RUbIn: A Framework for Reliable and Ubiquitous Inference in WSNs

Abolhassan Shamsaie\textsuperscript{a,∗}, Jafar Habibi\textsuperscript{a}, Erfan Abdi\textsuperscript{a}, Fatemeh Ghassemi\textsuperscript{b}

\textsuperscript{a}Department of Computer Engineering, Sharif University of Technology, Tehran, Iran
\textsuperscript{b}School of Electrical and Computer Engineering, College of Engineering, University of Tehran, Tehran, Iran

Abstract

Development of 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 for such ubiquitous inferences. RUbIn brings the reliability and ubiquity for inferences using the redundancy characteristic of the gossiping protocols. With RUbIn, the implementation of such inferences and the control of their speed and cost is abstracted by providing developers with a proposed middleware and some dissemination control services.

We develop a prototype implementation of the RUbIn framework and a few inference examples on TinyOS. For evaluation, we utilize both the TOSSIM simulator and a testbed of MicaZ motes in various densities and different number of nodes. Results of the evaluations demonstrate that in all nodes, the inferring time after a change is about a few seconds and the cost of maintenance in stability state is about a few sends per hour.

Keywords: Internet of things, Wireless sensor network, Ubiquitous inference, Framework, Gossiping protocol.

1. Introduction

Sensor motes are small smart devices that integrate the potential of computing, communication and sensing systems into a compact element. These potentials bring the ability of intelligence to WSNs 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, 3, 4, 5]. The difficulty of perceiving the constraints of the nodes’ resources and the complexities

\textsuperscript{∗}Corresponding author: Tel +98 21 66166244, Mobile +98 912 3355324

Email addresses: shamsaie@ce.sharif.edu (Abolhassan Shamsaie), jhabibi@sharif.edu (Jafar Habibi),
eabdi@ce.sharif.edu (Erfan Abdi), fghassemi@ut.ac.ir (Fatemeh Ghassemi)
brought by these constraints to application development should not be a barrier for the development of WSN/IoT applications. Simplifying the application development with 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 middlewares are increasing. These frameworks or middlewares infold the constraints and complexities of WSNs and provide a convenient abstraction for programmers [6, 7, 8].

In intelligence WSNs, nodes collaborate to exchange the information needed to achieve the required inference or smartness [9, 10, 11, 12, 13]. The efficiency, correctness or smartness of many protocols or applications of WSNs rely on an 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 constraints of 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 for inference problems are neglected in WSN. The similar characteristics of inference algorithms and resource constraints of WSNs motivated us to propound a framework as a generic approach for 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 with the inference-specific logic. The framework abstracts the inference algorithms away from the propagation protocol (gossiping) by providing some standard services which programmer can exploit them 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 statement while 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 sorts of ubiquitous, active and reliable inferences can be found within the 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 middlewares.
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 finding the overlay neighboring nodes being physically neighboring nodes having a shared key, finding the overlay path and finally the formation of 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 are actively inferring the last version of a code such that if a node obtains a code with a newer version, after a while, all nodes will obtain it. There are other protocols for the dissemination of code in WSNs [16, 17, 18]. These protocols use a gossiping protocol to reliably disseminate a meta-data of a new code to all the nodes and make them aware of the new code. In these protocols if no change happens for a while, then the period of gossiping will increase to reduce maintenance cost; otherwise, the period is reset to its lowest value to increase the dissemination speed and so 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 like inferring an average of surrounding temperature, an approximation of local density, or the shortest path to a sink, each node may infer a different value in comparison to other nodes. Consequently, many inference algorithms which can be developed in RUbIn are more complicated than just 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, the 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. Using a gossiping protocol and a link quality estimator result in an efficient, robust and reliable routing protocol in WSNs even with a high number of topology changes. The design of the RUbIn architecture is inspired by this protocol to take advantage of its characteristics.

In [20] a middleware which simplifies application development in WSNs using the publish/subscribe model, is proposed. Behind this middleware, there 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, a 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 or identical to the 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 and so on. All the nodes, regardless of their running application, have an active inference on a deterministic set of information \( C = \{I_1, I_2, \ldots, I_{|C|}\} \) 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 value $v_0$, and then is updated for various reasons such as changes in the number of active nodes, variation in the quality of links, updates in 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 other node information. Therefore, all nodes should trace these changes and consider them in their inference algorithms. Also, they should inform the other nodes of any changes to 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 an extra overhead, so a mechanism is needed to moderate this overhead. In general, the inference framework must 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, an update of information should start and accomplish 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: reliability, inference speed, and scalability should be achieved considering these constraints.

