A Lightweight Network Discovery Algorithm for Resource-constrained IoT Devices

Article

Accepted version

S. Gaglio, G. Lo Re, G. Martorella, D. Peri

In Proceedings of 2019 International Conference on Computing, Networking and Communications (ICNC), Honolulu, HI, USA, 2019, pp. 355-359

NDS LAB - Networking and Distributed Systems
http://www.diid.unipa.it/~networks/ndslab/

A Lightweight Network Discovery Algorithm for Resource-constrained IoT Devices

Salvatore Gaglio*, Giuseppe Lo Re†, Gloria Martorella‡ and Daniele Peri§
Università degli Studi di Palermo - Dipartimento dell'Innovazione Industriale e Digitale (DIID),
Viale delle Scienze, Ed.6, Palermo, Italy
Email: *salvatore.gaglio@unipa.it, †giuseppe.lore@unipa.it, ‡gloria.martorella@unipa.it, §daniele.peri@unipa.it

***Abstract*—Although quite simple, existing protocols for the IoT suffer from the inflexibility of centralized infrastructures and require several configuration stages. The implementation of these protocols is often prohibitive on resource-constrained devices. In this work, we propose a distributed lightweight implementation of network discovery for simple IoT devices. Our approach is based on the exchange of symbolic executable code among nodes. Based on this abstraction, we propose an algorithm that makes even IoT resource-constrained nodes able to construct the network topology graph incrementally and without any *a priori* information about device positioning and presence. The minimal set of executable symbols to be defined on the devices is identified and simulation results for different topologies are reported.**

***Index Terms*—Symbolic processing, Executable code exchange, Distributed processing, Resource-constrained devices, Topology construction, Internet of Things**

I. Introduction

The Internet of Things (IoT) paradigm opened up the possibility that heterogeneous devices, such as sensor, actuators, and appliances be integrated into applications by the users themselves, through programming models, protocols and software platforms designed on the purpose [1]. Different protocols, which have been proposed as standards for the IoT, offer service discovery abilities [2] using both centralized or distributed infrastructures. Most of centralized implementations are based on request/response [3], [4] and publish/subscribe schemes [2], [5]. Mainstream approaches for distributed algorithms are often based on Distributed Hash Table (DHT) [6], [7]. To avoid the overhead of implementing efficient routing, flooding-based discovery is an alternative solution [8], [9].

However, these solutions are not easily implemented on resource-constrained devices [10]–[13]. Device and protocol heterogeneities hinder interoperability, especially when seamless integration between resource-rich devices, which support TCP/IP, and more constrained ones is required. Resource-rich devices, e.g. gateways, are often equipped with protocol translation capabilities [14]. While even reprogrammability of gateways has been addressed, horizontal integration with IoT resource-constrained devices does not seem to be solvable in the foreseeable future [2]. Efforts to embed IoT abilities in resource-constrained IoT devices have also been done. Rather than proposing novel algorithms, most of the approaches in the literature adapt existing protocols for resource-rich devices, to resource-constrained ones. This is the case of uBonjour [11] which provides service discovery above the Contiki operating system. Due to a thick software layer above the hardware that makes little memory available, just a restricted set of functionalities are implemented while optimization techniques for memory management are adopted. However, as stated by the authors, optimization generally increases code size and memory consumption. A similar approach proposes the implementation of traditional mDNS/DNS-SD protocols on limited devices [10] maintaining the fixed structure of messages. Optimizations mainly focus on network traffic reduction through the suppression of probing and advertising messages, as well as of discovery responses. Additionally, message compression techniques have also been considered [15]. The implementation of the Message Queue Telemetry Transport (MQTT) protocol without TCP/IP support, as in Zigbee-based Wireless Sensor Networks (WSNs), was also proposed [16]. In this case, WSN nodes are clients that communicate with a gateway acting as a server. The latter is connected to a traditional MQTT broker. To discover a gateway node, the client broadcasts a specific message and the gateway answers broadcasting its address. If another client receives the message, it can reply with the gateway address, if it holds such an information and if any other node, i.e. client or gateway, replied. As in our approach, plain text is used without any intermediate encoding but only for transmitting topic names. Gateway search, connection establishment and device registration, for instance, use messages with fixed structure. As most of the protocol logic is handled by the broker and the gateway, the client implementation is lightweight and requires limited resources, such as those available in Tmote and MicaZ WSN devices.

