Emerging Distributed Programming Paradigm for Cyber-Physical Systems over LoRaWANs

Danilo Pianini, Ahmed Elzanaty, Andrea Giorgetti, and Marco Chiani

***Abstract*—The growing interest around the cyber-physical systems (CPS), populated with open systems counting myriads of devices, is calling for new technologies both in telecommunications and software engineering with full integration among them. One of the most promising wireless communication technologies for the CPS is LoRaWAN, which enables long range transmission with low power consumption. Typical application scenarios include smart-homes, smart-cities, precision agriculture, and intelligent transportation. On the software side, novel paradigms are emerging to dominate the complexity introduced by the CPS with a large number of spatially distributed devices. Among them, aggregate computing is gaining traction, for it enables expressing the behavior of aggregates of devices by considering their ensemble as a single computational entity, allowing expressive space-time computations. In this paper, we introduce a software architecture which allows aggregate programming software to execute on a network of LoRa-communicating devices. We also provide an open source prototype implementing such architecture, which we use to study the current limitations of existing aggregate programming interpreters in resource-constrained scenarios. We conclude by drawing recommendations for developing such interpreters in order to pave the way to a more power- and data-efficient design.**

I. Introduction

Cyber-physical systems (CPS) are dynamic systems where software, networks, and computational elements are integrated to monitor and control physical systems in a collaborative and possibly distributed manner. This integration provides abstractions and design methods for the overall system. Possible applications include smart grids and cities, intelligent transportation, e-health care, and precision agriculture. The growing interest around the CPS is fostering development both in the telecommunications and computer engineering. In many cases, however, these two disciplines progress separately, hindering the full integration required by CPS. More precisely, novel software paradigms and techniques do not take full advantage, nor deal with the limitations of the latest advancements in telecommunication technologies, and the latter are often not designed with innovative (but not yet industrialized) software stacks in mind.

One of the notable lines of work in telecommunications engineering devoted to the CPS are Low Power Wide Area Network (LPWAN) systems, which offer communication links with high robustness to noise, enabling information exchanges on ranges longer than a dozen kilometers. This robustness is achieved at the expense of data rate, and whether or not this is an issue depends on the specific application: many scenarios of interest in the CPS are known to require limited data rates (e.g., smart metering, smart grid, data collection from wireless sensor networks (WSNs) for environmental monitoring), as demonstrated in existing analyses on data traffic requirements in several CPS and Internet of Things (IoT) applications [1]–[3].

Currently, the dominant technologies in unlicensed frequency bands are Sigfox and Low power, Long Range Wide Area Network (LoRaWAN). Sigfox systems depends on ultra narrow band technology, with bit rate ranging around 10 kb/s. Sigfox systems feature a limited throughput (15 packet/day per device with a maximum payload of 12 bytes) and a proprietary network stack (subscription is required for network access), two factors limiting the diffusion of the technology. On the other hand, LoRaWAN is a wireless network that implements the low power long range (LoRa) protocol in the physical layer, achieving ranges up to 15 km, with a maximum data rate of 11 kb/s (50 kb/s using Frequency Shift Key (FSK) modulation). Consequently, LoRa is considered one of the most promising enabling technologies for CPS, as witnessed by the growing count of applications relying on it, e.g., in smart energy production [4], e-health [5], and water network control [6].

At the same time, aggregate computing emerged as a novel programming paradigm aiming at providing an effective way to write composable behaviors for ensembles of devices. Its core idea is shifting the focus of the software designer from the single device to the aggregate of devices: the aggregate is considered as a single computational entity, whose computation evolves distributed data structures known as computational fields [7]. A comprehensive discussion of the computational model and programming abstractions provided by aggregate programming is beyond the scope of this work; however, the interested reader may refer to [8]. This programming paradigm is currently available in two programming platforms: *Protelis*[1] [9], an interpreted, duck-typed language hosted on the JVM; and *Scafi* [10], a Scala implementation of the aggregate computing semantics. The approach has been demonstrated to be effective for a variety of systems on diverse scale, ranging from service recovery [11], to WSN [12], to large urban events [7]. To the best of our knowledge, however, most of the aggregate programming applications presented to this

