Least-Latency Routing over Time-Dependent Wireless Sensor Networks

Shouwen Lai, *Member, IEEE,* and Binoy Ravindran, *Senior Member, IEEE*

Abstract—We consider the problem of least-latency end-to-end routing over adaptively duty-cycled wireless sensor networks. Such networks exhibit a time-dependent feature, where the link cost and transmission latency from one node to other nodes vary constantly in different discrete time moments. We model the problem as the time-dependent Bellman-Ford problem. We show that such networks satisfy the FIFO property, which makes the time-dependent Bellman-Ford problem solvable in polynomial-time. Using the β-synchronizer, we propose a fast distributed algorithm to construct all-to-one shortest paths with polynomial message complexity and time complexity. The algorithm determines the shortest paths for all discrete times in a single execution, in contrast with multiple executions needed by previous solutions. We further propose an efficient distributed algorithm for time-dependent shortest path maintenance. The proposed algorithm is loop-free with low message complexity and low space complexity of $O(maxdeg)$, where $maxdeg$ is the maximum degree for all nodes. We discuss a sub-optimal implementation of our proposed algorithms that reduces their memory requirement. The performance of our algorithms are experimentally evaluated under diverse network configurations. The results reveal that our algorithms are more efficient than previous solutions in terms of message cost and space cost.

Index Terms—Time-dependent, shortest path, wireless sensor networks, routing, routing maintenance, least-latency.

1 Introduction

Multihop data routing over wireless sensor networks (WSNs) has attracted extensive attention in the recent years. Since there is no infrastructure in sensor networks, the routing problem is different from the one in traditional wired networks or the Internet. Some routing protocols [2], [3] over WSNs presented in the literature are extended from the related approaches over wired/wireless ad-hoc networks. They usually find a path with the minimum hop count to the destination, which is based on the assumption that the link cost (or one-hop transmission latency) is relatively static for all wired/wireless links. However, for duty-cycled WSNs [4]–[6], that assumption may not always be valid.

Duty-cycled WSNs include sleep-wakeup mechanisms, which can violate the assumption of static link costs. Currently, many MAC protocols support WSNs operating with low duty cycle, e.g., B-MAC [5], X-MAC [6]. In such protocols, sensor nodes operate in low power listening (or LPL) mode. In the LPL mode, a node periodically switches between the active and sleep states. The time duration of an active state and an immediately following sleep state is called the LPL checking interval, which can be identical, or can be adaptively varied for different nodes, referred to as ALPL [7]. The duty-cycled mechanism has been shown to achieve excellent idle energy savings, scalability, and easiness in implementation. However, they suffer from time-varying neighbor discovery latencies (the time between data arrival and discovery of the adjacent receiver), which is also pointed out by Ye *et.al.* [8]. As shown in Figure 1, the neighbor discovery latency between two neighbors is varying with different departure times. Even with synchronized duty-cycling, the neighbor discovery latency is varying at different time moments due to adaptive duty cycle setting as shown in Figure 1.

To formally define the problem, we first define the *link cost* as the time delay between data dispatching time, which is the earliest time when a sender wakes up for data transmission, and the data arrival time at the receiver. The link cost is time-varying in adaptively duty-cycled WSNs due to varying neighbor discovery latencies, even though the physical propagation condition does not change with time. The dispatching time is the time moment when the data is ready for transmission at the sender side. Thus, this raises a non-trivial problem: with time-varying link costs, how to find optimal paths with least nodes-to-sink latency for all nodes at all discrete dispatching time moments?

A similar problem has been modeled in previous works as the time-dependent shortest path problem (or TDSP) [9], [10] in the field of traffic networks [11], time-dependent graphs [12], and GPS navigation [13]. The general time-dependent shortest path problem is at least NP-Hard, since it may be used to solve a variety of NP-Hard optimization problems such as the knapsack problem. However, depending on how one defines the problem, it may not be in NP, since its output is not polynomially bounded. Moreover, there are even continuous-time instances of the TDSP problem in which shortest paths consist of an infinite sequence of arcs, as shown by Orda and Rom [14]. In this paper, we study a special case where the networks are known as FIFO networks, in which commodities travel along links in a First-In-First-Out manner. Under the FIFO condition, the time-

- *S. Lai is with Qualcomm, Inc, 5775 Morehouse Drive, San Diego, CA, 92121. E-mail: shouwenl@qualcomm.com*
- *B. Ravindran is with the Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, 24061. E-mail: binoy@vt.edu*

- *The preliminary result was presented at IEEE INFOCOM 2010 [1].*

dependent shortest path problem is solvable in polynomial time.

The TDSP problem has also been studied with a distributed approach. The only previous distributed solution [10] computes the shortest paths for a specific departure time in each execution. If the whole time period has M discrete intervals (M is ∞ for infinite time intervals), we have to execute the algorithm in [10] M times, which is inefficient in terms of message complexity and time complexity, given the limited power and radio resource in WSNs. Therefore, the first motivation of our work is to design a fast distributed algorithm for the problem, which can efficiently enumerate all optimal paths with least end-to-sink latency for infinite time intervals.

The second motivation of our work is to propose an algorithm which can dynamically and distributively maintain time-dependent least-latency paths. In WSNs, a node may update its duty-cycle configuration (e.g., based on its residual energy), or join or leave the network, thereby changing the network topology. In such situations, the duty-cycle updating node or the joining/leaving node may change the cost of all the links with its neighbors, which means that a single node update can cause multiple link updates. Previous efforts on this problem [15], [16] are efficient in handling single link updates. Applying such solutions for multiple link updates would imply that multiple distributed updates execute concurrently for a single node update, which is not efficient in terms of message cost and memory cost for resource-limited WSNs.

The third motivation is to address practical implementation issues. Our work requires schedule-awareness in neighborhood. One scenario for such requirement is that all nodes have none information from each other after initial deployment in field. In this scenario, we would like to get schedule-awareness without global time synchronization. Local synchronization and light information exchange is desirable to get duty-cycling information and time difference from neighbors for a sensor node.

One candidate to achieve this is by neighbor discovery protocol, just similar as the link layer neighbor discovery protocol in the Internet. Nodes without schedules of neighbors would stay awake and broadcast neighbor discovery message periodically (i.e., once every multiple predefined time slots) and neighbors can response back their schedule information, which is a kind of local level synchronization. Once reaching a stable stage, any node would get its neighbor's schedule and the time difference from its neighbors, and switch back to duty-cycling state. The update of nodes schedule can be undergoing in background either probatively or reactively.

Because our work is not limited to any specified MAC-protocol, we discuss different methods to achieve schedule awareness over several underlying mechanisms, such as B-MAC, S-MAC, and quorum-based wakeup scheduling in Section 7.1.

Regarding to another practical implementation issues, we also need to understand how to simplify the vector presentation so that only smaller vector sizes are required, given the limited memory resource of sensor nodes. We present a sub-optimal implementation, which achieves a trade-off between latency and memory usage. Finally, we discuss the complexities of our algorithms in some special scenarios, like static link costs and multiple sink nodes.

In this paper, we first propose a distributed algorithm to compute the time-dependent paths with least-latency for all nodes in a duty-cycled WSN. The algorithm has low message and space complexities. The algorithm is based on the observation that the time-varying link cost function is periodic, and hence by derivation, the time-varying distance function for each node is also periodic. We show that the link cost function satisfies the FIFO property [9]. Therefore, the time-dependent shortest path problem is *not* an NP-hard problem, and thus is solvable in polynomial time. We also propose distributed algorithms for maintaining the shortest paths. The proposed algorithms re-compute the routing paths based on previous path information.

The message complexity of our algorithms is $O(\delta^2)$ per node update, where δ is the number of nodes that change either the distance or the parents in their shortest paths to the sink as a consequence of the corresponding nodes' update. The algorithms' space complexity is $O(maxdeg)$. Finally, we propose a sub-optimal implementation, which requires vectors with smaller sizes to represent link cost functions and the distance function.

The contributions of the paper are as follows: 1) We model adaptively duty-cycled WSNs as time-dependent networks. We show that such networks satisfy the FIFO condition and the triangular path condition 2) We present distributed algorithms for finding the time-dependent shortest paths to the sink node for all nodes. When compared to the previous solution [10], our algorithms find the shortest paths in a single execution for infinite time intervals 3) We present distributed shortest path maintenance algorithms with low message complexity and space complexity 4) We propose sub-optimal implementation with vector compression.

To the best of our knowledge, we are not aware of any other efforts that consider duty-cycled WSNs as time-dependent networks and solve the problem of finding or updating the shortest paths with efficient message and space costs.

The rest of the paper is organized as follows: We survey past and related works in Section 2, and outline our assumptions and define the problem in Section 3. We formally model the link cost function and the distance function in Section 4. The algorithms for route construction and route maintenance are described in Sections 5 and 6, respectively. We discuss practical implementation issues in Section 7. Simulation results are reported in Section 8. We conclude in Section 9.

2 Related Work

We summarize the literature on LPL scheduling and the time-dependent shortest path problem as follows.

LPL/ALPL in WSNs. LPL means that a node only wakes up and listens the channel state for a short time

period. Examples include B-MAC [5], which is a CSMA-based technique that utilizes low power listening and an extended preamble to achieve low power communication. In B-MAC, nodes have an awake and a sleep period, and an independent sleep schedule. If a node wishes to transmit, it precedes the data packet with a preamble that is slightly longer than the sleep period of the receiver. During the awake period, a node samples the medium, and if a preamble is detected, it remains awake to receive the data. With the extended preamble, a sender is assured that at some point during the preamble, the receiver will wake up, detect the preamble, and remain awake in order to receive the data. The designers of B-MAC show that B-MAC surpasses existing protocols in terms of throughput, latency, and for most cases, energy consumption. While B-MAC performs quite well, it suffers from the overhearing problem, and the long preamble dominates the energy usage.

To overcome some of B-MAC's disadvantages, XMAC [6] and DPS-MAC [17] were proposed. In X-MAC or DPS-MAC, short preamble was proposed to replace the long preamble in B-MAC. Also, receiver information is embedded in the short preamble to avoid the overhearing problem. The main disadvantage of B-MAC, X-MAC, and DPS-MAC is that it is difficult to reconfigure the protocols after deployment, thus lacking in flexibility. X-MAC [6] and DPS-MAC [17] are compatible with LPL mechanisms. However, they do not explicitly support adaptive duty cycling, where nodes choose their duty cycle depending on their residual energy.

Jurdak *et. al.* [7] and Vigorito *et. al.* [4] present adaptive low power listening (ALPL) mode based on nodes' residual energy. These works provide the application spaces for our work. In ALPL, since nodes have heterogenous duty cycle setting, it is more difficult for neighbor discovery since a node cannot differentiate whether a neighbor is sleeping or failing when it does not receive feed-back from the neighbor. ALPL also incurs time-dependent link-cost and end-to-end latency as illustrated in Section 4. Recently, B-MAC [5] was also extended to support the ALPL mode in TinyOS.

Delay-efficient routing over adaptively duty-cycled WSNs. Over adaptively duty-cycled WSNs, routing becomes more difficult due to two reasons: intermittent connection between two neighbor nodes and changes in the transmission latency at different times. Some works have studied the delay-efficient routing problem over adaptively duty-cycled WSNs in recent years.

Lu [18] *et al*. proposed two methods to solve routing over intermittently connected WSNs due to duty cycling. One is by an on-demand approach that uses probe messages to determine the least latency route. The other one is a proactive method, where all least latency routes at different departure times are computed at the beginning. The first method does not work well for frequent data deliveries. The second method is a centralized approach, and is not flexible for distributed construction. Our algorithms also follow the proactive approach, but are distributed.

Yu [19] *et al*. considered the problem with a different perspective. They studied how to consume a minimum amount of energy while satisfying an end-to-end delay bound specified by the application. In [20], they studied how to guarantee the end-to-end latency by adjusting duty cycling in individual nodes. There are also some other works that have studied the energy-delay trade-off for duty-cycled WSNs, such as [21] and [22]. These efforts are similar to our work, but there is a fundamental difference: we study the least routing latency, given the duty-cycle or energy configuration on each node.

