Mobile Matrix: A Multihop Address allocation and Any-To-Any Routing in Mobile 6LoWPAN

Bruno P. Santos, Olga Goussevskaia, Luiz F. M. Vieira, Marcos A. M. Vieira, Antonio A.F. Loureiro
Computer Science Department, Universidade Federal de Minas Gerais – Brazil
{bruno.ps,olga,lfvieira,mmvieira,loureiro}@dcc.ufmg.br

ABSTRACT

In this work, we present Mobile Matrix, a routing protocol for 6LoWPAN that uses hierarchical IPv6 address allocation to perform any-to-any routing and mobility management without changing a node's IPv6 address. In this way, device mobility is transparent to the application level. The protocol has low memory footprint, adjustable control message overhead and achieves optimal routing path distortion. Moreover, it does not rely on any special hardware for mobility detection, such as an accelerometer. Instead, it provides a passive mechanism to detect that a device has moved. We present analytic proofs for the computational complexity and efficiency of Mobile Matrix, as well as an evaluation of the protocol through simulations. Finally, we propose a new mobility model, to which we refer as cyclical random waypoint mobility model, that we use to simulate mobility scenarios, where communication is carried out in environments with limited mobility, such as 6LoWPANs deployed in office buildings, university campuses, concert halls or sports stadiums. Results show that μMatrix deliveries 3x times more packets than RPL for top-down traffic over high mobility scenario.

CCS CONCEPTS

•Networks →Network protocol design; Network layer protocols;

KEYWORDS

Mobility; 6LoWPAN; IPv6; CTP; RPL; any-to-any routing;

ACM Reference format:
Bruno P. Santos, Olga Goussevskaia, Luiz F. M. Vieira, Marcos A. M. Vieira, Antonio A.F. Loureiro. 2017. Mobile Matrix: A Multihop Address allocation and Any-To-Any Routing in Mobile 6LoWPAN. In *Proceedings of ACM MSWiM, Miami Beach, USA, November 2017 (MSWiM '17),* 8 pages.
DOI: 10.475/123_4

1 INTRODUCTION

IPv6 over Low-power Wireless Personal Area Networks (6LoWPAN) is an IETF working group that defines standards for low-power devices to communicate with Internet Protocol. It can be applied even to the small devices to become part of the Internet of Things (IoT). It has defined protocols, including encapsulation and header compression mechanisms, that allow IPv6 packets to be sent and received over low-power devices. These protocols, such as CTP [11] and RPL [20], typically build an acyclic network topology to collect data, such as a tree or a directed acyclic graph. However, they do not handle any-to-any communication or mobility [12].

Mobility is an important factor present in everyday life. It makes life easier and turns applications more flexible. The usage of many devices for IoT can benefit from it, as is the case of today adoption of smartphones and tablets. By extending IoT protocols to handle mobility, IoT becomes even more ubiquitous.

Matrix (Multihop Address allocation and dynamic any-To-any Routing for 6LoWPAN) [17] is a platform-independent routing protocol for dynamic network topologies and fault-tolerant any-to-any data lows in 6LoWPAN. Matrix uses hierarchical IPv6 address allocation and preserves bidirectional routing.

We present Mobile Matrix (μMatrix), a solution for handling mobility in 6LoWPAN built upon the Matrix protocol. It provides the benefits from Matrix, including any-to-any routing, memory efficiency, reliability, communication efficiency, hardware independence while dealing with mobility in an efficient way. It enables Matrix to be used in scenarios and applications where mobility is present.

μMatrix handles mobility at the network layer, so the IPv6 address of each node is assigned once and kept unchanged despite mobility. In this way, routing and mobility management is transparent to the application level. The proposed communication protocol has low memory footprint, being suitable for low memory devices, such as wireless sensor networks and IoT. Since there is an intrinsic trade-off between the delay to detect that a node has moved and the number of control messages, μMatrix is able to tune the frequency of control messages according to the application or the mobility pattern. Moreover, μMatrix has optimal routing path distortion, i.e., messages addressed to a mobile node, from anywhere in the network, are sent along the shortest path from the source to its current location, using its original IPv6 address.

To the extent of out knowledge, previous mobile routing protocols for 6LoWPAN have not used hierarchical IPv6 address allocation, but a flat address structure, which incurs in more memory consumption to store the bidirectional routes. On the other hand, protocols for mobile ad hoc networks, like AODVand OLSR, have high memory footprint and control message overhead, which makes them not suitable for low power devices or 6LoWPAN.

The main contributions of this paper can be summarized as follows. We present μMatrix, a communication protocol that performs hierarchical IPv6 address allocation and manages routing and mobility without ever changing a node's IPv6 address. The protocol has low memory footprint, adjustable control message overhead and achieves optimal routing path distortion. We provide analytic proofs for the computational complexity and efficiency of μMatrix,

MSWiM '17, Miami Beach, USA

DOI: 10.475/123_4

as well as a an evaluation of the protocol through simulations. An important building block of μMatrix is the passive mobility detection mechanism that captures changes in topology without requiring additional hardware (e.g. accelerometer or compass).

Moreover, we propose a new mobility model, to which we refer as *Cyclical Random Waypoint mobility model*, that we use to simulate mobility scenarios, in which communication nodes are assigned a home location, and might make several moves in random directions, connecting to the 6LoWPAN at different attachment points, and eventually returning to their home locations. Our motivation for proposing a new mobility model comes from application scenarios, where communication is carried out in environments with limited mobility, such as 6LoWPANs deployed in office or school buildings, university campuses or concert halls or sports stadiums.