The authors are with the University of Bologna, Italy; A. Giorgetti and M. Chiani are also with IEIIT-CNR and CNIT, Italy (e-mail:{danilo.pianini, ahmed.elzanaty, andrea.giorgetti, marco.chiani}@unibo.it). This work is supported in part by project eCircular (EIT Climate-KIC), by project HyVar (EU Horizon 2020) under grant agreement No 644298, by ICT COST Action IC1402 ARVI, and by Ateneo/CSP project RunVar.

[1]Protelis is publicly available at www.protelis.org

day are either executed in environments which can provide high data rates (e.g. enterprise services [11]), or are validated via simulation (e.g. in [7], [12]), with little or no analysis on the load imposed on the network.

In this paper, we contribute to the state of the art by *i*) discussing the portability of aggregate programming on devices with access to a LoRa network interface; *ii*) proposing a networking architecture for aggregate programming interpreters that can cope with the LoRa requirements; *iii*) providing a reference, open source implementation of such architecture; and *iv*) drawing recommendations for further development and refinement of aggregate programming interpreters to better cope with limited data rates. Note that providing an aggregate programming interpreter suitable for *computationally* constrained nodes is not in the scope of this work: our aim is to leverage the existing LoRa devices as networking interfaces of nodes that already support aggregate programming.

To the best of our knowledge, this is the first work tackling the problem of enable execution of programming languages based on next-generation paradigms in environments with non-negligible network constrains. Other works discussing aggregate programming on a physical (non simulated) setup either do not feature a resource-constrained environment [11]; do not mention the actual execution platform [13], or focus on a general software architecture which is not specifically tailored for resource constrained environments [14].

The remainder of this paper is organized as follows. Section II overviews the LoRa communication system; Section III discusses how aggregate programming interpreters deal with networking; Section IV introduces our proposed architecture, whose prototype implementation is exercised in Section V providing insights on future development of aggregate programming interpreters. Finally, Section VI concludes the work.

II. LoRaWAN Overview

The foundational elements of LoRaWANs are end devices (EDs), gateways (GWs), and LoRa servers. End devices (sometimes called "nodes") and gateways form a star topology. Gateways communicate with a LoRa server, building an overall star of stars topology. In such network, each ED willing to communicate (uplink) broadcasts its message to all nearby GWs through single hub wireless links. All the GWs receiving the packet forward it to the server through any available networking backhaul (the protocol does not govern how the GWs should be connected to the server). In downlink communications (from servers to ED), the server selects the GW that deems best to deliver data to the recipient.

A. LoRa Air Interface: physical layer

The physical layer of LoRaWAN is called LoRa. It is considered as the air interface of the system, where two modulation schemes are adopted: FSK and a version of chirp spread spectrum modified to ensure a continuous phase modulation [15], [16]. Therefore, it is robust to non-linearities introduced by power amplifiers, to multipath fading, and to noise, and, as a consequence, enables long range communication with a low cost receiver [17]. One important parameter of the modulation scheme is the spreading factor (SF) which indicates the number of bits in each symbol and determines the symbol length. The SF ranges from 7 to 12, and provides a compromise between range and bit-rate, with higher SFs leading to lower bit-rates and longer ranges [18]. LoRa uses the unlicensed sub-GHz frequency Industrial, Scientific and Medical (ISM) band, e.g., $863 - 870\,\text{MHz}$ in Europe and $902 - 928\,\text{MHz}$ in the USA.

B. Multiple access scheme and device classes: MAC layer

In contrast to synchronous multiple access schemes, requiring high cost nodes (as in cellular networks), LoRaWAN specifies an asynchronous Aloha-like protocol for random access. For uplink communications, EDs randomly select a transmission slot based on their own communication requirements. Downlink communications must instead deal with an important trade-off: being able to receive messages requires the wireless hardware to be powered on, and, as a consequence, increases the power consumption; on the other hand though, reducing the occasions in which a message could be received negatively impacts the communication latency. Consequently, LoRa defines three device classes (A, B, and C) to fit different application domains with distinct requirements of power consumption and communication latency.

