An Innovative Negotiations and Enactment Smart Contract-based Framework for on-line Sharing Economy Platforms

Layth Sliman, Benoit Charroux, Nazim Agoulmine

An Innovative Negotiations and Enactment Smart Contract-based Framework for on-line Sharing Economy Platforms

Layth Sliman[1], Benoit Charroux[1], and Nazim Agoulmine[2]

[1] EFREI Paris, Villejuif, France
layth.sliman@efrei.fr, Benoit.charroux@efrei.fr
[2] IBISC Laboratory, Evry University, Evry, France
nazim.agoulmine@ibisc.univ-evry.fr

Abstract

Despite the spread of sharing economy platforms, as the best of our knowledge, no on-line solution has been proposed to handle the negotiation of new agreements and contracts between the participants in such platforms, which entails losing major business opportunities due to a lack of negotiation frameworks enabling mutual business and legal agreements. This paper describes an innovative smart contract-based negotiation framework integrated into sharing economy platforms to enable dynamic negotiation and electronic signature of digital agreements between partners. The proposed framework itself is technology agnostic. It can be used with any distributed collaborative platform regardless of the used technologies (web service, blockchain, etc.). We have used smart contract system as a mean to initiate and submit negotiated calls for tenders to respond to a business opportunity by multiple actors. The implementation uses the Orcha language, a new high-level smart contract language, to validate the framework concepts.

1 Introduction

New economic models are emerging in the global markets, such as demand-driven economy, virtual marketplaces, Crowd Funding, Crowd Sourcing, etc. These models are boost-up by the spread of ICT technologies [8]. These economic and technological forces are producing more and more complex systems, where the interconnection between actors, the availability of trusted information, as well as cost and revenue sharing among the actors are the key factors to obtain sustainable and cost-effective businesses. These systems require a decentralized yet trusted negotiation framework so that mutual agreements that govern the collaboration and the usage of the shared resources can be constituted, enforced, and verified. This is the case, for example, of the food delivery services, which are increasingly offered by the producers and vendors by externalizing the delivery dynamically using service registries such as Uber Eats, JustEat, TooGoodToGo... etc. The principles and techniques that appears to suit these applications better are the "decentralized consensus" (e.g. Blockchain) that allow participants on a distributed network to reach a perfect agreement on a shared resource. Even in the case of very simple multiple-actors infrastructures, as in some food chains, the value created in the short and medium time-horizon is sufficient to justify the introduction of the new technology while reducing the need for trust among the different partners [7] .

In this context, smart contracts is emerging as the disruptive technology able to fuel such systems characterized by multiple actors strongly interconnected while maintaining a low level of mutual trust. Smart contracts are decentralized and autonomous computer programs that are executed on the distributed ledger upon predefined events. However, despite its rapid

development, this technology is still in its early stages of potential, it still presents some inherent defects which hiders its deployment in a factual project [4]. In particular, the possibility of integrating the business models, the choices, and the preferences of the different actors in the smart contract appears as a critical factor in democratizing more largely this technology.

The literature mainly considers the Business Process Modelling and Design for Blockchain-based solutions, which are then transformed into executable Smart Contracts [1] and [2]. Yet, the literature regarding the mutual definition by multiple actors of new smart contracts adapted to a particular need or business opportunities are quite limited [5]. In particular, to the best of our knowledge, no solutions have been proposed to handle the negotiation of new smart contracts to respond to business opportunities. To overcome this technical limitation, we describe an innovative smart contract-based negotiation mechanism that can be integrated into a blockchain. In particular, the presented solution introduces a smart contract system that is able to automatically launch and negotiate a call for tenders until completion (lifecycle management) to respond to a business opportunity by multiple actors.

This paper is organized as follows: in section 2, we present the concept of smart-contract as launched that is a cornerstone of the solution. Section 3 describes the proposed framework, which uses the smart contract language called Orcha[1]. After that, we present the implemented proof of concept system in section 4. Finally, we conclude and propose some perspectives to our work.

2 Smart Contracts

Smart contracts are programs coded with a programming language and executed in a runtime environment on a decentralized consensus system i.e. blockchain.

In the following, we briefly illustrate the smart contract programming and runtime environment focusing on Ethereum [3], a blockchain platform that was the first to introduce the smart contract concept.