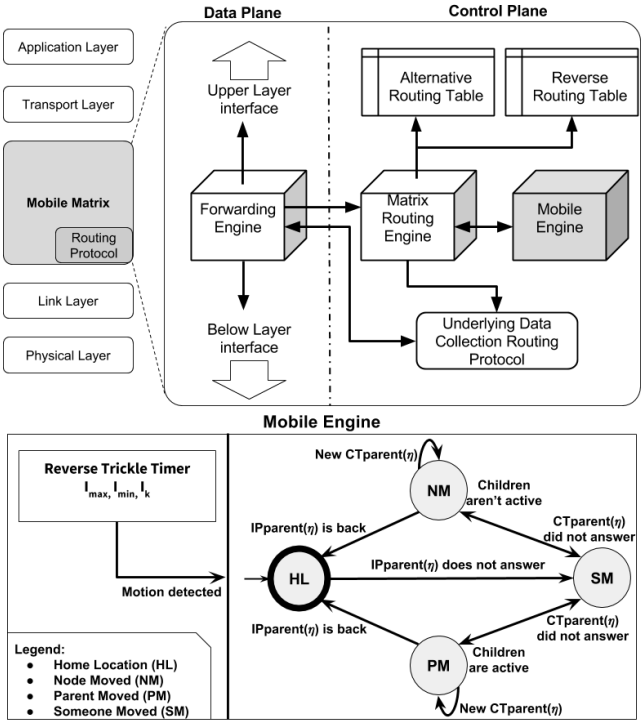


Figure 1: μMatrix protocol's architecture.

2 DESIGN OVERVIEW

The objective of μMatrix is to enable any-to-any communication for static and mobile nodes into 6LoWPANs. μMatrix preserves positive aspects of the Matrix protocol, such as memory efficiency and fault tolerance, while providing a way to manage mobile nodes without ever changing its IPv6 address. μMatrix works at the network layer together with an underlying data collection protocol, such as CTP or RPL. Figure 1 presents the protocol's architecture, which is divided into two planes: i) **Control plane** is responsible for partitioning the address space; distributing and managing the route tables; and handling mobile nodes, ii) **Data plane** is responsible for querying the route tables and packet forwarding.

The μMatrix protocol operation consists of the following phases:

1. Collection tree initialization (Ctree): a collection routing tree is built by an underlying routing protocol (e.g CTP [11] or RPL [13]);
2. Descendants convergecast, IPv6 tree broadcast: once the collection tree is stable, the address hierarchy tree (IPtree) is built using MHCL [16, 17]. The resulting address hierarchy is stored in the distributed IPtree, which initially has the same topology as $Ctree^R$, i.e., in top-down direction.
3. Mobility management: after the properly initialization, μMatrix manages the *RCtree*, a tree that reflects the topology changes caused due to mobility.
4. Standard routing: bottom-up routing follows the Ctree built in phase 1, while top-down routing follows the *IPtree*. Any-to-any routing is done by combining both previous schemes, i.e., a packet can be forwarded bottom-up until a Least Common Ancestor (LCA) between the sender and receiver, and then forwarded top-down until the destination.

2.1 Passive mobility detection

Trickle [14] is an adaptive algorithm to mitigate control message overhead. However, the algorithm lacks in agility and efficiency to detect changes in a highly dynamic network with mobile nodes. To support mobile nodes, we propose a Reverse Trickle timer that operates similarly to the standard algorithm, but in reverse order.

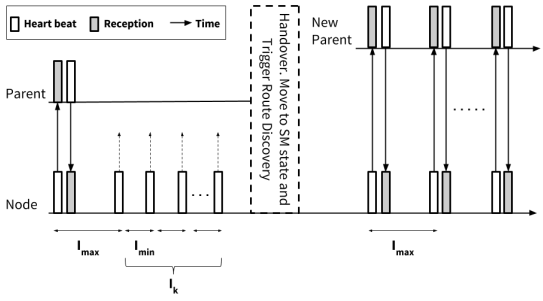


Figure 2: Reserve Trickle timer operation.

Reverse Trickle introduce a control message and three parameters: i) `hasMoved` beacon; ii) I_{max} and I_{min} the maximum and minimum time interval to send a `hasMoved` beacon; iii) I_k the number of attempts to query a node before declaring a inconsistency. These parameters must defined by the network operator before.

Figure 2 illustrates the reverse trickle timer procedure. First, it starts with I_{max} interval between two consecutive `hasMoved` beacons. Then, if the node did not receive an acknowledgment from a `hasMoved` beacon, then it goes to I_{min} interval. After I_k unsuccessful attempts, the node declares a inconsistency and knows that someone moved. Therefore, the node can take actions, for example, properly perform a handover to another parent. Note that by setting the Reverse Trickle parameters, the network operator should consider the intrinsic trade-off between delay to detect that a node has moved and the number `hasMoved` beacons. For instance, for a smaller delay to mobility detection, I_{max} must be tuned to small values at cost of more `hasMoved` beacons. In our experiments (Section 5) reverse trickle parameters were set according to application data rate (Table 1).

In [15], the authors argue that a common modification to support mobility is to change the control message periodicity. The typical approach uses a simple periodic timer or the standardized Trickle timer. While reverse trickle waits for $I_{max} + T_k \times I_{min}$ to detect a topology change, where $I_{min} \ll I_{max}$, the periodic and standardized Trickle approaches wait for at least $2 \times I_{max}$.

2.2 Control Plane

2.2.1 Routing data structures. μMatrix maintains three routing trees structures: i) **Ctree**: a collection tree built by the underlying collection protocol; ii) **IPtree**: an IPv6 hierarchical tree built by MATRIX initialization and kept static afterward, except when new nodes join the network; iii) **RCtree**: a tree reflecting the topology changes caused by node mobility.

Initially, $IPtree = Ctree^R$ and $RCtree = \emptyset$ (see Figures 3(a)(b)). Whenever a topology change occurs due to mobility in one link in $Ctree$, the new link is added into $RCtree$ and maintained as long as the change remain, therefore $RCtree = Ctree^R \setminus IPtree$ (see Figures 3(c)(d)).

$RCtree$ is not really a tree since it contains only reversed links present in $Ctree$ but not int $IPtree$. Nevertheless, its union with links from $IPtree$ is, in fact, a tree, which is used with as possible alternative paths to downward routing.

