Scientia Iranica D (2019) 26(6), 3540–3555

Sharif University of Technology
Scientia Iranica
Transactions D: Computer Science & Engineering and Electrical Engineering
http://scientiairanica.sharif.edu

RUbIn: A framework for reliable and ubiquitous inference in WSNs

A. Shamsaie[a,*], J. Habibi[a], E. Abdi[a], and F. Ghassemi[b]

a. *Department of Computer Engineering, Sharif University of Technology, Tehran, Iran.*
b. *School of Electrical and Computer Engineering, College of Engineering, University of Tehran, Tehran, Iran.*

Received 10 August 2018; received in revised form 8 February 2019; accepted 15 April 2019

KEYWORDS
Internet of Things (IoT);
Wireless Sensor Network (WSN);
Ubiquitous inference;
Framework;
Gossiping protocol.

Abstract. Development of (Internet of Things) IoT applications brings a new movement to the functionality of Wireless Sensor Networks (WSNs) from only environment sensing and data gathering to collaborative inferring and ubiquitous intelligence. In intelligent WSNs, nodes collaborate to exchange the information needed to achieve the required inference or smartness. Efficiency or correctness of many smart applications relies on the efficient, timely, reliable, and ubiquitous inference of information. In this paper, we introduce the RUbIn framework, which provides a generic solution to such ubiquitous inferences. It brings reliability and ubiquity to inferences using the redundancy characteristic of the gossiping protocols. With RUbIn, the implementation of such inferences and the control of their speed and cost are abstracted by providing developers with a proposed middleware and some dissemination control services. We developed an implementation prototype of the RUbIn framework and a few inference examples of TinyOS. For evaluation, we utilized both the TOSSIM simulator and a testbed of MicaZ motes in various densities and different numbers of nodes. Results of the evaluations demonstrated that in all nodes, the inferring time after a change was about a few seconds and the cost of maintenance in stability state was about a few messages sent per hour.

1. Introduction

Sensor motes are small smart devices that integrate the advantages of computing, communication, and sensing systems into a compact element. These advantages provide WSNs with the ability of intelligence, which ensures their deployment as networked embedded systems in smart applications [1]. Development of IoT applications brings a new movement to the functionality of WSNs from only environment sensing and data gathering to collaborative inferring and ubiquitous intelligence [2-5]. The difficulty of perceiving the constraints on the resources of nodes and the complexities brought by these constraints to application development should not be a barrier for the development of WSN/IoT applications. Simplifying the application development by the contribution of software and programming language experts can increase the speed of WSN development. In order to reduce these complexities, it is necessary to create new programming paradigms. Hence, the number of research studies and projects focusing on effective frameworks or middleware are increasing. These frameworks or middleware enfold the constraints and complexities of WSNs and provide a convenient abstraction for programmers [6-8].

In intelligence WSNs, nodes collaborate to exchange the information needed to achieve the required

*. *Corresponding author. Tel.: +98 21 66166244*
E-mail addresses: shamsaie@ce.sharif.edu (A. Shamsaie); jhabibi@sharif.edu (J. Habibi); eabdi@ce.sharif.edu (E. Abdi); fghassemi@ut.ac.ir (F. Ghassemi)

doi: 10.24200/sci.2019.51495.2215

inference or smartness [9-13]. The efficiency, correctness, or smartness of many protocols or applications of WSNs rely on efficient, timely, reliable, and ubiquitous inference of information. Some necessary inferences in WSN are ubiquitous as it is required at all the nodes. They are often active as all the nodes are continually tracing changes to keep their inferred information up-to-date. They should also be reliable because, in a connected network, the inferred information at all the nodes should be updated in a short time after a change at any node. In this paper, our focus is on such inference problems. Thus, hereafter, the term inference refers to a ubiquitous, active, and reliable inference of information. Additionally, efficiency in energy consumption, the speed of inference after a change, effectiveness in different densities, and number of nodes are other characteristics which can be found out in most of these inferences in response to the constraints on nodes and the requirements of applications. We refer to these characteristics as low maintenance cost, fast inference, and scalability, respectively.

Research studies focused on a generic solution to inference problems are neglected in WSN. Similar characteristics of inference algorithms and resource constraints on WSNs motivated us to propound a framework as a generic approach to the development of inference algorithms. It provides functionalities common to the whole class of inference algorithms and a set of left-blank modules to be filled in by the programmers. An inference algorithm is implemented only by instantiating the left-blank modules and filling them in by the inference-specific logic. The framework abstracts the inference algorithms from the propagation protocol (gossiping) by providing some standard services, which the programmer can exploit to moderate the cost, speed, and scalability of an inference algorithm. It brings separation-of-concerns for a complex protocol or application when an inference is needed.

The paper continues as follows. The next section studies related work. Section 3 describes the problem and Section 4 analyzes the RUbIn requirements. Section 5 presents the RUbIn framework and Section 6 evaluates it. Finally, Section 7 concludes the paper and describes possible future work.

2. Related work

Some types of ubiquitous, active, and reliable inferences can be found within different software layers of many applications in WSNs. A framework like RUbIn brings efficiency and robustness to these inferences, which are essential prerequisites for the efficiency of the main applications relying on them. To the best of our knowledge, there is no similar framework to facilitate the development of such inferences. Only a few inference algorithms based on periodic message passing are found in some applications or middleware.

In WSNs, key-distribution algorithms are categorized into two types of random and regular [14]. In both types of these algorithms, you can find tracks of inference in identifying the overlay neighboring nodes that are also physically neighboring nodes through a shared key, finding the overlay path, and finally, formulating the overlay network. With RUbIn, this inference can be simply and efficiently implemented such that not only existing nodes but also future joined nodes will participate in algorithms.

In Mate [15] middleware, nodes actively infer the latest version of a code such that if a node obtains a newer version of a code, after a while, all nodes will obtain it. There are other protocols for the dissemination of codes in WSNs [16-18]. These protocols use a gossiping protocol to reliably disseminate the meta-data of a new code to all the nodes and make them aware of the new code. In these protocols, if no change occurs for a while, then the period of gossiping will be increased to reduce maintenance cost; otherwise, the period is reset to its lowest value to increase the dissemination speed and hence, the inferring speed.

The RUbIn framework is also based on gossiping protocols with some programming interfaces to increase or decrease the gossiping period. Although RUbIn employs the idea of these two protocols, it is more than a dissemination protocol. For example, in many inference problems, such as inferring the average surrounding temperature, making an approximation of local density, or identifying the shortest path to a sink, each node may infer a different value. Consequently, many inference algorithms, which can be developed in RUbIn, are more complicated than only inferring a shared data (here, meta-data). Furthermore, in many cases, in inference algorithms, a measure to score the surrounding nodes or the information received from them is required. To this aim, RUbIn provides one of the most common measures, namely link quality, as an existing default service. In many cases of inference problems, the quality of links to surrounding nodes can be exploited to develop more efficient or precise algorithms.

In collection routing protocol in [19], a tree is established to collect information from nodes. Through a gossiping protocol and a link quality estimator, an efficient, robust, and reliable routing protocol in WSNs is achieved, even if the number of topology changes is high. The design of the RUbIn architecture is inspired by this protocol to take advantage of its characteristics.

In [20], middleware for simplifying application development in WSNs using the publish/subscribe model is proposed. Behind this middleware is a routing protocol based on a tree construction, which should be updated with any change in publishers, subscribers,

or the network topology. In this middleware, periodic beacon is used to establish a routing tree while in RUbIn, a more stable routing tree can be efficiently inferred.

3. Problem statement

There is a multi-hop WSN consisting of n nodes. Every node m executes an application a_m, which can be different from or identical to other applications. The link between nodes m and k denoted by $\ell_{m,k}$ can vary in quality for several reasons, such as noise, congestion, battery energy reduction, periodic sleep, etc. All the nodes, regardless of their running applications, have an active inference on a deterministic set of information $C = [I_1, I_2, \cdots, I_{|C|}]$ in their interaction with each other. The value of any information I_j at node m at time t, denoted by $I_j(m,t)$, where $1 \leq j \leq |C|$, may vary in all nodes over the time. Every information I_j in every node m is initialized with the value $v0_j$ and then, updated for various reasons, such as changes in the number of active nodes, variation in the quality of links, updates of information of neighboring nodes, changes applied by the application or a user, or changes in sensing values. Even though these changes may be mild and localized, they still may affect the accuracy of information in other nodes. Therefore, all nodes should trace these changes and consider them in their inference algorithms. Also, they should inform the other nodes of any changes in their own information to ensure that after a short time, the information at all nodes is accurate and up-to-date. In contrast, sometimes, there is a high interval between changes and, meanwhile, the information is stable. In this situation, the message passing for keeping the information up-to-date is extra overhead. Thus, a mechanism is needed to moderate this overhead. In general, the inference framework should consider the following challenges:

- **Reliability:** Topology changes should not prevent a node from inferring accurate and up-to-date information. Thus, all nodes connected to the network should ultimately obtain any information needed to update their information;
- **Inference speed:** Latency in an inference after a change taking place anywhere in a network may have side effects on the efficiency or behavior of an application. Thus, updating information should start and end immediately from the changing origin to where it is necessary;
- **Scalability:** The efficiency of an inference framework should be independent of the size or density of the network, and it should preserve its characteristics in large or small and in dense or sparse networks;
- **Maintenance cost:** Resource constraints in WSNs, especially energy constraint, should be considered in all mechanisms within the inference framework. Thus, all the above characteristics, namely reliability, inference speed, and scalability, should be achieved considering these constraints.

