The PL-Gossip Algorithm

Konrad Iwanicki*† and Maarten van Steen*
*Vrije Universiteit, Amsterdam, The Netherlands
†Development Laboratories (DevLab), Eindhoven, The Netherlands
{iwanicki, steen}@few.vu.nl

Technical Report IR-CS-034
Vrije Universiteit Amsterdam, March 2007

ABSTRACT

Many recently proposed sensornet applications require large number of sensor nodes operating over long periods of time. In contrast to the first-generation sensornet deployments, these applications involve sophisticated internode communication rather than simple tree-based data collection. The examples include network maintenance, data-centric storage, object tracking, and various query engines.

If these proposals for next-generation applications are ever to become reality, we will need solutions for self-organization of very large networks. We argue that these applications need methods for organizing nodes into recursive geometric structures, for example, proximity-based hierarchies. Such structures should provide naming that facilitates amongst others, routing, multicasting, and data aggregation and fusion.

This paper presents a novel algorithm for dynamically organizing nodes in a sensornet into an area hierarchy. The algorithm employs gossiping, guaranteeing predictable maintenance traffic, which is a crucial property when it comes to energy conservation. Simulations show that the algorithm scales to large networks, works well in the presence of message loss and network dynamics, and has low bandwidth and memory requirements.

Faculty of Sciences
Vrije Universiteit Amsterdam
De Boelelaan 1081A
1081 HV Amsterdam
The Netherlands

vrije Universiteit amsterdam

Contents

1 Introduction

Many existing and novel wireless sensor network (sensornet) applications, like habitat monitoring [1], vehicle tracking [2], and border protection [3], require large numbers of sensor nodes operating over long periods of time. The effort involved in the deployment and durable maintenance of such networks can be brought down if the nodes are able to *autonomously* organize into a required logical network structure.

A typical way of self-organizing the network in the first-generation sensornet applications was having nodes build and maintain a tree rooted at a base station [4, 5]. This approach provides one-to-many and many-to-one routing primitives, which enable the base station to broadcast commands and collect data, possibly with various forms of aggregation along the collection path.

However, a rapidly growing number of compelling sensornet applications requires a much more sophisticated, geometry-based network organization. Examples of such applications include data-centric storage [6], reactive tasking [7] (based on local observations, sensor nodes trigger actuator nodes), object tracking [8, 2], network debugging [9], and various query engines, like multi-dimensional range queries [10], spatial range queries [11], or multi-resolution queries [12]. The required organization for these emerging systems is based on node proximity and connectivity. It provides scalable recursive naming of network areas, that is, we can name a network area, the subareas of this area, and so forth. Moreover, the structure enables routing between any of such areas or between any pair of nodes.

Devising a protocol in which nodes autonomously build and maintain such an organization poses a number of challenges. The combination of a possibly large network size and a very short radio range of sensor nodes leads to high-diameter multi-hop topologies. Because of the memory and bandwidth limitations of individual nodes, the state stored by each of them must scale gracefully with the network size. To enable predicting the network lifetime and provisioning the battery power accordingly, the protocol must ensure predictable maintenance traffic. Yet, the maintenance must guarantee adaptability to network dynamics. For practical reasons, nodes should be able to build and maintain the organization in many heterogeneous settings, ranging from "planar" regions (e.g., parking lots), to "volumetric" deployments (e.g., interiors of multi-story buildings). To the best of our knowledge, none of the existing solutions meets all of the above goals.

In this paper, we make a single contribution by presenting a sensornet protocol for the self-management of nodes into a recursive geometry-based organization, known as an area hierarchy [13]. Our protocol, dubbed PL-Gossip[1], meets the aforementioned requirements. In particular, we show that it is scalable, adapts the organization to system dynamics, and, very important, exhibits predictable maintenance traffic. Moreover, the hierarchy offers efficient point-to-point routing and multicasting, even while a number of nodes are in the process of organizing themselves.

The rest of the paper is organized as follows. We begin with surveying related work in Section 2. In Section 3, we define our area hierarchy employed by PL-Gossip, and in Section 4, we explain naming and routing in this hierarchy. The core of our contribution, the PL-Gossip maintenance algorithm, is presented in Section 5 and evaluated using simulations in Section 6. Finally, in Section 7 we conclude.

2 Context and Related Work

The research described in this paper is a result of our collaboration with a large Dutch consortium building a real-world $10,000$-node sensornet.[2] Maintenance of this large multi-hop network requires the nodes to be organized in a scalable, *recursive* fashion taking the geometric proximity and connectivity between nodes into account. This implies that the parts of the network are structured according to the same rules as the whole. Such an organization is needed, among others, to diagnose problems occurring in particular network areas or to aggregate multi-resolution statistics on various regions.

There are essentially three techniques for providing a recursive, geometry-based network organization: *geographic coordinates*, *graph embedding*, and *area hierarchies*.

In the first approach, each node is identified by its geographic coordinates. An area covering a group of nodes is identified by vertices of a polygon encapsulating these nodes [11]. Naming is based on geographic coordinates and is simple. Structure maintenance requires a small, constant state per node. However, practical, efficient routing is essentially still an open research question. Assuming that nodes are deployed on a planar surface, normally, the routing is done in a greedy way: at each step, nodes pick as the next hop the neighbor that is closest to the destination. Problems arise when a node has no neighbor that is closer to the destination. In that case, most of the solutions employ *face routing* [14, 15, 16, 17], which requires the maintenance protocol to planarize the neighborhood graph. For the planarization to work in practice [18], the maintenance algorithm runs a *cross-link detection protocol* (CLDP) [19], which probes the actual internode connectivity and removes links violating the graph planarity using two-phase locking to ensure consistency. While this solves the planarization problem, it makes the maintenance protocol complex and somewhat costly [20]. Also, face routing requires handling

[1] The name "PL-Gossip" is an abbreviaton for "Polish Gossip" (K. Iwanicki is Polish while "PL" is the European Union abbreviation and the internet domain of Poland).

[2] Although the particular application is classified, the information about the network itself can be found at `http://www.devlab.nl/myrianed/`.

many subtle corner cases [18]. For these reasons, Leong et al. [20] propose an alternative approach: building a hull tree. However, with this approach, we have to face the same problems as with GEM (see below). In either case, providing nodes with their geographic coordinates requires special hardware and/or additional localization algorithms. Finally, there is no clear and practical way of porting geographic routing to three dimensions, and thus, deploying a large volumetric indoor network requires different principles for organizing the nodes.

In the second technique, graph embedding, instead of geographic coordinates, a node and an area are identified by virtual coordinates synthesized by the maintenance algorithm. The algorithm by Rao et al. [21] synthesizes coordinates through an iterative relaxation that embeds nodes in a Cartesian space. The setup phase requires nodes in the perimeter (roughly $O(\sqrt{N})$ nodes) to flood the entire network, and moreover, each such a node must store a matrix of distances between every pair of perimeter nodes (roughly $O(N)$ entries). However, maintaining such a state by a memory-constrained sensor node is practically impossible in large networks. GEM [22] embeds nodes in a polar coordinate space based on a spanning tree rooted at a base station. Therefore, each node maintains only a small constant state. However, as its authors admit, GEM has an intricate failure recovery algorithm, in which after a single node/link failure, a potentially large number of nodes must reassign their virtual coordinates using mechanisms such as two-phase locking. Due to intricate failure recovery and other drawbacks, existing graph embedding techniques are of a limited use in real-world sensornet deployments.

The last approach involves self-organizing nodes into a multi-level hierarchy of network areas by logically grouping connected nodes into areas, grouping areas into superareas, and so on [13, 23]. A node or an area is identified based on its membership in the hierarchy. Such naming enables efficient routing with only $O(logN)$ state per node [23]. Moreover, this approach does not require any special hardware and works equally well in both planar and volumetric deployments. However, maintenance algorithms proposed so far [13, 24] do not sufficiently specify mechanisms to construct a multi-level hierarchy that adapts to network dynamics. In addition, they require multi-phase multi-hop internode coordination (which is difficult to handle reliably in sensornets) or do not guarantee that all nodes in the network will be associated with some area [8]. To the best of our knowledge, PL-Gossip is the first complete solution for maintaining the area hierarchy in the presence of network dynamics and significant resource constraints of the nodes.

It is arguable that a related landmark hierarchy [23], in which some nodes are appointed as various-level landmarks and other nodes bind themselves to the closest landmark at each level, provides similar properties. However, the landmark hierarchy is *not recursive* as two nodes bound to the same level i landmark, can be bound to different level $i+1$ landmarks. As such, it disqualifies for many modern sensornet applications. Recursiveness is important not only for routing (which, arguably, the landmark organization handles well), but because it allows for many other aspects, notably *data aggregation* and *fusion* [11, 12]. These aspects are crucial when scalability is at stake, as in our case. However, efficiently providing recursiveness requires special mechanisms. Therefore, even though there are solutions for dynamically maintaining the landmark hierarchy in sensornets [8, 25], they cannot be simply modified to support the area hierarchy.

Finally, most of the existing maintenance protocols are *reactive* which complicates power provisioning. Since a sensornet node is battery powered and an active radio is the paramount power consumer, it is crucial to keep the radio off for most of a node's duty cycle in order to ensure a long network lifetime. This, in particular, requires the ability to predict the inbound and outbound maintenance traffic of each node. However, such predictable behavior is rarely feasible for reactive maintenance protocols, in which a node sends messages immediately in reaction to a change in the system or due to a message received from another node. Therefore, it is difficult to predict when to turn a node's radio off. In contrast, PL-Gossip, being a *periodic* protocol, generates constant traffic of one message per node in every time period. This allows for synchronizing duty cycles and turning off radios accordingly. However, because in a sleep period potentially many changes in the system may occur, special measures must be taken to ensure that every node will finally have a correct local state.

3 System Model and Definitions

In our model, the network consists of nodes with unique identifiers. Nodes communicate using wireless medium: each message is broadcast and the nodes able to hear the broadcast receive the message. There exists a link between two nodes if they are able to receive each other's messages.

Messages may be lost. The *bidirectional link quality*, defined as the percentage of messages delivered through a link in either direction in a time period, reflects the rate of losses. PL-Gossip measures this value in a standard way [26]. In short, making use of PL-Gossip's traffic predictability, a node computes reverse link qualities for all its peers and piggybacks these values in its messages. Based on this information, the peers compute their forward link quality to this node. The bidirectional link quality is the minimum of these two values.

Two nodes are *neighbors* if and only if (abbr. *iff*) the bidirectional quality of their link is above a certain threshold, θ (e.g., $\theta = 90\%$). The link quality estimation ensures that the neighbor relation is symmetric. Moreover, we assume that the graph represented by this relation is connected.

