A Service-Oriented Perspective on Blockchain Smart Contracts

Florian Daniel
Politecnico di Milano, Milan, Italy

Luca Guida
Politecnico di Milano, Milan, Italy

Smart contracts turn blockchains into distributed computing platforms. This article studies whether smart contracts as implemented by state-of-the-art blockchain technology may serve as component technology for a computing paradigm like service-oriented computing (SOC) in the blockchain, in order to foster reuse and increase cost-effectiveness.

A *blockchain* is a shared, distributed ledger, that is, a log of transactions that provides for persistency and verifiability of transactions [1]. A *transaction* is a cryptographically signed instruction constructed by a user of the blockchain [2], for example, the transfer of cryptocurrency from one account to another. Transactions are grouped into blocks, linked and secured using cryptographic hashes. A *consensus protocol* enables the nodes of the blockchain network to create trust in the state of the log and makes blockchains inherently resistant to tampering [3]. Thanks to these properties, blockchain technology is able to eliminate the need for a middleman from the management of transactions, such as a bank in the transfer of money.

Next to logging transactions, blockchain platforms support the execution of pieces of code, so-called *smart contracts* [4, 5], able to perform computations inside the blockchain. For example, a smart contract may be used to automatically release a given amount of cryptocurrency upon the satisfaction of a condition agreed on by two partners. If we put multiple smart contracts (and partners) into communication, we turn the blockchain into a proper distributed computing platform [6]. This makes the technology appealing to application scenarios that ask for code execution that is reliable, verifiable and transactional.

For example, Xu et al. [7] propose the use of smart contracts as software connectors for reliable, decentralized data sharing, while Weber et al. [8] propose the integration of multiple smart contracts for distributed business process execution. The first example aims to support data providers in publishing data sets and data consumers in finding and selecting data sets; using cryptocurrency, data providers are automatically paid according to the value of the provided data, establishing an open, blockchain-based marketplace for data. The second example generates smart contracts starting from a BPMN choreography diagram and puts them into direct communication; the idea is to enable the execution of business processes even among potentially untrusted partners. The common ingredients of both examples are smart contracts and verifiable transactions.

Developing applications that integrate multiple smart contracts is however not easy, and today's predominant ad-hoc development practice won't be able to scale and be sustainable in the long term. In fact, Atzei et al. [9] show that already today even simple smart contracts are often affected by a variety of security vulnerabilities. Nikolić et al. [10] show that several of the smart contracts deployed on Ethereum either "lock funds indefinitely, leak them carelessly to arbitrary users, or can be killed by anyone." Singh and Chopra [11] go beyond implementation aspects and discuss existing socio-technical limitations of smart contracts, such as lack of control, lack of understanding and lack of social meaning.

We argue that future blockchain applications ask for abstractions, methods, and instruments that help developers to cope with complexity, such as those proposed by Service-Oriented Computing (SOC). In fact, the characteristics of the described data sharing scenario directly map to those of SOC (service provider, service consumer, service broker), yet smart contracts still lack equivalent support for description, discovery and the specification of non-functional properties. Similarly, the business process scenario resembles very much that of service-based business processes, yet the smart contracts generated in the scenario are tailored to specific tasks and partner interactions and are not directly applicable in processes with different partners and/or choreography needs. That is, while they present significant opportunities for reuse, they do not yet explore them.

In the following, we thus look at smart contracts from a SOC perspective and study their suitability as elementary pieces for a blockchain-based, distributed computing paradigm. The assumption is that principled reuse not only helps to lower complexity but also increases correctness by design.

BLOCKCHAIN AND SMART CONTRACTS

Next to Bitcoin, several alternative platforms have emerged over the last few years. Besides the *type of cryptocurrency* adopted as incentive mechanism, these platforms distinguish themselves by few key properties.

The *access policy* tells who can participate in the blockchain network. *Public* blockchains allow anyone to join and to access the information stored in the blockchain via the Internet; *private* blockchains are restricted to private networks and selected nodes only.

The *validation policy* tells who among the nodes can participate in consensus creation and deploy smart contracts. *Permissionless* blockchains allow every node to perform both; *permissioned* blockchains limit these capabilities to special nodes only, e.g., qualified through direct invitation.