Furthermore, in many cases, in inference algorithms, quality of the link is a common measure to score the surrounding nodes or the information received from them. Using the quality of links in inference algorithms may result in more stable information and lower cost. In other words, inference algorithms based on the information received from nodes having more stable links prevent temporarily inferred values and their side effects (successive inferences). Therefore, not only the accuracy of the information is increased, but also its maintenance cost also decreased.

4. Requirements analysis

In this section, we analyze the requirements of the inference framework mentioned in Section 3 and discuss important points that should be considered in its architecture.

Monitoring all changes which have an effect on information is one of the requirements of a ubiquitous and reliable inference algorithm. We studied these changes and divided them into three categories. In other words, three main factors were identified:

1. **Application:** Running applications may have their own values in contributing to an inference. Sometimes, they reset the information to a given value. The given value can be a result of a new sensor value, a new command from a user, the logic of the application, etc. Thus, inference framework should provide an interface for the applications to contribute their values to the inference;
2. **Time:** Occasionally, the elapsed time from a given point in the past, like the latest update time or the latest confirmation time, may be considered in inference algorithms. Thus, time is another factor affecting the information. Generally, in inference algorithms, the time factor appears as a periodic check of the validity of information. In some inference problems, in which elapsed time is not important, this factor is not considered;
3. **Message:** The most common factor in WSNs, which participates in all inference algorithms, is the messages received from neighboring nodes. Nodes should inform each other about their information and merge the received information with their own. Changes in the received values from neighboring nodes are the results of changes in the topology or the nodes caused by the three main factors in the neighboring nodes.

The information is initialized at the beginning of the inference algorithm and then, updated by these three factors. To better understand these three factors, three examples of inference problems with different levels of complexity are explained.

1. **Providing a shared memory:** To realize a shared memory in WSNs, all nodes should have their own allocated memory, which is always updated with the latest modification at any node. In this inference problem, all nodes infer the values of the node at which the latest modification occurred. Here, the two factors of application and message are effective. The application factor initiates a change in the local memory of a node while the message factor transmits an update in the memories of all the other nodes. Here, the time factor has no role;
2. **Consensus on a quantity:** One of the research topics in WSNs is the consensus problem. For example, when all nodes have their own values of a quantity, and they all want to infer the maximum value, a consensus to find out the maximum quantity is required. This problem is an inference problem in which all three factors are present. The application factor contributes a value of the quantity to the inference. The time factor checks whether the last consensus result is still valid by keeping the elapse time from the latest update. The message factor informs all the nodes of any change in consensus result at any node;
3. **Finding a robust route to a sink:** In a WSN, one or more nodes play the role of sink to collect information. Nodes in interaction with each other find a route (usually the shortest robust path) to a sink to send their information. In this inference problem, the three factors are present. The application factor allows only a node to introduce itself as a new or removed sink. The time factor checks the route validity so that if the current route is not confirmed for a given time period, then it is expired. The message factor informs other nodes if a new route is found at any node. The other nodes update their routes if a better one (shorter robust path) is found.

Almost all the inference algorithms have a data structure for inferring information and the required metadata. The data structure consists of multiple data fields, which can be divided into private and public parts. Both of these parts may change over time, but only the public part is sent to the neighboring nodes.

Because of the high dynamics of the network, ensuring reliability and robustness of WSNs applications is possible only through repetition. In other words, a message from a node will be reliably received by its neighboring nodes if it is periodically disseminated for a finite or infinite number of times. Thus, to reliably achieve a precise inference in all nodes after a change in any node, periodic information dissemination, like gossiping protocols in wireless networks, is needed.

Wireless communication is the main energy consumer in a sensor node. Thus, hereafter, cost refers to the number of sent messages. The speed and cost of gossiping protocols are inversely related to the gossiping period; i.e., the shorter the period, the higher the speed and the cost and vice versa. Thus, a dynamic gossiping period is recommended.

Due to repetition in gossiping protocols, increasing the network size will not decrease efficiency, unless this increase brings an excessive increment to the density. A gossiping protocol in a dense network results in congestion and collision of messages and, consequently, reduction in efficiency. In most inference algorithms, the number of needed messages for a reliable inference in a proximity is independent of the number of nodes located in that proximity. This fact is not considered in gossiping protocols. A solution to this problem is to provide a mechanism by which the nodes that eavesdrop the messages in their proximity can eliminate sending if it is wasteful. Consequently, whenever density of a proximity increases, the probability of such eliminations is also increased. Therefore, this mechanism can restrict sending to a small number in each proximity.

Link quality estimation in WSNs is a kind of ubiquitous inference algorithm that is frequently needed in many other inference algorithms. The 4-bit link estimation algorithm [21] is adapted in our framework. To reduce the overhead of this algorithm, its messages (beacons) can piggyback on other messages of the framework.

5. Framework design

The RUbIn framework is a general solution to the inference problems mentioned in Section 3. This framework facilitates the development of inference algorithms by providing all the functionalities common to this class of inferences. We designed the RUbIn framework with regard to the analysis of its requirements in Section 4.

5.1. RUbIn framework stack

As depicted in Figure 1, the stack of the RUbIn framework consists of three layers and each layer has a data unit.

In the information inference layer, information is included in a data structure consisting of public and private parts. Both of these parts participate in an inference algorithm and are accessible by the applications, but only the public part is available to the lower layers and consequently, to other nodes. Thus, the length of the public part is restricted to a few bytes

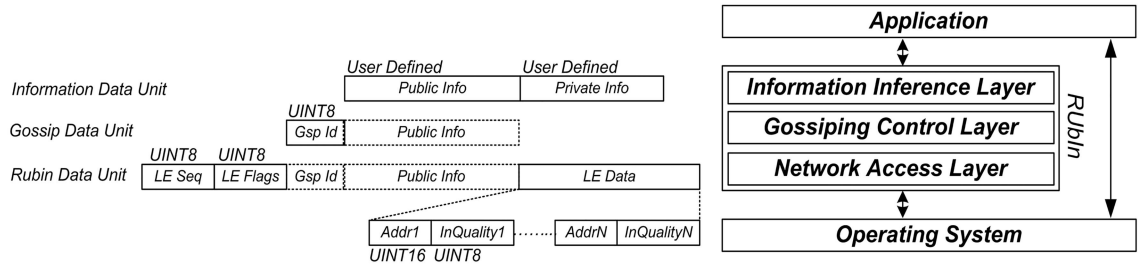


Figure 1. Stack of the RUbIn framework and data unit of each layer.

less than the maximum packet size so that it can be sent in one packet. Unlike the public part, the private part consists of information only beneficial to the current node with an arbitrary length.

The gossiping control layer controls the dissemination of information. As depicted in Figure 1, the data unit of this layer adds an 8-bit unique identifier of the information as a header to the public part of the upper-layer information.

The network access control provides services for the gossiping control layer to interact with the network. The data unit of this layer (RUbIn data unit) consists of a header and a footer in addition to a gossiping message of the upper layer. Both the header and the footer are used for link quality estimation according to our modified version of the 4-bit link quality estimation algorithm. The header contains two fields: an 8-bit field as the sequence number of the sent messages and another 8-bit field consisting of a 4 bits as the number of entries in the footer and a 4 bits as the flags used in the link quality estimation algorithm. The footer consists of some pairs each including a 16-bit node address and an 8-bit estimation of the input link from this node. The number of pairs, N, depends on the extra available spaces of each packet, so a round-robin manner is used to send all such pairs. If we consider L_{pckt} as the maximum length of a packet and L_{pub} as the length of the public part of information, then we have $L_{pub} \leq L_{pckt} - 3$. Therefore, $N = \lfloor \frac{L_{pckt} - L_{pub} - 3}{3} \rfloor$. When $N > 0$, the information in the estimation algorithm can piggyback on gossiping messages. Thus, at least for one case, we should have $L_{pub} \leq L_{pckt} - 6$ to attain a precise estimation of the links.