We group nodes into sets based on their connectivity. The groups correspond to network areas and form a multi-level hierarchy that provides an addressing scheme and enables

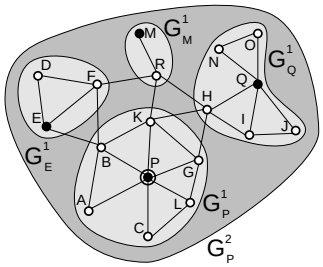


Fig. 1. An example of a group hierarchy. *For the purpose of presentation, the neighborhood graph is planar.*

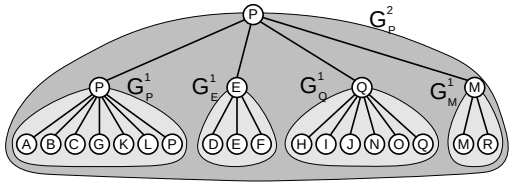


Fig. 2. The membership tree for the network from Fig. 1. *The circles denote the head nodes of the groups on consecutive levels. The label of a node can be obtained by concatenating node identifiers on the path from the leaf representing the node to the root of the tree.*

efficient routing and multicasting. We say that two groups are *adjacent* iff they contain two nodes (one in each group) that are neighbors. The hierarchy satisfies the following properties.

Property 1 *Level* 0 *groups correspond to individual nodes.*

Property 2 *There exists a single, level $\mathcal{H}$ group that contains all nodes. We call this group the* top-level *group.*

Property 3 *Level $i+1$ groups (where $0 \leq i < \mathcal{H}$) are composed out of level i groups, such that each level i group is in exactly one level $i+1$ group. A level $i+1$ group is the* supergroup *for its level i groups, and likewise, these level i groups are the* subgroups *of their level $i+1$ group.*

Property 4 *Each level $i+1$ group (where $0 \leq i < \mathcal{H}$) contains a subgroup which is adjacent to all other subgroups of this group. We call one (if there is more) such a subgroup the* central *subgroup of the group.*

The *head* node of a level i group is defined recursively: (1) if $i = 0$, then the head of the group is the sole node constituting the group; (2) if $i > 0$, then the head of the group is equal to the head of the central subgroup. We also say that a node is a *level i head* iff it is the head of a level i group, but not the head of a level $i+1$ group. Intuitively, the head of a group allows for identifying the group and is responsible for maintaining consistent group hierarchy membership among the nodes in the group.

Let G_X^L denote a level L group with head node X. A sample group hierarchy is depicted in Fig. 1. The circles represent nodes and corresponding level 0 groups. The straight lines between nodes represent the neighbor relation. The light-gray areas constitute level 1 groups, whereas the dark-gray area corresponds to the level 2 group, which in this case is also the top-level group.

Group G_F^0 is adjacent to groups G_D^0, G_E^0, G_B^0, and G_R^0. Similarly, group G_E^1 is adjacent to groups G_P^1 and G_M^1, but not to group G_Q^1. Group G_P^1 is the central subgroup of group G_P^2.

Node P (marked with a double circle) is a level 2 head, as it is the head of groups G_P^0, G_P^1, and G_P^2. Node E (black circle) is a level 1 head as it is the head of groups G_E^0 and G_E^1. Finally, since node D (empty circle) is only the head of group G_D^0, it is a level 0 head.

Note that the properties of our hierarchy differ slightly from the original definition [23]. These modifications not only enable efficient hierarchy maintenace, but also allow for proving certain features of the hierarchy. In particular, we can derive tight bounds on the distances between nodes belonging to groups at various levels, as formalized by the lemmas below. The proofs of these lemmas (by induction) can be found in Appendix A.

Lemma 1 *A node from a level i group can reach a node in any adjacent level i group in at most 3^i hops.*

Lemma 2 *The distance between the head nodes of two adjacent level i groups is at most 3^i hops.*

Lemma 3 *The distance between any two members of a level i group is at most $3^i - 1$ hops.*

4 Naming and Routing

Nodes and groups are named using *labels* that facilitate routing and multicasting. The label of a node reflects the membership of this node in the group hierarchy. It is a vector of $\mathcal{H}+1$ node identifiers. The k-th element in this vector is the identifier of the head of the level k group the node is member of. In the example from Fig. 1, as node D belongs to groups G_D^0, G_E^1, and G_P^2, its label $L(D)$ is equal to $D.E.P$, while the label of node E is $L(E) = E.E.P$. Likewise, the label of a level i group is an $(\mathcal{H}+1)$-element vector, in which the elements at positions $0 \ldots i-1$ are undefined and the elements at positions $i \ldots \mathcal{H}$ are the suffix of the label of any member of the group. For instance, the labels of groups G_D^0, G_E^1, and G_P^2 from Fig. 1 are $D.E.P$, $?.E.P$, and $?.?.P$ respectively.

The labels form the *membership tree* mirroring the group hierarchy. Fig. 2 presents the tree for the network from Fig. 1.

Based on its label, each node maintains a routing table, which is used by PL-Gossip for maintaining the hierarchy

ROW/ LEVEL	ENTRY Group	ENTRY Fields		ENTRY Group	ENTRY Fields		ENTRY Group	ENTRY Fields		ENTRY Group	ENTRY Fields	
2	$P\ (G_P^2)$	*nextHop*	F	null			null			null		
		hops	3									
		isAdj.	YES									
1	$P\ (G_P^1)$	*nextHop*	E	$E\ (G_E^1)$	*nextHop*	E	$Q\ (G_Q^1)$	*nextHop*	F	$M\ (G_M^1)$	*nextHop*	F
		hops	3		*hops*	1		*hops*	4		*hops*	3
		isAdj.	YES		*isAdj.*	YES		*isAdj.*	NO		*isAdj.*	YES
0	$D\ (G_D^0)$	*nextHop*	D	$E\ (G_E^0)$	*nextHop*	E	$F\ (G_F^0)$	*nextHop*	F	null		
		hops	0		*hops*	1		*hops*	1			
		isAdj.	YES		*isAdj.*	YES		*isAdj.*	YES			

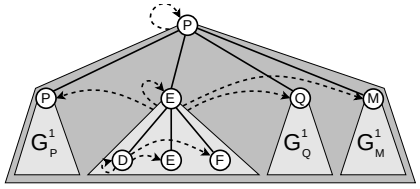


Fig. 3. The routing table of node D (label $D.E.P$) from Fig. 1 and its visualization in the membership tree from Fig. 2.

and implementing routing for the applications. The routing table has $\mathcal{H}+1$ rows corresponding to the levels in the group hierarchy. The entries in the i-th row ($0 \leq i \leq \mathcal{H}$) of a node's routing table each refer to a level i group whose label shares the node's label starting from position $i+1$ (see Fig. 3). An entry for a level i group contains the identifier of the next hop neighbor on the shortest path to the head of this group, the number of hops to reach the head, and a flag indicating whether the group is adjacent to the present node's level i group. To tolerate message loss, each entry also contains an age denoting the number of rounds that passed since the entry was refreshed for the last time.

The routing (see Appendix B) is performed hierarchically, by resolving consecutive elements of the destination's label starting from the maximal level differing at the sender's label and down to zero [23]. In order to resolve a single element, multiple hops might be necessary, depending on the level of this element. For instance, in order to route a message sent by node D ($L(D) = D.E.P$) from Fig. 1 to node N ($L(N) = N.Q.P$), we need to first resolve the level 1 group from G_E^1 to G_Q^1. This can be achieved in 3 hops by routing through F and R to H. Then we resolve the level 0 group from G_H^0 to G_N^0 in 2 hops by routing through Q. Multicasting is similar: we need to reach a member of the destination group and then broadcast the message across the other members, for instance, using a gossiping protocol like Trickle [27].

The location problem, that is, obtaining the label of a destination node, is generic to all routing algorithms, and thus, we do not address it here. For instance, it can be addressed using the hierarchy and the routing algorithm itself [8, 25].

5 Hierarchy Maintenance

Because the system is dynamic (nodes and links can fail, the network may partition, new nodes may be added, etc.), the group hierarchy and routes evolve, and thus, require constant maintenance. In order to ensure predictable maintenance traffic and implicit means of coordination, each node divides the time into *rounds*, each lasting ΔT time units.[3] In every round, each node broadcasts a single *beacon message* to its neighbors and receives messages from these neighbors. A beacon message contains the label of the sender, related consistency information (as explained below), and (fragments of) the sender's routing table. In other words, in every round each node gossips hierarchy information and routes with its neighbors.

Gossiping guarantees that the changes in the system will eventually be propagated among affected nodes. However, to accomplish this, it requires the means for detecting violations of the hierarchy properties, ensuring consistent decisions regarding hierarchy changes that eliminate such violations, and maintaining route information. In the following sections, we discuss these issues (in a slightly different order).

5.1 Ensuring Hierarchy Consistency

The group hierarchy is represented by the membership tree, which is a distributed data structure consisting of the labels of all nodes (see Fig. 2). The changes in the hierarchy, explained in detail in Section 5.3, correspond to moving, creating, and removing subtrees representing different-level groups. Such operations require updating node labels on different positions. However, the updates are not independent, can occur concurrently and accumulate while a node sleeps, and may be noticed in a different order by different nodes. Therefore, we must ensure that all nodes adopt the updates consistently. More formally, we require that in the absence of changes in the system, for any group G_X^i and any node A, if A is eventually a member of G_X^i ($L(A)[i] = X$), then A and X must eventually have equal labels starting from position i ($L(A)[i+] = L(X)[i+]$[4]). In other words, we require that for each group, any two members of this group eventually have the same information on the membership of the group in the hierarchy.

To this end, we adopted the single-master update model on a per-group basis. In essence, each group designates a single node making all the label updates regarding the

[3] From the perspective of PL-Gossip, we do not require clocks to be synchronized, only that they run with similar frequencies. However, power management may require lose synchronization.

[4] Where $L(A)[i+]$ is an abbreviated notation for $L(A)[k], k \geq i$.

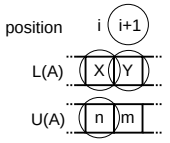


Fig. 4. A fragment of the label and the update vector of a sample node. *Node A knows that the last update performed by X at level i+1 has number n and corresponds to writing Y.*

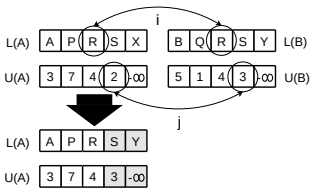


Fig. 5. An example of update propagation. *Node A determined that B has a fresher update performed by S at level j and thus, A adopts B's updates.*

membership of this group in the hierarchy, as formalized by the rule below. Other group members adopt the updates by such a node.

Responsibility Rule: *For every node, the decision regarding membership of that node in a level $i+1$ group is made by the head of the level i group the node is member of.*

Intuitively, the rule states that the head of a group is responsible for moving the subtree of that group between trees corresponding to different supergroups in the membership tree.

Moreover, to facilitate the adoption of the label updates by all group members, we introduced a local ordering of the updates. More specifically, each node maintains an *update vector* corresponding to the node's label. The i-th element of this vector denotes the number of the last known label update at position $i+1$ made by the head node represented by the i-th element in the node's label, as dictated by the responsibility rule (see also Fig. 4). For instance, assume that in some round D's label is $L(D) = D.E.P$ and its update vector is $U(D) = \langle 7.5.-\infty \rangle$. Consequently, D knows that (1) the last update performed by D at level 1 has number 7 and corresponds to writing E at position 1 in the label; (2) the last update performed by E at level 2 has number 5 and corresponds to writing P at position 2 in the label; (3) P acting as the head of the top-level group has not yet made any updates at level 3 ($U(D)[2] = -\infty$, $L(D)[3] = null$).

