-Type: application/json BODY: List of measurements

/azure/services/< *serviceKey* >	
Method	PUT
By passing the service key in the url information about a service will be retrieved from the service provider and written to the ledger. It uses the function *updateService* of the Azure VM contract. This operation changes the state of the ledger.	
Request	-
Response	-