Blockchain for Trustworthy Coordination: A First Study with LINDA and Ethereum

Conference Paper · December 2018

DOI: 10.1109/WI.2018.000-9

3 authors:

Giovanni Ciatto
University of Bologna
39 PUBLICATIONS **185** CITATIONS
SEE PROFILE

Stefano Mariani
Università degli Studi di Modena e Reggio Emilia
82 PUBLICATIONS **433** CITATIONS
SEE PROFILE

Andrea Omicini
University of Bologna
478 PUBLICATIONS **7,581** CITATIONS
SEE PROFILE

2018 IEEE/WIC/ACM International Conference on Web Intelligence (WI)

Blockchain for Trustworthy Coordination: A First Study with LINDA and Ethereum

1st Giovanni Ciatto
[0000-0002-1841-8996]
ALMA MATER STUDIORUM
Università di Bologna
Cesena, Italy
giovanni.ciatto@unibo.it

2nd Stefano Mariani
[0000-0001-8921-8150]
Università degli Studi di
Modena e Reggio Emilia
Reggio Emilia, Italy
stefano.mariani@unimore.it

3rd Andrea Omicini
[0000-0002-6655-3869]
ALMA MATER STUDIORUM
Università di Bologna
Bologna, Italy
andrea.omicini@unibo.it

***Abstract*—Blockchain technologies are rapidly gaining attention in the multi-agent systems (MAS) community to face critical issues such as trust, secured communications, and data consistency. In particular, the notion of *smart contract* can be exploited to deploy trustworthy computations automatically executed by the network in a consistent way. MAS *coordination* – modelling and engineering of agents interaction in a MAS – thus represents an appealing application field for smart contracts, potentially enabling fully-decentralised, trustworthy coordination. Along this line, we focus on the Ethereum blockchain technology, map it onto LINDA tuple-based coordination model, and discuss two proof-of-concept implementations of LINDA on Ethereum. We hence demonstrate conceptual and technical feasibility of blockchain-based coordination in MAS, while emphasising issues of applying the blockchain beyond accountability and identity management.**

***Index Terms*—blockchain, smart contracts, MAS, coordination, LINDA, Ethereum, economy of coordination**

I. INTRODUCTION

The term *blockchain* is used to denote a heterogeneous number of technologies providing a novel way of dealing, essentially, with *distributed asset tracking* or *identity management*. The most prominent use case is represented by cryptocurrencies, where users' money is the asset to keep track of, and identity management is, therefore, crucial.

Blockchain technologies (BCTs) represent a novel approach to *secure decentralisation* of both *data* and *control* in distributed systems: the former is achieved by exploiting consensus protocols to (eventually) consistently replicate data among nodes of the system; the latter is supported by enabling autonomous computations to be triggered and regulated by the blockchain itself. To provide these features, BCTs effectively combine several results from cryptography, distributed consensus, game theory, and state machine replication. As a consequence, BCTs allow well-known problems of distributed systems [2] to be dealt with—i.e. CAP [12], Byzantine Generals [16], Sybil attacks [6].

This is why BCTs have recently gained attention from the *multi-agent systems* (MAS) community: trust, secured communications, and data consistency are critical issues for MAS models and technologies. BCTs are seen as a straightforward way for injecting such features into MAS.

BCTs are also appealing from the more specific perspective of *coordination models and languages* [22]. A coordination model is "a framework in which the interaction of active and independent entities called agents can be expressed" [3], whereas a coordination language provides operations to *synchronise* these interactions. Within this scope, BCTs provide highly-desirable properties such as *total ordering* of events, data *consistency*, *accountability* of actions, *identity management*, and *fault tolerance*. Also, they present analogies with *tuple-based* coordination models, featuring data repositories called *tuple spaces* work as *shared blackboards* where interacting agents put information chunks called *tuples* and synchronise activities based upon their availability.

Under this perspective, we advocate that a promising basic brick for BCT-based coordination is represented by *smart contracts* [30], that is, automatic and trusted activities executed on the blockchain itself.

A. Contribution

In this paper we explore how to ground *trustworthy, decentralised coordination* in MAS upon BCTs, by taking the LINDA model [11] – the most well-studied and influential one [5] – and the Ethereum BCT [8] as reference, while adopting the perspective of *coordination as a service* (CaaS) [33]. In particular, our contribution is threefold:

1) a *conceptual framework* mapping LINDA abstractions onto Ethereum
2) two *proof-of-concept* implementations of LINDA on top of Ethereum
3) *insights* on the issues and opportunities arising when doing the above, there including a definition of the *economical dimension* of coordination

B. Outline

Accordingly, the remainder of the paper is organised as follows: Section II describes Ethereum abstractions and mechanisms necessary to understand Section IV; Section III recalls the archetypal LINDA model for tuple-based coordination; Section IV describes the proposed conceptual framework for BCT-based coordination and its proof-of-concept implementations; Section V discusses the general issues arising when

978-1-5386-7325-6/18/$31.00 ©2018 IEEE
DOI 10.1109/WI.2018.000-9

mixing the Blockchain and coordination; Section VI concludes the paper.