Whenever a node acting as a level i head makes a membership decision at level $i+1$ (as dictated by the responsibility rule), it updates the $(i+1)$-st element of its label to reflect the decision, increments its *update counter*, and stores the value of this counter at the i-th position of the update vector. The update vector, together with the node's label, is broadcast in beacon messages allowing other members of the node's level i group to learn about the new update, as follows.

Upon reception of a beacon from node B, node A checks if it shares a group with B. More specifically, A looks for the minimal i such that $L(A)[i] = L(B)[i]$ (see Fig. 5). If such i does not exists, then A has just discovered a hierarchy violation (see Section 5.3.1). Otherwise, A determines which of the two labels is fresher by comparing its update vector $U(A)$ with B's update vector $U(B)$ starting from position i. If for some $j \geq i$, $U(A)[j] \neq U(B)[j]$ (see Fig. 5) then one of the labels is fresher than the other [they can differ starting from the $(j+1)$-st element]. If B's label is fresher ($U(A)[j] < U(B)[j]$), then A copies B's label and update vector starting from position j: $L(A)[j+] \leftarrow L(B)[j+]$ and $U(A)[j+] \leftarrow U(B)[j+]$. This way, A's information on the hierarchy membership becomes consistent with the fresher information from B, and moreover, when A broadcasts the next beacon, its neighbors can also adopt the fresh information.

Theorem 1 *The algorithm guarantees eventual consistency.*

The proof can be found in Appendix C. Essentially, at every moment in time, each element in the node's label has a well defined head node that can update this element. Since a head orders its updates and we prioritize updates made by lower-level heads, for any two labels we can unambiguously choose the one that has fresher updates. By gossiping constantly, nodes gradually adopt the freshest updates. It is possible to optimize the algorithm a bit, but we do not present the optimization here due to lack of space.

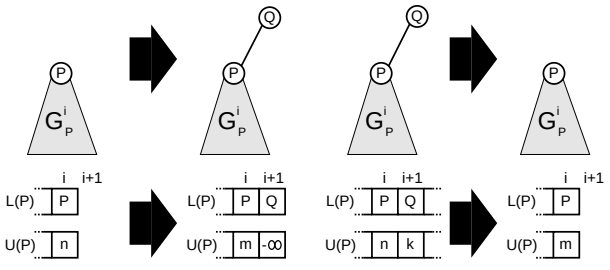
5.2 Maintaining Route Information

Apart from hierarchy maintenance, beacon messages are used for maintaining routes between nodes. Following the definition of the routing table (see Section 4), route maintenance is straightforward. After a possible update of its label, as discussed in the previous section, node A uses parts of B's routing table contained in the beacon for updating its own routing table. More specifically, if G_X^i is the minimal-level common group of A and B ($i > 0$), A can update its routing table with those entries from B's routing table that are in rows no lower than $i-1$. Such an approach guarantees locality of information and consequently, scalability: routes internal to a group are not propagated outside this group (see Fig. 3).

The entries are updated to minimize the number of hops (i.e., ensure the shortest paths) and maximize freshness. Additionally, an entry with the adjacency flag is always preferred over a corresponding entry without this flag. If there are no failures, the algorithm ensures the lack of routing cycles. In the presence of failures, unrefreshed entries time out and are removed. Possible cycles that can occur in this situation are also gradually broken, as from Lemma 3 we can effectively bound the maximal path length for any entry. Consequently, all invalid entries are *always* eventually evicted.

5.3 Detecting and Reacting to Hierarchy Violations

Building and maintaining the group hierarchy requires no dedicated coordination. Instead, by gossiping continuously,



(a) label extension (b) label cut

Fig. 6. Basic operations on a label. *The m value is the next value of P's update counter. Note that both these operations abide by the responsibility rule. Label cut may be viewed as writing null at the $(i+1)$-st position of P's label, assuming that the minimal position with null indicates the end of a label. Label extension, in turn, corresponds to writing the identifier of a node in the place of the null value.*

nodes learn about violations of the hierarchy properties and autonomously react to such violations, abiding by the responsibility rule. The only two operations that can be performed by a node in reaction to a hierarchy violation are *label extension* and *label cut* (see Fig. 6). The only means of coordination between nodes when performing such autonomous and local operations are the periodicity of gossiping and the bounds defined by Lemma 1-3. In the remainder of this section we discuss how the hierarchy violations are detected and how label extension and label cut are used to react to such violations.

5.3.1 Detecting Hierarchy Violations

There are two types of group hierarchy violations: "hierarchy incomplete" violation and "hierarchy fault" violation. The former type corresponds to violating Property 2, whereas the latter to violating Property 4. Property 1 always holds, and Property 3 is guaranteed by the responsibility rule. The violations can occur simultaneously at various levels.

A *head* node learns about a hierarchy violation from its routing table. For instance, if the routing table of the head of a group suddenly lacks an entry for the central group of the supergroup, a "hierarchy fault" violation occurs. In turn, the routing table of the head of a "top-level" group containing an entry for another same- or higher-level group indicates a "hierarchy incomplete" violation.

"Hierarchy fault" violations occur due to node and link failures and thus, their detection is guaranteed by the mechanism for evicting unrefreshed entries from the routing table (see Section 5.2). In contrast, "hierarchy incomplete" violations, occurring during hierarchy construction and failure recovery, require relaxing some properties of the routing table. More specifically, suppose that upon reception of a beacon from node B, as described in Section 5.1, node A discovers that it does not share any group with B (no common elements in their labels, Property 2 is violated). If the length of B's label (l_B) is greater or equal to the length of A's label (l_A), then, in rows $l_A-1,\ldots,l_B-1$, A adds routing table entries that correspond to the groups B is member of. For instance, if A's label is $L(A)=A.L.T$ and B's label is $L(B)=B.R.I.P$, node A adds routing entries for groups G^2_I and G^3_P with B as the next-hop neighbor and the adjacency flag set to true. In other words, we allow the routing table of a node to also contain entries for adjacent groups that do not share the label with the groups this node is member of (cf. Section 4). The route maintenance mechanism (see Section 5.2) guarantees that all members of A's top-level group, in particular the head, learn about B's same- and higher-level groups just discovered by A. Consequently, the head, based on its routing table, detects a "hierarchy incomplete" violation and reacts to it. Note that if $l_A=l_B$, then both A and B add entries to their routing tables, allowing the heads of both groups to learn about each other.

Since a group is uniquely identified by its level and the node identifier of its head, relaxing the routing table definition by allowing additional entries does not disrupt routing. Moreover, such additional entries will eventually time out after a reaction of the group head to the "hierarchy incomplete" violation.

To sum up, by examining its routing table, each node acting as a level i head, is able to detect a violation of the hierarchy properties at level i and react to this violation, as described below.

5.3.2 Hierarchy Construction

Initially, each node is a top-level head (the head of its level 0 group), that is, its label length is equal to 1. Hierarchy construction is performed by top-level head nodes detecting the "hierarchy incomplete" violations and reacting to such violations using the label extension operation, which corresponds to merging groups into higher-level groups (see Fig. 6a). Such an approach of group merging inherently ensures recovery after network partitions and other massive faults. However, it requires the head nodes to use the label extension operation in a way that ensures the convergence of the labels into the membership tree.

The head, P, of a "top-level" group, G^i_P, discovers a "hierarchy incomplete" violation iff its routing table contains entries for an adjacent G^k_Q, where $k \geq i$. There are two possible scenarios: (1) if $k=i+1$, P can try to make G^i_P a subgroup of G^{i+1}_Q; (2) P can spawn a new supergroup, G^{i+1}_P, hoping that other adjacent level i groups will join this group or that it will be possible to make G^{i+1}_P a subgroup of some level $i+2$ group. Making G^i_P a subgroup of G^{i+1}_Q corresponds to P extending its label with Q at level $i+1$ and modifying its update vector at level i (see Fig. 6a). Other members of G^i_P will gradually learn about the membership

update and extend their labels, as described in Section 5.1. Note that our consistency enforcement algorithm guarantees that if G_Q^{i+1} is itself a member of some G_R^{i+2}, all members of G_Q^{i+1} (in particular, the members of G_P^i) will also gradually extend their labels at level $i+2$ with R, and so forth. Likewise, spawning a new supergroup, G_P^{i+1}, corresponds to P extending its label with P.

Making G_P^i a subgroup of some existing G_Q^{i+1} is always preferred, as it decreases the number of groups at level $i+1$ compared to level i. However, due to Property 4, it is only possible if G_P^i is adjacent to the central subgroup of G_Q^{i+1}, that is, G_Q^i. In other words, P can extend its label at level $i+1$ with Q iff its routing table contains entries for G_Q^i and G_Q^{i+1} with the adjacency flag set.

Otherwise, P cannot *immediately* make G_P^i a subgroup of any level $i+1$ group, and thus, it must potentially spawn a new level $i+1$ group, G_P^{i+1}. To ensure convergence, we must prevent all groups from spawning supergroups in the same round. In particular, in the beginning, each node forms a single level 0 group, so allowing all nodes to create singleton level 1 groups would not guarantee convergence. To this end, P probabilistically defers spawning a supergroup for a number of rounds. More specifically, we cluster rounds into S virtual slots, each lasting R rounds. Upon discovering that it must potentially spawn a supergroup, P randomly selects a slot, $s \in \{0 \ldots S-1\}$. It then defers spawning a supergroup for $R \cdot s + 1$ rounds, hoping that in that time, some adjacent group spawns a supergroup, so that it will be possible to make G_P^i a subgroup of this supergroup.

Selecting $S = 2$ already ensures that the number of groups on consecutive levels drops fast, provided that the slot size, R, is long enough. Such a decrement is a direct consequence of the following lemma, with a simple proof in Appendix D.

Lemma 4 *Assume that the slot size is longer than the number of rounds it takes to propagate information between the heads, P and Q, of two adjacent groups G_P^i and G_Q^i. In this case, with probability $\geq \frac{1}{4}$, G_P^i will be able to join G_Q^{i+1} or vice versa.*

Oversimplifying things, assuming $S = 2$ and R meeting the above assumption, we could expect that half of the groups (the ones that chose slot 1) will be able to join the supergroups formed by the other half (the ones that chose slot 0), that is, the number of groups decreases exponentially fast with the level, resulting in a logarithmic height of the membership tree.

We can choose the slot size, R, guaranteeing the above requirements based on the entries in the routing table. More formally, assuming no message loss, a level i head deferring supergroup creation chooses R equal to the number of hops to the furthest adjacent level i head. Note that although this value is bounded by Lemma 2, it is smaller on average. Below, we describe how to deal with message losses.

5.3.3 Handling Failures

Failures can be divided into two classes: benign failures and disruptive failures. A benign failure causes no violation of the group hierarchy properties, but may only change the routing paths. Therefore, such a failure is automatically repaired by the route maintenance algorithm (see Section 5.2) and may even pass undetected by a node. In contrast, a disruptive failure, like a group head crash, causes violation of the properties of the group hierarchy. Consequently, disruptive failures are handled by head nodes detecting the "hierarchy fault" violations and reacting to such violations using the label cut operation, which corresponds to removing a subgroup from a group (see Fig. 6b). Later if necessary, the hierarchy construction algorithm, described above, will join such a removed subgroup to a different group, restoring the hierarchy properties.