5.2. RUbIn framework architecture

In Figure 2, the architecture of the RUbIn framework and its layers is depicted by a component diagram. In this diagram, components are divided into the skeleton and extended components. The skeleton components are components which have already been implemented in the RUbIn framework, while the extended ones denote the components that users develop and add to the framework. Therefore, the network access layer and the gossiping control layer belong to the skeleton part, while the information inference layer has components

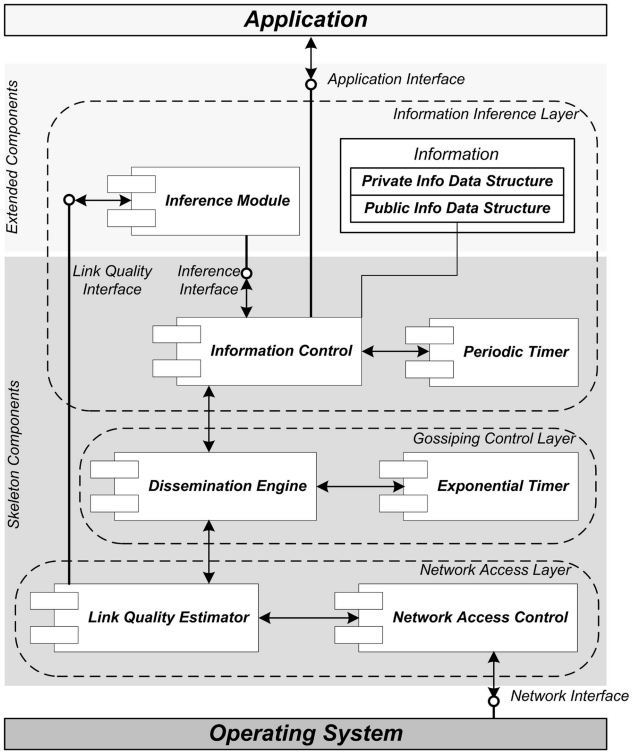


Figure 2. Component diagram of the RUbIn framework.

in both parts. In the following, we describe each of the RUbIn components in more details.

The network access control component manages the transmission of RUbIn packets between network access layers of neighboring nodes. This component uses the network interface provided by the operating system to distinguish, send, and receive RUbIn packets from the network.

The estimation information can piggyback on messages of the dissemination engine through the link quality estimator component. Therefore, link quality estimation is performed during gossiping of other information and in case there is no information for inference or no space for piggybacking, quality estimation of the links would be unfeasible. To solve this problem, when the sending rate of the information on the links is lower than a threshold, we send a few distinct beacons to achieve a precise estimation of the links. The following services to access the quality of input, output, and

bidirectional links are now available by the link quality interface for use in inference modules:

- *getBackwardLinkQuality (neighborId:Addr): int*;
- *getForwardLinkQuality (neighborId:Addr): int*;
- *getLinkQuality (neighborId:Addr): int*.

The exponential timer component provides an array of exponential timers for the dissemination engine, one for each inferring information. An exponential timer is a virtual timer the period T of which increases exponentially so that it is initialized at a minimal value t_ℓ (about a few seconds) and becomes automatically c times after each period. The period increases at most k times and finally reaches a maximum value t_h, where $t_h = c^k t_\ell$ (about a few hours). Afterwards, the number of periods remains constant at about a few in a day. When an exponential timer fires, it requests the dissemination engine to send the corresponding information. To decrease the probability of congestion, collision, and energy waste, we follow the idea of the trickle timer [22] so that in a period T, the timer fires at a random time t_{tick}, where $t_{tick} \in [\frac{T}{2}, T]$, instead of the end of the period. At time t_{tick} in each period, if a timer has not received a cancellation request from the dissemination engine, then it fires immediately. Each timer can be reset by the dissemination engine to t_ℓ at any moment of time, even before $T = t_h$.

The dissemination engine is the core of the RUbIn framework and responsible for information gossiping. This component is in interaction with some information control units equal to the number of inferring information units, and an exponential timer component with one virtual timer per information unit. When this engine is initiated, it initiates all the information control units and then, requests the exponential timer component to launch an exponential timer per control unit. Then, when a timer fires, the dissemination engine sends a gossiping message consisting of the public part of the corresponding information to the link quality estimator component. Also, if a gossiping message is received from the link quality estimator component, this engine delivers it to the corresponding information control unit. Furthermore, the dissemination engine provides the following services for each of the information control units with the aim of managing the dissemination period. In other words, we summarize all modifications to the default gossiping trend of each information unit by the following four services.

1. *SendFast():* This service increases the dissemination speed of the information. To this aim, it resets the exponential timer to t_ℓ, which increases the dissemination speed and consequently, the inference speed of the corresponding information for a while. In contrast, this service will also increase the inference cost;
2. *SendImmediately():* This service immediately sends the information. However, the engine disseminates the information once per period, but immediate sending of a message before or after t_{tick} may occasionally be needed. This service does not influence the dissemination period, but can be used instead of *SendFast* in inference algorithms to immediately disseminate information and increase inference speed with no significant change in cost;
3. *BeQuiet():* This service eliminates dissemination of information in the current period. In fact, if this service is invoked before t_{tick}, the corresponding exponential timer will not fire in the current period. This service does not influence the dissemination period and is used to decrease the inference cost, especially when the density of nodes is very high;
4. *SendImmediatelyAndFast():* This service combines the first two services such that, at first, immediate sending is performed and then, the dissemination period is reset to t_ℓ with the aim of increasing the inference speed. In this service, propagation over one path is at least $\frac{t_\ell}{2}$ less than the SendFast service per hop and can totally be about a fraction of one second.

Some information control units are in interaction with the dissemination engine. A programmer instantiates these units as many as the number of distinct inferences required in an application so that each one knows its information structure and the relevant inference module defined by the programmer. An information control unit is a gateway for all the three main factors to participate in an inference. In other words, an information control unit can receive messages (message factor) from the dissemination engine, commands from the application (application factor) using the application interface, and check requests from a dedicated periodic timer (time factor).

The inference module of information is the only place in which the information can be modified. The information is initialized in this module and then, modified in response to the requests of the three main factors over time. The requests of these factors are sent to this module by the corresponding information control unit using the inference interface. Therefore, for each inference module, the following list of services (defined in inference interface) should be implemented:

- *init (curInfo:InfoType): void*;
- *set (curInfo:InfoType, newInfo:InfoType): DissCmd*;
- *check (curInfo:InfoType): DissCmd*;

- *aggregate (curInfo:InfoType, newPubInfo: PubInfoType, senderId:Addr): DissCmd.*

The information control unit initializes the information by calling the *init* service. Then, it handles requests of the application, time, and message factors by calling the *set*, *check*, and *aggregate* services, respectively. The return value of these three services is the only means a programmer has to manage the dissemination trend and consequently, the inference speed and cost. The *set* service sets or merges the information value with an application value. The *check* service performs periodic validation or modification of information, if needed. Finally, the *aggregate* service aggregates the newly received message with the local information. Since the sender identifier or the quality of its input or output links is needed in some inferences, the sender identifier is also known in a request for *aggregate* service. The type of return values ($DissCmd$) in these services is one of the following five values:

enum DissCmd{*GoOn, SendFast, SendImmediately, BeQuiet, SendImmediatelyAndFast*}.

If the return value is *GoOn*, the information control unit will not do anything. Otherwise, it calls the equivalent service of the dissemination engine for the information.

An application can access the following services provided by the information control unit using the application interface.

- *get (): InfoType*;
- *set (newInfo:InfoType): void*;
- *setChangeEventHandler (funcName:FuncPtr): void*;
- *setCheckTimer (period:Time): void.*

The first two services request the information control unit to set and get the information, while the third one asks it to register an event handler for the case a modification occurs. The last service is used to activate the time factor by setting the period duration to a positive value. Zero value means that no time factor is needed.

In summary, to add a new inference algorithm, a programmer should define the data structure of information and implement *init*, *set*, *check*, and *aggregate* services.

6. Evaluation

We evaluate the RUbIn framework from two aspects of effectiveness and efficiency. To this aim, with the two examples in Section 6.1, we demonstrate how an inference algorithm can be implemented. Then, in Section 6.2, we evaluate the effectiveness of RUbIn and the efficiency of inferences developed with RUbIn.

