TuSoW: Tuple Spaces for Edge Computing

Giovanni Ciatto*, Lorenzo Rizzato†, Andrea Omicini‡
ALMA MATER STUDIORUM–Università di Bologna
Cesena, Italy
*giovanni.ciatto@unibo.it
†lorenzo.rizzato@studio.unibo.it
‡andrea.omicini@unibo.it

Stefano Mariani
Università di Modena e Reggio Emilia
Reggio Emilia, Italy
stefano.mariani@unimore.it

***Abstract*—Edge Computing is rapidly gaining traction in scenarios such as Cyber-Physical Systems and Web of Things. Whereas the Cloud hides heterogeneity of devices behind its standard interfaces and protocols, the Edge should deal with it, as well as with embracing openness and governing interactions. In this paper we propose TuSoW as a model and technology for bringing tuple-based coordination to the Edge.**

***Index Terms*—TuSoW, Edge computing, Coordination, Tuple spaces, LINDA, Web of Things**

I. INTRODUCTION

Edge Computing (EC) has been recently proposed as a *complementary* approach to Cloud Computing (CC) moving some of the computation and data processing closer to the "edge of the network"—that is, out of the Cloud and *into devices*, where data is firstly produced and the knowledge it conveys is ultimately consumed [1]. EC promotes an alternative view to a common Cloud-centred perspective over, for instance, pervasive and ubiquitous computing, where devices act as mere sensors (respectively, actuators) for some control software executing on the Cloud and perceiving (respectively, affecting) the external world through them.

Conversely, EC fosters the idea of partially delegating computation and data processing to Edge devices scattered in the environment, as in *Cyber-Phsyical Systems* (CPS) [2] or within the *Web / Internet of Things* (WoT/IoT) [3], [4]. The expected benefits range from increased efficiency and scalability – by reducing the overhead of repeatedly exchanging data between the Edge and the Cloud –, to a better tackling of non-technical concerns such as privacy and disclosure of sensitive data.

Of course, EC is no silver bullet for distributed computing: not every computation can be performed "at the edge", simply because some computation may require a global view on the state of the system—which is why EC and CC should complement each other. The sorts of computation that leverage local, *contextual* data are instead those that can be effectively, successfully, and more efficiently performed at the Edge. Among the many application domains sharing such a specific trait, CPS and the WoT/IoT are already experiencing a move to EC [5], [6], [7]. There, the many devices and services may exploit *local* computational resources to locally process data and to locally exploit such data for taking decisions and act.

Besides the aforementioned benefits, as for any new paradigm / technology, EC brings along new challenges to deal with so as to unleash its full potential. For instance, when the computation is moved from the Cloud to the Edge, one fundamental hypothesis is broken: homogeneity. As long as the Cloud *mediates interaction* among the many devices composing the IoT ecosystem, their *heterogeneity* is somehow hidden and flattened by usage of the Cloud API and the unifying abstractions it provides. On the contrary, when computation is retained at the Edge the need arises to let devices consistently maintain their shared context and communicate, regardless of the computational platforms they rely on.

Key here is the role of Cloud as *mediator of interactions*—which brings about the need for *coordination* in EC. Coordination is the discipline of enabling and constraining interactions among software components [8] or, more generally, of managing the dependencies among activities [9]. In the EC context, this translates to governing the interactions among devices and services, so as to fully realise the system functionality, through appropriate technologies backed by well-founded models.

In the attempt to contribute to dealing with the aforementioned challenges, in this paper we propose a layered reference model, architecture, and technology based on the concept of *tuple-based coordination* [10], aimed at supporting interaction among heterogeneous devices without strictly requiring mediation of the Cloud. Accordingly, Section II provides the background to understand tuple-based coordination; Section III describes TuSoW model, architecture, and implementation; Section IV discusses its further developments either already in place or forthcoming; Section V provides an exemplary scenario to clarify TuSoW target applications; Section VI concludes with some final remarks.

II. TUPLE-BASED COORDINATION

Among the many coordination models, tuple-based ones are the most studied [10], mainly due to their expressiveness, elegance, and flexibility. There, interaction among computational entities of a system are mediated by a *tuple space*, the data repository ruling *associative access* to information chunks called *tuples*. Tuples may represent messages by interacting components, data they intend to share, or a reification of the events of interest for them. In tuple spaces, interaction is governed by defining *how* and *when* agents are allowed to *insert*, *read*, or *consume* data.