The head, P, of a group, G_P^i, which is a subgroup of G_Q^{i+1} discovers a "hierarchy fault" violation iff its routing table does not contain an entry for the central subgroup, G_Q^i, of group G_Q^{i+1} or such an entry exists but its adjacency flag is not set. This implies that G_P^i should no longer be a subgroup of G_Q^{i+1}. To this end, P cuts its label down to position i and modifies its update vector at position i (see Fig. 6b), which corresponds to removing G_P^i from G_Q^{i+1}. Such an operation may generate a "hierarchy incomplete" violation that will be subsequently handled by the hierarchy construction algorithm. Our consistency enforcement mechanism guarantees that all members of G_P^i will adopt the decision of P to leave G_Q^{i+1} and later, possibly to join some other level $i+1$ group.

As an optimization a head may also cut its label to decrease the height of the hierarchy. However, special mechanisms are necessary to prevent thrashing between extending and cutting labels. We do not present them here due to lack of space.

Message losses may also be viewed as failures. We try to tolerate a certain percentage of message loss in three ways. First, by measuring the bidirectional link quality and using only high quality links to define neighbors (see Section 3), we ensure certain value $(1-\theta)$ of the expected message loss rate. Second, by introducing the local age field of each entry in a node's routing table, we allow several consecutive beacon messages that refresh this entry to be lost. For instance, assuming the bidirectional link quality threshold $\theta = 80\%$ and the maximal age equal to 4 (4 rounds are necessary to remove an unrefreshed entry), the expected probability of removing an entry corresponding to a live node/link is at most $(\frac{100-80}{100})^4 = 0.0016$. Finally, while constructing the hierarchy, we normalize the slot size, R with θ: $R^* = \lceil R \cdot (1 + 2 \cdot (1-\theta)) \rceil$, which essentially forces a head to defer spawning a supergroup a bit longer to compensate for the expected message loss that otherwise might prevent timely delivery of information from another head. Correlated message losses above the expected value are simply treated as transient link failures, handled in a standard way by our algorithm.

6 Evaluation

We evaluated PL-Gossip with simulations performed using our own event-driven high-level simulator. The simulator abstracts many peculiarities of the wireless medium, allowing us to simulate very large networks and to repeat experiments multiple times. It makes several simplifying assumptions, standard for high-level sensornet simulations. First, it models nodes as having a fixed circular radio range: a node has links to all and only those nodes that fall within its range. Second, it ignores the capacity of, and congestion in, the network. Finally, the message loss is fixed to $1 - \theta$ for all links (i.e., it matches the bidirectional link quality threshold). We believe that these assumptions do not severely impair real-world operation of PL-Gossip because (1) PL-Gossip creates the logical network structure based solely on *physical links* and the *measured* value of the link quality, and thus, it makes no implicit assumptions regarding connectivity or message loss; (2) the state exchanged between nodes is *small* (see Section 6.3) whereas the round length is *large*, and therefore, the MAC layer can efficiently schedule packet transmissions without exceeding the network capacity or causing congestion above the expected message loss.

We simulated PL-Gossip with various network sizes, densities, and topologies. Because the results were consistent in all cases, due to lack of space and for the sake of brevity, here we present only a subset of the experiments. More specifically, in these experiments, we arranged nodes into a square grid with unit spacing between nodes. The radio range of a node was 2 units, giving a node at least 5 (corner nodes) and at most 12 (most of the nodes) neighbors.

We first demonstrate the scalability of the algorithm and the quality of routing. Then, we show how the algorithm behaves in the presence of message loss and network dynamics. Finally, we give estimates on the state maintained and the bandwidth used by each node.

6.1 Scalability and Routing Quality

Since we wanted to get insight into general properties of the algorithm, for the experiments presented in this section we assumed no failures or message loss. All nodes were booted simultaneously in round 0 and the experiment was stopped when the membership tree had converged, that is, all the nodes had equal-length labels with the same last element. Simultaneous boot is a pessimistic scenario for PL-Gossip, as there are no higher-level groups formed, and consequently, all nodes must potentially spawn such groups. When deferring spawning a supergroup, the number of slots, S, used by a "top-level" head (see Section 5.3.2) varied based on the level: at level 0, $S = 10$; at higher levels, $S = 2$. The rationale behind such a choice is minimizing the hierarchy height for high-density networks. Oversimplifying things, by having 10 slots instead of 2 at level 0, we reduce the number of level 1 heads with respect to the number of level 0 heads (all nodes) roughly 10 times instead of 2 times. This generates shorter membership tree. Moreover, the convergence time does not grow drastically, as from Lemma 2, the slot length at level 0 is equal to only 1 round, that is, after at most 10 rounds each node is guaranteed to be a member of some level 1 group.

The scalability of PL-Gossip with respect to the network size is analyzed based on the height of the hierarchy, the average size of a node's routing table, and the number of rounds necessary to form the hierarchy. The routing quality is measured using a standard metric, the average *dilation/hop stretch*: the average ratio of the number of hops on the route between two nodes to the number of hops in the shortest path in the neighborhood graph.

We conducted experiments for exponentially growing network sizes, with 100 runs for each size. Figure 7 presents the results. We see that both the hierarchy height (see Fig. 7a) and the average size of the routing table (see Fig. 7b), grow logarithmically with the network size. In particular, for a 1024-node network, in 95% of the cases, these values are below 11 and 33 respectively. This is a direct consequence of our hierarchy properties and the construction algorithm.

The convergence time depends on the diameter of the network, and thus, it grows exponentially with the exponentially growing network size (see Fig. 7c). However, the absolute values indicate that the convergence is relatively fast. For instance, for a 1024-node network, the hierarchy is formed within 38.4 rounds on average and at most 70 rounds in 95% of the cases. Assuming the gossiping period $\Delta T = 5$ *min.*, we need 3.2 hours on average and at most 5.8 hours in 95% of the cases. We believe that this is insignificant compared to the expected network lifetime of weeks or even months, achievable with such a gossiping period [1].

Finally, the average hop stretch is relatively stable for increasing network sizes (see Fig. 7d). Although the results reported for various algorithms are not directly comparable, they indicate that the routing overhead of PL-Gossip is small. For instance, both GEM [22] and geographic routing with hull trees [20] report similar hop-stretch values.

The experiments performed with different node densities and network topologies produced similar results. Consequently, we do not present them here.

6.2 Message Loss and Network Dynamics

Message loss is an inherent feature of sensornets. In the case of PL-Gossip, message loss can have two effects: (1) it may prevent a head node deferring supergroup creation from learning in time about a newly created supergroup it could join, which can affect the hierarchy quality; (2) it may cause a node to falsely determine that a link is dead, which may possibly lead to unnecessary changes in the group hierarchy.

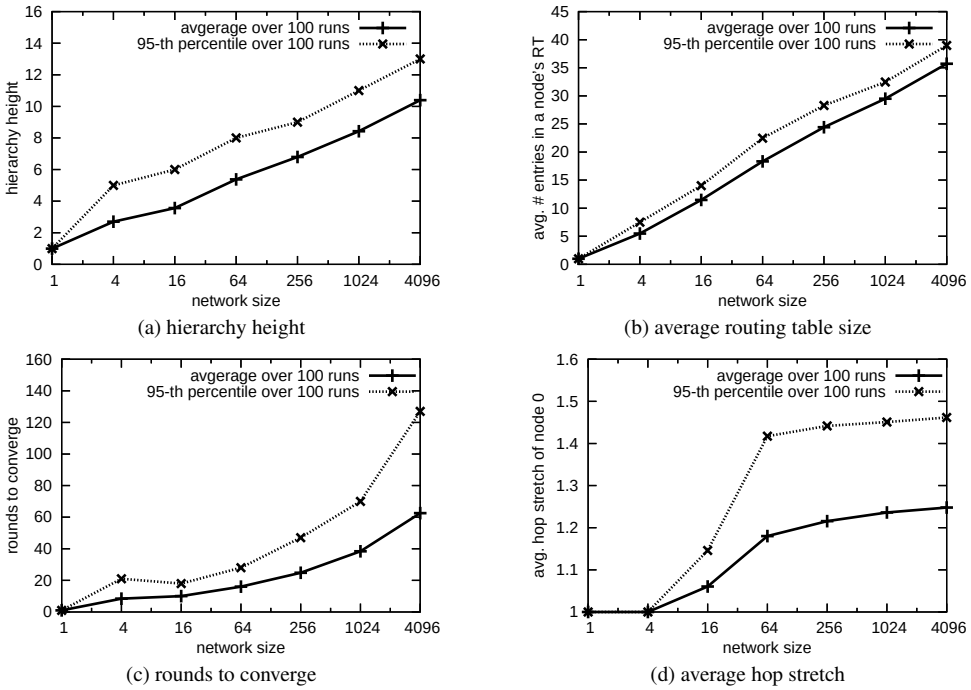


Fig. 7. Scalability of PL-Gossip with respect to the network size. *All values were obtained over* 100 *runs. Note the logarithmic scale of the x-axis.*

To analyze the first effect of message loss and countermeasures employed by PL-Gossip, we repeated the experiments from the previous section with message loss rates of 1%, 5%, 10%, and 20% ($\theta = 99\%$, 95%, 90%, and 80%, respectively). In these experiments, we isolated the first effect of message loss from the second one by blocking the eviction of unrefreshed entries from routing tables. However, to enable comparison of the final routing table sizes with the results from the previous section, at the end of each experiment, we removed the additional routing table entries created in the hierarchy construction process (see Section 5.3.1). Normally, such entries would have been evicted in due course.

The results (not plotted) demonstrate that the hierarchy height, the average routing table size, and the average hop stretch practically do not differ from the environment without message losses, presented in the previous section. This indicates that, in practice, the selection (see Section 5.3.2) and normalization (see Section 5.3.3) of the virtual slot size (R) by a head node deferring a supergroup creation ensures timely propagation of information on newly created groups. As a result, the quality of the hierarchy and routes does not deteriorate in the presence of message loss. Similarly, although normalizing R with the expected message loss derived from the bidirectional link quality, θ, effectively increases the duration of a slot, the final effect on the convergence time is marginal. In particular, the number of extra rounds for a 1024-node network to converge in the presence of 20% message loss is 3.34 on average and at most 7 in 95% of the cases (cf. Fig. 7c). In other words, the hierarchy and the routes can be efficiently constructed even in the presence of message loss.

Since the second effect of message loss is directly correlated with failure detection, due to space constraints here, we combine the experiments on this effect with the experiments on network dynamics. In both cases, a (seeming) failure of a node or a link may turn out disruptive and trigger changes in the hierarchy. Applying such changes can temporarily lengthen or brake routes between nodes and also enlarge routing tables or change the hierarchy height.

In each of the experiments, we simulated a 1024-node network for $21,000$ rounds. In any round, 128 nodes out of 1024 (12.5%) were dead (896 were alive). Moreover, 32 nodes, selected randomly at the beginning of each experiment, were always alive and were used as reference nodes for measuring the routing quality in each round. In the initial 1000 rounds there were no changes in the node population. During the next $10,000$ rounds, we generated node churn of a given rate. For instance, the churn rate of 2 denotes that in every round 1 random live node was killed and 1 random dead node was rebooted. Finally, during the last $10,000$ rounds there was again no churn.

