Garrido-Castellano and Murillo-Fuentes *EURASIP Journal on Wireless Communications and Networking* [illegible]
DOI 10.1186/s13638-015-0382-6

EURASIP Journal on
Wireless Communications and Networking
a SpringerOpen Journal

RESEARCH **Open Access**

On the implementation of distributed asynchronous non-linear kernel methods over wireless sensor networks

Juan A. Garrido-Castellano* and Juan J. Murillo-Fuentes

Abstract

In this paper, we face the implementation of a non-linear kernel method for regression on a wireless sensor network (WSN) based on MICAz motes. The operating system used is TinyOS 2.1.1. The algorithm estimates the value of some magnitude from the measurements of the motes in a distributed approach where information and computations are performed asynchronously. This proposal includes a research on the potential problems encountered along with the developed solutions. Namely, matrix and floating computations, acknowledgement mechanisms and data loss.

Keywords: Wireless sensor network; Distributed; Linear; Kernel method; Asynchronous; Algorithm; MICAz; Mote; TinyOS

1 Introduction

Wireless sensor networks (WSNs) are very useful to monitor physical conditions in large or difficult access environments. Due to their wireless capabilities and their use of batteries, these networks have low costs of deployment. The nodes, usually known as *motes*, may include sensors of many kinds. Since they are also endowed with processing capabilities, they can perform a set of algorithms accordingly to the result of the measurements. These features open a wide spectrum of applications for WSNs [1]. They can be seen as intelligent and autonomous networks.

A WSN is often understood as a network with a central node that runs the main operations such as network synchronization, data processing and storage, while the rest of nodes would just take measures to later send them to the central node. We may find several real implementations of centralized WSNs, e.g. in agriculture [2] or tracking [3]. However, it is well known that this topology has several problems in large networks due to its dependency on the central node. To name a few:

- Unbalanced energy consumption: since computations and communications are concentrated in one and a few nodes, respectively.
- Inefficient use of network bandwidth: large number of relays and nodes around central node with high rates associated, also wasting energy.
- Unreliability: if the central node or nodes around it get down for any reason, the network is unavailable.
- Poor response time: in large networks, we have large latencies associated to relays and management of huge amounts of data.

These problems can be solved by using a distributed network model (see Fig. 1), where all nodes compute the solution for the locations in its neighbourhood by interchanging information locally.

There are simple methods to make distributed estimations such as least squares regression. Nevertheless, attending to the application, more improved algorithms may be needed. For example, in structural health monitoring (SHM), some approaches in WSN need to manage non-linear cases in a distributed architecture [4]. And support vector machine-based non-linear distributed approaches can be applied to solve localization problems from RSSI parameters [5]. In this sense, some general frameworks for distributed kernel approaches have been presented in [6, 7] to solve non-linear regression in a distributed and real-time way. The design of distributed intelligent WSN involves both the design of distributed algorithms (like [4–8]) and their implementation. While the former is platform independent, the latter sets out

*Correspondence: jgarrido2@us.es
Department of Signal and Communications Theory, Universidad de Sevilla, Camino Descubrimientos, s/n - Isla Cartuja, Seville, Spain

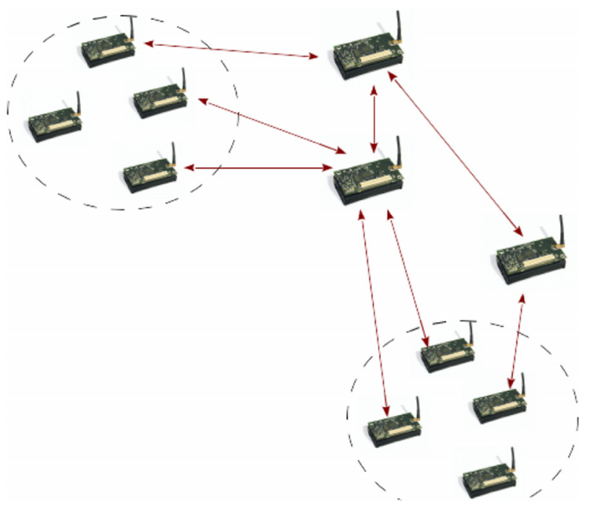

Fig. 1 Example of distributed wireless sensor network

some interesting problems when translated to a WSN architecture.

Some works on distributed implementations have already been proposed [9, 10]. However, to our knowledge, there is no implementation description of complex non-linear kernel-based algorithms reported in the literature, like the ones in [6, 7]. Due to the important benefits of these algorithms, we have selected them as a target to study as a real implementation over a network composed by simple motes like MICAz, using the temperature field to easily illustrate the performance of the algorithm. These algorithms are quite demanding on both communication and processing capabilities. So, the main question to face is about its feasibility when implemented on an standard WSN with low resources motes. We positively answer this question by proposing a solution for this problem.

The algorithm in [7] is based on the kernel least squares (KLS) algorithm. Its main advantage is that although it is computed in a distributed way, it converges to the solution of the centralized version. As discussed, the distributed KLS (DKLS) is highly demanding in terms of communication and computation capabilities. It requires matrix and kernel operations at the same time than multi-node communication management. Our solution is possible thanks to the versatile features of TinyOS, the operating system that will be used to program and compile the application into the motes.

Besides the computational complexity of the algorithm, we have found severe problems with data packet loss, management of the transceiver buffers and unsupported hardware floating point data computation. The two first problems have been solved developing a new layer for communication handling that works as an extension of the related native libraries of TinyOS, in which not only data buffering features have been extended but also some changes into these native libraries have been done in order to improve the performance. On the other hand, regarding the floating point data management, TinyOS provides a software emulation with severe limitations when using with matrix data. In this way, a two-step solution has been adopted to allow these operations.

This paper is organized as follows. In Section 2, we introduce the regression algorithm to be implemented, starting with the classic KLS regression and ending with the DKLS algorithm. Next, the modified DKLS (m-DKLS) algorithm, an evolution of DKLS, is explained in Section 3. Its implementation is our target in this paper. In Section 4, we provide an overview on the main concepts of this operating system, in order to better understand the limitations later found, which are detailed in Section 5. These limitations are the starting point of our work. The ideas to solve these problems are presented in Section 6. The next step is detailed in Section 7, where we develop the implementation in a MICAz network. In Section 8, the testing scenario is defined, and some obtained results are included. Finally, in Section 9, we summarize the work done, establishing in Section 9.1 some points of improvement and pending work to be done in this line of research.

2 Distributed kernel least squares

2.1 Classic kernel least squares regression

As introduced in [11], the classic *kernel least squares* regression method is a well-known approach based on applying the kernel trick to the linear least squares algorithm. Given $\mathbf{x}_i \in \mathbb{R}^d$, a set of training inputs, and $y_i \in \mathbb{R}$, the corresponding outputs, with $i = 1, ..., n$, the algorithm finds the hyperplane in the non-linearly transformed kernel space, $f(\mathbf{x})$, that better fits a given set of training data, minimizing a least squares criteria. Then, for any new or test input, $\mathbf{x}_\star$, the method predicts the output as $f(\mathbf{x}_\star)$. In a WSN, the input $\mathbf{x}_i$ could be the position coordinates while the output y_i could be some environment measure, e.g. the temperature.

In the computation of the prediction, $f(\mathbf{x})$, an optimization problem is solved where a loss function, $\mathcal{L}$, between the truth y_i and its prediction $f(\mathbf{x}_i)$

$$\mathcal{L}(y_i, f(\mathbf{x}_i)) \geq 0, \tag{1}$$

is averaged over the joint probability density function of the input and outputs,

$$\mathcal{R}(f) = \int_{\mathcal{Y}\times\mathcal{X}} \mathcal{L}(y, f(\mathbf{x}))p(y, \mathbf{x})dyd\mathbf{x} \tag{2}$$

where $\mathcal{R}(f)$ is the so-called risk function.

