Towards Event-Driven Decentralized Marketplaces on the BlockChain

Akash Pateria
apateri@ncsu.edu
North Carolina State University
Raleigh, United States of America

Kemafor Anyanwu
kogan@ncsu.edu
North Carolina State University
Raleigh, United States of America

ABSTRACT

Blockchains have become a popular technology for lowering the trust-tax burden between transacting parties that cannot necessarily trust each other. They are used as substitutes for the centralized authorities typically incorporated in transactional workflows to perform verification tasks and have the advantage of being objective and incorruptible. For applications such as supply chain marketplaces, auxilliary functionalities beyond the core blockchain roles of recording and validating transactions such as event detection are important for enabling application participants be responsive to business conditions. Unfortunately, existing blockchain event frameworks are immature, syntactic, inflexible and not expressive enough for many application needs.

In this paper, we propose an approach that involves an event model which "semantifies blockchain transactions" and an implementation architecture that integrates a open-source blockchain database BigChainDB with a semantic engine and publish-subscribe messaging platform. Finally, we model and simulate a use-case inspired from the manufacturing domain and present usability and preliminary performance results that demonstrate the discriminatory ability of semantics-enabled event model.

CCS CONCEPTS

• **Information systems** → *Computing platforms*; **Enterprise applications**.

KEYWORDS

Blockchains, Decentralized Marketplaces, Semantic Events, Declarative transactions

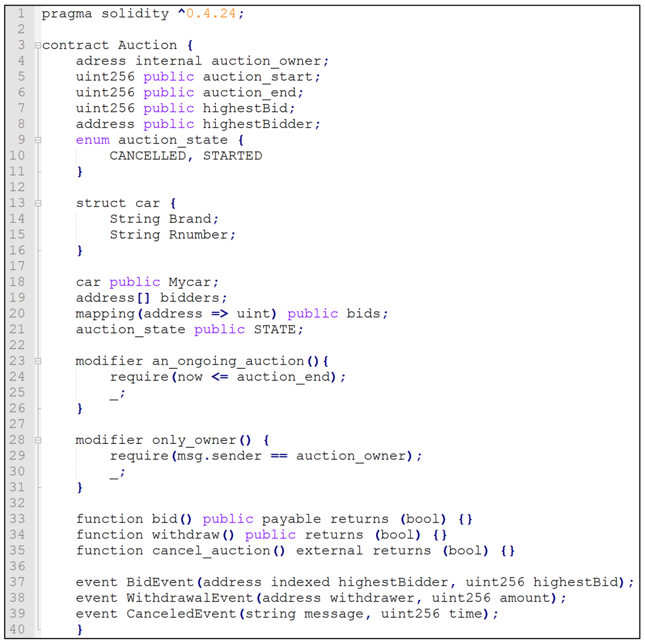
ACM Reference Format:
Akash Pateria and Kemafor Anyanwu. 2021. Towards Event-Driven Decentralized Marketplaces on the BlockChain. In *The 15th ACM International Conference on Distributed and Event-based Systems (DEBS '21), June 28-July 2, 2021, Virtual Event, Italy*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3465480.3466921

1 INTRODUCTION

Supply chains comprise of different interacting workflows covering multiple phases in the business and production processes beginning from procurement, through order processing, production, shipping to delivery. The participants in the workflows are often a dynamic set of humans and machines or automated processes that are managed by separate and independent organizations. Depending on the application domain, there can also be dynamism in terms of demand and operational requirements which can further complicate matters. For example, manufacturing companies operate in environments characterized by constant change which may demand adaptations to changes in design, configuration and processing functions, production capacity, and dispatching of the orders [28]. Consequently, there has been a dominant trend towards more digitization of processes to help support improvement in responsiveness to business demands while improving operational and cost efficiencies.

Blockchains are increasingly becoming an important component of the technology ecosystem, particularly for complex supply chain applications. Blockchains cooperatively maintain ledgers or databases that are emerging as the so-called "technology of trust". Being immutable and without centralized control, blockchains insulate against tampering of data (at least not without being noticed), enabling data consumers to trust its data. This is in contrast to the traditional data management model in which trust is typically vested in a central entity (banks for managing financial data, registered entities for real estate escrow) without any guarantees against such an entity going rogue. Popular blockchain platforms include Bitcoin [32], Ethereum [10], Hyperledger[7] and so on. As an example of its use in supply chains, the global furniture retailer IKEA [6] has investigated the use of blockchains in the supply chain sector where decentralization combined with immutability is expected to enhance the transparency and robustness of transaction processing drastically" [35]. IKEA's goal was to use blockchains as a "transparent and trustworthy documentation of events in the supply chain, where multiple organizations are involved, and where no entity should be able to manipulate information without it being noticed" [35]. To this end, its blockchain solution recorded all the events and many participants handling different products in its supply chains from creation to final ownership (28 event types associated with Order creation, Product creation, Shipment, Logistics, Delivery, etc.). Other efforts [31], [18], [21] go beyond the needs of a single company to try to provide support for specific vertical services e.g. procurement, by offering "decentralized marketplaces".

Decentralized marketplaces are digital platforms where matchmaking is done between product and service *requestors* and appropriate `Suppliers` using the blockchain rather than a central



```
pragma solidity ^0.4.24;

contract Auction {
    adress internal auction_owner;
    uint256 public auction_start;
    uint256 public auction_end;
    uint256 public highestBid;
    address public highestBidder;
    enum auction_state {
        CANCELLED, STARTED
    }

    struct car {
        String Brand;
        String Rnumber;
    }

    car public Mycar;
    address[] bidders;
    mapping(address => uint) public bids;
    auction_state public STATE;

    modifier an_ongoing_auction(){
        require(now <= auction_end);
        _;
    }

    modifier only_owner() {
        require(msg.sender == auction_owner);
        _;
    }

    function bid() public payable returns (bool) {}
    function withdraw() public returns (bool) {}
    function cancel_auction() external returns (bool) {}

    event BidEvent(address indexed highestBidder, uint256 highestBid);
    event WithdrawalEvent(address withdrawer, uint256 amount);
    event CanceledEvent(string message, uint256 time);
    }
```

Figure 1: Car Auction Smart Contract Example

authority. In this context, blockchains can be used to record important aspects of the negotiation activity between requestors and providers as well as to maintain a historical log of provider activities as a means of verification of stated capability by the provider. The experience during the early phase of the COVID pandemic where there was an acute shortage of PPE created significant demand for any manufacturers with the required manufacturing capabilities and capacity for PPE (regardless of whether those entities produced PPE as their core business as long as purported capability could be verified). Online marketplaces like "American Makes"[1] offer a service with which hospitals could enter their PPE needs, manufacturers submit what they can make and general users share designs of products to be 3D printed. Other centralized manufacturing marketplaces do exist [8], [12]. However, the limitations of a centralized entity are that it is vulnerable to risks of bias, collusion between the entity and some of its users. In general, there is significant interest in enabling these sorts of digital and automated, but trustworthy marketplaces because the existing approach for matchmaking is lengthy (on the order of up to a year sometimes), manual, tedious, and expensive, particularly in high stake domains like aircraft manufacturing. Here, a major manufacturer that needs to vet potential suppliers or subcontractors engages in a manual process of reviewing certifications and job history, inspecting plants, and so on. Consequently, such major manufacturers find it difficult to diversify their supplier pool or to find niche area suppliers and to resort to sticking with a few well-known ones, creating a significant barrier to entry for other potential suppliers. Finally, in the context of public procurement, government regulations often demand fair competition and transparency between the participants in the bidding and selection process for an awardee of a public contract. Other efforts [31] [18] [17] have focused on developing blockchain-enabled marketplaces, particularly in the IoT domain.

