PICO-MP: De-Centralised Macro-Programming for Wireless Sensor and Actuator Networks

Naranker Dulay
Department of Computing
Imperial College London
London, UK
n.dulay@imperial.ac.uk

Matteo Micheletti
Computer Science Department
University of Camerino
Camerino, IT
matteo.micheletti@unicam.it

Leonardo Mostarda
Computer Science Department
University of Camerino
Camerino, IT
leonardo.mostarda@unicam.it

Andrea Piermarteri
Andrea Piermarteri
Computer Science Department
University of Camerino
Camerino, IT
andrea.piermarteri@unicam.it

***Abstract*—Macro-programming advocates the use of high-level abstractions to specify distributed systems as a whole. However, macro-programming implementations are often centralised. In this paper we present PICO-MP, the first fully decentralised macro-programming middleware for wireless sensor and actuator network (WSAN) applications. PICO-MP incorporates a novel publish-subscribe service that can correlate events scattered across a WSAN using global formulae specifications that are automatically checked in a distributed fashion. PICO-MP has been implemented for the TinyOS operating system and validated on a case study that uses global formulae to improve energy efficiency (lifetime) of the implementation.**

***Index Terms*—Wireless Sensor and Actuator Networks; Distributed Computation; Macro-programming; Publish / Subscribe Paradigm; Energy Efficiency.**

I. Introduction

In order to build dependable WSAN applications, many aspects need to be addressed including the inherent challenges of distributed systems. For example, timing, communication and failure assumptions, security, the constraints of devices (cpu/memory/bandwidth/battery capacity), as well as the problems of the physical environment such as radio interference, bad weather, physical access. Having good software engineering tools for designing, reasoning, programming, testing/simulating, deploying, evolving and managing WSAN applications is also crucial.

In this paper we present a new approach for writing WSAN applications based on macro-programming (like Kairos [5], SOSNA [8], or [1]).

In our macro-programming system (PICO-MP) we define the global behaviour of an application using a novel and expressive first order filtering language that correlates distributed sensing and actuating events. Global formulae in this language are checked in a distributed fashion by a network of brokers. Each broker uses the global formula to calculate a local sub-formula (called projection) which can be locally verified.PICO-MP ensures that the distributed checking of projections produces the same results as the corresponding global formula. One major advantage of distributed checking is that it leads to provide opportunities to optimise the energy efficiency (lifetime) of WSAN applications. To the best of our knowledge, PICO-MP is the first macro-programming system for WSANs with a de-centralised implementation and optimisations for energy efficiency.

The remainder of the paper is structured as follows: Section II presents the related work. Section III introduces an example of a smart house system which will be used to describe our approach. Section IV provides an overview of our middleware. Section V explains the PICO-MP type-based data model, and Section VI details the PICO-MP subscription language. Section VII discusses the projection of a global formula and its distributed checking by the network of brokers. Section VIII provides an analytical model which compares the performance of PICO-MP decentralised checking against a centralised solution. Finally, Section IX provides conclusions.

II. State of art

In this section we present some macroprogramming frameworks and middlewares based on the pub/sub paradigm, and their main features. As a matter of fact, the high decoupling between information producers and consumers provided by the pub/sub paradigm is a key factor in WSAN application development, where nodes can be added, removed, or can simply switch off their radio. Pub/sub systems can be categorised as topic-based and content/filter-based ([3]).

MQTTs is a topic-based pub/sub that has been proposed by IBM [12]. This is an adaptation of MQTT, their famous pub/sub based middleware. MQTTs aims at coping with issues such as unstable connections, low energy and limited resources; its main advantage is the aggregation mechanism for notifications provided in order to save energy at network gateways. Although it guarantees three levels of quality of service, it lacks, by nature, in performance and flexibility: the real computation can only happen into a full MQTT broker, outside the WSAN, after a translation process from MQTTs.

Another topic-based pub/sub middleware is PS-QUASAR [2]. This is a lightweight middleware composed of a network maintenance protocol, a simple API, and a routing protocol. This last component is the most important part in its functioning: it performs an intelligent multicast message passing between publishers and subscribers using the information provided by the maintenance protocol. This middleware generates good results in terms of energy consumption, and it also provides advanced quality of service mechanisms. Unfortunately, it is limited to topic-based routing.

TinyCOPS [6] is a component-based middleware that supports both content and topic based pub/sub routing. Its main feature consists in its vast customisation possibility. This framework decouples the communication mechanism from the pub/sub broker functionalities, and it allows the addition of customised notifications and subscription paths; it also gives the chance to add service extensions. TinyCOPS uses a very expressive attribute-based naming scheme, giving a fine grained control over the pub/sub mechanism. Performance considerations must be done according to the selected components; however, TinyCOPS remains a very general-purpose middleware which is not so focused on energy efficiency.

PADRES [9] shares some interesting features with our proposal. PADRES is a distributed pub/sub content based middleware that has two main innovative features: the first one is the differentiation between atomic and composite events. Subscribing to a composite event means that the overlay network of brokers will only deliver information when a set of different events are matched together. Then, the other important feature is the distribution of the subscription checking task among many brokers in the network, in order to further decrease useless messaging. However, PADRES is implemented as a Java middleware which uses RMI and other powerful full-desktop tools, making it not suitable for lightweight IoT implementations. MERC [7] proposes an interesting decoupling technique of event matching and event routing. Its main goal is the optimisation of throughout and latency in full desktop environments. The authors in [10], [11] propose a middleware that automatically distribute global state machines. These define policies that can correlate data related to distributed sensors. While the approach can run on small battery powered sensor nodes the state machine language is less expressive when compared to the PICO-MP first order filtering language.