II. Ethereum

The Ethereum blockchain consists of a peer-to-peer network of nodes enacting a *consensus protocol* that lets them globally (and logically) behave as a single *state machine*. This is why some authors consider the blockchain as a novel approach to State Machine Replication [27]: here, the state machine is an interpreter executing the smart contracts deployed by end users, and consensus ensures that all replicas evolve in the exact same way.

From the application standpoint, smart contracts encapsulate arbitrary and stateful functionalities, like keeping track of end users' balance—i.e., in terms of owned money. On purpose, they have their own state which may change as they are triggered by *transactions*, published by end users and directed towards a particular smart contract (along with all its replicas).

Execution of transactions, as well as their chronological ordering, is responsibility of special nodes of the blockchain called *miners*, which commit the results to *blocks* of data linked by hash chains, following the principles described in [13]. This process creates a hard-to-tamper sequence of blocks – hence, the name *block*chain – tracking the history of system evolution. Indeed, the interest around the blockchain lies essentially in its capability of maintaining a consistent shared state between mutually untrusted parties.

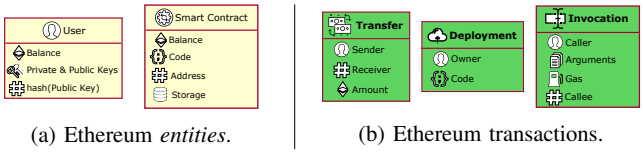
In the following, we describe the abstractions upon which we develop the proof-of-concepts (Section IV).

A. Entities & accounts

The state of the system conceptually consists of a map associating *entity identifiers* (essentially, network addresses) to *accounts*. Entities can be of two sorts (Fig. 1a): end users and smart contracts. In both cases the corresponding account consists of a data structure containing a *balance* – i.e., a counter representing the amount of ETH (*Ether*, the Ethereum currency) owned – and a secured storage area containing arbitrary user data. In the case of a smart contract, the data structure also exposes a field containing its source code.

B. Transactions

Users may publish their *transactions* (essentially, messages) at any time. Publishing triggers a gossiping algorithm eventually spreading information to all the participants to the consensus protocol. Transactions are of three sorts (Fig. 1b): *deployment* of a smart contract, value *transfers*, or *invocations* of smart contracts. In any case, the transaction is composed of:



(a) Ethereum *entities*. (b) Ethereum transactions.

Fig. 1: Main components of the Ethereum blockchain.

(i) the destination address, *(ii)* the amount of ETH (possibly 0) to be transferred, *(iii)* the cryptographic signature of the transaction, and *(iv)* in case of value transfers and invocations, an input field for the smart contract; in case of deployment, the bytecode of a program that, if executed, produces the source code of the smart contract to be deployed.

C. Smart-contracts

Smart contracts are stateful, user-defined, reactive, immutable, trustable, and deterministic processes executing decentralised computations on the blockchain:

- **stateful**: each smart contract encapsulates its own state
- **user-defined**: any user may publish a smart contract
- **reactive**: only users may trigger a smart contract
- **immutable**: smart contracts code cannot change
- **trustable**: no entity can tamper with the specification of a published smart contract, no user can lie about its smart contract invocations, and the side effects (possibly) caused by computations are guaranteed to (eventually) produce a consistent change of the system state
- **deterministic**: each computation always provides the same outputs if given the same inputs, regardless of the actual execution node
- **decentralised**: there is no single, centralised coordinator governing the distributed execution of smart contracts, which happens concurrently

Computations are expressed by means of a *quasi*-Turing-complete language [34, Sec. 9] – meaning that "the computation is intrinsically bounded through a parameter [...] which limits the total amount of computation done" – thus they are virtually capable of implementing any computation. Smart contracts are objects in the OOP sense, which interact by means of *synchronous* method calls, whose control flow originates from the issuer of the transaction. As discussed in Section V, this limits their expressiveness w.r.t. coordination.

D. Consensus, miners, and blocks

The goal of the consensus protocol is to make *every* node in the blockchain participate in *validation* and *consistency check* of transactions. A transaction is valid if its signature proves it to be untampered, and it transfers an amount of ETH which is at most equal to the sender's balance. The consensus protocol makes an arbitrary number of nodes perform *exactly the same* state transition for exactly the same state machine – provided that the nodes agree on the starting state –, forcing them to perceive the published messages in the same order while preventing Byzantine faulty nodes [16] – that is, nodes deliberately lying to neighbours – from creating inconsistencies.

Miners are the nodes in charge of validation and consistency checking, as well as of including transactions in blocks. These track: *(i)* the cryptographic hash of the previous block; *(ii)* a timestamp; *(iii)* a list of (valid and consistent) transactions; *(iv)* the hash of the global state, obtained by orderly applying all transactions to the previous block; *(iv)* a Bloom filter enabling clients to find logs possibly published due to smart contract

execution—see "Logs & API" at the end of this section. Transactions triggering a faulty computation are included into blocks, but the side effects possibly provoked are not.