The *consensus protocol* specifies how trust is created among participants: *Proof of work* (e.g., adopted by Bitcoin) requires nodes, so-called miners, to invest significant hashing power to create trust. *Proof of stake* (Cardano) requires nodes to prove ownership of sufficient cryptocurrency to establish trust. *Byzantine Fault Tolerance* uses replication to establish trust in the state of the network, even if faced with failing network nodes. Variants are redundant BFT (Hyperledger Indy) and practical BFT (Quorum), which aim at increased redundancy and speed, respectively. Other notable consensus protocols are *proof of elapsed time* (Hyperledger Sawtooth), *proof of importance* (NEM), *proof of state* (Universa Blockchain Protocol), *Raft-based consensus* (Quorum), *stream-processing ordering services* (Hyperledger Fabric), and *Tempo* (Radix DLT).

The choice of the consensus protocol affects the *transaction processing time* (time till a transaction is added to a block) and the *transaction rate* (number of transactions processed per second). These properties and the access and validation policies determine a blockchain's ability to support different distributed computing scenarios.

As for the implementation of smart contracts, each platform typically supports one or more *programming languages*. Some support general-purpose languages like C, C++, C#, F#, Go, Java, JavaScript, Kotlin, Objective-C, PHP, Python, Rust, Visual Basic .Net. Others propose platform-specific languages like Bitcoin Script or Ethereum Solidity. The former are Turing complete, the latter not necessarily (e.g., Bitcoin Script is not).

In Table 1, we summarize these characteristics for four platforms: Bitcoin (bitcoin.org), the first blockchain platform; Ethereum (ethereum.org), the platform that first introduced Turing-complete smart contracts; Hyperledger Fabric (hyperledger.org/projects/fabric), a private, permissioned platform hosted by the Linux Foundation and supported by more than 200 industry leaders; and Corda (corda.net), a private, permissioned platform by a consortium of more than 200 financial institutions and technology firms with a focus on interoperability. These platforms represent an opportunistic selection (far from exhaustive) based on our own knowledge and the goal of communicating some of the diversity that characterizes current blockchain technology.

	Bitcoin	**Ethereum**	**Hyperledger Fabric**	**Corda**
Cryptocurrency	Bitcoin (BTC)	Ethereum (ETH)	No built-in currency	No built-in currency
Access policy	Public	Public	Private	Private
Validation policy	Permissionless	Permissionless	Permissioned	Permissioned
Consensus protocol	Proof of work	Proof of work (proof of stake under review*)	Voting-based algorithm (Apache Kafka)	Validity consensus, Uniqueness consensus
Transaction processing time (average)	~ 10 minutes	~ 15 seconds	Almost instantaneous	Almost instantaneous
Max transaction rate	~ 7 TPS	~ 20 TPS	3,500+ TPS	~ 170 TPS
Smart contract language	Bitcoin Script, high-level languages (BALZaC, BitML) compilable to Bitcoin native transactions	Solidity, Serpent, low-level Lisp-like language (LLL), Mutan	Go	JVM programming languages like Kotlin, Java
Turing completeness	No	Yes	Yes	Yes

* https://cryptoslate.com/ethereums-proof-of-stake-protocol-in-review/

Table 1. Core characteristics of four example blockchain platforms

SERVICE ORIENTATION

Service orientation is commonly associated with the binomial SOAP/WSDL or the REST architectural style. Smart contracts use neither of these, so we fall back to the generic definition by Alonso et al. [12] who define services as "components that can be integrated into more complex distributed applications." In order to compare different web service technologies, Lagares Lemos et al. [13] distinguish services by their type, interaction style, interaction protocol, data format, and descriptor. We discuss these characteristics next for smart contracts, in order to enable identifying analogies and differences between the proposed service-oriented interpretation of smart contracts and traditional web service technologies. We specifically focus on Ethereum as such is currently the most used blockchain platform for smart contract development.

Contract type

Components encapsulate data to be fetched and visualized or integrated and/or application logic to be interacted with. What the component delivers is a function of the type of the component. For smart contracts we can distinguish the following contract types:

- *Generic contracts* implement application logic, e.g., for deposit management, that can be invoked by blockchain clients or by other contracts; in general, this type of contract is stateful in that it maintains application state across interactions.

- *Libraries* implement one or more functions, e.g., a math library, that are meant for reuse by other contracts; libraries do not store internal variables and are stateless.