978-1-7281-1856-7/19/$31.00 ©2019 IEEE

A. The LINDA model

LINDA – as the archetypal tuple-based coordination model [11] – lets interacting agents share tuples in tuple spaces by means of three primitive operations: `out`, to produce a tuple; `rd`, to read a tuple; and `in`, to consume a tuple. The set of the basic LINDA primitives could be interpreted as constituting a sort of API of a tuple space, whose essential features enable agents to synchronise their activities (hence coordinate): *generative* communication, *associative* access, and *suspensive* semantics.

Generative communication makes tuples exist in tuple spaces *independently* of their creators – that is, tuples sprout with an `out` and die with an `in`, regardless of the life of the producer –, and supports *reference* (or, *name*), *time*, and *space uncoupling* among interacting processes. Associative access makes tuples be accessed (i.e. either consumed or observed) through *templates* predicating over their content – e.g., regular expressions can work as templates for tuples represented as strings –, and supports dealing with *partial information* and *incomplete knowledge*: *getter primitives* (`rd`, `in`) expect a template as their argument and return a *matching tuple*. Finally, suspensive semantics makes getter primitives *wait to be served* until a tuple matching given the provided template becomes available – when the waiting primitive is eventually *completed* returning the matching tuple –, and supports *synchronisation* (ordering) of actions, hence coordination of activities.

B. Tuple Spaces at the Edge

Since the basic features of LINDA naturally fit the Edge Computing paradigm, tuple-based coordination could be suitable for governing devices interaction in EC application domains, such as CPS and WoT/IoT. Reference, space, and time uncoupling enhances *robustness* (fault-tolerance) w.r.t. connectivity issues and faulty devices, and also accounts for mobile computing scenarios and opportunistic networking, where devices deliberately disconnect and reconnect at will. Also, it favours *scalability* and load-balancing, by allowing devices and services to replicate and delegate computations at will without the need to re-wire requests routing among specific service instances. Associative access helps dealing with *heterogeneity* and *openness*, as partial knowledge of data structure is sufficient to get access to it, while "conversational continuity" [11] allow parties to become aware of the existence of others dynamically, hence join/leave the system anytime. Finally, suspensive semantics provides an elegant, simple yet expressive way of synchronising agents actions, that is, coordinate their activities. In the case of EC devices, this could translate to orchestrating provided services or executing distributed workflows, for instance.

In fact, the LINDA model can effortlessly emulate interaction patterns such as synchronous and asynchronous message passing, as well as remote procedure call, communication architectures such as master-workers and publish-subscribe, and also coordination constructs such as distributed locking and queues [11], all of which are widely used in the aforementioned EC application domains—as in distributed computing in general.

Nevertheless, despite LINDA being such a powerful model, a recent study [10] emphasised that only a few implementations are available to date, and even fewer can be considered as "off-the-shelf" usable. Furthermore, they are poorly suited for EC as they are usually tightly bound to a particular platform, data representation format, query language, communication protocol, etc. The TuSoW model and technology here proposed is specifically conceived to overcome such limitations, in the attempt to create an expressive, flexible, and modern LINDA-based "off-the-shelf" technology, fully geared to fit EC deployments.

III. TUSOW

TuSoW (Tuple Spaces over the Web) aims at providing a lightweight, modular, flexible, and highly interoperable implementation of LINDA-like tuple spaces, specifically suited for coordinating software agents in Edge Computing settings.

A. The TuSoW model

TuSoW is designed with REST [12] principles in mind. A TuSoW system is composed by a number of *services* [13] and a number of *clients*. Each service wraps a number of *named* tuple spaces, which are thus modelled as *collection resources*, and are referenceable through an URI of the form:

$$\langle \texttt{protocol} \rangle : //\langle \texttt{service\_name} \rangle / \langle \texttt{tuple\_space} \rangle$$

Tuples are modelled as (sub-)resources, too, but are not referenceable by URI, since clients can manipulate tuples only by means of *primitives*.

The specific modelling of primitives depends on the particular *remote API* of choice among the many extensively discussed in next section—e.g. CRUD operations, RPC. Generally speaking, primitives accept one or more representations of either tuples or tuple templates as input, and return zero or more *matching* tuple representations as output. It is worth noting that both templates and matchings are modelled as resources as well.