Agreement about ordering of transactions is achieved by means of the consensus mechanism known as *Proof-of-Work* (PoW) [15], [19], requiring miners to compete in solving a resource-intensive computational puzzle, and granting the right to publish the new block containing the validated transactions to the first succeeding. This mechanism is essential for BCT since it provides consistency and fault-tolerance despite decentralisation. Notice that the PoW mechanism, and in particular the GHOST protocol adopted [29], provides *probabilistic* consistency: inconsistencies can temporarily arise in the form of blockchain *forks*, yet the protocol takes care of making one eventually prevail on the others. In other words, the probability of a block B being inconsistent drops down to 0 exponentially with the amount N of blocks after B. In this context, N is also known as the amount of *confirmations* of B.

E. Miners & gas

A malicious miner may produce great damage, thus, the blockchain protocol assumes them to be *rational agents*, and promotes honesty by compensating their computational effort with the right to generate and claim cryptocurrency for each block successfully mined.

Miners are also in charge of executing smart contracts and deployment messages, thus, they may store and execute arbitrary code, which requires additional effort. To compensate and prevent Denial of Service attacks – i.e., by simply deploying an infinite loop – the Ethereum protocol requires users to pay some ETH, lately reclaimed by miners, for their transactions to be accepted—and the corresponding computation executed. More precisely, users must endow messages with a finite amount of *gas* (converted from user's ETHs) to be spent while executing the smart contract deployed. If the computation fails, for an error or the consumption of all the gas, its side effects are reverted—with no refund of ETH.

As discussed in Section V, the above economical aspects rise challenging questions related to, i.e., associating a price to coordination primitives.

F. Logs & API

The Ethereum bytecode provides instructions to publish *logs* from smart contracts to represent occurrence of some *events* to be stored within blocks. This aims at letting off-chain clients inspect the blockchain. A number of high-level languages (along with their compilers to EVM bytecode) have been produced by the Ethereum community: the most widespread is Solidity [28], coming with a JavaScript-like syntax and resembling OO programming, where transactions are mapped to methods and member fields correspond to account data.

III. LINDA

LINDA [11] is the archetypal tuple-based coordination model, inspiring and influencing a huge number of coordination models and technologies throughout the years [5].

A. Main elements

The main elements of LINDA are *tuples*, *templates*, *tuple spaces*, and *communication primitives*. A tuple is a piece of information represented according to a well-defined *tuple language*, specifying the structure of admissible tuples. A template is a concise way of representing a set of tuples: it consists of a pattern, represented according to a particular *template language*, which may be matched by several tuples. A tuple space is a repository where tuples may be inserted, observed, or withdrawn by an arbitrary number of agents willing to synchronise while being uncoupled in reference, space, and time. On purpose, a communication primitive is an operation provided to interacting entities to *synchronise* themselves upon tuples' insertion, observation, and consumption.

B. Main features

LINDA owes its success in the field of MAS coordination [24] to a few peculiar features: *(i) generative communication*, that is, tuples existing independently of the agents who produced them (decoupling in space and time); *(ii) associative access*, namely, agents can access (i.e., observe or withdraw) the tuples stored in a tuple space by simply specifying a template, without the need of knowing the tuple "address" neither its "name" (reference uncoupling); and *(iii) suspensive semantics*, that is, agents' attempts of accessing a tuple matching a particular template are suspended until a tuple of such a sort actually exists.

LINDA provides three communication primitives: `out` to insert a tuple in a tuple space, `in` to withdraw one, `rd` to read one. Despite their simplicity, such primitives are expressive enough to cope with several common interaction patterns [11]. Suspensive semantics, in particular, is the cornerstone of the coordination mechanism proposed by LINDA, since it deals with synchronisation: whereas the `out` primitive always puts a tuple in the tuple space, `in` and `rd` *attempt* to get one based on a provided tuple template. If a tuple *matching* the template is found, it is returned to the caller agent that can continue execution; otherwise, the caller agent is *suspended* until a matching tuple becomes available.

C. Architecture

In the following we refer to the LINDA model and architecture proposed in [33] as an example of *coordination as a service*, where the issues arising when tuple spaces are provided in a distributed fashion are pointed out—like, for instance, the need to split LINDA operations in two phases (namely, the *invocation* of an operation, and its *completion* [21]) to support their suspensive semantics. There, a coordinated system is seen as the composition of three spaces: *(i)* the *coordinated space*, composed by those entities to which coordination services are provided, namely the *coordinated entities*; *(ii)* the *coordination space*, composed by the (possibly many) coordination media, i.e., one or more tuple spaces; and *(iii)* the *interaction space*, where the pending invocations and completions – possibly reifyied as *requests* and *response* messages, respectively – are at rest, waiting to be consumed.

D. Technology

Several variants of LINDA have been proposed throughout the years, either extending the set of communication primitives, adding features such as mobility or access control [18], [20], enabling distribution of multiple tuple spaces on a network of interconnected computers [10], [17], and much more [23]. Nevertheless, only a few have been developed as a *technology* [5], and among these even fewer have been implemented in a decentralised way [25], [26]. Making a coordination model such as LINDA actually work on top of BCT would deliver a number of benefits, among which:

- a *fault tolerant* implementation of tuple spaces, with tuples replicated on each machine and consistency handled by the consensus protocol
- a *trustable immutable* ledger of agents' interactions, where no one can lie thus malicious social behaviour is easily spotted
- *secure*, *consistent*, and *accountable* interactions within MAS, where coordination among agents is fully decentralised and does not rely on a trusted third-party (a "controller" agent) to spot and tolerate malicious behaviour

IV. LINDA ON ETHEREUM

The discussion presented in this section originates from the mapping between the LINDA abstractions described in [33] and the Ethereum ones, as depicted in TABLE I. The mapping aims at providing a sound conceptual basis upon which our vision of blockchain-based coordination can be fruitfully designed and built. Accordingly, it aims at answering basic but necessary architectural questions such as: Which blockchain abstraction will serve as a coordination media? Where to store tuples? Is it possible to mimic LINDA suspensive semantics?

