# Smart Contract Invocation Protocol (SCIP): A Protocol for the Uniform Integration of Heterogeneous Blockchain Smart Contracts

Ghareeb Falazi[1(✉)], Uwe Breitenbücher[1], Florian Daniel[2], Andrea Lamparelli[2], Frank Leymann[1], and Vladimir Yussupov[1]

[1] IAAS, University of Stuttgart, Stuttgart, Germany
{falazi,breitenbuecher,leymann,yussupov}@iaas.uni-stuttgart.de
[2] DEIB, Politecnico di Milano, Milan, Italy
florian.daniel@polimi.it, andrea.lamparelli@mail.polimi.it

**Abstract.** Blockchains are distributed ledgers that enable the disintermediation of collaborative processes and, at the same time, foster trust among partners. Modern blockchains support smart contracts, i.e., software deployed on the blockchain, and guarantee their repeatable, deterministic execution. Alas, blockchains and smart contracts lack standardization. Therefore, smart contracts come with heterogeneous properties, APIs and data formats. This hinders the integration of smart contracts running in different blockchains, e.g., into enterprise business processes. This paper introduces the Smart Contract Invocation Protocol (SCIP), which unifies interacting with smart contracts of different blockchains. The protocol supports invoking smart contract functions, monitoring function executions, emitted events, and transaction finality, as well as querying a blockchain. The protocol is accompanied by a prototypical implementation of a SCIP endpoint in the form of a gateway.

**Keywords:** Blockchain · Smart Contract · Integration · SCIP

## 1 Introduction

Blockchains allow autonomous parties to engage in collaborative processes even if they have a limited degree of mutual trust. What they trust is the blockchain, which hosts exchanged data in a distributed, persistent, and immutable fashion. Blockchains thus eliminate the need for trusted third-parties, e.g., certification authorities, and lower complexity and operational costs. *Smart contracts* are user-defined applications deployed on blockchains that can be executed deterministically. They were first introduced by Ethereum [15] and later adopted by other blockchain systems. Smart contracts are usually used to model the sensitive business logic of collaborative scenarios that are governed by blockchains.

Blockchains can be categorized as permissionless and permissioned. Permissionless blockchains favor total decentralization, censorship resistance, and data immutability; permissioned blockchains favor data confidentiality, performance, and strong transaction processing (TP) semantics [5]. Different blockchains choose their own trade-offs at a finer degree [14]. This means that there is no "one size fits all" blockchain system, and different types of blockchains will continue to evolve and coexist. This results in the possibility that large-scale (e.g., enterprise) applications have to deal with multiple blockchains at once, each of which handling a subset of use-cases required for its own purposes [6,8].

However, blockchains lack sufficient standardization, which results in many heterogeneous APIs, different protocols and message formats. This applies to the modalities of invoking smart contract functions and monitoring their execution as well. As a result, enterprises that need to integrate the smart contracts of multiple blockchains into their processes will be faced with a tedious and error-prone task adapting to many heterogeneous interfaces.

In this work, which is a continuation of our previous research [3,4,10], we aim at providing a uniform way to interact with heterogeneous smart contracts by conceptualizing and specifying the *Smart Contract Invocation Protocol* (SCIP), which supports invoking smart contract functions and querying past and future smart contract-related events uniformly, regardless of the underlying blockchain technology. The goal is to provide an abstraction layer on top of blockchains, exposing a uniform set of operations that allow external applications to interface with smart contracts without needing to adapt to heterogeneous protocols and APIs. We also implement a prototypical gateway as a reference implementation of the protocol, and demonstrate the protocol's benefits in a case study.

## 2 Motivation

Figure 1 illustrates a collaboration setup in the domain of smart grids (from the point of view of an electrical energy provider) that requires the use of different blockchain systems: a blockchain-based energy exchange system. The management of the system is implemented using a permissioned blockchain backed by a consortium involving the various stakeholders, e.g., power plants, energy providers, and consumer households. Using a permissioned blockchain guarantees the necessary transaction processing performance and ensures confidentiality. The actual interactions between the stakeholders, e.g., selling and buying energy, is implemented using a set of smart contracts deployed on the blockchain. The scenario depicted here takes place when a power plant announces the availability of electricity at a low price. The energy provider, which monitors price changes, reacts by buying electricity in bulk from this plant and reducing the retail price of the energy it sells to households. These operations are implemented as functions of the smart contracts. To facilitate public audits, a signed digest of the previous operations is stored on a permissionless blockchain, such as Ethereum, which ensures its immutability and assigns it to a specific time instant. The digest guarantees auditors that a respective blockchain state indeed existed at that specific time instant. The necessary logic is implemented by a dedicated smart contract deployed on the permissionless blockchain.