6.1. Developing inferences in RUbIn

We develop the pseudo-codes of the two inference problems discussed in Section 4 using our framework. The required data structures and the services of relevant inference modules are defined. Development of these two examples demonstrates how the RUbIn framework effectively helps the programmer to focus on the inference algorithm and simply manage the dissemination trend.

6.1.1. Awareness of the latest version of an application (shared memory)

In every in-situ reprogramming protocol in WSNs, all nodes should be aware of the latest version of an application introduced by any node and make an effort to receive it. Furthermore, when a node has recently resumed from sleep mode or has been joined to the network, it should also infer the information and then, proceed to receiving any new applications, if required. Thus, a ubiquitous and reliable inference about application version is appealing and this can be simply and efficiently implemented using RUbIn. The required data structure and the services of the relevant inference module are depicted in Algorithm 1.

As can be seen in this algorithm, the information does not have a private part. The application version is an ordered pair of a version number and a node address $< VerNo, Addr >$. The element $Addr$ is the address of the node introducing a new version ($VerNo$) of an application to the network. When more than one new application are introduced simultaneously to different nodes, all will be labeled by the same version, which is more than the latest known version. To break the tie, when $VerNo$s are identical, the information with the biggest $Addr$ will be inferred in all nodes. The *set* service is in charge of introducing a new version of the application to a node. This service returns a $SendImmediatelyAndFast$ request to the inference module to immediately and more frequently disseminate the new version information. Here, the version information will never expire and thus, the time factor (*check* service) does not influence the inference.

The *aggregate* service processes all receive messages with the aim of inferring the latest version considering speed, cost, and scalability. In other words, when a node hears the same information as it already has, it returns a $BeQuiet$ request in line 24 of Algorithm 1 to eliminate the dissemination in the current period. This brings scalability to the inference independent of the network density. When a new version is inferred, the $SendFast$ request (lines 27 and 34) is returned to request more frequent information dissemination. Also, when a lower version from one of the neighboring nodes is heard, the $SendImmediately$ request (lines 29 and 36) is returned to immediately inform the neighbor

```
1  Type PubAppInfo =< VerNo, Addr >
2  Type AppInfo =< VerNo, Addr >

3  Function init(curInfo:AppInfo): void
4      curInfo.VerNo = 0
5      curInfo.Addr = UNKNOWN
6  end

7  Function set(curInfo:AppInfo , newInfo:AppInfo): DissCmd
8      if newInfo.Addr <> NODEID then
9          return GoOn
10     else
11         curInfo.VerNo = curInfo.VerNo + 1
12         curInfo.Addr = newInfo.Addr
13         return SendImmediatelyAndFast
14     end if
15 end

16 Function check(curInfo:AppInfo): DissCmd
17     return GoOn
18 end

19 Function aggregate(curInfo:AppInfo , newPubInfo:PubAppInfo , senderId:Addr): DissCmd
20     curVerNo = curInfo.VerNo
21     newVerNo = newPubInfo.VerNo
22     if curVerNo == newVerNo then
23         if curInfo.Addr == newPubInfo.Addr then
24             return BeQuiet
25         else if curInfo.Addr < newPubInfo.Addr then
26             curInfo.Addr = newPubInfo.Addr
27             return SendFast
28         else
29             return SendImmediately
30         end if
31     else if curVerNo < newVerNo then
32         curInfo.VerNo = newPubInfo.VerNo
33         curInfo.Addr = newPubInfo.Addr
34         return SendFast
35     else
36         return SendImmediately
37     end if
38 end
```

Algorithm 1: Inference of the latest version of an application using RUbIn.

node of the newer version without any change in the dissemination period. The incorrect use of these return values can significantly decrease the inferring speed or increase its cost.

6.1.2. Finding the shortest path to a sink (routing)
In a WSN, some nodes play the role of a sink to collect information from other nodes. Finding an optimum path to one of these sinks is the problem of routing algorithms. This problem is an inference needed at all nodes and should be quickly updated when a change in the network topology occurs. An optimum path has different definitions. Here, we consider two of them. The first is to find the shortest path to a sink, and the second is to find the shortest robust path (the path through which the intermediate links have an appropriate quality to relay messages) to a sink. To this aim, the *getLinkQuality* service of the link quality estimator component (named *LE* here) is utilized. Each path is specified by its length, next hop, and update time. The public part consists of the length while the private part encapsulates the other two. The required data structure and the services of the relevant inference module are depicted in Algorithm 2.

The sink nodes ($HopCount = 0$) are added and removed using the *set* service. The *check* service investigates the validity of a path in non-sink nodes. A path should be confirmed or updated once in each $MAXVALIDTIME$ seconds. Otherwise, it will be expired. In both *set* and *check* services, if a change in the information occurs, a $SendImmediatelyAndFast$ is returned. Therefore, all the paths will be quickly updated according to the new change.

In the *aggregate* service, only messages of senders whose bidirectional link quality is equal to or more than $LQTHERSHOLD$ are processed. This condition means that each node selects a path in which the link to the first node of the path is qualified. Compliance with this condition will implicitly result in reliable and qualified paths in all of the nodes. This condition is checked in line 29 of Algorithm 2. Removing this condition results in the shortest path to a sink while considering it results in the shortest robust path to a sink in each node. In this service, when a node

```
1  Type PubSRPInfo =< HopCount >
2  Type PriSRPInfo =< NextHopAddr, UpdateTime >
3  Type SRPInfo =< HopCount, NextHopAddr, UpdateTime >

4  Function init(curInfo:SRPInfo): void
5      curInfo.HopCount = UNKNOWN
6      curInfo.NextHopAddr = UNKNOWN
7      curInfo.UpdateTime = NOW
8  end

9  Function set(curInfo:SRPInfo , newInfo:SRPInfo): DissCmd
10     if newInfo.HopCount == 0 and curInfo.HopCount <> 0 then
11         curInfo.HopCount = 0
12         curInfo.NextHopAddr = NODEID
13         curInfo.UpdateTime = NOW
14         return SendImmediatelyAndFast
15     else if newInfo.HopCount <> 0 and curInfo.HopCount == 0 then
16         init(curInfo)
17         return SendImmediatelyAndFast
18     end if
19     return GoOn
20 end

21 Function check(curInfo:SRPInfo): DissCmd
22     if curInfo.HopCount <> 0 and NOW − curInfo.UpdateTime > MAXVALIDTIME then
23         init(curInfo)
24         return SendImmediatelyAndFast
25     end if
26     return GoOn
27 end


28 Function aggregate(curInfo:SRPInfo , newPubInfo:PubSRPInfo , senderId:Addr): DissCmd
29     if LE.getLinkQuality(senderId) < LQTHERSHOD then
30         return GoOn
31     else if curInfo.NextHopAddr == UNKNOWN then
32         if newPubInfo.HopCount <> UNKNOWN then
33             curInfo.HopCount = newPubInfo.HopCount + 1
34             curInfo.NextHopAddr = senderId
35             curInfo.UpdateTime = NOW
36             return SendImmediatelyAndFast
37         else
38             return BeQuiet
39         end if
40     else if curInfo.NextHopAddr == senderId then
41         if newPubInfo.HopCount == UNKNOWN then
42             init(curInfo)
43             return SendFast
44         else if curInfo.HopCount <> newPubInfo.HopCount + 1 then
45             curInfo.UpdateTime = NOW
46             curInfo.HopCount = newPubInfo.HopCount + 1
47             return SendFast
48         end if
49     else
50         if curInfo.HopCount > newPubInfo.HopCount + 1 then
51             curInfo.HopCount = newPubInfo.HopCount + 1
52             curInfo.NextHopAddr = senderId
53             curInfo.UpdateTime = NOW
54             return SendFast
55         else if curInfo.HopCount + 1 < newPubInfo.HopCount then
56             return SendImmediately
57         else if curInfo.HopCount == newPubInfo.HopCount then
58             return BeQuiet
59         end if
60     end if
61     return GoOn
62 end
```

Algorithm 2: Inference of the shortest robust path to a sink using RUbIn.

hears information the same as what it has, it sends the *BeQuiet* request to eliminate the redundant sending in its proximity. Furthermore, when a better path is inferred, by *SendFast* or *SendImmediatelyAndFast* request, the node will inform the other nodes to update accordingly. Also, if a node infers that a path is a better one to one of its neighbors, it informs the neighbor by a *SendImmediately* request. In other situations, the *GoOn* request is returned.