B. The TuSoW architecture

TuSoW is modelled according to the *Coordination as a Service* (Coord-aaS) architecture [13], where clients are the software entities requiring coordination services to achieve their goals. Clients may communicate with services, hence interact *through* tuple spaces, by means of a set of *remote API* geared toward different settings, including HTTP, WebSockets, gRPC, and MQTT. This is made possible by TuSoW modular architecture, whose main components are shown in Figure 1.

First of all, TuSoW consists of a core module defining the base types for tuples, templates, matchings, and tuple spaces, which is then specialised by different implementation modules—one for each pair of tuple-template languages. All implementations support *local* coordination of multiple concurrent processes, threads, or agents running on the same machine. Distribution is enabled by the *remote API*.

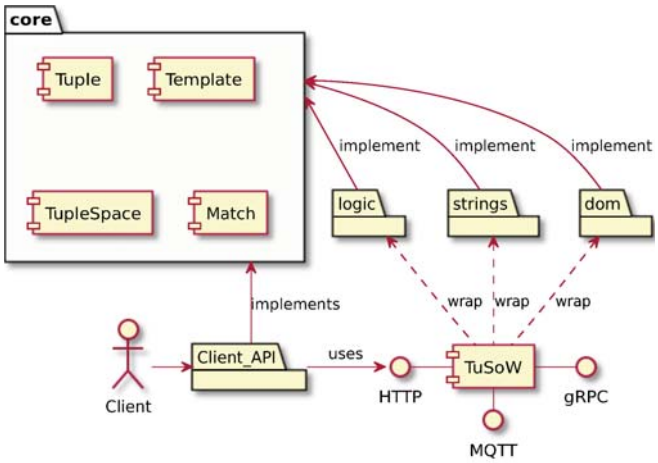

Fig. 1. TuSoW architecture. Circles represent TuSoW remote API. Clients API exploiting MQTT and gRPC remote API are not depicted.

Remote API reifies the Coord-aaS notion upon different technologies, targeting different EC settings. Currently, TuSoW supports three mainstream technologies widely used in CPS and WoT/IoT scenarios: HTTP, gRPC, and MQTT.

1) HTTP: The RESTful API is TuSoW warhorse. By exploiting HTTP as the transport protocol, it maps LINDA primitives onto HTTP standard CRUD methods, while letting clients choose either long-polling or WebSockets for retrieving suspensive operations results. Tuples, templates, and matchings (the data) are represented in a presentation format among YAML, JSON, XML. This is the reference remote API for generic Web-based applications requiring coordination.

2) gRPC: The gRPC API leverages a *remote procedure call* (RPC) interface defined by means of a platform-agnostic *interface description language* (IDL). By construction, gRPC supports both synchronous and reactive programming styles, letting clients consume operation results as streams—an appealing opportunity especially for WoT/IoT. Network encoding exploits Google *Protocol Buffer* platform-neutral, binary-based serialisation framework. This is the reference remote API for WoT systems featuring mobile devices.

3) MQTT: The MQTT API maps LINDA primitives invocations and completions to `PUBLISH` messages directed to several specific *topics*, for which both clients and services have subscribed. Topics are structured in such a way that primitives are invoked on the correct tuple space and responses are returned to the invoking client. MQTT default access control mechanism is adopted so as to hinder malicious behaviours. Data is again transported on the network using standard presentation formats such as YAML, JSON, and XML—yet in principle binary formats may be adopted, too. TuSoW provides ready-to-use MQTT brokers to propagate publication and subscription events, hence dispatching primitive invocations and completions between the clients and TuSoW. This is the reference API for IoT resource constrained or legacy devices willing to take advantage of tuple-based coordination.

4) Overall: Altogether, the technologies exploited by TuSoW make it versatile enough to be seamlessly deployed in either the Cloud or at the Edge: for the former, URI of TuSoW services are made publicly available through DNS; for the latter, some *ZeroConf* implementation – e.g. mDNS, Avahi, or Apple's Bonjour – can enable discovery by clients.

C. TuSoW data representation

The way TuSoW represents data (e.g. tuples) and queries (e.g. templates) is modular as well, and independent of the remote API. The idea of making LINDA primitives orthogonal w.r.t. the specific tuple / template language adopted is not new, as it has already been extensively discussed in [14], for instance. Nevertheless, to the best of our knowledge, no existing implementation allows for this, although in [15] an intuition concerning the potential of supporting several tuple / template languages is discussed.