More precisely, Class A nodes open two receive windows at specific predefined times after their own transmission to allow for the reception of downlink messages from the GW. This class has the lowest power consumption, however, since downlink packets can only be received after a successful uplink, the latency in the downlink communication increases. For applications that require lower downlink latency, Class B devices can schedule additional downlink time slots (besides the two receive windows defined in class A) by synchronizing with the GW through periodically sent beacons. Finally, Class C devices are designed for applications requiring the lowest possible latency with relaxed constrains on the power consumption: EDs are always in receive mode except when transmitting themselves.

Besides the discussed trade-off between downlink latency and power requirements, LoRa has a number of other limitations. Usage of unlicensed bands decreases the costs, but negatively affects the system performance. In fact, the European telecommunications standards institute (ETSI) imposes restrictions on the maximum duty cycle for occupying each channel (i.e., the maximum time percentage during which a channel can be occupied by an ED) to prevent congestion. For some channels, the maximum duty cycle allowed in ISM band is less than 1%, severely limiting the network throughput, and restricting the number of generated packets per node [19], [20]. Additionally, the maximum payload size is restricted to 51 to $222\,\text{bytes}$ for $\text{SF} = 7$ and $\text{SF} = 12$, respectively. Reliability is affected by the count of ED in the system: since in pure Aloha protocol there is no coordination, packet collisions increase with the number of users. In order to guarantee successful

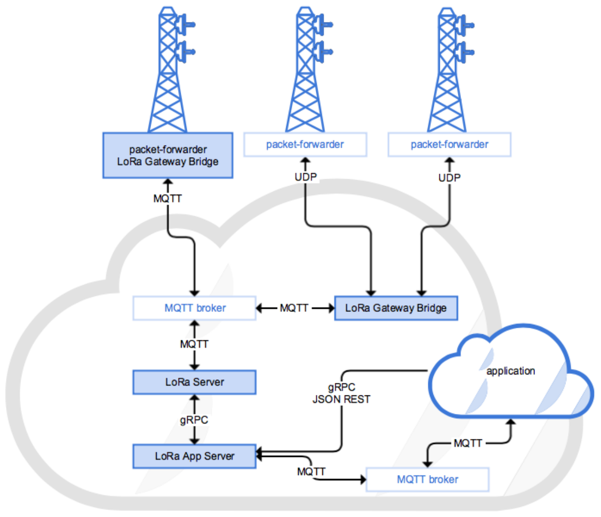

Fig. 1. LoRaServer architecture, blue boxes are part of the LoRaServer software stack. Multiple packet-forwarders (gateways) communicate with a LoRa gateway bridge, which abstracts the communication as JSON over MQTT. MQTT packets are forwarded to the LoRa server and in turn to the LoRa app server, with wich the actual application can communicate by subscribing and publishing to the appropriate MQTT topics. Image courtesy of the LoRaServer website.

packet transmissions, a confirmation message should be send by the GW. However, this is not always feasible as the GW is subject to duty cycle limitations as well. Moreover, acknowledgments decrease the throughput, as GWs can not receive while transmitting.

C. LoRa software stack: network, transport, application layers

Packets received by GWs get forwarded to the network server (NS) via UDP. These packets are usually not directly intercepted at the application-level: in fact, working directly with the UDP protocol complicates debugging, poses security challenges, and makes scheduling downlinks difficult, as it requires manual intervention to deal with node sessions, data de-duplication, authentication, and encryption. Fortunately, some available software products lower the burden for the developer, by providing the ability to abstract UDP over a higher level protocol, as well as the possibility to organize communication with publish-subscribe interaction pattern. Among them, LoRaServer[2] is particularly notable due to its modular architecture, feature richness, and permissive MIT license. The LoRaServer software architecture is depicted in Figure 1. It comprises three main components: *i*) the LoRa Gateway Bridge, which abstracts the UDP communication from GWs into JSON over MQTT; *ii*) the LoRa network server, which deals with cryptography, session, authentication, and other network-related aspects; and *iii*) the LoRa App Server, which implements an application server compatible with LoRa Server, offering abstractions like users, organizations, and LoRa-enabled applications. In order for this architecture to operate, additional external components are required: a database (e.g. Postgresql) where the Server can store its configuration, and an MQTT broker.