Time-Dependent Shortest Path Problem. This problem was first proposed by Cooke and Halsey [9]. It has been well studied in the field of traffic networks [11], time-dependent graphs [12], and GPS navigation [13]. Previous solutions for this problem mostly work offline using a centralized approach [12]. Although these solutions can provide inspirations, they cannot be applied to WSNs where the global network topology is not known by a centralized node, given the large-scale size of a WSN.

For the distributed time-dependent shortest path problem, the only previous work [10] computes the shortest paths for a specific departure time in each execution, which is not time-efficient. If the whole time period has M discrete intervals (M is ∞ for infinite time intervals), we have to execute the algorithm in [10] M times, which is inefficient in terms of message complexity and time complexity. For multiple executions, the algorithm in [10] suffers from high message cost, which is undesirable for resource-limited WSNs.

The work in [10] discusses two policies for the time-dependent shortest path problem: waiting and non-waiting. Waiting does not mean waiting in the buffer, but means waiting for some time after the data has been delivered (i.e., the receiver is awake). Non-waiting means that a sender will immediately send the data once the receiver is awake. We do not consider the waiting policy in our work, since the end-to-end latency does not benefit from waiting.

Dynamic Shortest-Path maintenance. Many works [15], [16], [23] exist for handling link decreases and increases, and node deletions and insertions in static networks. In [24], an algorithm is given for computing all-pairs shortest paths, which requires $O(n^2)$ messages when the network size is n. In [25], an efficient incremental solution has been proposed for the distributed all-pairs shortest paths problem, requiring $O(nlog(nW))$ amortized number of messages over a sequence of edge insertions and edge weight decreases. Here, W is the largest positive integer edge weight. In [26], Awerbuch *et al.* propose a general technique that allows to update the all-pairs shortest paths in a distributed network in $O(n)$ amortized number of messages and $O(n)$ time, by using $O(n^2)$ space per node.

In [23], Ramarao and Venkatesan give a solution for updating all-pairs shortest paths that requires $O(n^3)$ messages, $O(n^3)$ time, and $O(n)$ space. They also show that, in the worst case, the problem of updating shortest

paths is as difficult as computing shortest paths. They suggest two possible directions toward devising efficient fully dynamic algorithms for updating all-pairs shortest paths: 1) explore the trade-off between the message, time and space complexity for each kind of dynamic change 2) devise algorithms that are efficient in different complexity models (with respect to worst case and amortized analysis).

However, the algorithms in [15], [23] need $O(n)$ space at each node, which is impractical for sensor nodes with limited memory capacity. In addition, none of the previous works are efficient for shortest path maintenance in time-dependent networks.

β-Synchronizer [27]. As described in [27], the *synchronizer* is a methodology for designing efficient distributed algorithms in asynchronous networks. Researchers have used synchronizers to reduce message complexity of some asynchronous algorithms, such as Bellman-Ford. A synchronizer works as follows. A synchronizer generates sequences of "clock-pulses" at each node of a network. At each node, a new pulse is generated only after it receives all the messages which were sent to that node by its neighbors at the previous pulse. Thus, a synchronizer runs in a phase-by-phase manner.

A β-synchronizer is a special type of synchronizer, which has an initialization phase, in which a leader s is chosen in the network and a spanning tree rooted at s is constructed (e.g., by a Breadth-First-Search). After the execution of one phase, the leader s will eventually learn that all the nodes in the network are "safe". At that time, it broadcasts a message along the spanning tree, notifying all the nodes that they may generate a new pulse. The communication pattern for receiving all acknowledgments is just like convergecast. Therefore, with a β-synchronizer, whenever a node learns that it is safe and all its descendants in the tree are safe, it sends an acknowledgment to its parent. In each phase, a β-synchronizer incurs low message complexity, which is $O(|V|)$ [27], where $|V|$ is the network size, but at a higher time cost.

For a distributed shortest path algorithm armed with a β-synchronizer, if the number of phases are limited, the total message complexity will be bounded.

3 Assumptions and Problem Definition

We model a WSN as a directed graph $G = (V, E, C)$, with $|V|$ nodes and $|E|$ links. $C = \{\tau_{i,j}(t) | (i,j) \in E\}$ is a set of time-dependent link delays, i.e., $\tau_{i,j}(t)$ is a strictly positive function of time defined for $[0, \infty)$, describing the delay of a message over link (i,j) at time t. Each node n_i only knows the identity of the nodes in its neighbor set, defined as N_i.

We assume that time axis are arranged as consecutive numbered time slots. We denote the duration of one time slot for node n_i as T_i. It is possible that $T_i \neq T_j$ (ALPL) for two nodes n_i and n_j. The time expansion of each node n_i is modeled as discrete and infinite, where $\mathfrak{T}_i = \{t_i^0, t_i^1, t_i^2, \cdots, t_i^M\}$, M is $+\infty$, and $t_i^k - t_i^{k-1} = T_i$. We use the terms checking interval and time slot interchangeably.

The wakeup schedule depends on the underlying MAC protocol. We first assume that a node can be operated in the LPL mode: a node wakes up at the beginning of a time slot to check the channel state. If there is no activity, the node goes back to sleep; otherwise, it stays awake. Then, we relax the assumption and discuss how our work can be applied to other wakeup schedules, such as quorum schedules [28].

We consider the non-waiting policy (i.e., the sender immediately delivers data once the receiver is awake) at each node, since the node-to-sink delay will not benefit from waiting. Thus, once the data arrives at an intermediate node, the node will attempt to dispatch the data immediately. Dispatching time represents the earliest time when a sendor is awake for data transmission. Thus, dispatching times are not the same as the data departure times, as the data may still be buffered in the sender's memory. For simplicity in modeling and design, a node dispatches the received data at $t_i^k \in \mathfrak{T}_i$.

A nonnegative travel time $\tau_{i,j}(t_i^k)$ is associated with each link (i,j) with the following meaning: if t_i^k is the data dispatching time from node n_i along the link (i,j), then $t_i^k + \tau_{i,j}(t_i^k)$ is the data arrival time at node n_j.

The general problem of determining the shortest paths with the least latency in time-dependent WSNs can be defined as follows: Find the least-time paths from all nodes to the sink node n_s corresponding to the minimum achievable delay d_i, $\forall n_i \in V$ and $\forall t_i^k \in \mathfrak{T}_i$, where:

$$d_i(t_i^k) = \min_{n_j \in N_i} \{\tau_{i,j}(t_i^k) + d_j\big(t_i^k + \tau_{i,j}(t_i^k)\big)\} \qquad (1)$$

Equation 1 is an extension of Bellman's equations [29] for the time-dependent network and is referred to as TD-Bellman's equation hereafter.

We also assume that a message arrives correctly in finite time from a sender to a receiver, which can be achieved by any reliable MAC-layer transmission mechanism.

We do not assume that the entire network is time-synchronized, i.e., all nodes are not equipped with GPS devices or are not operated by global time-synchronization protocols [30]. However, due to the proactive routing nature in our proposed algorithms, we need to know the link costs at the beginning of route construction. Therefore, we assume awareness of wakeup schedules of the neighborhood for each node. Such schedule awareness can be achieved by neighbor discovery protocols such as periodic neighbor detection mechanisms. We further assume that all nodes have the same time frequency, and their clocks drift at relatively slow speeds.

4 Modeling Adaptively Duty-cycled WSNs

In this section, we model adaptively duty-cycled WSNs as time-dependent networks. The algorithms we propose are based on Equation 1. Basically, t_i^k is infinitely discrete. Note that sensor nodes have limited memory and exchanging messages is expensive. Thus, in order to implement Equation 1 in practice, we must make $\tau_{i,j}(t_i^k)$

and $d_j(t_i^k + \tau_{i,j}(t_i^k))$ (where $k \in [0, \infty]$) finite, so that the time-dependent link cost and distance can be represented by vectors.

We will now show that the link cost function is periodic and establish that the time-varying distance function is also periodic. Having done so, we will show how TD-Bellman's Equation can be implemented by vector representations of link costs and distances.

4.1 Link Cost Function

Without loss of generality, suppose there are two adjacent nodes n_i and n_j, where n_i is the sender and n_j is the receiver.

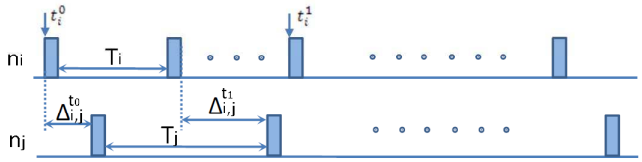

Fig. 1. Varying neighbor discovery latency in heterogenous LPL mode

Suppose at time t_i^0, the neighbor discovery latency is $\Delta_{i,j}^{t_i^0}$. Then at time $t_i^k = t_i^0 + k * T_i$, the neighbor discovery latency can be expressed as:

$$\Delta_{i,j}^{t_i^k} = T_j - (k * T_i - \Delta_{i,j}^{t_i^0}) mod\ T_j \tag{2}$$

In an actual deployment, for example with the X-MAC protocol, we can measure $\Delta_{i,j}^{t_i^0}$ in the following way: at the beginning of the time slot which starts at t_i^0, node n_i sends out a preamble which contains the node ID of n_j. n_j immediately feeds-back an ACK containing the value of T_j once it receives the preamble. After receiving the ACK by n_i, the one-hop round trip delay from t_i^0 to the time at which the ACK is received is set to $\Delta_{i,j}^{t_i^0}$. Once we have measured $\Delta_{i,j}^{t_i^0}$, $\Delta_{i,j}^{t_i^k}$ ($k \geq 0$) can be computed by Equation 2.

For data transmission with fixed length data packets, we define the data propagation time as τ_{data}. Now, for a directed link (i, j), we can set the link cost function as, for $\forall t_i^k \in \mathfrak{T}_i$,

$$\tau_{i,j}(t_i^k) = T_j - (k * T_i - \Delta_{i,j}^{t_i^0}) mod\ T_j + \tau_{data} \tag{3}$$

If τ_{data} is relatively small when compared with T_i and T_j, we can set $\tau_{data} = 0$. This is especially true for some WSN applications with small information reports, such as target tracking and environment monitoring.

Theorem 1: For every link (i, j), the time-varying link cost function is periodic. The minimum period for the function regarding k is,

$$P(\tau_{i,j}) = \frac{LCM(T_i, T_j)}{T_i} \tag{4}$$

where $\tau_{i,j}(t_i^k) = \tau_{i,j}(t_i^{k+P(\tau_{i,j})})$ ($k \geq 0$) and LCM is the least common multiple.

4.2 Distance to Sink

We refer to the node-to-sink delay as *distance*, for compatible representation with that in the static Bellman-Ford algorithm [29].

Consider node n_i and its neighbor n_j, where $T_i \neq T_j$. For dispatching time t_i^0 at n_i, let us suppose that the data arriving time at n_j is $t_j^{j_0}$. Then, for the dispatching time t_i^k at n_i, with the same time frequency for the two nodes, the corresponding time instant at n_j is $t_j^{j_0} - \Delta_{i,j}^{t_i^0} + k * T_i$. Hence, based on the neighbor discovery mechanism (e.g., B-MAC [5], X-MAC [6]), n_j will be discovered by n_i at the time instant $t_j^{j_0} + \lceil \frac{k*T_i - \Delta_{i,j}^{t_i^0}}{T_j} \rceil * T_j$, with respect to n_j's time clock. Therefore, the function of distance from n_i to n_s through the path through n_j is:

$$d_i(t_i^k) = \tau_{i,j}(t_i^k) + d_j(t_j^{k' + j_0}) \tag{5}$$

for $\forall t_i^k \in \mathfrak{T}_i$, where $k' = \lceil (k * T_i - \Delta_{i,j}^{t_i^0})/T_j \rceil$ for $t_j^{k'} \in \mathfrak{T}_j$, and j_0 is the data arriving time slot at n_j with respect to dispatching time t_i^0.