In particular, TuSoW lets clients represent tuples with

- YAML, JSON, and XML formats — to target web-related contexts such as SOA and RESTful web services;
- first order logic (FOL) terms – in particular Prolog concrete syntax – to target intelligent agents and multi-agents systems;
- plain text documents as a fallback for contexts where structuring knowledge is neither possible nor desirable.

In principle, any other data representation format / language may be supported, thanks to TuSoW modular architecture. Each data representation format comes with one or more preferred ways for retrieving or querying data stored with that format, hence, to express templates and perform matching. For instance, JSON or YAML data can be queried through JsonPath, in the same way as XML data are queried through XPath. Plain text is often queried through Regular Expressions, whereas FOL terms may be matched against other FOL terms through unification. TABLE I summarises and exemplifies the tuple and template languages supported by TuSoW.

D. TuSoW Client API

Interoperability is not just a matter of data representation: TuSoW comes with a number of client-side modules, one for each mainstream platform – including JVM, .NET, JavaScript, Python – making it transparent to the adoption of any of the aforementioned remote API by client-side developers. TuSoW client API uses promises, callbacks, and coroutines – depending on the target platform – to let developers handle suspensive semantics in the most natural way for their programming language of choice.

TABLE I
TUSOW DATA REPRESENTATIONS.

Sort of tuple space	Tuple language	Templage language
textual	strings	regex
logic	terms	non-ground terms
objects	YAML/JSON	JsonPath
DOM	XML	XPath

1) Java & JVM: For instance, let us consider the JVM platform, and the (simplified) tuple space class supporting only the three canonical LINDA primitives for tuple management—namely `out` (a.k.a. `write`), `rd` (a.k.a. `read`), and `in` (a.k.a. `take`). In our example we assume strings are adopted for tuples representations, whereas regular expressions (a.k.a. regexp) are adopted as templates, leading to a *textual* tuple space. The corresponding Java type would then look as follows:

```
interface TextualTupleSpace {
  CompletableFuture<StringTuple> write(StringTuple t);
  CompletableFuture<RegularMatch> read(RegexTemplate tt);
  CompletableFuture<RegularMatch> take(RegexTemplate tt);
}
```

There, `StringTuple` is a wrapper class for Java strings, making them tuples in the eyes of our framework, whereas `RegexTemplate` is a wrapper for regular expressions, making them templates. A `tuple` can be created using, for instance, the following factory method:

```
StringTuple tuple = StringTuple.of(
        "name=John, surname=Smith"
);
```

whereas a matching regex template can be created by typing

```
RegexTemplate template = RegexTemplate.of(
        "name=(?<n>[a-zA-Z]+), surname=Smith"
    );
```

where `"n"` is the name of a *capturing group* matching any sequence of letters. A string tuple *matches* a regex template if and only if the wrapped string matches the wrapped regular expression—as in the case of the tuple and template above. The result of a matching operation is not a bare boolean value, but a more complex data structure carrying not just the matching result but also some handy methods aimed at quickly retrieving the portions of the matched string that are interesting for the template creator. `RegularMatch` is a particular sort of such data structures, whose instances represent the (possibly failing) matches among a string tuple and a regex template:

```
RegularMatch match = template.matches(tuple);
```

Such objects are returned by access primitives such as `read` and `take`, and their purpose is to let tuple spaces' clients quickly gather several useful information about their access primitive result, including *(i)* whether a matching tuple has been found in the first place – through the `match.isSuccess()` method –, and *(ii)* what is the value of a capturing group in the case of the matched tuple, given the groups name—e.g. `m.get("n")` should return `"John"`. This feature lets developers quickly access to specific parts of the matching tuple, leveraging associative access to easily gather the content even of complex tuples—as is the case of capturing groups for regexps, or bound variables for FOL.