In this paper, interoperability in network discovery is addressed from a thing-centric perspective by adopting symbolic processing on resource-constrained objects. Instead of optimizing traditional protocols to work on limited devices, we propose a novel symbolic algorithm that can be executed by both resource-rich and resource-constrained IoT devices. The symbolic code of the protocol is included as network packet payload in the form of plain strings and is exchanged among different devices. This abstraction allows packets be not rigidly encoded, as symbols in the payload are efficiently executed upon receipt even on limited devices [17].

Two main advantages arise from adopting symbolic process-

DRAFT

ing and executable code exchange on IoT resource-constrained devices. The first is the possibility to reprogram nodes, even deployed ones, through wireless links, at runtime. This includes partial changes involving parameter's definitions or updates–e.g. timings, goals, conditions, values–as well as entire algorithms and system code. The second advantage is that the high-level protocol implementation is the same on different devices, while only the low-level coding of each symbol is specific to the underlying hardware.

The paper is organized as follows. The methodology based on symbolic programming and executable code exchange is outlined in Section II. Section III details the distributed algorithm for the construction of the network topology graph and its implementation through executable code exchange. Section IV describes the experimental setup and the simulation results. Finally, conclusions and future work are reported in Section V.

II. Approach

The proposed approach is based on a stack-based symbolic computational paradigm running on resource-constrained IoT nodes. Devices execute symbols that produce hardware and stack effects. Executing a symbol may imply the invocation of other symbols until terminal ones, which are defined in terms of machine code, are reached. A device can send code, in the form of a chain of symbols, which is executed as soon as it is received by another device. The code is transmitted as strings. The set of symbols that can be executed varies from node to node and depends on the tasks to be performed. With mainstream programming methodologies not enough space is left for applications beyond environmental monitoring, let alone symbolic processing or distributed computing, on resource-constrained devices. Our approach, instead, allows for effective high-level programming, ranging from hardware specification and verification [19] to distributed application development [17], even on tiny WSN nodes.

With no loss of generality, in the rest of the discussion it is assumed that each device in the network is equipped with a IEEE 802.15.4 radio. Indeed, the symbolic abstraction for executable code exchange is so simple that it can be easily adapted to other interconnections, the only requirement being the existence of an unique ID for each node.

Executable code exchange is used with the following assumptions:

- Symbolic code is embedded as payload in IEEE 802.15.4 data frames;
- MAC address is used as unique ID for nodes;
- During IEEE 802.15.4-compliant frame construction, special symbols are used as placeholders for numeric values, strings, fuzzy values, which are replaced at runtime;
- A new symbol can be defined to take place of a sequence of symbols, similarly to procedure definitions in other development models.
- The exchanged code can (i) modify the memory of one or more receivers, either RAM or Flash memory, by adding or removing the definition of symbols; (ii) modify the stack and, more generally, the hardware configuration of receivers; (iii) include the reply code in the sender outgoing frame.

The basic mechanism of executable code exchange is the execution of the `tell:` symbol. In case of radio transceiver-based devices, it creates a frame that complies with the 802.15.4-2003 standard. The sequence of symbols following the `tell:` are included as payload in the data frame. The payload construction stops once the `:tell` symbol is encountered. At this point, in fact, the frame is transmitted. For everything to work properly, the receiver's MAC address must precede the `tell:` symbol. Upon receipt, the node executes the payload of the incoming frame. For comparison, the ContikiOS implementation of the algorithm proposed in Section III spans more than 200 lines of code and requires to define in advance the packet structure that must be crafted, along with specific serialization and deserialization code, to encode the network graph.

Distributed algorithms can be implemented embedding reply code in outbound frames. For example, a node requesting a certain service from another, e.g. temperature value, can send both the code that commands the node to make the measurement along with the code to make it reply with the required measurement:

```
A301 tell:
temperature   reply tell: ~ :tell
:tell
```

The symbolic code addressed to node A301 is between the first `tell:` and the last `:tell`. The receiver executes `temperature`, which leaves a temperature reading on the stack. The inner `tell:` and `:tell` couple commands the receiver to create a frame with the value that the placeholder ~ pops from the stack, in this case the temperature reading that will be replaced at runtime after the measurement. The symbol `reply` is a self-reference that provides the sender address for the outgoing frame by extracting it from the received frame.