Theorem 2: For a path $n_i \rightarrow n_{i-1} \cdots \rightarrow n_1 \rightarrow n_s$, the distance function $d_i(t_i^k)$,$\forall t_i^k \in \mathfrak{T}_i$, is a periodic function, where the minimum period for the function regarding k is:

$$P(d_i) = \frac{LCM(T_0, T_1, \cdots, T_i)}{T_i} \tag{6}$$

for $d_i(t_i^k) = d_i(t_i^{k+P(d_i)})$, where T_0, T_1, $\cdots$, T_i are durations of the LPL checking intervals of nodes n_s, n_1, $\cdots$, n_i, respectively.

Proof: The proof is by induction. For $i = 1$, $d_1(t_1^k) = \tau_{1,s}(t_1^k) + d_s(t_1^{k'})$. Since $d_s \equiv 0$, $d_1(t) = \tau_{1,s}(t_1^k)$. According to Theorem 1, $d_1(t_1^k)$ is periodic and its period is $LCM(T_0, T_1)/T_1$. Assume that the claim is true for node n_{i-1}.

Now, for node n_i, $d_i(t_i^k) = \tau_{i,i-1}(t_i^k) + d_{i-1}(t_{i-1}^{k'+j_0})$, where $k' = \lceil (k * T_i - \Delta_{i,j}^{t_i^0})/T_j \rceil$ and j_0 is the data arrival time slot at n_{i-1} with respect to t_i^k, based on Equation 5. Let us define $f_1 = \tau_{i,i-1}(t_i^k)$ and $f_2 = d_{i-1}(t_{i-1}^{k'+j_0})$. The minimum period for function f_1 is $P(f_1) = LCM(T_i, T_{i-1})/T_i$. Let $\prod = LCM(T_0, T_1, \cdots, T_{i-1})$. Based on the induction step, for distance function $d_{i-1}(t_{i-1}^k)$, the period is $\prod / T_{i-1}$. The period for function f_2 is $P(f_2) = \prod / gcd(\prod, T_i)$. Since $gcd(\prod, T_i) = \frac{\prod * T_i}{LCM(\prod, T_i)}$, we have $P(f_2) = \frac{LCM(\prod, T_i)}{T_i}$.

Therefore, the minimum period for $d_i(t_i^k)$ is: $P(d_i) = LCM[P(f_1), P(f_2)] = LCM(T_0, T_1, \cdots, T_i)/T_i$. □

Given a WSN with different LPL checking intervals, the period for the distance function of any node is bounded by $LCM(T_0, T_1, \cdots, T_n)/\min\{T_i\}$, from Equation 6.

In practical implementations, it is recommended that $LCM(T_0, T_1, \cdots, T_n)/\min\{T_i\}$ is not arbitrarily large. Thus, our mechanism for finding the shortest paths at the routing layer should be based on cross-layer design. For example, {100ms, 200ms, 500ms, 1000ms} is a good configuration set, where there is a bounded period $LCM(100, 200, 500, 1000)/min\{100, 200, 500, 1000\} = 10$ for the distance function. It means that for any node, its

distance to the sink will repeat at most every 10 checking intervals.

4.3 Implementation via Vectors

We implement the discrete, periodic, and infinite link cost functions and the distance functions as vectors, and implement the TD-Bellman's equation (Section 3) by vector operations. Our goal is to use vectors with limited sizes in order to implement our algorithm with limited memory and same message size over the air, although the time axis is infinite.

We implement the link cost function $\tau_{i,j}(t_i^k)$ ($t_i^k \in \mathfrak{T}_i$) with a vector $\vec{\tau_{i,j}}$, where $|\vec{\tau_{i,j}}| = LCM(T_i, T_j)/T_i$ and $\tau_{i,j}[k]$ represents a set of numbers as follows:

$$\tau_{i,j}[k] = \{\tau_{i,j}(t_i^k), \tau_{i,j}(t_i^{k+|\vec{\tau_{i,j}}|}), \tau_{i,j}(t_i^{k+2*|\vec{\tau_{i,j}}|}), \cdots\} \quad (7)$$

For the node n_i, its distance function $d_i(t^k)$ ($t_i^k \in \mathfrak{T}_i$) can be implemented by $\vec{d_i}$, where $|\vec{d_i}| = P(d_i(t^k))$. $d_i[k]$ represents a set of numbers as follows:

$$d_i[k] = \{d_i(t_i^k), d_i(t_i^{k+|\vec{d_i}|}), d_i(t_i^{k+2*|\vec{d_i}|}), \cdots\} \quad (8)$$

However, there are two difficulties for the implementation of the TD-Bellman's equation by vector operations.

The first one is vector mapping. To implement Equation 5, even if we know $\vec{\tau_{i,j}}$ and $\vec{d_j}$, we cannot add up the two vectors directly. We define a new vector $\vec{d_j'}$ as:

$$d_j'[k] = d_j[(k' + j_0) \ mod \ |\vec{d_j'}|] \quad (9)$$

where $k' = \lceil (k*T_i - \Delta_{i,j}^{t_i^0})/T_j \rceil$ and j_0 is the corresponding time slot for $\tau_{i,j}[0]$ at n_j (i.e., $t_i^0 + \Delta_{i,j}^{t_i^0} = t_j^{j_0}$).

Only after mapping $d_j[k]$ to $d_j'[k]$, we can add $\tau_{i,j}[k]$ to $d_j'[k]$. By vector mapping, the size of the new vector $\vec{d_j'}$ is:

$$|\vec{d_j'}| = \frac{|\vec{d_j}| * T_j}{gcd(|\vec{d_j}| * T_j, T_i)} \quad (10)$$

The second difficulty comes from the various sizes of vectors for link cost and distance. Suppose $d_i\vec{(j)}$ is the vector representing the distance of n_i from a path through n_j in discrete time intervals. To implement Equation 5, if $\vec{\tau_{i,j}}$ and $\vec{d_j'}$ have the same size, we can directly add them up for computing $d_i\vec{(j)}$. Otherwise, we need to expand the two vectors to be of the same size, which means expanding $\vec{\tau_{i,j}}$ by $LCM(|\vec{\tau_{i,j}}|, |\vec{d_j'}|)/|\vec{\tau_{i,j}}|$ times and $\vec{d_j'}$ by $LCM(|\vec{\tau_{i,j}}|, |\vec{d_j'}|)/|\vec{d_j'}|$ times. After the expansion, we can directly add up the expanded vectors. We call such an operation, *vector expansion*.

Vector mapping and expansion do not change the value of the discrete functions $\tau_{i,j}$ and d_j. They just change the representation of values of the two discrete functions. The vector expansion is valid since the time expansion is infinite.

We define the following functions for implementation:

Definition 1: For a vector $\vec{v}$:

- $ror(\vec{v}, ofs)$: output $\vec{v'}$ where $\forall k \in [0..|\vec{v}|-1]$, $v'[k] = v[(k + ofs) \ mod \ |\vec{v}|]$;
- $rol(\vec{v}, ofs)$: output $\vec{v'}$ where $\forall k \in [0..|\vec{v}|-1]$, $v'[k] = v[(\vec{v}| + k - ofs) \ mod \ |\vec{v}|]$;
- $map(\vec{v}, a, b, \Delta, ofs)$: output v' where $\forall k \in [0..|\vec{v}|-1]$, $v'[k] = v[(\lceil \frac{a*k-\Delta}{b} \rceil + ofs) \ mod \ |\vec{v}|]$;
- $exp(\vec{v}, e) = \vec{v}||\vec{v}...||\vec{v}$ (e $times$) (|| presents catenating operation)

We utilize these functions to implement the TD-Bellman's equation. Suppose that n_i has received the distance vector $\vec{d_j}$ of node n_j. Suppose $\tau_{i,j}[0]$ is associated with the time slot $l_{i,j}^0$ at node n_i, and the data arriving time slot for $\tau_{i,j}[0]$ is l_j^0 at n_j. Then, $d_i\vec{(j)}$, the distance vector of n_i to the sink from the path through n_j, can be calculated as:

$$\begin{aligned} \vec{d_j}' &= map[ror(\vec{d_j}, l_j^0), T_i, T_j, \tau_{i,j}[0], 0]; \\ d_i\vec{(j)}' &= exp(\vec{\tau_{i,j}}, e_1) + exp(\vec{d_j}', e_2); \\ d_i\vec{(j)} &= rol[d_i\vec{(j)}', l_{i,j}^0] \\ &= vec\_add(\vec{\tau_{i,j}}, \vec{d_j}, l_{i,j}^0, l_j^0) \end{aligned} \quad (11)$$

where $|\vec{d_j}'|$ is defined in Equation 10 and by defining $A = LCM(|\vec{\tau_{i,j}}|, |\vec{d_j'}|)$, $e_1 = A/|\vec{\tau_{i,j}}|$ and $e_2 = A/|\vec{d_j'}|$.

We update $\vec{d_i}$ and the corresponding parent vector $\vec{p_i}$ in the following way. Suppose the original $d_i[0]$ is associated with the time slot 0 at n_i (i.e., $d_i[0] = d_i(t_i^0)$). Now,

$$(\vec{d_i}, \vec{p_i}) = vmin\{exp(d_i, e_1'), exp(d_i\vec{(j)}, , e_2')\} \quad (12)$$

where $B = LCM(|d_i\vec{(j)}|, |\vec{d_i}|)$, $e_1' = B/|\vec{d_i}|$, and $e_2' = B/|d_i\vec{(j)}|$. The function $(\vec{d_i}, \vec{p_i}) = vmin(\vec{v_1}, \vec{v_2})$ compares the corresponding elements in the two vectors $\vec{v_1}$ and $\vec{v_2}$, and copies the smaller element of each pair into the corresponding element in $\vec{d_i}$ and the corresponding vector ID into $\vec{p_i}$.

In addition, we define an operator $\vec{<}$ for comparing two vectors $\vec{v_1}$ and $\vec{v_2}$. Let $C = LCM(|\vec{v_1}|, |\vec{v_2}|)$. If $\forall k \in [0..C-1]$ $exp(\vec{v_1}, C/|\vec{v_1}|)[k] \leq exp(\vec{v_2}, C/|\vec{v_2}|)[k]$, then $\vec{v_1} \vec{<} \vec{v_2}$. For example, $[1,3] \vec{<} [2,3]$.

Example. We now give an example to illustrate the vector implementation. In Figure 2, there are four nodes n_0, n_1, n_2, and n_3. n_0 is the sink. By the measurement method introduced in Section 4.1 and Equation 2, we have $\tau_{1,0} = [155, 5]$, $\tau_{2,0} = [50, 200]$, $\tau_{3,1} = [20, 120, 70]$, and $\tau_{3,2} = [175, 275, 75]$.

By Equation 5, we directly have $\vec{d_1} = [155, 5]$ and $\vec{d_2} = [50, 200]$, which means that the shortest latencies from node n_1 to node n_0 will be repeatedly 155 and 5 every two time slots, starting from t_1^0. Similarly, this applies for node n_2.

Then node n_1 and n_2 send their distance vectors d_1 and d_2 to n_3 by messages. After n_3 receives the messages, by Equation 11, it will compute the distance vector $d_3\vec{(1)}$, which is the latency of routing through node n_1, and the distance vector $d_3\vec{(2)}$, which is the latency of routing through node n_2.

We now have $\vec{d_1'} = [155, 5, 155]$ (the extension of d_1). Thus, $d_3\vec{(1)} = \vec{d_1'} + \tau_{3,1} = [20, 120, 70] + [155, 5, 155] = [175, 125, 225]$. Similarly, $d_2' = [50, 200, 50]$ (the extension of d_1), and $d_3\vec{(2)} = d_2' + \tau_{3,2} = [50, 200, 50] + [175, 275, 75] = [225, 475, 125]$.

