Ricardian Contracts for Industry 4.0 via the Arrowhead Contract Proxy

Emanuel Palm, Olov Schelén, Ulf Bodin
Luleå University of Technology, Luleå, Sweden
E-mail: {firstname.lastname}@ltu.se

Christian Lagerkvist
SEB, Solna, Sweden
E-mail: {firstname.lastname}@seb.se

Abstract—Industry 4.0 will require unprecedented degrees of integration across organizational boundaries, which will put new demands on infrastructure for managing agreements between industrial stakeholders. In this paper, we present a system-of-systems architecture for cross-organizational negotiation of Ricardian contracts. We also describe our implementation of it, based on Eclipse Arrowhead, and how it can produce non-repudiable contracts between local clouds, potentially owned by distinct parties. We discuss how our architecture could impact current business paradigms, as well as arguing that our design, in contrast to most solutions based on smart contracts, avoids to deviate significantly from contemporary legal praxis, which should create better opportunity for industry adoption.

I. INTRODUCTION

Virtually every industrial activity can be traced back to either an existing agreement or an anticipated such. While the forms and complexity of these agreements may vary, ranging from highly intricate service contracts to simple sales, they are, by and large, the primary determinant of what products will be produced or what services will be provided by any given industrial stakeholder. As a consequence, digitalizing the (1) entering into, (2) fulfillment, (3) renegotiation and (4) analysis of contracts, whatever the variety, has great potential to reduce costs and lead times in present day supply chains. A digitalized contract could, for example, allow for key details, such as product specifications or delivery locations, to be automatically fed directly into systems controlling automated production or service rendition. Furthermore, it could serve to open up for new forms of dispute handling, financial services and business models.

However, a digitalized contract must be facilitated by a concrete technology, and the degree to which that technology can realize this potential depends on many factors. Among those we have identified (1) compatibility with contemporary legal praxis, (2) the risk of fraud, (3) privacy, (4) generality, (5) cost, (6) latency and (7) throughput. While surveying the state-of-the-art, we found no existing digital contract platform we have been able to find [5], rely on the smart contract [6] popularized by Bitcoin [7]. Even though smart contracts, which are computer programs controlling various types of ledgers, might superficially seem legally sound [8], these solutions tend to have technical [9] and legal [10] caveats that bar them from most real-world applications [11]. Examples of such limitations are (1) being hard to interpret for current legal institutions and practitioners, (2) not directly providing for contracts to be negotiated and renegotiated, or (3) by imposing novel privacy or governance structures.

In this paper, we present our solution for industrial contract digitalization based on Ricardian contracts [12], which consist of regular legal prose and machine-readable parameters. By our sticking to established technologies, such as unmediated message passing over secure transports, digital signatures and hash pointers, we believe to sufficiently satisfy our seven factors for our solution to become practical in real-world industrial use cases, even though our solution is currently limited to contract negotiations between exactly two parties. Furthermore, as the Ricardian contract is partially rather than predominantly machine-readable, as are other comparable formats [13] [14], its users can decide how much to rely on legal prose or machine formalisms. This leaves room for established legal praxis, such as strategic ambiguity [15], and for the gradual introduction of increasingly complex formalisms.

What we consider the primary contribution of this paper is, however, our outlining of a concrete realization of Ricardian contracts based on a system-of-systems framework designed for the Industry 4.0 paradigm, Eclipse Arrowhead [16] [17], as depicted in Figure 1. By presenting our implementation, we make the case that decentralized and practically applicable designs can be produced without reliance on smart contracts.

Fig. 1. The Ricardian contract is a key part of our Contract Network Architecture, which we realized as a Contract Proxy system-of-systems.

The work was funded through the DigiPrime (H2020-EU.2.1.1., ID 873111) and Arrowhead Tools (H2020-EU.2.1.1.7., ID 826452) projects of ECSEL.
We begin to make the case about the suitability of our design by describing what a Ricardian is and how we represent it. After that, we outline our Contract Network Architecture, and then proceed to describe its Arrowhead realization through our Contract Proxy system. We end the paper by describing our design explicitly allows for default values in templates.

II. THE RICARDIAN CONTRACT
Our Ricardian contract [12] consist of two parts, templates and contracts, as exemplified in Figure 2. Each template consists of a list of provisions and parameters, where each provision is a human-readable text and every parameter is a name/type pair. Each contract parameterizes exactly one template by providing a list of arguments, which are name/value pairs matched against the corresponding name/type pairs of its template. Template provisions cannot be affected by contracts. This structure mostly aligns with the one proposed by Clack et al. in [18], with exceptions such as naming differences and that their design explicitly allows for default values in templates.