[2]https://www.loraserver.io/

III. Networking in Aggregate Computing

This section discusses the execution model of an aggregate programming interpreter, the support it expects from the networking subsystem, and introduces some of the relevant details of the existing interpreter implementations.

Aggregate programming's effectiveness at providing support for concisely expressing elaborate distributed algorithms is founded on two key aspects: *i*) the approach is composable and layered, namely functional blocks can be stacked and replaced, allowing for building increasingly complex systems; *ii*) all the low-level aspects of device interaction (messages, protocols, etc.) are abstracted away and hidden under-the-hood. However, for the purpose of providing actual networking support to aggregate programming interpreters, some details of what happens under the hood need to be understood.

Aggregate programming languages are rooted on the higher-order field calculus [21], in which devices undergo computation in non-synchronous rounds. In each round, a device sleeps for some time, wakes up, gathers information about messages received from neighbors while sleeping, performs an evaluation of the program, and finally emits a message to all neighbors with information about the outcome of computation before going back to sleep. Messages are broadcasted to neighbors: the *logical* network is purely peer to peer, but no requirements are set for *physical* network. In some scenarios (e.g. WSN) the logical and physical network may be identical, in other cases the former may be an overlay on the latter [13].

The way aggregate programming abstracts away network protocols is by providing a single primitive, `nbr`, which shares data and accesses shared data. A call to `nbr(foo)` implies a bidirectional communication: the local value of `foo` will get shared (sent away at the end of the round), and values computed in the neighborhood (and received by mean of previous messages) will be returned mapped with the origin device identifier. The interpreter allows for an arbitrary number of `nbr` calls: consistency preserved because, internally and transparently to the programmer, the shared data is decorated with "code paths", namely metadata tracing the sequence of calls and instructions that led to each `nbr` invocation. Consequently, payloads sent over the network are shaped as mappings between "code paths" and values, with one entry for each `nbr` call. The implementation of code paths is an internal detail of aggregate programming interpreters, often entirely unknown to programmers. Intuitively, however, in bandwith-constrained environments such detail could massively impact the network requirements, as there will be one code path sent for each piece of shared data.

IV. Architecture for a LoRa-enabled Aggregate Programming networking back-end

In this section we compare the LoRa network protocol and the network requirements of aggregate programming,

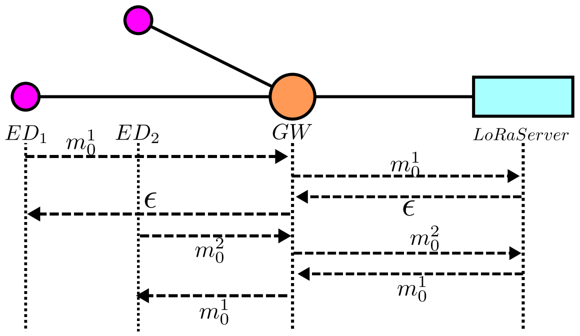

Fig. 2. Aggregate computing interaction rebuilt on top of a network subject to class A LoRa devices restrictions: the LoRa server is responsible for storing and relaying messages from / to nodes. In the figure, ϵ is an empty message.

proposing a software architecture that enables interoperability.