`CompletableFutures` (a.k.a. *promises* [16]) are the preferred way in Java to handle asynchronous computations. By returning a `CompletableFuture`, we let each tuple space client choose whether to block waiting for a primitive to complete, or add a callback to the completion of the executed primitive. For example, in the following snippet:

```
tuplespace.write(tuple).get();          // blocking
tuplespace.take(tuple)                  // asynchronous
    .whenComplete(match -> {            // add callback
      if (match.isSuccess())
            println(match.get("n"));    // print "John"
    });
// no tuple has still been taken here
```

the `write` primitive is invoked in a synchronous way, whereas `take` is handled asynchronously. In particular, a callback is registered for handling the tuple obtained as soon as the operation completes. Notice that this snippet may be backed by either a remote TuSoW service or a local tuple space.

2) JavaScript and the Web: In JavaScript (JS), the Continuation Passing Style [17] is usually preferred. Asynchronous operations in JavaScript commonly return no value, but they accept one more argument instead, representing their *continuation*, that is, a callback to be invoked after they terminate. The aforementioned textual tuple space interface is then transposed in JavaScript accordingly, whereas `StringTuples`, `RegexTemplates`, and `RegulatMatchs` are created using a syntax which is analogous to the Java one. Nevertheless, usage of the API is quite different, as follows:

```
const tuple = StringTuple.of("name=John, surname=Smith");
const template = RegexTemplate.of("name=(?<n>[a-zA-Z]+),
     surname=Smith")};

tupleSpace.write(tuple,
    function(err, written) {
        tupleSpace.read(template,
            function(err, match) {
                if (match.isSuccess())
                    console.log(match.get("n"))
            });
    });
```

Notice that JavaScript supports promises and coroutines from v. 6: we plan to extend support to such programming styles, too.

3) Python and coroutines: Python natively supports coroutines through its `async`/`await` keywords, making asynchronous code very similar to ordinary sequential programs. The aforementioned textual tuple space API is then transposed in Python as follows:

```
class TextualTupleSpace:
    async def write(self, tuple):
        pass
    async def read(self, template):
        pass
    async def take(self, template):
        pass
```

whereas `StringTuples`, `RegexTemplates`, and `RegulatMatchs` are created with a Java-like syntax. Usage of the API is again different from previous cases:

```
tuple = string_tuple("name=John, surname=Smith");
template = regex_tuple("name=(?<n>[a-zA-Z]+), surname=Smith
    ");

await tuple_space.write(tuple)
match = await tuple_space.take(template)

if match.is_success:
    println(match["n"])
```

Since .NET supports coroutines, too – through C# `async`/`await` keywords and their non-blocking semantics – the .NET example looks very similar to the Python one, hence is omitted.

IV. Further Developments

TuSoW multi-platform support aims at providing an high-level interaction medium for heterogeneous clients, letting developers abstract away the inherent intricacies of low-level networking, thus focusing on how the many participants of a distributed application should interact. However, on the long term, TuSoW aims at easing the development of distributed applications along a number of other dimensions, discussed below.

A. Interaction patterns

On top of TuSoW client-side API, higher-level interaction patterns can be supported, using tuple spaces as their most natural mediators. To this end, a great standardisation effort has already been performed by the Foundation for Intelligent Physical Agents (FIPA[1]) by precisely defining a number of interaction protocols (IP). Despite FIPA IP [18] assume agents to interact through message passing, TuSoW models such protocols in terms of LINDA primitives, their content, and who and when should invoke or complete them, according to the specific protocol at a hand. Such models are then reified and encapsulated within ad-hoc classes keeping track of each protocol state and exposing callbacks in order to let developers easily specialise protocols—similarly to what JADE behaviours do [19].

B. Security and fault tolerance

By "security" here we mean that TuSoW tuple spaces should support authentication, access control, and confidentiality – if needed – for agents interacting through them.

A number of low-level protocols and conventions must be in place for modern distributed applications to support authentication and authorisation means—such as SAML[2], JWT[3], OAuth[4], etc. TuSoW aims at supporting all such mainstream standards for authentication / authorisation, in order for its clients to be able to interact with tuple spaces by only leveraging on their Google or Facebook accounts, or any other Single-Sign-On service. Furthermore, in real world scenarios it may be useful to constrain the pool of operations each client is allowed to perform upon TuSoW resources, i.e., tuples and tuple spaces. To this end, TuSoW adopts a Role Based Access Control (RBAC) subsystem letting administrators specify which roles or which specific agents are allowed to *(i)* know that a given tuple space exists, *(ii)* write tuples to it, or *(iii)* read or consume tuples from it.