While blockchain platforms have as their primary function to process, validate and record transactions, important auxilliary functions need to be incorporated into the application ecosystem to provide broader support of business functions. Event frameworks which enable application participants be notified of relevant transactional activity is critical for a business's responsiveness and its ability to maintain operational efficiency and competitive advantage. For example, it will be important for suppliers of products or services to be informed as soon as there are requests that match their offerings in order for them to submit proposal bids in a timely manner. Such event frameworks should support discriminatory filtering of events, so that event subscribers are not being bombarded with irrelevant events. Currently, the main method for users to discover about transactional activity is to constantly poll the blockchain using querying tools such as [4], [3], [22], [19], [20] to scan and parse blockchain data and logs. A few efforts like Eventeum [5] have integrated a publish-subscribe mechanism into the architecture, allowing users to subscribe to events of interest. However, both the querying and subscription-based approaches have significant limitations with respect to the needs of applications like the sort of marketplaces we are considering. Existing event models and frameworks have:

- **Limited expressiveness in event representation**: events are typically represented as *static and limited declarativeness* signatures. Each event is a *predefined event name/identifier with a fixed set of associated values.* Figure 1 shows some examples event signatures for a Car Auction smart contract e.g. BidEvent(..) to represent an auction bid. For one thing, event consumers must not only be aware of the keyword used to identify an event, but also the role and semantics of associated values or parameters in the event signature given that the degree of declarative-ness is with respect to data typing. This approach may be sufficient for single-provider marketplaces and very narrow vertical applications where assuming that event consumers know event specification in sufficient detail may be reasonable.
  However, for many-many marketplaces (many providers, many consumers), different users will need to have the flexibility to express their needs in a way that suits their requirements (not a one-signature-fits-all). For example, different REQUEST_FOR_QUOTEs for manufacturing services may be described with varying details including maybe material to be used for manufacturing, the product shape to be made, color etc., because these features have an impact manufacturing capability. Such flexibility in event specification requires declarative, extensible and semantic specification mechanisms so that users can add as much detail as desired and what information is added can be properly interpreted. Unfortunately, in addition to the already mentioned limitations, the event models supported in platforms like Ethereum only allow maximum of 4 keywords as event signatures (each a maximum of 32 bytes, and one dedicated to event name, leaving only three values of parameters possible). This is as result of the memory structure of the virtual machine used to execute "smart contracts" [14] in which events are hosted.
- **Limited discriminatory ability**: a direct consequence of the limited expressiveness of event representation is the fact that rich or complex events will have to assume such generalized representations that their discriminatory ability with respect to

subscription will be limited. For example, if appropriate details of REQUEST_FOR_QUOTE are missing in a such an event, then event consumers may have to receive all REQUEST_FOR_QUOTE and the locally process to filter out if the request is relevant to them. Indeed, in Eventeum [5], all application events are received by subscribers in single topic. The only level of discriminatory ability is discriminating between system and application events.

In this paper, we focus on enabling a *many-many requestors-suppliers marketplace* as opposed to a single requestor or provider marketplace in which there is a single predefined way to communicate requests. For a *many-many* marketplace, maximum flexibility must be provided for different users to express their needs as they deem fit. In other words, (i.) requestors should be able to present their needs and requests in as much or little detail as desired using vocabulary of choice while (ii.) suppliers must express interests in receiving notifications about such requests in a from that is decoupled from request descriptions both in form and conceptual models.

Specifically, we present:

(1) an implementation architecture that integrates a distributed publish-subscribe platform, Apache Kafka, and a semantic engine StarDog with a blockchain database based on BigChainDB.
(2) "semanticizing" a topic-based publish-subscribe event description space as well as BigChainDB's declarative and extensible transaction model to be used as a foundation for blockchain marketplace transaction types. This approach effectively decouples event specifications from event subscriptions meeting the desired flexibility requirement.
(3) an ontology-based semantic framework for mediating between the decoupled "transaction-based" event model space and the "topic-based" event subscription space.
(4) an experimental usecase evaluation using some sample data and ontologies from the manufacturing domain. The results demonstrate the discriminatory power of using our semantics-enabled, expressive event specification model over existing approaches that rely on purely syntactic event models.

2 BACKGROUND & RELATED WORK

Blockchains are essentially decentralized databases in which "records" are organized into a sequence or "chain of blocks" such that there is a dependency between adjacent blocks in a chain and once blocks are modified the chain is broken (making it easy to detect changes). While the core functions of blockchains are creating and transferring assets, some platforms such as Ethereum [10] and Hyperledger[7] support extensibility features that allow additional business process behavior whose execution also needs to be proceeded within the blockchain for reliability. The so-called *smart contracts* [14] encode business processes and terms of the agreement between parties as arbitrary program code which self-execute under the control of the blockchain, once specified conditions have been met. These blockchains also offer support for events as part of their smart contract frameworks. Figure 1 shows an example of Ethereum smart contract implementing a car auction with three event types (adapted from [16]).

Our introductory discussion highlighted some of the limitations of existing event models. In the following, we will elaborate on these limitations by using illustrative examples.

2.1 Motivating Example

Marketplaces for domains like manufacturing are expected to have a broad range of users ranging from casual users who know about the product they want to manufacture but do not have deep knowledge about relevant manufacturing techniques, to more expert users who have a good idea of their manufacturing needs. Some examples of possible manufacturing requests follow:

(1) Manufacturing of a hip implant with titanium and a surface roughness Ra value of 2-5 microns and delivery time of 3 weeks.
(2) Mass manufacturing of a toy (> 100,000 units) along with plastic, transparent packaging for the toy so that customers can view it while on store shelves.
(3) Manufacturing of medical device with polycarbonate material and small quantity (<1000).
(4) Ambulatory bags, Pressure relief valves, Ball joint rod ends, and sheet metal fabrication (laser or water jet).

The examples show that manufacturing requests can be described using different information dimensions e.g. product type or product components, material type, manufacturing process and so on depending on how much the user understands their requirements. Therefore, in order to allow transactional activity such as these service requests to form the basis of event messages, *event specifications are going to need to be flexible and expressive*. On the other hand, for event consumers what is critical is to determine if a request for service falls within their service capabilities. For example, for request (1.) *Abrasive Polishing* is a required process because of the material and low surface roughness requirement; (2.) *CNC Milling* and *Thermoforming* because the arbitrariness of toy shapes requires a CNC machine to create a 'mold', and then the mold is used to form the plastic with the desired shape using the thermoforming machine; (3.) *3D-Printing* because of the small quantity. (4.) specifies its request in terms of a combination of product types (Ambulatory bags, Pressure relief valves, Ball joint rod ends) and a manufacturing capability (sheet metal fabrication). These are components for producing a ventilator assembly and the sheet metal fabrication capability is required to cut the metal needed for the ventilator cover (this was a real use case in the early phase of COVID). Careful consideration will reveal that it is impractical to attempt to match requests with subscriptions expressed in a similar model i.e. in terms of different dimensions of product type, material etc. because there will be too many possible alternative descriptions that could be considered "matches". For example, different kinds of request descriptions would imply *3D-Printing* . In-fact, not only the degree of detail in description could be different, but also terminologies could. For instance, *3D-Printing* and *Additive Manufacturing* are synonymous. Therefore, matching event messages to subscriptions just based on textual representations would fail if, for example, the event message used one term while the subscription used another.

In summary, what we can observe is that while for event descriptions, we want to allow expressive specifications, for event subscriptions, we need more generalized abstractions of specifications. Consequently, *we need a semantic mediation mechanism to bridge the heterogeneities in event message and subscription specifications, as well as, differences in terminology usage.*

2.2 Semantics-Enablement with Ontologies

When there is a need to reconcile terminological differences, ontological data representations rather than mere textual representations are the most commonly accepted method. *Ontologies* are formal conceptualizations of concepts and relations in an application domain (which could be associated with textual labels in a well-defined way). The W3C has standards for specifying formal conceptualizations or ontologies in the form of ontology modeling languages with an accompanying formal syntax and rules for unambiguous interpretations of language constructs. The standards also introduce ontological primitives that serve as the foundation for application and domain ontologies. For example, the concept of a *class* and a *property* (a named binary relation) is defined as primitives. Then, certain specific relationships are also defined as primitives such as the rdfs : subclassOf property which allows two classes to be linked by subsumption while the properties/relations owl : disjointWith and owl : equivalentClass allow one to assert that two classes are disjoint or that they are equivalent respectively. Such assertions allow reasoners to make inferences about data characteristics.