The first step is quite natural for both implementations: Ethereum end users – which may be computer programs, not only humans – are LINDA coordinated entities. We then define a *class* of smart contracts (`TupleSpace`) whose instances are coodination media exposing a LINDA-like interface. The coordination space is therefore composed by the set of smart contracts instantiating the `TupleSpace` class. Coordinated entities can request execution of coordination operations by publishing regular Ethereum invocation transactions, specifying a `TupleSpace` instance as the intended recipient and the invoked operation as payload. Such transactions would conceptually "rest on the blockchain" until miners execute the corresponding `TupleSpace` smart contract. Then, in the first implementation ("Writers Pay"), the coordination media eventually publishes the completion of invocations by publishing a *log* "on the blockchain", representing the completion event; whereas in the second implementation ("Readers Pay"), an explicit operation checks wether the result of the invocation is available (`getResult`). In either case, the interaction space coincides with the structure collecting both issued requests and published responses, that is, the blockchain itself.

In what follows, we thoroughly describe the design and implementation choices made for the two *contract spaces* prototypes we built, while highlighting the impact that such choices have on expressiveness and faithfulness of the proposed Ethereum-based implementation of LINDA.

A. Contract spaces: "Writers Pay"

In this implementation, contract spaces are called "Writers Pay" because tuples producers are the one who pay the cost of coordination.

On tuple spaces & primitives: Tuple spaces can be implemented on top of Ethereum smart contracts by storing tuples in a dedicated `tupleSpace` field, as a key-value map associating tuples to a corresponding `TupleInfo` record with two fields: `quantity`, as the amount of identical copies of the tuple, and `queue`, tracking the list of pending requests waiting for the tuple. Each pending request is a record of type `WaitingPeer`, storing the address of the requesting process and the type of its request (either `READ` or `TAKE`, see below).

Contract spaces defined this way support three operations (the communication primitives): `Write` (for LINDA `out`), `Read` (for `rd`), and `Take` (for `in`), which any end user may invoke by signing and publishing an invocation transaction containing the operation name, its actual argument (i.e. either the tuple or the template), and the address of the `TupleSpace` smart contract. Being this an ordinary transaction in the eyes of the Ethereum network, it must be endowed with some gas. Technically, the necessary amount of cash is automatically and transparently computed by the end user's client software. In fact, client libraries, such as `web3js`, usually come with some built-in method to estimate the gas consumption of a transaction, e.g. [9]. In any case, the general rule for out-of-gas exceptions applies: side effects are reverted as if the operation was never invoked—except for wasted amount of ETH equivalent to the gas provided.

On primitives completion (events): The completion of an operation requires publication of a block containing the invoked transaction and the corresponding *event notification* to be logged on the blockchain, too. Solidity comes equipped with a built-in *event* abstraction, enabling developers to define the structure (namely, name + formal parameters) of arbitrary event notifications, and to rise them accordingly (name + actual parameters).

Events are reified as Ethereum logs. "Writers Pay" contract spaces support (generation of) `TupleWritten`, `TupleRead`, and `TupleTaken` event notifications, representing completion of the corresponding coordination operation. A `TupleWritten` notification conveys information about the tuple inserted and the address of the inserting entity. Similarly, `TupleRead` and `TupleTaken` notifications track the tuple retrieved and the address of the entity accessing it.

As clarified in the last paragraph below, decomposition in "invocation" and "completion" stages is required to emulate LINDA suspensive semantics. Since computations in Ethereum are *atomic* and *transactional*, the event abstraction is one way to decouple invocation and completion so as to emulate suspension—the other one is discussed in the following section, as part of the alternative implementation

Coordination as a service (LINDA)	Ethereum *(Writers pay)*	Ethereum *(Readers pay)*
Coordinated entity	Off-chain end user	
Coordinated space	The set of end users	
Coordination medium (tuple space)	`TupleSpace` smart contracts	
Coordination space	The set of currently deployed `TupleSpace` smart contracts	
Request (operation invocation)	`Take`, `Read` or `Write` transaction invocation	
Response (operation completion)	Log/event within a confirmed block	`GetResult` transaction invocation
Interaction space	The blockchain itself	

TABLE I: Conceptual mapping between LINDA and Ethereum.

therein described. Thus, in the case where a matching tuple is missing, the `Read` or `Take` operations are split over two transactions: the one published by the operation requestor simply records that a new operation is pending, then, `TupleRead` or `TupleTaken` events are raised – i.e., the operation is completed – within the transaction of an entity performing a `Write` operation which provides the missing tuple. This leads to interesting implications discussed in Section V—namely, tuples producers incur the cost of coordination, hence the name "Writers Pay".