Each node η keeps the following information in order to build and maintain theses trees:

- $CTparent(\eta)$: the ID of the current parent of a node η in the dynamic collection tree;
- $PRVparent(\eta)$: the ID of η's previous $CTparent(\eta)$.
- $IPparent(\eta)$: the ID of the node that assigned η its IPv6 range initially $CTparent(\eta) = IPparent(\eta)$;
- $Mtable(\eta)$: the temporary alternative routing table for mobility management with IPv6 addresses or ranges;
- $IPchildren(\eta)$: the standard (top-down) routing table with IPv6 ranges for one-hop descendants of η in IPtree;

μMatrix introduce two control messages and one meaningful parameter: i) `keepRoute` beacon: sent by a node when a mobility event occurs, `keepRoute` beacons helps μMatrix to create entries in $Mtable$; ii) δ: time between sending two consecutive `keepRoute` beacons, which is choosen by the network operator; iii) `rmBeacon` beacon: sent by a node to remove inconsistent routing info in $Mtable$ after a mobility event.

In mobile scenarios, entries in a $Mtable$ are created and kept for TTL_{max} (Time To Live) time-slots, where TTL_{max} is a parameter defined by the network operator. The entry is removed unless a `keepRoute` beacon is received from the mobile node. In static scenarios any η node stores only one-hop neighborhood information in $IPparent(\eta)$, so the memory requirement is $O(k)$, where k is the number of node's children. This is better than current state-of-the-art protocols, considering that literature top-down routing mechanism, e.g. RPL, would need at least 1 routing entry for every child in a node sub-tree.

2.2.2 IPv6 multihop host configuration. μMatrix relies on an underlying collection routing protocol to build the $Ctree$. Once the $Ctree$ is stable[1], the address space available to the border router of the 6LoWPAN, for instance, the 64 least-significant bits of the IPv6 address (or a compressed 16-bit representation of the latter), is hierarchically partitioned among nodes in the $Ctree$. The (top-down) address distribution is preceded by a (bottom-up) convergecast phase, in which each node counts the total number of its descendants and propagates it to its parent, thus node knows how many descendants each child has. Such information is required to distribute IP ranges in a fairly way. As result of this procedure is obtained the $IPtree$.

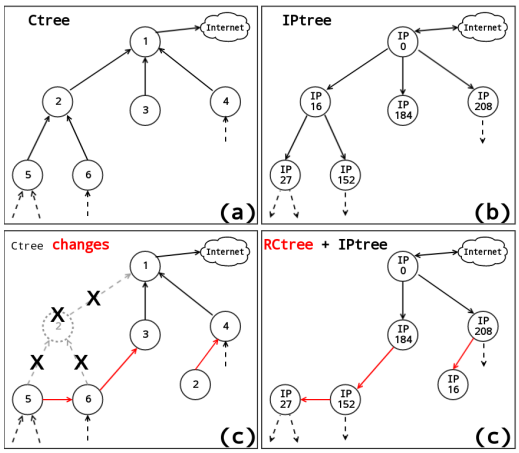


Figure 3: Trees maintained by μMatrix protocol: Ctree, IPtree, and RCtree.

Figure 4 on the left illustrates this process. First, the $Ctree$ is built (upwards arrows), and then, after the $Ctree$ stabilization, the convergecast phase occurs, which allows the nodes to be aware of the size of theirs sub-tree (the percentage next to each node). Finally, the root starts the IP distribution by auto-setting its IP (e.g. the first IP of the available range), reserving a portion of the range for later, and partitioning its range fairly between its children. Each node repeats the IP distribution process.

2.2.3 Mobility management. Once host configuration was already done, the Mobile Engine starts working, this allows nodes to move around the 6LoWPAN. The μMatrix Engine uses a finite-state machine (Figure 1 rightmost) with 4 states. Each node can be in one of these states depending on its previous condition and on the present knowledge about the node neighborhood. The adaptive beaconing mechanism (Reverse Trickle Timer described in Section 2.1) helps Mobile Engine to maintain in which state a node η is. In the following, we present the Mobile Matrix operation.

Algorithm 1 shows in a simplified pseudocode of Mobile Engine routine. After μMatrix initialization, Mobile Engine starts with Home Location as the initial state. In this case, the module only starts the adaptive beaconing (line 3) with its $IPparent(n)$ in order to track movements. Whenever the reverse trickle detects a motion, it sends a Someone Moved (SM) state to the Mobile Engine.

[1] A node is stable if it reaches k times the maximum maintenance beacon period of $Ctree$ protocol without changing its parent. Trickle [14] was used as beacon scheme.

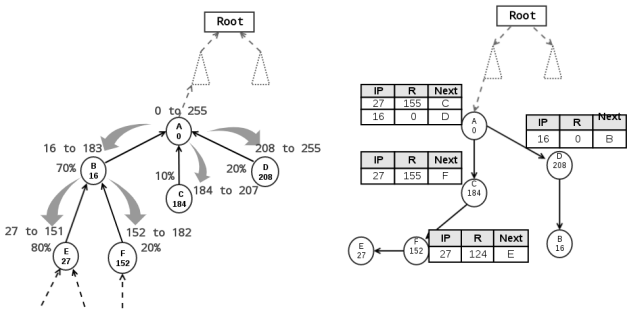


Figure 4: Simplified hierarchical address assignment with 8-bit available address space for an in-network node and 6.25 % of address reserve for delayed nodes. In the right, *Mtable* after B moves.

On the SM state, a node is aware that someone moved, but it does not know who moved (if it was itself or its current parent). Mobile Engine triggers a routine to infer who moved (line 17). This information is essential to take the right decision in mobility management, perform quickly a handover to another *CTparent* and rebuild the routing structures.

There are, at last, two ways to a node automatically find out who moved. The first one, a node can actively query its children (if it has anyone), if no one answer then the node moved, otherwise the parent moved. The second one, a node waits for a period (e.g. one I_{max}) to receives queries (heartbeats) from its children and then infer who moved. As soon as the node identify who moved, it must moves towards NM or PM state according to if the node or the parent moved respectively. We used the second one in our implementation.

If a node μ moves, $findWhoMoved()$ (line 17) should inform to Mobile Engine the state Node Moved (NM) as the newest state. In this case, some actions are taken (see lines 19 to 32). First, all entries in $Mtable(\mu)$ must be cleaned, because they must be outdated since μ moves. Then, μ waits for a new parent. When μ is attached again to the network, the Reverse Trickle is restarted, but now with $CTparent(\mu)$ (after the movement). Also, *startKR(...)* routine start sending `keepRoute` beacons (line 30) addressed to $IPparent(\mu)$ at a rate of δ s. The `keepRoute` beacon contains as payload only μ's IP. These beacons help to create entries in *Mtable* for nodes only on the shortest path between $CTparent_f(\mu)$ and $IPparent(\mu)$. For instance, in Figure 4, after B (IP 16) moves, it goes to NM state, then entries only in A and D are created to reflect B's new position.