6.2. Studying effectiveness of RUbIn and efficiency of inferences

We implemented RUbIn and two inference samples given in Section 6.1 with NesC on TinyOS. Then, we examined both of these samples with a TinyOS simulator, namely TOSSIM. Also, we examined these samples in a real testbed with MicaZ nodes.

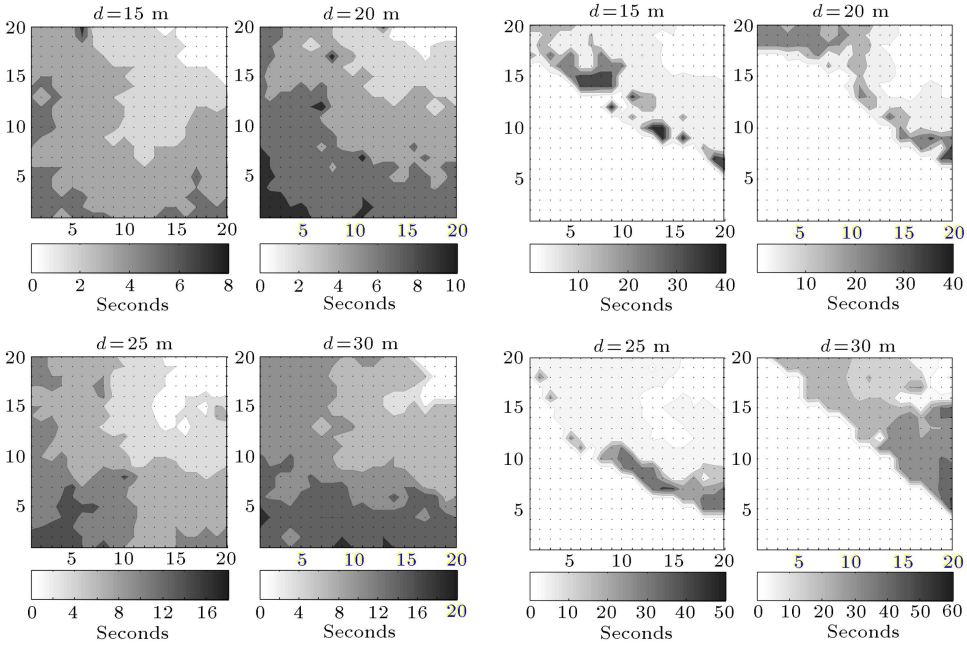
We establish a multi-hop WSN with prerequisite conditions for each experiment. Some initial changes needed to make the network ready for the experiment are made. We should wait for the whole network to become stable after the initial changes. Then, according to the experiment, a new change in information is made somewhere in the network and all the subsequent changes in the network for a few hours are studied. Hereafter, we refer to a change occurring when the whole network is in the stability state as a wake-up change. Also, the inference time of a node refers to the duration between the occurrence times of a wake-up change in network and the consequent inference of information in that node. Inference speed is the inverse of inference time. The maximum inference time of nodes is the inference time of the network. Also, the instability time of network refers to the duration between the occurrence time of a wake-up change in the network ($T = t_\ell$ in the changing node) and the time the whole network becomes stable ($T = t_h$ in all nodes) again. During instability time, at least one node in the period of T, where $t_\ell \leq T < t_h$, exists.

6.2.1. Evaluation by TOSSIM simulator

TOSSIM enables us to run a real application on a virtual network with custom setting. Accordingly, we tested RUbIn on large-scale networks with TOSSIM. Figure 3 demonstrates the reliability of inferences developed in RUbIn. This figure shows the inference times of nodes for inference of the latest version of an application and the shortest robust path to a sink in four different topologies of a $20 * 20$ grid network. These four networks are distinguished by the distances of physically neighboring nodes, which are 15, 20, 25, and 30 meters.

In inference of the latest version of an application (Figure 3(a)) at time zero, the top-right node introduces a new version to the network. After a short time, between 8 and 20 seconds, all 400 nodes infer the latest version of the application. As depicted in this figure, the new information is disseminated through the network from the source node to all other nodes. Nevertheless, there are nodes that infer the new information later than their neighbors due to collisions and topology changes. The repetition characteristics of RUbIn reliably lead to updates in these nodes as well.

In inference of the shortest robust path to a sink



(a) The latest version of an application

(b) The shortest robust path to a sink

Figure 3. Inference times of nodes for two inference examples in four different topologies of a $20 * 20$ grid network.

(Figure 3(b)), the node in the bottom left is a sink. At time zero, the top-right node introduces itself as a new sink. The nodes that are closer to the new sink will update their paths. Change in path information is disseminated from the top-right node to the middle of the network. Path information of the other nodes will not be changed as they are still located closer to the old sink. Like the prior sample, there are nodes that infer the new information later than their neighbors. Nevertheless, the repetition causes all nodes to reliably infer the correct information after a short period.

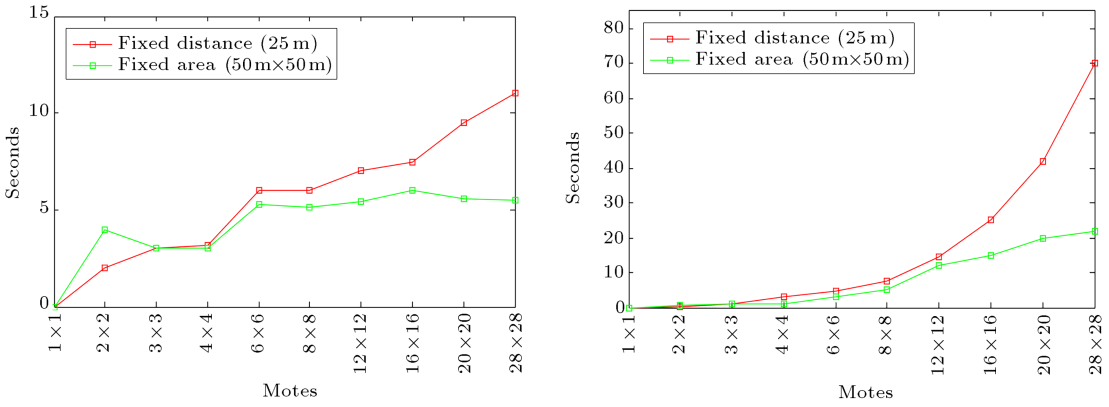
Although the network is a large multi-hop one in these inferences, the speed of inference is an order of seconds (at most one minute).

Figure 4 demonstrates the scalability of inferences developed in RUbIn. We show the inference time for the inference of the latest version of an application (Figure 4(a)) and the shortest robust path to a sink (Figure 4(b)) in 10 $n*n(n \in 1, 2, 3, 4, 6, 8, 12, 16, 20, 28)$ grid networks. In each of these figures, we show the effect of increasing nodes in two different scenarios. In the first scenario (fixed distance), the distance of neighboring nodes is fixed at 25 meters while in the second scenario (fixed area), all nodes are placed in an environment with a fixed area ($50 * 50$ meters).

In the fixed distance scenario, nodes are increased with the aim of increasing the covered area and diameter of the network. An increase in the diameter of the network results in an increase in its inference time. Figure 4 demonstrates that the ratios of the inference times of the two networks for both inference examples are almost equal to the ratios of their network diameters while in some rare cases, they are at most equal to the ratios of their covered areas.

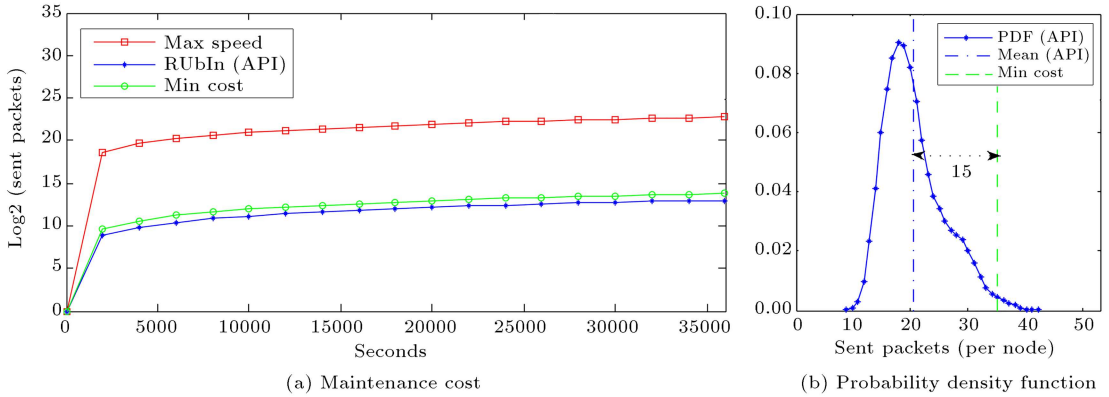
In the fixed area scenario, an increase in the number of nodes leads to an increase in density with a mild change in the diameter of the network. Increasing the density leads to an increase in collisions in the network, which lead to a longer inference time. Nevertheless, the *BeQuiet* mechanism prevents redundant messages and their collisions in dense networks. Accordingly, we can observe in the figure that the increase in the number of nodes slightly affects the inference times in both inference examples.