Given that type support exists for a given implementation, contract arguments could contain references to any kind of data for which suitable identifiers can be produced, including templates, or the offers, acceptances and rejections we outline in Section III. In fact, the structure we propose does not put any hard limits on what data can be represented by a contract, as long as suitable types are available. Contracts can refer to each other in chains, certificates, new templates, vehicle routes, models to be 3D-printed, multi-step financial transactions with conditions, among many other possible examples.

A more advanced Ricardian type system could also allow for our template provisions, which now must be plain texts, to be expressed as machine-readable formalisms, such as those of LegalRuleML [13]. This could leave room for more exact forms of machine analysis and verification.

1The original paper by I. Grigg [12] is not very exact about what the structure of a Ricardian contract is. It also requires that contracts only represent issues of value and that certain servers be identified in each contract, both of which arbitrarily limits what the contracts can be used for. Perhaps unsurprisingly, this has lead to variants of the Ricardian contract emerging, as long as suitable types are available. Contracts can refer to each other in chains, certificates, new templates, vehicle routes, models to be 3D-printed, multi-step financial transactions with conditions, among many other possible examples.

III. THE CONTRACT NETWORK ARCHITECTURE
The purpose of the Contract Network Architecture is to facilitate contract networks. Such a network consists of a collection of communicating computer agents, each of which represents a party that all potential contractees can determine the legal identity of. Each agent can communicate with all other agents via message-passing over a trusted medium, can identify all parties endorsing each received message, and stores all received messages permanently. There are two kinds of communication each agent must be capable of, (1) negotiation and (2) reference tracing, which we will now consider in turn.

A. Negotiation
Negotiation is the process through which agents produce contracts. In the naive procedure we use here, two agents can send (1) offers, (2) acceptances and (3) rejections, on behalf of their respective parties. All of these messages must be non-repudiable, whereby we mean that the identity of their senders or their contents cannot be modified or denied after being sent. An agent sending an offer to another agent starts a new negotiation, unless the offer counters a previous offer, as shown in Figure 3. An offer receiver can send a counter-offer, acceptance, rejection, or wait for the offer to expire.

As illustrated by the example contract network in Figure 4, each offer contains one or more contracts, as defined in Section II, as well as identifying the sender and receiver of that offer. It must be possible to tie each acceptance or rejection to a certain offer, part of a specific negotiation.

1The original paper by I. Grigg [12] is not very exact about what the structure of a Ricardian contract is. It also requires that contracts only represent issues of value and that certain servers be identified in each contract, both of which arbitrarily limits what the contracts can be used for. Perhaps unsurprisingly, this has lead to variants of the Ricardian contract emerging, as long as suitable types are available. Contracts can refer to each other in chains, certificates, new templates, vehicle routes, models to be 3D-printed, multi-step financial transactions with conditions, among many other possible examples.
We started this section by claiming that our negotiation procedure is naive. What do we mean by that? We mean that we do not support making proposals, multi-party negotiations, auctions, or other more advanced features that could be relevant to many use cases. By making proposals, we refer to the practice of sending offers that are not specific enough to be acceptable by their receivers. In conventional contracting, proposals are a key way of signaling interest without knowing the details about what a counter-party wants or is willing to offer. Why do we only present a naive procedure? Because a negotiation procedure is required to make our solution general-purpose, and by presenting a simple one we get a pedagogical advantage. We revisit this discussion in Section VI-A.

B. Reference Tracing

When an offer is received by an agent, it may or it may not be the case that the agent already knows of the templates, or any other artifacts, referred to by the contracts of that offer. To make sure that the referenced data can be inspected, as appropriate, before a decision is made about an offer, each agent is able to request the artifacts referred to by the offers they receive. Upon receiving such a request, an agent could (1) respond with the artifact, (2) respond with the details required to acquire the artifact from some other source or (3) respond with an indication that permission is not granted for the artifact to be revealed.

Some may consider why this particular scheme always waits for data to be explicitly requested before being sent, rather than having offers include data when it is known to be unknown to their receivers. There are two problems with this line of reasoning, however. The first problem is knowing what the offer receiver already knows. The second problem is knowing what the offer receiver considers relevant to know. Not pushing any data before it is requested may not lead to optimal latency or the smallest number of transmissions, but it does guarantees that no data is sent redundantly. If a strong enough case can be made for being able to push data preemptively to offer receivers, such a mechanism could, of course, be added to our architecture.