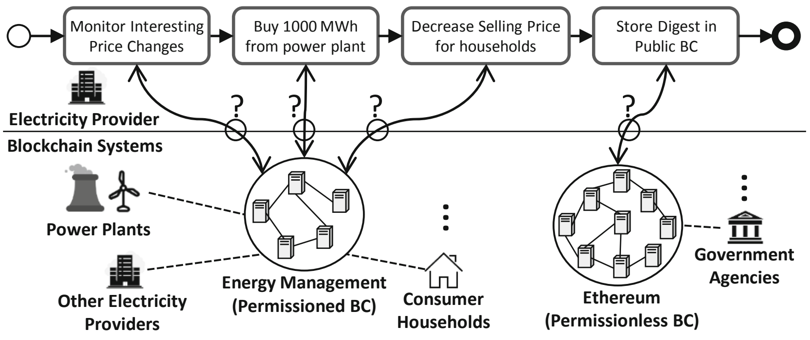
Monitor Interesting Price Changes
Buy 1000 MWh from power plant
Decrease Selling Price for households
Store Digest in Public BC
Electricity Provider
Blockchain Systems
?
?
?
?
Power Plants
Other Electricity Providers
Energy Management (Permissioned BC)
Consumer Households
Ethereum (Permissionless BC)
Government Agencies

**Fig. 1.** Motivating scenario: energy management system with two blockchains.

Integrating different blockchains using traditional programming languages, such as Java or a workflow language, requires dealing with different APIs, and supporting different protocols and message formats. A typical invocation flow of a state-changing smart contract function (one that writes on the blockchain) is demonstrated in Fig. 2. Here, an external consumer application, such as our electricity provider, wants to call the function `Fn1` of the smart contract `SC1`, so it submits a *technology-specific, signed blockchain transaction* (Tx) to one of the nodes of the desired blockchain system ❶. Then the node verifies the request and broadcasts it to the other nodes of the system ❷. Afterwards, the transaction enters the consensus process, which produces a so-called *block*, i.e., a set of agreed-upon transactions that the system has to execute next. This block is announced throughout the network and is cryptographically chained with the previous blocks at each node ❸. When the execution of the new block reaches Tx, all nodes extract the invocation parameters from its body, and use them to invoke the function `Fn1` inside `SC1` ❹. To this end, some form of a virtualization technology, like Docker, or Ethereum Virtual Machine (EVM) [15] is used to instantiate and run the code of the desired smart contract. During the execution, the code may read from and write to the current state causing it to deterministically change on all nodes. Although some blockchains, like Hyperledger Fabric [1], adopt a different internal flow of request processing, using blockchain smart contracts still looks similar to external consumers: they need to access a blockchain node and send *technology-specific requests* to it.

The previous scenario shows that common interactions between consumer applications and smart contracts involve: (i) invoking smart contract functions, (ii) the live monitoring of events that occur due to smart contract function invocations, and (iii) querying of past events and invocations required for data analytics and auditing. The detailed explanation of a smart contract invocation flow further shows that (iv) interacting with smart contracts involves different technologies – as a matter of fact, different networks. In addition, it is important to note that certain blockchains only support a probabilistic model of transaction durability [3,4], which (v) leaves it up to the client application to ensure that a submitted transaction has sufficient probability of being permanently stored.

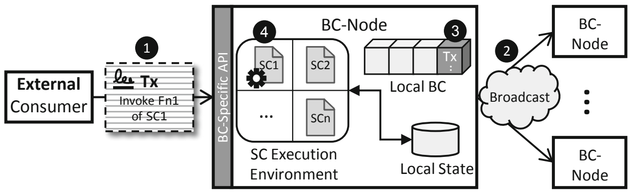

**Fig. 2.** The typical invocation process of a blockchain smart contract function.

**Problem Statement:** The previous scenario shows that participating in multiple heterogeneous blockchains requires dealing with the specificities of each blockchain system individually. This is a tedious and an error-prone task. Therefore, the problem this research approaches is *defining a set of common operations that allow smart contract clients to invoke and monitor smart contracts of heterogeneous permissioned and permissionless blockchain systems and conceptualizing and implementing a uniform protocol that supports these operations.*

## 3 Smart Contract Invocation Protocol (SCIP)

Our answer to the problem statement is the *Smart Contract Invocation Protocol* (SCIP) for the uniform interaction with heterogeneous smart contracts.

The SCIP protocol provides a homogeneous interface (roles, methods, data and message formats) for heterogeneous blockchains. The core of the interface consists of a set of *methods* that can be used by blockchain-external consumer applications, which we will refer to as *client applications*, to interact with *smart contracts*. The methods are provided to client applications via an entity, which we will refer to as the *gateway*, as this entity mediates between two or more different network technologies: the Internet and the blockchain networks. This gateway is reachable using a *Smart Contract Locator* (SCL), which is a uniform URL defined in a previous work [10], that uniquely identifies a smart contract outside the blockchain. For example, the SCL address to locate the digest-storing smart contract of the motivating scenario in Sect. 2 could look as follows: `https://gateway.com?blockchain=ethereum&blockchain-id=eth-mainnet&address=0xa0b73...0b80914`. Here, `gateway.com` is the domain of the gateway, which is thus addressable from the Internet; `ethereum` tells that the blockchain type is Ethereum; `eth-mainnet` indicates that the intended Ethereum instance is the main chain; and `0xa0b73...0b80914` is the address (shortened for brevity) at which the smart contract can be accessed within the the blockchain.