Our main goal is energy efficiency, that is why PICO-MP introduces a distributed event matching to reduce the dissemination (and the routing) of events. Subscribers can also be notified only about the truth value of properties they request, thus reducing messaging and improving energy efficiency.

III. Case study

Monitoring and automatic control of a building environment ([13], [4]) can include the following functionalities: (i) heating, ventilation, and air conditioning (HVAC) systems; (ii) fire alarms; (iii) centralised lighting control; and (iv) other systems, to provide comfort, energy efficiency and security. In order to demonstrate macro-programming for WSANs we will use a simplified home-automation case study with 2 applications, a fire alarm system and a home heating system.

The fire alarm system is composed of temperature sensors, smoke detectors and sprinkler actuators. When a temperature sensor reads a value that exceeds a specified threshold (e.g., 50C) and a smoke sensor detects smoke, all the sprinklers are activated.

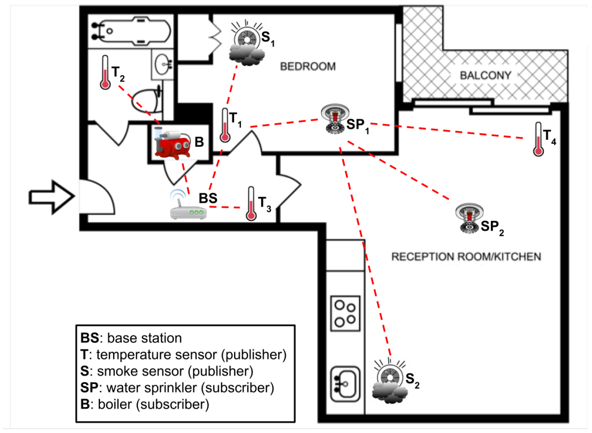

Fig. 1. Example home automation scenario

The automatic heating application is composed of different temperature sensors and various heaters. We provide the following automatic heating application:

- if all temperature sensors are greater than the maximum temperature T_{max} (e.g., 24C), the central heating system turns off.
- if all temperature sensors are less than the minimum temperature T_{min} (e.g., 16C), the central heating system turns on.

We use the home automation scenario of Figure 1 (which is mapped onto the PICO-MP deployment network of figure 2). The house is composed of the following four rooms: (i) the bathroom; (ii) a reception/kitchen; (iii) a bedroom; and (iv) a corridor. We have deployed one temperature sensor in each room. Sprinklers and smoke detectors are deployed in the bedroom and in the kitchen. The base station is in the corridor and the boiler inside a dedicated room. We assume sensors and actuators have some memory and processing capabilities (i.e., they can locally run some C code).

IV. Overview

PICO-MP is a type-based pub/sub system [3]. In a type-based subscription the declaration of a desired type is the main discriminating attribute by giving a coarse-grained structure on events (like in topic-based) on which fine-grained constraints can be expressed over attributes (like in content-based). Type-based pub/sub in this sense resembles the filtered topic model.

Publishers periodically send information to the PICO-MP infrastructure in the form of typed events. Subscribers express their interests in the form of first order logic formulae which predicate over sets of events. Each set is composed of events of the same type. Notifications are sent to subscribers whenever a formula becomes true or false. PICO-MP minimises the size of the notifications since subscribers only get a true/false notification; subscribers with particular needs can request, together with the notification, the set of events that triggered the notification.

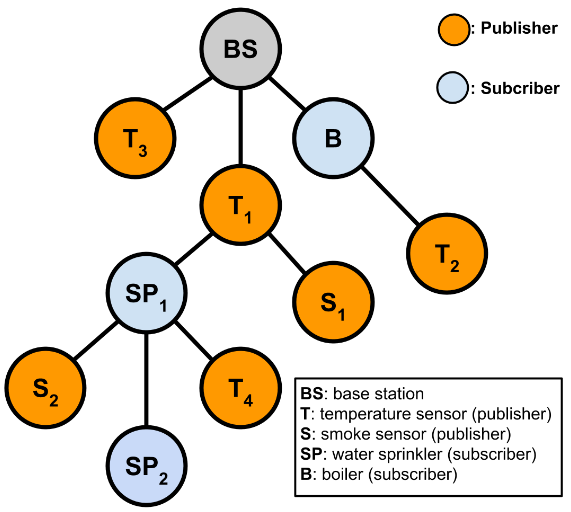
: Publisher
: Subcriber
BS
T3
B
T1
T2
SP1
S1
S2
T4
SP2
BS: base station
T: temperature sensor (publisher)
S: smoke sensor (publisher)
SP: water sprinkler (subscriber)
B: boiler (subscriber)

Fig. 2. PICO-MP deployment network of example home automation scenario

PICO-MP brokers are organised in a tree overlay network. Publishers and subscribers connect to exactly one broker. Brokers cooperate in order to manage subscriptions, events and distributed formula checking. Brokers ensure that all subscriptions flow up till the root node. A broker can prevent a subscription to reach the root when an equivalent subscription was already sent. This optimisation process can reduce the number of subscriptions that flow into the notification service.

The root forwards the global formulae down to the network of brokers in order to perform their truth check as close as possible to the event sources (i.e., the publishers). More precisely, each broker receives the global formula and generates a local projection of it. This contains parts of the global formula whose truth can be locally verified at the broker. The projection generation process assures that if a projection truth does not change, then the truth of the originating formula does not change too. When a projection switches its truth value the broker sends a message to its father, communicating which parts of the formula changed their truth and how. Eventually, the root will receive enough information to state that a certain global formula changed its truth value, and will notify the subscribers.