The characteristics of a property can be refined further by asserting its membership in special classes of properties. For example, if in a family ontology the property $ancestorOf$ was introduced, its semantics could be further refined by saying that it is an owl : transitiveProperty while something like $friendOf$ could be asserted to owl : symmetricProperty. These primitives in the `rdfs` and `owl` standards have associated logical axioms used for logical inferencing with those primitives. Ontological reasoners that implement the primitives in the standards can support the automatic inferencing that is needed when semantic heterogeneities occur in data models. For example, the synonymous terms *3D-Printing* and *Additive Manufacturing* can be asserted to be equivalent using the i.e. *3D-Printing* owl : sameAs *Additive Manufacturing* . Then, data processing involving ontology reasoners will be able to handle the terms as equivalent (which would not happen in non-semantic representations). Using these primitives in the standards as a foundation, domain concepts and relationships can be formalized and organized into semantic structures. Many ontologies for different domains have been curated and made publicly available in the Linked Open Data Cloud. Even in the manufacturing domain, ontologies such as [30] [9] and many others have been proposed to represent different aspects of manufacturing, from product data models to manufacturing processes, resources, and so on. In-fact, some of the ontologies have been proposed to enable this idea of a Manufacturing-As-A-Service model (MaaS) which we are discussing here where "explicit representation of service requests in global manufacturing-service networks" can be modeled. Figure 2 shows some key concepts of a summarized graphical representation of the Manufacturing Resource Capability Ontology (MaRCO) [27]. MaRCO is an OWL-based information model for representing manufacturing capabilities and their relationships to manufacturing processes, resources or devices, products and so on. Each of the high level concepts shown is further refined by additional ontologies such as the Product ontology which introduces concepts like a product's material and the process taxonomy which classifies different manufacturing and assembly processes. Detailed discussion of MaRCO and its extensions can be found in [25], [34], [26].

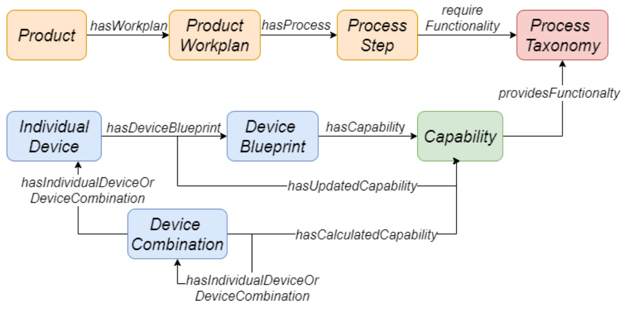


Figure 2: MaRCO Ontology – Key Concepts and Predicates

Beyond enabling reasoning due to ontology axioms specified in the `rdfs` and `owl` W3C standards, there are inferences that need to be made based on domain-specific knowledge and there are other standards that allow for such user-defined domain-specific rules. For example, in [24], the SPARQL Inferencing Notation (SPIN) is used to specify rules for inferring manufacturing resources needed based on product requirements specifications e.g. if a product manufacturing process requires a *Picking* capability, then the requirement can be fulfilled with a resource/machine that supports Grasping and Moving capabilities. Whereas for the Placing capability, Releasing is required along with the *Grasping* and *Moving* capabilities.

$$\frac{\langle Picking\rangle}{\langle Grasping\rangle.\langle Moving\rangle} \quad \frac{\langle Placing\rangle}{\langle Releasing\rangle.\langle Grasping\rangle.\langle Moving\rangle} \qquad (1)$$

Such rules can be built on top of ontologies by using ontological concepts and relationships as terms in the inference rules. In other words, the terms like *Grasping* can be a concept in an ontology.

One other emerging technique that has the potential of enriching traditional blockchain event frameworks based on smart contracts is the use of richer, declarative specification languages such as domain-specific languages [33] [23] [15] for encoding smart contracts, which potentially could improve the expressiveness of event models based on such. However, this is still a very new area of research with no major project implementations and adoptions.

3 APPROACH

Our approach is based on capturing transactional activity on the blockchain (e.g. REQUEST_FOR_QUOTE) as the basis for events, and then matching them to event subscriptions defined as generalizations of the event descriptions. In order to achieve matching between transactional event descriptions and the generalized event descriptions, domain knowledge in the form of ontologies and domain rules are used to bridge the decoupling of both definition spaces. In effect, the key components of our approach include: (i.) an open-source blockchain database (BigChainDB [13]) that has a declarative and extensible transaction data model used to form the basis of expressive event modeling in terms of what we call *event message descriptors*; (ii.) a distributed, topic-based, publish-subscribe messaging platform brokering messages between event producers and event consumers; (iii.) a semantic engine that hosts domain ontologies used for semantic extensions to the transaction model and publish-subscribe topic space and, domain rules and a rules-based reasoner drawing inferences between transactional events and event subscriptions.

Figure 3 shows an architecture that comprises three main layers.

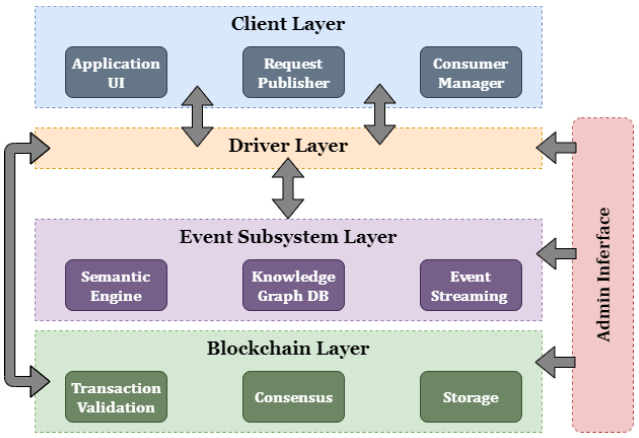


Figure 3: System Architecture

3.1 SMARTCHAINDB's Event Framework

BigChainDB supports key block-chain characteristics such as decentralization, immutability, owner-controlled assets while offering appealing database characteristics such as high transaction rate, low latency, indexing, and querying of structured data. It is Byzantine fault-tolerant (BFT), allowing up to a third of the equal-voting-power validator nodes to fail arbitrarily before the system stops operating and committing blocks. Although, it does not support smart contracts in the real sense, it supports simple cases of smart contracts called *cryptoconditions*.

BigChainDB's transaction model supports traditional blockchain transactions: CREATE and TRANSFER for creating and transferring assets respectively. The transaction data model is declarative, semi-structured (key-value pairs), allowing for extensibility. The SMARTCHAINDB platform in our architecture is an extended BigChainDB which introduces the new transaction types such as REQUEST_FOR_QUOTE, BID. The fundamental difference between this approach and existing approaches is that in existing platforms, these extraneous functionalities are modeled as smart contract methods which have several limitations, whereas in our model, they are captured as first-class transactions. Introducing new transaction classes amounts to introducing appropriate schemas for the transactions types and corresponding validation algorithms. The details of these transaction extensions are outside of the scope of this paper; however, we overview with illustrations.

Figure 4 shows our model's BID transaction "equivalent" of the bid function in the smart contract in Figure 1. It illustrates the modeling of a transaction using a nested key-value pair data model.