Ethereum is a platform that intends to make a programming universe for the development, deployment, and execution of smart contracts for decentralized organizations over the blockchain. Ethereum integrates a Turing-complete programming language, called Solidity [3]. Solidity contains a set of instructions that enable arbitrary management of transactional states [6]. Solidity smart contracts are compiled into bytecode and encapsulated in Ethereum Virtual Machine (EVM). This later is intended to serve as a runtime environment for Solidity-based smart contracts. It focuses on providing a decentralized implement self-enforced smart contracts execution environment.

Finally, it is worth mentioning that each Ethereum node in the blockchain runs and maintains its own EVM implementation. EVM has been implemented in Python, Ruby, C ++, and some other programming languages [6].

Another language was specified by the Ethereum organization to respond to the Python language called Serpent. It is an Ethereum smart contract language that is close to Python. It is designed to encompass the benefits of Python in its simplicity, minimalism, and dynamic typing. When building the executable smart contract, serpent code is first compiled into LLL and then into bytecode for the EVM. The LLL name is diminutive to Low-Level Lisp-like Language. LLL refers to a language similar to Assembly that came to add a low-level layer into the EVM. The language adopts the syntax of Lisp. It is used when there is a need to deal

[1] https://github.com/orchaland/orchalang/tree/master/orchalang-spring-boot-autoconfigure/src/main/java/orcha/lang

with particular problems that are, by nature, low-level, e.g., require direct access to memory or storage.

The business logic that governs collaboration in a blockchain is supposed to be handled using smart contracts. However, the languages used to write smart contracts lack the elements necessary to negotiate dynamic business collaborations. This is due to the absence of interactions between the contractors (human and software) essential to defining collaborative business processes' functional and non-functional properties. Furthermore, current smart contract languages are very technical and do not incorporate business semantics, leading to a non-uniform interpretation of the smart contract by the different stakeholders. Consequently, it is crucial to define a new collaboration language tailored to the definition of smart contracts. Such a language should handle the complexity of the interactions between the different parties involved in a collaboration.

3 Framework Description

In the following, we introduce a new framework that helps construct and agree on a smart contract (called here contractualization process). We use also a novel high-level collaboration oriented smart contract language called Orcha [2] to validate the framework.

3.1 Orcha Language

Orcha is a business process modelling, deployment and coordination language developed by our team. It includes simple instructions that describes in a generic way business process activities. Orcha programs include a reduced number of instructions (Compute, Receive/From, Send/To, Condition, and When) represented by abstract syntax trees so that analysers can efficiently process them. Orcha contains a small and simple set of instructions designed to describe the answer of the essential questions around a collaborative process:

Who is involved? How an actor receives the needs to do the assigned tasks? When (timewise and in which order) to do the task? What to do? With whom? On what? What external events should be taken into account? To whom, when and how produced events are sent?

The Orcha programs are event-based. They are triggered and run by events to manage the interactions between business processes human and virtual activities. They can be customized to any business field, i.e., the user according to their own business terminology can define the semantic of the instructions.

Rather than handling the specifications of individual tasks, Orcha allows to specify the data exchanged between during the coordination of activities. In that way, Orcha programs remain independent from any underlying technical implementations of tasks. In other words, in Orcha, there is a clear separation between the collaborative process (described in Orcha language) and the individual business tasks (that can be implemented in any other language). That is, Compute instruction in Orcha programs calls business services that implement individual business tasks.

An Orcha program describes eventually a business process, i.e., human actors, applications,

[2] https://github.com/orchaland/orchalang/tree/master/orchalang-spring-boot-autoconfigure/src/main/java/orcha/lang

and devices exchanging messages and coordinates their activities (Figure 1).

Orcha uses business terms to express the business process. For instance, a process to handle a passenger language delivery procedure in an airport could be:

receive passport from passenger
controlIdentity with passport.photo
receive luggage from agent
scanLuggage with luggage.value
when "scanLuggage fails and controlIdentity terminates"
alertAuthorities with controlIdentity.result

Business processes written in Orcha are executable programs. Consequently, one needs only to configure its Orcha program and simply run it to drive its business. The configuration defines the input and output data sources for Receive and Send instructions; for instance, (*receive order from customerBase*) and (*send order to customer by eMail*) as well as the service to be activated by the Compute instruction is running (compute service with...). Orcha programs perfectly match Smart Contract requirements in that they are event driven, decentralized and, portable (executed in their own containers). Orcha enables services applications and IoT devices integration. For instance, when you write: *receive passport from passenger*, data for the passenger can come from Sensors, SQL and NoSQL databases or files. Similarly, the control service in *controlIdentity with passport.photo* can be a remote Service or a local application. A Smart Contract written in Orcha is compiled using the following steps: preprocessing, lexical analysis, syntax analysis, grammatical analysis, post-processing, linkage, and output generation (generate a Spring Integration Java program).