V. PICO-MP DATA MODEL

Our projection process requires to assign a type and a state to events, publishers, predicates and formulae. This is described in this section.

A PICO-MP event (i.e., a publication) is an instance of a structure that encapsulates various attributes of a primitive type. These include numeric (byte, short, int, long, double), boolean and string types. More precisely, a structure t is a tuple $t = (a_1 : type_1, \dots, a_n : type_n)$ where each a_i is an attribute of a primitive type $type_i$. The set $\mathbb{T}$ denotes the set of all possible structures; $\{t_1, \dots, t_n\}$ and $\{T_1, \dots, T_n\}$ are elements in $\mathbb{T}$ and the power set $\mathcal{P}(\mathbb{T})$. In the rest of the paper we use structure as synonymous of type. The set $\mathbb{E}$ denotes the set of all possible events; $\{e_1, \dots, e_n\}$ and $\{E_1, \dots, E_{2^n}\}$ are elements in $\mathbb{E}$ and $\mathcal{P}(\mathbb{E})$. An event e is an n-tuple $e(v_1, \dots, v_n)$ that is an instance of a type $t = (a_1 : type_1, \dots, a_n : type_n)$ where each v_i is the value of the attribute a_i. We can define a function $t_e : \mathbb{E} \to \mathbb{T}$ that assigns to each event e its type t, i.e., $t_e(e) = t$. In order to ease the notation we use t_e to denote the application of the function to the event e. We generalise this mapping to sets of events and types. We can define the function $T_E : \mathcal{P}(\mathbb{E}) \to \mathcal{P}(\mathbb{T})$ that assigns to a set of events $E = \{e_1, \dots, e_k\}$ the set of types $T_E(E) = T$ with $T = \bigcup_{i=1}^{k} t_{e_i}$. We use the notation T_E to denote the application of the function to the set of events E. The set $\mathbb{PUB}$ denotes the set of all possible publishers; $\{pub_1, \dots, pub_n\}$ and $\{PUB_1, \dots, PUB_n\}$ are elements in $\mathbb{PUB}$ and $\mathcal{P}(\mathbb{PUB})$. A publisher pub always generates events of the same type t. We use t_{pub} to denote that pub generates events of the type t. Let PUB be a set of publishers $\{pub_1, \dots, pub_k\}$ we use $T_{PUB} = \bigcup_{i=1}^{k} t_{pub_i}$ to assign a set of types to the set of publishers PUB. The function T_{PUB} allows us to relate the following set of types to a broker b:

- the *registration type set* T_{PUB_b} where PUB_b is a set that contains all publishers that are directly registered at b. For the sake of notation, we use T_b to denote T_{PUB_b};
- the *subtree type set* $T_{tree(b)}$ where $tree(b)$ is a set that contains all publishers in the subtree rooted at b except the publishers in T_b, i.e. the ones which are directly connected to b.

In our system, we say that each publisher pub has a state $S_{pub} = e$ where e is the last event that has been generated by pub. The set $\mathbb{S}$ denotes the state of the system; $\{S_{pub_1}, \dots, S_{pub_n}\}$ and $\{S_1, \dots, S_n\}$ are elements in $\mathbb{S}$ and $\mathcal{P}(\mathbb{S})$. We can define a function $S_t : \mathbb{T} \to \mathbb{S}$ that assigns to each type t a state $S_t(t) = \{S(t_{pub_1}), \dots, S(t_{pub_n})\}$ where $\{pub_1, \dots, pub_n\}$ are all the publishers of the type t in the system; we denote it with S_t.

Let PUB be a set of publishers $PUB = \{pub_1, \dots, pub_k\}$ we use $S_{PUB} = \bigcup_{i=1}^{k} S_{pub_k}$ to denote the state of all publishers inside PUB. In the rest of the paper we denote with S_b the state of the set of publishers which are directly connected to b, and with $S_{tree(b)}$ the state of the set of publishers which are connected to some broker in the subtree rooted at b (except for the publishers which are directly connected to b).

Finally, we define the function $filter : \mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{E}) \to \mathcal{P}(\mathbb{E})$. Given a set of types T and a set of events E, $filter(T, E)$ is the subset of E that contains all events of the type $t \in T$.

For instance, in our home automation case study (see Section III for details) we use the structure $temperature(value : int\ , unit : string)$ in order to abstract all sensors of temperature type. The attributes $value$ and $unit$ define the value and the unit of measurement of a temperature event. The event $e = (24, "celsius")$ can be generated by a sensor pub detecting a temperature of 24 $celsius$ degree. This is also the state of the publisher (S_{pub}). $S_{temperature}$ will be composed of the event e plus the states of all publishers of the type

```
1   Prenex_formula := Prefix Matrix
2
3   Prefix :=
4       Prefix QuantifiedVar | QuantifiedVar
5
6   QuantifiedVar :=
7       Quantifier Var '∈ S_t'
8
9   Quantifier := '∀' | '∃'
10
11  Matrix := Matrix '∧' Predicate | Predicate
12
13  Predicate :=
14      Var'.'Field relop Const
15      | Δ'('Var'.'Field')' '>=' Const
16      | PredicateName '('TermsList')'
17      | '¬' PredicateName '('TermsList')'
18
19  TermsList :=
20      'Var'.'Field'
21      | Constant
22      | Constant ',' TermsList
23      | 'Var'.'Field' ',' TermsList
24
25  relop := '<'  |  '>'  | '=='
```

Fig. 3. PICO-MP subscription language BNF grammar

$temperature$.