On primitives invocation (transactions): Whenever an agent publishes a transaction invoking the `Take` operation with template `"tt"`, the behaviour of the smart contract depends on the value of field `tupleSpace["tt"].quantity`:

- if it is greater than zero, then it is decremented, and a `TupleTaken("tt", msg.sender)` event is raised
- otherwise, a new `WaitingPeer(msg.sender, SuspensiveOperations.TAKE)` object is put into the `tupleSpace["tt"].queue` list

In both cases, `msg.sender` enables developers to refer to the agent invoking the operation—that is, the one having published the transaction. The `Read` case is the same, except that `tupleSpace["tt"].quantity` is not decremented, `TupleRead` is the event to be raised, and `SuspensiveOperations.READ` is used within the `WaitingPeer` object.

Whenever an agent publishes a transaction invoking the `Write` operation with tuple `"t"`, the smart contract increases `tupleSpace["t"].quantity` and then, for each `WaitingPeer` item `w` in `tupleSpace["t"].queue`:

- if `w.operation` equals `SuspensiveOperations.READ`, then a `TupleRead("t", msg.sender)` event is raised
- otherwise a `TupleTaken("t", msg.sender)` event is raised, and `tupleSpace["t"].quantity` is consequently decreased

Anyway, `w` is removed from `tupleSpace["t"].queue`. Queue iteration ends if `tupleSpace["t"].quantity` reaches 0.

The Solidity source code of the contract spaces proof-of-concept just described is available from a dedicated GitLab repository [7].

B. Contract spaces: "Readers Pay"

In this alternative implementation, contract spaces are called "readers pay" because tuples consumers are the one who pay the cost of coordination. As for previous section, this aspect is extensively discussed in Section V.

On tuple spaces & primitives: We define a novel *class* for coordination media, namely $\texttt{TupleSpace}_{ER}$ (*E*xplicit *R*esult), which is a variant of the `TupleSpace` class described above. Instances of the $\texttt{TupleSpace}_{ER}$ class expose the same LINDA-like interface – that is, `Write`, `Read`, and `Take` –, plus a `GetResult` operation to *explicitly* retrieve the completion of a pending request, if any. Indeed, the conceptual mapping from TABLE I is almost unaffected, except for the "response" entry: operation completions are modelled here as invocation transaction too, similarly to invocations (requests). An end user may *request* a getter operation with template `"tt"` by issuing a `Read("tt")` or `Take("tt")` invocation transaction over a contract space, and she can later check for its *completion* by issuing a `GetResult()` operation, which may return a failure or the tuple found. In other words, coordinated entities are now expected to perform a "busy wait" on completion of their coordination operations—until an invocation to `GetResult()` provides a result.

On primitives completion (`getResult`): As in the case of "Writers Pay" contract spaces, tuple spaces are smart contracts storing tuples, in a `tupleSpace` field, as a key-value map associating tuples to the corresponding `TupleInfo` record, which in this case just contains a single entry: `quantity`. Pending requests are now tracked by a single key-value map, namely `pendingOperations`, associating entities IDs to a `PendingOperations` record made of a `queue` of `PendingOp` records. Each `PendingOp` record keeps track of *(i)* `operation`, i.e., the `SuspensiveOperations` which is currently pending, *(ii)* `template`, i.e, the tuples the operation is currently waiting for, and *(iii)* `alreadySatisfied`, i.e., a boolean value stating whether the operation has already been served or not.

The effect of invoking a `Read` or `Take` operation is to append a new `PendingOp` entry to the `queue` corresponding to that user. If the tuple requested by `Read` or `Take` operations is available, the corresponding `PendingOp` records are marked as `alreadySatisfied`—and the tuple `quantity` is decreased, in case of `Take`. The effect of invoking a

`GetResult` operation is to remove any satisfied or satisfiable `PendingOp` entry from that user's `queue`. Finally, the effect of a `Write` operation is simply to increase the corresponding tuple's `quantity` field.

On primitives invocation (transactions): Every time an operation is requested by a user `caller` invoking the `Take` method over a template `"tt"`, a new `PendingOp` entry `op` is appended to `pendingOperations[caller].queue` in order to keep track of it. In case `tupleSpace["tt"].quantity > 0`, field `op.alreadySatisfied` is set to `true` and `tupleSpace["tt"].quantity` is decreased, since the request can already be satisfied given the current state of the contract space. The `Read` operation has a similar behaviour, except that `tupleSpace["tt"].quantity` is never decreased; whereas the `Write("t")` operation simply increases `tupleSpace["t"].quantity`.

Operation `GetResult` works as follows. Upon invocation, field `pendingOperations[caller].queue` is scanned for a `PendingOp` record which is either already satisfied or satisfiable—that is, the corresponding tuple's `quantity` is strictly positive. If some satisfied or satisfiable pending operation is found, say `pending`, it is removed from `pendingOperations[caller].queue` and `pending.template` is successfully returned to the calling user. In case `pending` is not currently satisfied and the corresponding `operation` is `SuspensiveOperations.TAKE`, then the operation is satisfied on the fly, i.e., `tupleSpace[pending.template].quantity` is decreased. Otherwise, if no satisfied or satisfiable pending operation is found, an empty tuple is returned to the calling user as failure.