Usually, the risk of making estimations cannot be computed because the joint distribution between the inputs and the outputs is unknown. However, we can compute an approximation averaging the error function of the available data, as it is formulated by the *empirical risk minimization* (ERM) principle. The formulation in (2) yields

$$\mathcal{R}_{\text{emp}}(f) = \sum_{i=1}^{n} \mathcal{L}(y_i, f(\mathbf{x}_i)). \tag{3}$$

Using the least squares (LS) loss function, we get:

$$\mathcal{R}_{\text{emp}}(f) = \sum_{i=1}^{n} (y_i - f(\mathbf{x}_i))^2. \tag{4}$$

Since there is an infinite set of non-linear prediction functions, $f(\mathbf{x})$, that fit the output data, we need to constrain the solution. This is achieved through Thikonov regularization. We get the classic formulation (5) of the KLS problem,

$$f_\lambda(\cdot) = \arg\min_{f \in \mathcal{H}_K} \frac{1}{n} \sum_{i=1}^{n} (f(\mathbf{x}_i) - y_i)^2 + \lambda \|f\|^2_{\mathcal{H}_K}. \tag{5}$$

The optimization variable is f, which is a function constrained to be in $\mathcal{H}_K$, the *reproducing kernel Hilbert space* induced by the kernel $k(\cdot,\cdot)$, denoting by $\|\cdot\|_{\mathcal{H}_K}$ its norm. $\mathcal{H}_K$ is a vector space of functions with a certain (and convenient) inner product. Note that in (5), we compute $f(\cdot)$ to minimize the mean square error with the first term, while with the last one we force the solution $f(\cdot)$ to have minimum norm to avoid overfitting. The inner-product structure implies that the solution to (5), denoted by $f_\lambda(\cdot)$, satisfies:

$$f_\lambda(\cdot) = \sum_{i=1}^{n} c_{\lambda,i} k(\cdot, \mathbf{x}_i) \tag{6}$$

for some $\mathbf{c}_\lambda \in \mathbb{R}^r$. This fact is known as the *representer theorem* in [12]. In the case of least squares, $\mathbf{c}_\lambda$ is the solution to a system of n linear equations, satisfying:

$$\mathbf{c}_\lambda = (\mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{y} \tag{7}$$

where $\mathbf{K}$ is the kernel matrix whose elements are defined by $k_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$, and the kernel is pre-specified.

2.2 Distributed kernel least squares (DKLS)

2.2.1 Distributed definition of KLS

The previous solution is a centralized algorithm and cannot be implemented in a distributed approach as it is. Let us suppose that we have a wireless sensor network of m nodes and we have $n \leq m$ measurements from them as training samples. Using the same notation as in Section 2.1, we could think of position as inputs $\mathbf{x}_i \in \mathbb{R}^3$, and temperature measures y_i as outputs. The training samples are ensembles in the set S_n. Let us suppose that not all nodes have access to all the samples, so the training samples accessible from node j is the subset S_n^j. Let us also denote the set of the indices of the training samples in S_n by $\overline{S}_n$ and the indices of training samples accessible by node j as $\overline{S}_n^j$.

The first approximation to a distributed problem in this scenario is to compute m centralized solutions, one for each node of the network, so the classical KLS problem could be written as:

$$\min_{f_j \in \mathcal{H}_K} \sum_{i=1}^{n} (z_i - y_i)^2 + \sum_{j=1}^{m} \lambda_j \|f_j\|^2_{\mathcal{H}_K} \tag{8}$$

$$s.t.\ z_i = f_j(x_i),\ \forall i \in \overline{S}_n,\ j = 1, ..., m. \tag{9}$$

In this problem, the optimization variables are $\mathbf{z} \in \mathbb{R}^n$, i.e. $\{f_j\}_{j=1}^m$, and rather than finding a function $f(\cdot)$, we are estimating a set of them.

The constraints in (9) require that all nodes agree on the training data. This fact makes it to be equivalent to the classic KLS problem, getting the centralized solution, i.e. $f_j(\cdot) = f_\lambda(\cdot)$ for $j = 1, ..., m$ (see *Lemma 1* in Appendix of [6]). So, we can associate a centralized regression to a global agreement of nodes on the training samples. But we could think of an association of a distributed regression to a local agreement instead. Local agreement would involve that only a limited number of samples are shared between each two nodes. This last problem can be described as follows,

$$\min_{f_j \in \mathcal{H}_K} \sum_{i \in \overline{S}_n^j} (z_i - y_i)^2 + \sum_{j=1}^{m} \lambda_j \|f_j\|^2_{\mathcal{H}_K} \tag{10}$$

$$s.t.\ z_i = f_j(x_i),\ \forall i \in \overline{S}_n^j,\ j = 1, ..., m. \tag{11}$$

In this formulation, the solution is feasible if and only if $f_j(x_i) = z_i = f_k(x_i)$ for $(x_i, y_i) \in S_n^j \cap S_n^k$ and for $j,k = 1, ..., m$; that is, if and only if every pair of node decision rules agree on samples they share. We get $(\mathbf{z}, f_1, ..., f_m)$ as the minimizer solution of (10), and f_j is a function of only the training samples in S_n^j as part of the joint minimizer.

2.2.2 Successive orthogonal projections algorithm

A distributed approach of KLS problem has been shown in the previous subsection. Here, we face its solution, for which an alternate projections algorithm is proposed in [6], taking into account the similarities between both problems. In particular, the algorithm uses the non-relaxed successive orthogonal projection (SOP) algorithm, next described.

Let $C_1, ..., C_m$ be closed convex subsets of the Hilbert space $\mathcal{H}$, whose intersection $C = \cap_{i=1}^m C_i$ is non-empty. Let $P_C(\hat{v})$ denote the orthogonal projection of $\hat{v} \in \mathcal{H}$ onto C:

$$P_C(\hat{v}) \triangleq \arg\min_{v \in C} \| v - \hat{v} \| \tag{12}$$

And the orthogonal projection of $\hat{v} \in \mathcal{H}$ onto C_i:

$$P_{C_i}(\hat{x}) \triangleq \arg\min_{v \in C_i} \| v - \hat{v} \| \tag{13}$$

In [6, 13], it is defined the successive orthogonal projection (SOP) algorithm to compute $P_C(\cdot)$ using $\left\{P_{C_i}(\cdot)\right\}_{i=1}^{m}$ as follows:

$$v_0 := \hat{v} \qquad v_t := P_{C_{(t \bmod m)+1}}(v_{t-1}) \tag{14}$$

In this definition (14), we denote by $(t\ mod\ m)$ to the remainder of the division t/m. It establishes that the $P_C(\cdot)$ can be computed projecting sequentially onto all the convex subsets C_i, using for the $P_{C_{i+1}}$ the result of the previous projection: first, it projects $\hat{v}$ onto C_1, the result P_{C_1} is projected onto C_2, and it iterates in this way successively a certain number of times.

As pointed out in [6] (*Theorem 2*), it is demonstrated in [14] that for every $v \in C$ and every $t \geq 1$

$$\| v_t - v \| \leq \| v_{t-1} - v \| \tag{15}$$

and that

$$\lim_{n\to\infty} v_n \in (\cap_{i=1}^{m} C_i) \tag{16}$$

$$\lim_{n\to\infty} \| v_t - P_C(\hat{v}) \| = 0 \tag{17}$$

if C_i are affine for all $i \in \{1, ..., m\}$. Hence, the more iterations we perform, the more accurate result we get.

2.2.3 Distributed KLS solution

It is possible to redefine the problem in (10) in terms of the SOP algorithm [6], where the Hilbert space $\mathcal{H} = \mathbb{R}^n \times \mathcal{H}_K^m$ with norm

$$\| (\mathbf{z}, f_1, ..., f_m) \|^2 = \| z \|_2^2 + \sum_{i=1}^{m} \lambda_i \| f_i \|_{\mathcal{H}_K}^2 \tag{18}$$

is defined. With it, (10) can be interpreted as the orthogonal projection of the vector $(\mathbf{y}, 0, ..., 0) \in \mathcal{H}$ onto the set $C = \cap_{j=1}^{m} C_j \subset \mathcal{H}$, with

$$C_j = \Big\{ (\mathbf{z}, f_1, ..., f_m) : f_j(x_i) = z_i, \forall i \in \overline{S}_n^j, \\ \mathbf{z} \in \mathbb{R}^n, \{f_j\}_{j=1}^{m} \subset \mathcal{H}_K \Big\} \subset \mathcal{H} \tag{19}$$

It is important to note that, for any $v = (\mathbf{z}, f_1, ..., f_m) \in \mathcal{H}$, the computation of

$$P_{C_j}(v) = \arg\min_{v' \in C_j} \| v - v' \| \tag{20}$$

is restricted to the locally accessible training examples by node j. It means that computing $P_{C_j}(v)$ leaves z_i unchanged for all $i \notin \overline{S}_n^j$ and leaves f_k unchanged for all $k \neq j$.