III. Network Discovery

In the following, we show how devices cooperate to build the network topology graph through executable code exchange. The architecture proposed in this paper relies on a node which acts as a bridge to the outside. Differently from centralized implementation, however, the bridge neither requires more resources nor holds a comprehensive view of the network topology and available services. In fact, the main task of the bridge is to provide Internet connectivity and to forward queries to the other nodes. In the case of the network discovery algorithm (Algorithm 1), the bridge act as the initiator after a request coming from the outside for the distributed mechanism that builds the topology tree (Figure 1). Each node executes broadcast code that makes it add some information to the packet payload, as shown in Figure 2. Whenever a leaf node receives the broadcast code and executes the code contained in it, the information held in the received packet, along with the code to process is sent back to the querying node. At the initialization, each device is marked as leaf (leaf flag set to 1).

DRAFT

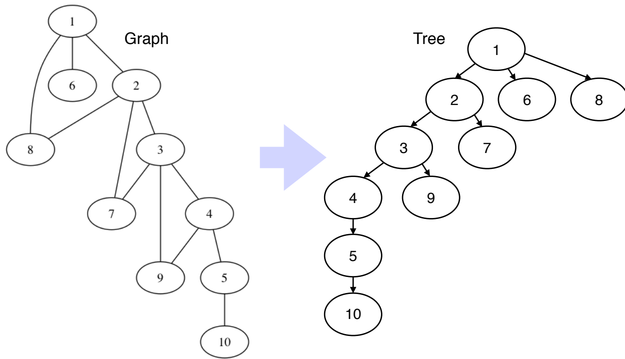

Fig. 1: Sample topology graph. Nodes broadcast in turn as shown in the topology tree. Nodes 2 and 8 are not neighbors in the tree as node 1 triggers node 8 by broadcasting the algorithm code before node 2 could do the same.

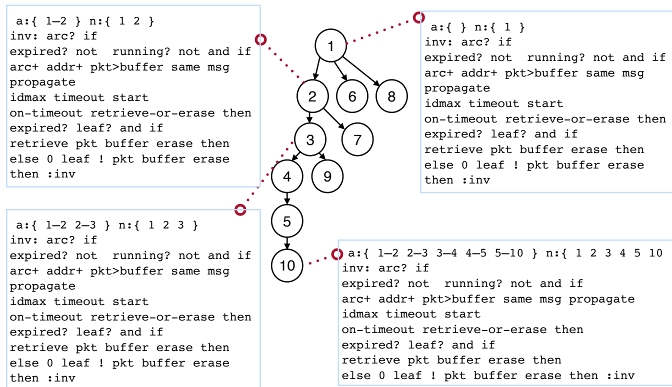

Fig. 2: Symbolic code broadcast samples. The invariant part of the code is enclosed between the `inv:` and `:inv` executable symbols.

The node ID, interpreted as time interval measured in milliseconds, is used as a time interval value to provide a deterministic ordering to the discovery process and reduce the risk of collisions in packet transmission. In the beginning (line 3), in fact, the code makes the node wait for a time interval equal to its ID in milliseconds. Then, if there is no description of an arc connecting itself to the sender in the packet and a timeout has not expired or is not running, the receiver inserts both the arc and the sender ID in the packet and stores the packet payload in the incoming packet list. Then the node starts a timer and broadcasts the old message augmented with the new information. At this point, even the original sender receives the message, possibly while its timer, which it has previously fired, has not expired yet. Since the receiver replies with the same message of the sender plus the information it has added, when the message is received and executed by the original sender the latter is made to check that the arc between the two nodes is present in the packet and its ID is in the route path. In this case the sender understands it is not a leaf, flushes the incoming packet list and sets the leaf flag to 0. Instead, when a node is a leaf, once it broadcasts the message, any node responds and once the timeout expires, the leaf flag is still set to 1. Therefore, it retrieves the packets using the reversed route path.

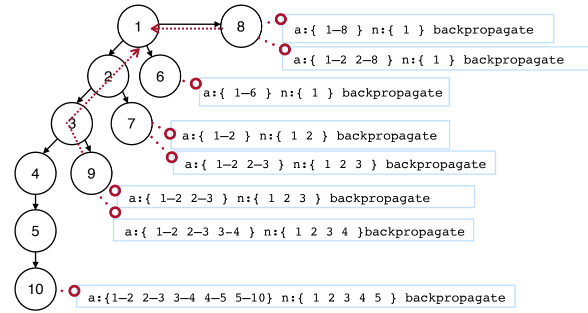

Fig. 3: Code sent by leafs during retrieval. Each leaf sends the arc list along with the route path to its predecessor. The message crosses the nodes in the reverse route path. Each internal node inspects the packet, erases its own address and backpropagates the same message. Once the message reaches the root, it matches its own ID in the route path and the arc list is used to update the tree with the discovered arcs (Figure 3).