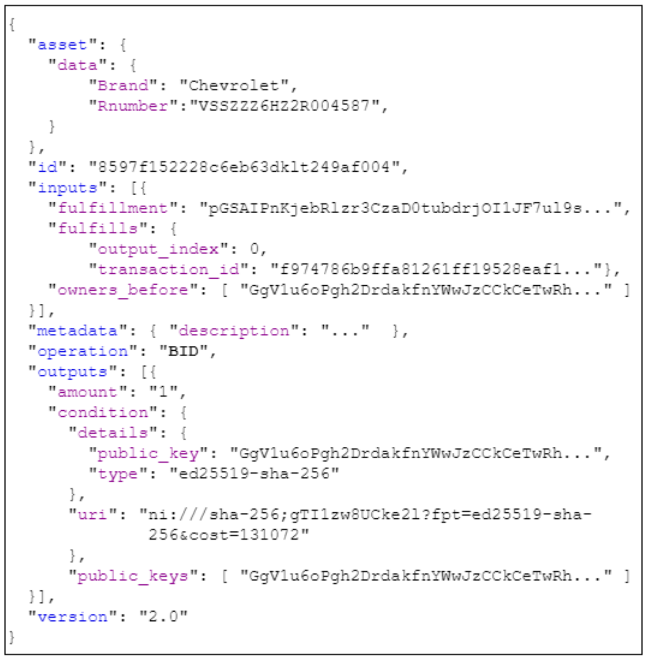
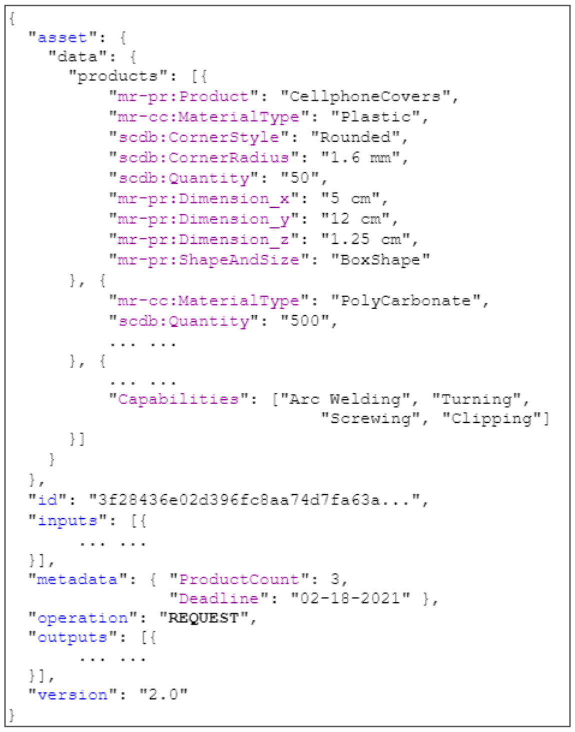

```
{
  "asset": {
    "data": {
        "Brand": "Chevrolet",
        "Rnumber":"VSSZZZ6HZ2R004587",
    }
  },
  "id": "8597f152228c6eb63dklt249af004",
  "inputs": [{
    "fulfillment": "pGSAIPnKjebRlzr3CzaD0tubdrjOI1JF7ul9s...",
    "fulfills": {
        "output_index": 0,
        "transaction_id": "f974786b9ffa81261ff19528eaf1..."},
    "owners_before": [ "GgV1u6oPgh2DrdakfnYWwJzCCkCeTwRh..." ]
  }],
  "metadata": { "description": "..."  },
  "operation": "BID",
  "outputs": [{
    "amount": "1",
    "condition": {
      "details": {
        "public_key": "GgV1u6oPgh2DrdakfnYWwJzCCkCeTwRh...",
        "type": "ed25519-sha-256"
      },
      "uri": "ni:///sha-256;gTI1zw8UCke21?fpt=ed25519-sha-
              256&cost=131072"
      },
      "public_keys": [ "GgV1u6oPgh2DrdakfnYWwJzCCkCeTwRh..." ]
  }],
  "version": "2.0"
}
```

Figure 4: ***BID*** **Transaction**

Some of the keys/attributes are generic and common to all transaction types. However, three key attributes that are transaction type-specific are the `operation` (in our example BID), `asset` and `metadata` attributes. The `asset` attribute captures fundamental, immutable attributes of an asset. For example, for the car asset, its brand, model, vin number, year, etc., are immutable attributes of a car. One the other hand, the `metadata` attribute allows for modeling of mutable attributes e.g. the color of a car which can change over time. Transaction types are supported by transaction type schemas that define such attributes and possible values, as well as appropriate validation algorithms that capture the correct behavioral semantics. For example, when one places a big in a BID operation, the asset used in the bid should be "locked" or held in some form of escrow and returned when higher bid is submitted or at end of auction.

To support semantic mediation based on transactional data content, we "semantify" the transaction schemas so that the attribute-value space for the transaction-specific attributes are drawn from an ontology rather than considering them as mere text-based keywords. Figure 5 shows the "semantified" version of SMARTCHAINDB's REQUEST_FOR_QUOTE transaction. This transaction includes a request to make a couple (3) of products. The details of the request are specified as part of the *asset* and *metadata* attributes descriptions. The kind and detail given about each product in the request are different. For one product, it specifies the product type as attribute $mr - pr$: *Product* and some additional product characteristics like material to be used as $mr - cc$: *MaterialType* and other product feature characteristics related to shape and so on. For another product, it does not include a product type but includes a material type and quantity. Then, for another one, it explicitly gives a set of capabilities required such as *ArcWelding*, *Screwing*, *Clipping*, and



```
{
  "asset": {
    "data": {
      "products": [{
          "mr-pr:Product": "CellphoneCovers",
          "mr-cc:MaterialType": "Plastic",
          "scdb:CornerStyle": "Rounded",
          "scdb:CornerRadius": "1.6 mm",
          "scdb:Quantity": "50",
          "mr-pr:Dimension_x": "5 cm",
          "mr-pr:Dimension_y": "12 cm",
          "mr-pr:Dimension_z": "1.25 cm",
          "mr-pr:ShapeAndSize": "BoxShape"
      }, {
          "mr-cc:MaterialType": "PolyCarbonate",
          "scdb:Quantity": "500",
          ... ...
      }, {
          ... ...
          "Capabilities": ["Arc Welding", "Turning",
                           "Screwing", "Clipping"]
      }]
    }
  },
  "id": "3f28436e02d396fc8aa74d7fa63a...",
  "inputs": [{
      ... ...
  }],
  "metadata": { "ProductCount": 3,
                "Deadline": "02-18-2021" },
  "operation": "REQUEST",
  "outputs": [{
      ... ...
  }],
  "version": "2.0"
}
```

Figure 5: ***REQUEST*** **Transaction**

so on. This example is reminiscent of example (4.) in the motivation section that had different products in the request, described in different ways. As can be observed, some of the attributes have prefixes in their names. These prefixes are aliases for an ontology from which the term is obtained. The manufacturing ontology earlier described *MARCO* 2 is used in this example. As a matter of fact, the *MARCO* ontology imports several other ontologies that are used to refine specific dimensions of the *MARCO* ontology. For example, there is a *Product* ontology that defines different product features and is aliased with the prefix $mr-pr$:. There is also a common concepts ontology aliased with the prefix $mr-cc$:. Another small local ontology is created to supplement links between the *MARCO* ontologies for the purposes of demonstrating some use cases. This local ontology is aliased by the SMARTCHAINDB namespace *scdb* :.

3.1.1 SMARTCHAINDB*'s Event Model.*

Transactions are used as the basis for event modeling. However, the transaction description space (attribute-value pairs) is decoupled from the event subscription space (topic-based). There might also be a decoupling in conceptual space which needs to be reconciled using semantic mediation. Consequently, transactions need to undergo a transformation in order to be mapped into the event message topic space. So from transactions, *event message descriptors* are generated and then eventually mapped to the event space using ontological relationships. More precisely, let's assume we have a semantic knowledge base $SKB_O = (O = C \cup P \cup T, R_O)$ where C is the set of ontology classes, P is ontology properties (or binary relation types), T is the set of terms i.e. instances, and R_O is a rulebase whose terms are from ontology O.