The new function associated with the node j can be computed using f_j, $\{x_i\}_{i\in\overline{S}_n^j}$ and the message variables $\{z_i\}_{i\in\overline{S}_n^j}$. This method defines the DKLS algorithm (using the notation of [7]), shown in Algorithm 1.

Algorithm 1 DKLS algorithm

```
Initialization

Each node $j$ broadcasts their location $\mathbf{x}_j$ to its neighbouring sensors.
Each node $j$ broadcasts their measurement $y_j$ to their neighbouring sensors.
Each node $j$ initializes $z_k = y_k, \forall k \in \overline{S}_n^j$.
Each node $j$ initializes $f_{j,0} = 0$.

Training

for t=1,...,T do
  for $j = 1, ..., m$ do
    $f_{j,t} = \underset{f\in\mathcal{H}_K}{\text{argmin}} \sum_{i\in\overline{S}_n^j} (f(\mathbf{x_i}) - z_i)^2 + \lambda_j \|f - f_{j,t-1}\|_{\mathcal{H}_K}^2$
    Node $j$ broadcasts $f_{j,t}(\mathbf{x}_k) \forall k \in \overline{S}_n^j$
    Every node sharing data $i$ replaces $z_k$ by $f_{j,t}(\mathbf{x}_k), \forall k \in \overline{S}_n^i$
  end for
end for
```

It is interesting to note that the solution in (10) is an approximation to the centralized KLS. As discussed in [6], the neighbourhood of a mote limits the accuracy of its estimations, so local connectivity influences an estimator's *bias*. In [6, 7], there are some studies that show through simulations that the error decays exponentially with the number of neighbours.

3 Non-linear asynchronous distributed algorithm

The DKLS algorithm in [6] is beneficial in many ways, but in [7], the authors highlight several limitations:

- After each node completes its training stage, it must generate a prediction for each neighbour (a different message computation for each one).
- Due to that, the communication burden grows with the number of neighbours, and it means that the communication and computation load can be high.
- Each node broadcasts one estimation for each neighbour node, and all of these messages should be received by all of its neighbours. Let us denote n_0 as the sender mote and n_1, n_2 as neighbours of n_0. Node n_0 would broadcast both $f_{j,t}(\mathbf{x}_{n_1})$ and $f_{j,t}(\mathbf{x}_{n_2})$ and both would be received by n_1 and n_2. If n_1 and n_2 are not neighbours between them, they must discard the non-corresponding message, but the packet already have been received and read. This means waste of resources, in terms of processing and energy consumption.
- It needs a synchronization of the network to do the training step, because we can only train one node at a

time. Otherwise, a node could receive several updates for z_k, and then it would not know which one should keep.
- Finally, if a sensor stops working (or some radio link fails) then the learning procedure stops, and the next node does not receive new predictions so it does not start its training stage.

To overcome these limitations, in [7], the authors propose a simple modification in the algorithm: instead of transmitting $f_{j,t}(\mathbf{x}_k)$ *for* $k \in \overline{S}_n^j$, the node would only broadcast $f_{j,t}(\mathbf{x}_j)$ (the prediction for the current node) to all its neighbours. The resulting algorithm is the modified DKLS algorithm (m-DKLS, see Algorithm 2).

Algorithm 2 Modified DKLS algorithm (m-DKLS)

```
Initialization

Each node $j$ broadcasts their location $\mathbf{x}_j$ to its neighbouring sensors.
Each node $j$ broadcasts their measurement $y_j$ to their neighbouring sensors.
Each node $j$ initializes $z_k = y_k, \forall k \in \overline{S}_n^j$.
Each node $j$ initializes $f_{j,0} = 0$.

Training

for t=1,...,T do
    for $j = 1, ..., m$ do
        $f_{j,t} = \underset{f \in \mathcal{H}_K}{\operatorname{argmin}} \sum_{i \in \overline{S}_n^j} \left(f(\mathbf{x}_i) - z_i\right)^2 + \lambda_j \|f - f_{j,t-1}\|^2_{\mathcal{H}_K}$
        Node $j$ broadcasts $f_{j,t}(\mathbf{x}_j)$
        Each neighbouring node replaces $z_j$ by $f_{j,t}(\mathbf{x}_j)$
    end for
end for
```

The benefits of the algorithm can be summarized as follows:

- It reduces the number of messages generated from N_j (the number of neighbours of node j) to 1. It only needs to broadcast one message.
- Since it broadcasts only one data, there is no need to synchronize the network. Each node decides when to transmit.
- If a connection between two nodes stops working, it does not stop the training stage of the other nodes of the network.

Although it exhibits a slightly worse convergence than the DKLS approach, again the error reduces exponentially with the number of neighbours. The number of retransmissions reduces significantly while the number of iterations increases to achieve a given error level [7]. In addition, the retransmissions can be performed asynchronously.

At this point it is interesting to note that the m-DKLS algorithm can be easily extended to other kernel methods different to the KLS.

4 Brief introduction to TinyOS

4.1 NesC overview

TinyOS is an open source operating system designed to work specifically with wireless sensor networks. It has an event-oriented architecture, and it has a set of libraries that provides data acquisition tools for the motes. These libraries are open source, so the code is available to be modified. It uses the *NesC* programming language, which is a dialect of the C language that adds some additional event-oriented features.

Every *NesC* application is based on modular programming. An application is composed by one or more *components*. Each component can be seen as a functional part of the application. Each component has *interfaces*. These interfaces are used to connect this component to other components of the application. Interfaces use two kind of operations:

- *Commands* (input): to allow external components to trigger operations to the component.
- *Events* (output): to send notifications to other components.

4.2 Tasks

Tasks are a very important feature of TinyOS. They are code blocks that are executed only when the processor is available:

- When the processor is running the operations of a *task*, it can not be stopped to execute any other *task*. For this reason, it's desirable not to include too many operations in one *task* in order to share the processor by all the modules of the application. All *tasks* have same execution priority.
- *Tasks* do not work as functions do; they do not admit parameters.

4.3 Event and commands types

In TinyOS, it is quite important to introduce the different ways to use events and commands (see [15], Section 4.5).

Two kind of events and commands are available in TinyOS:

- Synchronous operations have the same execution priority, and they are related to *tasks*.
- Asynchronous operations have higher priority: they interrupt the current execution (even *tasks*), and they are related to *interrupts*.

In the following sections, we explain how important all these features are, in order to implement the algorithm.

5 Platform features and limitations

5.1 Hardware features: MICAz

The selected hardware for this work are the MICAz motes. They have limited resources compared to other devices such as Imote2. Hence, a successful implementation on this platform ensures the compatibility with other more powerful devices. They have the following features:

- ATMEGA128L 8-bit micro-controller [16].
- Temperature and humidity sensors.
- ChipCon CC2420 IEEE 802.15.4 compliant RF transceiver. It works with the standard IEEE 802.15.4 at 2.4 GHz. Maximum data rate of 250 kbps [17]. It has 128-byte transmission and reception-independent FIFOs.
- Internal flash 128 kB, RAM 4 kB, external flash 512 kB.

5.2 Limitations

5.2.1 Floating point data limitations

The m-DKLS algorithm is based on kernel least squares learning, and it requires to perform floating point operation capabilities.

We need to develop a linear algebra library to implement the algorithm. This library is compound by several functions: matrix inversion, matrix product, scalar product etc. Each function executes a certain matrix operation and needs to receive the arguments to be used (matrices). The only way to pass matrices to functions in NesC is passing arguments by reference (pointers).