Within the scope of TuSoW, we intend "fault-tolerance" as its capability of maintaining a *consistent* representation of the interaction space resulting from the agents interactions. In the EoT scenario, deploying TuSoW on a single machine may create a single point of failure. We believe the responsibility of maintaining the local, *shared* context consistent and available should be shared as well, within a particular neighbourhood. Based on that, TuSoW is designed to support *replication* of tuple spaces among a number of machines, following the *State Machine Replication* (SMR) approach [20], similarly to what happens for replicated databases or Blockchain technologies. The same idea could be exploited to make TuSoW tuple spaces fault tolerant by replicating them on a number of devices belonging to the same neighbourhood.

C. Deployment and inspectability

Nowadays, an emerging, effective way of releasing software products or services is through *containers*, and, in particular, Docker ones. The adoption of containerisation relaxes the need for transposing TuSoW codebase on most platforms, since most of them support containers. We thus let TuSoW be released as a container too, in order to make it easy to install and deploy on most architectures.

As far as TuSoW operations are concerned, we expect its tuple spaces will one day act as intermediaries for the complex interactions of many EoT devices. In this perspective, it is of paramount importance for administrators to be able to inspect the state of each tuple space handled by a TuSoW service.

By "inspect" we mean that administrators should be able to look into tuple spaces, thus observing – and possibly affecting – which and how many tuples they contains, as well as which and how many pending requests are waiting to be served in a given moment. To this end, each tuple space actually exposes an *inspectability* interface, too, other than the canonical LINDA one. The inspectability interface enables the creation of web-based graphical interfaces allowing administrators inspect and manipulate TuSoW tuple spaces through their browsers.

V. Case Study

As a simple case study meant to quickly hint at TuSoW capabilities, let us consider the case of a university library where students are allowed to stay and study before or after their classes. The library has a fixed amount of seats available for students to take; if the hall is big and crowded enough, it might be difficult to find a vacant seat. A local, TuSoW-based computing system is employed to track the library seats, aimed at *(i)* notifying students about seats available, and *(ii)* signalling the receptionist whenever a student has left her seat unattended for too long. The system leverages on a library-wise tuple space whose tuples represent seats, and on heterogeneous agents running on top of students smartphones, or tablets, and the receptionist's PC.

In such a setting, we imagine the following scenario: *(a)* A student arrives at the library. In order to get in, the student must identify him/herself through his/her smartphone. Simultaneously, the smartphone sends an `inp` request to the system to check if there are any available seats. *(b)* If the request succeeds, the system will consider the corresponding seat as "reserved", while the student's smartphone shows to him/her the path leading to the designated spot. *(c)* To get out, the student must identify him/herself again. Simultaneously, the smartphone `outs` a tuple stating that the seat is either free or still busy, depending on whether the student intends to leave

[1] http://fipa.org/
[2] http://saml.xml.org/saml-specifications
[3] https://tools.ietf.org/html/rfc7519
[4] https://oauth.net

or just have a break. *(d)* In the meanwhile, the receptionist is waiting, through a `rd`, for any busy-seat tuples. Whenever one is inserted, the receptionist PC starts a countdown. When time is over, the place is automatically freed by a sequence of `rds` and `outs`.

Even in such a simple setting, interoperability between an heterogeneous pool of technologies and flexibility of representations and usage API is highly desirable—if not a requirement. TuSoW client on the students side may be a Javascript web app, whereas the receptionist might exploit a Java desktop client, and the seat agent could consist of an embedded device communicating with the coordination service via MQTT. In such circumstances, a TuSoW-based solution could be well-suited to tackle heterogeneity while providing expressive abstractions simplifying the coordination of the involved agents.

VI. Conclusion

Edge Computing is increasingly becoming a complementary approach to Cloud-based processing for many application domains including Cyber-Physical Systems, Internet of Things, and Web of Things. There, heterogeneity of services and devices mandates for interoperability as a first concern of any technology aimed at enabling and governing interaction and communication among the distributed components of the system at hand. Accordingly, we presented TuSoW, a tuple-based model and technology for the coordination of things, devices, and agents at the Edge.

