Logical Neighborhoods: A Programming Abstraction for Wireless Sensor Networks

Luca Mottola and Gian Pietro Picco
Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy
{mottola,picco}@elet.polimi.it

Abstract. Wireless sensor networks (WSNs) typically exploit a single base station for collecting data and coordinating activities. However, decentralized architectures are rapidly emerging, as witnessed by wireless sensor and actuator networks (WSANs), and in general by solutions involving multiple data sinks, heterogeneous nodes, and in-network coordination. These settings demand new programming abstractions to tame complexity without sacrificing efficiency. In this work we introduce the notion of *logical neighborhood*, which replaces the physical neighborhood provided by wireless broadcast with a higher-level, application-defined notion of proximity. The span of a logical neighborhood is specified declaratively based on the characteristics of nodes, along with requirements about communication costs. This paper presents the SPIDEY programming language for defining logical neighborhoods, and a routing strategy that efficiently supports the communication enabled by its programming constructs.

1 Introduction

Wireless sensor networks (WSNs) typically exploit a single base station for collecting data and coordinating activities. Habitat monitoring [1], a common example application, is paradigmatic in this respect, featuring a *single* base station collecting data from a high number of *homogeneous* nodes. Nevertheless, decentralized architectures are rapidly emerging where multiple base stations are employed, different applications run on the same hardware, or heterogeneous nodes are deployed. These approaches find their extreme realization in wireless sensor and actor networks (WSANs) [2], where nodes not only gather data from the environment, but are also capable of affecting it by performing a variety of actions. Applications range from localization to control systems in tunnels or buildings, interactive museums, and home automation [3].

In contrast with mainstream WSNs, characterized by a single application gathering and reporting data, these decentralized settings are composed of many collaborating tasks, each affecting only a portion of the system. For instance, a WSAN for building control and monitoring can be decomposed in at least three main tasks, i.e., structural monitoring, in-door environment monitoring, and response to extreme events such as fire or earthquakes [4]. To realize the latter functionality, the nodes controlling water sprinklers must monitor nearby temperature sensors and smoke detectors and take appropriate measures when and *where* needed. Therefore, the application logic now resides *in the network*: including a central base station in the control loop degrades

system performance and reliability without any sensible advantage [2]. Dealing with this change of perspective demands new programming abstractions to tame complexity without sacrificing efficiency. Indeed, the developer is concerned not only with the application logic, but also with identifying the system portions to be involved and how to reach them. As no dedicated programming constructs exist for the latter task, the result is additional programming effort, increased complexity, and less reliable code.

This work tackles the aforementioned issues through the notion of *logical neighborhood*, an abstraction replacing the conventional notion of physical neighborhood—i.e., the set of nodes in the communication range of a given device—with a logical notion of proximity determined by applicative information. Logical neighborhoods are specified declaratively using the SPIDEY language, conceived to be a simple extension of existing WSN programming languages (e.g., nesC [5] in the case of TinyOS [6]). Using our enhanced communication API, a message can be broadcast to a logical neighborhood, instead of nodes within communication range. This way, application programmers still reason in terms of neighborhood relations and broadcast messages, but can now specify declaratively *which* nodes to consider as neighbors and, therefore, the span of communication. As such, our abstraction may foster a fresh look at existing mechanisms, algorithms, and programming models by replacing their conventional notion of physical neighborhood with our programmer-defined, logical one.

Clearly, our programming abstraction is ultimately of practical interest only in the presence of an appropriate and efficient routing mechanism supporting it. In principle, existing solutions can be exploited (e.g., [7]), but they exhibit various performance drawbacks, as they are based on different assumptions and scenarios. Therefore, in this paper we also present a novel routing protocol that is expressly devised to support our abstraction and leverages the kind of *localized interactions* [8, 9] characterizing the aforementioned decentralized scenarios. The evaluation included in this paper shows that indeed this routing protocol efficiently supports logical neighborhoods, therefore demonstrating the feasibility of our overall approach.

The rest of the paper is organized as follows. Section 2 describes the logical neighborhood abstraction and the SPIDEY language. Section 3 illustrates the novel routing strategy supporting our communication abstraction, while Section 4 evaluates its performance. Section 5 compares our approach against related work. Finally, Section 6 ends the paper with brief concluding remarks.

2 Programming Constructs for Logical Neighborhoods

The proposed abstraction revolves around only two concepts: *nodes* and *neighborhoods*.