VI. PICO-MP SUBSCRIPTION LANGUAGE

The PICO-MP subscription language is defined by the grammar of Figure 3. A subscription is a first order logic formula in Prenex normal form (line 1 of the grammar of Figure 3) with no free variables. This is composed of a prefix part that is followed by a matrix. The prefix is a list of quantified variables (lines $3-4$ of the grammar of Figure 3) while the matrix is a conjunction of predicates (line 11 of the grammar of Figure 3). PICO-MP includes the following predefined predicates (although the definition of cumstom ones is supported):

- **Var.Field relop Constant**: $Var.Field$ is an attribute $Field$ of a quantified variable Var, $relop$ is a relational operator (<,>,==) and $Constant$ is a constant value. This predicate is used to compare the value of an attribute with a constant value (e.g., $x.value > 50$);
- **Δ(Var.Field) >= Constant** : $Var.Field$ is an attribute $Field$ of a quantified variable Var and $Constant$ is a constant value. This predicate is true when the variation of the value of the attribute is greater than or equal to a constant. This predicate is useful to get notification of a sensor reading when significant variations take place. A special case is when the constant is zero. In this case any variation of the sensor data is sent.

Constants and variable fields can be of the PICO-MP primitive types that are numeric (byte, short, int, long, double), boolean and string, while a quantified variable x ranges over the set S_t (lines $6-7$ of the grammar of Figure 3). We recall that S_t is the state of all the publishers of the type t. In the rest of the papers we use t_x^f in order to denote that the variable x has type t inside the formula f. A formula is checked every time a new event occurs. When a new subscription occurs, its formula is supposed to be false. Whenever a formula changes its truth value its subscribers are notified.

```
publisherTable = {(t_1,pub_1),...,(t_n,pub_n)}

brokerTable = {(t_1,b_1),...,(t_n,b_n)}

subTable = {(f_1,sub_1),...,(f_n,sub_n)}

projTable = {(f1,f1_∀,f1_∃[]),...,(fn,fn_∀,fn_∃[])}
```

Fig. 4. PICO-MP tables.

PICO-MP distributed checking procedure requires not only to have typed events but also to associate a type (or a set of types) to each formula and each predicate call.

A predicate call of a formula f defines a set of types. This includes the type of each variable that is used in the predicate call; the total number of different variable types is referred to as variety:

Definition 1: Let f be a formula $q_1 x_1 \in S_{t_1}, \ldots, q_m x_m \in S_{t_m}$ $s.t.$ $p_1 \wedge \ldots \wedge p_n$ where $q_h \in \{\forall, \exists\}$, x_h is a quantified variable of the type t_h (with $1 \leq h \leq m$). $p_1 \ldots p_n$ are predicate calls in the matrix of f. Suppose that the i-th predicate call of f is of the form $p_i(x_k.a_k, \ldots, x_q.a_q)$. The set of types of the predicate call p_i in f is defined as

$$T_{p_i}^f = \{t_{x_k}^f, \ldots, t_{x_q}^f\}$$

where $x_k \ldots x_q$ are all the variables in the predicate call p_i of f. The variety $v_{p_i}^f$ is equal to $|T_{p_i}^f|$.

The definition of a predicate call type allows us to assign a set of types to a formula f. This is defined with $T_f = \bigcup_{i=1}^n T_{p_i}^f$ where $\{p_1, \ldots, p_n\}$ are all the predicates in the matrix of f.

VII. PICO-MP DISTRIBUTED NOTIFICATION SERVICE

In this section we discuss in details the flow of information in our middleware, from publishers to subscribers.

A. Event publication and event type management

Each broker b (note that the root is only considered a particular case of broker) implements the registration type set T_b and the subtree type set $T_{tree(b)}$ by using a `publisherTable` and a `brokerTable`, respectively (see Figure 4). A broker uses these tables in order to obtain a projection from a global formula. A `publisherTable` of a broker b contains a tuple (t, pub) when a publisher pub of the type t registered at b. A `brokerTable` of a broker b contains a tuple (t, b) when one of its child broker b is the root of a subtree that contains publishers of the type t.

The `publisherTable` and `brokerTable` tables are kept updated by using the registration, unregistration, advertisement and unadvertisement PICO-MP messages (as shown in Figures 5 and 6). A publisher pub of the type t, registers to a broker b by using a `registration(t,pub)` message. This registration is used by the publisher pub to declare the type of events it generates (i.e., t). The broker b receives the registration and adds (t, pub) to its `publisherTable`. Effectively the broker recognises the publisher as its child and keeps information about its address and the type of events it generates. When the publisher pub exits the system, it must send an `unregistration(pub)` message

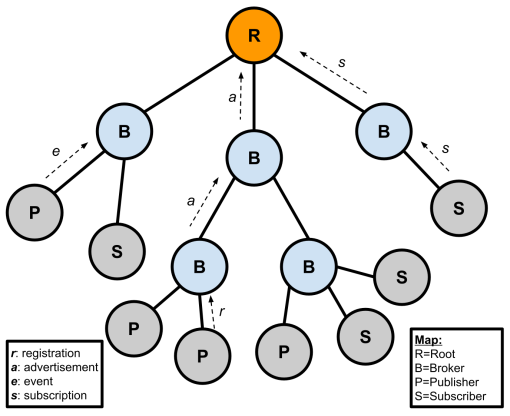

Fig. 5. Messages populating PICO-MP data structures.