We assume that client applications *authenticate* themselves with the gateway using OAuth 2.0, and that attacks like the Man-In-The-Middle (MITM) are thus prevented. We further assume that the gateway is aware of the client's *digital*

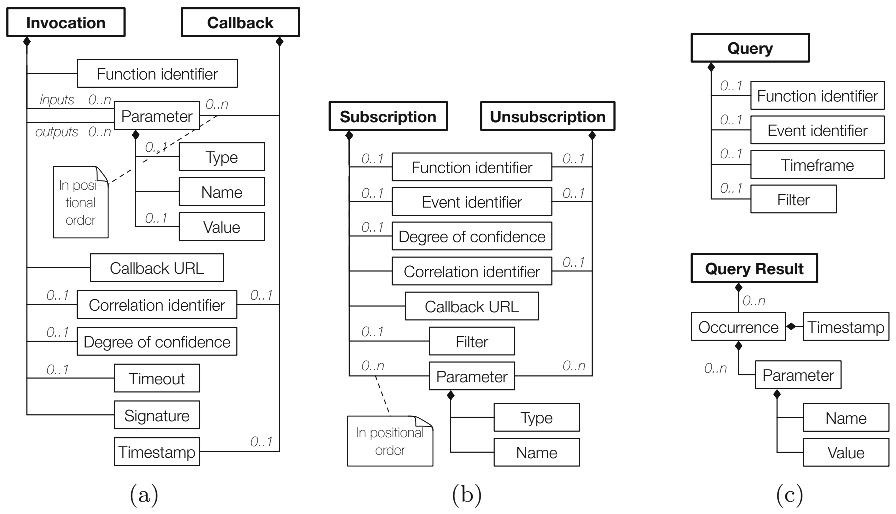

**Fig. 3.** The metamodel of SCIP messages (bold boxes) and message content fields (regular boxes). For readability, the metamodel is split into related sub-models.

*certificate(s)*. These assumptions are in line with those by typical Blockchain-as-a-Service (BCaaS [11]) vendors like Amazon[1], Upvest[2] or Kaleido[3].

## 3.1 Protocol Specification

Figure 3 shows the metamodel of four request and two response messages of the protocol. Table 1 specifies the details of call constructs. The four request messages define four methods: (i) the *invocation* of a smart contract function, (ii) the *subscription* to notifications regarding function invocations or event occurrences, (iii) the *unsubscription* from live monitoring, and (iv) the *querying* of past invocations or events. All methods return a synchronous response message indicating the success or failure of the request (standard HTTP responses), and some of them additionally return one or more asynchronous responses or errors. Some methods refer to the point in time at which an event or a function invocation took place. In this context, *time* refers to the UTC timestamp of the transaction that triggered the event or invoked the function. Time is represented in SCIP using the ISO 8601-1:2019 combined date and time representation. Certain other methods have a parameter called *degree of confidence* (DoC). This refers to the likelihood that a transaction included in a block will remain persistently stored on the blockchain [4]: if a block turns out to be on a side branch of the blockchain it – including the transaction – may eventually be dropped from the blockchain. A value close to 1 means that the client application wants to receive

[1] https://aws.amazon.com/managed-blockchain. Visited on May 6, 2020.
[2] https://upvest.co Visited on May 6, 2020.
[3] https://kaleido.io Visited on May 6, 2020.

**Table 1.** Description of the fields used in SCIP request and response messages.

| Name | Type | Description |
|---|---|---|
| Function Identifier | string | The name of the function |
| Event Identifier | string | The name of the event |
| Inputs | Parameter[ ] | A list of function inputs |
| Outputs | Parameter[ ] | A list of function/event outputs |
| Callback URL | string | The URL to which the callback message must be sent |
| Correlation Identifier | string | A client-provided correlation identifier |
| Degree of Confidence | number | The degree of confidence required from the transaction |
| Timeout | number | The number of seconds the gateway should wait for the transaction to gain the required degree of confidence |
| Signature | string | The client's base 64-encoded signature of the contents of a request message |
| Timestamp | string | The time at which an event occurrence/function invocation happened |
| Filter | string | A C-style Boolean expression to select only certain event occurrences or function invocations |
| Timeframe | string | The timeframe in which to consider event occurrences/function invocations |
| Occurrences | Occurrence[ ] | A list of event occurrences/function invocations |
| Parameter | | |
| Name | string | The name of the parameter |
| Type | JSON schema | The abstract blockchain-agnostic type of this parameter |
| Value | any | The value of this parameter |
| Occurrence | | |
| Parameters | Parameter[ ] | A list of event/function parameters |
| Timestamp | see above | |

the result only after ruling out this possibility, whereas a value close to 0 means that it wants to receive the result as soon as it is available. It is further important to note that client messages are sent to the smart contract's SCL, which triggers the gateway. The actual SCIP endpoint is thus the gateway, which is able to extract the id of the target smart contract from the SCL and to forward incoming messages. The messages thus do not need any specific address in their body.