3.2 Contractualization of smart-contracts

In this section, we describe a contractualization framework that enables collaborators to dynamically create a smart contract to quickly respond to a business opportunity. In this framework, collaborators submit and respond to tenders via an IHM that allows to specify the needs. As a result, a smart contract representing the collaboration process is dynamically created. The framework is described in Figure 1. As highlighted, in the framework:

- A customer defines its needs and specifies the desired outcomes.
- The customer sends a Call for Tender (CfT) to the contract ledger workplace.
- Providers can retrieve the CfT and formulate a response as a Tender Proposal (TP),
- TP is sent to the customer.
- Customer retrieves the TP and either;
- It validates and diffuses it in the blockchain or;
- It can re-formulate a new CfT based on the received TP. In this case;
- The process restarts again until an agreement is reached.

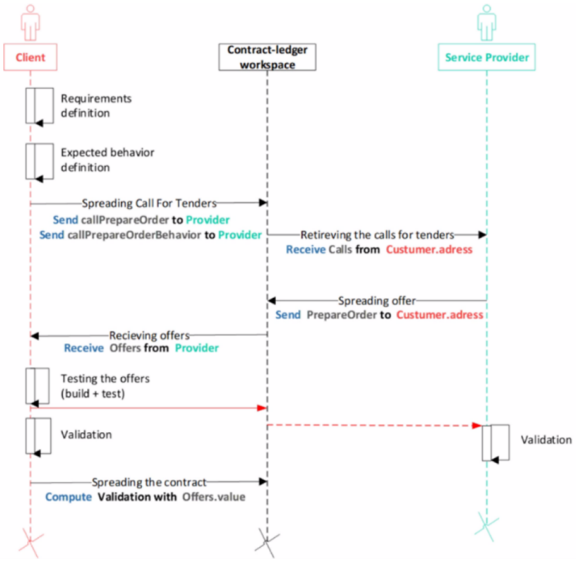


Figure 1: Sequence Diagram Describing the Contractualization Process

The obtained smart contract is signed by all stakeholders involved in the collaboration to confirm and maintain an inviolable record of the collaborators' contractualization. It is then submitted to the blockchain to be validated via the consensus mechanism. As a result of the consensus, if the smart contract is approved, it is added to the blockchain as a definitive transaction.

In the end, according to the conditions agreed by the collaborators, the smart contract is executed. Its execution creates new transactions that must be added to the blockchain by a consensus mechanism.

It is worth mentioning that the contractualization process itself is represented as a smart contract that uses Orcha as a language to specify the exchanging calls and responses of tenders. To this end, an interactive Shell interface for users to launch a call for tenders or respond to tenders is defined.

3.3 Proof of Concept Implementation

In this section we will present a proof of concept (PoC) that demonstrates this framework's feasibility. The PoC is implemented on the Git version control system. Git was selected because of its similarity to blockchain in terms of versioning mechanism based on hash functions, its distributed storage, and its pseudo decentralization.

The PoC considers a simple collaboration case where a client and a set of sub-contractors can interact to contract a service (as highlighted in Figure 2).

In the following, we explain the business logic implemented by presenting Orcha commands customized to cover the exchanges necessary for contractualization and the technical architecture that allowed us to make the command shell.