Event message descriptor. Given a SMARTCHAINDB transaction (the nested key-value model earlier described), we define the following data subtuple as an *event message descriptor emp* = $(op, A = (k_a, v_a), M = (k_m, v_m), \ldots)$ where op is a valid transaction operator such as RFQ, BID, etc. A and M are the associative arrays for the transaction's asset and metadata attributes respectively. As mentioned before, the keys for the both associative arrays k_a, k_m are drawn from classes and literals in an ontology O. As an example, for our REQUEST_FOR_QUOTE transaction example, its associated *event message descriptor* is the subtuple *emp* = (REQUEST_FOR_QUOTE, *(mr-pr:Product, "CellphoneCovers"), (mr-cc:MaterialType, "Plastic"), ... , (ProductCount, 3), ...*

Event topic space. An *event topic space ETS* is some *non-redundant* subset of $C \cup T$ (more accurately the textual labels associated with the ontological classes and terms). By *non-redundant* we mean that no pair of classes within *ETS* are "semantically equivalent". This space is expected to capture the allowable subscription possibilities and is assumed to be selected by an administrator. For example, for the manufacturing marketplace usecase that we have been alluding to, while *MARCO* ontology (including embedded ones) cover concepts such as manufacturing *resources*, *materials*, *product types*, *processes*, etc., the manufacturing *capability* concepts e.g. $ArcWelding$, $Screwing$, etc., are the concepts that make more sense for a subscription. The concept of *nonredundant* means that such as selected space should not contain both the concepts *3D-Printing* and *Additive Manufacturing* because both are equivalent concepts. Redundant topics in the topic space could create problems of missed event matches. For example, a subscriber may not receive a message posted on a synonymous topic to one they are subscribed to, unless they subscribe to both topics.

Event topic subscription. An *event topic subscription es* is a tuple (u, t_i) which represents the subscription of user u to topic $t_i \in ETS$. A user can of-course be subscribed to multiple event topics. We let ETS_u denote the set of all topics subscribed to by u called u's *topic subscription set.*

3.1.2 *Event Matching.*

Event matching involves first mapping event space descriptors into the event topic space prior to matching events to subscriptions. The mapping process involves using relationships in domain ontologies and rules to identify relationships between the attributes and values in an event descriptor and the event topic space. For example, if a *key* is *Material* and *value* is $Titanium$ implemented in a rule that captures the paths and relationships in the ontology as well as other domain knowledge, may link that pair to the manufacturing capability say *Asachining*. As another example, involving multiple attributes assume that we know if the product *material* is *PolyCarbonate* and the production quantity is not large (<1000), then the manufacturing capability required is *3D-Printing* . We can represent such rules using rule languages like SWRL or SPIN but we show the example using Stardog's rule language which is SWRL-like.

```
PREFIX cp: <http://resourcedescription.tut.
    fi/ontology/capabilityModel#>
PREFIX cc: <http://resourcedescription.tut.
    fi/ontology/commonConcepts#>
```

```
RULE :3DPrintingRule
IF {
   ?x.material a cc:PolyCarbonate .
   ?x.quantity < 1000 .
}
THEN {
   ?x.capability a cp:3DPrinting
}
```

Having the event topic space, rules and semanticized transaction data model linked to ontologies (not necessarily the same but integrated or linked), we are able to find associations on which reasoning can be based. A similar idea of using domain rules to infer manufacturing capabilities based on product specifications was proposed in [29]. The user-defined rule-based reasoning so far described is over and beyond the standard ontological reasoning using ontological classes, properties and inferencing axioms in the RDFS and OWL standards. For example, reasoning about $rdfs : subClassOf$, $owl : equivalentClass$, $owl : sameAs$ and so many others.

Event message descriptor mapping. Given the above semantic framework, an *event message descriptor* is mapped to a subset of topics in the topic space via semantic reasoning that exploits relationships between the terms in the *event message descriptor* and the topic space defined in ontologies and ontology rule-bases. More precisely, an *event message descriptor* $emp = (op, A = (k_a, v_a), M = (k_m, v_m), ...)$, an *event* is defined as $ev = (e, v \in 2^E TS, A, M)$ such that:

1. $e \in v$
2. $\forall v_i \in v$, there exists some k_{aj} or k_{ml} such that there is a *semantic derivation* to v_i
3. A, M constitute the original message contents

where *semantic derivation* means either derivation by axiomatic ontological reasoning or by rule-based reasoning.

In other words, an *event* is essentially as set of topics in the topic space derived ultimately from the transaction payload, via the event descriptor generation mechanism. For example, it is possible for our event descriptor example to be mapped to the set of topics { $CNC\ Milling$, $Thermoforming$ }.

Event matching. Given a user u's set of subscribed topics ETS_u it is straightforward to define a *match* for an event $ev = (e, v \in 2^E TS, A, M)$ if $v \subset ETS_u$. In other words, a match happens if a subscriber has subscribed to all topics in the event message (including both the topic designated as event identifier and the remaining topics). It is important to note that even though messages that were originally mapped to multiple topics via semantic derivation, are published on the message bus labeled with one out of the mapped topics, the entire set of mapped topics are included in the event payload and event subscribers evaluating matching conditions on the entire set not just based on e.

3.2 SMARTCHAINDB's Implementation Architecture Overview

In this section, we review the key components of the SMARTCHAINDB architecture and elaborate on their functionality.

Sequence Flow: Actors take part in a marketplace workflow wherein Requesters request for particular resources while Suppliers serve these requests with their owned assets. Step sequences are shown in Figure 3:

1. Requester prepares a REQUEST_FOR_QUOTE transactions and sends it to the blockchain through SMARTCHAINDB driver for its commit.
2. On receipt of the commit confirmation, the SMARTCHAINDB driver communicates with the semantic engine to infer features to publish the request onto Kafka.
3. SMARTCHAINDB driver then produces the request on one of the feature topics(inferred in step 2).
4. Supplier receives the request, if it offers the given feature, and determines whether it can fulfill it through the consumer manager.
5. If the match is found, Supplier may decide on triggering a BID transaction and send it to the blockchain to show its interest to the requester.

Admin Interface: The Admin interface provides utilities for configuring and managing a SMARTCHAINDB deployment including tasks like account creation and management, knowledge model configuration by importing ontologies and rule bases and, configuring the event topic space as well as system-level configurations e.g. cluster configuration for the blockchain nodes, event streaming service, and semantic engine, etc.

Blockchain Layer: is responsible for validating and committing transactions. It is built upon the BigChainDB[2] platform to provide blockchain and database characteristics. Each validator node runs a MongoDB database – for storing the blockchain state, and Tendermint – for blockchain consensus, as its core components. The SMARTCHAINDB server extends the storage model to incorporate the schema for the newly introduced transaction types. It also extends the execution layer with the validation logic for each new or updated transaction type.

Driver Layer: provides an interface for submitting transactions to the blockchain server. It coordinates the interaction between the blockchain, the semantic engine, and the event streaming platform. As a transaction is committed into the blockchain, on receipt of a transaction (request), the Driver performs the semantic mediation to infer the features and sends the transaction to the blockchain layer for the commit. It, then, posts the committed transaction (request) to the messaging platform to encourage bids from the potential suppliers.

Semantic Engine: contains the ontology and rule-based reasoners and interacts with the knowledge graph database to perform the semantic mediation. Semantic mediation maps the provided requested features to the configured Kafka topic features and, if needed, infers the requested features from the raw item specifications. It uses Stardog [11], a knowledge-graph platform, application-specific to store and query domain-specific ontologies and user-defined rules. As depicted in Figure 6, requested features (*Arc Welding* and *Screwing*) are mapped to their ontologically-related immediate parent feature topics, i.e. *Welding* and *Fastening* respectively, after reasoning the request payload. Whereas, features such as *Clipping* and *Riveting* are mapped to *Deforming Fixating* as these features are subclasses of *Elastic Deforming Fixating*, which is a type of *Deforming Fixating*. Semantic mediation is also required when the supplier registers the owned resources while not explicitly specifying their associated features. Wherein, the consumer manager communicates with the semantic engine to infer the underlying features through reasoning over the product specification information. When a new request *req* arrives on the SMARTCHAINDB Driver:

(1) Extract ReqMeta: a list of key-value pair objects that define a particular item within the request
(2) Maintain FeatureList: a collection of unique characteristics that the given request possesses.