A (logical) node is the application-level representation of a physical node, and defines which portion of its data and characteristics is made available by the programmer to the definition of any logical neighborhood. The definition of a logical node is encoded in a *node template*, which specifies the node's exported attributes. This is used to instantiate the (logical) node, by specifying the actual source of data. To make these concepts more concrete, Figure 1 (top) shows a SPIDEY code fragment that defines a template for a generic device and instantiates it by binding each template attribute to an expression of the target language, e.g., a constant or function. Template attributes

```
node template Device
  static  Function
  static  Type
  static  Location
  dynamic Reading
  dynamic BatteryPower

create node ts from Device
  Function as "sensor"
  Type as "temperature"
  Location as "room1"
  Reading as getTempReading()
  BatteryPower as getBatteryPower()
```

```
neighborhood template HighTempSens(threshold)
  with Function = "sensor" and
       Type = "temperature" and
       Reading > threshold

create neighborhood hts100
  from HighTempSens(threshold : 100)
  max hops 2
  credits 30
```

Fig. 1: *Sample node (top) and neighborhood (bottom) definition and instantiation.*

can be **`static`** or **`dynamic`**. The former represent information assumed to be time-invariant (e.g., the type of measurement a sensor provides), while the latter represents information changing with time, (e.g., the current sensor reading). The decision about whether an attribute is static or dynamic depends on the deployment scenario. Making the distinction explicit may enable optimizations at the routing layer, as discussed in Section 3.

A (logical) neighborhood is the set of nodes satisfying a constraint on the nodes' attributes. As with nodes, the definition of neighborhoods is encoded in a template, which contains a predicate that essentially serves as the membership function determining whether a node belongs to the logical neighborhood. For instance, the neighborhood template `HighTempSens` at the bottom of Figure 1 is based on the `Device` template in the same figure, and selects nodes that host temperature sensors and are currently reading a value higher than a given threshold. As exemplified in the SPIDEY code fragment, a neighborhood template can be parameterized, with the actual parameter values provided by expressions of the target language upon neighborhood instantiation. Moreover, the instantiation of a neighborhood template specifies additional requirements about *where* and *how* the neighborhood is to be constructed and maintained.

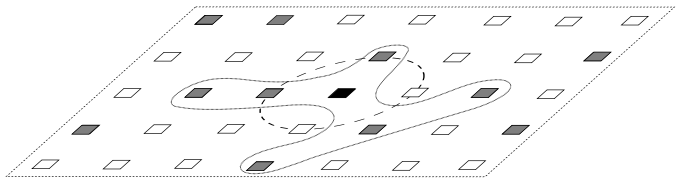


Fig. 2: *A visualization of logical neighborhoods.*

For instance, Figure 1 specifies that the predicate defined in the `HighTempSens` template is evaluated only on nodes that are at a maximum of 2 hops away and by spending a maximum of 30 "credits". The latter is an application-defined measure of cost, further detailed next, which enables the programmer to retain some control over the resources being consumed during the distributed processing necessary to deliver messages to members of a logical neighborhood. A pictorial representation of the example, visualizing the logical neighborhood concept, is provided in Figure 2. There, the black node is the one defining the logical neighborhood, and its physical neighborhood (i.e., nodes lying in its direct communication range) is denoted by the dashed circle. The dark nodes are those satisfying the predicate in the neighborhood template in Figure 1 (bottom) when the threshold is set to 100^{o}C. However, the nodes included in the actual neighborhood instance `hts100` are only those lying within 2 hops from the sending node, as specified through the **`hops`** clause during instantiation.

In essence, as graphically illustrated in Figure 3, templates define *what* data is relevant to the application, while the instantiation process constrains *how* this data should be made available by the underlying system. Separating the two perspectives has several beneficial effects. The same template can be "customized" through different instantiations. For instance, the very same template at the bottom of Figure 1 could be used to specify a logical neighborhood with a different threshold or a different physical span. Moreover, this distinction naturally maps on an implementation that maintains a neighborhood by disseminating its template to be evaluated against the values exported by a node instance, and uses instead the additional constraints specified at instantiation time to direct the dissemination process.

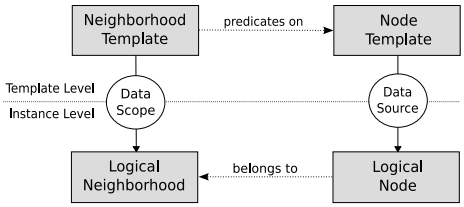


Fig. 3: *Templates and their instantiation.*

SPIDEY provides additional simple and yet expressive constructs. Logical operators such as **`and`**, **`or`**, and **`not`** are provided to define complex predicates on node templates. Moreover, as logical neighborhoods essentially identify sets of nodes, it becomes natural to express a neighborhood as a composition with already existing ones, using conventional set operators such as union, intersection, subtraction, and inclusion. Finally, the SPIDEY language contains also features enabling the creation of *virtual nodes*, built by binding node attributes to aggregation functions operating on a logical neighborhood. Virtual nodes spare programmers from the burden of directly handling the communication needed to gather and aggregate data from the neighborhood members, and can be used recursively to create higher-level abstractions. More details can be found in [10]. The complete grammar of the SPIDEY language is shown in Appendix A.