Also the Solidity source code of the contract spaces proof-of-concept just described is available from a dedicated GitLab repository [7].

V. On blockchain-based Coordination: Discussion

Regardless of the many industries the blockchain is hyped to be disrupting [31] – from healthcare to insurance, from supply chain to IoT – the actual issues it is exploited to solve are always the same: *identity management* and *asset tracking*. The former deals with guaranteeing and enabling verification of the *identity* of a given entity, which can be a stakeholder, a user, a software component, a hardware device, or any other goods—e.g., food [32]. The latter deals with securely and safely keeping track of a given *asset* – which usually is money but can be anything, from legal documents to insurance policies – and of all the actions which involve moving its ownership. The foremost goal of both is to enable *accountability*—that is, the ability to track *who* did *what* in a secure and verifiable way.

We advocate that the real potential of BCTs would be to go *beyond* identity management and asset tracking, towards brand new use cases taking advantage of its peculiar features such as security, decentralisation of trust, fault tolerance, and consistency. In our case, this novel use case is represented by the *coordination* of distributed agents of an open MAS in a *fully-decentralised* way.

A. Beyond blockchain "comfort-zone"

The aim of the contract spaces proof-of-concept we propose is twofold. On the one hand, we argue that tuple-based coordination is a promising application for *permissionless public blockchains* such as Ethereum, taking full advantage of those properties highly desirable from the coordination standpoint – especially for *open* systems – such as inherent *decentralisation*, events *ordering*, eventual *consistency*, and *fault tolerance*. Then, our study shows the feasibility of the approach, and paves the way towards further well-grounded research efforts. On the other hand, we are interested in understanding the *practical expressiveness* of the novel computational approach represented by smart contracts – starting from Ethereum ones – w.r.t. their capability of serving as a basic brick for the design of brand new coordination models and technologies. Thus, we aim at stretching the blockchain out of its "comfort-zone" so as to early detect issues and opportunities. For instance, our study already raises the issue that:

- smart contracts are purely reactive, that is, the control flow of their computations always originates from off-chain entities
- these entities pay for the computation to be performed
- the cost of the computation is proportional to its computational complexity

This implies that coordination models designed on top of a BCT must take into account this sort of *economical* aspect and the reactive, synchronous nature of smart contracts. The following two sections discuss both issues.

B. Towards an economy of coordination

In the "Writers Pay" implementation of contract spaces the financial burden that entities have to bear for accessing information is constant, since the `Read` and `Take` operations have a $O(1)$ complexity: in fact, for both operations, the employed algorithm either raises an event, in case the read/taken tuple is available, or adds an operation to the list of pending ones otherwise. In both cases, the amount of computational steps to be performed is constant. Conversely, the cost that entities performing a `Write` operation incur is proportional to the amount n of pending requests waiting for the tuple. Thus, `Write` complexity is $O(n)$: in the worst case, it is required to iterate over all the pending requests.

So, for "Writers Pay" contract spaces, *the higher the demand for a tuple, the higher the cost in providing it*. This resembles the way in which ads are paid, where the cost of the ad is somehow proportional to the number of potential customers reached.

Symmetrically, in the "Readers Pay" implementation of contract spaces the financial burden for information production is constant, whereas the cost for information access is variable and potentially unbounded for the calling entity E. In fact, all three canonical operations – namely, `Write`, `Take`, and `Read` – have $O(1)$ complexity since they either increase the

`quantity` counter for a given tuple (`Write`) or simply append an operation to the queue of pending ones (`Take` or `Read`) for entity E. At the same time, the `GetResult` operation needs to iterate on the whole queue of pending operations for entity E (in the worst case), so it has $O(m)$ complexity, being m the amount of pending requests for E. Furthermore, an execution of the `GetResult` operation is not guaranteed to reduce the size of the aforementioned queue, e.g., because no pending operation can be satisfied. In the worst case, E is expected to invoke the `GetResult` operation l times, before it can complete the operation(s) it requested. This is why, in practice, we consider it to have $O(m \cdot l)$ worst case complexity.

Thus, for "Readers Pay" contract spaces, *the higher the need for a tuple, the higher the cost in providing it*. This resembles the way in which market works, where the cost of an item is somehow proportional to the demand for it.

Possibly, other different implementations could be conceived where financial aspects are balanced in other ways. Yet, the point here is: researchers and practitioners *can no longer ignore* the economical aspect inevitably bound to smart contracts. Therefore, w.r.t. to blockchain-based coordination, the *economy of coordination* is intended as the cost associated to communication and coordination primitives—in our case, according to the contract-spaces model.

C. On the control flow of smart contracts

An even more subtle aspect needs to be further discussed: as already said, smart contracts do not have their own control flow, but "borrow" that of off-chain entities to perform their computations. Hence, they are purely *reactive*. As discussed in [4], this is not simply an implementation detail of Solidity, but it is a trait hard-coded into the Ethereum operational semantics. This is a serious constraint when designing synchronisation mechanisms and coordination policies requiring to either schedule future computations or postpone them, since the delayed computation will not be able to start unless someone else's control flow becomes available.