A. Scheduling and retransmission

At first glance, aggregate computing seems to be requiring LoRa class C devices. In fact, the computational model requires to be able to receive messages from neighbors while in sleep phase, and class C devices are designed to do so. Actually, however, the only requirement is that, at the beginning of the computation round, each node can retrieve a table where, for each neighbor, every code path is mapped to its value. The interpreter is agnostic with respect to the way such structure is built and maintained. Consequently, it is possible to devise a strategy which allows any kind of LoRa device to work, an example of which is provided in Figure 2: *i*)) EDs send their messages at the end of their round to the GW; *ii*) the GW forwards them to the server, which is in turn responsible for collecting and storing such messages; and *iii*) once a new message is received from an ED, the server initiates a downlink transmission during the receive window of that ED, forwarding all messages received by all other neighboring EDs since the previous uplink of the same node.

B. Segmentation

The aggregate programming interpreters expect to be enabled to send any amount of data through the communication channel. However, due to the LoRa limitations discussed in Section II, the payload sent over the network must get segmented (and joined on the other side) in order to fit the packet size. This issue is rather prominent and cannot get ignored even in a simplistic implementation designed to work only if programs are guaranteed to provide payloads smaller than a packet size: in fact, in the reasonable hypothesis that devices compute rounds at approximately the same frequency, the downlink response message size grows linearly with the number of neighboring devices, likely requiring segmentation also for very simple programs when a sufficient number of nodes joins the system.

C. Software architecture

With a consistent mapping between the LoRaWAN protocol and aggregate programming networking expectations, we can

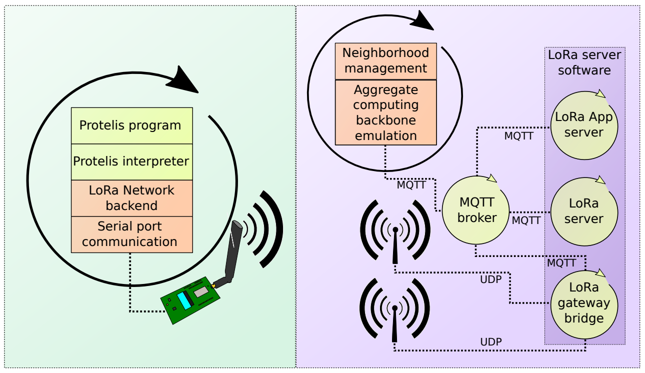

Fig. 3. The proposed architecture, comprising both the node side (left) and the server side (right). Circles with arrow depict active components. Novel components are depicted in orange, pre-existing ones in lime green. The LoRa server software is surrounded by a darker box. Two new modules bridge the existing Protelis interpreter with the serial interface exposed by a LoRa module. Multiple gateways forward their packets to a LoRa gateway bridge, which translates to MQTT. The LoRa server and LoRa app server deal with encryption and authentication issue. The novel Aggregate Computing backbone emulation builds the concept of neighborhood, receives and assembles uplinks from the node, computes and sends back downlinks.

include in the picture the existing LoRa and aggregate programming software, and identify a possible logical architecture to serve as foundation for an actual design. Our proposal is depicted in Figure 3. It includes two macro-modules: one, in charge of using the LoRa node as a network interface on the LoRa enabled devices; the other, sitting on a device which can communicate with the MQTT broker, in charge of orchestrating the communication among multiple nodes.

The "client side" components implement a Protelis `NetworkManager`, which serializes, segments, and sends the message produced at the end of the computation. In order to send it, it may need to open the communication multiple times in order to transmit the whole payload, as the packet size can be as small as 51 bytes. After each message sent, it tries to receive new messages, originally sent by neighbors, and relayed by the gateway. Because the size of messages to be received could be (and usually is) larger than the size of the message sent, and because (at least for class A devices) a downlink to the node can happen only after a successful uplink, the node may initiate a sequence of communications with no payload with the sole goal of successfully receiving all the data being sent from the gateway.