Other works exist that target the coordination of activities, processes, or computations at the Edge. For instance, in [21] an operative system for the EC – namely, EOS, the Edge Operative System – is discussed. EOS comes with a hierarchical structure where devices or nodes at the Edge dynamically discover each other by means of tuple spaces. While it represents another good example of the potential of tuple-based coordination for EC, it mostly leverages on tuple spaces to provide a distributed service discovery sub-system as a component of EOS—thus clearly differing in scope and goals w.r.t. TuSoW. Conversely, in [22] a number of stream processing technologies are surveyed, along with their reference model, the way they support elasticity, and, in particular, their adoption within the scope of the EC. While stream processing is actually a means for coordinating activities, processes, and computations, the focus here is mostly on data flows, where the interacting entities are basically neglected—unlike tuple-based coordination in general, and TuSoW in particular. In any case, both works agree on the importance of providing a broad technological support, in a pursue for interoperability. To this end, we believe TuSoW modular architecture and the number of languages, data representation formats, and communication protocols built-in makes it suitable for a great range of application domains and comfortable to a wide community of developers. Also, the envisioned further developments could strengthen its appeal and applicability to real-world modern applications.

References

[1] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[2] E. A. Lee, "CPS foundations," in *Design Automation Conference*. ACM, Jun. 2010, pp. 737–742.

[3] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[4] D. Guinard and V. Trifa, "Towards the Web of Things: Web mashups for embedded devices," in *2nd Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009) @ WWW 2009*, Madrid, Spain, 20 Apr. 2009.

[5] I. Stojmenovic, "Machine-to-machine communications with in-network data aggregation, processing, and actuation for large-scale Cyber-Physical Systems," *IEEE Internet of Things Journal*, vol. 1, no. 2, pp. 122–128, Apr. 2014.

[6] R. Want, B. N. Schilit, and S. Jenson, "Enabling the Internet of Things," *Computer*, vol. 48, no. 1, pp. 28–35, Jan. 2015.

[7] A. M. Rahmani, T. Nguyen Gia, B. Negash, A. Anzanpour, I. Azimi, M. Jiang, and P. Liljeberg, "Exploiting smart e-Health gateways at the edge of healthcare Internet-of-Things: A fog computing approach," *Future Generation Computer Systems*, vol. 78, part 2, pp. 641– 658, Jan. 2018.

[8] P. Ciancarini, "Coordination models and languages as software integrators," *ACM Computing Surveys*, vol. 28, no. 2, pp. 300–302, Jun. 1996.

[9] T. W. Malone and K. Crowston, "The interdisciplinary study of coordination," *ACM Computing Surveys*, vol. 26, no. 1, pp. 87–119, 1994.

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

[11] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, Jan. 1985.

[12] R. T. Fielding and R. N. Taylor, "Principled design of the modern Web architecture," *ACM Transactions on Internet Technology*, vol. 2, no. 2, pp. 115–150, May 2002.

[13] M. Viroli and A. Omicini, "Coordination as a service," *Fundamenta Informaticae*, vol. 73, no. 4, pp. 507–534, 2006.

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

[15] S. Mariani, A. Omicini, and G. Ciatto, "Novel opportunities for tuple-based coordination: XPath, the Blockchain, and stream processing," in *WOA 2017 – 18th Workshop "From Objects to Agents"*, ser. CEUR Workshop Proceedings, vol. 1867. Sun SITE Central Europe, RWTH Aachen University, Jun. 2017, pp. 61–64.

[16] D. P. Friedman and D. S. Wise, "The impact of applicative programming on multiprocessing," Indiana University, Computer Science Department, Technical Report 52, 1976.

[17] G. J. Sussman and G. L. Steele, "Scheme: A interpreter for extended lambda calculus," *Higher-Order and Symbolic Computation*, vol. 11, no. 4, pp. 405–439, Dec. 1998.

[18] S. Poslad, "Specifying protocols for multi-agent systems interaction," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 2, no. 4, Nov. 2007.

[19] F. L. Bellifemine, G. Caire, and D. Greenwood, *Developing Multi-Agent Systems with JADE*. Wiley, Feb. 2007.

[20] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 71–98, Jul. 2003.

[21] A. Manzalini and N. Crespi, "An edge operating system enabling anything-as -a-service," *IEEE Communications Magazine*, vol. 54, no. 3, pp. 62–67, Mar. 2016.

[22] M. D. de Assunção, A. Da Silva Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *Journal of Network and Computer Applications*, vol. 103, pp. 1–17, Feb. 2018.