**The `Invoke` Method:** This method allows an external application to invoke a specific smart contract function. The structure of the `Invocation` request message, as well as the asynchronous `Callback` message are explained in Fig. 3a. Figure 4 shows the steps taken by the client application and the gateway when this method is triggered: The client application formulates an `Invocation` request message, signs it using the algorithm "SHA256withECDSA" [2] and the normative curve "secp256k1", and sends it to the gateway ❶. Then, the gateway formulates a blockchain transaction out of the request message (using the *function identifier*, and *input* fields), and signs it on behalf of the client application. Afterwards, it permanently stores the pair defined by the signed transaction (Tx) and the Signed Request Message (SRM) ❷.

The reason is that blockchain transactions are always signed by their submitters. However, the client application has no chance to do that itself since

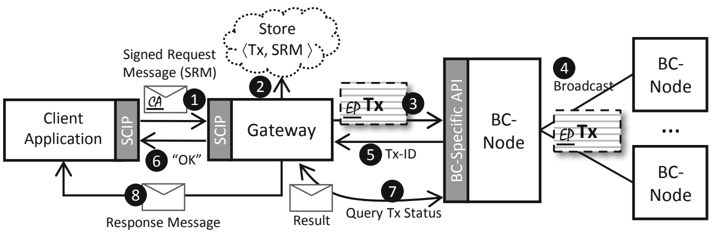

**Fig. 4.** The steps performed during the execution of an `Invoke` method.

it is unaware of the structure of the technology-specific transaction, which prevents it from formulating it in the first place. Therefore, the gateway, which knows the technical details, formulates the transaction and signs it on behalf of the client application. If the client application and the gateway are managed by two different legal entities, then the gateway might need to prove, e.g., to auditors, that the resulting transaction is indeed based on actual inputs from a unique client application request. To this end, the client application is obliged to accompany the request message with a digital signature of its content. Then, the gateway can store the pair consisting of this message along with the formulated transaction in order to prove the desired property at any time. One option for this pair to be stored is a dedicated smart contract deployed on the underlying blockchain or on a different blockchain. Another option is to store it locally by the gateway. It is left to the gateway to decide how to store it. On the other side, if the client application and the gateway are managed by the same legal entity, e.g., a gateway that manages the blockchain access for an enterprise with many internal applications, then signing the request message is not mandatory.

In the next step, the gateway sends the signed transaction to a blockchain node using its API ❸. The node, then, validates it and starts the consensus process by announcing it to the network of nodes ❹, assigning a unique id to the transaction and informing the gateway about it ❺. In response, the gateway informs the client application about the successful submission of the transaction (synchronous response to the original client request) ❻, and at the same time, starts querying the blockchain node about the status of the transaction ❼. If the transaction receives enough confidence, according to the *degree of confidence* field of the request message before the *timeout* is reached, the gateway sends an asynchronous message to the address specified by the *callback URL* field containing the execution results ❽. Note that the gateway is allowed to have its own internal timeout for such requests, which may differ from the one provided by the client. Therefore, clients should expect an asynchronous `timeout error` (wrapped as a response message) from the gateway even before the *timeout* they provided is over. To facilitate the correlation between the request message and the response message by the client application, the callback contains a copy of the *correlation identifier* provided in the request message.

**The `Subscribe` Method:** This method facilitates the live monitoring of smart contracts by enabling a client application to receive asynchronous notifications about the occurrences of custom or system-defined smart contract-related events or smart contract function invocations. The structure of the `Subscription` request message is explained in Fig. 3b. When receiving such a message, the gateway identifies the designated event/function using the fields *event identifier* or *function identifier*, respectively. The field *parameters* further helps the gateway to differentiate between overloads (functions with the same name but different parameters). Then the gateway starts monitoring the events or functions. When an occurrence is detected, the associated event outputs/function inputs are used to populate and evaluate the Boolean expression specified in the *filter*. If the expression returns `true` and the transaction causing the occurrence has reached the desired *degree of confidence*, a callback to the address specified in the *callback URL* field of the respective `Subscription` message is issued. The `Callback` message, described in Fig. 3c, includes details about the *parameters* associated with this occurrence as well as its *timestamp*. The *correlation identifier* of the request message is further included in the response message to enable message correlation by the client application. Note that every subscription made by a client application to the events/functions of a specific smart contract is identified by its *correlation identifier*. A new `Subscription` with a *correlation identifier* already in use overwrites the former one.

**The `Unsubscribe` Method:** This method is used to explicitly cancel subscriptions of a client created using previous invocations of the `Subscribe` method. The structure of the request messages is explained in Fig. 3b. It has four optional fields, which can be used in three ways: (i) if only either *function identifier* or *event identifier* plus *parameters* are present, then all respective subscriptions that belong to the target smart contract are canceled; (ii) if only the *correlation identifier* is provided, then only the subscription corresponding to the identifier is canceled; (iii) if none of the parameters is provided, then all subscriptions to the target smart contract are canceled. All other combinations are invalid. This methods only has a synchronous response that indicates its success or failure.