The "server side" components depicted in the right-hand side of Figure 3 could be living on the same server or be spread on different machines, given that: *i*) the LoRa gateway bridge can successfully communicate with the gateway(s); and *ii*) all the components can publish and subscribe to topics of the same MQTT broker. There are two key novel components: the backbone emulator which is in charge of implementing the communication protocol, and the neighborhood manager that encodes the strategy for building the logical network overlay on top of the physical star of star network native of LoRaWAN. In this architecture, the backbone emulator subscribes to a topic where the LoRa software infrastructure publishes messages received from registered nodes, appropriately decrypted and decorated with network information. Such data is reassembled in case of segmentation and then stored, along with relevant network information. Upon the reception of the first chunk of an end-round message, the backbone emulator queries the neighborhood manager for the current set of devices neighboring the sender, gathers their most recent message, serializes (and, if necessary, segments) them, then schedules a sequence of downlinks to send responses to the sender, realizing the protocol depicted in Figure 2.

The overlay configuration produced by the neighborhood manager may be changed in order to achieve the best trade off between network diameter and bandwidth usage. For instance, an aggregate application with small payloads, whose performance scale with the network diameter (such as a replicated gossip [22]), and with a number of devices in the order of tenths, could benefit from a fully connected topology. Other applications may require or work best with a topology defined on the basis of the physical distance between devices, and other may require different logical topologies.

V. Experiments with the Protelis LoRa-backend

We provide a prototype implementation of the proposed architecture, written in Java and Kotlin, for the Protelis language.[3] The aim of the prototype, besides providing a reference implementation, is to better understand the issues related to the interplay of the two technologies in analysis.

A. Hardware platform and software configuration

We deployed the system using a Microchip LoRa evaluation kit, comprising a single gateway and two class A end nodes. All the hardware was connected to the same desktop computer via USB for power and direct control. Both EDs and the GW expose themselves as serial devices, and can be configured and controlled directly from any USB-equipped machine (including Android phones). The gateway is connected with a point-to-point network to a server where the LoRa server stack, the MQTT broker, and the Protelis backbone emulator are executing. As stated in the introduction, we do not aim to provide a new aggregate programming interpreter which can execute on the computationally-limited microcontroller integrated on the LoRa devices. In our experiments, we used LoRa nodes as supplemental networking interfaces for computationally better equipped systems, e.g., Rasperry PI-like devices and smartphones.

B. Experiment design

The goal of our initial investigation is understanding the impact of using a LoRa communication backend on the Protelis interpreter, the limitations it imposes on programs, and collect a set of recommendations for future, LoRa-aware development of the interpreter internals. Hence, we used the payload size as metric for our investigation to measure how big are payloads generated by Protelis, using different techniques for encoding them. Payload size directly impacts the count of

[3]The prototype is available at https://github.com/DanySK/protelis-lora-mqtt

TABLE I
The application payload sizes, where all values are in bytes and the size of LZMA-compressed data is between parentheses.

Local program

	Java	Kryo	Elsa
Identity	135 (138)	27 (52)	2 (25)
Flatten	44 (67)	23 (48)	20 (44)
Json	9 (33)	4 (27)	3 (26)

Simple share program

	Java	Kryo	Elsa
Identity	297 (261)	71 (86)	57 (79)
Flatten	148 (142)	39 (60)	24 (49)
Json	29 (49)	24 (44)	24 (44)

Gradient program

	Java	Kryo	Elsa
Identity	740 (490)	368 (242)	439 (292)
Flatten	549 (355)	315 (209)	398 (260)
Json	360 (240)	357 (238)	356 (235)

Device count program

	Java	Kryo	Elsa
Identity	1536 (749)	898 (420)	1005 (497)
Flatten	1309 (611)	839 (384)	940 (466)
Json	1032 (479)	1028 (474)	1027 (473)

uplinks and downlinks required, with direct consequences on power and time-on-air, which in turn limits the round rate a Protelis program can be executed at.