We were interested in how the hierarchy and route properties deteriorate in the presence of both message loss and network dynamics. In each round, we measured the hierarchy height, the average routing table size, the average hop stretch, and the *reachability* (the existence of a route from a node to another node). While the first two metrics

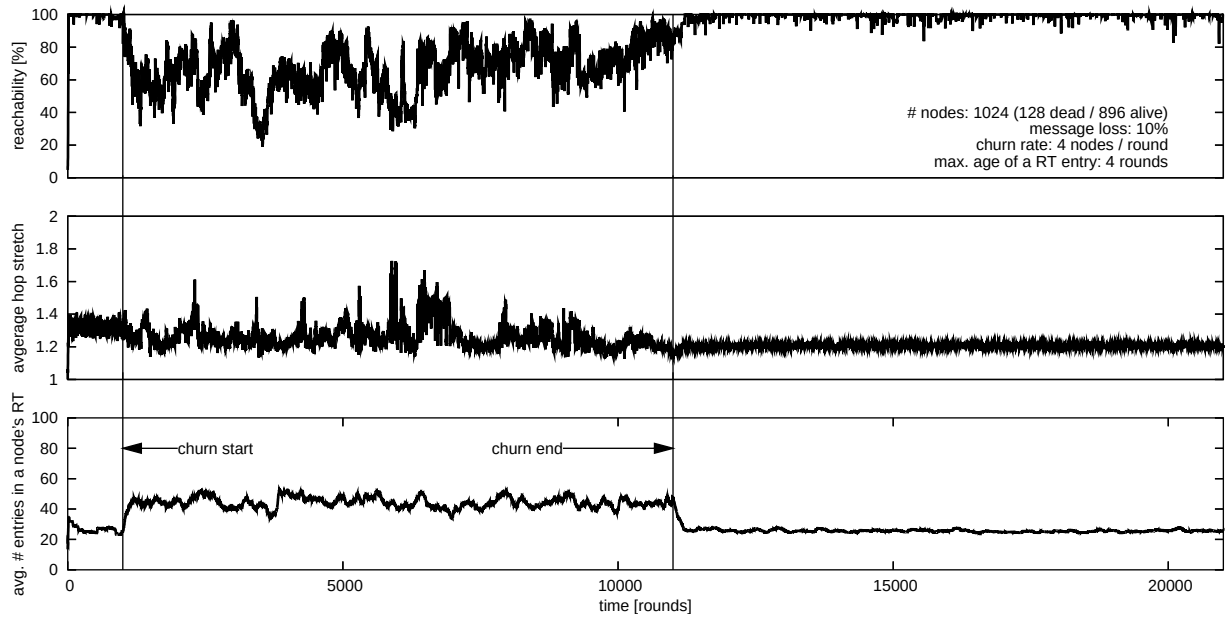


Fig. 8. An example of the system behavior with message loss and network dynamics. *The parameter values are visible in the top plot.*

were measured over the whole population of live nodes, the last two were measured only over the aforementioned 32 static nodes. We believe that these 32 nodes ($32 \cdot 31 = 992$ pairs) were enough to capture changes in the routes, and moreover, the static node population they constituted had an insignificant size of 3.125%. Additionally, measuring routing quality only between 32 nodes greatly reduced the excessive time of a single experiment (to approximately a day of computing).

We ran the experiments for different message loss rates (mentioned before) and exponentially growing churn rates. We also varied the maximal age of a routing table entry, which determines how fast an unrefreshed entry can be removed from a node's routing table. The huge number of possible configurations and the long duration of a single experiment prevented us from conducting more than one experiment per configuration.

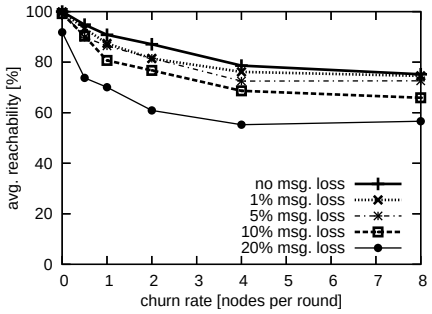
Figure 8 presents the results of a sample run. The reachability (top plot) grows fast (not really visible) as the hierarchy is being constructed. The occasional falls during the initial 1000 and the last $10,000$ rounds are caused by message losses triggering unnecessary hierarchy changes. With 10% message loss and the maximal age of a routing table entry equal to 4, the probability that some node falsely determines that a link is dead is high considering the total number links. If such an uncommon link "failure" causes a "hierarchy fault" violation in a group, the label cut operation executed by the head of this group may prevent communication to and from the group (the communication within the group is preserved). This reduces reachability of a number of nodes depending on the level of the group in the hierarchy. Node churn, which introduces real failures in the system, only amplifies this effect and causes greater oscillations in reachability.

Similarly, network dynamics generate peaks in the average hop stretch value (center plot). This is because it takes some time before a new short route via a just-booted node is propagated within the affected group.

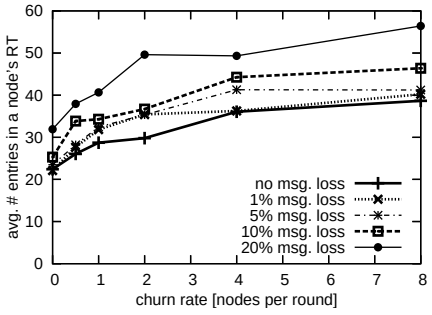
Node churn also leads to a higher average routing table size (bottom plot). It takes a few rounds, depending on the maximal age of a routing table entry, to determine that a node is dead or a group ceased to exist, while new nodes are constantly added to the system. Consequently, the routing tables are polluted with entries corresponding to no-longer existing groups and additional entries due to hierarchy recovery. Nevertheless, even under high churn, the average routing table size is relatively small and stable, and it decreases fast as soon as the churn stops.

Finally, message loss and network dynamics may result in the increments or decrements of the hierarchy height (not plotted). Such events, however, occur very rarely.

Figure 9 shows the reachability deterioration and the average routing table growth for different rates of churn and message loss. We argue that the reachability with the churn rates visible in the figure seems low, because these churn rates are excessively high. For example, with the churn rate of 8, the mean inter-failure interval (MIFI) of a node is $896 \cdot (\frac{2}{8}) = 224$ rounds. For the round length $\Delta T = 5$ *min.*, MIFI is equal to 18.67 hours. In practice, since a sensor node, unlike a PC in a peer-to-peer network, is *dedicated* to the system, once successfully deployed, it usually works for weeks or months. For instance, Szewczyk et al. [1] report the mean lifetimes of 104, 60, 50, and 29 *days*, depending on the network type and tasks. Consequently, even the churn



(a) reachability



(b) average routing table size

Fig. 9. The hierarchy and route behavior in the presence of churn and message loss. *The maximal age of the routing table entry was* 4. *Each value was obtained over* $10,000$ *rounds from one* $21,000$*-round run.*

rate of 1, resulting in a MIFI equal to 6 days 5 hours and 20 min., is still relatively high. We must emphasize that the reachability can be improved by certain optimizations. However, we did not do this, as we wanted to conform to the simple algorithm presented in the paper and to show trends rather than absolute values.

Unsurprisingly, the growth of routing tables also depends on the rate of churn. The entries to no-longer existing groups are evicted from a routing table after some time. Consequently, if the changes in the system are more frequent, routing tables contain more stale entries. In all our experiments, the average size of the routing table was relatively stable.

The average hop stretch and the hierarchy height (not plotted) essentially do not change with the increase in the churn rate.

Considering the above results, a valuable feature of PL-Gossip is its adaptability with respect to message loss and network dynamics, expressed through two parameters: the maximal age of a routing table entry and the bidirectional link quality threshold. Increasing the former parameter increases the time it takes to remove an unrefreshed entry from the routing table, and consequently, improves the tolerance of the algorithm to accumulated message loss (or to the variation in message loss). The price to pay, is a slower reaction to failures in the system (new nodes are always admitted fast) and bigger routing tables. Increasing the latter parameter, in turn, decreases the expected message loss. The trade-off, however, is a possibly higher diameter of the network, and thus, a higher hierarchy and bigger routing tables. Moreover, depending on the deployment environment, increasing this parameter is not always possible as it may result in a network partition.

We finish the discussion on fault tolerance by stressing two important properties of PL-Gossip, obtained through analysis and experimental results. First, a single node/link failure is rarely disruptive. Unless a failed node is a head of some group or the sole node connecting two groups, no changes in the hierarchy are necessary. Since the number of such nodes decreases exponentially with the level, the probability that an occurring failure is disruptive also decreases exponentially with the level. As a result, the work involved in repairing a single failure is small on average. Second, the reachability depends on node proximity. If a disruptive failure occurs within a group the reachability between the members of this group deteriorates. However, if a failure occurs outside the group (or in higher-level groups containing the group), all members of the group are able to reach each other anyway. Such behavior is crucial in many applications, like reactive tasking in emergency systems (e.g., temperature and smoke sensor readings activate nearby water sprinklers even though some far nodes might have been already damaged).

6.3 Bandwidth and Storage Costs

Consider a 1024-node network. A node can be uniquely identified with 10 bits. Using experimental data, we estimate the size of a node's label, update vector, and routing table, which are the only structures exchanged and stored by PL-Gossip. A routing table entry occupies 4 bytes (31 bits): 10 bits for the identifier of the group head, 10 bits for the identifier of the next hop neighbor, 10 bits for the path length, and 1 bit as the adjacency flag. Also assume that an element of a node's update vector has 20 bits, which for $\Delta T = 5$ *min.* would be sufficient for several years, even in the most pessimistic scenario. From Fig. 7a, the expected size of the label and of the update vector is 11 and 22 bytes, respectively. Let us take the highest message loss (20%) and churn rate (8) from Fig. 9b. They result in the biggest average routing table of 226 bytes (excluding the overhead for organization), so in total, the average size of a beacon message (and the state stored by a node, excluding some counters) is roughly $11 + 22 + 226 = 259$ bytes. Therefore, for such a big churn rate and message loss, with a gossip period $\Delta T = 5$ *min.*, the outgoing bandwidth consumed by a node is roughly 6.9 *bits* per second.

In other words, the bandwidth and memory requirements of PL-Gossip are so low, that the algorithm can run even on the early sensor nodes with $4kB$ RAM and $250kbps$ radios ($100kbps$ accounting for the MAC overhead). Moreover, the simplicity of the code facilitates the implementation on hardware-constrained devices (the listing of the core has

only 130 lines including comments, see Appendix E).

7 Summary and Future Work

There is a growing demand for a scalable, recursive, geometry-based network organization for sensornets. The area hierarchy is a practical instance of such an organization. However, it has been considered difficult to maintain in the presence of message loss and network dynamics, especially in the networks of resource-constrained sensor nodes. PL-Gossip approaches this problem in a novel way, by having nodes gossip messages periodically: one message broadcast per long time period. This generates a well specified, predictable traffic, which is crucial when it comes to the conservation of energy, one of the most important resources of sensor nodes. The traffic restrictions, however, lead to the problem of changes in the system accumulating while a node is inactive. PL-Gossip deals with such accumulated changes by defining invariants of the hierarchy, designating responsibility for maintaining these invariants, and ensuring consistent adoption of any updates. As confirmed by the experimental results, this approach is scalable, works well in the presence of message loss and constant changes in the node population, and has low requirements with respect to bandwidth and storage.