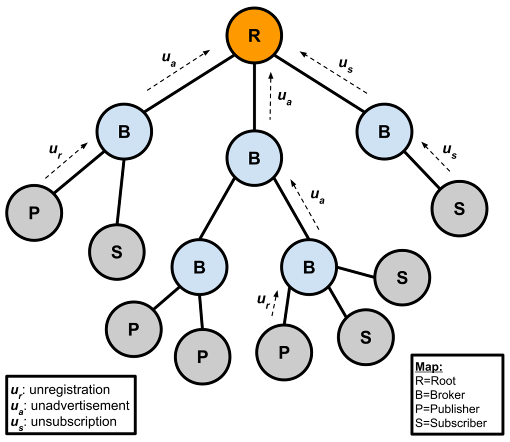

Fig. 6. Messages emptying PICO-MP data structures.

to its broker b. This message is received by b that deletes the entry (t, pub) from its `publisherTable`. A broker b can send messages of the type `advertisement(t,b)` to its parent broker $par(b)$[1]. This uses the advertisements to keep its `brokerTable` updated. More precisely, when $par(b)$ receives the message `advertisement(t,b)`, it adds the entry (t, b) to its `brokerTable`. A broker b can send an `unadvertisement(t,b)` message to its parent $par(b)$ when it cannot receive anymore events of the type t. This happens when all the publishers in $tree(b)$ (of the type t) unregistered. When $par(b)$ receives the message `unadvertisement(t,b)`, it removes the entry (t, b) from its `brokerTable`.

B. Subscription management

A broker b and a root r use the `subTable` (see Figure 4) in order to keep information about subscriptions that originated in their sub-tree.

The `subTable` is kept updated by using the subscription and unsubscription PICO-MP messages (as shown in Figures 5 and 6). A subscriber sub can send a `subscription(f,sub)` message to a broker b. When b receives the message `subscription(f,sub)` its `subTable` (see Figure 4) is updated with the entry (f, sub). Subscriptions are always propagated till the root (i.e., a broker always forwards the message `subscription(f,sub)` to its parent).

A subscriber sub which is no longer interested in a certain formula f can send an `unsubscription(f,sub)` message to its parent $par(sub)$. When b receives this message it removes the entry (f, sub) from its `subTable`. Unsubscriptions are always propagated till the root (i.e., a broker always forwards the message `unsubscription(f,sub)` to its parent). An optimised version of PICO-MP can use equivalence amongst formulae in order to minimise the subscription flow. More precisely, a broker does not forward a subscription formula f_1 that is equivalent to a subscription f that was previously sent. When the broker receives a notification for the subscription formula f also produces a notification for the equivalent formula f_1.

[1]For the sake of presentation we denote with $par(pub) = b$ the broker b where the publisher pub is directly connected. We denote with $par(b) = b_1$ the broker b_1 that is the father of b in the broker notification tree.

C. Projection generation process

A root always forwards down to the broker tree each new subscription f. When a broker b receives the formula f, it can perform the projection procedure of Figure 7. This projection decomposes f into a set of local formulae, i.e., a $f_\forall$ formula and an array $f_\exists$ of formulae. In the rest of the paper these formulae are referred to as projections. The $f_\forall$ projection contains all predicates of f that have all universally quantified variables (lines 6-9) and have a type set contained in the type set of the broker b (condition of line 5). This last condition ensures that the truth of the $f_\forall$ projection can be locally verified by the broker. A $f_\forall$ projection has a $f_\forall\_truth$, a $f_\forall\_children\_number$ and a $f_\forall\_truth\_counter$ variables. The $f_\forall\_truth$ defines the truth of the $f_\forall$ projection. $f_\forall\_children\_number$ is the number of broker children which have a non empty $f_\forall$ formula; these brokers are contained into $\forall\_childrenSet$. The $f_\forall\_truth\_counter$ is the number of children which informed b that their $f_\forall$ projection is true. Each element of the $f_\exists$ array contains a projection that has one predicate with all existentially quantified variables (lines 10-15). Each $f_\exists\_truths[i]$ defines the truth value of the $f_\exists[i]$ projection. The $f_\exists\_truth\_counters[i]$ is the number of children which informed b that their $f_\exists[i]$ projection is true. The projection procedure forwards f to all children brokers that can generate at least a projection from f (lines 26-28).

D. Distributed event matching

A broker b performs the distributed event matching to check the truth of projections. This check reduces the number of pubs that are forwarded towards the root and allows a distributed computation of each global formula f.

1) $f_\forall$ projection checking: A broker b uses the $check\_truth\_\forall$ procedure of Figure 8 for evaluating the truth of all universally quantified predicates inside a $f_\forall$

```
1  Projection project(Formula $f$)
2      Formula $f_\forall$ = empty
3      Formula[] $f_\exists$ = empty[]
4
5      for each $p_i \in f$.matrix, if $T^f_{p_i} \subseteq T_b$
6          if ($p_i$ has all universally quantified variables)
7              for each variable $x_j$ of $p_i$
8                  $f_\forall$.prefix.add($\forall\ x_j \in t^f_{x_j}$)
9              $f_\forall$.matrix = $f_\forall$.matrix $\wedge$ {$p_i$}
10         if ($p_i$ has all existentially quantified variables)
11             Formula $\exists$prj
12             for each variable $x_j$ of $p_i$
13                 $\exists$prj.prefix.add($\exists\ x_j \in t^f_{x_j}$)
14             $\exists$prj.matrix = $p_i$
15             $f_\exists$.add($\exists$prj)
16
17     bool $f_\forall$_truth = false
18     int $f_\forall$_children_number = |$\forall$_childrenSet|
19     int $f_\forall$_truth_counter = 0
20     int[] $f_\exists$_truth_counters = 0[]
21     bool[] $f_\exists$_truths = false[]
22
23     Projection prj = {$f, f_\forall, f_\exists$}
24     projTable.add(prj)
25
26     for each $b_i$ $s.t.$ $par(b_i)$ == $b$
27         if (exist $p_i$ in $f$.matrix s.t. $T^f_{p_i} \subseteq T_{b_i}$)
28             send($f, projection, b_i$)
29
30     return prj
```