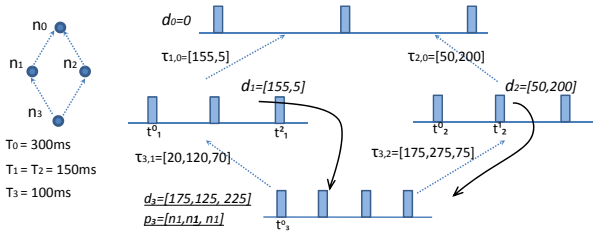

Fig. 2. Example vector implementation

Therefore, $\vec{d_3} = vmin\{[175, 125, 225], [225, 475, 125]\} = [175, 125, 125]$, and correspondingly $\vec{p_3} = [n_1\ n_1\ n_2]$), which means that the shortest distance will be repeatedly 175, 125, and 125 every three time slots, starting from t_3^0. The corresponding parents are repeatedly n_1, n_1, and n_2 every three time slots, starting from t_3^0.

4.4 Properties

The FIFO condition [9] means that a packet which was delivered earlier will always arrive at a direct neighbor earlier. We will prove that an adaptively duty-cycled WSN satisfies the FIFO condition.

Theorem 3: FIFO condition: The link cost function $\tau_{i,j}(t_i^k)$ satisfies the FIFO property, which means, for any $t_i^{k_1} < t_i^{k_2}$,

$$t_i^{k_1} + \tau_{i,j}(t_i^{k_1}) \leq t_i^{k_2} + \tau_{i,j}(t_i^{k_2}) \tag{13}$$

Proof: From Equation 3, we have $t_i^{k_2} + \tau_{i,j}(t_i^{k_2}) - t_i^{k_1} - \tau_{i,j}(t_i^{k_1}) = (k_2 - k_1) * T_i + (k_1 * T_i - \Delta_{i,j}^{t_i^0})\ mod\ T_j - (k_2 * T_i - \Delta_{i,j}^{t_i^0})\ mod\ T_j = (k_2 - k_1) * T_i + [(k_1 - k_2) * T_i]\ mod\ T_j = (k_2 - k_1) * T_i - [(k_2 - k_1) * T_i]\ mod\ T_j \geq 0$. □

We now give an intuitive explanation for Theorem 3. In a static network, $\tau_{i,j}(t_i^{k_2}) - \tau_{i,j}(t_i^{k_1}) = 0$, because the link cost is constant. In adaptively duty cycled WSNs, the link cost $\tau_{i,j}(t)$ captures the time difference between when node n_i wakes up at t and when node n_j wakes up. Since node n_i, which wakes up at $t_i^{k_1}$, can always detect the awake neighbor n_j earlier (or at least at the same time) than the case in which n_i wakes up at $t_i^{k_2}$, we have $t_i^{k_1} + \tau_{i,j}(t_i^{k_1}) \leq t_i^{k_2} + \tau_{i,j}(t_i^{k_2})$.

By Theorem 3, the time-dependent shortest path problem in adaptively duty-cycled WSNs is not NP-hard and is solvable in polynomial-time [9].

Theorem 4: Suppose node n_i has two neighbors n_j and n_k, which are one-hop away from each other. Then, at a time instant t_i, we have the triangular property:

$$\tau_{i,j}(t_i) \leq \tau_{i,k}(t_i) + \tau_{k,j}[t_i + \tau_{i,k}(t_i)] \tag{14}$$

Proof: Suppose at time t_i, the data arriving time slot at n_j is t_j, and the data arriving time at n_k is t_k.

If $t_k \leq t_j$, which means that the data arriving time at n_k is earlier than the data arriving time at n_j, $\tau_{i,k}(t_i) + \tau_{k,j}[t_i + \tau_{i,k}(t_i)] = t_k - t_i + t_j - t_k = t_j - t_i = \tau_{i,j}(t_i)$.

If $t_k > t_j$, which means that the data arriving time at n_k is later than the data arriving time at n_j, we have: $\tau_{i,k}(t_i) + \tau_{k,j}[t_i + \tau_{i,k}(t_i)] = t_k - t_i + t_j^{'} - t_k > t_j - t_i + t_j^{'} - t_k > t_j - t_i > \tau_{i,j}(t_i)$. The theorem follows. □

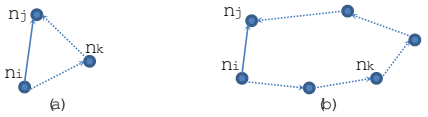

Fig. 3. Triangular path condition: the direct path $n_i \rightarrow n_j$ always achieves the least latency among all paths from n_i to n_j

Theorem 4, as illustrated in Figure 3(a), illustrates that node n_i will always directly arrive at its neighbor n_j without going through other nodes. We now have the following claim:

Lemma 1: Triangular Path Condition: For a node n_i and its neighbor n_j, at any dispatching time, the one-hop path $n_i \rightarrow n_j$ always has the least time delay for data transmission.

Proof: We prove this by induction. Suppose there are multiple nodes along the path from n_i to n_j. We define these nodes as n_k, n_{k+1}, $\cdots$, n_{k+i}. If $i = 0$, based on Theorem 4, Lemma 1 is true.

Now, assume that $i = k$ and Lemma 1 is true. We prove that Lemma 1 is true when $i = k+1$. Suppose the adjacent node to n_i along the path is n_k. Then the direct path $n_k \rightarrow n_j$ has shorter latency than the path along $n_{k+1}, \cdots, n_{k+i}$. Also, based on Theorem 4, $n_i \rightarrow n_j$ has shorter latency than the path ($n_i - > n_k - > n_j$). Therefore, the direct path $n_i \rightarrow n_j$ has shorter latency than the path along n_k, n_{k+1}, $\cdots$, n_{k+i}. The lemma follows. □

An illustration is given in Figure 3(b). Note that the triangular path condition does not exist in static networks.

5 Algorithm for Initial Route Construction

We now present an algorithm for initial time-dependent shortest path route construction in duty-cycled WSNs, where the distances from all nodes to the sink node are initially infinite. The proposed algorithm, referred to as the FTSP algorithm, for Fast Time-Dependent Shortest Path algorithm, is inspired by the work in [10].

As described in Section 4, although the time axis is infinite, the time-varying link costs and distance can be implemented by vectors. Therefore, our algorithm which implements Equation 1 is basically similar to the distributed Bellman-Ford algorithm. The difference is that our algorithm is exchanging vectors (i.e., for time-varying link costs and distances), rather than single values of static link cost and distance.

FTSP is adapted from the distributed Bellman-Ford algorithm and is augmented with a β-synchronizer [27]. We choose a β-synchronizer in order to avoid exponential message complexity, as discussed in Section 2. With a β-synchronizer, the time cost and message cost are bounded and there is no gain in the best case. However, due to the limited resource in WSNs, we believe it is more important to avoid the exponential message complexity of the traditional Bellman-Ford algorithm.

Equipped with the vector implementation, FTSP computes the shortest paths for infinite discrete time intervals

in one execution, in contrast with the solution in [10], which only calculates the shortest paths for a specified discrete time.

Let $|D_m|$ denote the diameter of the longest shortest path for all nodes. We show that the message complexity of FTSP is $O(|D_m||E|)$ and the time complexity is $O(|D_m||V|)$. FTSP does not suffer from exponential message complexity, like that in previous works for the static shortest path problem over asynchronous networks (e.g., [31]).

5.1 Distributed Algorithm Description

There are essentially two steps in our algorithm for constructing routing paths. In the first step, we build a spanning tree. In the second step, we calculate the shortest paths and send back the acknowledgment to the sink for nodes in the network with a layer-by-layer approach. In our algorithm FTSP, we combine the two steps and implement them through iterations. In the first iteration, FTSP computes the shortest paths for nodes in the nearest layer to the sink. In the following iterations, FTSP goes beyond one layer each time, until it reaches the last layer.

We present the data structures and message formats in FTSP:

- $\vec{d_i}$: vector of distance from n_i to n_s, defined in Equation 8; initially all elements are ∞;
- $\vec{\tau_{i,j}}$: link delay from n_i to its neighbor n_j, defined in Equation 7; $\tau_{i,j}[0]$ is obtained by measurement;
- $\vec{p_i}$: vector of parents for n_i in the shortest path $n_i \rightarrow n_s$ for infinite time intervals; initially all elements are node n_i;
- $MSG(ID_{src}, ID_{from}, \vec{d}, updated)$: control message; ID_{src} is the node ID of sink node, or update source node (see Section 6); ID_{from} is the sender's node ID; $\vec{d}$ is the distance vector of the sender; $updated$ indicates whether there is an update in the current iteration, which will be explained later;
- $ACK[j]$: boolean indicating whether a node receives a control message from its neighbor n_j

We assume that n_i knows the duration of the checking interval T_j of all its neighbors $n_j \in N_i$ after measuring link delays. Initially, a directed spanning tree rooted at n_s is built upon the network. We assume that n_i knows its parent st_p_i in the spanning tree. We also assume that FTSP is invoked by higher-level protocols that create "START" impetuses at n_s.

The first iteration of FTSP begins when node n_s receives the "START" impetus. Subsequent ones begin whenever n_s completes an iteration and determines whether another one is necessary by checking whether there is a node whose distance was minimized in the last iteration.

Each iteration begins at node n_s by sending a control message to all its neighbors. When replies from all its neighbors have been received, node n_s concludes that an iteration is completed. Every other node, i.e., n_i ($n_i \neq n_s$), begins an iteration upon receiving a control message from its parent st_p_i in the spanning tree, upon which it sends control messages to all its neighbors except st_p_i. When replies are received from all these neighbors, a control message is sent to the parent, thereby completing the current iteration at node n_i.

The control message from node n_j contains the distance vector $\vec{d_j}$ (known thus far during the previous iterations) between n_j and n_s. When such a message is received at node n_i, node n_i checks whether the new information decreases the value of any element in the current distance vector. It does so by considering the path that goes through n_j, taking into account the most recent information from n_j.

We describe the procedures in Algorithms 1 and 2.

Algorithm 1: Algorithm for sink node n_s in FTSP:

Initialization:
  $\forall n_j \in N_s$, $ACK[j] = 0$, $updated[j] = true$;
On receiving START:
  send MSG(s, s, 0, false) to $\forall n_j \in N_s$;
On receiving MSG(j,$\vec{d_j}$, l_j, bChanged):
  $ACK[j] = 1$; $updated[j] = bChanged$;
  if ($\forall n_j \in N_s, ACK[j] == 1$) **then**
    $ACK[j] = 0$;
    if ($\forall n_j \in N_s$, $updated[j] == false$) **then**
      STOP;
    else
      send MSG(s, s, 0, false) to $\forall n_j \in N_s$;

Algorithm 2: Algorithm for node $n_i (i \neq s)$ in FTSP:

Initialization:
  $st\_p_i = NULL$;
  for $\forall n_j \in N_i$ **do**
    $ACK[j] = 0$; $updated[j] = true$;
    Measure link delay $\Delta^0_{i,j}$ at the beginning of any time slot;
    $l^0_{i,j}$ = the time slot number for measurement;
    l_j = the data arriving time slot number at n_j;
    for $\forall k \in [0..LCM(T_i, T_j)/T_j]$ **do**
      $\tau_{i,j}[k] = T_j - (k * T_i - \Delta^0_{i,j}) mod\ T_j$;

On receiving MSG(*s*, *j*, $\vec{d_j}$, bChanged) from n_j:
  $ACK[j] = 1$; updated[j] = bChanged; $\vec{d_i}^{prev} = \vec{d_i}$;
  if $n_j == st\_p_i$ **then**
    send MSG(s, *i*, $\vec{d_i}$, false) to $\forall n_j \in N_i$ except st_p_i;
  $\vec{d_i}(j)$ = vec_add($\vec{d_i}, \vec{d_j}, l^0_{i,j}, l_j$); /* Equation 11*/
  ($\vec{d_i}, \vec{p_i}$) = vmin($\vec{d_i}, \vec{d_i}(j)$); /* Equation 12*/
  if ($\vec{d_i} \vec{<} \vec{d_i}^{prev}$) **then** updated[j] = true;
  if ($\forall n_j \in N_i, ACK[j] == 1$) **then**
    if ($\exists n_j \in N_i, updated[j] == true$) **then** bUpdated = true;
    else then bUpdated = false;
    send MSG(*s*, *i*, $\vec{d_i}$, bUpdated) to st_p_i;
    $\forall n_j \in N_i, ACK[j] = 0$;

The functions $vec\_add(\cdot)$, $vmin(\cdot)$, and operator $\vec{<}$ are defined in Section 4.3.