- *Data contracts* provide data storage services inside the blockchain, e.g., a client references manager, that are meant for use by other contracts; by design, they are stateful.

- *Oracles* deliver data services from the outside of the blockchain to the inside of the blockchain, e.g., currency conversation rates. Contracts cannot make calls outside the blockchain, as outside dependencies may prevent verifiability (conversion rates change over time). If data from the outside is needed, it can be pushed by clients to oracles using transactions; these then allow other contracts to query for the data.

Interaction style

Integrating a component into a composite application usually does not only involve a one-shot query or call. It may be necessary to interact with the component multiple times and to establish some form of conversation with it. For smart contracts we have:

- *Pull* interactions enable a client or contract to initiate an interaction and to invoke a contract that otherwise would be passive; for instance, a client may invoke a contract to withdraw a deposit.

- *Push* interactions enable the contract to become active and to initiate an interaction with clients or other contracts; for instance, a contract may invoke a data contract to obtain a list of accounts to send cryptocurrency to.

- *Business-protocol*-based interactions support patterns that may involve multiple interactions and multiple clients or contracts; the protocol specifies the order of interactions and the roles of the involved parties.

As running smart contracts costs money, contracts are activated only in response to explicit invocations. A contract or a group of interacting contracts is thus always triggered by a client transaction, and independent, active behaviors are typically not supported.

Interaction protocol

This tells how a component implements its interactions. Conventional web services use message-oriented protocols like SOAP or HTTP, while all major programming languages also support RPC-like interactions (Remote Procedure Calls). Ethereum uses a message-based protocol supporting the following interaction features:

- *Transactions* are used by blockchain clients (the users of the blockchain) to create new contracts or to invoke existing contracts; once validated, which consumes cryptocurrency, transactions are added to the blockchain and remain publicly accessible.

- *Events* enable a contract to push information to the outside world in response to a transaction invoking the contract; when the transaction is added to the blockchain, also the event becomes publicly accessible.

- *Calls* (so-called *message calls*) are used by contracts to interact with each other in a fashion that uses different state spaces for each contract for isolation; calls are executed locally to each blockchain node and do not consume cryptocurrency.

- *Delegate calls* are used by contracts to invoke libraries in a fashion where functions are executed in one, the caller's, state space; delegate calls too are node-local and do not consume cryptocurrency.

If an interaction originates from a blockchain client, it uses JSON-RPC or is enacted using the command line; if it originates from a smart contract, the message is exchanged via RPC. Transactions contain a set of predefined parameters: the number of transactions sent by the sender, the amount of cryptocurrency the sender is willing to pay for consumed resources (so-called *gas*), the maximum consumable amount of gas, the address of the recipient, the amount of cryptocurrency to be transferred, possible signatures of the sender, and either the code of the contract to be created or input data to be processed. Events contain, among others, one or more topics that allow clients to search for and subscribe to events and a data field. Calls contain the sender and receiver addresses, a possible value and data; calls may return a value.

Data format

The data format determines how exchanged data is formatted. Message-oriented interaction protocols typically support self-describing document formats like XML and JSON; RPC-oriented protocols enable the exchange of native data structures, such as Java or JavaScript objects, using an internal, binary format hidden to developers.

Data in Ethereum transactions and events is encoded using the Application Binary Interface (ABI), which specifies how functions are called and data are formatted. Clients either serialize data in a binary format on their own, e.g., when using the command line or by using a suitable library function, e.g., the function `toPayload` of the library web3.js. Values are encoded in sequential order and according to their data types and are not self-describing. In order to allow the receiver to identify which function is called, the sequence of values is preceded by 4 bytes of a Keccak-256 hash of the respective function signature. This allows everybody to parse the binary formatted data.

Data in message/delegate calls between contracts is exchanged by passing variables, masking the underlying ABI formatting.

Description

The final aspect of components is component description, which enables discovery and selection. For web services, description languages like WSDL and WADL and semantics-oriented languages like OWL-S, WSDL-S, and WSMO are used to describe service endpoints, operations and data formats.