As long as μ keeps moving and being attached to new parents, μ remains in NM state (line 24) performing handovers. Whenever, μ finds another *new parent*, μ also sends a `rmRoute` beacon to its previous $PRVparent(\eta)$ (line 26) in order to quickly remove inconsistent temporary routes in *Mtable* for every node comprised in the shortest path between $CTparent(\mu)$ and $PRVparent(\mu)$. When $new\,parent = IPparent(\mu)$, then μ returned to home position in *IPtree*, this trigger a $rmRoutine(...)$ (line 8) to remove inconsistent entries in *Mtable* of nodes comprised in the shortest path between $IPparent(\mu)$ and $PRVparent(\mu)$.

Algorithm 1: Mobile Engine

```
(1)  Func Main(newState):
(2)      if newState = HL then
(3)          startReverseTrickleTimer(IPparent(n));
(4)          if PRVparent(n) ≠ ∅ then
(5)              rmRoute(PRVparent(n), {n.IP});
(6)              PRVparent(n) = CTparent(n);
(7)          end
(8)          if previousState = NM then
(9)              rmRoute(PRVparent(n), {n.IP});
(10)             stopKR();
(11)         else if previousState = PM then
(12)             rmRoute(PRVparent(n), {n.IP, n.Range});
(13)             stopKR();
(14)         end
(15)     end
(16)     if newState = SM then
(17)         findWhoMoved();
(18)     end
(19)     if newState = NM then
(20)         cleanMtable();
(21)         while findNewCTparent() = NULL do
(22)             triggerNewParent();
(23)         end
(24)         if newParent() ≠ IPparent(n) then
(25)             if PRVparent(n) ≠ {∅} and
                   newParent() ≠ PRVparent(n) then
(26)                 rmRoute(PRVparent(n), {n.IP});
(27)                 PRVparent(n) = CTparent(n);
(28)             end
(29)             restartReverseTricklerTimer(CTparent(n));
(30)             startKR(IPparent(n), {n.IP}, δ);
(31)         end
(32)     end
(33)     if newState = PM then
(34)         while findNewCTparent() = NULL do
(35)             triggerNewParent();
(36)         end
(37)         if newParent() ≠ IPparent(n) then
(38)             if IPchildren(n) ≠ {∅} then
(39)                 startKR(n.IP, {n.IP, n.Range}, δ);
(40)             else
(41)                 startKR(n.IP, {n.IP}, δ);
(42)             end
(43)             if PRVparent(n) ≠ {∅} and
                   newParent() ≠ PRVparent(n) then
(44)                 rmRoute(PRVparent(n), {n.IP});
(45)                 PRVparent(n) = CTparent(n);
(46)             end
(47)             restartReverseTricklerTimer(CTparent(n));
(48)         end
(49)     end
(50) return
(51) Func startKR(dest, payload, f):
(52)     foreach δ time-slots do
(53)         sendBeacon(dest, payload, type.KeepRoute)
(54)     end
(55) return
(56) Func rmRotine(dest, payload):
(57)     sendBeacon(dest, payload, type.rmRoute)
(58) return
```

A node η goes to Parent Moved (PM) state when its *IPparent* moves. This case, Mobile Engine triggers the underlying route discovery mechanism. Then, as long as $IPparent(\eta)$ is away from its location, η remains in PM state sending periodically `keepRoute` beacons, in which contains η IP and range if $IPchildren(\eta) \neq \{\emptyset\}$, and only η IP otherwise (see lines 37 to 48). Note that, we use the $\eta.IP$ as

destination in *setartKeepRoute*() routine, this is a trick to routing messages towards η's IPgrandparent, IPgreatgrandparent etc. As η does not keep any information for more than one-hop, if η sends beacons for its own IP to $CTparent(\eta)$, then the beacons will flow upwards until $LCA(CTparent(\eta), IPparent(...IPparent(IPparent(\eta))...)$ and then downwards to some ancestors of η (see Section 2.3 to understand any-to-any routing). When $IPparent(\eta)$ returns to home location in $IPtree$, η must trigger $rmRoute(...)$ routine (line 11) to remove inconsistent states in $Mtable$ of nodes comprised in the shortest path between $IPparent(\eta)$ and $PCTparent(\eta)$.

In Figure 4, when B moves, E and F go to PM state. Then, only one entry in F, C and A need to be created to rebuild the path for the whole sub-trees of E and F. Note that, as E and F have contiguous range, then A and C can aggregate them into one entry.

2.2.4 Loop avoidance. Handle loops is a hard task for μMatrix since it relies directly on underlying collection protocol. Such protocols are not loop-free, but they can identify them and try to recover from it [11, 13]. μMatrix identify loops when a control message is received more than once in a too short time. But, what is a short time? Each entry in the *Mtable* has a Time Has Lived (THL) field, therefore, if a node receives a duplicate control message in a time $THL < t \ll \delta$, where t is defined by the network operator, then it should be a loop. We set $t = 1$ s. When loops happen, then the node suppress the beacon. Also, there is in `keepRoute` beacon a Time To Live (TTL) field. Thus, μMatrix uses both THL and TTL to remove inconsistent routes and messages from the network.

2.3 Data Plane: any-to-any routing

The Forwarding Engine (see Figure 1) is responsible for data forwarding. Any-to-any routing is performed by combining bottom-up forwarding, until the LCA between the sender and receiver, and then top-down forwarding to the destination. Upon receiving a data packet, each node η checks whether the destination matches with an entry $e \in Mtable(\eta)$: if yes then the packet is forwarded according, otherwise, η checks if the destination fall within some range in $r \in IPchildren(\eta)$, if yes the packet is forwarded downwards properly. Finally, if the previous attempts fail, then the packet is forwarded (upwards) to $CTparent(\eta)$.