**The `Query` Method:** This method allows a client application to query the previous occurrences of events or function invocations. The structure of the `Query` request message, as well as the synchronous `Query Result` response message are explained in Fig. 3c. When receiving a `Query` request message, the gateway scans the history of the blockchain and searches for event occurrences/function invocations with a prototype that matches the provided *event identifier*/*function identifier* and *parameters*. Furthermore, *timeframe* specifies the time frame in which the search results should be considered. If the start of this timeframe is not provided, then the time of the genesis block is taken. Similarly, if the end time is not provided, then the time of the latest known block is taken. An optional *filter* can be specified similar to the `Subscription` message. Finally, the gateway synchronously returns a list of *occurrences*. Each occurrence indicates the corresponding event/function and which values were emitted from it or passed to it. It also indicates the *timestamp* when the occurrence took place.

```
--> {"jsonrpc": "2.0" , "method": "Subscribe" , "id": 1,
     "params": { "eventId": "priceChanged",
                 "params": [{"name": "newPrice",
                             "type": { "type":"integer",
                                       "minValue": 0,
                                       "maxValue": 65535 }
                            }, ...],
                 "doc": 98.9,
                 "corrId": "abcdefg12345",
                 "callback": "https://my-domain.com/callbacks",
                 "filter": "newPrice <= 500" }
    }
<-- {"jsonrpc": "2.0", "result": "OK", "id": 1}
<-- {"jsonrpc": "2.0", "method": "ReceiveCallback",
     "params": { "eventId": "priceChanged",
                 "params": [{"name": "newPrice", "value": 410}, ...],
                 "timestamp": "2019-11-06T17:08:00Z",
                 "corrId": "abcdefg12345" }
    }
```

**Fig. 5.** Example JSON-RPC message exchange for the `Subscribe` SCIP method (`-->` from client application to gateway, whereas `<--` in the other direction).

## 3.2 SCIP JSON-RPC Binding

SCIP does not prescribe SCIP endpoints which transport protocol to use to exchange its messages. That is, it does not prescribe its binding to a lower-level network protocol. In this paper, we propose a JSON-RPC [9] binding for SCIP[4]. JSON-RPC is a stateless transport-agnostic remote procedure call (RPC) protocol that uses JSON as its serialization format. It is widely used to publish the APIs of blockchains, such as Ethereum, and using it as a SCIP binding thus maintains consistency with existing blockchain conventions. Figure 5 provides an example SCIP message exchange using the JSON-RPC binding; a client application subscribes to an event and receives a synchronous confirmation and an asynchronous callback with an occurrence of the event.

## 3.3 Handling Data Types in SCIP

Generally, different blockchains support different encodings and types for parameters passed to or returned from smart contract functions or events. To hide this heterogeneity, SCIP proposes a technology-agnostic, abstract format for the data values using JSON Schema[5]. JSON Schema describes the structure of JSON data instances using basic JSON types and allows one to declare constraints on values, group values into arrays and tuples, and nest values into JSON objects. This way, native blockchain data types can be uniquely and abstractly described, and client applications can formulate function inputs in text-based JSON, without having to understand native data types.

Given the abstract specification of data inputs, the gateway can translate them into blockchain-specific formats to interact with the blockchain. For exam-

[4] Complete binding available at: https://github.com/lampajr/scip.
[5] JSON Schema: https://json-schema.org.

```
{"scdl_version": "1.0.1", ...       // generic smart contract properties
 "name": "SC3",                     // smart contract name
 "functions": [{                    // list of functions
    "name": "getDigest", ...        // function name and other properties
    "inputs": [{                    // function inputs
         "name": "client",          // parameter name
         "type": {                  // parameter type in JSON Schema
            "type": "string",
            "pattern": "^0x[a-fA-F0-9]{40}$"
         }}, ...],                  // other inputs of the function
    "outputs": [],                  // function outputs
    }, ...],                        // other functions of the smart contract
 "events": [{                       // event definitions
    "name": "digestStored", ...     // event name and other properties
    "outputs": [...]                // event outputs
    }, ...]                         // other events of the smart contract
}                                   // end of SCDL descriptor
```

**Fig. 6.** Example SCDL descriptor [10] for an Ethereum smart contract.

ple, in the case of a smart contract invocation in Ethereum, the gateway will formulate a suitable *function selector* in order to invoke the intended function. This function selector is defined as the first four bytes of the SHA-3 hash of the signature of the function, which is a string composed of the function name and the parenthesised list of parameter types separated by commas, e.g., `"getDigest(address,string)"`. In order for the gateway to know which native data types to use, the request messages sent by the client application must include the abstract parameter types embedded in the *type* fields of each parameter. Then the gateway, uses 1-to-1 mapping rules predefined for each blockchain system to generate the corresponding native types out of the abstract ones[6]. This means that the gateway performs an encoding of function inputs to exactly match what the underlying blockchain expects, e.g., in terms of byte padding, arrays bracketing, serialization of complex objects, etc. The described mapping of course also applies when the client application specifies a function or event to be monitored (using the `Subscribe` method), or to be queried (using the `Query` method). Even though certain blockchains, like Hyperledger Fabric, have untyped parameters (strings), the SCIP protocol still supports abstract types to address cases like Ethereum and to enable clients to know whether a given parameter expects a numerical or textual input.