C. Implementation Strategies

You might have noticed that we were quite specific about things such as hashes and signatures in Figures 2 and 4, but used more general language in the rest of this section. The reason for this is that we believe it could be motivated to implement this architecture in different ways to facilitate different kinds of use cases, even though hashes and signatures are used by the realization we outline in Section IV. If interested to read more about such strategies, please refer to [19], in which we describe the implications of implementing a predecessor of this architecture in three different ways. Those approaches are (1) relying on a trusted centrally managed application, (2) implementing it on top of a blockchain network, as well as (3) using an approach reminiscent of the one presented in this paper. The differences between the architecture presented in this paper and the one in [19] are outlined in Section V.

IV. The Arrowhead Contract Proxy

Having described both our Ricardian contract format and our Contract Network Architecture, we are ready to consider how we realized those in our Contract Proxy implementation. First and foremost, our realization is built to be compatible with Eclipse Arrowhead [17] [16], which is a service-oriented systems framework designed to help create Industry 4.0 supply chain and manufacturing systems-of-systems. The framework both provides ready-made systems, as well as specifications for how to make custom-made systems. Of particular interest to us are the ready-made systems for (1) service registration, (2) authentication and authorization, (3) orchestration, (4) event distribution and (5) secure inter-cloud communication, which can facilitate the infrastructure we need for our use case.

As per Arrowhead terminology, each deployment of the framework with its own service and system management capabilities is referred to as a local cloud. In the context of our use case, depicted in Figure 5, we make the assumption that each such cloud is managed by a distinct stakeholder. In addition to the already listed systems and two of our Contract Proxy systems, we add two custom-made agent systems, which use the contract proxies to negotiate contracts. The source code of our implementation, as well as a simple use case example, can be reviewed and downloaded from https://github.com/emanuelpalm/arrowhead-contract-proxy.

![Fig. 5. Two Arrowhead Framework local clouds, each with its own agent and Contract Proxy systems. The agents can send and receive offers, acceptances and rejections via their contract proxies, which are connected via Gateway systems. Event Handler systems are used in each cloud to notify agent systems about new contract messages. We assume the Service registry, Authorization, Orchestrator and Gatekeeper systems to be used in each local cloud [17].](https://example.com/arrowhead-contract-proxy)

Contract proxies can be regarded as making up a key part of the trusted medium through which agents must communicate, as described in Section III. Their primary utilities are

a) making it possible to regulate what agents within the local cloud can negotiate or see negotiations by setting access control rules,

b) making it straightforward to configure and monitor any negotiability occurring between agents in its own local cloud and other local clouds, as well as

c) alleviating agent programmers from having to deal with cryptography, which removes a potentially significant source of security issues.
More specifically, each Contract Proxy
1) controls access to an owned set of certificates (to which it has the private keys) and a trusted set of certificates (representing the agents it can relay messages to),
2) adds the necessary signatures and other cryptography to offers, acceptances and rejections relayed on behalf of agents from inside its own local cloud,
3) verifies and removes signatures and other cryptographic elements from messages relayed on behalf of agents from outside its own local cloud,
4) ensures that all signed messages it relays, in either direction, are stored persistently, as well as
5) responds to reference tracing requests, as defined in Section III-B.

The process through which contract proxies relay messages between agents is depicted in Figure 6. In that example, the agent in local cloud A (aA) sends a message to the agent in local cloud B (aB) via a sequence of six steps, which are
1) aA sends a message (mA), composed using identifiers only guaranteed to be relevant within A, to the Contract Proxy of A (cA);
2) cA verifies that aA is authorized to send the message, replaces all identifiers specific to A with their global cryptographic equivalents, producing mG, signs it to create s(mG), and then relays it to cB through two gateways, which both authorize the sender before passing s(mG) on;
3) cB validates s(mG), removes its signatures to create mG, replaces its global cryptographic identifiers with such relevant only in B, producing mB, and then announces the availability of mB via the Event Handler in B (eB), which first verifies that cB is authorized to publish events;
4) eB announces the availability of mB to all properly authorized subscribers that have specified matching filter criteria, which here is the only agent of B (aB);
5) aB requests a copy of mB; after which
6) cB ensures aB is authorized to receive mB and sends it.