The construct that gets closest to a description of Ethereum smart contracts is the so-called "ABI in JSON" interface description produced by the Solidity compiler during compilation, as exemplified by the following lines of code:

```
[{
  "type": "function",
  "inputs": [{"name": "username", "type": "string"},
             {"name": "password", "type": "string"}],
  "name": "create_user",
  "outputs": [{"name": "success", "type": "bool"}]
}, {
  "type": "event",
  "inputs": [{"name": "username", "type": "string", "indexed": true},
             {"name": "count", "type": "uint256", "indexed": false}],
  "name": "user_created"
}]
```

The description specifies one function (`create_user`) and one event (`user_created`), along with their inputs and outputs. The inputs of the event are their publicly accessible arguments stored in the blockchain; indexed arguments are searchable. What this description does not include is the name of the contract, its address, the network/chain ID if the contract is deployed on a test network, and non-functional properties (e.g., the cost of invoking the function). These are essential for search and discovery. Also, Ethereum does not come with a registry for smart contracts, although contract metadata (containing the ABI in JSON description) can be stored in Swarm, a redundant and decentralized store of Ethereum's public record.

STATE OF TECHNOLOGY

In Table 2, we summarize how these SOA characteristics are manifest (or not) in the four platforms we introduced earlier.

As expected, Bitcoin is the most limited platform in terms of features supported when it comes to smart contracts. In fact, it was born as support for its homonymous cryptocurrency and less to support generic computations. Ethereum, on the other hand, is the most complete platform, with Hyperledger Fabric and Corda providing comparable features.

In terms of contract types, all platforms support oracles, except Hyperledger Fabric for which so-called "gateway services" are still under discussion (as of June 2018). Reusable code libraries are supported only by Ethereum and Corda. It is important to note that contracts generally encapsulate application logic; data contracts are typically very limited in their storage capacity, as storing data on the blockchain may incur significant costs.

All platforms except Bitcoin support pull and push interactions; Bitcoin features only client-originated pull transactions. Looking at the interaction protocols, Ethereum, Hyperledger Fabric and Corda support transactions, calls between contracts, and events; Bitcoin has only transactions.

Payload data is binary formatted in Bitcoin and Ethereum transactions and events, while Ethereum message/delegate calls pass native Solidity data structures. Hyperledger Fabric structures data as key-value pairs in binary and/or JSON format. Corda, in addition to generic Kotlin/Java data objects, also supports transactions with generic attachments; attachments are zipped and hash referenced.

As for the description of smart contracts for search and reuse, support is very limited. Only Ethereum and Hyperledger Fabric provide basic metadata describing a contract's interface (operations and arguments), but we are far from a common description format let alone a registry for the discovery of contracts.

	Bitcoin	Ethereum	Hyperledger Fabric	Corda
Contract type	Contracts, oracles	Contracts, libraries, data contracts, oracles	Contracts (chaincode), data contracts	Contracts, libraries, oracles
Interaction style	Pull interactions	Pull and push interactions, business protocols	Pull and push interactions	Pull and push interactions, business protocols
Interaction protocol	Transactions	Transactions, events, message calls, delegate calls	Transactions, calls (limited to contracts on same node and channel), events — exposes REST APIs toward these	Transactions, inter-node messages (so-called flows), scheduled invocations of contracts
Data format	Binary payloads in transactions	Binary payloads in transactions and events, Solidity data types in message/delegate calls	Binary or JSON formatted key-value pairs	Any type of the contract language, zip attachments referenced using hashes
Description	No contract description	Contract metadata (JSON) to be published on a public storage platform (e.g., Swarm)	Chaincode metadata with interfaces, endpoints, and interaction schemas	No contract description

Table 2. The SOC perspective on selected smart contract technologies

DISCUSSION AND OUTLOOK

By now, there is a general consensus that the impact of blockchain will go far beyond cryptocurrencies, possibly with disruptive effects on distributed application development [14]. The key enabler for this impact are smart contracts able to support a new kind of distributed computing [6]. While the number and types of platforms for smart contracts are constantly growing — this article studies four of them, dozens of others have emerged — the resulting technological landscape is getting increasingly intricate and heterogeneous.

Yet, this article shows that from an application point of view the conceptual underpinnings of this new landscape are more integrated than one would expect and that smart contracts, to some extent, may indeed be interpreted as elementary pieces, that is, services, of a blockchain-based, service-oriented computing paradigm. The article however also shows that we are still far from a smart contract model that sees interoperability and reusability as beneficial features, as instead we are used to in the context of service-oriented computing.