To start a collaboration simulation, a client who needs a service must create a git directory with public access as a distributed Smart-Contract registry. He/she then enters the address of this directory in the contracting system. From this moment, the client can issue a call for tenders. A call for tenders is a file that includes the customer's needs as well as the expected

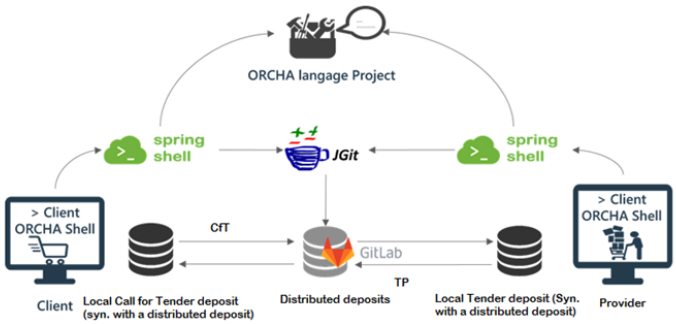


Figure 2: Technical architecture of the proof of concept

behavior of the service providers' responses. The file, written in Orcha, is completed with additional configuration files and a set of input/output data representing the desired service outcome.

Service providers must respond to a request for a proposal that interests them, create a branch on the collaboration's directory, and deposit the Orcha program of their offer. Similarly, as for the call for tenders, the program is accompanied by the configuration and data files.

After analyzing and testing the various response received, the client selects one or more offers. The customer will have, at this time, all the necessary information to write a complete smart-contract that would correctly govern future collaborations.

If an offer does not fully match the client's preferences, the client may continue a back-and-forth exchanging with the supplier to negotiate the terms.

A) **Shell orders made for the contractualization phase**

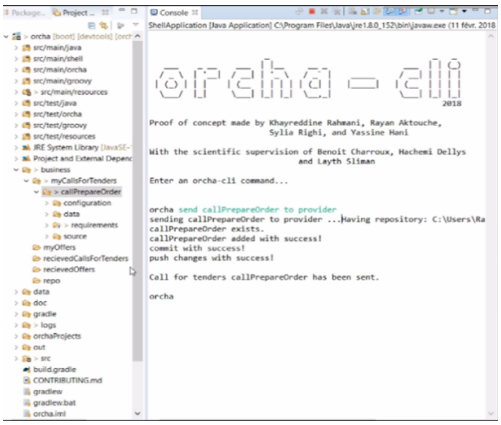
The distributed registry of each user's smart-contract contains the folder named "Business" that includes the following main folders:

- "myCallsForTenders": intended to include the calls for tenders sent by the customer;
- "myOffers": intended to include the offers provided by the service provider;
- "receivedCallsForTenders": If the user is a service provider, he or she will receive customer calls for tenders on this folder.
- "receivedOffers": If the user is a customer who has issued the CfT, he will receive the service provider response files on this folder.

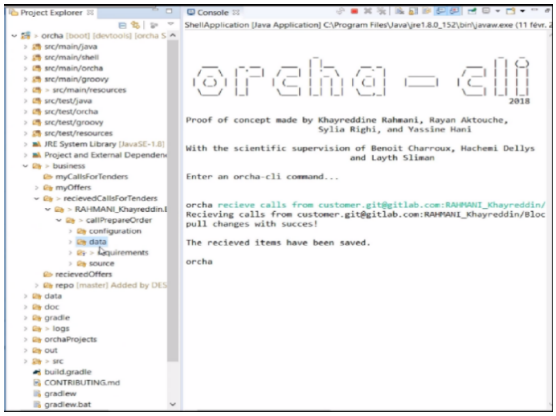
In the following, we explain the different commands that we defined using Orcha language to enable a customer to interact with providers during the contractualization process:

i. **Command 01 : Distribution of a call for tenders - customer order**
Orcha > send callForTender to providers.
This command allows a client to broadcast the "call for Tender" call for tender written in ORCHA language, which it has deposited in the "my Calls For Tenders" folder.

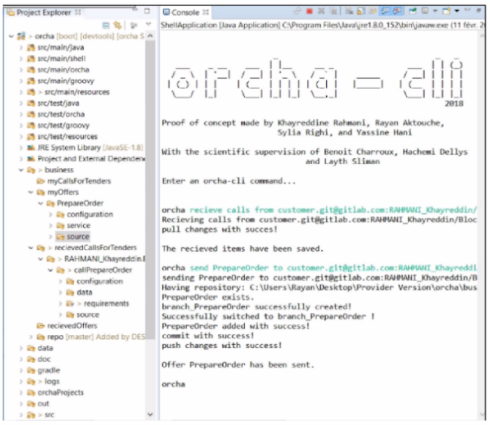
ii. **Command 02: Reception of a Request for Proposal - Service Provider Order**
Orcha > recieve calls from customer.customer@