3 COMPLEXITY ANALYSIS

For the formal analysis, we assume a synchronous communication message-passing model with no faults. Thus, all nodes start executing the algorithm simultaneously and the time is divided into synchronous rounds, i.e., when a message is sent from node v to its neighbor u in time-slot t, it must arrive at u before time-slot $t + 1$, and $d(v, u)$ is the shortest path length between v and u in $Ctree \cup IPtree \cup RCtree$. The performance of μMatrix in faulty scenarios is analyzed through simulations in Section 5.

3.1 Memory footprint

As described in Section 2, the temporary routing information needed to manage mobility is stored in the *Mtable* data structure of some nodes. Each entry is kept for at most TTL_{max} seconds, a time interval pre-configured by the network operator, and is deleted unless a `keepRoute` beacon is received. In the following theorem, we bound the total number of *Mtable* entries in the network, necessary to manage routing of each mobile node $\mu \in CTree$.

Theorem 3.1. *The memory footprint to manage the mobility of one node $\mu \in Ctree$ with μMatrix is $\mathcal{M}(\mu) = O(depth(Ctree))$.*

Proof. Consider a node $\mu \in Ctree$ that has moved from its home location in time-slot t_0 and returned in time-slot t_f. Consider the (permanent) $IPparent(\mu)$ and the (temporary) $CTparent_i(\mu)$ in time-slot $t_0 < t_i < t_f$. A routing entry for the temporary location of μ will be stored in the *Mtable* of every node comprising the shortest path between $IPparent(\mu)$ and $CTparent_i(\mu)$. Moreover, if μ has descendants in the IPtree, i.e, $k(\mu) = |IPchildren(\mu)| > 0$, then each node $c \in IPchilde(\mu)$ will send a temporary (bi-directional) route request to their respective $CTparent_i(c)$, and a (temporary) routing entry will be stored in the Mtable of every node comprising the shortest path between $CTparent_i(c)$ and $IPparent(\mu)$. Therefore, the total memory footprint to manage the mobility of a node μ is:

$$\begin{aligned}
\mathcal{M}(\mu) &= d(CTparent_i(\mu), IPparent(\mu)) + 1 \\
&+ \sum_{c \in IPchildren(\mu)} (d(CTparent_i(c), IPparent(\mu)) + 1) \\
&\leq (k(\mu) + 1) \times (depth(Ctree) + 1) \\
&= O(depth(Ctree)) \quad \square
\end{aligned}$$

Theorem 3.1 implies that the total memory footprint to manage the mobility of m nodes is $O(m \times depth(Ctree))$. Note that μMatrix preserves locality when managing mobile routing information of a node μ, since no *Mtable* needs to be updated at nodes above the $LCA(IPparent(\mu), CTparent(\mu))$.

3.2 Control message overhead

Control messages used by μMatrix are comprised of three types: (1) those used by Matrix to set up the initial IPtree and address allocation; (2) `hasMoved` beacons, defined in Section 2.1; and (3) `keepRoute` beacons, defined in Section 2.2.1.

For any network of size n with a spanning collection tree Ctree rooted at node r, the message and time complexity of Matrix protocol in the address allocation phase is $\mathcal{M}sg(Matrix^{IP}(Ctree)) = O(n)$ and $\mathcal{T}(Matrix^{IP}(Ctree)) = O(depth(Ctree))$, respectively, which is asymptotically optimal, as proved in [17]. Next we bound the number of control messages of type (2) and (3).

Theorem 3.2. *Consider a network with n nodes, with a spanning collection tree $Ctree$ rooted at node r, and m mobility events, consisting of m nodes μ_i, changing location during time intervals $\Delta_i \leq \Delta$ time-slots. Moreover, consider the* `hasMoved` *beacon parameters I_{min}, I_{max} and I_k and the* `keepRoute` *beacon interval of δ time-slots. The control message complexity of μMatrix to perform routing under mobility of m nodes is*

$$\begin{aligned}
\mathcal{M}sg(\mu Matrix(Ctree)) &= O\left(\frac{m \times I_k}{I_{min}} + \frac{n}{I_{max}}\right) \\
&+ O\left(\frac{m \times \Delta}{\delta} depth(Ctree)\right).
\end{aligned}$$

Proof. Firstly, we bound the number of `hasMoved` beacons, which are sent periodically by all nodes in order to detect mobility events. As described in Section 2.1, when there is no mobility, the

periodicity of hasMoved beacons is $1/I_{max}$. If some node μ has moved (an ack is lost), then I_k messages are sent in intervals of I_{min} time-slots. Using the fact that the network is a tree and the number of edges is $O(n)$, this gives a total of messages

$$Msg(\mu Matrix^{\mathsf{hM}}(Ctree)) = O\left(\frac{m \times I_k}{I_{min}} + \frac{n}{I_{max}}\right).$$

Now, we bound the number of keepRoute beacons. As described in Section 2, mobile nodes send periodic keepRoute beacons at a frequency of δ to keep the $Mtables$ up-to-date. Consider a node $\mu \in Ctree$ that has moved from its home location in time-slot t_0 and returned in time-slot t_f. Consider the $IPparent(\mu)$, $CTparent_i(\mu)$ in time-slot $t_0 < t_i < t_f$, and $\Delta = t_f - t_0$. When μ is attached to a $CTparent_i(\mu)$, μ sends keepRoute beacons at a rate of δ for at most Δ time-slots, such beacons travel the shortest path $|(CTparent_i(\mu), IPparent(\mu))| \leq 2 \times depth(Ctree)$. Furthermore, if μ has descendants, i.e., $k(\mu) = |IPchildren(\mu)| > 0$, then each node $c \in IPchildren(\mu)$ will also send keepRoute beacons at a rate of δ for at most Δ time-slots, such beacons will travel the shortest path $|(CTparent_i(c), IPparent(\mu))| \leq 2 \times depth(Ctree)$. Therefore, the total control overhead to manage the mobility of a node μ is $\leq 2 \times depth(Ctree)(k(\mu)+1)\Delta/\delta$, which results in

$$\mathcal{M}sg(\mu Matrix^{\mathsf{kR}}(Ctree)) = O\left(\frac{m \times \Delta}{\delta} depth(Ctree)\right).$$

Finally, the total control overhead is bounded by:

$$\mathcal{M}sg(\mu Matrix) = \mathcal{M}sg(\mu Matrix^{\mathsf{hM}}) + \mathcal{M}sg(\mu Matrix^{\mathsf{kR}}) \quad \square$$

Once again μMatrix preserves locality when managing mobile routing state of a node μ, since no messsage needs to be sent to nodes above the $LCA(IPparent(\mu), CTparent(\mu))$.