Despite promising results, we are aware that PL-Gossip requires much more evaluation. To this end, we are currently working on a TinyOS implementation. We hope to field-test it on a $10,000$-node sensornet we collaborate on. This would give us a lot of insight into the behavior of PL-Gossip in a real-world deployment. Finally, we believe that our algorithm will enable taming the challenges involved in applications such as data-centric storage and in the maintenance of large sensornets.

Acknowledgment

The authors would like to thank A. Bakker, M. Szymaniak, and G. Urdaneta for providing the necessary computing power. M. Szymaniak deserves additional gratitude for his comments regarding early drafts of an internal report that turned into this paper.

References

[1] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler, "An analysis of a large scale habitat monitoring application," in *Proceedings of the Second ACM Int. Conf. on Embedded Networked Sensor Systems (SenSys)*, Baltimore, MD, USA, November 2004, pp. 214–226.

[2] C. Sharp, S. Schaffert, A. Woo, N. Sastry, C. Karlof, S. Sastry, and D. Culler, "Design and implementation of a sensor network system for vehicle tracking and autonomous interception," in *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN)*, Istanbul, Turkey, January 2005, pp. 93–107.

[3] A. S. Tanenbaum, C. Gamage, and B. Crispo, "Taking sensor networks from the lab to the jungle," *IEEE Computer Magazine*, vol. 39, no. 8, pp. 98–100, August 2006.

[4] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed Diffusion: A scalable and robust communication paradigm for sensor networks," in *Proceedings of the Sixth ACM Annual International Conference on Mobile Computing and Networking (MobiCom)*, Boston, MA, USA, August 2000, pp. 56–67.

[5] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TAG: A Tiny AGgregation service for ad-hoc sensor networks," in *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, USA, December 2002, pp. 131–146.

[6] S. Shenker, S. Ratnasamy, B. Karp, R. Govindan, and D. Estrin, "Data-centric storage in sensornets," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 1, pp. 137–142, January 2003.

[7] I. F. Akyildiz and I. H. Kasimoglu, "Wireless sensor and actor networks: research challenges," *Ad Hoc Networks (Elsevier)*, vol. 2, no. 4, pp. 351–367, October 2004.

[8] S. Kumar, C. Alaettinoglu, and D. Estrin, "SCalable Object-tracking through Unattended Techniques (SCOUT)," in *Proceedings of the Eighth IEEE International Conference on Network Protocols (ICNP)*, Osaka, Japan, November 2000, pp. 253–262.

[9] K. Iwanicki and M. van Steen, "Sensor network bugs under the magnifying glass," Vrije Universiteit, Amsterdam, the Netherlands, Tech. Rep. IR-CS-033, December 2006, available at: http://www.few.vu.nl/~iwanicki/.

[10] X. Li, Y. J. Kim, R. Govindan, and W. Hong, "Multi-dimensional range queries in sensor networks," in *Proceedings of the First ACM Int. Conf. on Embedded Networked Sensor Systems (SenSys)*, Los Angeles, CA, USA, November 2003, pp. 63–75.

[11] B. Greenstein, S. Ratnasamy, S. Shenker, R. Govindan, and D. Estrin, "DIFS: A distributed index for features in sensor networks," *Ad Hoc Networks (Elsevier)*, vol. 1, no. 2-3, pp. 333–349, 2003.

[12] D. Ganesan, D. Estrin, and J. Heidemann, "Dimensions: Why do we need a new data handling architecture for sensor networks?" *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 1, pp. 143–148, January 2003.

[13] J. Hagouel, "Issues in routing for large and dynamic networks," Ph.D. dissertation, Columbia University, 1983.

[14] B. Karp and H. T. Kung, "GPSR: Greedy perimeter stateless routing for wireless networks," in *Proceedings of the Sixth ACM Annual International Conference on Mobile Computing and Networking (MobiCom)*, Boston, MA, USA, August 2000, pp. 243–254.

[15] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia, "Routing with guaranteed delivery in ad hoc wireless networks," *Wireless Networks*, vol. 7, no. 6, pp. 609–616, November 2001.

[16] F. Kuhn, R. Wattenhofer, Y. Zhang, and A. Zollinger, "Geometric ad-hoc routing: Of theory and practice," in *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, Boston, MA, USA, July 2003, pp. 63–72.

[17] B. Leong, S. Mitra, and B. Liskov, "Path vector face routing: Geographic routing with local face information," in *Proceedings of the Thirteenth IEEE International Conference on Network Protocols (ICNP)*, Boston, MA, USA, November 2005, pp. 147–158.

[18] Y.-J. Kim, R. Govindan, B. Karp, and S. Shenker, "On the pitfalls of geographic face routing," in *Proceedings of the 2005 Joint Workshop on Foundations of Mobile Computing (DIALM-POMC)*, Cologne, Germany, September 2005, pp. 34–43.

[19] Y.-J. Kim, R. Govindan, B. Karp, and S. Shenker, "Geographic routing made practical," in *Proceedings of the Second USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, USA, May 2005, pp. 217–230.

[20] B. Leong, B. Liskov, and R. Morris, "Geographic routing without planarization," in *Proceedings of the Third USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, USA, May 2006, pp. 339–352.

[21] A. Rao, S. Ratnasamy, C. Papadimitriou, S. Shenker, and I. Stoica, "Geographic routing without location information," in *Proceedings of the Ninth ACM Annual International Conference on Mobile*

Computing and Networking (MobiCom), San Diego, CA, USA, September 2003, pp. 96–108.

[22] J. Newsome and D. Song, "GEM: Graph EMbedding for routing and data-centric storage in sensor networks without geographic information," in *Proceedings of the First ACM Int. Conf. on Embedded Networked Sensor Systems (SenSys)*, Los Angeles, CA, USA, November 2003, pp. 76–88.

[23] P. F. Tsuchiya, "The landmark hierarchy: A new hierarchy for routing in very large networks," *ACM SIGCOMM Computer Communication Review*, vol. 18, no. 4, pp. 35–42, August 1988.

[24] N. Shacham and J. Westcott, "Future directions in packet radio architectures and protocols," *Proceedings of the IEEE*, vol. 75, no. 1, pp. 83–99, January 1987.

[25] B. Chen and R. Morris, "L^+: Scalable landmark routing and address lookup for multi-hop wireless networks," Massachusetts Institute of Technology, Cambridge, MA, USA, Tech. Rep. MIT-LCS-TR-837, March 2002.

[26] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, and I. Stoica, "A unifying link abstraction for wireless sensor networks," in *Proceedings of the Third ACM Int. Conf. on Embedded Networked Sensor Systems (SenSys)*, San Diego, CA, USA, November 2005, pp. 76–89.

[27] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks," in *Proceedings of the First USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, USA, March 2004, pp. 15–28.

A Proofs of Lemma 1-3

PROOF *of Lemma 1:* The proof is performed by induction.

Basis: $i = 0$. Let's take two arbitrary adjacent level 0 groups: G_A^0 and G_B^0. From Property 1, $G_A^0 = \{A\}$ and $G_B^0 = \{B\}$. G_A^0 and G_B^0 are adjacent, thus A and B are neighbors, that is, A can reach B in $1 = 3^0$ hop. Since we chose G_A^0 and G_B^0 arbitrarily, the lemma is true for $i = 0$.

Inductive step: $i = k+1$ (where $k \geq 0$). We assume that the lemma holds for all levels $\leq k$. Let's take two arbitrary adjacent level i groups G_A^i and G_B^i, and an arbitrary node $P \in G_A^i$. Let R denote a node in G_B^i that has a neighbor Q such that $Q \in G_A^i$ (existence of Q is guaranteed by the definition of adjacent groups). Consider level i subgroups, that is, G_C^{i-1}, G_D^{i-1}, and G_E^{i-1}, such that $P \in G_C^{i-1}$, $Q \in G_D^{i-1}$, and $R \in G_E^{i-1}$. We have the following three situations:

1. $C = D$ (group G_C^{i-1} is adjacent to group G_E^{i-1}). In this case, from the inductive assumption P can reach a node from G_E^{i-1} in at most $3^{i-1} < 3^i$ hops.

2. G_C^{i-1} is adjacent to G_D^{i-1}. In this case, from the inductive assumption P can get to a node in G_D^{i-1} in a most 3^{i-1} hops and any node from G_D^{i-1} can get to a node from G_E^{i-1} in at most 3^{i-1} hops. Consequently, P can get to a node from G_E^{i-1} in at most $2 \cdot 3^{i-1} < 3^i$ hops.

3. G_C^{i-1} is not adjacent to G_D^{i-1} (but from Property 4, G_C^{i-1} and G_D^{i-1} are both adjacent to G_A^{i-1}). In this case, from the inductive assumption P can get to a node in G_A^{i-1} in at most 3^{i-1} hops. Likewise, any node from G_A^{i-1} can get to a node from G_D^{i-1} in at most 3^{i-1} hops and any node from G_D^{i-1} can get to a node from G_E^{i-1} in at most 3^{i-1} hops. Therefore, P can get to a node from G_E^{i-1} in at most $3 \cdot 3^{i-1} = 3^i$ hops.

Consequently P can get to a node from G_B^i in at most 3^i hops. Since we chose P arbitrarily, any node from G_A^i can get to a node from G_B^i in at most 3^i. Because G_A^i and G_B^i were also chosen arbitrarily, the lemma is true for $i = k+1$.

By applying mathematical induction to the basis and the inductive step, we proved the lemma for all i. Moreover, 3^i is a *tight bound*, that is, it is reachable for some configurations. ■

PROOF *of Lemma 2:* The proof is performed by induction. Let $d(A,B)$ denote the distance in hops between nodes A and B.

Basis: $i = 0$. Let's take two arbitrary adjacent level 0 groups: G_A^0 and G_B^0. From Property 1, $G_A^0 = \{A\}$ and $G_B^0 = \{B\}$. G_A^0 and G_B^0 are adjacent, thus A and B are neighbors, that is, $d(A,B) = 1 = 3^0$. As we chose G_A^0 and G_B^0 arbitrarily, the lemma is true for $i = 0$.

Inductive step: $i = k+1$ (where $k \geq 0$). We assume that the lemma holds for all levels $\leq k$. Let's take two arbitrary adjacent level i groups G_A^i and G_B^i. We have three possible situations:

1. G_A^{i-1} is adjacent to G_B^{i-1}. In this case, from the inductive assumption, $d(A,B) \leq 3^{i-1} < 3^i$.

2. There exists G_C^{i-1} such that it belongs to G_A^i or G_B^i and it is adjacent to both G_A^{i-1} and G_B^{i-1}. In this case, $d(A,B) \leq d(A,C) + d(B,C)$. From the inductive assumption $d(A,C), d(B,C) \leq 3^{i-1}$, thus $d(A,B) \leq 2 \cdot 3^{i-1} < 3^i$.