In order for a client application to learn about the abstract parameter types of functions, we assume that they have access to a *Smart Contract Description Language* (SCDL) descriptor of their target smart contracts. Such is obtained either via a dedicated *SCDL registry* or through direct contact with service providers. SCDL is a result of our previous work [10], in which we analyzed the smart contract capabilities of six prominent permissioned and permissionless blockchains, and proposed an abstract, technology-agnostic language to describe the external interfaces of smart contracts. For example, Fig. 6 shows a snippet of the SCDL descriptor of the digest-storing Ethereum smart contract presented

[6] Current mapping rules: https://github.com/floriandanielit/scdl#data-encoding.

in the motivating scenario of Sect. 2. The figure shows how abstract types are associated with parameter descriptions (see the `inputs` of the function).

### 3.4 Deployment Modes for SCIP Gateways

SCIP gateways represent concrete implementations of SCIP and can be deployed in a variety of configurations based on certain trade-offs. The two most immediate ones are: (a) BCaaS [11] providers, which act as intermediaries that facilitate the integration of heterogeneous blockchain smart contracts for their clients. These providers assume the responsibility of configuring the necessary SCIP gateways and managing user credentials and permissions. In addition, they are in charge of any costs incurred by accessing the underlying blockchain systems, e.g., the gas costs associated with Ethereum smart contract invocations [15]. To avoid introducing a single-point-of-failure, they may create a distributed SCIP gateway by replicating it so that it tolerates crash- or even Byzantine failures depending on the replication method used [17]. Nonetheless, such a deployment requires client applications to have a certain degree of trust in the BCaaS provider. (b) Alternatively, an enterprise may deploy its own SCIP gateway on a trusted infrastructure, e.g. on-premise, so that availability, security, and trust concerns can be controlled while still supporting its client applications with the uniform interface provided by SCIP. A disadvantage of this deployment is that the enterprise needs to manage the gateway and configure its access to the relevant blockchains itself. In future work, we will investigate the benefits and drawbacks of these and other deployments.

## 4 Validation

### 4.1 SCIP Gateway Implementation

We implemented a prototype of a SCIP gateway extending prior work intended to allow business process engines to access blockchains [3,4]. Figure 7 shows the resulting software architecture and highlights reused (light gray) and newly implemented (dark gray) components. At the top, we implemented a JSON-RPC server that exposes the SCIP methods and a JSON-RPC client that sends callback messages to client applications. Below them, the BAL (Blockchain Access Layer) core provides the logic for handling requests and sending callbacks. An *Expression Parser* supports the *filter* field of certain SCIP methods. A *Security Manager* authenticates client applications using OAuth 2.0, and handles signatures of the `Invoke` method. The core is managed by the *Blockchain Manager*, which coordinates the work of the other components and communicates with the adapter layer below it. For each supported blockchain (at the moment, Ethereum and Hyperledger Fabric), we implemented a pluggable adapter module that handles the specificities of the corresponding blockchain system. For example, it handles how smart contracts are invoked, how parameters are encoded/decoded, how events can be monitored and queried etc. Adapters communicate directly

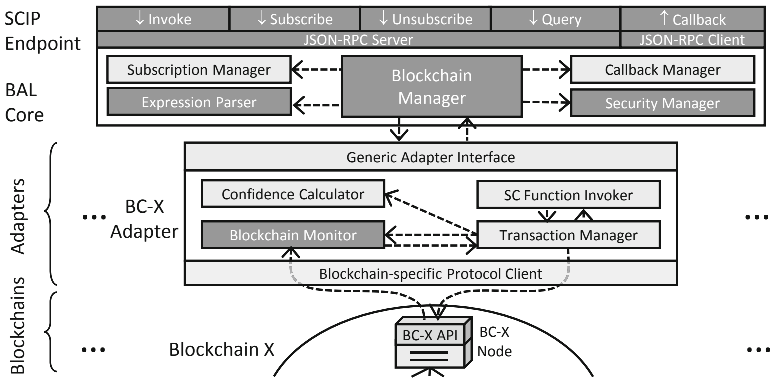

**Fig. 7.** The architecture of a prototypical SCIP gateway based on our previous work [3, 4]. Dark gray components are newly added or significantly modified.

with a blockchain node, and therefore, include a blockchain-specific protocol client to perform blockchain-specific actions, e.g., submitting a transaction. Further details and configuration instructions, e.g., on how to configure access to multiple blockchains simultaneously, can be found on Github[7].

## 4.2 Case Study Implementation

We now realize the motivating scenario of Sect. 2 using the prototypical SCIP gateway. This case study implementation logic is illustrated in Fig. 8[8].