3.3 Routing path distortion

We analyze the route length of messages, addressed to mobile nodes. Consider the underlying collection protocol (e.g. CTP or RPL), which dynamically optimizes the (bottom-up, or upwards) links in the collection tree $CTree$, according some metric, such as ETX. We define an *optimal route* length as the length of the shortest path between (s, d), comprised of the upwards links of the collection tree $CTree$ and the downwards links of the union of the IPv6 address tree and the reverse-collection tree, i.e., $IPtree \cup RCtree$.

Theorem 3.3. *μMatrix presents optimal path distortion under mobility, i.e., all messages are routed along shortest paths towards mobile destination nodes.*

Proof. Consider a mobile node $\mu \in CTree$, which has moved from its home location in time-slot t_0. Messages addressed to μ and originated by some node $\eta \in Ctree$ in time-slot $t_i > t_0$ can belong to traffic flows of three kinds: (1) bottom-up: $LCA_i(\mu, \eta) = \mu$; (2) top-down: $LCA_i(\mu, \eta) = \eta$; and (3) any-to-any: $LCA_i(\mu, \eta) \neq \mu \neq \eta$. In case (1), messages are forwarded using the underlying collection protocol, using the upwards links of the collection tree CTree, which is optimal. In case (2), messages are forwarded using Mtables of η and its descendents, until reaching the mobile location of μ in some time-slot $t_f > t_0$. This path is comprised of the downwards links of $IPtree \cup RCtree$ in time-slot $t_0 < t_i \leq t_f$, which is the optimal route from η to the mobile location of μ in that time-slot. In case (3), the route between η and $LCA_i(\mu, \eta)$ falls into the case (1) and the route between $LCA_j(\mu, \eta)$ and μ falls into the case (2), for some $t_0 < t_i \leq t_j \leq t_f$, which is optimal. $\square$

4 CRWP MOBILITY MODEL

Here, we propose the Cyclical Random Waypoint Mobility Model (CRWP), a mobility model based on the Random Waypoint [2]. CRWP is useful to model scenarios where some of the entities move to different destinations and eventually they return to their initial positions. This is the case of people and their portable devices in offices, universities, hospitals, factories, etc.

In CRWP, the entities move independently to random destinations and speeds as in RWP. When an entity arrives at the destination, it stops for a given time T_{pause}. A difference in CRWP is that after n destinations are chosen, the mobile entity returns to its initial position. Besides that, only k % of mobile entities are outside of their initial position in each instant of time. CRWP has four parameters: i) *PerMobNodes:* maximum percentage of entities that are mobile in each instant of time; ii) *Stops:* number of stops that the mobile entity do before returning to its original position; iii) *Speed:* speed which the mobile entity moves; iv) T_{pause}*:* the amount of time that the entity stays in a destination position.

5 SIMULATION RESULTS

Table 1: Simulation parameters

Simulation parameter	Values		
Simulation time	1.5 h		
# Nodes	1 center root, 100 nodes in grid		
Mobility Model	CRWP		
Application data packets	20 pkt/node, Rate = 1 pkt/min		
Radio environment	50 m UDGM constant loss		
Area of deployment	400 m ×400 m		
Reverse Trickle	I_{max} = 60 s, I_{min} = 1 s, I_k = 3		
RPL Trickle	I_{max} = 60 s		
keepRoute beaconing period	δ = 60 s		
$Mtable$	TTL_{max} = 90 s, Size = 20 entries		
RPL downwards table	Size = 20 entries		
# mobility traces	10 traces/scenario		
Number of experiments	10 runs/trace		
Node Speed	constant 4 m/s		
T_{pause}	constant 300 s		
# node stops	Uniform Dist. in [1, 3] stops		
	Low	**Moderate**	**High**
PerMobNode	5%	10%	15%

μMatrix was implemented as a subroutine of collection protocol available in ContikiOS [6] and the experiments were simulated on Cooja [8]. We compare μMatrix with ContikiOS' RPL implementation. We use the BonnMotion [1] to implement CRWP as well as to generate and analyze mobility traces. We simulated four different scenarios. The first scenario represents the static network, in which nodes do not move. The remaining represent mobility scenarios named low, moderate, and high with mobile nodes. Table 1 lists the default simulation parameters used for each scenario.

On top of the network layer, we ran an application, in which each node sends 20 data packets to the root. Upon receiving a data packet, the root confirms to the sender with an ack packet that

has the size of a data packet. The application waits for 10 min for protocols initialization and stabilization before it starts sending data. The nodes start sending their data in a simulation time randomly chosen in (10, 20] min. The mobility traces were configured to start after the stabilization time. Additionally, we generate 10 mobility traces for each scenario. Each trace and the static scenario were run 10 times, totaling 3010 runs. In each plot, the bars represent the average, the error bars the confidence interval of 95 %, and the curves the maximum table usage for a given mobility scenario.

Table 2: Mobility Metrics

Mobility Metrics	Low Mob. sce.	Mod. Mob. sce.	High Mob. sce.
Avg. Link Breaks	1621	3057	4838
Avg. Link duration	761.90	457.4	345
Avg. Degree	4.12	4.36	4.44
Avg. Time to link break	227.6	216.1	204.5

Mobility scenario: We simulated a scenario, where $n = 100$ people are assumed to be in an office and can move around and return to a predefined home position. This scenario is expected to present relatively low mobility, thus in our set up k % of the nodes are moving at any moment in time, where $k \in \{5\%, 10\%, 15\%\}$. Table 2 presents some mobility metrics [1] for each scenario. We highlight that link breaks play a key role in the performance of the network protocol, note that high mobility scenario presents up to 20 % more topology changes than in low mobility. As expected, the average link duration decrease when *PerMobNode* increases. The averages of node degrees and the time to a link break do not show much variability, they reflect the simulation parameters, where the node deployment is a grid and time for a link to break is less than T_{pause}.