Fig. 7. Projection procedure performed by a broker b.

```
1  State check_truth_$\forall$(Formula $f_\forall$)
2      State update = $\emptyset$
3      $f_\forall$_truth = false
4
5      if ($f_\forall$_truth_counter < $f_\forall$_children_number)
6          return $\emptyset$
7
8      for each predicate $p_i$ in $f_\forall$.matrix
9          if ($v^f_{p_i}$ == 1)
10             if (!check$\forall$($p_i$, $S_b$)) return $\emptyset$
11         if ($v^f_{p_i}$ > 1)
12             if (!check$\forall$($p_i$, $S_b \cup S_{tree(b)}$))
13                 return $\emptyset$
14             else
15                 TypeSet $T$ = $T^f_{p_i}$
16                 State $S$ = $S_b$
17                 update = update $\cup$ $filter(T, S)$
18
19     $f_\forall$_truth = true
20     return update
```

Fig. 8. Check $\forall$ part of a projection.

projection. This evaluation is performed on all publication events that can be locally observed at b. This forwards synchronisation information (i.e., publications) only when all $f_\forall$ predicates are true. In fact, local publications that falsify a universally quantified predicate of $f_\forall$, are sufficient to falsify the truth of the global formula f. In this case events are locally blocked. In the following we describe in details the $check\_truth\_\forall$ procedure:

- the $update$ state contains all events that are forwarded when the $f_\forall$ projection of f is true (line 2)
- The $check\_truth\_\forall$ procedure sets $f_\forall$ as $false$ when there is at least one children broker that has locally evaluated its $f_\forall$ as $false$ (lines 3-6).
- A broker b evaluates all $f_\forall$ predicates with variety one (line 9) by considering the state of its directly connected publishers (line 10). When at least one of the $f_\forall$ predicates is false the $f_\forall$ projection is false.
- The truth of $f_\forall$ predicates with variety more than one requires the broker local state and the state of all the children brokers to be considered (line 16). In fact, predicates that contain variables of different types must consider n-tuples that are defined across the state of all the children brokers. When a $f_\forall$ predicate with variety more than 1 is true the state of its variable type is added to the state update variable to be returned. This state will be forwarded to the parent of b.

```
1  State check_truth_$\exists$(Formula[] $f_\exists$)
2      State update = $\emptyset$
3      $f_\exists$_truths = false[]
4
5      for each formula $f_i \in f_\exists$
6          predicate $p_j$ = $f_i$.matrix
7          if $f_\exists$_truth_counters[j]>0
8              $f_\exists$_truths[j]=true
9
10     for each formula $f_i \in f_\exists$
11         predicate $p_j$ = $f_i$.matrix
12         if ($v^f_{p_j}$ == 1)
13             $f_\exists$_truths[i] = $f_\exists$_truths[i] $\vee$ check$\exists$($p_j$, $S_b$)
14
15         if ($v^f_{p_j}$ > 1)
16             bool $\exists$truth = check$\exists$($p_j$, $S_b \cup S_{tree(b)}$)
17             $f_\exists$_truths[i] = $\exists$truth $\vee$ $f_\exists$_truths[i]
18             if ($f_\exists$_truths[i] $\neq$ old$\exists$truth[i])
19                 TypeSet $T$ = $T^f_{p_j}$
20                 State $S$ = $S_b$
21                 update = update $\cup$ $filter(T, S)$
22     return update
```

Fig. 9. Check $\exists$ part of a projection.

2) *$f_\exists$ projection checking:* The $check\_truth\_\exists$ procedure of Figure 9 is quite similar to the $check\_truth\_\forall$ one. Each p_j of an existentially quantified projection inside the array $f_\exists$ is considered. When p_j is true in one of the children broker (lines 5-8) the projection is set to true for the local broker b as well. When p_j has variety one the truth is checked by using the local state of b (line 13) otherwise the state of the children brokers is used as well (line 116). The ith element of the array $f_\exists\_truths$ is true when the ith projection of the array $f_\exists$ is true at least in one of the children broker or in the local broker.

3) *Projection checking:* Figure 10 shows the check procedure that a broker b performs to evaluate all projections of a formula f. These projections are created when they are not in the projection table (lines 3-5). The $check$ procedure first checks the $f_\forall$ projections of f by calling the $check\_truth\_\forall$ procedure that is described in section VII-D1. When the $f_\forall$ projection is evaluated to false the procedure exits and no publication event is forwarded to the parent broker. A particular case is when the $f_\forall$ projection was previously $true$ and becomes $false$. In this case the parent broker of b is informed by using a $\forall$ synchronisation message (line 12). Symmetrically, when the $f_\forall$ projection was $false$ and becomes true the parent broker is informed (line 16). The $f_\exists$ projections are also checked when the $f_\forall$ projection is true (line 19). The $check$ procedure also forwards (if the

```
1  State check(Formula $f$)
2      State update = $\emptyset$
3      Projection prj = projTable.getProjection($f$)
4      if prj = null
5      prj = project($f$)
6
7      bool previous$\forall$truth = $f_\forall$_truth
8      bool[] previous$\exists$truths= $f_\exists$_truths
9
10     State update = check_truth_$\forall$(prj.$f_\forall$)
11
12     if ($f_\forall$_truth == false $\wedge$ previous$\forall$truth == true)
13         send($par(b)$, $\forall$, $f$.id, false)
14         return $\emptyset$
15
16     if ($f_\forall$_truth == true $\wedge$ previous$\forall$truth == false)
17         send($par(b)$, $\forall$, $f$.id, true)
18
19     update = update $\cup$ check_truth_$\exists$(prj)
20     for each $p_i$ in $f_\exists$
21         if(previous$\exists$truths[i] != $\exists$truths[i])
22             send($par(b)$, $\exists$, $f$.id, $p_i$.id, $f_\exists$_truths[i])
23
24     for each predicate $p$ in $f$
25         if ($p$ has at least a universally and an
               existentially quantified variable)
26             TypeSet $T$ = $T_p^f$
27             State $S$ = $S_b$
28             update = update $\cup$ $filter(T, S)$
29
30     return update
```