MICAz motes do not support floating point operations by hardware, and they are needed by the algorithm, which uses matrices of floating point data. Floating point variables are, then, managed by software emulation when the application is compiled, but passing floating point parameters by reference to functions are not supported; we need this feature to compute matrix operations.

5.2.2 Reception buffer limitations

It is necessary to define a frame structure to send and receive data in the network. We have used the frame structure in Fig. 2. Note that not all the fields are required for an implementation but are useful for monitoring purposes:

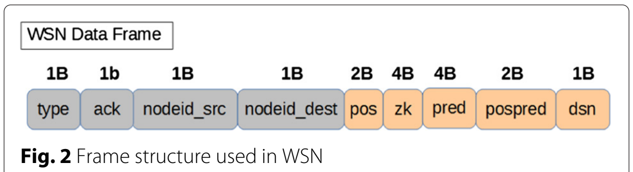

Fig. 2 Frame structure used in WSN

- *type* (1 byte): this field is used to classify the frame by the step of the algorithm which it belongs to.
- *ack* (1 bit): Boolean field to indicate if the message is an ACK.
- *nodeid_src* (1 byte): identification of the sender node.
- *nodeid_dest* (1 byte): identification of the destination node.
- *pos* (2 bytes): position of the *nodeid_src* node.
- *zk* (4 bytes): its value depends on the step of the algorithm which the message belongs to. If the message is an *initialization step* message, then this field contains the temperature measured by *nodeid_src*. If the message is a *training step* message, then it contains the $f_{j,t}(\mathbf{x}_j)$ of m-DKLS.
- *pred* (4 bytes): this field only takes value when the *training step* has been completed and a result is sent. The value sent is corresponding to the prediction of temperature in the position indicated by the field *pospred.*
- *pospred* (2 bytes): position in which the latest prediction has been done, and which temperature value is in the field *pred.*
- *dsn* (1 byte): field used for acknowledgement control.

In the m-DKLS algorithm, each node must compute an estimate of $f_{j,t}$ after receiving a new data from its neighbours. Let us suppose the following buffer-overflow scenario:

- The number of neighbours of a mote is greater than the number of frames that the receiving buffer of the CC2420 chip can hold.
- All the neighbours of this mote send a new data at the same time while the mote is computing the current data, or the incoming messages in the mote arrive too quickly to process them in real time.
- The default mode of events and commands of TinyOS is used to manage the events of all modules, i.e. synchronous mode.

In such a situation, we have the problem described in Fig. 3. This figure represents the mote by its two main parts, namely, the micro-controller and the CC2420 transceiver, both with a vertical computation timeline. We have a three-message length reception buffer and a neighbourhood of five nodes. Due to the default synchronous event mode, all the events have the same priority of execution, including the event that signals the reception of a new packet into the buffer. This means that the microprocessor does not attend that event until the current task execution ends. In this scenario, it is easy to lose incoming packets due to buffer overflow.

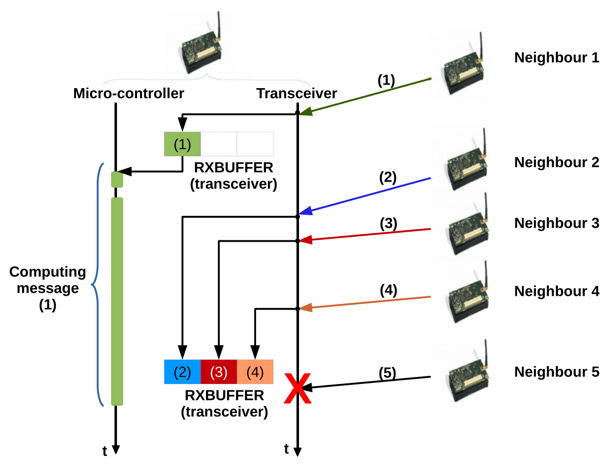

Fig. 3 Overflow in reception buffer of transceiver. In this example, the number of neighbours is greater than the number of packets that the transceiver buffer can hold. The last packet would be lost if no retransmissions are done

5.2.3 *Transmission buffer limitations*

We could have similar problems when transmitting data to the network. The native libraries to handle the transmission of data only use one RAM memory variable to hold the packet until it is accepted by the transceiver and saved into its transmission buffer. If the processor requests to transmit several packets (for example, the calculated z_j to all the neighbours) while the hardware transmission buffer is full, only the last of them will be transmitted due to variable overwriting. Therefore, it is needed an additional control by software to transmit messages.

5.2.4 *Acknowledgement mechanism limitations*

CC2420 chips provide an acknowledgement mechanism (a hardware mechanism). This can protect the radio link from undesired problems in the radio channel. But as it is referenced in [18] (and we have observed during our work), some additional issues are detected, like *false acknowledgements*.

A false acknowledgement occurs when the radio chip receives a packet and it acknowledges its reception, but the micro-controller would never receive it. For this reason, it is advisable to use a software acknowledgement mechanism at the application level with certain special behaviour. This idea will be detailed as part of the proposed solution (in Sections 6 and 7).

5.2.5 *The problem of the initialization step with communication errors*

The initialization step of the m-DKLS algorithm is critical. If a node starts the training step without the sample $(\mathbf{x}_i, y_i)$ of one of its neighbours, the algorithm will get quite wrong results, as it will use a 0 value for the initial measure of that neighbour. So the way by which the nodes acquire the knowledge of the neighbourhood is of high importance:

- We need a very reliable transmission and reception of data if the network topology is fixed.
- We must protect each node not to start its training step until all the neighbour measures have been received.

When using a fixed topology, each mote knows its neighbourhood because it is defined in the application source code. This means that matrix dimensions are also fixed, so all the initialization messages from the neighbours must be received in order to start the training step.

6 Proposed solution

In the last section, we detected two main problems when implementing the algorithm in the MICAz motes:

- Packet loss.
- Short time between packets preventing their processing in real time, in scenarios like Fig. 3.

To avoid on these problems, we have based the implementation only on unicast messages, although the broadcasting data would be desirable in terms of communication energy consumption. We have developed an additional layer to control the communications among motes and to ensure that all the unicast messages are received and processed by the destination motes. Finally, we focus on minimizing the number of retransmissions in scenarios like shown in Fig. 3.

To sum up, the following points must be addressed:

- To solve the floating point data limitations.
- Each mote must have transmission and reception buffers with enough capacity to attend to all the neighbours.
- If a received packet into the transceiver buffer cannot be attended by the microprocessor, it should be removed from the buffer to avoid overflow if possible.
- If a packet must be sent and the transceiver buffer is full, we must ensure that it will be sent when possible.
- The required time to process a packet must not affect to the communications to other motes.
- All the received packets must be processed.
- We need to achieve zero packet loss, trying to minimize as possible the number of retransmissions in the network.

6.1 Floating point computation

To face the limitations described in Section 5.2.1, we could scale the floating point variables and treat them like integer variables, but overflow problems were present.

To deal with these problems, we have adopted a two-step solution with good results:

1. Matrices are saved into the RAM-like scaled values (integers). Doing this we can pass them by reference to the matrix functions.
2. Once into each matrix function, all pointed data are saved into local variables (matrices), converting them to float variables without scale. Hence, internally, each function will do all the operations with floating point variables avoiding overflow problems. Finally, the results will be scaled again and returned like integer values.

The library is set to allow the user to select not only the scale of the input data but also the scale of the result.

6.2 Reception

Our proposed solution for reception is shown in Fig. 4. Let us have a comprehensive overview with a few points:

1. First, we must modify the native events and commands of TinyOS that manage the reception of packets, converting them from synchronous mode to asynchronous mode. This will make that whenever exist packets into the buffer, no processing operations will be done until all of them have been saved into the RAM. If some processing operation is running when a new packet gets into the transceiver buffer, the process will be stopped and continued once the packet has been extracted from the transceiver and saved into the RAM memory.
2. A new reception module must be implemented. It must include a circular buffer into the RAM memory to save all the incoming messages. Each time a packet is received from the transceiver (in asynchronous mode), it will be saved in this buffer.
3. The new module must handle a new acknowledgement mechanism at microprocessor level. A message is not acknowledged until it is saved into the circular buffer. This mechanism will require a pre-processing operation, as it will be detailed in *New-acknowledgement-mechanism*.
4. The reception operations must be transparent to the user.