5.2 Correctness and Complexity

Let $PATH(i, t^k_i)$ be a path obtained by node n_i, which is starting at time t^k_i and moving along its parent $p_i[k]$. Let $d_i[k]_m$ denote the value of $d_i[k]$ after the m^{th} iteration in FTSP. We have the following properties:

Theorem 5: 1) After termination, $PATH(i, t^k_i)$ is loop-free and concatenated. 2) In the m^{th} ($m \geq 0$) iteration, a node n_i whose shortest path is at most m-hop away from the sink node will be determined, for all discrete time intervals $t^k_i \in \mathfrak{T}_i$.

Proof: For part 1, $\forall t_i^k \in \mathfrak{T}_i$, after termination, suppose $p_i[k]$ is set to the node n_j for the shortest path with respect to n_s. Since n_j is the parent of n_i at the time slot t_i^k, $PATH(i, t_i^k)$ is a path composed of $PATH(j, t_i^k + \tau_{i,j}(t_i^k))$, which is appended to node n_i $\forall t_i^k \in \mathfrak{T}_i$. Thus, for any node n_i and $\forall t_i^k \in \mathfrak{T}_i$, $PATH(i, t_i^k)$ is concatenated.

We prove that $PATH(i, t_i^k)$ is loop-free by contradiction. Without loss of generality, assume that there is a loop: $(n_{i_0} \rightarrow n_{i_1} \rightarrow n_{i_2} \cdots n_{i+k} \rightarrow n_{i_0})$. This means that, there is a shortest path $(n_{i_0} \rightarrow n_{i_1} \cdots, \rightarrow, n_{i+k})$, where n_{i+k} is one-hop away from n_{i_0}. According to the triangular path condition in Lemma 1, such a shortest path cannot exist because $n_{i_0} \rightarrow n_{i+k}$ is always the shortest one among all paths from n_{i_0} to n_{i+k}, contradicting the assumption.

We prove part 2 by induction on m. It is easy to find that the claim is true for $m = 0$. Now, assume that the claim is true for $m-1$ (i.e., the inductive hypothesis). We now prove for m by induction.

Consider a specific time t_i^k and a node n_i such that there is a shortest path with at most m hops between n_i and n_s. Let $SP(i, s, t_i^k, m)$ be the shortest path, which is at most m hops from n_i to n_s at time t_i^k. Let n_j be n_i's parent on $SP(i, s, t_i^k, m)$ at t_i^k. This means that, there is a path with at most $m-1$ hops between n_j and n_s.

By the inductive hypothesis, n_j determined its shortest distance at the $(m-1)^{th}$ iteration. In the m-*th* iteration, node n_j sends its minimized distance vector $\vec{d}_{j_{m-1}}$ to node n_i. Thus, $SP(i, s, t_i^k, m)$ is determined after receiving the vector $\vec{d}_{j_{m-1}}$ in the m^{th} iteration, which completes the inductive step. Since t_i^k is chosen arbitrarily, this holds for all values of $t_i^k \in \mathfrak{T}_i$. □

Part 2 in Theorem 5 implies that, for $m \geq D_m$, all nodes determine their minimum delay and the corresponding parents for all time intervals, since all shortest paths contain at most D_m nodes.

Theorem 6: The message and time complexity of FTSP is $O(D_m|E|)$ and $O(D_m|V|)$, respectively.

Proof: Based on the implementation of the β-synchronizer in [32], in each iteration, there is exactly one message traversing each link in the spanning tree, totally $|E|$ messages exchanged. By Theorem 5, the number of iterations is upper bounded by the longest shortest path's length D_m. Thus, the message complexity is $O(D_m|E|)$.

Suppose the largest delay for transmitting a message in the spanning tree is a constant, denoted by $|C|$. In each iteration, the time consumed is at most $|V| * |C|$. Since there are at most D_m iterations, the time complexity is $O(D_m|V|)$. □

6 Algorithm for Dynamic Route Maintenance

When compared with static networks, link changes and node changes are more frequent in duty-cycled WSNs. If a node changes its duty-cycle configuration, or dynamically joins or leaves the network, the links connecting with all its neighbors will be changed at multiple time intervals. In such a situation, a single node update usually causes multiple link updates.

Some previous works in static networks (e.g., [15]) have proposed solutions that efficiently deal with single link updates. They are inefficient for multiple link updates caused by a single node update. The algorithms in [15] are also memory-inefficient, since each node stores the route entries for all other nodes, incurring the space complexity of $O(|V|)$.

Unlike previous works [15], where each node stores the route information for all other nodes, we propose a solution in which a node only stores the route to the sink, which is more practical in WSNs due to their memory constraints. In our proposed algorithm, when one node is updated (denoted as the source node), the algorithm does not update the shortest path for the whole network from scratch, but only updates necessary nodes. Thus, the main idea is first to identify which nodes need to be updated. After that, the algorithm updates the shortest path for these identified nodes. The updating process is similar to the route construction described in Section 5. But the starting point is the source node, rather than from the sink node, and the updating scope is just a subset of the whole network.

The proposed distributed algorithms for path maintenance are also equipped with the $\beta-$synchronizer. Again, we choose the β-synchronizer in order to avoid exponential message complexity as discussed in Section 2.

The proposed algorithms, referred to as FTSP-M ("M" meaning maintenance), focus on per-node update and can be easily extended to node insertion and deletion. If there are multiple node updates, the algorithms will run concurrently at multiple nodes.

6.1 Overview and Rational of the Distributed Algorithms

Suppose the source update node is n_u and the corresponding input change is σ. We divide σ into two parts: σ_{inc} and σ_{dec}, where σ_{inc} includes the increasing links for $\forall t_u^k \in \mathfrak{T}_u$, and σ_{dec} includes the decreasing links for $\forall t_u^k \in \mathfrak{T}_u$. Let $\delta(\sigma_{inc})$ be the set of nodes that change either the distance or the parents for all infinite discrete time intervals, as a consequence of σ_{inc}. Similarly, let $\delta(\sigma_{dec})$ be the set of nodes affected by σ_{dec}. Apparently, $\delta(\sigma) = \delta(\sigma_{inc}) \cup \delta(\sigma_{dec})$.

We identify the input change $\delta(\sigma)$ as $\delta(\sigma_{inc})$ and $\delta(\sigma_{dec})$, because $\delta(\sigma_{dec})$ is easy to handle given the sufficient loop-free condition claimed in [33], which is referred to as the *distance increase condition* (or DIC). With DIC, at time t, if node n_i detects a link-cost decrease or a decrease in the distance reported by a neighbor, node n_i is free to choose its new parents. Therefore, the node in $\delta(\sigma_{dec})$ can safely select a new parent.

However, things become more complicated for nodes in $\delta(\sigma_{inc})$, since a loop can be formed if they directly choose a new parent [33]. In order to address this issue, we adopt two phases. In the first phase, we increase the node-to-sink distance for all nodes, which are descendants of n_u. In the second phase, those nodes will re-select their parents based on the Bellman-Ford approach.

We use the β-synchronizer in phase 1 and phase 2, in order to bound the message complexity, though it will

introduce additional time cost. For WSNs, which have limited energy resource, less communication is usually more important for some applications, which is the motivation for using the β-synchronizer.

6.2 Algorithm Descriptions

We use similar data structures and message formats as that in Section 5.1. FTSP-M consists of two phases for node n_u and all nodes $n_i \in \delta(\sigma)$. We describe our algorithms in Algorithms 3 and 5.

In phase 1, an initial spanning tree is built up gradually to contain all nodes in $\delta(\sigma_{inc})$. The purpose of phase 1 is to let all nodes in $\delta(\sigma_{inc})$ increase their distances to the sink node as a consequence of σ_{inc}, along the time-expanded shortest path trees rooted at n_u. After the termination of phase 1, all nodes in $\delta(\sigma)$ will never increase their distance again.

In each iteration, node n_u sends a control message to all neighbors. Every other node, i.e., n_i ($n_i \neq n_u$) will send control messages to all its neighbors if it is in the spanning tree and receives a message from its parent. If n_i is not in the spanning tree in the current iteration, it checks whether n_j is its parent in its shortest path after receiving a control message from n_j, which can be done by checking whether $n_j \in \vec{p_i}$. If true, n_v will join the spanning tree and set $newsp_i = n_j$. By doing so, the spanning tree will increase at most one level in each iteration.

Algorithm 3: Operations at update node n_u in FTSP-M:

Initialization:
Same initialization as in Algorithm 2;
if $p_u \neq null$ **then**
  for $\forall n_j \in N_u$ **do**
    ($\vec{d_j}$,l_j^0) = get_dist(n_j); /* retrieve $\vec{d_j}$, detailed implementation is omitted */
    $\vec{d_j}(u)$=vec_add($\tau_{u,j}^{\rightarrow}$,$\vec{d_j}$,$l_{u,j}^0$,l_j^0); /* Equation 11*/
    inc_update($\vec{d_u}$,$\vec{p_u}$, n_j, $\vec{d_j}(u)$);

Phase 1: On receiving START:
send MSG(u,u,$\vec{d_u}$,false) to $\forall n_j \in N_u$;

Phase 1: On receiving MSG(u, j, $\vec{d_j}$, bChanged):
$ACK[j] = 1$; $updated[j] = bChanged$;
if ($\forall n_j \in N_u, ACK[j] == 1$) **then**
  if ($\forall n_j \in N_u$, $updated[j] == false$) **then**
    Beginning Phase 2;
  else
    send MSG(u,u,$\vec{d_u}$,false) to $\forall n_j \in N_u$;
    $\forall n_j \in N_u, ACK[j] = 0$;

Phase 2: On Beginning Phase 2:
send MSG(u,u,$\vec{d_u}$,false) to $\forall n_j \in N_u$;

Phase 2: On receiving MSG(0, j, $\vec{d_j}$, bChanged) from n_j:
$ACK[j] = 1$; $updated[j] = bChanged$; $d_u^{\vec{prev}} = \vec{d_u}$;
$\vec{d_u}(j)$ = vec_add($\vec{d_u}$, $\vec{d_j}$, $l_{i,j}^0$, l_j,); /* Equation 11 */
($\vec{d_u}$, $\vec{p_u}$) = vmin($\vec{d_u}$, $\vec{d_i}(j)$); /* Equation 12 */
if ($\vec{d_u} \vec{<} d_u^{\vec{prev}}$) updated[j] = true;
if ($\forall n_j \in N_u, ACK[j] == 1$) **then**
  if ($\forall n_j \in N_u$, $updated[j] == false$) **then**
    STOP;
  else
    send MSG(u,u,$\vec{d_u}$,false) to $\forall n_j \in N_u$;
    $\forall n_j \in N_u, ACK[j] = 0$;

Algorithm 4: Function: inc_update($\vec{d_1}$,$\vec{p_1}$,n_2,$\vec{d_2}$)

d_1 = exp($\vec{d_1}$,($|\vec{d_1}| * |\vec{d_2}|$)/$|\vec{d_1}|$); d_2 = exp($\vec{d_2}$,($|\vec{d_1}| * |\vec{d_2}|$)/$|\vec{d_2}|$);
p_1 = exp($\vec{p_1}$,($|\vec{d_1}| * |\vec{d_2}|$)/$|\vec{d_1}|$); flag = false;
for $k = 0$ **to** $|\vec{d_1}| * |\vec{d_2}| - 1$ **do**
  if $p_1[k] == n_2$ && $d_1[k] < d_2[k]$ **then**
    $d_1[k] = d_2[k]$; flag = true;
return flag;

A control message will traverse from the root (n_u) to all other nodes in the spanning tree just like that in FTSP. When node n_i receives a control message from n_j, it only updates the element to be increased in its distance vector, as illustrated in Algorithm 4.