Furthermore, in many cases in inference algorithms, the link quality 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 information received from nodes having more stable links prevents temporary inferred values and their side effects (successive inferences). Therefore not only the accuracy of the information is increased, but its maintenance cost is also decreased.

### 4. Requirements analysis

In this section we analyze the requirements of an 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 are 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 to contribute 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, logic of the application, etc. Thus inference framework should provide an interface for the application to contribute their value in the inference.
2. **Time:** Occasionally, the elapsed time from a given point in the past, like the last update time or the last 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 information validity. In some inference problems where 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 messages received from neighboring nodes. Nodes should inform each other about their information and also merge the received ones with their own information. Changes in receiving values from neighboring nodes are the results of changes in the topology or the ones caused by the three main factors in the neighboring nodes.

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

**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. In this inference, the two factors of application and message play a role. The application factor initiates a change in the local memory of a node while the message factor results in an update in memory of all the other nodes. Here the time factor has no role.

**Consensus on a quantity:** One of the research topics in WSNs is the consensus problem. For example, when all nodes have their own value of a quantity, and they all want to infer the maximum of these values, a consensus to find out the maximum of this 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 last update. The message factor informs all the node of any change in consensus result at any node.

**Finding a robust route to a sink:** In a WSN, one or more nodes play the role of sink to collect information. Nodes in interactions 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 only allows 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 the inferring information and the required meta-data. This data structure consists of multiple data fields which can be divided into a private and public part. Both of these parts may be changed over time, but only the public part is sent to the neighboring nodes.

Because of the high dynamics of the network, ensuring the reliability and robustness of WSNs applications is possible only through repetition. In other words, a message of 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 at 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 the cost refers to the number of sent messages. The speed and cost of gossiping protocols are inversely related to the gossiping period; 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 the efficiency, unless this increase brings an excessive increment to its 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 for this problem is to provide a mechanism such that the nodes that eavesdrop the messages in their proximity can eliminate the send if it is wasteful. Consequently, whenever a proximity density is increased, the probability of such eliminations is also increased. Therefore, this mechanism can restrict the sends to a few numbers in each proximity.

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

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 regarding 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 a public and a private part. Both of these parts participate in an inference algorithm and are accessible by the applications, but only the public part is available for the lower layers, and consequently
to other nodes. Thus, the length of the public part is restricted to a few bytes 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 for 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 (RUbln data unit) consists of a header and footer in addition to a gossiping message of the upper layer. Both of this header and footer are used for the 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 sent messages and another 8-bit field consisting of a 4-bit as the number of entries in the footer and a 4-bit 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 \( L_{\text{pckt}} \) as the maximum length of a packet and \( L_{\text{pub}} \) as the length of the public part of information, then we have \( L_{\text{pub}} \leq L_{\text{pckt}} - 3 \). Therefore \( N = \left\lfloor \frac{L_{\text{pckt}} - L_{\text{pub}} - 3}{3} \right\rfloor \). When \( N > 0 \) the estimation algorithm can piggyback its information on gossiping messages, thus at least for one case we should have \( L_{\text{pub}} \leq L_{\text{pckt}} - 6 \) to attain a precise estimation of the links.

5.2. RUbln framework architecture

![Component diagram of the RUbln framework.](image-url)
In Figure 2, the architecture of the RUbIn framework and its layers are depicted by a component diagram. In this diagram, components are divided into the skeleton and extended components. The skeleton components are components which are already 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 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 link quality estimator component piggybacks estimation information on messages of the dissemination engine. Therefore, the link quality estimation is obtained during gossiping of other information, and in case there is no information for inference or no space for piggybacking, the quality estimation of the links would be unknown. To solve this problem, when the send rate of the links information is lower than a threshold we send a few distinct beacons to achieve a precise estimation of the links. Following services to access to the quality of input, output, and bidirectional links are now available by the link quality interface to be used 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 that its period $T$ increases exponentially so that it is initialized to a minimal value $t_\ell$ (about a few seconds) and becomes automatically $c$ times after each period. The period increases at most $k$ times and finally reaches to a maximum value $t_h$, where $t_h = c^k t_\ell$ (about a few hours). Afterward, 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 of $T$, the timer fire at a random time $t_{\text{tick}}$, where $t_{\text{tick}} \in \left[\frac{T}{2}, T\right]$ instead of the end of the period. At time $t_{\text{tick}}$ 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 $t_\ell$ 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, an exponential timer component with one virtual timer per information. 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
to each of the information control units with the aim of managing the dissemination period. In other words, we summarize all modification to the default gossiping trend of each information by the four following services.