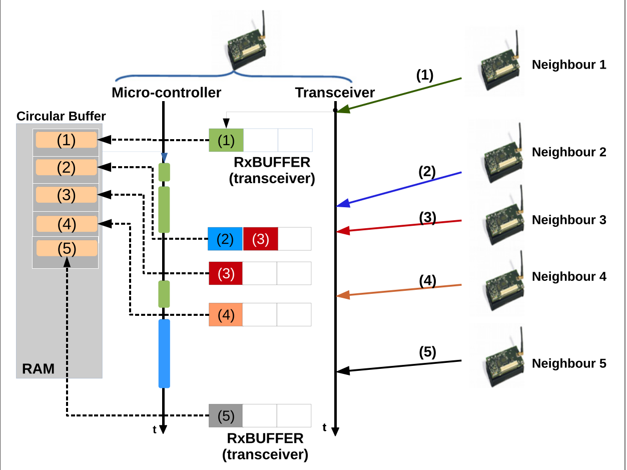

Fig. 4 Overview of the proposed solution for reception of data. Combining asynchronous reception events with the extension of the transceiver capabilities allows the handling of both communication and computation operations. Both the circular buffer and the data resulting from the m-DKLS algorithm are saved into the RAM memory

6.3 Transmission

The solution for transmission is similar, but in this case, it is not necessary to convert the corresponding native events and commands into asynchronous mode:

1. We must provide a circular buffer into the RAM memory to save all the messages that must be sent.
2. The new module for transmission must manage the new acknowledgement mechanism (detailed in *New-acknowledgement-mechanism*). It must manage the timer of the mechanism.
3. The transmission operations must be transparent to the user.

6.4 New acknowledgement mechanism

We propose to use the direct-sequence number (DSN) mechanism. The transmitter mote labels each packet and waits for a given time for a response. It must control situations like the scenario shown in Fig. 5, where we have a retransmission because processing times to

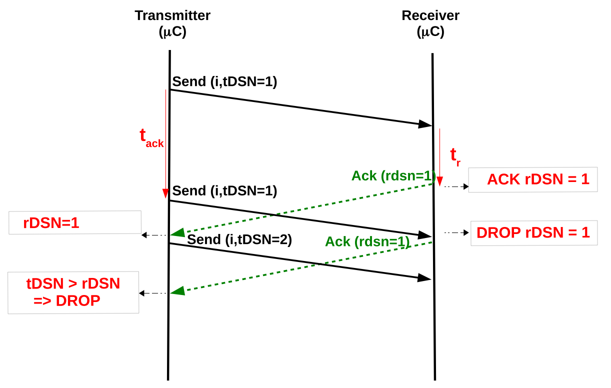

Fig. 5 Acknowledgement mechanism details. An example of special scenarios that must be controlled by the acknowledgement mechanism

acknowledge are larger than timers. The acknowledgement mechanism uses a timer to evaluate if a packet must be resent. We want to minimize the number of retransmissions in the network. In our solution, we are processing the incoming packets sequentially in the circular buffer, so if we do not send the acknowledgement until its turn of processing, retransmissions will be done.

In the reception of the acknowledgement, we have a similar problem. Suppose mote m_1 sends a packet to m_2 and starts the timer. When it is received by m_2, it will be attended in asynchronous mode to be saved into its reception circular buffer. Once it is saved, the m_2 sends an acknowledgement packet to m_1. Once the acknowledgement packet is received by m_1, it will attend it but a pre-processing is needed: this kind of packet must not wait their turn into the circular buffer in order to avoid unnecessary retransmissions. So, all the incoming packets must be pre-processed: if they are ACK packets, they will not be saved into the circular buffer; they must be processed immediately to clear the timer.

7 The implementation

Finally, we detail the components that provide the new functionalities described in Section 6. The solution is based on three new modules (see Fig. 6):

- The main component (moteC).
- The new transmitter component (QueuedSenderC).
- The new reception component (QueuedReceiverC).

We propose an upper layer to handle the wireless communications, apart from the module responsible for the execution of the m-DKLS algorithm.

7.1 The main module *moteC*

This module is connected to the native TinyOS component of receiving data from the CC2420. It means that when the transceiver component signals the reception of a new packet into its hardware buffer, that event is received in the module *moteC*. Previously, we have set that event in asynchronous mode.

The *moteC* module uses the interface *QReceive* provided by the new module *QueuedReceiverC* (detailed later). Through this interface, the module *QueuedReceiverC* gets new messages and pre-process them.

When a new message event is received by *moteC*, it gets the message from the transceiver and sends it to *QueuedReceiverC* through the *QReceive* interface. Everything in this chain is working in asynchronous mode so all these operations will be done with the highest priority.

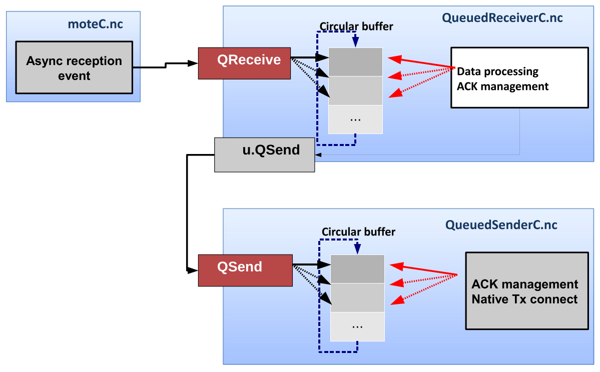

Fig. 6 Simple representation of the solution. Representation of the relationships between the modules of the new communication layer

7.2 The new reception component *QueuedReceiverC*

This new component implements our reception circular buffer. It also provides the *QReceive* interface and uses the interface *QSend* provided by the new module *QueuedSenderC*. When a new message is received from the *QReceive* interface, this component must pre-process it:

1. If the message is an acknowledgement of a previously sent message by this mote, this message must be sent to the *QueuedSenderC* component (through its interface *QSend*) to reset the timer. This operation must be immediately done, so the *QSend* interface must work in asynchronous mode too.
2. If the message is not an acknowledgement, it must be saved into the circular buffer. Once saved, an acknowledgement message must be sent immediately to the sender of the message, so *QueuedReceiverC* puts an ACK message into the *QSend* interface.

In our implementation, the *QueuedReceiverC* is also responsible for the execution of the m-DKLS algorithm, but it is executed in synchronous mode through several *tasks*. The operations related to m-DKLS algorithm could also be easily implemented into a separated module to have the communications independent from the algorithm. In synchronous mode, this component does the following operations:

1. Detection of non-processed messages by the m-DKLS algorithm into the circular buffer.
2. If there are non-processed messages, extract the oldest one, delete it from the circular buffer and process it in m-DKLS, obtaining a result.
3. Send the result to all the neighbours through the interface *QSend* of the component *QueuedSenderC*.

To avoid a continuous check loop to control the status of the circular buffer, we have used two pointers, one of them pointing to the last received message and another one pointing to the last processed message. Depending on the relative positions between them, we can easily know if there are non-processed messages. This involves a very low computational load. A simple flow chart of this new component is shown in Fig. 7.

7.3 The new transmission component *QueuedSenderC*

This component also implements a circular buffer for transmission, and it is controlled with the same mechanism of two pointers used in the module *QueuedReceiverC*. It provides the *QSend* interface in asynchronous mode and uses the native TinyOS modules to send the messages to the network. The following functionalities are implemented here in relation to the messages received by the interface *QSend*:

1. If the message is an acknowledgement of a packet previously sent by this mote, the module must reset the timer of the acknowledgement and continue processing the next message into the circular buffer of QueuedSenderC.
2. If the message is an ACK packet to be sent to another mote, the QueuedSenderC module sends it directly to the native TinyOS sender component without using the circular buffer, because of the highest priority of this kind of messages.
3. If the message is not an ACK packet, then it will be saved into the circular buffer. They will be sent at their turn according to the control pointers of the circular buffer.