Fig. 10. Checking procedure of formulae performed by a broker b.

```
1  void parse(Message msg)
2      State update=$\emptyset$
3
4      if (msg == receive($b_i$, $formula$, $f$))
5          update = update $\cup$ check($f$)
6
7      if(msg == receive($b_i$, $state$, update))
8          $S_{tree(b)}$ = $S_{tree(b)}$ $\cup$ update
9          for each $f$ $\in$ subTable $s.t.$ $T_f \cap T_{update}$ != $\emptyset$
10             update = update $\cup$ check($f$)
11
12     if(msg == receive($b_i$, $event$, e))
13         $S_b$ = $S_b$ $\cup$ {e}
14         for each $f$ $\in$ subTable $s.t.$ $T_e \in T_f$
15             update = update $\cup$ check($f$)
16
17     if (msg == receive($b_i$, $\forall$, $f$.id, false))
18         $f_\forall$_truth_counter --
19         if ($f_\forall$_truth_counter == $f_\forall$_truth_number - 1)
20             if ($f_\forall$_truth)
21                 $f_\forall$_truth = false
22                 send($par(b)$, $\forall$, $f$.id, false)
23
24     if (msg == receive($b_i$, $\forall$, $f$.id, true))
25         $f_\forall$_truth_counter ++
26         if ($f_\forall$_children_number == $f_\forall$_truth_counter)
27             update = update $\cup$ check($f$)
28
29     if (msg == receive($b_i$, $\exists$, $f$.id, $p_i$.id, truth))
30         if (truth) $f_\exists$_truth_counter[i]++
31         else $f_\exists$_truth_counter[i]--
32         update = update $\cup$ check($f$)
33
34     if (update != $\emptyset$) send($par(b)$,$state$,update)
35
36     return
```

Fig. 11. Parsing of messages at broker b.

$f_\forall$ projection is true) all publications events that are needed by the parent broker to check predicates that have at least a universally and an existentially quantified variable (line 25). These predicates are not checked at any broker but their truth is globally checked by the root by using the state forwarded by all brokers.

4) Parsing of messages: Figure 11 outlines how a broker b parses messages arriving from other brokers, publishers and subscribers. This procedures always sends any publication events produced (if any) at the end (line 34). When a new formula f is received the broker uses the check procedure to update its projection table (line 5) and checks the truth value of f. The local subtree state of a broker can be synchronised upon the reception of a $state$ synchronisation message (lines 7-10). This requires the checking of some formulae to be performed. The broker b can also receive updates on the $f_\forall$ projections from its children brokers; in this case it updates the $f_\forall\_truth\_counter$ (we recall this stores the number of children brokers which informed b that their $f_\forall$ projection is true). The parent broker is notified with a $\forall$ message $false$ (lines 19-22) when the counter gets lower than $f_\forall\_children\_number$. In fact, the parent needs to be updated when the $f_\forall$ projection is not true in any of the children brokers that are evaluating it. When an $\exists$ truth is received (line 29) the $f_\exists\_truth\_counters$ array is updated and the check procedure is called.

E. Notification delivery

A root is a particular type of broker that has access to the entire state of the system. The root receives synchronisation messages from its children brokers and can calculate the truth of every global formula f. This is done by procedures that are similar to the broker ones. Whenever the root detects a change in the truth of a subscription formula f, it notifies its subscriber of it. To do this, each broker (starting with the root itself) uses its `subTable` to trace back the subscriber which submitted the subscription. In this way, a `notification(f)` message is sent to the subscriber of f and all equivalent formulae. At this point, subscribers can react to notifications with a customised callback procedure, which gives the programmer the power to define any possible behaviour.

We emphasise that that predicates with variety one are always computed in a distributed manner, i.e., the root wait for the truth of their children brokers and evaluate the truth in its local state. Predicates with variety more than one requires a state synchronisation from the children brokers in order to consider tuples which contain events that are observed by different brokers.

VIII. PICO-MP PERFORMANCE

In this section we discuss the energy efficiency of the PICO-MP middleware. To this ending we study the traffic generated inside the distributed notification service. We compare PICO-MP with an MQTT-like centralised implementation. The following assumptions are made: (i) WSAN devices and brokers are arranged in a tree like structure (see Figure 5); (ii) the centralised implementation can only perform the event matching at the root of the tree (devices are not enough performant to run the broker); (iii) PICO-MP brokers run in all WSANs devices. In our study we do not consider subscription and notification messages since they flow in the same way in both centralised and PICO-MP implementation.

It is worth mentioning that PICO-MP can optimise the traffic generated by subscriptions and notifications. More precisely, a subscription does not reach the root when an equivalent one was already sent. Notification traffic can be reduced when subscribers only require changes in truth value of a formula and not the entire state.