As these interactions occur within and between two local Arrowhead clouds, there are provisions in place for making sure that only explicitly authorized systems are permitted to be aware of each interaction. To be more concrete, for a system to be able to get notified of, for example, negotiational messages through an Event Handler, authorization rules be provided to the local cloud Authorization system by an administrator. The same is true for every other category of interaction, from requesting contractual messages to submitting them. Furthermore, the Contract Proxy systems can only interact with other Contract Proxies able to prove ownership of certificates that have been explicitly endorsed by administrators.

Why is the Event Handler part of our design? Using it, rather than having the contract proxies immediately pushing new messages to their intended recipients, allows for having systems that observe on-going negotiations without being part of them. Such an observing system could, when seeing a certain type of message, be programmed to trigger product manufacturing, service rendition, or the granting of access to private data, among many other possible examples.

Another detail worthy of mention is that the Contract Proxy does not strictly have to communicate with another Contract Proxy, as in our example. In fact, considering Figure 6, the Contract Proxy in A does not know whether the system receiving its messages in B is a Contract Proxy or some other kind of system. aB is completely invisible to both aA and cA. Any system being able to represent a certificate trusted by our Contract Proxy, as well as implementing the same negotiation protocol, would be able to negotiate contracts via that Contract Proxy. In certain scenarios it may, for example, be relevant to have a single system both initiating and reacting to negotiations while also managing all cryptography.

V. PRIOR ART

A predecessor to our Contract Network Architecture, based on token exchanges instead of Ricardian contracts, was presented in [1] and [19]. That model, known as the Exchange Network Architecture, introduced the need for keeping track of what obligations are acquired by whom as tokens change owners, which made them rather unintuitive as soon as they were meant to represent anything else than ownerships of assets. We eventually realized that service agreements, which have been important for us to express, are markedly less complicated to describe as regular contracts, and that asset exchanges do not become more complicated to express this way. The reason for this, we believe, is that contracts carry less assumptions about their utility than tokens do, making them more general-purpose. This made us decide that the Ricardian contract was a better fit for the use cases we focus on.

Another key difference between the two architectures is that the Contract Network Architecture, presented in this paper, has an implementation compatible with an industrial framework, Eclipse Arrowhead, while no such implementation was ever produced for our Exchange Network Architecture. We are, as a consequence, better able to argue for the suitability of the former design in the context of Industry 4.0.
VI. DISCUSSION

Before closing this paper, we will briefly consider a why our architecture does not support sophisticated negotiations, some weaknesses we discovered in our Contract Proxy design, how our architecture could facilitate new forms of dispute handling, financial services and business models and a brief reflection over the potential performance of our designs.

A. Limited Negotiational Capabilities

Why does our Contract Network Architecture, as well as its predecessor [1] [19], only directly support negotiation between exactly two parties? As we already mentioned in Section III-A, primarily because negotiation has not been one of our main research subjects up until this point. Since conventional contracts are always produced via some form of negotiation, our architecture had to support it to be as general-purpose. However, since our primary interests were in non-repudiation and industry application, we settled with what we saw as the simplest possible solution. More sophisticated negotiation protocols are presented in [20] and [21], for example.

B. Contract Proxy Weaknesses

1) Reliance on Event Handler: The point of our announcing messages through an Event Handler system, rather than having the contract proxies immediately push new messages to their intended recipients, was motivated by (1) making it possible to monitor and distribute incoming messages in a standardized fashion, as well as (2) not requiring any Contract Proxy to keep track of subscribers or publishing messages, which should leave room for better performance when the volume of messages is high. However, after implementing and evaluating this setup, we are not convinced about its suitability. While the availability of incoming messages can be logged via the Event Handler, it cannot log the actual messages. Only announcements about messages being available are sent to the Event Handler, not any actual messages. Furthermore, while using the Event Handler could lead to performance benefits, the complexity increase inherent in introducing another node to a distributed system may not be warranted for all kinds of use cases. A better approach might have been to have the Contract Proxy be designed to keep track of and notify subscribers by itself. One advantage is that fewer systems need to know of each contract message, which could help keep them confidential. If performance becomes a concern, a multi-node Contract Proxy could be likely be produced.

2) Insufficient type system: While we have argued for the merits and possibilities that comes with the type systems of Ricardian contracts, we implemented a rather weak variant of such a type system for our Contract Proxy system. The only types we support are strings and hashes, the latter of which are used as input to our reference tracing algorithm. Even tough it is not critical for that type system to be feature complete before being used in production, care must be taken for it to be straightforward to extend it with more features in the future. We suspect that our type system will have to be revised before we can be entirely confident such extensions can be facilitated.