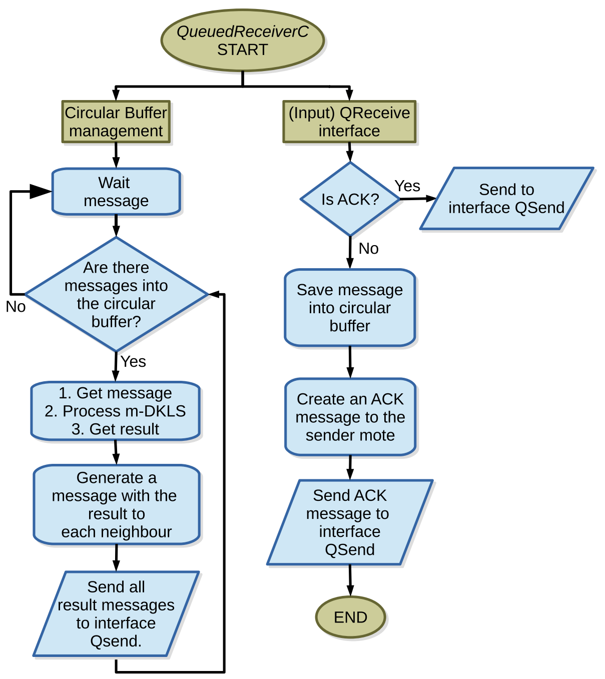
QueuedReceiverC START
Circular Buffer management
(Input) QReceive interface
Wait message
Are there messages into the circular buffer?
No
Yes
1. Get message
2. Process m-DKLS
3. Get result
Generate a message with the result to each neighbour
Send all result messages to interface Qsend.
Is ACK?
Yes
Send to interface QSend
No
Save message into circular buffer
Create an ACK message to the sender mote
Send ACK message to interface QSend
END

Fig. 7 Flow chart of the QueuedReceiverC component. A simple flow chart where the main operations managed by this new reception component can be easily understood

A simple flow chart of this new component is shown in Fig. 8.

7.4 Example of task execution

In this section, a simple example of this solution is included in Fig. 9 to illustrate the behaviour of the previous components. In the figure, the computation of one packet is compound by three tasks. The execution of the first task (the longest one) is interrupted when a new packet has been received by the transceiver and it is signalled to the processor. When the interruption is done, the received packet is saved into the circular reception buffer and cleaned from the transceiver buffer. All the incoming messages are processed sequentially.

8 Results

For the sake of simplicity and for demonstration purposes, we next include the implementation of the m-DKLS into a WSN with five motes linearly positioned and equally spaced, and in which each mote communicates with its nearest nodes. The distance between each two motes is 1 m.

We have used the following network configuration:

- Five motes executing the m-DKLS algorithm.
- One gateway mote, to get the results from the WSN and send them to the PC.
- One mote without sensor board, to broadcast a beacon signal each 6 s. Each time a mote receives this signal, it starts the m-DKLS algorithm.

We have used the temperature field to perform the experiments. The positions where we have estimated the

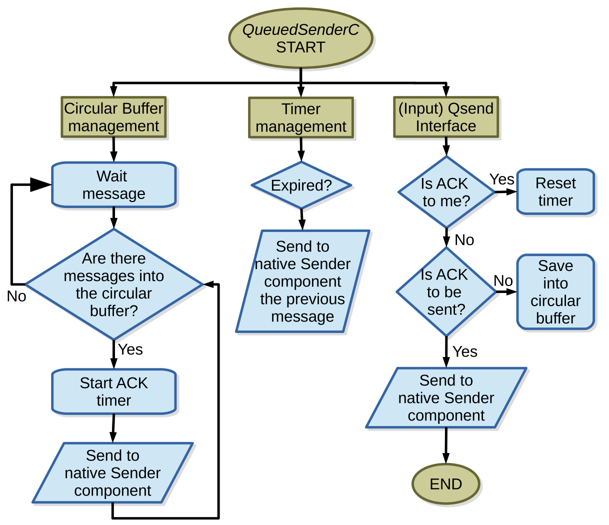
QueuedSenderC START
Circular Buffer management
Timer management
(Input) Qsend Interface
Wait message
Are there messages into the circular buffer?
No
Yes
Start ACK timer
Send to native Sender component
Expired?
Send to native Sender component the previous message
Is ACK to me?
Yes
Reset timer
No
Is ACK to be sent?
No
Save into circular buffer
Yes
Send to native Sender component
END

Fig. 8 Flow chart of the QueuedSenderC component. A simple flow chart where the main operations managed by this new transmission component can be easily understood

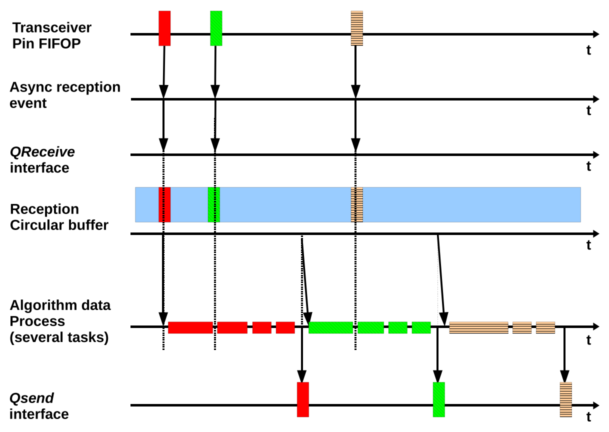

Fig. 9 Timeline for the reception of data in the training step. In this example, each packet needs three tasks to complete its processing, and the first one is interrupted by an asynchronous event of a new packet reception. Then, the mote saves the incoming packet into the new circular reception buffer before continuing the processing of this first task, emptying the transceiver buffer

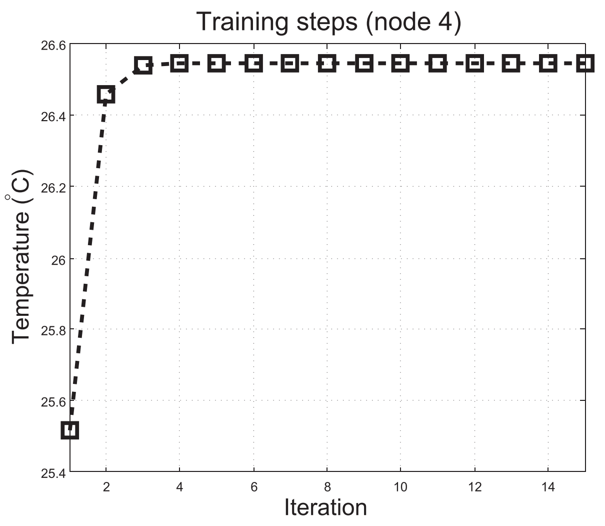

Fig. 10 Convergence in the training step of node 4. This figure illustrates the convergence of the iterations in the training step to the final value at point 3.6 and time 6 s

temperatures are in the same axis of the motes, in positions 1.2, 1.6, 2.4, 2.8, 3.2, 3.6, 4.4 and 4.8, while the motes are placed in positions 1.0, 2.0, 3.0, 4.0 and 5.0.

The selected kernel for this experiment has been the Gaussian kernel, and the number of training iterations has been fixed to 15. The neighbourhoods have been fixed into the source code.

To test the adaptation of the network to substantial variations of the measured field, we have used a heater, moving it sequentially from the position of one mote to the next one. Each variation in the position of the heater needs some time to get a stable measure by the sensors. We concluded that this time of convergence of a new accurate data is limited by the temperature sensor response rather than by the training step of the algorithm. Each 6 s, we computed 15 iterations of the algorithm (being the execution time shorter), although a quite good result can be reached using only five iterations, as shown in Fig. 10. However, in Fig. 11, it can be observed that during 100 s, the measure taken by the sensor is not stabilized.

As shown in previous sections, each node processes sequentially each received message. Some of them will be treated as acknowledgement messages and others as m-DKLS messages. Furthermore, these last packets can belong to the initialization step of the algorithm, to the training step or can be result messages.