Algorithm 1: Event Production with Request Semantic Mediation

```
1  ReqMeta = extractMetadata(formData);
2  Rules = fetchRulesFromStardog();
3  FeatureList = mediate(ReqMeta, Rules);
4  Function mediate(ReqMeta: Map, Rules: List) : List is
5      for every item in ReqMeta do
6          for every rule in Rules do
7              matchFound = item.apply(rule);
8              if matchFound is true then
9                  features = item.extractFeatures(rule);
10                 add features in FeatureList;
11                 continue with the next item;
12             else
13                 continue with the next rule.;
14         if no rule matched for the current item then
15             add Miscellaneous in FeatureList;
16             continue with the next item;
17     return FeatureList;
18 preparedTx = req.append(FeatureList);
19 sendTransactionToServer(preparedTx, callback);
20 callback.postedSuccessfully() {
21     randomTopic = Random.rand(FeatureList);
22     eventMessage = prepareEventMessage(preparedTx);
23     kafkaProducerDriver.produce(randomTopic);
24 }
```

(3) Fetch Rules, a list of ontological and user-defined rules, from the knowledge graph database (e.g. Stardog) to infer the embedded features of the given item.
(4) Iterate over the ReqMeta list and match the requested attributes with the available rules for populating the FeatureList (line 3-17 in Algorithm SMARTCHAINDB's Implementation Architecture Overview).
(5) Once all the items are processed and their respective features detected, attach the FeatureList to the final request and send it over to the SMARTCHAINDB Server as a domain-specific transaction. Upon the transaction commit, the request will be posted on only one of the Kafka topics from FeatureList to reduce the number of messages in the messaging subsystem (line 21-23 in Algorithm SMARTCHAINDB's Implementation Architecture Overview). Posting the event messages onto a single topic, instead of all FeatureList topics, will never lead to supplier missing the potential requests case since the supplier needs to fulfill all the requested features (FeatureList) to serve the given request. On receipt of the request, the supplier extracts and checks all the FeatureList features before making the subsequent decisions such as bid etc.
(6) Semantic Mediation also takes advantage of a locally maintained cache to lower the end-to-end latency. The cache stores requested features to inferred features mappings and assists in circumventing the mediation overheads for future look-ups.

Client Layer: This layer comprises the graphical user interface for interacting with the system. It enables the user to formulate transactions using ontology-driven menu forms, to add or to drop topic subscriptions and to intuitively interact with the subscribed notifications. In addition, it also contains Request Publisher, a simple Kafka producer to publish event messages onto Kafka topics, and Consumer Manager(discussed in detail later). This layer primarily communicates with the driver layer for interacting with other layers.

Consumer Manager: Every user in a supplier role with the registered resources runs a Consumer Manager to listen to their respective offering topics' messages. Consumer manager polls multiple subscribed topics to identify the potential request out of the incoming message stream. It extracts the received request details and tries to find the match between the requested features and the supplier's offerings. The matched potential requests are communicated to the client UI to enable the subsequent actions in the workflow (such as BID). The consumer manager, as a wrapper, extends the classical Kafka consumer to allow users to update the subscription space and to impose additional filters on the incoming requests. Such filters enable suppliers to only consider a small set of requests matching the given conditions. For instance, a manufacturer who subscribes to the "Drilling" offering topic can impose additional filters to only receive requests that require diamond drilling, i.e. a specific type of drilling. The instantiation of a supplier's consumer manager includes:

(1) Extract the required fields from the supplier's signing form, *form*:
    - Unique username or Id.
    - List of owned/offering resources and their respective parameters
    - List of conditions <attribute name, attribute value> for every offering topic, TopicConditionMap.
(2) Construct OfferingList and TopicConditionMap:
    - OfferingList: a list of features that the bidder's resources own.
    - TopicConditionMap: a map that will hold all the conditions associated with all the offering topics.
(3) Instantiate a Kafka Consumer, Consumer, that subscribes to OfferingList topics.
(4) Maintain MatchedRequests for the entire consumer lifetime. MatchedRequests is a list of transactions that the supplier can fulfill and can likely be the candidate for the bid.
(5) Consumer Manager supports on-the-fly changes to the subscribed Kafka topics space and TopicConditionMap.

The pseudocode for this process is shown in Algorithm 2's function subscribeOfferingTopics.

Algorithm 2: Consumer Manager: Supplier Registration/-Subscription and Event Matching

```
Class ConsumerManager:
    TopicConditionMap : Map;
    OfferingList : List;
    Consumer : KafkaConsumer;

    Function subscribeOfferingTopics(form : Map)is
        for every resource in form do
            if offerings specified in resource catalog then
                add offerings into OfferingList;
                continue with the next resource;
            else
                infer offerings over resource parameters
                  using the semantic rule engine(as achieved
                  in Algorithm 1) and populate OfferingList.;
        for True do
            Consumer.subscribe(OfferingList);

    Function eventMatch(req : Map)is
        FeatureList = retreiveRequestedFeatures(req)
         for every condition in TopicConditionMap do
            if ReqMeta satisfies condition then
                continue with the next condition;
            else
                skip the further processing;
                continue with the next request;
        if req satisfies all the required conditions and
         FeatureList subset of OfferingList then
            MatchedRequests.append(req);
            showMatchedRequestsUpdatesToUI();
        else
            skip the further processing;
            continue with the next request;
```

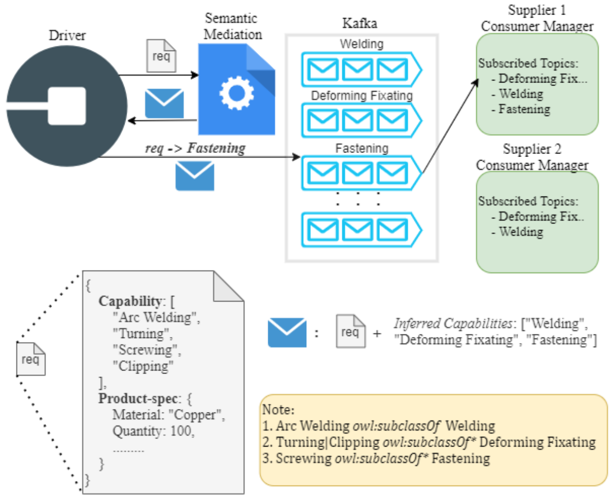
Figure 6 gives a schematic of the process. Two suppliers *Supplier*1 and *Supplier*2 and the subscriptions to event or offering topics for each are shown as *Welding*, *Deforming Fixating* and *Fastening* for *Supplier*1 while *Supplier*2 is subscribed to offering topics *Deforming Fixating* and *Welding*. The figure also shows an *event topic descriptor* (bottom left) with keys *Capability*, *Product−spec* and some corresponding values. The *event topic descriptor* is transformed into an event in the subscription space using *owl* : *subClassOf* relationships (bottom). This allows a mapping from *Screwing* to its nearest concept *Fastening*. The generated event derived from the event topic descriptor comprises of three capability topics : *Welding*, *Deforming Fixating* and *Fastening*. The event is published on the message bus with one of the topics (in this case *Fastening*) but it contains the entire subscription topic list within its payload. The message is received by all subscribers of the *Fastening* topic.

On receipt of an event message, a subscriber's consumer manager uses the entire payload to consider if they are subscribed to all topics in the offerings subscription list. Only if this is true, it is considered to be a match. In our example, *Supplier1* would be considered a match.

Event matching procedure

(1) Extract the required FeatureList, TransactionId, and request metadata ReqMeta from the req.
(2) Process request req:
  (a) Check if ReqMeta satisfy all the conditions:
  (b) Check whether all the requested features, FeatureList, can be fulfilled. In other words, check if FeatureList is a subset of OfferingList.
(3) Add req to the MatchedRequests if MatchFound is true and is communicated to the application UI.