The algorithm is coded as a sequence of symbols in the packet payload. Symbols are executed one after another once the packet is received. The bridge node broadcasts the following symbolic code:

```
a:{ } n:{ }
inv: arc?  if
  expired? not  running? not   and  if
     arc+  addr+ pkt>buffer
     same msg propagate
     idmax timeout start
     on-timeout retrieve-or-erase  then
  expired? leaf? and if
  retrieve pkt buffer erase  then
else
0 leaf !  pkt buffer erase then :inv
```

The arc list is enclosed between the a: , while the node list between the n: symbols. The symbols `inv:` and `:inv` mark the invariant part of the message. The symbol `arc?` leaves the true value on the stack depending if the arc between the sender and receiver is not present and the node is not already in the path. If so, the symbols `arc+` and `addr+` add the arc and the node ID respectively, in the list and leave the string with the addresses on the stack. Then the execution of the symbol `pkt>buffer` causes the temporary storage of the incoming packet payload in a buffer. The symbol `same` leaves on the stack the pointers to the invariant part of the message, while `msg` copies the code in the outgoing message buffer. Then, the symbol `propagate` uses the `tell: :tell` to build and transmit the message. The symbol `expired?` leaves a true value on the stack if the timeout has already expired, while the symbol `running?` checks if the timer is currently active. The symbol `leaf?` leaves a boolean value on the stack depending on the fact the leaf flag is set to 1. Such a value is then consumed by the symbol `if` during comparison. The `idmax` symbol leaves on the stack the value of the maximum ID. Considering a 16-bit ID, the maximum possible value is 65535 milliseconds. The symbol `timeout` sets registers for timeout and `start` make the timer run. The symbol `on-timeout` parses the next symbols and sets it as the ISR to be executed on timeout expiration. In this case, `retrieve-or-erase` checks the leaf flag. If it is already set to 1, then all the packets

DRAFT

stored in the incoming packet list are sent to their original senders along with the code to make them reach the node that requested the network topology. This is possible because packets hold the route path. Otherwise, just flushing is done. The symbol `!` assigns the value on top of the stack,– i.e. 0– to the symbol `leaf`. The symbol `erase` resets the packet buffer pointers thus restoring the initial situation. Another important advantage of the proposed approach is the possibility to change the algorithm at runtime by modifying the payload code. For instance, a topology partition can be constructed by making the node with even IDs to answer first leaving nodes with odd IDs the role of mere forwarders. This behavior is obtained with the code:

```
a:{ } n:{ }
my id even? if
  inv: arc?  if
     expired? not
     running? not  and  if
     arc+  addr+ pkt>buffer
     same msg propagate
     idmax timeout start
     on-timeout retrieve-or-erase  then
     expired? leaf? and if
     retrieve pkt buffer erase  then
  else 0 leaf !  pkt buffer erase then
else same msg propagate
then :inv
```

However, as the frame size is 128 bytes, and the list of nodes and arcs may be quite long, code can be compacted by defining more abstracted symbols on nodes. For instance, all the actions performed by a node can be executed by the symbol `topology`. This way, the code sent by the bridge node becomes:

```
a:{ } n:{ } topology
```

IV. Experimental Setup

To evaluate the algorithm, we considered three different topologies (Figure 4) and reported the number of messages exchanged as the number of nodes increased. Results were obtained using a simulator written in Prolog. The simulator includes the formal description about device architectures, network topologies, as well as the protocol functional specification. The system automatically generates the desired topology by positioning the nodes according to both its formal specification and the device communication range. The first topology, a linear arrangement, is a simple chain of nodes. In this case, the first node in the chain starts the discovery and the broadcast message is received by its sole neighbor. From this point on, each node adds its information and propagates the request to the following node. The last node, which is a leaf, retrieves the messages until the first node is reached. Complementarily, we considered a topology structure in which each node is connected with all the others. Finally, we considered a topology in which each node had at most four neighbors. Experimental results show that in the latter, the number of code exchanges is the highest. This is due to the fact that the number of leafs is greater than in the other topologies we considered. This implies that the number of message retrievals is substantial. This is shown in Figure 5. However, the number of code exchanges is prominently due to retrieval and node depth, as shown in Table I.