One mote does not send a message if it has not received the acknowledgement of the previous one. This means that a mote can only have one message of each neighbour into its buffers regardless of the type of the message, so the maximum incoming messages that a mote can have is equal to the number of neighbours. The new transmission and reception queues should have enough space to save, at least, one message per neighbour. In the framework of our experiment, just a capacity of two data frames is needed.

In relation to processing and delay issues of this implementation, we have obviously a delay in comparison to a more capable mote in terms of transceiver and processor capabilities. Working with MICAz motes, we need several additional operations that make the execution of the algorithm slower: not only the additional communication layer and the speed of processing data take account, but also the floating point emulation gets importance in delay.

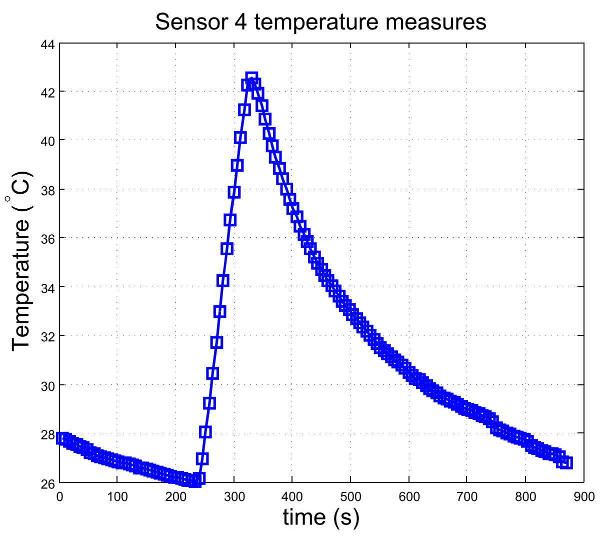

Fig. 11 Real measures taken from sensor 4 during the experiment. The heater was near to the sensor 4 position for 96 s (approximately), from time 234 s to 330 s

In a fixed topology like the one used in our work, the inclusion of more motes into the network would need modifications of the software in the existing nodes, as the neighbourhoods are fixed by source code. Nevertheless, if a dynamic procedure was used to establish the neighbourhood, as will be commented in Section 9.1, the scalability of the network would be also dynamic and no additional work would be necessary over the existing motes. The only limit would be given by the hardware capabilities of the motes.

The increase of RAM memory usage according to larger neighbourhoods does not vary in a linear way, since the size of the matrices depend on the number of neighbours. Figure 12 illustrates the relationship between the neighbourhood size and the static RAM memory increase. As an example (our implementation), if we set as reference value the approximate RAM memory used by a mote with two neighbours, the used RAM memory would increase in 450 bytes approximately if the neighbourhood was composed by five motes. For eight neighbours, it would increase in 600 bytes with respect to the case of five neighbours. Note that we need to allocate RAM memory to variables associated to the computation (matrices, auxiliary variables, topology definition...) apart from the buffers. If we use neighbourhoods up to 50 nodes, the necessary RAM memory of the new circular buffers would be approximately of 1.6 kBytes. In Fig. 12, it can be observed that we would need approximately 25 kBytes in the case of a neighbourhood of 50 nodes. The 6.5 % of this needed RAM memory would be allocated for the buffers, concluding that in large neighbourhoods, the RAM memory needs are controlled by the algorithm rather than by the communication layer.

In computational terms, the increase due to larger neighbourhoods are not linear, neither. To compute the m-DKLS, we need operations like matrix inversion, which we have implemented using the Gauss-Jordan method. This implies a computational load of $\mathcal{O}(n^3)$, where n is the number of neighbours plus 1. Also, we need matrix product, which has the same computational load in our implementation.

The RAM memory becomes the real limit in terms of scalability of the algorithm. In relation to computational load, due to the sequential execution model of our implementation, it is not a strict limit as the only consequence is to have larger delays in the computation.

To experimentally validate the implementation, we compare the results obtained by the m-DKLS implemented in the WSN to the results obtained offline by the theoretical m-DKLS solution. To achieve this last point we have numerically[1] simulated the same network topology, and the real temperature measures from the motes have been incorporated too. Figures 13, 14 and 15 illustrate the comparison at different times of the experiment. The black dashed line connects the different temperature estimations obtained from the numerical simulation by computer at all the intermediate positions. It can be observed that the real-time implementation with the MICAz network (red squares) provides the same results of the numerically offline simulated solution (dashed), validating our development and demonstrating the implementation feasibility.

As already discussed, in Fig. 10 it is illustrated an example of how the iterations of node 4 in the training step converge to the final estimation of the position 3.6 shown in Fig. 13. This has been obtained from a theoretical m-DKLS simulation using the real temperature measures at time 6 s.

A demonstration video can be found at [19] and the corresponding source code in [20].

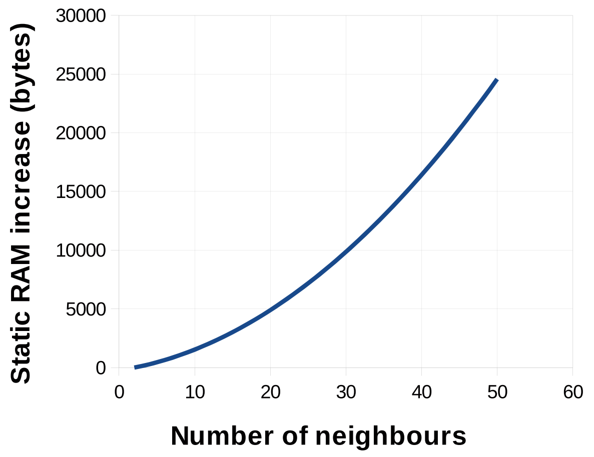

Fig. 12 Theoretical approximation of RAM memory increase. Estimated RAM memory increase with respect to a neighbourhood size of two of our implementation

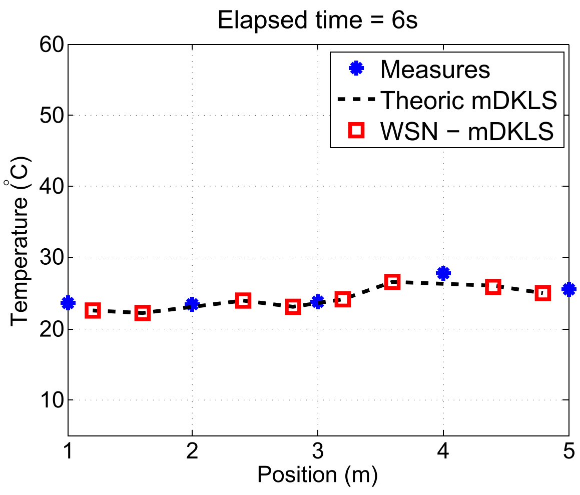

Fig. 13 Experiment result start time. The heater is off yet

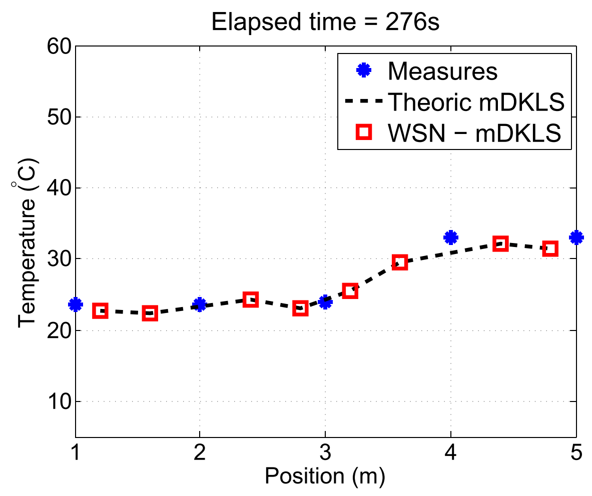

Fig. 14 Experiment result at 276-s elapsed time. The heater is positioned near to node 5

9 Conclusions

Non-linear kernel methods can be defined in a distributed approach. This distributed description, such as one of the m-DKLS, allows for a suitable implementation in WSN, which is asynchronous by nature. However, WSN are simple architectures where motes have limited communication and computation capabilities that make the implementation of this complex algorithm difficult. In this work, we identify the major problems to be solved, namely 1) emulate floating point (and matrix) operation with integer ones, 2) avoid synchronous standard operation to prioritize message transmission and reception, 3) provide a safe acknowledgement mechanism and 4) prevent a poor initialization of the algorithm.