When replies from all its neighbors have been received, node n_u concludes that an iteration is completed. When replies are received from all neighbors by a node, a control message is sent to the parent, thereby completing the iteration at the node. When there is no distance increase for all nodes in $\delta(\sigma_{inc})$, phase 1 will be terminated and node n_u will start phase 2.

In phase 2, the initial spanning tree built up in *phase 1* is continuously growing until it contains all nodes in $\delta(\sigma)$. Phase 2 is also running by iterations. In each iteration, when a node n_i not in the spanning tree receives a control message from n_j, if the value of any elements in its distance vector is decreased, it will join the spanning tree by setting its parent $newsp_i$ to n_j. The distance update and message traversing in phase 2 of FTSP-M are just similar to that in FTSP.

6.3 Correctness and Complexity

In phase 1, all nodes in $\delta(\sigma_{inc})$ do not change their parents, but increase their distances as a consequence of σ_{inc}. Thus, there is no loop in phase 1. In phase 2, all nodes in $\delta(\sigma)$ will never increase their distances, thereby satisfying the distance increase condition [33]. All paths are therefore loop-free.

Theorem 7: In phase 1, each node in $\delta(\sigma_{inc})$ with at most m hops away from n_u along the time-dependent shortest path will not increase its distance after m iterations.

Proof: The proof is by induction on m. It is easy to find that the claim is true for $m = 0$. Now, assume that the claim is true for $m-1$ (i.e., the inductive hypothesis). We now prove for m by induction.

Consider a specific time t_i^k and a node $n_i \in \delta(\sigma_{inc})$ such that there is a shortest path with at most $m-1$ hops between n_i and n_s after node update. Let $SP(i, u, t_i^k, m-1)$ be the shortest path which is at most $m-1$ hops from n_i to n_u at time t_i^k. Let n_j be n_i's neighbor which is not updated yet due to σ_{inc}. Then n_j is at most m hops away from n_u.

By the inductive hypothesis, n_i will not increase its distance after $(m-1)^{th}$ iteration. In the m-*th* iteration, node n_i sends its updated distance vector $\vec{d}_{j_{m-1}}$ to node n_j, and n_j will determine its updated distance in the iteration if $n_j \in \delta(\sigma_{inc})$. Thus, $SP(j, u, t_i^k, m)$ is determined after receiving the vector $\vec{d}_{i_{m-1}}$ in the m^{th} iteration, which completes the inductive step. Since t_i^k is chosen arbitrarily, this holds for all values of $t_i^k \in \mathfrak{T}_i$.

Algorithm 5: Operations in node $n_i (n_i \neq n_u)$ in FTSP-M:

Initialization: $newsp_i = null$; $\forall n_j \in N_i$,$updated[j] = false$;

Phase 1: On receiving MSG(*u*, *j*, $\vec{d_j}$, bChanged) from n_j:

```
if n_j ∈ N_u then
    re-measure τ_{i,u} at the beginning of one time slot;
    reset l^0_{i,j} and l^0_u;
d_i(j)=vec_add(τ_{i,j},d_j,l^0_{i,j},l^0_j);
if newsp_i == null then
    if (inc_update(d_i,p_i,n_j,d_i(j))) then
        newsp_i = n_j; send MSG(u,i,d_j,true) to n_j;
    else send MSG(u, i, d_j, false) to n_j;
else
    FORWARD(u); /* u means the MACRO is executed in phase 1 */
    ACK[j] = 1; updated[j] = inc_update(d_i,p_i,n_j,d_i(j));
    ACK_REPLY(u);
```

Phase 2: On receiving MSG(0, *j*, $\vec{d_j}$, bChanged):

```
if newsp_i == null then
    if (updated[j]) then
        newsp_i = n_j; send MSG(0, i, d_j, true) to n_j;
    else send MSG(0, i, d_j, false) to n_j;
else
    FORWARD(0); /* 0 means the MACRO is executed in phase 2 */
    ACK[j] = 1; updated[j] = bChanged; d_i^prev = d_i;
    d_u(j) = vec_add(d_u, d_j, l^0_{i,j}, l_j); /* Equation 11*/
    (d_u, p_u) = vmin(d_u, d_i(j)); /* Equation 12*/
    if (d_u < d_u^prev) updated[j] = true;
    ACK_REPLY(0);
```

FORWARD(int dict): code macro

```
if (newsp_i == n_j) then
    send MSG(dict,i,d_i,false) to ∀n_j ∈ N_i except newsp_i;
```

ACK_REPLY(int dict): code macro

```
if (∀n_j ∈ N_i, ACK[j] == 1) then
    if (∃n_j ∈ N_i, updated[j] == true) then bUpdated = true;
    else then bUpdated = false;
    send MSG(dict, i, d_i,bUpdated) to newsp_i;
    ∀n_j ∈ N_i, ACK[j] = 1; bUpdated = false;
```

□

By Theorem 7, after $|\delta(\sigma_{inc})|$ iterations, all nodes in $\delta(\sigma_{inc})$ will not increase their distance anymore.

Definition 2: Updated-subpath: for any node $n_i \in \delta$, the updated-subpath is from n_i to the first node n_e not in δ along the shortest path from n_i to n_u.

Theorem 8: In phase 2, all generated updated-subpaths are loop-free, and updated-subpaths with at most m hops long are determined in the m^{th} iteration.

Proof: After phase 1, the DIC loop-free condition [33] is satisfied. Thus, all updated-subpaths are loop free in phase 2.

The proof for at most m hops-long updated-subpaths being determined in the m^{th} iteration can be done by induction. It is easy to find that the claim is true for $m = 0$. Now, assume that the claim is true for $m - 1$ (i.e., the inductive hypothesis). We now prove for m by induction.

Consider a specific time t_i^k and a node $n_i \in \delta$ such that there is an updated-subpath which contains m hops between n_i and n_u. Let $UP(i, u, t_i^k, m)$ be the updated-subpath which is at most m hops long at time t_i^k. Let n_j be n_i's parent on $UP(i, u, t_i^k, m)$ at t_i^k. This means that there is an updated-subpath with at most $m - 1$ hops between n_j and n_s.

By the inductive hypothesis, the updated subpath from n_j to n_u is determined at the $(m-1)^{th}$ iteration. In the m-*th* iteration, node n_j sends its minimized distance vector $\vec{d}_{j_{m-1}}$ to node n_i. Thus, $UP(i, u, t_i^k, m)$ is determined after receiving the vector $\vec{d}_{j_{m-1}}$ in the m^{th} iteration, which completes the inductive step. Since we choose t_i^k arbitrarily, the theorem holds for $\forall t_i^k \in \mathfrak{T}_i$. □

Theorem 9: The message complexity for per node update with $\delta(\sigma)$ output change is $O(|\delta(\sigma)|^2 * maxdeg)$. The time complexity is $O(|\delta(\sigma)|^2)$, and space complexity is $O(maxdeg)$.

Proof: In phase 1, the number of iterations is $\delta(\sigma_{inc}) \leq \delta(\sigma)$. In phase 2, there are at most $\delta(\sigma)$ iterations before all updated-subpaths are decided. Thus totally, there are $O(|\delta(\sigma)|^2 * maxdeg)$ messages. In each iteration, the consumed time is at most $|\delta(\sigma)| * C$ (C is the largest transmission delay for all links). Thus, the message complexity is $O(|\delta(\sigma)|^2)$. Since a node only stores the information of all its neighbors, the space complexity is $O(maxdeg)$. □

7 Practical Implementation

We now discuss some practical implementation issues. In particular, we discuss how to achieve awareness of schedules of neighborhood and how to reduce the vector size given a large LCM in Equation 6.

7.1 Schedule Awareness

FTSP and FTSP-M require awareness of wakeup schedules of the neighborhood. Achieving this requirement depends on the specific underlying MAC protocol. We discuss three scenarios for achieving schedule awareness over different MAC protocols.

Active Neighbor Discovery: This means that a node needs to probe the schedules of its neighbors actively. We consider two scenarios which need active neighbor discovery. One is the LPL mode as adopted by B-MAC [5] and X-MAC [6]. The other is the low duty-cycling mode, where time axis are arranged as consecutive short time slots, and all slots have the same duration.

For either scenario, we assume that beacon messages are sent out at the beginning of wakeup slots, similar to [28], [34]. In order to discover neighbors, a node has to stay awake in order to detect the beacon message of its neighborhood. The node should wait until beacons are received from its neighbors. The frequency with which a node should detect its neighbors' schedule depends on implementation considerations.

Quorum-based Duty-Cycling: Active neighbor discovery mechanisms requires a node to stay awake actively. Now we introduce quorum-based duty-cycling which does not have that requirement. Here, the wakeup schedule follows a quorum system design [35]. In quorum-based duty cycling, two neighbor nodes can hear each other at least once within bounded time slots via the non-empty intersection property. We choose cyclic quorum systems [28] for presentation.

We use the following definitions for briefly reviewing quorum systems (used for wakeup scheduling). Consider a cycle length n and $U = \{0, \cdots, n-1\}$.

Definition 3: A *quorum system* $\mathcal{Q}$ under U is a superset of non-empty subsets of U, each called a quorum, which satisfies the intersection property: $\forall G, H \in \mathcal{Q} : G \cap H \neq \emptyset$. If $\forall G, H \in \mathcal{Q}$, $i \in \{0, 1, ...n-1\}$: $G \cap (H+i) \neq \emptyset$, where $H + i = \{(x+i) \ mod \ n : x \in H\}$, $\mathcal{Q}$ is said to have the *rotation closure property*.

Cyclic quorum systems (or cqs) satisfy the rotation closure property, and are denoted as $C(A, n)$, where A is a quorum and n is the cycle length. For example, the cqs $\{\{1,2,4\},\{2,3,5\}\cdots,\{7,1,3\}\}$ can be denoted as $C(\{1,2,4\},7)$.

Given two different cyclic quorum systems $C(A_1, n_1)$ and $C(A_2, n_2)$, if two quorums from them, respectively, have non-empty intersections, then, even with clock drift, they can be used for heterogenous wakeup scheduling in WSNs. For example, given $C(\{1,2,4\},7)$ and $C(\{1,2,4,10\},13)$, two quorums from them, respectively, have non-empty intersection for every 13 time slots. Therefore, two nodes that wake up with schedules complying with any two quorums from these two cyclic quorum systems can hear each other.

By embedding the wakeup schedule information in the beacon message, a node can always detect the wakeup schedules of its neighborhood through the non-empty intersection property of the quorum design.

Synchronization: MAC protocols with synchronization require that all neighboring nodes wake up at the same time. The simplest method for doing this is to use a fully synchronized pattern, like that in the S-MAC protocol [36]. In this case, all nodes in the network wakeup at the same time according to a periodic pattern. A further improvement can be achieved by allowing nodes to switch off their radio when no activity is detected for a timeout value, like that in the TMAC protocol [37]. In this scheme, neighboring nodes form virtual clusters to set up a common sleep schedule. The main disadvantages of such scheduled rendezvous schemes are the complexity of implementation and the overhead for synchronization. Through synchronization, a node can conveniently know the schedules of its neighbors. Schedule awareness can be achieved by periodic message exchange between a node and its neighbors.