First, we implemented a simplified energy management system via a minimal setup of the Hyperledger Fabric permissioned blockchain that contains an endorsing peer, a transaction ordering service, a certificate authority, and a CLI node, which acts as an interface to external clients. We also deployed two "chaincodes," i.e., smart contracts in Hyperledger terminology, on this setup: `SC1` handles the relationship between power plants and electricity providers via the functions `changeWholesalePrice` and `buyWholesale`, and `SC2` handles the relationship between electricity providers and consumer households via the function `changeRetailPrice`. The function `changeWholesalePrice` of `SC1` emits the event `priceChanged` when a new price is registered by a power plant. We simulate storing digests of the operations performed on the permissioned blockchain on the Ethereum permissionless blockchain by using the Ganache[9] Etherum simulator. On this blockchain, we deploy the smart contract `SC3`, which contains the function `storeDigest` that is responsible for storing the digests. The access to this blockchain is provided to client applications via the *SCIP Gateway 1* and *SCIP Gateway 2* that use adapters to interface with their blockchains.

[7] Available at https://github.com/ghareeb-falazi/BlockchainAccessLayer/.
[8] Implementation available at: https://github.com/ghareeb-falazi/SCIP-CaseStudy.
[9] Ganache: https://www.trufflesuite.com/ganache. Visited on May 6, 2020.

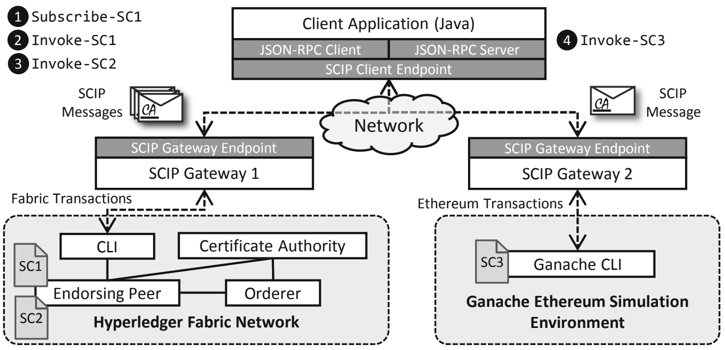

**Fig. 8.** Description of the case study showing some of the exchanged messages.

The Java application that represents the electricity provider is connected to the two gateways over a network. The client application monitors the event `priceChangedEvent` in real time by submitting a `Subscription` request message with a *filter* value of `"newPrice <= 500"`. When this condition is detected by the gateway, it sends a callback to the client. The contents of the messages exchanged during this interaction are shown in Fig. 5. In response to the event, the client application buys energy from the power plant using the `Invoke` method, which triggers the `buyWholesale` function. Afterwards, the application reduces the retail price of the energy it is selling with another call to `Invoke`, but to trigger the `changeRetailPrice` function this time. As the resulting transactions reach confidence, the client application stores the digest of the two `Invocation` messages it issued in the `SC3` smart contract of the permissionless blockchain. To this end, it issues a third `Invoke` method call, but this time to Gateway 2. This triggers the `storeDigest` function of `SC3`.

The use case shows how using the SCIP protocol to communicate with heterogeneous blockchains via gateways (that are transparent to clients) eases integration. SCIP allows a client to deal with a single protocol and to obtain a standard handling of smart contract interaction tasks, such as subscription management, event filtering, and the estimation of the degree of confidence of transactions. These would be intricate and tedious tasks if the client had to master them for each individual blockchain it needs to communicate with.

## 5 Related Work

*Blockchain interoperability* focuses on allowing blockchains to exchange data and events. It can be approached in multiple ways: notary schemes, such as Interledger [7], relay schemes, like Polkadot[10] and Cosmos[11], and hash-locking

[10] Polkadot: https://polkadot.network/. Visited on: May 6, 2020.
[11] Cosmos: https://cosmos.network/. Visited on: May 6, 2020.

schemes, like the Lightning Network[12]. The problem interoperability solves is enabling blockchains to communicate events though they cannot directly invoke external systems. SCIP is different in that it provides a uniform interface for external client applications to communicate with blockchains and smart contracts.

The idea of *blockchain gateways* was introduced by Thomas et al. [6]. They argue that the blockchain architecture should satisfy the same fundamental goals of the Internet architecture, and, thus, they view blockchains as autonomous systems that communicate with each other via gateways. These gateways collectively support the reachability of data stored intra-domain, and facilitate inter-domain transaction mediation. However, the proposed approach does not provide a uniform entry point for external applications to blockchains.

On the other hand, *connector-based integration approaches*, like Unibright [12], introduce own platforms that communicate with blockchains on one hand and with various kinds of other, blockchain-external applications via extensible connectors on the other hand. However, unlike the SCIP protocol, these approaches can cause vendor lock-in, as they rely on proprietary platforms. In addition, they delegate the task of handling blockchain uncertainty to client applications, which requires the involvement of blockchain experts.