In order to enable service orientation in blockchain and to unleash the full power of smart contracts, several challenges need to be faced, among which we mention:

- *Search, discovery and reuse*: It is striking that so little attention has been paid so far to enable developers to reuse already deployed contracts, especially if we consider that deploying a new contract is typically more cost-intensive then just invoking an already deployed one. Suitable abstract descriptors and searchable registries are badly needed.

- *Cost awareness*: Smart contracts natively incorporate the concept of resource consumption and cost of invocations. It is crucial that smart contracts be able to properly communicate and negotiate these kinds of service levels, enabling a natural pay-per-invocation model.

- *Performance*: Libraries and data contracts are executed locally inside each node and have thus negligible response times; oracles and generic contracts, which may require transaction processing, may lead to higher, unpredictable response times. The challenge is improving performance in terms of transaction rates and processing times.

- *Interoperability and standardization*: Today, platforms concentrate on own technologies as distinguishing feature, which is understandable. This, however, slows down integration, which eventually will nevertheless be needed. The challenge is agreeing on shared interaction styles and protocols as well as data formats and, of course, authentication and certification mechanisms. A particular challenge is cross-blockchain integration.

- *Composition*: Finally, in order to be able to exploit the full power of smart contracts (and to collectively save resources and money) it is necessary to conceive and implement composition solutions able to abstract away from technicalities and to provide developers with instruments and infrastructures that enhance productivity effectively.

In short, what we envision is an evolution from today's technology silos to an abstract, reuse-oriented contract ecosystem able to preserve the guarantees proper of blockchain technology.

REFERENCES

1. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
2. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper 151 (2014): 1-32.
3. D. Mingxiao, M. Xiaofeng, Z. Zhe, W. Xiangwei, and C. Qijun. A review on consensus algorithm of blockchain. 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC).
4. N. Szabo. Smart contracts: building blocks for digital markets. EXTROPY: The Journal of Transhumanist Thought (16), 1996.

5. R. M. Parizi and A. Dehghantanha. Smart Contract Programming Languages on Blockchains: An Empirical Evaluation of Usability and Security. In International Conference on Blockchain, pp. 75-91. Springer, Cham, 2018.
6. B. Dickson. How blockchain can create the world's biggest supercomputer. TechCrunch, December 2016.
7. X. Xu, P. Pautasso, L. Zhu, V. Gramoli, A. Pnomarev, A. B. Tran and S. Chen. The blockchain as a software connector. 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA).
8. I. Weber, X. Xu, R. Riveret, G. Governatori, A. Ponomarev and J. Mendling. Untrusted business process monitoring and execution using blockchain. In International Conference on Business Process Management, pp. 329-347. Springer, Cham, 2016.
9. N. Atzei, M. Bartoletti and T. Cimoli. A survey of attacks on Ethereum smart contracts (sok). In Principles of Security and Trust, pp. 164-186. Springer, Berlin, Heidelberg, 2017.
10. I. Nikolić, A. Kolluri, I. Sergey, P. Saxena and A. Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. arXiv preprint arXiv:1802.06038, 2018.
11. M. P. Singh and A. K. Chopra. Violable Contracts and Governance for Blockchain Applications. arXiv preprint arXiv:1801.02672, 2018.
12. G. Alonso, F. Casati, H. Kuno and V. Machiraju. Web Services, Springer, Berlin, Heidelberg, 2004.
13. A. Lagares Lemos, F. Daniel and B. Benatallah. Web Service Composition: A Survey of Techniques and Tools. ACM Computing Surveys 48(3), February 2016, article 33.
14. L. Mearian. What is blockchain? The most disruptive tech in decades. IDG Communications, Inc., May 2018.

ABOUT THE AUTHORS

Florian Daniel is an associate professor at Politecnico di Milano, Italy. His research interests include service-oriented computing, blockchain, business process management and data science. Daniel received a PhD in information technology from Politecnico di Milano. Contact him at florian.daniel@polimi.it.

Luca Guida has a Master's degree (cum laude) in Computer Science and Engineering from Politecnico di Milano and Alta Scuola Politecnica. His research interests include blockchain, service-oriented computing, and spatial data analysis. Contact him at luca.guida@mail.polimi.it.