3. There exist G_C^{i-1} and G_D^{i-1} such that G_C^{i-1} belongs to G_A^i and G_D^{i-1} belongs to G_B^i and G_C^{i-1} is adjacent to G_D^{i-1}. (From Def. 4, G_C^{i-1} is adjacent to G_A^{i-1}, and G_D^{i-1} is adjacent to G_B^{i-1}.) In this case, $d(A,B) \leq d(A,C) + d(C,D) + d(D,A)$. From the inductive assumption $d(A,C), d(C,D), d(D,A) \leq 3^{i-1}$, thus $d(A,B) \leq 3 \cdot 3^{i-1} = 3^i$.

Consequently, we have $d(A,B) \leq 3^i$. Because G_A^i and G_B^i were also chosen arbitrarily, the lemma is true for $i = k+1$.

By applying mathematical induction to the basis and the inductive step, we proved the lemma for all i. Moreover, 3^i is a *tight bound*, that is, it is reachable for some configurations. ■

PROOF *of Lemma 3:* We choose an arbitrary group G_A^i and two arbitrary nodes P and Q that are members of this group. We add another node R to the system such that R's only neighbor is Q.[5] R forms singleton groups $G_R^0 \ldots G_R^i$. From Lemma 1, P can reach R in at most 3^i hops. Since P can reach R only through Q, P can reach Q in at most $3^i - 1$ hops, that is the distance between P and Q is at most $3^i - 1$.

[5] Although in a practical setting this may be impossible, it is perfectly valid from the graph theory perspective, and consequently, does not invalidate the proof.

```
1  FUNCTION getNextHop(msg) {
2
3    // change TTL of the message
4    --msg.ttl;
5    if (msg.ttl < 0)
6      return null;
7
8    // determine if we share any group
9    int cpos = 0;
10   for (; cpos < min(this.lab.len, msg.dstLab.len); ++cpos)
11     if (this.lab[cpos] == msg.dstLab[cpos])
12       break;
13
14   if (cpos == 0} {
15     // we are the destination node
16     acceptMessage(msg);
17     return null;
18
19   } else if (this.neighbors.contains(msg.dstLab[0])) {
20     // one of our neighbors is the destination node
21     // (this is just an optimization)
22     return msg.dstLab[0];
23
24   } else if (cpos <= min(this.lab.len, msg.dstLab.len)) {
25     // resolve the next hop based on the routing table
26     Entry entry = this.rt[cpos - 1][msg.dstLab[cpos - 1]];
27     return entry != null ? entry.nextHop : null;
28
29   } else {
30     // we cannot forward the message
31     return null;
32   }
33 }
```

Listing 1. The main routing function.

Because P and Q were chosen arbitrarily, the lemma holds for any members of group G_A^i. Likewise, the arbitrary choice of G_A^i and i proves the lemma for all i. Again, $3^i - 1$ is a tight bound. ■

B Hierarchical Suffix-Based Routing

Routing is performed by resolving consecutive elements of the destination label starting from the maximal-position element differing at the sender. The main routing method, executed by a node on each hop, is presented in Listing 1.

Upon reception of an *application* message (which is different from a beacon message used by PL-Gossip to maintain the network structure), a node decrements the *time-to-live* (TTL) counter associated with the message and examines this counter to decide whether the message should be dropped (listing line 2-6). TTL is a mechanism for dropping messages that cannot be delivered to their receivers due to network dynamics (e.g., receiver failures). The TTL counter of a message is set by the originator of this message based on Lemma 3 (see Listing 2). More specifically, the originator resolves the minimal-level group it shares with the destination node (ll. 41-46; suppose the level of this group is i), and sets the TTL counter accordingly to $3^i - 1$.

If the message has not been dropped, the node determines how many elements of the destination label are left to be resolved (ll. 8-12). If there are no such elements left, then the present node is the destination and thus, it accepts the message (ll. 14-17). Otherwise, the message must

```
34 FUNCTION initMessage(dstLab, data) {
35
36   // create a new message
37   Message msg = new Message();
38   msg.dstLab = dstLab;
39   msg.data = data;
40
41   // compute TTL based on Lemma 3
42   int i;
43   for (i = 0; i < min(dstLab.len, this.lab.len; ++i) {
44     if (dstLab[i] == this.lab[i])
45       break;
46   }
47   msg.ttl = min(intpow(3, i) - 1, MAX_PATH);
48 }
```

Listing 2. The message initialization function.

be forwarded. As an optimization, the node first checks whether one of its neighbors is the destination node and if so, it forwards the message to this neighbor (ll. 19-22). If there are no such neighbors, the next hop is determined based on the routing table. More specifically, the present nodes looks up an entry for the next unresolved element of the destination, and forwards the message to the next hop neighbor associated with this entry (ll. 24-37). Finally, it may happen that due to hierarchy disturbance, the next hop cannot be resolved. In this case, the node drops the message (the main routing method returns *null*).

C Eventual Consistency Proof

PROOF *of Theorem 1:* Assume that after round r^* there are no more changes in the system. Let $\langle L(A)[i]\rangle_r$ denote the i-th element of the label of node A in round r. Moreover, we treat the first (starting from position 0) *null* value in the label as the end of the label.

Consider an arbitrary group G_X^i and an arbitrary node A. Node A is eventually a member of G_X^i iff:

$$\exists_{r_A^s}\ \forall_{r \geq r_A^s}\ \Big(\big(\langle L(A)[i]\rangle_r = X\big)\ \wedge\ \forall_{0 \leq j < i}\ \big(\langle L(A)[j]\rangle_r \neq null\big)\Big).$$

In other words there exists a *stabilization round*, r_A^s, in and after which A's label length is greater than i and the i-th element of A's label is always equal to X.

All eventual members of G_X^i constitute a set:

$$\{A \mid A \textit{ is eventually a member of } G_X^i\}.$$

Since this set is finite, we can choose the maximum stabilization round, r^s, of all its members. In other words, r^s denotes the round after which no nodes join or leave group G_X^i. We will use $\langle G_X^i\rangle_{r^s}$ to denote the stable set of nodes constituting group G_X^i.

Since the group membership is based on connectivity, all members of $\langle G_X^i\rangle_{r^s}$ form a connected graph. Note that if this was not true, then in some round a "hierarchy fault" violations would have been detected (see Section 5.3) and more changes in the system would have occurred, which contradicts our assumptions.

Assume that the last update performed by X at the $(i+1)$-st position of its label, as dictated by the responsibility rule, wrote $\square$ on that position. First, we show that each node in $\langle G^i_X\rangle_{r^s}$ will eventually have $\square$ at the $(i+1)$-st position of its label, as formalized by the following lemma.

Lemma 5 $\forall_{A\in\langle G^i_X\rangle_{r^s}}\ \exists_{r^\mu_A\geq r^s}\ \forall_{r\geq r^\mu_A}\ \big(\langle L(A)[i+1]\rangle_r = \square\big)$

The proof is performed by contradiction. Assume that there exists a node, $A\in\langle G^i_X\rangle_{r^s}$, that does not update its label with $\square$ at position $i+1$ in any round $\geq r^s$. We (virtually) mark A with Ⓢ. Consider, all the neighbors of A that also belong to $\langle G^i_X\rangle_{r^s}$. They too cannot update their labels with $\square$ in any round, because otherwise A would adopt the update in some round, as guaranteed by the algorithm from Section 5.1. Therefore, we mark these neighbors with Ⓢ as well. Now, consider the neighbors of these neighbors, and so forth. Since the graph of the members of $\langle G^i_X\rangle_{r^s}$ is finite and connected, at some point, we would have to mark X with Ⓢ, which means that X does not update its label with $\square$. **Contradiction!**, as X performs the update locally. Consequently, starting from some round r^μ_A, node A has $\square$ at the $(i+1)$-st position of its label. Because A was chosen arbitrarily, this condition holds for all nodes in $\langle G^i_X\rangle_{r^s}$. ■

Second, we will show that in some round the labels of any two members of $\langle G^i_X\rangle_{r^s}$ are identical starting from position i, as formalized below. This proves the eventual consistency.

$$\exists_{r^c}\ \forall_{A\in\langle G^i_X\rangle_{r^s}}\ \forall_{r\geq r^c}\ \big(\langle L(A)[i{+}]\rangle_r = \langle L(X)[i{+}]\rangle_r\big)$$

There are two possible values of $\square$: (1) $\square = null$ (X performed the label cut operation, see Fig. 6b); (2) $\square = Y$, where Y is an identifier of a level-$(i+1)$ head (X performed the label extension operation, see Fig. 6a).

(1) From Lemma 5, for each $A\in\langle G^i_X\rangle_{r^s}$, there exists a round r^μ_A, such that in any round $r\geq r^\mu_A$, $\langle L(A)[i+1]\rangle_r = null$. Let r^c denote the maximal value of r^μ_A for all $A\in\langle G^i_X\rangle_{r^s}$. In other words, for all $r\geq r^c$, any two members of $\langle G^i_X\rangle_{r^s}$ have identical values at position i (X) and position $i+1$ (*null*). Since the *null* value denotes the end of the label, the labels of all the members are identical starting from position i.

(2) From Lemma 5, for each $A\in\langle G^i_X\rangle_{r^s}$, there exists a round r^μ_A, so that in any round $r\geq r^\mu_A$, $\langle L(A)[i+1]\rangle_r = Y$. If r^{μ_1} denotes the maximal value of r^μ_A for all $A\in\langle G^i_X\rangle_{r^s}$, then for any $r\geq r^{\mu_1}$ and any $A\in\langle G^i_X\rangle_{r^s}$, we have: $\langle L(A)[i+1]\rangle_r = Y$, that is, $A\in\langle G^{i+1}_Y\rangle_r$. In other words, we showed that because starting at round r^s any two members of G^i_X have equal labels at position i, starting at round $r^u\geq r^s$ any two members of G^i_X have equal labels at position $i+1$.

We can repeat the whole reasoning for G^{i+1}_Y, and so forth, leading to a result that for each $k\geq 1$, there exists r^{μ_k} such that starting at round r^{μ_k} any two members of G^i_X have equal labels from position i to position $i+k$. Because in total a finite number of nodes in a finite number of rounds performed a finite number of updates, for some k we encounter situation (1), that is, for any $r\geq r^{\mu_k}$, for any $A\in\langle G^i_X\rangle_{r^s}$, we have: $\langle L(A)[i+k]\rangle_r = null$. Assuming $r^c = r^{\mu_k}$, for any $r\geq r^c$, the labels of all the members of $A\in\langle G^i_X\rangle_{r^s}$ are identical starting from position i.

This ends the proof. ■

D Proof of Lemma 4

PROOF *of Lemma 4:* Consider two arbitrary nodes, P and Q, that are heads of groups G^i_P and G^i_Q respectively. Suppose that P and Q must potentially spawn level $i+1$ groups.

Let r_P and r_Q denote the round in which P and Q respectively choose their virtual slots, as described in Section 5.3.2. Note that this implies that in round r_P, P knows about Q, and similarly, in round r_Q, Q knows about P. Let the number of slots $S=2$. Moreover, assume that the slot size, R, meets the requirements, that is, it is longer than the number of rounds necessary to propagate information between P and Q.

We have four possible slot selection configurations, each obtained with probability $\frac{1}{4}$, as presented in Fig. 10.