1. **SendFast()**: This service increases the dissemination speed of the information. To this aim, it resets the exponential timer to \( t_{\ell} \) which increases the dissemination speed and consequently the inference speed of 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 occasionally an immediate send before or after \( t_{\text{tick}} \) may be needed. This service does not influence on 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 its cost.

3. **BeQuiet()**: This service eliminates dissemination of the information in the current period. In fact, if this service is invoked before \( t_{\text{tick}} \), the corresponding exponential timer will not fire in the current period. This service does not influence on 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 an immediate send is performed and then the dissemination period is reset to \( t_{\ell} \) with the aim to increase 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 be totally about a fraction of one second.

Some information control units are in interaction with the dissemination engine. A programmer instantiates these units as much 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 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 the information can be modified. The information is initialized in this module and then is 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): DistCmd
- **check** (curInfo:InfoType): DistCmd
- **aggregate** (curInfo:InfoType, newPubInfo:PubInfoType, senderId:Addr): DistCmd

The information control unit initializes the information by calling the **init** service, then 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 on 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 an enumeration of the five following 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 this 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 two first 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. The zero value means 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 service.

6. Evaluation

We evaluate the RUbiN framework from two aspects of effectiveness and efficiency. To this aim, with 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 code of two inference problems of ones discussed in Section 4 using our framework. We define the required data structures and the services of relevant inference modules. 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. Being aware 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 been joined to the network, it also should infer this information and then proceed to receive any new application if required. Thus a ubiquitous and reliable inference on application version is appealing, and this can be simply and efficiently implemented using RUbIn. Required data structure and the services of the relevant inference module are depicted in Algorithm 1.

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

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 \(<\text{VerNo}, \text{Addr}>\). The element \text{Addr} is the address of the node introducing a new version (\text{VerNo}) of an application to the network. When more than one new application is introduced simultaneously in different nodes, all will be labeled by the same version equal to one more than the latest known version. To break the tie, when \text{VerNo}s are identical, the information with the biggest \text{Addr} would be inferred in all nodes. The \text{set} service is in charge of introducing a new version of the application in a node. This service returns a \text{SendImmediatelyAndFast} request to the inference module to immediately and more frequently disseminate the new version information. Here, a version information will never expire and so the time factor (\text{check} service) does not influence the inference.
The aggregate service processes all received 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 itself, it returns a BeQuiet request in line 24 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 (line 29 and 36) is returned to immediately inform the neighbor 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 a 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 (a path so that 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. 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 would be quickly updated according to the new change.

In the aggregate service, only messages of senders whose bidirectional link quality is equal or more than LQOTHERS Hold 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. Removing this condition results in the shortest path to a sink while considering it results in a shortest robust path to a sink in each node. In this service when a node hears information the same as itself, it sends the BeQuiet request to eliminate the redundant send 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 through itself is a better path for 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.
Algorithm 2: Inference of the shortest robust path to a sink using RUbIn

We establish a multi-hop WSN with prerequisite conditions for each experiment. We make some initial changes needed to make the network ready for the experiment and also wait for the whole network to become stable after the initial changes. Then, according to the experiment, we make a new change on information somewhere in the network and study all subsequent changes in the network for a few hours. Hereafter we refer to a change occurred 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 time of a wake-up change in network and 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 \( T = t_f \) 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 with period of \( T \) where \( t_f \leq T < t_h \) exists.

6.2.1. Evaluation by TOSSIM simulator

TOSSIM enables us to run a real application on a virtual network with custom settings. Ac-
cordingly, we have tested RUBIn on large-scale networks with TOSSIM. Figure 3 demonstrates
the reliability of inferences developed in RUBIn. This figure shows the inference time 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 \( \times \) 20 grid network. These 4 networks are distinguished by the distance
of physically neighboring nodes which is 15, 20, 25 and 30 meters.

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
dissemnates 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. Nonetheless, the repetition characteristics of RU\textsubscript{b}In reliably results in updates in these nodes as well.

In inference of the shortest robust path to a sink (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 would update their paths. Change in path information disseminates 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 nearer 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 was a large multi-hop network in these inferences, the speed of inference was an order of seconds (at most one minute).

![Figure 4: Network inference time for two inference examples in two different scenarios of fixed distance and fixed area for 10 different networks.](image)

![Figure 5: Maintenance cost of the inference of the latest version of an application in stability state in compare to Max-speed and Min-cost scenario.](image)
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 ten $n \times n$ ($n \in \{1, 2, 3, 4, 6, 8, 12, 16, 20, 28\}$) 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 to increment the covered area and diameter of the network. An increment in the diameter of the network results in an increase in the inference time of the network. This figure demonstrates that the ratio of the inference time of two networks in both inference examples is almost equal to the ratio of their network diameters while in some rare cases it is at most equal to the ratio of their covered area.