In Figures 5 and 6, the maintenance costs of both inference examples in the stability state are illustrate.



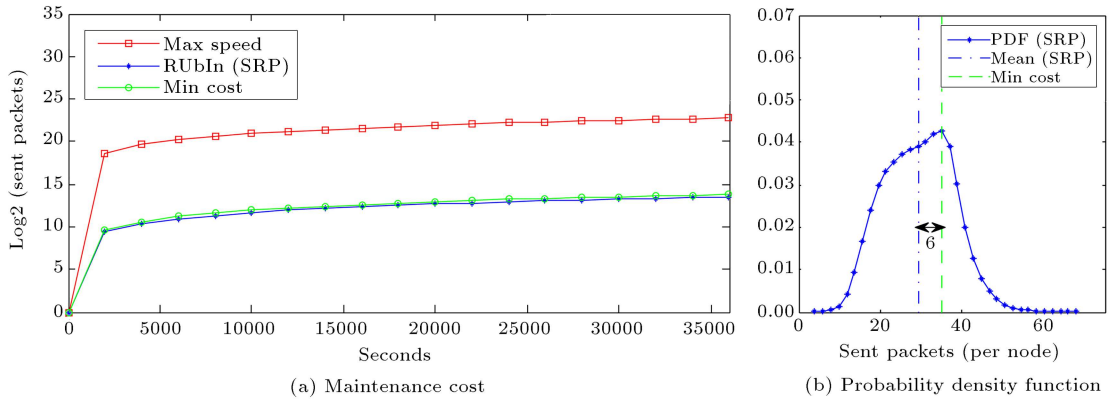
(a) The latest version of an application (b) The shortest robust path to a sink

Figure 4. Network inference time for two inference examples in two different scenarios of fixed distance and fixed area for 10 different networks.



(a) Maintenance cost (b) Probability density function

Figure 5. Maintenance cost of the inference of the latest version of an application in stability state in comparison to max-speed and min-cost scenarios.



(a) Maintenance cost (b) Probability density function

Figure 6. Maintenance cost of the inference of the shortest robust path to a sink in stability state in comparison to max-speed and min-cost scenarios.

We examined such inferences in a grid network of 400 nodes ($20 * 20$) with a distance of 20 meters between physically neighboring nodes for 10 hours after the stability of inferences. The cost was measured in terms of the number of sent messages. Figures 5 and 6 compares the maintenance costs of the two inference examples with a case in which the dissemination period is the constant t_ℓ (2 seconds in our implementation) in order to maximize dissemination speed (max-speed) and a case in which the dissemination period is the constant t_h (1024 seconds in our implementation) in order to minimize maintenance cost (min-cost).

Figure 5(a) shows the maintenance cost for the inference of the latest version of an application (API) using RUbIn in comparison with the max-speed and min-cost scenarios during 10 hours. These costs are drawn in log_2 of sent packets. As a result, the precise use of the *BeQuiet* mechanism declines the maintenance cost of this inference to even less than the cost in min-cost scenario.

Figure 5(b) shows the probability density function of the number of sent packets per node and its average during 10 hours. The average number of sent packets per node during 10 hours is about 21, which is 15 packets less than that in the min-cost scenario.

Figure 6(a) shows the maintenance cost for the inference of the Shortest Robust Path (SRP) to a sink using RUbIn in comparison with the costs of max-speed and min-cost scenarios during 10 hours. The maintenance cost of this inference is again less than the cost of min-cost by the use of *BeQuiet* mechanism. Figure 6(b) shows the probability density function of the number of sent packets per node and its average for 10 hours. The average number of sent packets during 10 hours is about 30, which is 6 packets less than those in the min-cost scenario.

These figures demonstrate the efficiency of inferences based on RUbIn such that using RUbIn and its mechanisms leads to an inference speed equivalent to the speed in max-speed scenario and maintenance cost less than or comparable to the cost in min-cost scenario.

6.2.2. Evaluation in a real testbed

We also evaluated and examined both developed inferences in a real testbed with MicaZ nodes. To this aim, we constructed a multi-hop network (Figure 7) in the laboratory by decreasing the RF power of nodes. We assumed $c = 2$, $t_h = 1024$ seconds, and $t_\ell = 2$ seconds.

Figure 8(a) show the inference times of nodes for the inference of the latest version of an application after introducing a new version by the top-right node at time zero. Both networks are grid networks consisting of 25 MicaZ nodes with two different distances of 23 and 30 cm between physically neighboring nodes. This figure demonstrates how reliably a new information is propagated in the network. Also, Figure 8(b) show the inference times of nodes for the inference of the shortest robust path to a sink after the top-right node introduces itself as a new sink. In this scenario, we suppose that all nodes have already recognized the bottom-left node as their old sink. Two grid networks consisting of 25 MicaZ nodes with different distances of 15 and 27 cm between physically neighboring nodes are

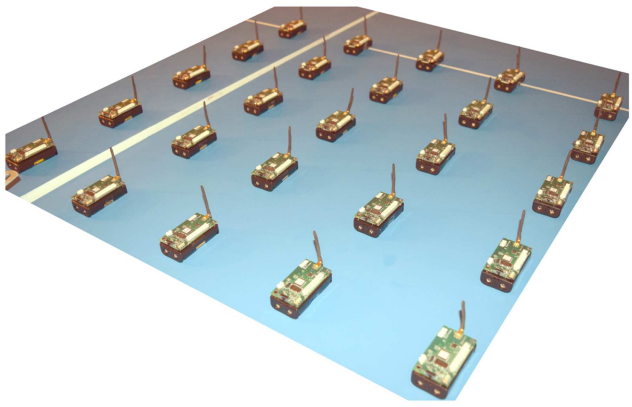
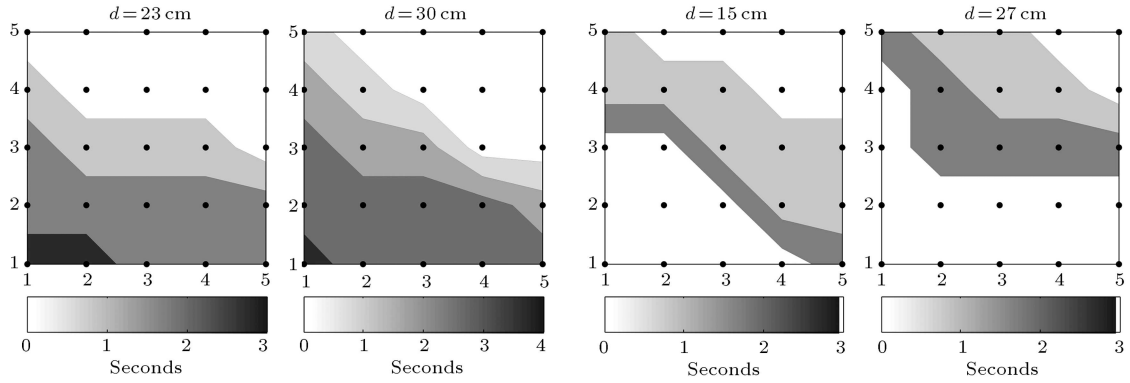
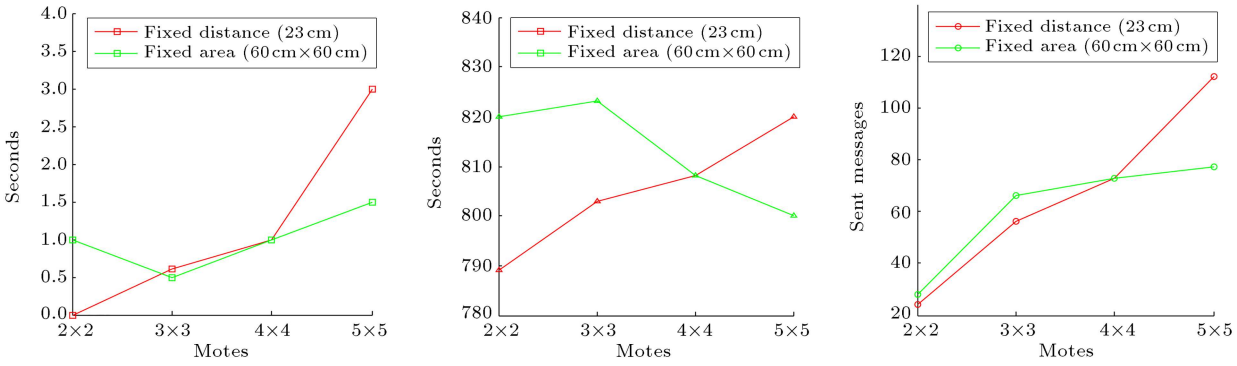


Figure 7. A multi-hop network consisting of MicaZ nodes with decreased RF power.