	I	II	III	IV
s_P	0	0	1	1
s_Q	0	1	0	1

Fig. 10. Slot selection configurations.

Without the loss of generality assume that $r_P\geq r_Q$, that is, P selects its slot in the same or later round than Q. Consider configuration III, in which P selects slot $s_P=1$ and Q selects slot $s_Q=0$. Let $r^*_P = r_P + s_P\cdot R+1$ denote the round in which P potentially spawns group G^{i+1}_P, as specified by the algorithm. Likewise, let $r^*_Q = r_Q + s_Q\cdot R+1$ denote the round in which Q potentially spawns group G^{i+1}_Q. We will show (by contradiction) that by the time it spawns group G^{i+1}_P, P discovers that Q spawned G^{i+1}_Q. Consequently, P can make G^i_P a subgroup of G^{i+1}_Q, decreasing the number of groups at level $i+1$.

To this end, assume that P spawns G^{i+1}_P in round r^*_P and Q spawns G^{i+1}_Q in round r^*_Q. Consider value $r^*_P - r^*_Q$ which denotes how many rounds after Q has spawned group G^{i+1}_Q, node P spawns group G^{i+1}_P.

$$\begin{aligned}
r^*_P - r^*_Q = \\
&= (r_P + s_P\cdot R+1) - (r_Q + s_Q\cdot R+1) = \\
&= r_P - r_Q + (s_P - s_Q)\cdot R = \\
&= r_P - r_Q + (1-0)\cdot R = \\
&= r_P - r_Q + R \geq^{(from:\ r_P\geq r_Q)} \\
&\geq r_P - r_P + R = \\
&= R.
\end{aligned}$$

From the above calculation P spawns group G^{i+1}_P at least R rounds after Q has spawned group G^{i+1}_Q. **Contradiction!**,

because within at most R rounds, P would have learned that Q spawned G_Q^{i+1}, and consequently, would have made G_P^i a subgroup of G_Q^{i+1}. Therefore, with probability at least $\frac{1}{4}$, G_P^i and G_Q^i will be subgroups of a common group $G_{P/Q}^{i+1}$. Because P, Q, G_P^i, and G_Q^i were chosen arbitrarily, Lemma 4 holds for any head node on any level. In practice, the aforementioned probability is higher than $\frac{1}{4}$. ■

E The Maintenance Algorithm

Each node running PL-Gossip reacts to two types of events: reception of a beacon message and periodical timeouts. Below, we describe these events in detail. We present the simplest version of the algorithm, without any optimizations.

E.1 Beacon Reception

Receiving a message (see Listing 3) allows a node to discover changes in the hierarchy and to update its routing table. A beacon message contains the label of the sender node with the corresponding update vector, (fragments of) the sender's routing table, and (a subset of) reverse link quality information of the sender's neighbors (necessary for forward link estimation, see Section 3). First, the node that received the message searches for the minimal common-level group it shares with the sender of the message (listing lines 3-7, see also Section 5.1). If such a group exists (ll. 9), the node compares its update vector with the sender's update vector to determine which of the two labels is more fresh (ll. 13-17, see also Section 5.1). If both the labels are fresh (ll. 19), the node only updates its routing table with the entries contained in the beacon message (ll. 20-21). If, however, the sender's label is more fresh (ll. 22), before updating its routing table (ll. 27-28), the node adopts that label as explained in Section 5.1 (ll. 23-26). Finally, if the sender's label is stale, the node can still use parts of the sender's routing table to update its own routing table (ll. 30-32).

If the node and the sender of the beacon message do not share any group (ll. 35), the node has just discovered a "hierarchy incomplete" violation (see Section 5.3.1). To propagate the information about this violation to the head of its top-level group, the node adds appropriate entries to its routing table, as explained in Section 5.3.1 (ll. 36-43). These entries will allow the head to react to the violation.

```
1  HANDLER onBeaconReceived(msg) {
2
3    // determine if we share any group
4    int i = 0;
5    for (; i < min(this.lab.len, msg.lab.len); ++i)
6      if (this.lab[i] == msg.lab[i])
7        break;
8
9    if (i < min(this.lab.len, msg.lab.len)) {
10     // we found a node that shares a group with us,
11     // so determine who has a more recent label
12
13     // find the minimal differing position
14     int j = i;
15     for (; j < min(this.lab.len, msg.lab.len); ++j)
16       if (this.uvec[j] != msg.uvec[j])
17         break;
18
19     if (j >= min(this.lab.len, msg.lab.len)) {
20       // we both have the same labels
21       this.rt.mergeWith(msg.rt, i − 1, msg.rt.topRow);
22     } else if (this.uvec[j] < msg.uvec[j]) {
23       // we are not up to date, so
24       // change our label and update vector
25       this.lab.copyFrom(msg.lab, j);
26       this.uvec.copyFrom(msg.uvec, j);
27       // merge routing tables
28       this.rt.mergeWith(msg.rt, i − 1, msg.rt.topRow);
29     } else {
30       // the other guy is not up to date, but we
31       // can still use a part of his routing table
32       this.rt.mergeWith(msg.rt, i − 1, j);
33     }
34
35   } else {
36     // we encountered a node from a completely
37     // different group (a hierarchy violation),
38     // so add it as a possible join candidate)
39     if (msg.lab.len >= this.lab.len) {
40       for (int k = this.lab.len − 1; k < msg.lab.len; ++i)
41         this.rt.addEntry(
42           k, msg.lab[k], msg.lab[0],
43           msg.rt[k][msg.lab[k]].hops + 1, true);
44   }
45 }
```

Listing 3. The handler of the beacon reception.

E.2 Periodical Timeout

The timeout event (see Listing 4) gives a node the opportunity to react to the changes in the system that occurred since the last timeout. First, the node removes stale entries from its routing table (listing lines 49-50), which enables detecting disruptive failures. More specifically, if the node,

being a level i head (where $i \geq 0$), is not the top-level head (ll. 55), it must check whether the central subgroup of its level $i+1$ group is still reachable and adjacent to the node's level i group (ll. 56-61), as explained in Section 5.3.3. If these conditions are not met (a "hierarchy fault" violation occurred), the node cuts its label down to level i (ll. 62-65), as described in Section 5.3.3. Otherwise, from the node's perspective, there were no disruptive failures in the system.

Second, if the node is the top-level head (ll. 72, possibly as a result of an earlier label cut), it must check whether the hierarchy construction is complete. To this end, the node first determines if its routing table contains entries for a level $i+1$ group it could join (ll. 73-76), as explained in Section 5.3.2. If this is the case (ll. 76), the node joins its level i group to the level $i+1$ group, by extending its label with the identifier of the head of this level $i+1$ group (ll. 77-81). It also cancels any possible pending suppression of label extension which corresponded to spawning a new group (ll. 82-83). As explained in Section 5.3.2, if the level $i+1$ group is itself a member of some higher-level groups, all members of the node's level i group will gradually extend their labels while exchanging beacon messages (ll. 22-28).

Even if an appropriate level $i+1$ group could not be found, it is still possible that the hierarchy is not complete. More specifically, the node must check whether its routing table contains any entries for other groups starting from level i (ll. 88). If so the node activates a suppression counter to defer spawning a new level $i+1$ group (ll. 99-103), as explained in Section 5.3.2. The suppression counter, once activated, is decremented during each timeout (ll. 107-108). When it reaches zero and the level $i+1$ group still has to be spawned (ll. 90), the node extends its label and cancels the counter (ll. 91-97), effectively spawning a new level $i+1$ group (with itself as the head of that group).

Finally, when the node reacted to all changes in the system, it broadcasts a beacon message (ll. 117-118), such that its neighbors can adopt any label updates and update the routes.

E.3 Remarks

When a node repaired after a failure rejoins the system, its membership decisions (label updates) made before the failure may still be present in the labels of other nodes. Therefore, it is crucial to ensure that any decision made by this node after the failure is perceived by other nodes as later than any decision made by this node before the failure. Otherwise, the ordering of label updates is not preserved, which disrupts the consistency enforcement algorithm. In that case, we cannot predict the behavior of the system.

To this end, whenever a node performs a label update it stores the new value of the update counter persistently, for instance, in the local flash memory (ll. 114-115). During boot, the node restores the last value of the counter from the persistent storage (see Listing 5, line 126), which ensures correct ordering of any subsequent membership decisions.

```
46  HANDLER onTimeout() {
47    int oldUcnt = this.ucnt;
48
49    // evict dead entries from the routing table
50    this.rt.ageAndClean();
51
52    int i = this.lab.getHeadLevel();
53
54    // check if we need to cut the label
55    if (i + 1 < this.lab.len) {
56      // we are not the top level head, so check if
57      // our superhead died or ceased to be adjacent
58      RtEntry centralSubgroupEntry =
59        this.rt[i][this.lab[i + 1]];
60      if (centralSubgroupEntry == null
61                  || !centralSubgroupEntry.isAdjacent) {
62        // perform the label cut operation
63        this.lab.cutTo(i);
64        this.uvec.cutTo(i);
65        this.uvec[i] = ++this.ucnt;
66      } else {
67        // our superhead works so there is nothing to do
68      }
69    }
70
71    // check if we need to extend the label
72    if (i + 1 == this.lab.len) {
73      // we are the top level head, so check if there is
74      // any same− or higher−level group we could join
75      JoinCandidate jc = this.rt.getJoinCandidate();
76      if (jc != null) {
77        // we have a group which we can join,
78        // so perform the label extension operation
79        this.lab.extendWith(jc.group);
80        this.uvec.extendWith(−INFINITY);
81        this.uvec[i] = ++this.ucnt;
82        // reset suppression counter
83        this.scnt = −1;
84      } else {
85        // we do not have such a group
86        if (this.scnt <= 0) {
87          // check if we need to extend the label
88          if (this.rt.hasOtherEntriesUpFrom(i)) {
89            // yes, we do have to extend the label...
90            if (this.scnt == 0) {
91              // our surrpression timer just fired,
92              // so perform the label extension
93              this.lab.extendWith(this.lab[0]);
94              this.uvec.extendWith(−INFINITY);
95              this.uvec[i] = ++this.ucnt;
96              // reset surrpression counter
97              this.scnt = −1;
98            } else {
99              // we have to activate the counter
100             this.scnt = selectRandSlot(i) ∗
101                 normalize(
102                   min(intpow(3, i), MAX_PATH),
103                   θ);
104           }
105         }
106       } else {
107         // our surrpression timer is ticking
108         −−this.scnt;
109       }
110     }
111   }
112
113
114   if (oldUcnt < this.ucnt)
115     save("UPDATE_CNT", this.ucnt);
116
117   // broadcast the beacon message
118   broadcastBeacon(this.lab, this.uvec, this.rt);
119 }
```

Listing 4. The periodical timer handler.

```
120 HANDLER onNodeBoot() {
121   // initialize
122   this.lab = {this.NODE_ID};
123   this.uvec = {−INFINITY};
124   this.rt = {}{};
125   this.scnt = −1;
126   this.ucnt = restore("UPDATE_CNT");
127
128   // set timer handler
129   setTimer(ΔT, &onTimeout);
130 }
```

Listing 5. The initialization handler.

Alternatively, a node rejoining the system obtains a new unique identifier which eliminates the problem completely.