5.1 Results

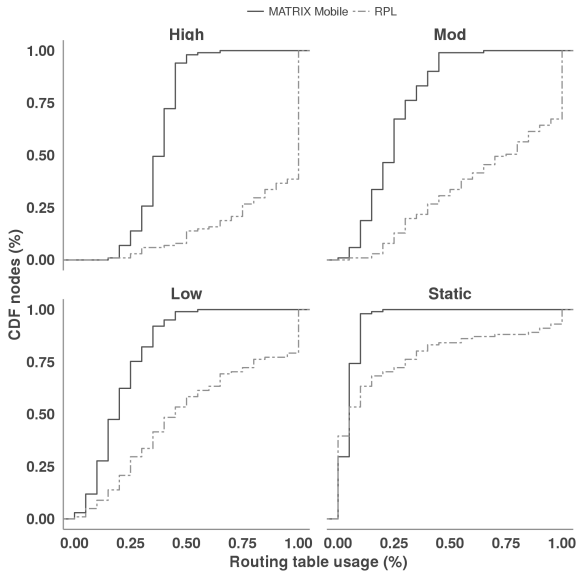


Figure 5: CDF of routing table usage. For μMatrix *Mtable + IPchildren*, for RPL only downwards routing table. The maximum table size is 20.

In Figure 5, we show the Cumulative Distribution Functions (CDFs) of the percentage of downward routing table usage among nodes for given mobility scenario. Note that in static scenario, for μMatrix all nodes use up to 25% of available downwards route entries, while for RPL < 75% of nodes use up to 25% of entries, indeed, for some RPL nodes, 100% of table entries are used. Usually, those nodes that use more memory are located near to the root and they play a key role in top-down routing. If they have a full downward routing table, then the traffic pattern top-down suffers from poor reliability, and some nodes may be unreachable. In mobility scenarios, μMatrix also presents more efficient memory footprint, the difference grows up in high mobility scenarios, where > 50% of RPL nodes have all table entries busy, while μMatrix nodes use at most 70% of downwards available routes.

Figure 6(a) shows the amount of control traffic overhead of the protocols (the total number of beacons sent during the entire simulation). RPL sends fewer control packets than μMatrix, but the difference between them does not exceed 7.4%. μMatrix sends more beacons in order to quickly react to topology changes. Most of the μMatrix beacons are fired by Reverse Trickle, which can be configured to reduce the sending beacons. Note that the adjustment reverse trickle faces a trade-off between fast mobility discovery and control overhead. In Table 1, we set I_{max} of RPL and μMatrix evenly and close to data packet rate. This gives the protocols the opportunity to identify topology changes and react upon them. In the following, we show the protocol performance for data delivery.

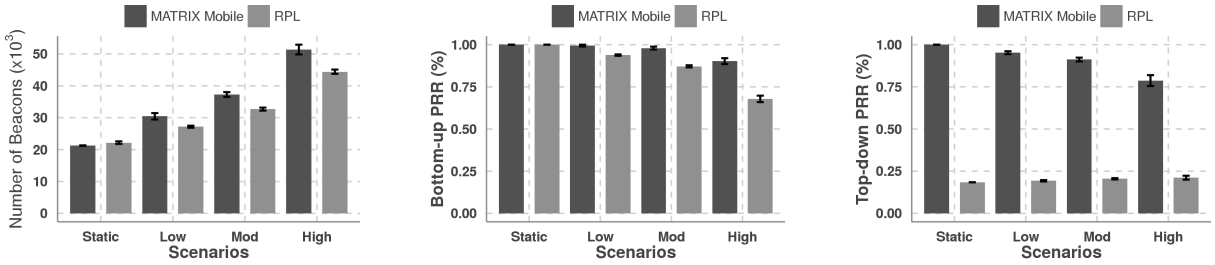
Packets Reception Rate (PRR) is a metric of network reliability. It is computed as the number of packets received successfully over all packets sent. Figure 6(b) shows the PRR in bottom-up data traffic. In all scenarios, μMatrix presents higher PRR rate than RPL. When μMatrix realizes that a topological change happened, it quickly triggers the underlying route discovery, consequently bottom-up routes are quickly rebuilt and the reliability increases.

Figure 6(c) shows the PRR for top-down data traffic. We can see that, when there is no mobility, μMatrix presents 99.9% of success rate. In mobility scenarios, μMatrix PRR decreases slowly when more mobility is allowed. In the harshest mobility scenario, the PRR > 75%. RPL, on the other hand, suffer from poor reliability, delivering < 21.1% in all simulated scenarios, which occurs due the lack of memory (see Figure 5) to store top-down routes.

6 RELATED WORK

In the world of tiny (IoT) several mobility-enabling routing protocols have been proposed. Firstly we highlight μMatrix's features against its original static version [17]. Then, we survey recent protocols in the context of 6LoWPAN and put them in perspective with μMatrix.

Matrix was originally proposed without support for mobility[17]. If a node moved from its home location, the hierarchical IPv6 address allocation would become invalid and compromise downward routing.Although RPL [20] is the standard protocol for 6LoWPANs, it presents limitations, for example, in mobility scenarios, scalability issues, reliability and robustness for point-to-multipoint traffic [12, 17]. Most recent mobile-enabled routing protocols are RPL extensions. They deal with mobile issues, but they do not handle RPL drawbacks. Co-RPL [10] provides mobility support to RPL



(a) Number of control packets (b) Bottom-up routing success rate. (c) Top-down routing success rate.

Figure 6: Simulation experiments

but without Trickle. This turns Co-RPL more responsive but has higher overhead. MMRPL [4] modifies the RPL beacon periodicity by replacing the Trickle mechanism with a Reverse Trickle-Like. Their Reverse Trickle decays exponentially, while our approach quickly goes to the minimum after an unacknowledged beacon. MMRPL also needs some static nodes. In ME-RPL [7], static nodes have higher priority than mobile ones. When a node is detached from a parent, it sends DIS messages in dynamic intervals. ME-RPL requires some fixed nodes. The memory requirement to downward routes is still prohibitive. mRPL [9] proposes a hand-off mechanism for mobile nodes in RPL by separating nodes into mobile (MN) or serving access point (AP). They use smart-HOP algorithm on MN nodes to perform hand-off between AP.