Algorithm 1 Network Discovery Algorithm

Before executing the procedure all nodes are initialized as leaves ($leaf \leftarrow 1$)

1: **procedure** NETWORK DISCOVERY
2: Check the existence of an arc between sender and self in the received packet
3: **if** arc is not present and not in the route path **then**
4: **if** timer not previously installed **then**
5: Store packet in the incoming packet list;
6: Add the arc between sender and receiver in the outbound packet;
7: Add sender ID to the route path of the outbound packet;
8: Wait for ID ms
9: Broadcast packet;
10: Install timer with timeout idmax (65535) ms;
11: **if** timeout expired and leaf=1 **then**
12: Retrieve all stored packets through
13: the reverse route path;
14: Erase all retrieved packets from the incoming packet list;
15: **else**
16: $leaf \leftarrow 0$;
17: Erase all stored packets from the incoming
18: packet list, if any.
19: **end**

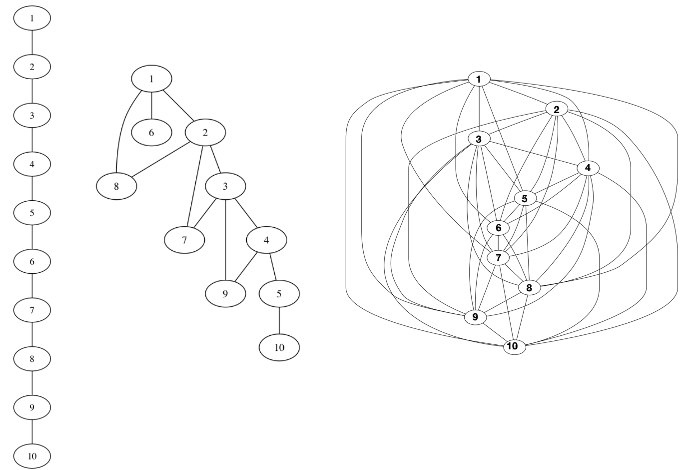

Fig. 4: Linear, 4-connected, and all-connected test topologies examples for networks of 10 nodes.

V. Conclusions

Centralized and distributed IoT protocols often require complex and multi-layered infrastructures. The implementation

DRAFT

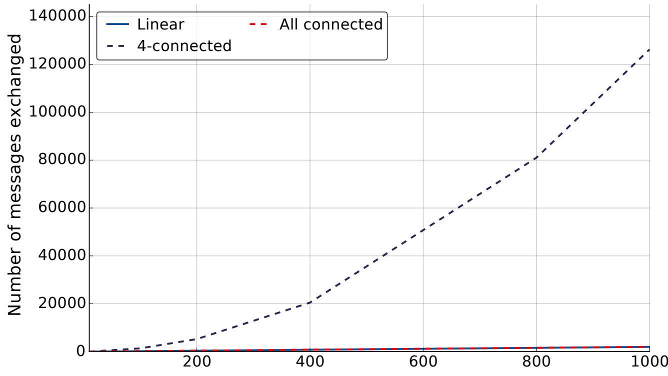

Fig. 5: Number of messages exchanged by nodes for different network topologies as the number of nodes increased.

TABLE I: Number of messages exchanged by nodes to propagate the request and retrieve the result for network tree discovery.

Topology	#Nodes	#Broadcast Messages	#Unicast Messages
Linear	10	10	9
	100	100	99
	200	200	199
	400	400	399
	800	800	799
	1000	1000	999
All-connected	10	10	9
	100	100	107
	200	200	222
	400	400	436
	800	800	799
	1000	1000	1047
4-connected	10	10	12
	100	100	1275
	200	200	5050
	400	400	20100
	800	800	80200
	1000	1000	125250

of these protocols on resource-constrained devices is often impracticable and simplified versions must be adopted instead. In this paper, an alternative approach is adopted which is based on the exchange of executable and symbolic code among devices with limited resources [18]. We described how network discovery can be undertaken even by resource-constrained devices without the need of complex infrastructures and architectures. We proposed an algorithm for network discovery and construction of the network graph on a remote node in a cooperative way by making nodes exchange directly executable symbolic code. Due to the symbolic approach, the algorithm can even be changed at runtime. Simulation results showed that unicast messages represents most of the network traffic, while the number of broadcast messages for code propagation equals node cardinality as each node transmits at least once. However, optimization and modifications can be sent at runtime. Future work will extend the applicability of the proposed methodology to the IoT scenario focusing on both communication and security aspects.