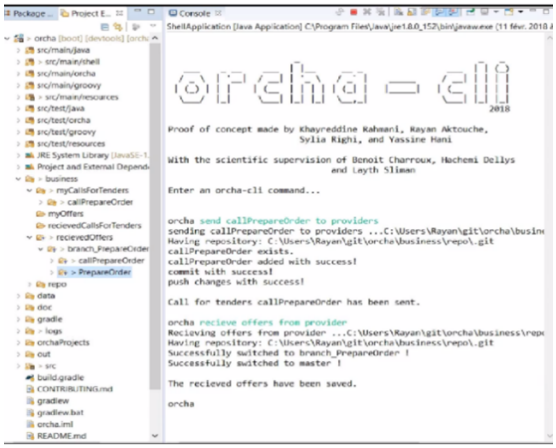
(a) Sending Call for Tenders



(b) Receiving Call for Tenders



(c) Call for Tenders response



(d) Call for Tenders Proposal acceptance

Figure 3: Contractualization framework implementation

This command allows a service provider to receive calls from the customer with the address "customer @" on his local copy of the distributed registry.

iii. **Command 03: Sending an offer - order service provider**

Orcha > send offerSP1 to customer.customer@

This order allows a service provider to offer their "offerSP1" offer to the customer with the address "customer @"

iv. **Command 04: tenders receiption - sales order**

Orcha > recieve offers from providers

This order allows a customer to receive the various "offer" offers offered by service providers "providers"

v. **Command 05 : Validation of an offer - customer order**

Orcha > send validation to providers.branchName

This command allows a customer to validate an offer he has received from a service provider on the branchName branch

B) **Technical architecture of proof of concept of contractualization**

The proof of concept is structured in the form of a client application based on shell commands orchestrating the different exchanges that perform the contractualization (Figure 2). The implementation of shell commands is done using SpringShell. Behind these commands, the sending and receiving operations are done by the encapsulation of Git functions, called from the Java environment using the JGit library, which acts on Gitlab hosted repositories [3] (see Figures 3).

4 Conclusions

In this paper, we introduced a new framework for smart contract negotiation. The framework follows Call for Tenders' business logic. It enables customers and providers to settle smart contracts in a negotiated way so that they can quickly respond to business opportunities. The framework has been validated using Orcha Language, a new high-level smart contract language. A Proof of Concept (PoC) has been implemented to assess the feasibility of the framework. The PoC used Git as the underlying platform. This latter has been chosen due to its similarity to blockchain logic. The PoC has shown that the concepts introduced by the framework are sound and permits it objectives. In future works, we aim to implement the framework using a real blockchain and a full smart contract lifecycle.

References

[1] Miguel Pincheira Caro, M. S. Ali, M. Vecchio, and R. Giaffreda. Blockchain-based traceability in agri-food supply chain management: A practical implementation. *2018 IoT Vertical and Topical Summit on Agriculture - Tuscany (IOT Tuscany)*, pages 1–4, 2018.

[2] Roberto Casado-Vara, Alfonso González Briones, Javier Prieto, and Juan Corchado Rodríguez. *Smart Contract for Monitoring and Control of Logistics Activities: Pharmaceutical Utilities Case Study*, pages 509–517. 06 2019.

[3] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. A programmer's guide to ethereum and serpent. *URL: https://mc2-umd. github. io/ethereumlab/docs/serpent_ tutorial. pdf.(2015).(Accessed May 06, 2016)*, pages 22–23, 2015.

[4] Deloitte. global blockchain survey. breaking blockchain open. 2018.

[5] Valentina Gatteschi, F. Lamberti, Claudio Demartini, Chiara Pranteda, and Victor Santamaria. To blockchain or not to blockchain: That is the question. *IT Professional*, 20:62–74, 03 2018.

[6] Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*, pages 520–535. Springer, 2017.

[7] Guido Perboli, Stefano Musso, and Mariangela Rosano. Blockchain in logistics and supply chain: A lean approach for designing real-world use cases. *IEEE Access*, 6:62018–62028, 2018.

[8] Roberto Tadei, Edoardo Fadda, Luca Gobbato, Guido Perboli, and Mariangela Rosano. An ict-based reference model for e-grocery in smart cities. In *International Conference on Smart Cities*, pages 22–31. Springer, 2016.

[3] visit https://www.youtube.com/watch?v=RSKe9oxuJfM