This is what happens, for instance, with our `Read`/`Take` implementation when no available tuple is matching the requested template: the request is stored, then the control flow of the smart contract goes back to the off-chain entity who requested the operation. Only *if* and *when* some other user lends its own control flow to the blockchain again, either by issuing a `Write` operation for a tuple matching that template, or a `GetResult`, the aforementioned `Read`/`Take` operation may complete. If such an implementation detail may be negligible in other contexts, in blockchain-based coordination it is not: as we discussed in previous section, in fact, since every computation has an associated cost, establishing who has to pay for it – and possibly, the rewards for doing so – is crucial. Indeed, we provided two implementations of the contract space concept precisely to bring this issue to light, emphasising the impact that different technical choices have on the cost of coordination.

For the above reasons, we argue that a well-founded and thorough research activity aimed at pushing the blockchain well beyond its traditional use cases – identity management and asset tracking – should start exactly by accounting for the aforementioned economic and control flow related issues.

D. Blockchain as a basic brick for coordination

Given all the considerations made so far, there is evidence that the blockchain is a promising technology to look at when designing fault-tolerant and secured solutions for fully decentralised coordination in open systems:

- agents are uniquely identified as Ethereum accounts and their actions are always tracked and visible to others
- all the interactions are logged persistently on an attack-proof distributed ledger, thus their history is available at all times
- no central authority for trusting participants is needed

Nevertheless, we do not envision coordination mechanisms and policies designed and implemented *directly* on the blockchain, exploiting its API with no mediation layer in between. Instead, we foresee the blockchain used as the *backbone* of a distributed coordination infrastructure, as one of the basic layers of a coordination middleware, providing *vital services* such as identification of clients, privacy of communications, secured and ordered event log, tolerance to failures and malicious behaviours. As depicted in Fig. 2, communication and coordination services could then be built on top of this *secured event ledger*, keeping track of all the coordination-related events that the higher layers of the middleware should deal with.

We understand that Ethereum exposes some issues regarding scalability of the approach, for instance strong serialisation of transactions – thus to the events occurring within a MAS – which may be both a blessing – since it gives event ordering

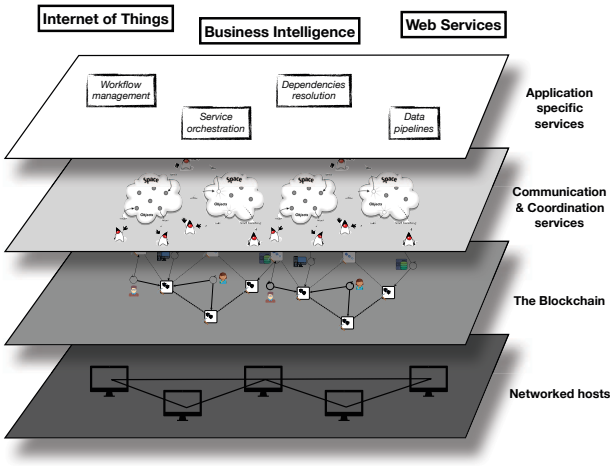


Fig. 2: A layered architecture for a coordination middleware: the blockchain serves as the *backbone* on top of which communication and coordination services are built.

in a distributed system – and a curse—especially for LINDA, which would otherwise allows for concurrent readings of the tuples. We also acknowledge the privacy concerns arising from a publicly available blockchain such as Ethereum. For these reasons we plan to reproduce our contract spaces on different BCTs like, for instance, HyperLedger Fabric (HLF), whose Execute-Order-Validate semantics and *permissioned* architecture [1] is expected to positively affect both concurrent execution of transactions and information visibility.

VI. CONCLUSION AND OUTLOOK

This paper pursues a twofold goal: *(i)* reporting about *feasibility* of implementing a LINDA-like tuple-based coordination service on top of a reference blockchain technology, and *(ii)* shedding some light on the most notable *issues* arising when doing so—e.g., the economical impact of performing coordination operations.

Accordingly, we show how LINDA can be implemented on Ethereum, focussing on how to *emulate* the suspensive semantics of LINDA communication primitives by leveraging Ethereum transactions and events. While doing so, we bring a few early issues to attention, concerning the cost of execution of a smart contract and a transaction, and the handling of multiple control flows.

The next steps we are planning for further advancing investigation of blockchain-based coordination are: *(i)* performing a comparison of different implementations of our contract space concept on top of different blockchain technologies, thus of different smart contracts implementations – for instance HLF [1] and Corda [14], – and *(ii)* defining a rigorous formalisation of the semantics behind different blockchain and smart contract models in terms of their potential coordination capabilities.