Xu et al. [16] take another approach on blockchain integration: they consider blockchains as *software connectors* providing external applications with communication, coordination, conversion and facilitation services. Essentially, they consider blockchains as a means to integrate applications with each other. However, they do not introduce the means that would allow client applications to communicate directly with the blockchains themselves.

Finally, the *Web Ledger Protocol* 1.0 [13] outlines a generic data model and syntax for blockchains. It also introduces the Ledger Agent HTTP API, which describes a standard mechanism to create, append, and query the blockchain. Unlike SCIP, this API does not support smart contracts and delegates too the task of handling blockchain uncertainty to client applications.

## 6 Concluding Remarks and Outlook

In this work we conceptualized and specified the Smart Contract Invocation Protocol (SCIP), a uniform protocol for the integration of heterogeneous smart contracts into enterprise applications. The protocol supports methods triggering smart contract functions, monitoring occurrences of events or function invocations in real time, and querying past occurrences. The protocol specification comes equipped with an implementation of a gateway endpoint, which we validated through a case study using it in practice. The case study shows that the benefits of SCIP are substantive in scenarios that involve multiple heterogeneous blockchains. SCIP thus advances the state of the art in blockchain integration.

[12] Lightning Network: https://lightning.network/. Visited on: May 6, 2020.

As future work, we plan to study alternative deployments of SCIP gateways and analyze their properties and trade-offs. We also plan to study benefits and drawbacks of alternative SCIP bindings. SCIP is available via GitHub (https://github.com/lampajr/scip) and open to contributions by the community.

More in general, SCIP paves the road for SOA-based interoperability of smart contracts and applications that transparently distribute application logic over the Internet and one or more blockchains. This may raise the need for a new wave of software engineering tools and methodologies.

## References

1. Cachin, C., Vukolic, M.: Blockchain consensus protocols in the wild (keynote talk). In: International Symposium on Distributed Computing (DISC 2017), pp. 1:1–1:16 (2017). https://doi.org/10.4230/LIPIcs.DISC.2017.1
2. Certicom Research: Standards for Efficient Cryptography 1 (SEC 1) Version 2.0. Technical report, Certicom Corp. (2009). http://www.secg.org/sec1-v2.pdf
3. Falazi, G., Hahn, M., Breitenbücher, U., Leymann, F.: Modeling andexecution of blockchain-aware business processes. SICS Softw.-Inensiv. Cyber-Phys. Syst. **34**(2–3), 105–116 (2019). https://doi.org/10.1007/s00450-019-00399-5
4. Falazi, G., Hahn, M., Breitenbücher, U., Leymann, F., Yussupov, V.: Process-based composition of permissioned and permissionless blockchain smart contracts. In: EDOC 2019 (2019). https://doi.org/10.1109/EDOC.2019.00019
5. Falazi, G., Khinchi, V., Breitenbücher, U., Leymann, F.: Transactional properties of permissioned blockchains. SICS Softw.-Inensiv. Cyber-Phys. Syst. 1–13 (2019). https://doi.org/10.1007/s00450-019-00411-y
6. Hardjono, T., Lipton, A., Pentland, A.: Towards a design philosophy for interoperable blockchain systems. CoRR (2018). http://arxiv.org/abs/1805.05934
7. Hope-Bailie, A., Thomas, S.: Interledger: creating a standard for payments. In: WWW 2016 Companion. ACM Press (2016)
8. Johnson, S., Robinson, P., Brainard, J.: Sidechains and interoperability. Preprint (2019). http://arxiv.org/abs/1903.04077
9. JSON-RPC Working Group: JSON-RPC 2.0 Specification. Technical report, JSON-RPC Working Group (2010). https://www.jsonrpc.org/specification
10. Lamparelli, A., Falazi, G., Breitenbücher, U., Daniel, F., Leymann, F.: Smart Contract Locator (SCL) and Smart Contract Description Language (SCDL). In: Yangui, S., et al. (eds.) Service-Oriented Computing, pp. 195–210. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-45989-5_16
11. Samaniego, M., Deters, R.: Blockchain as a service for IoT. In: 2016 IEEE iThings/-GreenCom/CPSCom/SmartData, pp. 433–436. IEEE (2016)
12. Schmidt, S., Jung, M., Schmidt, T., et al.: Unibright-the unified framework for blockchain based business integration. White paper, April 2018
13. Sporny, M., Longely, D.: The Web Ledger Protocol 1.0. Technical report, W3C Blockchain Community Group (2019). https://w3c.github.io/web-ledger/
14. Tasca, P., Tessone, C.J.: A taxonomy of blockchain technologies: principles of identification and classification. Ledger **4** (2019). https://doi.org/10.5195/ledger.2019.140
15. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger - Byzantium version. Whitepaper (2018)

16. Xu, X., et al.: The blockchain as a software connector. In: 2016 13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016 (2016). https://doi.org/10.1109/WICSA.2016.21
17. Zhao, W.: Design and implementation of a Byzantine fault tolerance framework for Web services. J. Syst. Softw. **82**(6), 1004–1015 (2009). https://doi.org/10.1016/j.jss.2008.12.037