Before being sent, the payload can get through three phases: preparation (e.g. to rip off ancillary data), serialization, and compression. Only the central phase is strictly required. We compared three preparations: *identity*, in which the phase is not performed; *flatten*, in which code paths are cleaned of data not strictly required to rebuild them, and the mapping between them and the computed object is reduced to a linear array of alternating values; and *json*, in which code paths are cleaned and transformed in base 64 strings, and the whole map is encoded in JSON. The latter strategy is fragile, as it requires all the data sent away to be JSON-serializable. We deployed three serialization mechanisms: the built-in *Java* serialization library; and two third party libraries, `Kryo`[4] and `Elsa`[5]. We tested both the size with uncompressed and LZMA-compressed (a variation of LZ77 [23]) payloads. We compared all the strategies across four Protelis programs with increasingly higher complexity: *Local* computes a local function, its payload serves the sole purpose of notifying neighbors that the node is still alive in the neighborhood; *Simple share* computes the minimum value across neighbors; *Gradient* builds a distributed data structure measuring distances from a source; *Device count* accumulates the count of devices along a spanning tree in a sink, and then broadcasts the result to every node in the network. Finally, we replaced the default `CodePath` class of Protelis, optimized for simulations, with a new version, featuring a smaller footprint.

[4]https://github.com/EsotericSoftware/kryo/
[5]https://github.com/jankotek/elsa

C. Results and discussion

Table I summarizes the results. Plain Java serialization is consistently worse than alternatives. Ripping unnecessary data off code paths is always beneficial, however, for very small payloads, using JSON provides the best results. Compression can greatly reduce the size, but it should only get applied if payloads are larger than, approximately, 140 bytes. Below this threshold, compression (at least LZMA) spoils results.

Implementation of code paths is critical to produce smaller payloads, and the existing implementations are not focused on the issue: we suggest further development to be devoted to reduce such size, in order for aggregate computing to become a viable programming alternative for networks of LoRa devices. Possibly, an analysis of the aggregate program could be performed in advance, allowing for enumeration of possible code paths, and great reduction in packet size. However, this is likely to interfere with some key features that depend on dynamicity, such as higher-order, code mobility, and meta algorithms. Another critical aspect is that every message is always sent in its entirety, regardless the differences with the previous ones. In many cases, the message is similar to its predecessor: deltas could lead to noteworthy improvements.

The current implementation uses three bytes for segmentation (used, respectively, as packet identifier, segment number, and total segments expected). This limits the maximum worst case payload to 12240 bytes; complex programs, however, are likely to exceed such threshold. A more sophisticated mechanism for segmentation should be devised to allow choosing between different segmentation mechanisms depending on the payload size, in order to save space for application data.

Finally, the limitations imposed by LoRa in terms of time-on-air and bandwidth call for better decoupling between round computation and network communication, which could execute concurrently, with the latter possibly at a frequency.

VI. Conclusion

In this paper, we presented the first attempt at executing aggregate computing programs on actual long range, low power communication networks. Our contribution includes a software architecture for executing aggregate programs on LoRa networks, as well as an open source prototype implementation. The prototype was exercised in order to better understand where to focus the efforts in order to improve the efficiency of aggregate programming in resource-constrained systems. Aggregate computing could be a viable programming paradigm in resource-constrained situations, however, the existing implementations should get improved by including *i*) a more size-efficient way of computing code paths, and *ii*) the possibility of using differences in place of whole messages.

References

[1] S. Grant, “3GPP low power wide area technologies,” *GSMA White Paper*, 2016.

[2] A. Giorgetti, M. Lucchi, E. Tavelli, M. Barla, G. Gigli, N. Casagli, M. Chiani, and D. Dardari, “A robust wireless sensor network for landslide risk analysis: System design, deployment, and field testing,” *IEEE Sensors J.*, vol. 16, no. 16, pp. 6374–6386, Aug. 2016.

[3] A. Giorgetti, M. Lucchi, M. Chiani, and M. Z. Win, “Throughput per pass for data aggregation from a wireless sensor network via a UAV,” *IEEE Trans. Aerosp. Electron. Syst.*, vol. 47, no. 4, pp. 2610–2626, Oct. 2011.

[4] H. Sherazi, G. Piro, L. Grieco, and G. Boggia, “When renewable energy meets LoRa: A feasibility analysis on cable-less deployments,” *IEEE Internet Things J.*, pp. 1–12, May 2018.