The Event Subsystem layer consists of multiple semantic engine instances with the knowledge stores and Kafka clusters. These instances receive uniformly distributed requests through an upstream load balancer. The ecosystem has two primary actors – Normal user and Admin, wherein a normal user can act as `Requester` when they seek the required resources as well as `Supplier` when they offer their registered resources. Terms such as "Resources" and "Features/Offerings" are referred throughout the paper for supplier's physical assets and their general features respectively, eg. 3D-Printer[Resource] has *3D-Printing* feature. Features are used when in reference to the `Requester`'s requested features while offerings for the supplier's offered resource features.



```
{
  Capability: [
    "Arc Welding",
    "Turning",
    "Screwing",
    "Clipping"
  ],
  Product-spec: {
    Material: "Copper",
    Quantity: 100,
    .........
  }
}
```

Inferred Capabilities: ["Welding", "Deforming Fixating", "Fastening"]

Note:
1. Arc Welding *owl:subclassOf* Welding
2. Turning|Clipping *owl:subclassOf** Deforming Fixating
3. Screwing *owl:subclassOf** Fastening

Figure 6: Topic subscription through Supplier's Consumer Manager

Event Streaming Service: comprises multiple Kafka Broker and Zookeeper instances for hosting the Kafka cluster. The source code for the project can be found in two repositories: *https://github.com/Akash-Pateria/Driver-DE* and *https://github.com/bigchaindb/bigchaindb* (blockchain server). For the blockchain server, we have provided the link to the vanilla BigChainDB. The other repository has extensions and introduction made on the client and event subsystems.

4 EVALUATION

In this section, we present our experiment evaluations on the Semantic-enabled Event Detection service. We first describe our evaluation methodology and then discuss our results in details.

4.1 Evaluation Methodology

Simulation Dataset: Our experiment generates workload, in the form of REQUEST_FOR_QUOTE transactions, to evaluate and compare the proposed system with different approaches. We use MaRCO [27] to populate product specification information (required in transaction's asset attribute) with the ontological concepts to simulate real-world manufacturing-related requests. MaRCO, an OWL-based manufacturing resource capability ontology, describes the capabilities of manufacturing resources using a structural model shown in Figure 2. MaRCO is composed of different ontologies such as Product – models the product assembly, Capability – specifies the capability hierarchy and its associated parameters, Process Taxonomy – contains all the manufacturing processes with their relations, Resource – includes individual device and device combinations used to manufacture or to assemble a product, etc.

For building product specifications, we generate product key-value pairs using the concepts from Marco's Product ontology. We extract the manufacturing processes and capabilities, that are required for assigning a list of capabilities for every generated product, from Marco's Process Taxonomy and Capability ontologies. These capabilities are further used to identify whether a supplier can fulfill the given request. Kafka topics are also based on a set of capabilities, chosen in such a way that they cover the complete requested capability space and do not contain any equivalent or synonyms duplicates. We will reference this capability set as topic capabilities in this section.

We leverage the rdfs : subclassOf relationships from rdfs, to create the ontological rules and map the previously generated requested capabilities to the topic capabilities. For instance, a request with a capability *Milling* lands up on *Machining* topic (since *Milling* is ontological child of *Machining*) when *Milling* topic does not exist. Similarly, with the same relationship, the same request may get published on *Material Removing* topic if the *Machining* topic is not present. Ontological rules can also be used to reason capabilities out of the raw product information. In addition to these ontological rules, the proposed system is designed to make use of the domain-specific rules to infer capabilities from more complex product specifications. However, our experiments do not include such rules due to the lack of publicly available data rulebases (beyond some small test data). Here are a few basic domain-specific rules to determine capabilities given the material type and quantity:

$$\frac{Kafka\,topic\;(Capability) = \langle scdb : 3D\_Printing" \rangle}{\langle mr - cc : MaterialType \rangle = \langle scdb : PolyCarbonate \rangle, \langle scdb : quantity \rangle < 1000} \quad (2)$$

$$\frac{Kafka\,topic\;(Capability) = \langle mr - pt : Plastics\_Manufacturing \rangle}{\langle mr - cc : MaterialType \rangle = \langle scdb : PolyCarbonate \rangle, \langle scdb : quantity \rangle >= 1000} \quad (3)$$

Here, $mr - cc$ (MaRCO Common Concepts), $mr - pt$ (MaRCO Process Taxonomy), *scdb* (SMARTCHAINDB's native ontology) are name-space prefixes for different ontologies. *scdb* tries to combine various supplementary application-specific ontologies (such as ManuService for manufacturing use-case) to extend the knowledge base. In the example rules, the reasoner makes use of domain knowledge to determine the feature nuances from the raw request. It deduces that when mass production is required and the product material type is *PolyCarbonate*, the required product/service possesses *Plastic Manufacturing* feature. While it infers *3D-Printing* in the case when small-scale manufacturing is desired with the same material type.

Experiment Setup: For the purpose of measuring the system performance and effectiveness, we set up a private network of 16 machines with an Intel Westmere E56 Quad-core 3.46 GHz CPU, 8 GB memory, running 64-bit Ubuntu with kernel v4.15.0. We configure a Kafka cluster with 4 Kafka broker instances running on 4 different nodes for supporting the event streaming service. We install the Driver service on 12 nodes (also called client nodes), each impersonating a pseudo requester and supplier for simulating the marketplace-specific workload. Also, we set up a Stardog [11] server on every client node for serving the local ontological reasoning requests.

Each driver, as a requester, triggers 10,000 RFQ transactions in total throughout the experiment run with the maximum possible message rate. It prepares an event message with product specifications, as outlined in Simulation Dataset subsection, for every request. Besides creating request messages, the driver node also polls the subscribed topics for potential requests to mimic the supplier behavior. Each pseudo-supplier is assigned a random list of capabilities, as OfferedCapabilities, from the possible topic capability set.

In this experiment, we focused on two variants of our approach. This is because the few existing blockchain event frameworks are all rooted in the smart contract model. Due to the limitations of expressiveness and inflexibility, modeling the examples in our cases would not be possible in those approaches. We attempt to simulate one aspect of existing approaches which is the limited discriminatory ability of subscription spaces derived from such models. For instance, in Eventeum, all smart contract events go to one topic. So, for the example in Figure 1, the three types of events would be posted on the same topic. Therefore, we design our experiments to two fundamentally-different messaging paradigms – Shared Message Queue and Event Streaming System, under the discussed workload. We compare the supplier's (consumer) efforts in terms of usability and efficiency in both approaches. We set up Kafka with a single topic to imitate the shared message queue functionality and Kafka with multiple domain-specific topics for our event-detection system.

4.2 Experimental Results

Figure 7 shows the latency metrics of the single-topic queue and the proposed event-detection system. Here, the latency times measure the average time difference between the request creation and the time when the consumer has finished the processing of the same request. In the case of single-topic, every created request is produced on a common topic imposing the processing overhead on the consumer's side to determine the match between the supplier's offered capabilities and the requested ones. The consumers perform the ontological reasoning, using the Semantic Engine, to

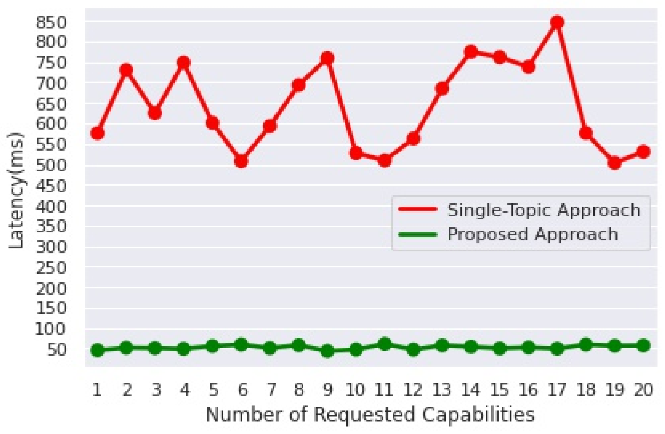


Figure 7: The performance comparison of proposed event detection system with the traditional single message queue on different number of requested capabilities.

Total message on a topic1, topic2, ..., topicN	Total Count
3669, 16662, 11162	31493
11162, 9114, 14341, 18314, 3669	56600
9114, 3669, 18314	31097
9006, 3652	12658
11162, 18314, 16662	46138
11162, 14341, 17484, 3669	46629
16662	16662
11162, 18314, 14341, 9006	52823
17484, 18314	35798
3669, 9006	12675
11162, 3669, 16586	31417
16586	16586

Table 1: Total messages processed by the Proposed Approach

map the requested capabilities onto the offered capabilities before taking any subsequent action (such as bids). On the contrary, the proposed system conducts this reasoning at the requester's end and only posts the request message onto the select topic capabilities that the interested suppliers are listening to. This design results in publishing the request message on the appropriate topics and significantly reducing consumer efforts.