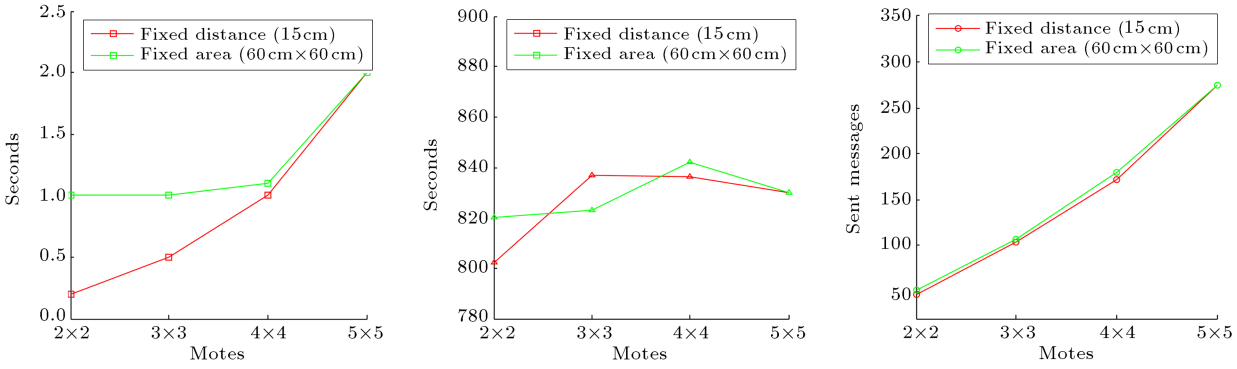
(a) The latest version of an application (b) The shortest robust path to a sink

Figure 8. Inference times for the two examples in a grid network consisting of 25 MicaZ nodes with different physical distances between neighboring nodes.



(a) The inference time (b) The instability duration (c) Cost in instability state

Figure 9. Results for the inference of the latest version of an application in two scenarios of fixed distance and fixed area.



(a) The inference time (b) The instability duration (c) Cost in instability state

Figure 10. Results for inference of the shortest robust path to a sink in two scenarios of fixed distance and fixed area.

used as a testbed. Inference about the new path begins from areas around the top-right node and spreads to the middle of the network. Inference time in all of these scenarios is between 3 and 4 seconds. In both of these examples, the repetition causes all nodes to reliably infer the correct information after a short period of time.

Figures 9 and 10 illustrate the scalability of these inferences in the two scenarios of fixed distance and fixed area.

Figures 9(a) and 10(a) show that when the number of nodes increases, the inference time in the fixed distance scenario increases such that the ratio of the inference times of the two networks is approximately an order of the ratio of their covered areas, while in the fixed area scenario, the inference time is approximately constant.

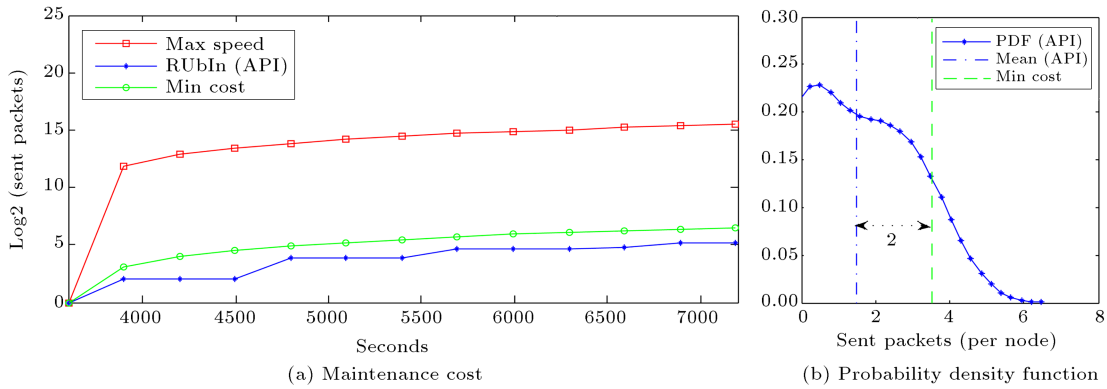
Also, in Figures 9(b) and 10(b), for both the fixed distance and fixed area scenarios, it is evident

that the instability time of network is between 800 and 900 seconds. After a change, the dissemination period is reset to 2 seconds and then, doubled each time. Finally, after 9 periods, it becomes 1024 seconds. Hence, it takes 510 $(2+4+8+16+32+64+128+256)$ seconds to reach the beginning of the period with a duration of 512 seconds. From the middle to the end of this period, nodes can send information and then, the next period is set. Therefore, when the period duration is 512 seconds, a message may be sent between seconds 256 and 512 of this period and then, for the next period, the duration of 1024 (stability state period) seconds is considered. In other words, the instability state lasts about 766 to 1024 seconds after the last change and we observe this in the empirical experiments depicted in Figures 9(b) and 10(b). These figures demonstrate that the duration of instability state is about 14 minutes after a significant change. To ensure a fast and reliable inference, even in a high dynamic topology, this duration should be met.

Figures 9(c) and 10(c) depict the sum of periodic and sporadic sent messages during instability time. Depending on the type of inference, the number of sent messages in this interval is various. In the inference of the latest version of an application, during instability time, the *BeQuiet* service decreases the the number of sent messages to less than 9 (number of periods takes T to reach 1024 seconds from 2) messages per node. This issue is more severe in a scenario of increasing the number of nodes in a fixed area. Nevertheless, in the inference of the shortest robust path to a sink, because of numerous path changes and gossiping period resetting, the number of sent messages to infer a correct path in each node increases slightly up to more than 9 messages per node. Thus, the instability state cost is a few messages per node in most inference algorithms.

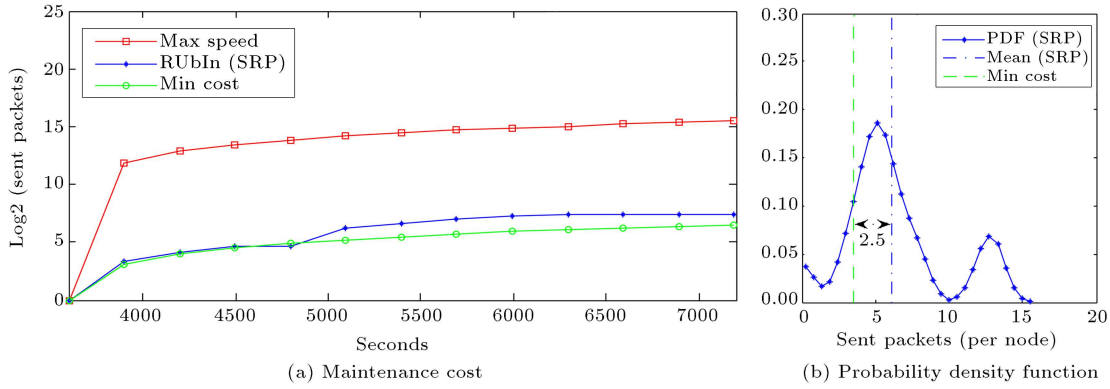
We measured the maintenance costs of the two inference examples in the stability state. To this aim, we traced 2 hours of sent messages after a change at time zero and then, considered the second hours as the stability state. The number of sent packets, the probability density function of the number of sent packets per node, and its average in stability state are depicted in Figures 11 and 12. In these figures, the maintenance costs of the two inference examples are compared with those of min-cost and max-speed scenarios.

In Figure 11(a), for inference of the latest version



(a) Maintenance cost (b) Probability density function

Figure 11. Maintenance cost of the inference of the latest version of an application in stability state in comparison with the max-speed and min-cost scenarios.