[5] F. Wu, J. Redout, and M. R. Yuce, “We-safe: A self-powered wearable IoT sensor network for safety applications based on LoRa,” *IEEE Access*, vol. 6, pp. 40 846–40 853, 2018.

[6] W. Zhao, S. Lin, J. Han, R. Xu, and L. Hou, “Design and implementation of smart irrigation system based on LoRa,” in *IEEE Globecom Workshop (GC Wkshps), Singapore, Singapore*, Dec 2017, pp. 1–6.

[7] J. Beal, D. Pianini, and M. Viroli, “Aggregate programming for the Internet of Things,” *IEEE Computer*, vol. 48, no. 9, pp. 22–30, 2015.

[8] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, and D. Pianini, “From field-based coordination to aggregate computing,” in *COORDINATION 2018, Proceedings*, 2018, pp. 252–279.

[9] D. Pianini, M. Viroli, and J. Beal, “Protelis: practical aggregate programming,” in *ACM Symposium on Applied Computing*, 2015, pp. 1846–1853.

[10] R. Casadei and M. Viroli, “Towards aggregate programming in Scala,” in *Proc. First Workshop on Program. Models and Lang. for Distr. Comput.* ACM, 2016, pp. 1–5.

[11] S. Clark, J. Beal, and P. Pal, “Distributed recovery for enterprise services,” in *Proc. IEEE 9th Inter.l Conf. on Self-Adaptive and Self-Organizing Sys.* IEEE, Sep 2015.

[12] D. Pianini, S. Dobson, and M. Viroli, “Self-stabilising target counting in wireless sensor networks using Euler integration,” in *Proc. 11th IEEE Inter. Conf. on Self-Adap. and Self-Organiz. Syst., SASO , Tucson, USA*, Sept. 2017, pp. 11–20.

[13] M. Viroli, R. Casadei, and D. Pianini, “On execution platforms for large-scale aggregate computing,” in *Proc. of the ACM Inter. Joint Conf. on Pervasive and Ubiquitous Computing*, Sept. 2016, pp. 1321–1326.

[14] R. Casadei and M. Viroli, “Programming actor-based collective adaptive systems,” in *Programming with Actors*, 2018, pp. 94–122.

[15] W. Hirt and S. Pasupathy, “Continuous phase chirp (CPC) signals for binary data communication-part I: Coherent detection,” *IEEE Trans. Commun.*, vol. 29, no. 6, pp. 836–847, Jun 1981.

[16] O. Seller and N. Sornin, “Low power long range transmitter,” Feb. 2016, U.S. Patent 9,252,834.

[17] H. Wang and A. O. Fapojuwo, “A survey of enabling technologies of low power and long range machine-to-machine communications,” *IEEE Commun. Surveys Tuts.*, vol. 19, no. 4, pp. 2621–2639, June 2017.

[18] *AN1200.22 - LoRa Modulation Basics*, Semtech, 2015.

[19] N. Sornin *et al.*, *LoRaWAN 1.1 Regional Specification*, Semtech, 2017.

[20] F. Adelantado, X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melia-Segui, and T. Watteyne, “Understanding the limits of LoRaWAN,” *IEEE Commun. Mag.*, vol. 55, no. 9, pp. 34–40, Sept. 2017.

[21] F. Damiani, M. Viroli, D. Pianini, and J. Beal, “Code mobility meets self-organisation: A higher-order calculus of computational fields,” in *Proc. Inter. Conf. Formal Techn, for Distrib. Objects, Compon., and Syst. FORTE , Grenoble, France*, June 2015, pp. 113–128.

[22] D. Pianini, J. Beal, and M. Viroli, “Improving Gossip dynamics through overlapping replicates,” in *Proc. Inter. Conf. on Coordination Models and Languages, Greece*, June 2016, pp. 192–207.

[23] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Trans. Inf. Theory*, vol. 23, no. 3, pp. 337–343, May 1977.