In the fixed area scenario, an increment 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 cause to a longer inference time. Nevertheless, the BeQuiet mechanism prevents redundant messages and their collisions in dense networks. Accordingly, we can observe in this figure that the increment in the number of nodes slightly affects the inference time in both inference examples.

In Figure 5 and 6 we illustrate the maintenance cost of both inference examples in the stability state. To this aim, we examine 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 is measured in term of the number of sent messages. This figures compares the maintenance costs of the two inference examples with a case that the dissemination period is the constant $t_l$ (2 seconds in our implementation) in order to maximize dissemination speed (Max-speed) and a case that 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 compare to 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 has declined the maintenance cost of this inference to even less than the 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. As a result, the average number of sent packets per node during 10 hours is about 21 packets which is 15 packets less than the Min-Cost scenario.

Figure 6-a shows the maintenance cost for the inference of the shortest robust path to a sink (SRP) using RUbIn in compare to Max-Speed and Min-Cost scenarios during 10 hours. As a result, the maintenance cost of this inference is also less than the Min-cost due to 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. As a result, the average number of sent packets during 10 hours is about 30 packets which is 6 packets less than 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 Max-speed scenario and maintenance cost less than or comparable to the 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 assume $c = 2$, $t_h = 1024$ seconds and $t_f = 2$ seconds as well.

![Figure 7: A multi-hop network consisting of MicaZ nodes with decreased RF power.](image)

Figure 8-a show the inference time 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 time of nodes for the inference of the shortest robust path to a sink after that 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 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.
(a) The latest version of an application
(b) The shortest robust path to a sink

Figure 8: Inference time for the inference of two examples in a grid network consisting of 25 MicaZ nodes with different physical distance between neighboring nodes.

Figure 9: Results for the inference of the latest version of an application in 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 time of two networks is approximately an order of the ratio of their covered area, while in the fixed area scenario the inference time is approximately constant.

Also, in Figures 9-b and 10-b for the both 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 doubles each time, and finally after 9 periods 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, a message may be sent between seconds 256 and 512 of this period and then for the next period 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 have observed this in 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.

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

Figures 9-c and 10-c depicts the sum of periodic and sporadic sent messages during instability time. As a result, depending on the type of inference, the number of sent messages in this interval is different. In the inference of the latest version of an application, during instability time, the BeQuiet service decreases the number of sent messages to less than 9 (number of periods takes T to reach from 2 to 1024 seconds) 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 resettings, 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.

Figure 11: Maintenance cost of the inference of the latest version of an application in stability state in compare to Max-speed and Min-cost scenario.
We also measured the maintenance cost of these two inference examples in the stability state. For this aim, we traced 2 hour of sent messages after a change at time zero and then considered the second hours as the stability state. The number of sent packets and 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, we compare the maintenance cost of these two inference examples with Min-cost and Max-speed scenarios.

In Figure 11-a for inference of the latest version of an application, it is evident that the cost of this inference in the stability state is always less than 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 2 sent messages less than the Min-cost scenario. Therefore, in a long time execution of this inference, despite that its inference speed is equal to the Max-speed scenario, the cost is less than the Min-cost scenario. Indeed, the BeQuiet service brings such efficiency to this inference algorithm.

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

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

7. Conclusion and future work

In this paper, we proposed the RUbln 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 RUbln approach and its supporting mechanisms bring 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 so that after
a significant change anywhere in the network, information at all nodes will be quickly updated. Furthermore, we demonstrated that the mechanisms of RUbIn bring efficiency to inferences so that in spite of the high inference speed, their cost for each node is about a few sends per hour.

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

References


Biographies

**Abolhassan Shamsaie** is a Ph.D. candidate in Sharif University of Technology. His research interests include software architecture, wireless networks and energy-constrained systems, cyber-physical systems, middleware for distributed systems, internet of things.

**Jafar Habibi** is an associate professor at computer engineering department in Sharif University of Technology. His research interests include software engineering, software architecture and evolution, simulation and performance evaluation, embedded and distributed systems. He received his Ph.D. in computer science from the University of Manchester in 1998.

**Erfan Abdi** has received his M.Sc. in computer science at ETH Zurich in 2017. His research interest is distributed computing, design and analysis of algorithms and protocols for wireless sensor networks, middleware for distributed systems.

**Fatemeh Ghassemi** has received her Ph.D in Software Engineering from Sharif university of technology in 2011, and in Computer Science from Vrije Universiteit of Amsterdam in 2018. She is 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.