7.2 Sub-optimal Implementation with Vector Compression

The key implementation aspect of our proposed algorithms is the vector representation of link cost functions and distance functions. However, if the vector size is too large (i.e., the LCM in Equation 6 is too large), the proposed algorithms, FTSP and FTSP-M, may not be feasible given the limited memory resource of embedded sensor nodes, and inefficient due to distributed message exchanging. Based on Theorem 2, the vector size is depending on $LCM(T_0, T_1, \cdots, T_i)/T_i$. The worst case is that all nodes have different cycles, and the size of the distance vector can be very large.

In a real implementation, to avoid arbitrarily long vectors, there are two possible solutions: 1) Use a predefined duty cycle set, so that the $\frac{LCM(T_0,T_1,\cdots,T_i)}{T_i}$ can be bounded by carefully selecting a duty cycle set, as discussed in Sections 4.1 and 4.2; 2) Adopt vector compression to achieve a trade-off, i.e., adopt a low-accurate distance vector, which takes less memory space, to represent the end-to-end latency. Hence, the output path is sub-optimal in terms of latency.

The first solution can be applied to small-scale networks, where the node number is not large and predefining a duty cycle set is not difficult. For a large-scale network, we might need the second solution in which a bounded, global $\frac{LCM(T_0,T_1,\cdots,T_i)}{T_i}$ is not necessary.

The basic idea of vector compression in the second solution is to smooth all values in a vector and represent the vector with less information. For example, for a vector [1 2 3 4 5 6] with 6 elements, we can approximately represent the vectors by a vector with 2 elements, such as $[(2,3),(5,3)]$ (where $2 = (1+2+3)/$ and $5 = (4+5+6)/3$). Each tuple (v, s) in the vector represents the average value of s elements in the original vector.

The formal description of vector compression is as follows:

Vector compression: Suppose the source vector is $v_s = [v_1, v_2, \cdots, v_n]$ and the target vector size is m ($n > m$). We compress v_s by:

$$v_t = [(\frac{v_1 + v_2 + \cdots + v_{len}}{len}, len),$$
$$(\frac{v_{1+len} + v_{2+len} + \cdots + v_{2*len}}{len}, len), \cdots,$$
$$(\frac{v_{1+(m-1)*len} + v_{2+(m-1)*len} + \cdots + v_n}{n-(m-1)*len}, n-(m-1)*len)]$$

where $len = \lceil \frac{n}{m} \rceil$.

We choose the average value $\frac{v_{1+i*len}+v_{2+len}+\cdots+v_{(i+1)*len}}{len}$ as the value of $v_t[i]$, because the expected deviation can be minimized by $P = \sum_{j=1}^{len} \frac{1}{len}|v_t[i] - v[j]|$.

In addition to the averaging filter used in the above equation, other filters, such as the Wavelet transform filter [38] can also be applied for vector compression, as typically used in image compression.

7.3 Remarks

The FTSP algorithm described in Section 5 is a proactive routing protocol. Although its time complexity is $O(D_m * |V|)$ for initial route construction, it is affordable in the initial stage of WSNs. The low space complexity ($O(|maxdeg|)$) for route maintenance makes the algorithm scalable for large-scale WSNs.

Note that FTSP and FTSP-M target the ALPL mode [7] with various checking intervals. When all nodes have homogenous LPL checking intervals (like that in the standard B-MAC), according to Equations 3 and 5, the link cost function and the distance function will become constants. In such a case, our algorithms will default to the static shortest path algorithm. However, FTSP and FTSP-M will yield the same message complexities and time complexities for the static situation.

Our work focuses on the scenario of a single sink node. However, it can be extended to multiple destinations. In WSNs, there is usually no end-to-end communication between two arbitrary nodes. We only consider the generalization of communication between one node and multiple

sink nodes, rather than the communication between two arbitrary nodes.

8 Experimental Results

We evaluated the performance of FTSP, FTSP-M, and the sub-optimal implementation through extensive simulations using the OMNET++ discrete event simulator [39]. We compared our algorithms with other related algorithms for the TDSP problem, including the distributed Bellman-Ford algorithm [31] adapted to the time-dependent model (Section 4), referred to here as TD-Bellman, and DSPP1 [10]. The following three major metrics were measured in the evaluation: 1) message count, 2) time cost, 3) average memory cost.

We examined two main factors that affect the performance of our algorithms, including network size and the underlying duty cycle setting. Our experimental settings were compatible with typical configurations, as in [3], [7], [8]. The wireless communication range in our simulation was set to 10m. We adopt the wireless loss model used in [40], which considers the oscillation of radio links.

We generated 8 network size sets with varying sizes, G_1,$\cdots$,G_8, which are listed in Table 1. For each network size, we randomly generated 10 topologies. Each data point presented in our simulation results in this section is the average of 10 topologies, with 10 runs on each topology.

TABLE 1
Network Size Sets

	G_1	G_2	G_3	G_4
$\|V\|$	50	80	200	400
	G_5	G_6	G_7	G_8
$\|V\|$	600	1K	1.5K	2K

TABLE 2
Time Slot Sets

C_1 (ms)	$\{100, 100, 100, 100\}$
C_2 (ms)	$\{100, 200, 300, 600\}$
C_3 (ms)	$\{100, 200, 400, 800\}$
C_4 (ms)	$\{100, 200, 500, 1000\}$

We chose two MAC protocols: ALPL (adaptive low power listening) and quorum-based duty-cycling. In the ALPL mode, a node just wakes up for a short time during a checking interval to check the channel activities. The duration of the checking interval varies for different nodes. We changed the duration of the checking interval in our simulation experiments with 4 sets, C_1, C_2, C_3, and C_4, as listed in Table 2. With each set, we randomly chose one element as the value of the LPL checking interval for each node. With different time slot sets, the size of a message (changed with vector size) is changing. Thus, we used a flexible packet size in our simulation. Each element in a vector occupied 1 byte in all experiments.

For quorum-based duty-cycling, we choose the $(7, 3, 1)$, and $(21, 5, 1)$ difference sets for the heterogenous wakeup schedule settings. The duration of one time slot was set to 100ms in quorum-based duty-cycling. Since FTSP, FTSP-M, TD-Bellman, and DSPP1 are independent of wakeup scheduling, we argue that the comparison is fair even when we choose quorum-based duty-cycling.

8.1 Least-latency Path Construction

In the first set of simulation experiments, we measured the ALPL mode and chose C_4 as the time slot setting, which indicates that the largest distance vector size is 10 by Equation 6, and varied the network size. With the number of nodes increasing from 50 to 2000 in G_1,$\cdots$,G_8, the average time consumed and the message count are shown in Figure 4.

The average execution time of DSPP1 is about 10 times larger than that of FTSP, since DSPP1 has to be executed 10 times to compute the shortest paths for all time intervals. FD-Bellman is better than FTSP when the network size is small, since FD-Bellman does not have a distributed synchronizer in its execution. When the network size becomes large (i.e., $\geq$ 1K), FTSP outperforms FD-Bellman due to the exponential worst case message complexity of the Bellman-Ford algorithm. We observed similar trends for time cost for the three algorithms, as shown in Figure 4.

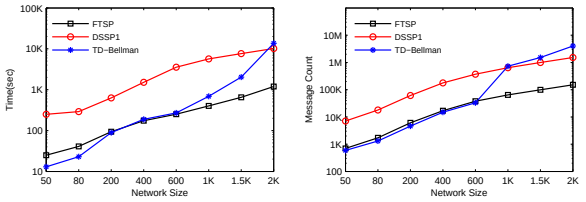

Fig. 4. Comparison of time efficiency and message efficiency by varying $|V|$

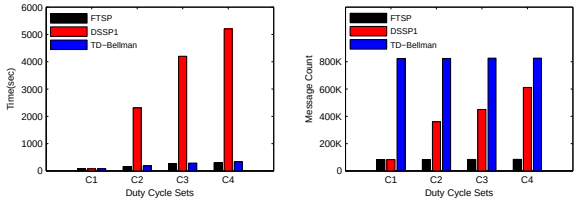

Fig. 5. Comparison of time efficiency and message efficiency by varying time slots in ALPL mode

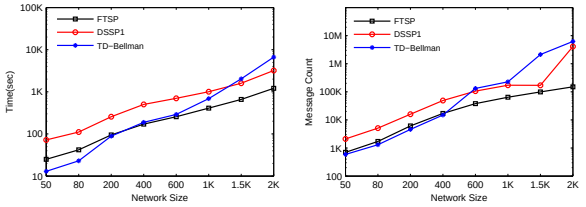

Fig. 6. Comparison of time efficiency and message efficiency for quorum-based duty-cycle setting

We also varied the time slot sets with a fixed network size of G_6, which represents medium-sized WSNs. The results are shown in Figure 5. We observe that FTSP and FD-Bellman do not change their message count significantly since they only depend on the network size. The time cost of all algorithms become worse when the average value of all elements in the selected time slot set becomes larger, since the average link delay is correspondingly increasing.

Finally, we measured the performance for quorum-based duty-cycling by fixing the network size of G_6. Each node randomly chose the $(7, 3, 1)$ and $(21, 5, 1)$ difference

sets for its heterogenous schedule settings. As shown in Figure 6, we observe similar trends for execution time and message count. The average execution time of DSPP1 is about 3 times larger than that of FTSP, since DSPP1 has to be executed 3 times to compute the shortest paths for all time intervals given the $(7, 3, 1)$ and $(21, 5, 1)$ difference sets. The time costs for all algorithms become worse for larger sized networks, which is consistent with the conclusion in Theorem 6.

8.2 Least-latency Path Maintenance

For evaluating the path maintenance performance of FTSP-M, we return to static networks by selecting the time slot set of C_1 in Table 2 for the ALPL mode. We do so for the purpose of a fair comparison with the previous work in [15], referred to here as Full-Dynamic and DSDV [41] with unicast support.

We first evaluate the effect of input changes on all algorithms for medium-sized networks by choosing G_6. As shown in Figure 7, FTSP-M achieves the median message cost and time cost when input changes become large. The reason is that, DSDV suffers from exponential message complexity. In addition, FTSP-M uses the synchronizer, which consumes additional time and messages, which is not as efficient as that in Full-Dynamic.

We also measured the average memory cost by varying the network size for the two underlying duty-cycling mechanisms. We first chose the ALPL mode and set the time slot set of C_1 in Table 2 for all nodes. The results shown in Figure 8 indicate that FTSP-M achieves the best memory cost, which does not depend on the network size. The memory costs of DSDV and Full-Dynamic increase with the network size, since each node stores an entry for all other nodes.

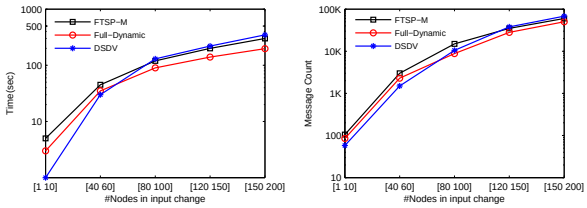

Fig. 7. Performance comparison for route maintenance by varying input change: ALPL mode

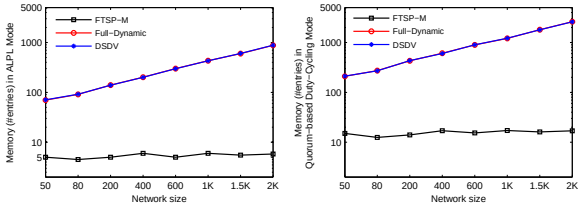

Fig. 8. Performance comparison for route maintenance on memory required in each node

Finally, we measured the average memory cost for the quorum-based duty-cycling mechanism. Each nodes randomly chose the $(7, 3, 1)$ difference sets for the heterogenous schedule settings. As shown in Figure 8, we observe similar trends for the quorum-based duty-cycling mechanism. The only difference is that with $(7, 3, 1)$ difference sets, each node maintains 3 entries in the routing table for all neighbors.