XCTP [19] extends CTP to support bidirectional traffic. XCTP does not support IPv6 addressing and any-to-any traffic. Hydro [5] fills the gap of any-to-any traffic, but it requires static nodes with a large memory to perform the routing and support mobility nodes.

Mobile IP [18] and Hierarchical Mobile IPv6 (HMIPv6) Mobility Management [3] are standards for IPv6 networks for handling local mobility. However, they are not designed for 6LoWPANs, they do not present a mobility detection or adjustable timers.

Table 3 summarizes properties of the related protocols.

Table 3: Routing protocol properties

Feature	μMatrix	RPL	Co-RPL	MMRPL	ME-RPL	mRPL	DMR	Hydro	XCTP
Bottom-p	✔	✔	✔	✔	✔	✔	✔	✔	✔
Top-down	✔	✔	✔	✔	✔	✔		✔	✔
Any-to-any	✔	✔	✔	✔	✔	✔		✔	
Address Allocation	✔								
IPv6 support	✔	✔	✔	✔	✔	✔	✔		
Memory efficiency	✔								
Fault Tollerance	✔								✔
Local Repair	✔								
Topological changes	Reverse Trickle	Trickle	Periodic fixed	Reverse Trickle-like	Trickle	Trickle	Trickle	Periodic fixed	Trickle
Constraints	Nodes should return to home location			Need static nodes	Need static nodes	Need static nodes	Need static nodes	Need static nodes	

7 CONCLUSIONS

In this work, we presented μMatrix: a memory efficient routing protocol for 6LoWPAN that performs any-to-any routing, hierarchical address allocation and mobility management. As a building block of μMatrix, we proposed a passive mobility detection mechanism that captures topological changes without requiring additional hardware. Finally, we introduced the CRWP, a mobility model suited for scenarios with mobile nodes that have cyclical movement patterns. As future work we plan to run experiments with physical devices and extend experimental evaluation to more mobile models, such as faulty communications scenarios.

Acknowledgements: We thank CAPES, CNPq and FAPEMIG.

REFERENCES

[1] Nils Aschenbruck, Raphael Ernst, Elmar Gerhards-Padilla, and Matthias Schwamborn. 2010. BonnMotion: a mobility scenario generation and analysis tool. In *EAI ICST*. 51.
[2] Fan Bai and Ahmed Helmy. 2004. A survey of mobility models. *Wireless Adhoc Networks* (2004).
[3] Ludovic Bellier, Karim El Malki, Claude Castelluccia, and Hesham Soliman. 2008. Hierarchical Mobile IPv6 (HMIPv6) Mobility Management. RFC 5380. (2008).
[4] Cosmin Cobarzan, Julien Montavont, and Thomas Noel. 2014. Analysis and performance evaluation of RPL under mobility. In *IEEE ISCC*. 1–6.
[5] S. Dawson-Haggerty, A. Tavakoli, and D. Culler. 2010. Hydro: A hybrid routing protocol for low-power and lossy networks. In *IEEE SmartGridComm*. 268–273.
[6] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. 2004. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *IEEE LCN*. 455–462.
[7] Inès El Korbi, Mohamed Ben Brahim, Cedric Adjih, and Leila Azouz Saidane. 2012. Mobility enhanced RPL for wireless sensor networks. In *IEEE ICUFN*. 1–8.
[8] Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, and Pedro José Marrón. 2009. COOJA/MSPSim: Interoperability Testing for Wireless Sensor Networks. In *Simutools'09*. 1–27.
[9] Hossein Fotouhi, Daniel Moreira, and Mário Alves. 2015. mRPL: Boosting mobility in the Internet of Things. (2015), 17–35 pages.
[10] Olfa Gaddour, Anis Koubâa, Raghuraman Rangarajan, Omar Cheikhrouhou, Eduardo Tovar, and Mohamed Abid. 2014. Co-RPL: RPL routing for mobile low power wireless sensor networks using Corona mechanism. In *IEEE SIES*.
[11] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, Maria Kazandjieva, David Moss, and Philip Levis. 2013. CTP: An efficient, robust, and reliable collection tree protocol for wireless sensor networks. *ACM TOSN* 10, 1 (2013), 16.
[12] O. Iova, P. Picco, T. Istomin, and C. Kiraly. 2016. RPL: The Routing Standard for the Internet of Things... Or Is It? *IEEE Communications Magazine* 54 (December 2016), 16–22.
[13] Kevin C Lee, Raghuram Sudhaakar, Lillian Dai, Sateesh Addepalli, and Mario Gerla. 2012. RPL under mobility. In *IEEE CCNC*. IEEE, 300–304.
[14] Philip Levis, Neil Patel, David Culler, and Scott Shenker. 2004. Trickle: A Self-regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *USENIX NSDI*. 2–2.
[15] Afonso Oliveira and Teresa Vazão. 2016. Low-power and lossy networks under mobility: A survey. *Computer Networks* (2016).
[16] Bruna Peres and Olga Goussevskaia. 2016. MHCL: IPv6 Multihop Host Configuration for Low-Power Wireless Networks. *arXiv:1606.02674* (2016).
[17] Bruna S. Peres, Otavio A. de O. Souza, Bruno P. Santos, Edson R. Araujo Junior, Olga Goussevskaia, Marcos A. M. Vieira, Luiz F. M. Vieira, and Antonio A. F. Loureiro. 2016. Matrix: Multihop Address Allocation and Dynamic Any-to-Any Routing for 6LoWPAN. In *ACM MSWiM*. 302–309.
[18] Charles Perkins, David Johnson, and Jari Arkko. 2011. Mobility support in IPv6. (2011).
[19] Bruno P Santos, Marcos AM Vieira, and Luiz FM Vieira. 2015. eXtend collection tree protocol. In *IEEE WCNC*. 1512–1517.
[20] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander. 2012. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550. (2012).