(a) Maintenance cost (b) Probability density function

Figure 12. Maintenance cost of the inference of the shortest robust path to a sink in stability state in comparison with max-speed and min-cost scenarios.

of an application, it is evident that the cost in the stability state is always less than that in the min-cost scenario. In other words, after one hour of being in the stability state, based on Figure 11(b), each node has on average of 2 sent messages less than those in the min-cost scenario. Therefore, in long-time execution of this inference, despite that its inference speed is equal to that in the max-speed scenario, the cost is less than that in the min-cost scenario. Indeed, the *BeQuiet* service brings such efficiency to this inference algorithm.

In Figure 12(a), for the inference of the shortest robust path to a sink, it is again evident that the cost in the stability state is approximately equal to the cost of the min-cost scenario, i.e., on average, 2.5 more sent messages (based on Figure 12(b)). Therefore, in long-time execution of this inference, despite that the in this application, inference speed is equal to that in the max-speed scenario, the cost is slightly more than that in the min-cost scenario.

Both inference examples evaluated by the TOSSIM simulator and the testbed of MicaZ nodes reveal the efficiency of our framework in developing inferences in terms of speed and cost while preserving scalability.

7. Conclusion and future work

In this paper, we proposed the RUbIn framework as an extendable middleware for the development of reliable and ubiquitous inferences in WSNs. We described the design of this framework and demonstrated that the RUbIn approach and its supporting mechanisms brought effectiveness to this framework. In other words, we showed that by using this framework, reliable inferences could be simply developed independent of the nodes density and the coverage area. After a significant change anywhere in the network, information at all nodes could be quickly updated. Furthermore, we demonstrated that the mechanisms of RUbIn provided efficiency of inferences so that despite the high inference speed, the cost for each node was about a few messages sent per hour.

RUbIn framework provided a completely distributed approach to solving the inference problems. As a result, our framework facilitated the development of networked smart systems by reducing their design and implementation costs when a ubiquitous inference needed to be intelligent. Therefore, as future work, we will use RUbIn to develop IoT applications when local smartness is appealing in line of the new computing paradigm named "Fog computing" [23].

References

1. Rawat, P., Singh, K.D., Chaouchi, H., and Bonnin, J. "Wireless sensor networks: a survey on recent developments and potential synergies", *The Journal of Supercomputing*, **68**(1), pp. 1-48 (2014).
2. Huang, C. "Study on cloud-dust based intelligent maximum performance analysis system for power generation with solar energy", *Scientia Iranica. Transactions B, Mechanical Engineering*, **22**(6), pp. 2170-2177 (2015).
3. Stojkoska, B.L.R. and Trivodaliev, K.V. "A review of internet of things for smart home: Challenges and solutions", *Journal of Cleaner Production*, **140**, pp. 1454-1464 (2017).
4. Sobral, J.V.V., Rodrigues, J.J.P.C., Rabelo, R.A.L., Filho, J.C.L., Sousa, N., Araujo, H.S., and Filho, R.H. "A framework for enhancing the performance of internet of things applications based on RFID and wsns", *J. of Network and Computer Applications*, **107**, pp. 56-68 (2018).
5. Ndiaye, M., Hancke, G.P., and Abu-Mahfouz, A.M. "Software defined networking for improved wireless sensor network management: A survey", *Sensors*, **17**(5), p. 1031 (2017).
6. Sahni, Y., Cao, J., and Liu, X. "Midshm: A middleware for wsn-based shm application using service-oriented architecture", *Future Generation Computer Systems*, **80**, pp. 263-274 (2017).
7. Portocarrero, J.M.T., Delicato, F.C., Pires, P.F., Costa, B., Li, W., Si, W., and Zomaya, A.Y. "RAMSES: A new reference architecture for self-adaptive middleware in wireless sensor networks", *Ad Hoc Networks*, **55**, pp. 3-27 (2017).
8. Al-Jaroodi, J. and Mohamed, N. "Service-oriented middleware: A survey", *Journal of Network and Computer Applications*, **35**(1), pp. 211-220 (2012).
9. Fortino, G., Galzarano, S., Gravina, R., and Li, W. "A framework for collaborative computing and multi-sensor data fusion in body sensor networks", *Information Fusion*, **22**, pp. 50-70 (2015).
10. Zhu, C., Leung, V.C., Yang, L.T., and Shu, L. "Collaborative location-based sleep scheduling for wireless sensor networks integratedwith mobile cloud computing", *IEEE Transactions on Computers*, **64**(7), pp. 1844-1856 (2015).
11. Srinivasan, S., Duttagupta, S., Kulkarni, P., and Ramamritham, K. "A survey of sensory data boundary estimation, covering and tracking techniques using collaborating sensors", *Pervasive and Mobile Computing*, **8**(3), pp. 358-375 (2012).
12. Plata-Chaves, J., Bertrand, A., Moonen, M., Theodoridis, S., and Zoubir, A.M. "Heterogeneous and multitask wireless sensor networks-algorithms, applications, and challenges", *J. Sel. Topics Signal Processing*, **11**(3), pp. 450-465 (2017).
13. Xiao, K., Wang, R., Fu, T., Li, J., and Deng, P. "Divide-and-conquer architecture based collaborative sensing for target monitoring in wireless sensor networks", *Information Fusion*, **36**, pp. 162-171 (2017).

14. Gharib, M., Yousefi'Zadeh, H., and Movaghar, A. "A survey of key pre-distribution and overlay routing in unstructured wireless networks", *Scientia Iranica. Transactions D, Computer Science & Engineering, Electrical*, **23**(6), pp. 2831-2844 (2016).

15. Levis, P. and Culler, D.E. "Maté: a tiny virtual machine for sensor networks", *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, San Jose, California, USA, pp. 85-95 (2002)

16. Saginbekov, S. and Jhumka, A. "Efficient code dissemination in wireless sensor networks", *Future Generation Computer Systems*, **39**, pp. 111-119 (2014).

17. Byun, H. and So, J. "Node scheduling control inspired by epidemic theory for data dissemination in wireless sensoractuator networks with delay constraints", *IEEE Trans. Wireless Communications*, **15**(3), pp. 1794-1807 (2016).

18. Hoefler, T., Barak, A., Shiloh, A., and Drezner, Z. "Corrected gossip algorithms for fast reliable broadcast on unreliable systems", *2017 IEEE International Parallel and Distributed Processing Symposium*, Orlando, FL, USA, pp. 357-366 (2017).

19. Gnawali, O., Fonseca, R., Jamieson, K., Kazandjieva, M.A., Moss, D., and Levis, P. "Ctp: An efficient, robust, and reliable collection tree protocol for wireless sensor networks", *TOSN*, **10**(1), pp. 16:1-16:49 (2013).

20. Chen, J., Díaz, M., Rubio, B., and Troya, J.M. "Ps-quasar: A publish/subscribe qos aware middleware for wireless sensor and actor networks", *Journal of Systems and Software*, **86**(6), pp. 1650-1662 (2013).

21. Fonseca, R., Gnawali, O., Jamieson, K., and Levis, P. "Four-bit wireless link estimation", *Sixth Workshop on Hot Topics in Networks (HotNets)*, Atlanta, Georgia, USA (Nov. 2007).

22. Levis, P., Patel, N., Culler, D.E., and Shenker, S. "Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks", *1st Conference on Symposium on Networked Systems Design and Implementation (NSDI'04)*, Berkeley, CA, USA, pp. 15-28 (2004).

23. Chang, C., Srirama, S.N., and Buyya, R. "Indie fog: An efficient fog-computing infrastructure for the internet of things", *IEEE Computer*, **50**(9), pp. 92-98 (2017).

Biographies

Abolhassan Shamsaie is a PhD candidate at Sharif University of Technology, Tehran, Iran. His research interests include software architecture, wireless networks and energy-constrained systems, cyber-physical systems, middleware for distributed systems, and internet of things.

Jafar Habibi is an Associate Professor in Computer Engineering Department at Sharif University of Technology. His research interests include software engineering, software architecture and evolution, simulation and performance evaluation, and embedded and distributed systems. He received his PhD in Computer Science from the University of Manchester in 1998.

Erfan Abdi received his MSc in Computer Science from ETH Zurich in 2017. His research interest is distributed computing, design and analysis of algorithms and protocols for wireless sensor networks, and middleware for distributed systems.

Fatemeh Ghassemi received her PhD in Software Engineering from Sharif University of Technology in 2011, and in Computer Science from Vrije Universiteit of Amsterdam in 2018. She has been an Assistant Professor at University of Tehran since 2012, supervising the Formal Methods Laboratory. Her research interest includes formal methods in software engineering, protocol verification, model checking, process algebra, software testing, and wireless systems.