DRAFT

References

[1] K. J. Singh and D. S. Kapoor, “Create Your Own Internet of Things: A survey of IoT platforms,” *IEEE Consumer Electronics Magazine*, vol. 6, no. 2, pp. 57–68, April 2017.

[2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, “Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications,” *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2347–2376, Fourthquarter 2015.

[3] C. Bormann, A. P. Castellani, and Z. Shelby, “CoAP: An Application Protocol for Billions of Tiny Internet Nodes,” *IEEE Internet Computing*, vol. 16, no. 2, pp. 62–67, March 2012.

[4] A. J. Jara, P. Martinez-Julia, and A. Skarmeta, “Light-Weight Multicast DNS and DNS-SD (lmDNS-SD): IPv6-Based Resource and Service Discovery for the Web of Things,” in *2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, July 2012, pp. 731–738.

[5] O. Standard, “OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0,” http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-complete-v1.0-os.pdf, 2012, online; accessed 06 June 2018.

[6] Q. He, J. Yan, Y. Yang, R. Kowalczyk, and H. Jin, “A Decentralized Service Discovery Approach on Peer-to-Peer Networks,” *IEEE Transactions on Services Computing*, vol. 6, no. 1, pp. 64–75, First 2013.

[7] B. Fabian and T. Feldhaus, “Privacy-preserving data infrastructure for smart home appliances based on the Octopus DHT,” *Computers in Industry*, vol. 65, no. 8, pp. 1147 – 1160, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0166361514001274

[8] S. Guo, L. He, Y. Gu, B. Jiang, and T. He, “Opportunistic Flooding in Low-Duty-Cycle Wireless Sensor Networks with Unreliable Links,” *IEEE Transactions on Computers*, vol. 63, no. 11, pp. 2787–2802, Nov 2014.

[9] J. Kim, X. Lin, and N. B. Shroff, “Optimal Anycast Technique for Delay-Sensitive Energy-Constrained Asynchronous Sensor Networks,” *IEEE/ACM Transactions on Networking*, vol. 19, no. 2, pp. 484–497, April 2011.

[10] M. Mahyoub, A. Mahmoud, and T. Sheltami, “An optimized discovery mechanism for smart objects in IoT,” in *2017 8th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, Oct 2017, pp. 649–655.

[11] R. Klauck and M. Kirsche, “Bonjour Contiki: A Case Study of a DNS-Based Discovery Service for the Internet of Things,” in *Ad-hoc, Mobile, and Wireless Networks*, X.-Y. Li, S. Papavassiliou, and S. Ruehrup, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 316–329.

[12] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, “MQTT-S: A publish/subscribe protocol for Wireless Sensor Networks,” in *Communication Systems Software and Middleware and Workshops, 2008. COMSWARE 2008. 3rd International Conference on*, Jan 2008, pp. 791–798.

[13] A. Siljanovski, A. Sehgal, and J. Schönwälder, “Service discovery in resource constrained networks using multicast DNS,” in *2014 European Conference on Networks and Communications (EuCNC)*, June 2014, pp. 1–5.

[14] G. Tanganelli, C. Vallati, and E. Mingozzi, “Edge-Centric Distributed Discovery and Access in the Internet of Things,” *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 425–438, Feb 2018.

[15] R. Klauck and M. Kirsche, “Enhanced DNS message compression - Optimizing mDNS/DNS-SD for the use in 6LoWPANs,” in *2013 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, March 2013, pp. 596–601.

[16] O. Standard, “Message Queue Telemetry Transport Documentation,” http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf, 2014, online; accessed 06 June 2018.

[17] S. Gaglio, G. Lo Re, G. Martorella, and D. Peri, “High-level Programming and Symbolic Reasoning on IoT Resource Constrained Devices,” *EAI Endorsed Transactions on Cognitive Communications*, vol. 15, no. 2, 5 2015.

[18] ——, “DC4CD: A Platform for Distributed Computing on Constrained Devices,” *ACM Trans. Embedded Comput. Syst.*, vol. 17, no. 1, pp. 27:1–27:25, 2018. [Online]. Available: http://doi.acm.org/10.1145/3105923

[19] ——, “WSN Design and Verification using On-board Executable Specifications,” *IEEE Transactions on Industrial Informatics*, pp. 1–8, 2018.