3) Removal of cryptographic primitives: Our Contract Proxy was designed to substitute cryptographic elements with local identifiers, alleviating systems within that could from having to implement any cryptographic scheme with direct relevance to contracting. We soon noticed, however, that even though the local identifiers were indeed very useful, that didn’t mean that the cryptographic primitives stopped to be relevant. While constructing a simple graphical user interface for our Contract Proxy, allowing for a human operator to receive and respond to negotiational messages, we wanted the hashes and signatures to be able to show the user how the different messages relate. An alternative could have been for the Contract Proxy not to substitute cryptographic elements, but rather complement them with local identifiers.

4) Single point of failure: Taking down the Contract Proxy of a local cloud, which requires breaking past an Arrowhead Gatekeeper, removes the local cloud’s capability to negotiate. How and when to mitigate that is left open for future research.

C. A New Business Paradigm

Given that a system such as our Contract Proxy becomes widely deployed, what kinds of innovations could we expect in terms of (1) dispute handling, (2) financial services and (3) business models? Before we present our assessment, let us consider some of the properties of the design we are proposing. Firstly, none of the activities of our Contract Proxy require significant computing power or sending a larger numbers of messages, which we consider in more depth in Section VI-D. As a consequence, contracts can be negotiated within very brief time spans. Secondly, negotiation produces records of permanently stored signed messages, which could be shown to and relied on by any third party able to verify their signatures. Thirdly, by supporting both textual and formal provisions, as well as typed templates and instances, our contract system could facilitate a gradual transition from something very reminiscent of traditional contracts to very machine-oriented such. Such a gradual transition should facilitate a gradual translation of existing legal praxis, which ought to make it possible for more companies to contribute to and adopt the new legal paradigm.

1) Dispute handling: Given that recorded messages can be directly uploaded to and verified by a court of law or arbitration firm, it should become possible to reduce the time and costs associated with contemporary dispute handling. In the longer term, as more machine-oriented contracts become the norm, more aspects of litigation could likely be automated.

2) Financial services: Recorded messages could not just be shown to courts of law, but also to insurance agencies, banks, or other financial service providers. These could use those records to improve their financial models, which should lead to cheaper insurance, lower interest rates, as well as other types of benefits associated with lower financial risks.

2 A demonstration of the user interface could, at the time of writing, be seen at https://www.youtube.com/watch?v=2wLtAUBhWaI (accessed 2021-04-26).
3) Business models: Contracts written by humans tend to include as many details of an agreement as reasonably possible and beneficial, as it means that the parties in question have to review and sign fewer documents. When contracts are negotiated and signed by machines, however, there is an incentive for that dynamic to change. Those machines are likely required to need one computer program per contract template they are meant to react to, which creates an incentive to compose agreements from many smaller contracts in order to save on engineering effort. Furthermore, computers should be able to negotiate contracts many orders of magnitude faster than humans, which means that explicit contract-forming can become part of many interactions that today instead would yield receipts or other kinds of proofs of fulfillment. The end result should be that many businesses move over to entering into vastly more contracts with much smaller scopes, which should allow for parties to collaborate with unprecedented degrees of precision. While it is hard for us to predict the exact implications of such a development, we feel inclined to believe that it would lead to the emergence of new classes of legal and technical professions, specialized in a new paradigm of industrial contracting praxis.

D. Performance

While we did implement our Contract Proxy design, we did never formally benchmark its performance. We can report, however, that no message exchange during any of our tests took more than a second or two to complete. While we believe those interactions should have completed in the tests took more than a second or two to complete. While port, however, that no message exchange during any of our did never formally benchmark its performance. We can re-

VII. Conclusions

We believe our Contract Proxy design to typically be a superior choice for industrial deployment if compared to all current forms of smart contracts. Our design does (a) not require any other parties than the actual contractees to approve each change in rights or obligations, (b) provides for all types of contract breaches to be corrected after they can be observed, not just a limited subset of breaches [1], as well as (c) uses Ricardian contracts, which are similar to traditional contracts.

3 A demonstrator of a use case in which our Contract Proxy is used can be seen at https://youtube.com/watch?v=dEB0XwWkqGw (accessed 2021-04-26). If nothing else, it should make it clear that message latency does not exceed a few seconds.