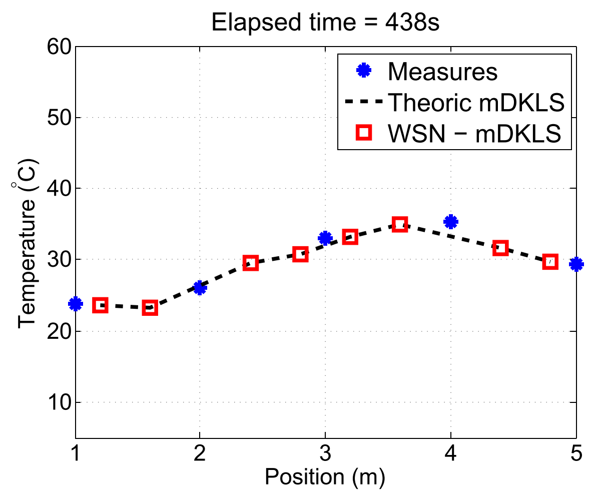

Fig. 15 Experiment result at 438-s elapsed time. The heater is positioned between nodes 3 and 4

In this paper, we propose novel solutions to all of them. Regarding to the avoiding of synchronous standard operation, we propose a new layer based on three blocks: moteC, QueuedReceiverC and QueuedSenderC. These blocks extend native communication capabilities with circular buffers and the integration of a suitable control mechanism. In addition, we propose to modify some native libraries related to communication handling as a complement of the new layer[2]. These improvements help managing the transmission of messages, avoiding losing them due to buffer overflow, and reducing the number of retransmissions, while increasing the maximum number of messages (neighbour motes) that can be handled and improving the network efficiency. These improvements allow for a real-time implementation. We demonstrate the performance of the proposed implementation in a scenario where temperature is monitored. The solution was able to provide a good non-linear regression estimation along with adaptation capabilities to changing conditions. We experimentally evaluated the performance of this approach. The real implementation in the wireless sensor network provided the same solution than the output computed offline, assuming perfect communications in a numeric programming language.

9.1 Future work

In addition to focusing on larger networks and different topologies to check for scalability, we propose a future line of research to study dynamic neighbourhoods, as it arises as a problem from the implementation point of view. In particular:

- To model a dynamic neighbourhood discovering step using broadcast messages. A possible starting point could be the following: the discoverer mote would send a certain broadcast message that should be answered by all accessible motes with an unicast message, and their initial field measures could be included. Once received the response messages, the discoverer could then define its neighbourhood to execute the algorithm.
  In relation to these unicast messages, we can consider two possibilities:
  - The unicast message is using a transmission control mechanism to ensure that it is received by the discoverer mote (desirable). In this case, the discoverer mote would then take account of all the accessible motes.
  - The unicast message is not using a control transmission mechanism. If this was the case and the message would not have been received, the discoverer mote would not take account of this neighbour mote when defining

its neighbourhood, and the algorithm would work properly.

- To use a mixed transmission mechanism for the training step by which both broadcast and unicast messages can be used to reduce the number of transmissions keeping control of transmissions.
- The study of a trade-off between the neighbourhood size and the number of iterations in the training step that can be achieved in terms of memory consumption and delay, this last one due to the increase of the computational load.
- The management of RAM memory allocation and limits in a dynamic way when creating neighbourhoods.

These lines of research can bring us useful results to achieve less dependence of the implementations from the mote hardware capacities, as well as better WSN performance when working with distributed WSNs: less message transmissions and energy consumption with similar reliability in communications.

Endnotes

[1]Using Matlab.

[2]See the reference guide in [20] for further details.

Abbreviations

WSN, wireless sensor network; KLS, kernel least squares; DKLS, distributed kernel least squares; m-DKLS, modified distributed kernel least squares.

Competing interests

The authors declare that they have no competing interests.

Acknowledgements

This work was partially funded by the Spanish government (Ministerio de Ciencia e Innovación, Consolider-Ingenio CSD2008-00010,TEC2012-38800-C03-{02} and European Union (FEDER).

Received: 31 October 2014 Accepted: 13 May 2015
Published online: 12 June 2015

References

1. F Zhao, L Guibas, *Wireless Sensor Networks. An Information Processing Approach*. (Morgan Kaufmann, Massachusetts, 2004), pp. 9–15
2. S Vijayakumar, JN Rosario, in *Int. Conference On Communication Software and Networks (ICCSN)*. Preliminary design for crop monitoring involving water and fertilizer conservation using wireless sensor networks (IEEE, Mumbai, 2011), pp. 662–666
3. P Choudhari, in *Int. Conference on Communication, Information & Computing Technology (ICCICT)*. Development of vehicle tracking system using passive sensors (IEEE, Mumbai, 2012)
4. E Lee, S Park, F Yu, S-H Kim, Exploiting mobility for efficient data dissemination in wireless sensor networks. Commun. Netw. J. **11**(4), 337–349 (2009). doi:10.1109/JCN.2009.6391347
5. X Nguyen, MI Jordan, B Sinopoli, A kernel-based learning approach to ad hoc sensor network localization. ACM Trans. Sensor Netw. **1**, 2005 (2005)
6. JB Predd, SR Kulkarni, HV Poor, A collaborative training algorithm for distributed learning. IEEE Trans. Inf. Theory. **55**(4), 1856–1871 (2009)
7. F Pérez-Cruz, SR Kulkarni, Robust and low complexity distributed kernel least squares learning in sensor networks. IEEE Signal Process. Lett. **17**(4), 355–358 (2010)
8. S Varcarcel, P Belanovic, S Zazo, in *Int. Workshop On Signal Processing Advances in Wireless Communications (SPAWC)*. Consensus-based distributed principal component analysis in wireless sensor networks (IEEE, Marrakech, Morocco, 2010)
9. D Djenouri, N Merabtine, F Zohra, M Doudou, in *Wireless Communications and Networking Conference (WCNC): Services & Applications*. Fault-tolerant implementation of a distributed MLE-based time synchronization protocol for wireless sensor networks (IEEE, Shanghai, 2013)
10. J Jao, B Sun, K Wu, in *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*. A prototype wireless sensor network for precision agriculture (IEEE, Philadelphia, 2013), pp. 280–285
11. JB Predd, SR Kulkarni, HV Poor, Distributed learning in wireless sensor networks. IEEE Signal Process. Mag. **23**(4), 56–61 (2006)
12. GS Kimeldorf, G Wahba, Some results in Tchebycheffian spline functions. **33**, 82–95 (1971)
13. Y Censor, SA Zenios, *Paralel Optimization: Theory, Algorithms, and Applications*. (Oxford University Press, New York, 1997)
14. HH Bauschke, JM Borwein, On projection algorithms for solving convex feasibility problems. SIAM Rev. **38**(3), 367–426 (1996)
15. P Levis, Tinyos programming. Technical report, 38–45. Stanford University (2006). http://csl.stanford.edu/~pal/pubs/tinyos-programming.pdf
16. ATMEL, 8-bit Atmel Microcontroller with 128 KBytes In-System Programmable Flash:ATMEGA128 and ATMEGA128L. ATMEL. http://www.atmel.com/images/doc2467.pdf
17. Chipcon, CC2420: 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver. Chipcon. http://www.ti.com/lit/ds/symlink/cc2420.pdf
18. TinyOs - CC2420 Hardware and Software Acks. http://tinyos.stanford.edu/tinyos-wiki/index.php/CC2420_Hardware_and_Software_Acks
19. JA Garrido-Castellano, JJ Murillo-Fuentes, On the implementation of distributed asynchronous non-linear kernel methods over wireless sensor networks: demonstration video. Universidad de Sevilla (2013). http://youtu.be/HaxlaGaYxqo
20. JA Garrido-Castellano, JJ Murillo-Fuentes, On the implementation of distributed asynchronous non-linear kernel methods over wireless sensor networks: source codes. Universidad de Sevilla (2013). http://personal.us.es/murillo/papers/mDKLS4WSNcode.gz