REFERENCES

[1] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M., Cocco, S.W., Yellick, J.: Hyperledger fabric: A distributed operating system for permissioned blockchains. In: 13th EuroSys Conference (EuroSys '18). ACM, New York, NY, USA (2018)

[2] Bashir, I.: Mastering Blockchain – Distributed ledgers, decentralization and smart contracts explained. Packt Publishing (2017)

[3] Ciancarini, P.: Coordination models and languages as software integrators. ACM Computing Surveys 28(2), 300–302 (Jun 1996)

[4] Ciatto, G., Calegari, R., Mariani, S., Denti, E., Omicini, A.: From the blockchain to logic programming and back: Research perspectives. In: WOA 2018 – 19th Workshop "From Objects to Agents". CEUR Workshop Proceedings (Jun 2018)

[5] Ciatto, G., Mariani, S., Omicini, A., Zambonelli, F., Louvel, M.: Twenty years of coordination technologies: State-of-the-art and perspectives. In: Di Marzo Serugendo, G., Loreti, M. (eds.) Coordination Models and Languages, Lecture Notes in Computer Science, vol. 10852, pp. 51–80. Springer (2018)

[6] Douceur, J.R.: The Sybil attack. In: 1st International Workshop on Peer-to-Peer Systems (IPTPS '01). pp. 251–260 (Jan 2002)

[7] Ether-Linda: Home. http://gitlab.com/das-lab/blockchain/ether-linda

[8] Ethereum: Home. http://www.ethereum.org

[9] Ethereum: Web3js library and the method for estimating transactions required gas. https://github.com/ethereum/wiki/wiki/JavaScript-API#web3ethestimategas

[10] Freeman, E., Arnold, K., Hupfer, S.: JavaSpaces Principles, Patterns, and Practice. Addison-Wesley Longman Ltd., Essex, UK (1999)

[11] Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems 7(1), 80–112 (Jan 1985)

[12] Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2), 51–59 (Jun 2002)

[13] Haber, S., Stornetta, W.S.: How to time-stamp a digital document. In: Menezes, A.J., Vanstone, S.A. (eds.) Advances in Cryptology-CRYPT0' 90. pp. 437–455. Springer (1991)

[14] Hearn, M.: Corda: A distributed ledger. http://docs.corda.net/_static/corda-technical-whitepaper.pdf (2016)

[15] Jakobsson, M., Juels, A.: Proofs of work and bread pudding protocols. In: Preneel, B. (ed.) Secure Information Networks, IFIPAICT, vol. 23, pp. 258–272. Springer (1999)

[16] Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Transactions on Programming Languages and Systems 4(3), 382–401 (Jul 1982)

[17] Louvel, M., Pacull, F.: LINC: A compact yet powerful coordination environment. In: Kühn, E., Pugliese, R. (eds.) Coordination Models and Languages, LNCS, vol. 8459, pp. 83–98. Springer (2014)

[18] Murphy, A.L., Picco, G.P., Roman, G.C.: LIME: A coordination model and middleware supporting mobility of hosts and agents. ACM Transactions on Software Engineering and Methodology (TOSEM) 15(3), 279–328 (Jul 2006)

[19] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008), http://bitcoin.org/bitcoin.pdf

[20] Nicola, R.D., Ferrari, G.L., Pugliese, R.: Klaim: a kernel language for agents interaction and mobility. IEEE Transactions on Software Engineering 24(5), 315–330 (May 1998)

[21] Omicini, A.: On the semantics of tuple-based coordination models. In: 1999 ACM Symposium on Applied Computing (SAC'99). pp. 175–182. ACM, New York, NY, USA (28 Feb – 2 Mar 1999)

[22] Omicini, A., Viroli, M.: Coordination models and languages: From parallel computing to self-organisation. The Knowledge Engineering Review 26(1), 53–59 (Mar 2011)

[23] Omicini, A., Zambonelli, F.: Coordination for Internet application development. Autonomous Agents and Multi-Agent Systems 2(3), 251–269 (Sep 1999)

[24] Omicini, A., Zambonelli, F., Klusch, M., Tolksdorf, R. (eds.): Coordination of Internet Agents: Models, Technologies, and Applications. Springer (Mar 2001)

[25] Papadopoulos, G.A.: Models and technologies for the coordination of Internet agents: A survey. In: Omicini et al. [24], chap. 2, pp. 25–56

[26] Rossi, D., Cabri, G., Denti, E.: Tuple-based technologies for coordination. In: Omicini et al. [24], chap. 4, pp. 83–109

[27] Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys (CSUR) 22(4), 299–319 (Dec 1990)

[28] Solidity: Home. http://solidity.readthedocs.io/

[29] Sompolinsky, Y., Zohar, A.: Accelerating Bitcoin's transaction processing. fast money grows on trees, not chains. Report 2013/881, Cryptology ePrint Archive (2013), http://eprint.iacr.org/2013/881

[30] Szabo, N.: Formalizing and securing relationships on public networks. First Monday 2(9) (1 Sep 1997), http://ojphi.org/ojs/index.php/fm/article/view/548/469

[31] Tapscott, D., Tapscott, A.: Blockchain Revolution: How the Technology Behind Bitcoin is Changing Money, Business, and the World. Penguin (2016)

[32] Tian, F.: An agri-food supply chain traceability system for China based on RFID & blockchain technology. In: 13th International Conference on Service Systems and Service Management (ICSSSM 2016). pp. 1–6 (Jun 2016)

[33] Viroli, M., Omicini, A.: Coordination as a service. Fundamenta Informaticae 73(4), 507–534 (2006)

[34] Wood, G.: Ethereum: a secure decentralised generalised transaction ledger (2014), http://ethereum.github.io/yellowpaper/paper.pdf