The evaluation results show that the single-topic approach takes 10-12 times longer than the proposed one for any number of requested capabilities. The number of unrelated messages that the supplier needs to process is the major reason for high latency in the single-topic approach, i.e. all the request messages in the workload(12 drivers * 10,000 messages/driver). Conversely, table 1 shows the total number of messages processed by the consumer/supplier in the proposed approach. There are in total 12 consumers and for each consumer, we consider the total number of messages received on each of the subscribed topics. The maximum number of messages that a consumer process is 56600 that is less than half of the total number of messages (i.e. 120000) in the workload. In the production system with the single-topic approach, the supplier will have to manually consider all the triggered requests and take appropriate action (like Bid against the request or ignore it altogether), which is notably cumbersome from the usability standpoint.

5 CONCLUSION AND FUTURE WORK

The paper presents an approach for enabling event detection over complex data models stored on a blockchain and argues the value of such functionality for emerging applications in the blockchain domain. It demonstrates an implementation strategy that builds on an existing blockchain platform and presents through the results of an empirical evaluation, the advantage of the proposed approach when compared with what might be possible with today's blockchain event models. Some future directions to consider include handling more complex event descriptions beyond that which can be modeled using a conjunction of key-value pairs e.g. including disjunctive predicates.

6 ACKNOWLEDGEMENTS

The work was partially funded by NSF grant CNS-1764025.

REFERENCES

[1] [n.d.]. America Makes. https://www.americamakes.us/statement-on-covid-19/.
[2] [n.d.]. BigchainDB. https://www.bigchaindb.com/.
[3] [n.d.]. Ethereum Event API. https://bitquery.io/blog/ethereum-events-api.
[4] [n.d.]. Ethereum Event Explorer for Smart-Contracts. https://www.sw-engineering-candies.com/blog-1/Ethereum-Event-Explorer-for-Smart-Contracts.
[5] [n.d.]. Eventeum. https://github.com/ConsenSys/eventeum.
[6] [n.d.]. Ikea. https://www.ikea.com/us/en/.
[7] [n.d.]. An Introduction to Hyperledger. Technical Report. The Linux Foundation. https://www.hyperledger.org/wp-content/uploads/2018/07/HL_Whitepaper_IntroductiontoHyperledger.pdf
[8] [n.d.]. Macrofab: Digital Manufacturing. https://macrofab.com/digital-manufacturing/.
[9] [n.d.]. ManuService Ontology. http://manunetwork.com/Static/index-en.html.
[10] [n.d.]. A Next-Generation Smart Contract and Decentralized Application Platform. Technical Report. Ethereum. https://github.com/ethereum/wiki/wiki/White-Paper#ethereum
[11] [n.d.]. Stardog. https://www.stardog.com/.
[12] [n.d.]. Xometry. https://www.xometry.com/.
[13] 2018. BigchainDB 2.0 The Blockchain Database. (May 2018). https://www.bigchaindb.com/whitepaper/bigchaindb-whitepaper.pdf
[14] Maher Alharby and Aad van Moorsel. 2017. Blockchain-based Smart Contracts: A Systematic Mapping Study. CoRR abs/1710.06372 (2017). arXiv:1710.06372 http://arxiv.org/abs/1710.06372
[15] Tara Astigarraga, Xiaoyan Chen, Yaoliang Chen, Jingxiao Gu, Richard Hull, Limei Jiao, Yuliang Li, and Petr Novotny. 2018. Empowering Business-Level Blockchain Users with a Rules Framework for Smart Contracts.
[16] Bellaj Badr, Richard Horrocks, and Xun Brian Wu. 2018. Blockchain By Example: A developer's guide to creating decentralized applications using Bitcoin, Ethereum, and Hyperledger. Packt Publishing Ltd.
[17] S. Bajoudah, C. Dong, and P. Missier. 2019. Toward a Decentralized, Trust-Less Marketplace for Brokered IoT Data Trading Using Blockchain. In 2019 IEEE International Conference on Blockchain (Blockchain). 339–346.
[18] Prabal Banerjee and Sushmita Ruj. 2018. Blockchain Enabled Data Marketplace – Design and Challenges. arXiv:1811.11462 [cs.CR]
[19] LLC Bitquery. 2021. Bitquery Explorer. https://bitquery.io/
[20] Blockchain. 2021. Bitcoin Explorer. https://www.blockchain.com/explorer
[21] Chiara Braghin, Stelvio Cimato, Ernesto Damiani, and Michael Baronchelli. 2020. Designing Smart-Contract Based Auctions. 54–64. https://doi.org/10.1007/978-3-030-16946-6_5
[22] Etherscan. 2021. The Ethereum Blockchain Explorer. https://etherscan.io/

[23] Guido Governatori, Florian Idelberger, Zoran Milosevic, Régis Riveret, Giovanni Sartor, and Xiwei Xu. 2018. On legal contracts, imperative and declarative smart contracts, and blockchain systems. Artificial Intelligence and Law 26 (2018), 377–409.

[24] Eeva Järvenpää, Otto Hylli, Niko Siltala, and Minna Lanz. 2018. Utilizing SPIN rules to infer the parameters for combined capabilities of aggregated manufacturing resources. IFAC-PapersOnLine 51, 11 (2018), 84–89.

[25] Eeva Järvenpää, Minna Lanz, and Niko Siltala. 2018. Formal resource and capability models supporting re-use of manufacturing resources. Procedia Manufacturing 19 (2018), 87–94.

[26] Eeva Järvenpää, Niko Siltala, Otto Hylli, and Minna Lanz. 2017. Capability matchmaking procedure to support rapid configuration and re-configuration of production systems. Procedia Manufacturing 11 (2017), 1053–1060.

[27] Eeva Järvenpää, Niko Siltala, Otto Hylli, and Minna Lanz. 2018. The development of an ontology for describing the capabilities of manufacturing resources. Journal of Intelligent Manufacturing (06 2018). https://doi.org/10.1007/s10845-018-1427-6

[28] Eeva Järvenpää, Niko Siltala, Otto Hylli, and Minna Lanz. 2021. Capability matchmaking software for rapid production system design and reconfiguration planning. Procedia CIRP 97 (2021), 435–440. https://doi.org/10.1016/j.procir.2020.05.264 8th CIRP Conference of Assembly Technology and Systems.

[29] Eeva Järvenpäa, Niko Siltala, Otto Hylli, and Minna Lanz. 2019. Implementation of capability matchmaking software facilitating faster production system design and reconfiguration planning. Journal of Manufacturing Systems 53 (2019), 261–270. https://doi.org/10.1016/j.jmsy.2019.10.003

[30] Giuseppe Landolfi, Andrea francesco Barni, Gabriele Izzo, Elias Montini, Andrea Bettoni, Marko Vujasinovic, Alessio Gugliotta, António Soares, and Henrique Silva. 2018. An Ontology Based Semantic Data Model Supporting A Maas Digital Platform. 896–904. https://doi.org/10.1109/IS.2018.8710519

[31] D. Miehle, M. M. Meyer, A. Luckow, B. Bruegge, and M. Essig. 2019. Toward a Decentralized Marketplace for Self-Maintaining Machines. In 2019 IEEE International Conference on Blockchain (Blockchain). 431–438.

[32] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. Cryptography Mailing list at https://metzdowd.com (03 2009).

[33] Dan Robinson. 2017. Ivy: A Declarative Predicate Language for Smart Contracts. Technical Report. Chain, Inc. https://cyber.stanford.edu/sites/g/files/sbiybj9936/f/danrobinson.pdf

[34] Stanisław Strzelczak. 2015. Towards ontology-aided manufacturing and supply chain management–a literature review. In IFIP International Conference on Advances in Production Management Systems. Springer, 467–475.

[35] Tobias Sund, Claes Lööf, Simin Nadjm-Tehrani, and Mikael Asplund. 2020. Blockchain-based event processing in supply chains—A case study at IKEA. Robotics and Computer-Integrated Manufacturing 65 (2020), 101971. https://doi.org/10.1016/j.rcim.2020.101971