The total amount of messages that are sent/received by a broker b (at the time t) can be summarised as follows:

$$M_b(t) = M_{pub}(t) + M_{reg}(t) + M_{int}(t)$$

where $M_{pub}(t)$ and $M_{reg}(t)$ is the total amount of $publication$ and $registration$ messages sent/received and $M_{int}(t)$ all additional control messages that are sent/received by the broker.

The centralised implementation has no registration messages and no additional control messages thus the total amount of messages can be summarised as $M_b^{centralised}(t) = M_{pub}(t)$.

The PICO-MP total amount of messages can be calculated as $M_b^{PICO}(t) = M_1(t) + M_2(F,t) + M_{pub}(F,t)$ where $M_1(t)$ are control messages that do not depend on the truth value of any formula (i.e., projections, registrations and unregistrations of publishers, advertisements and unadvertisements of brokers), $M_2(F,t)$ are messages that are generated when one or more formulae change their truth value (i.e, $\forall$ and $\exists$ control messages). The $M_{pub}(F,t)$ term identifies the publications that have been sent/received at time t (i.e.,the $state$ control messages). These are sent when the subscribers require the state that made the formulae truth changing.

We want to study the equation $M_b^{PICO}(t) \leq M_b^{centralised}(t)$. This can be written as:

$$M_1(t) + M_2(F,t) + M_{pub}(F,t) \leq M_{pub}(t)$$

This equation states that PICO-MP performs worse than a centralised implementation when the formulae are always true and the subscribers always require the entire state to be sent. In this case PICO-MP adds to the publication messages ($M_{pub}(F,t) = M_{pub}(t)$) the extra messages $M_1(t)$ and $M_2(F,t)$ for projections/registrations and formula truth notifications (i.e.,$\forall$ and $\exists$). PICO-MP performs much better when subscribers do not require the state (the term $M_{pub}(F,t)$ is zero), the formulae do not change their truth often (i.e., $M_2(F,t)$ is low) and subscriptions are not frequent (i.e., $M_1(t)$ is low due to reduced control messages). For instance this is the case of home and building automation where subscriptions are usually related to actuators. Actuators are not added/removed frequently and an actuator policy is rarely changed.

IX. Conclusions

In this paper we present PICO-MP, a fully decentralised macro-programming middleware for WSANs. An expressive language that is based on first order logic is used to specify a global behaviour. This can correlate distributed sensing and actuating events. A broker can locally create a projection of a global formula. PICO-MP ensures that the distributed checking of all projections of all brokers produces the same results of their global formula checking. The use of PICO-MP can improve the WSAN life time when publishers plays a very dynamic role with a lot of activity, while subscribers do not change their subscriptions very often.

References

[1] J. Cecílio and P. Furtado. A state-machine model for reliability eliciting over wireless sensor and actuator networks. In *Proceedings of the 3rd International Conference on Ambient Systems, Networks and Technologies (ANT 2012), the 9th International Conference on Mobile Web Information Systems (MobiWIS-2012), Niagara Falls, Ontario, Canada, August 27-29, 2012*, pages 422–431, 2012.

[2] J. Chen, M. Díaz, B. Rubio, and J. M. Troya. PS-QUASAR: A publish/subscribe qos aware middleware for wireless sensor and actor networks. *Journal of Systems and Software*, 86(6):1650–1662, 2013.

[3] P. T. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[4] K. Gill, S.-H. Yang, F. Yao, and X. Lu. A zigbee-based home automation system. *Consumer Electronics, IEEE Transactions on*, 55(2):422 –430, may 2009.

[5] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using *Kairos*. In *Distributed Computing in Sensor Systems, First IEEE International Conference, DCOSS 2005, Marina del Rey, CA, USA, June 30 - July 1, 2005, Proceedings*, pages 126–140, 2005.

[6] J. hinrich Hauer, H. Vlado, A. Köpke, A. Willig, and A. Wolisz. A component framework for content-based publish/subscribe in sensor networks. February 2008.

[7] S. Ji, C. Ye, J. Wei, and H.-A. Jacobsen. Merc: Match at edge and route intra–cluster for content-based publish/subscribe systems. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 13–24, New York, NY, USA, 2015. ACM.

[8] M. Karpinski and V. Cahill. Stream-based macro-programming of wireless sensor, actuator network applications with SOSNA. In *Proceedings of the 5th Workshop on Data Management for Sensor Networks, in conjunction with VLDB, DMSN 2008, Auckland, New Zealand, August 24, 2008*, pages 49–55, 2008.

[9] G. Li and H. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Middleware 2005, ACM/IFIP/USENIX, 6th International Middleware Conference, Grenoble, France, November 28 - December 2, 2005, Proceedings*, pages 249–269, 2005.

[10] L. Mostarda and A. Navarra. Distributed intrusion detection systems for enhancing security in mobile wireless sensor networks. *IJDSN*, 4(2):83–109, 2008.

[11] G. Russello, L. Mostarda, and N. Dulay. ESCAPE: A component-based policy framework for sense and react applications. In *Component-Based Software Engineering, 11th International Symposium, CBSE 2008, Karlsruhe, Germany, October 14-17, 2008. Proceedings*, pages 212–229, 2008.

[12] A. Stanford-Clark and H. L. Truong. Mqtt for sensor networks (mqtts) specifications. October 2007.

[13] C. Vannucchi, M. Diamanti, G. Mazzante, D. R. Cacciagrano, F. Corradini, R. Culmone, N. Gorogiannis, L. Mostarda, and F. Raimondi. virony: A tool for analysis and verification of ECA rules in intelligent environments. In *2017 International Conference on Intelligent Environments, IE 2017, Seoul, Korea (South), August 21-25, 2017*, pages 92–99, 2017.