8.3 Performance of Sub-Optimal Implementation

For sub-optimal implementation with vector compression, the performance is a trade-off between path latency and message size. We evaluated this tradeoff for both initial route construction and route maintenance. We first fixed the duty cycle setting by choosing C_4 and compared the performance between FTSP and its sub-optimal implementation. Since the message count does not rely on vector implementation, we compared the vector size (in terms of the number of elements in a vector). Since each element took one byte in a message packet, the packet size (in bytes) represents the vector size.

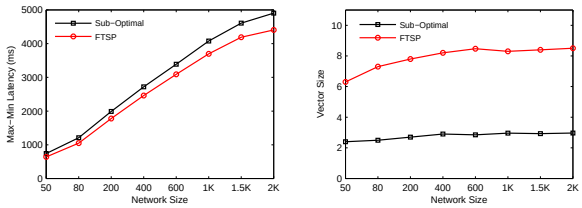

Fig. 9. Performance comparison for route construction: latency and vector size (in bytes) over ALPL mode

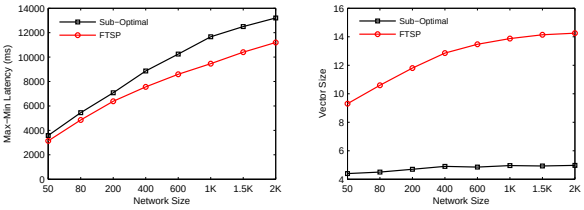

Fig. 10. Performance comparison for route construction: latency and message size over quorum-based duty-cycling mode

Figure 9 shows the results. We observe that the sub-optimal implementation achieves less message size with vector compression. To understand the end-to-end latency of the two techniques, we compared the maximum value of the least latency achieved by all nodes (defined as max-min latency). As shown in Figure 9, the sub-optimal implementation has higher max-min latency than FTSP. However, the sub-optimal implementation has a smaller average message size than FTSP.

We then fixed the duty cycle setting by choosing the $(7, 3, 1)$ and $(21, 5, 1)$ difference sets as the wakeup schedule for all nodes. We observed similar trends, as shown in Figure 10. We observe that the sub-optimal implementation has higher max-min latency than FTSP, but has a smaller vector size. The simulation results show the performance tradeoff after introducing the sub-optimal implementation, which has smaller message size but higher latency.

9 Conclusions

In this paper, we addressed the distributed shortest path routing problem in duty-cycled WSNs. Our contributions are four-fold. First, we modeled duty-cycled WSNs as time-dependent networks, which satisfy the FIFO condition. Second, we presented the FTSP algorithm for finding shortest paths in such networks. FTSP has polynomial message complexity and is more time-efficient than previous solutions. Third, we presented FTSP-M for distributed route maintenance with node insertion, updating, and deletion. FTSP-M is memory efficient and has polynomial message complexity. Finally, we proposed a sub-optimal implementation on vector representations to reduce memory requirements. The vector size of the sub-optimal solution does not depend on the largest LCM value as shown in Equation 6. Simulation results validated the effectiveness and efficiency of our solutions.

We envision several directions for future work. One is to investigate the time-dependent minimum spanning tree problem, which is NP-Hard in duty-cycled WSNs. Another direction is to study time-dependent multicast routing in duty-cycled WSNs, which is a required service for many applications and is the reverse direction of all-to-one least-latency routing.

References

[1] Shouwen Lai and Binoy Ravindran, "On distributed time-dependent shortest paths over duty-cycled wireless sensor networks," in *IEEE Conference on. Computer Communications (INFOCOM)*, 2010.

[2] C. Schurgers and M.B. Srivastava, "Energy efficient routing in wireless sensor networks," in *Military Communications Conference (MILCOM 2001)*, 2001, vol. 1, pp. 357–361.

[3] A. Woo, T. Tong, and D. Culler, "Taming the underlying challenges of reliable multihop routing in sensor networks," in *Proceedings of the 1st international conference on Embedded networked sensor systems (Sensys)*, 2003, pp. 14–27.

[4] C.M. Vigorito, D. Ganesan, and A.G. Barto, "Adaptive control of duty cycling in energy-harvesting wireless sensor networks," in *IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, June 2007, pp. 21–30.

[5] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in *International conference on Embedded networked sensor systems (Sensys'04)*, 2004, pp. 95–107.

[6] Michael Buettner, Gary V. Yee, Eric Anderson, and Richard Han, "X-mac: a short preamble mac protocol for duty-cycled wireless sensor networks," in *ACM Sensys*, New York, NY, USA, 2006, pp. 307–320.

[7] Raja J., Pierre B., and Cristina V., "Adaptive low power listening for wireless sensor networks," *IEEE Transactions on Mobile Computing*, vol. 6, no. 8, pp. 988–1004, 2007.

[8] Y. Gu and T. He, "Data forwarding in extremely low duty-cycle sensor networks with unreliable communication links," in *Proceedings of the 6th ACM conference on Embedded network sensor systems (Sensys)*, 2007, pp. 32–38.

[9] K. L. Cooke and E. Halsey, "The shortest route through a network with time-dependent internodal transit times," *J. Math. Anal. Appl.*, vol. 14, pp. 493–498, 1966.

[10] Ariel Orda and Raphael Rom, "Distributed shortest-path protocols for time-dependent networks," *Distributed Computing*, vol. 10, no. 1, pp. 49–62, 1996.

[11] I. Chabini, "Discrete dynamic shortest path problems in transportation applications: Complexity and algorithms with optimal run time," *Transportation Research Records*, vol. 1645, pp. 170–175, 1998.

[12] Bolin Ding, Jeffrey Xu Yu, and Lu Qin, "Finding time-dependent shortest paths over large graphs," in *Proceedings of Extending database technology (EDBT'08)*, 2008, pp. 205–216.

[13] H. Chon, D. Agrawa, and A. Abbadi., "Fates: Finding a time dependent shortest path," *Mobile Data Management*, vol. 2574, pp. 165–180, 2003.

[14] Ariel Orda and Raphael Rom, "Minimum weight paths in time-dependent networks," *Networks*, vol. 21, pp. 295–319, 1991.

[15] Serafino C., Gabriele D., Daniele F., and Umberto N., "A fully dynamic algorithm for distributed shortest paths," *Theoretical Computer Science*, vol. 297, no. 1-3, pp. 83–102, 2003.

[16] G. D'Angelo, S. Cicerone, G. Di Stefano, and D. Frigioni, "Partially dynamic concurrent update of distributed shortest paths," in *International Conference on Computing: Theory and Applications*, 2007, pp. 32–38.

[17] H. Wang, X. Zhang, F. Abdesselam, and A. Khokhar, "Dps-mac: An asynchronous mac protocol for wireless sensor networks," June 2007, vol. 7, pp. 393–404.

[18] Lu Su, Changlei Liu, Hui Song, and Guohong Cao, "Routing in intermittently connected sensor networks," 2008, pp. 278 –287.

[19] Tian He Yu Gu, "Bounding communication delay in energy harvesting sensor networks," 2010, pp. 837–847.

[20] Yu Gu, Tian He, Mingen Lin, and Jinhui Xu, "Spatiotemporal delay control for low-duty-cycle sensor networks," in *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium (RTSS)*, 2009, pp. 127–137.

[21] Xue Yang and N.H. Vaidya, "A wakeup scheme for sensor networks: achieving balance between energy saving and end-to-end delay," 2004.

[22] G. Lu, N. Sadagopan, B. Krishnamachari, and A. Goel, "Delay efficient sleep scheduling in wireless sensor networks," 2005, pp. 2470 – 2481.

[23] K. V.S. Ramarao and S. Venkatesan, "On finding and updating shortest paths distributively," *J. Algorithms*, vol. 13, no. 2, pp. 235–257, 1992.

[24] S. Haldar, "An "all pairs shortest paths" distributed algorithm using 2n2 messages," *J. Algorithms*, vol. 24, no. 1, pp. 20–36, 1997.

[25] Giuseppe F. Italiano, "Distributed algorithms for updating shortest paths (extended abstract)," in *Proceedings of the 5th International Workshop on Distributed Algorithms (WDAG)*, London, UK, 1992, pp. 200–211, Springer-Verlag.

[26] B. Awerbuch, I. Cidon, and S. Kutten, "Communication-optimal maintenance of replicated information," in *Proceedings of the 31st Annual Symposium on Foundations of Computer Science (SFCS)*, Washington, DC, USA, 1990, pp. 492–502 vol.2, IEEE Computer Society.

[27] Baruch Awerbuch, "Complexity of network synchronization," *Journal of the ACM (JACM)*, vol. 32, no. 4, pp. 804–823, 1985.

[28] S. Lai, B. Zhang, B. Ravindran, and H. Cho, "Cqs-pair: Cyclic quorum system pair for wakeup scheduling in wireless sensor networks.," in *International Conference on Principles of Distributed Systems (OPODIS)*. 2008, vol. 5401, pp. 295–310, Springer.

[29] Richard Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[30] Philipp Sommer and Roger Wattenhofer, "Gradient clock synchronization in wireless sensor networks," in *ACM IPSN*, 2009, pp. 37–48.

[31] K. Mani Chandy and J. Misra, "Distributed computation on graphs: shortest path algorithms," *Communications of the ACM*, vol. 25, no. 11, 1982.

[32] A. Segall, "Distributed network protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 1, pp. 23–35, Jan 1983.

[33] J. J. Garcia-Lunes-Aceves, "Loop-free routing using diffusing computations," *IEEE/ACM Trans. Netw.*, , no. 1, pp. 130–141, 1993.

[34] Yanjun Sun, Omer Gurewitz, and David B. Johnson, "Ri-mac: a receiver-initiated asynchronous duty cycle mac protocol for dynamic traffic loads in wireless sensor networks," in *Proceedings of the ACM conference on Embedded network sensor systems (Sensys)*, 2008, pp. 1–14.

[35] W.S. Luk and T.T. Huang, "Two new quorum based algorithms for distributed mutual exclusion," in *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 1997, pp. 100 – 106.

[36] W. Ye, J. Heidemann, and D. Estrin, "Medium access control with coordinated adaptive sleeping for wireless sensor networks," *IEEE/ACM Transactions on Networking (TON)*, vol. 12, pp. 493–506, 2004.

[37] T.V. Dam and K. Langendoen, "An adaptive energy-efficient mac protocol for wireless sensor networks," in *The First ACM Conference on Embedded Networked Sensor Systems (Sensys)*, 2003.

[38] Stphane Mallat, *A Wavelet Tour of Signal Processing, Third Edition: The Sparse Way*, Academic Press, 2008.

[39] OMNET++, ," http://www.omnetpp.org/.

[40] M. Zuniga and B. Krishnamachari, "Analyzing the transitional region in low power wireless links," in *IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, 2004, pp. 517–526.

[41] Charles E. Perkins and Pravin Bhagwat, "Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers," *SIGCOMM Comput. Commun. Rev.*, vol. 24, no. 4, pp. 234–244, 1994.

Shouwen Lai is a senior engineer in Qualcomm Inc where he is working on CDMA software development. He received his Ph.D degree in the Department of Electrical and Computer Engineering in Virginia Polytechnic Institute and State University (Virginia Tech) in 2010. His research topics include wireless sensor networks, real-time systems and mobile computing. He is also interested in system development and implementations for wireless networking, embedded systems and distributed systems. Before coming to Virginia Tech, he worked for two years as a research engineer in Hitachi (China) R & D corporation where he conducted research and development works for network mobility, security provision, multimedia applications over 3G and WLAN networks.

Binoy Ravindran is an Associate Professor in the ECE Department at Virginia Tech. His research interests include real-time, embedded, and networked systems, with a particular focus on resource management at various levels of abstraction from OSes to virtual machines to runtimes to middleware. He and his students have published more than 170 papers in this space, and some of his group's results have been transitioned to US DoD programs. Dr. Ravindran is an US Office of Naval Research Senior Faculty Fellow, an ACM Distinguished Speaker, and an Associate Editor of ACM Transactions on Embedded Computing Systems.