Our language also provides the ability to control the cost involved in communicating towards a neighborhood, through the **`credits`** clause. Communication cost is defined in terms of the basic operation of sending a broadcast message to physical neighbors (the node's *sending cost*), and is measured in *credits*. The mapping between cost and credits is specified by the programmer on a per-node basis through a **`use cost`** construct, which delegates the computation of this mapping to an expression of the target

language, e.g., a function. Therefore, the programmer can define a vast array of mappings, from a straightforward one where the sending cost is fixed, to sophisticated ones where it varies dynamically to adapt to context changes (e.g., low battery power). Moreover, different nodes can have different functions, e.g., yielding higher costs for tiny, battery-powered sensors, and lower costs for resource rich, externally-powered nodes. The overall number of credits necessary to communicate with the members of a logical neighborhood is evaluated as the sum of the costs incurred in by each node involved in routing, with each individual cost evaluated according to the function specified in the **`use cost`** declaration. Therefore, the ability to set the maximum amount of credits spent in communication in a logical neighborhood enables programmers to exploit different trade-offs between accuracy and costs. Neighborhoods endowed with many credits ensure a broader coverage but incur higher costs, while those with few credits may not reach all the specified nodes but limit resource consumption.

Logical neighborhoods must ultimately be used in conjunction with communication facilities, to enable interaction with the neighborhood members. On the other hand, the notion of logical neighborhood is essentially a scoping mechanism, and therefore is independent from the specific communication paradigm chosen. For instance, one could couple it with the tuple space paradigm to enable tuple sharing and access only within the realm of a logical neighborhood. In our current communication API we took the minimalist—and yet most general—approach of coupling logical neighborhoods with the standard broadcast-based message passing facility found in WSNs. As a result, our API includes simple `send` and `receive` operations mimicking those provided by the underlying operating system. For instance, our TinyOS implementation redefines the operations in the `GenericComm` module by extending the `send` operation with an additional parameter representing the logical neighborhood where a message must to be delivered, i.e., the scope of that particular message. Essentially, we are replacing the broadcast facility commonly made available by the operating system with one where message recipients are not determined by the physical communication range, rather by membership in a programmer-defined logical neighborhood. In addition, a `reply` primitive is also included to simplify communication from neighborhood members back to the message sender. To enable this degree of generality and flexibility, it is fundamental for our abstraction and API to be supported by efficient routing strategies. A description of our solution to the routing problem is described in the next section.

3 Routing for Logical Neighborhoods

The logical neighborhood abstraction is essentially independent of the underlying routing layer. Nevertheless, its characteristics cannot be easily accommodated by existing data-centric routing approaches. Indeed, these are usually conceived to solve the problem of data collection from a homogeneous nodes, thus focusing on how to collect efficiently the data from many sensors to a single node—the sink. In our approach the perspective is reversed: routing must efficiently transmit an application message from a single node (the sender) to those matching the neighborhood specification. Moreover, logical neighborhoods are a scoping mechanism, and therefore can be used in conjunc-

tion with several mechanisms other than data collection, e.g., to direct code updates only towards nodes with obsolete versions. As such, some of the techniques exploited by these protocols (e.g., route reinforcement based on data rates as in [7]) not only cannot be directly applied, but are actually complementary to ours. Moreover, our goal is to devise a protocol that captures the localized interactions that should characterize communication in decentralized, multi-sink WSNs and WSANs. This rules out solutions exploiting system-wide tree overlays as in TinyDB [11]. Finally, credit management is a distinctive feature of our approach that would anyway require appropriate integration.

Motivated by these considerations, this section describes a routing strategy designed to support efficiently and effectively the logical neighborhood abstraction. Our routing approach is *structure-less* (i.e., no overlay is explicitly maintained) and is based on the notion of *local search* [12]. Nodes advertise their *profile*, i.e., the list of attribute-value pairs specified by their template, and in doing so build a distributed state space containing information about the cost of reaching a node with given data. This information dissemination is localized and governed by the density of devices with similar profiles. Messages sent to a neighborhood contain its template, which determines a projection of the state space, i.e., the part to be considered for matching. In a nutshell, the message "navigates" towards members of a neighborhood by following paths along which the cost associated with a given neighborhood template is decreasing. The proposed routing approach is therefore composed of two parts: the *state space generation* and the *search algorithm*.

3.1 Building the State Space

In our scheme, whenever a new node is added to the system it broadcasts a PROFILEADV message containing the node identifier, a (logical) timestamp, the node's profile containing attributes and their values, and a cost field initialized to zero. The first two message fields are used to discriminate stale information, as the PROFILEADV message is periodically re-broadcast (possibly with different content) by the node. An example PROFILEADV is reported in Figure 4.

Source	Timestamp	Node Profile		Cost
		Attribute	Value	
N54	72	`Function`	*sensor*	2
		`Type`	*temperature*	
		`Location`	*room123*	

Fig. 4: *An example of* PROFILEADV.

In addition, each node in the system stores a *State Space Descriptor* (SSD) containing a summary of the received PROFILEADV messages. An example is shown in Figure 5. The *Attribute* and *Value* fields store information previously received through a PROFILEADV message. For each entry, *Cost* contains the minimum cost to reach a node with the corresponding information, and *Source* contains the identifier of such node. The *Links* field allows to store information more compactly, by retaining associations among entries instead of duplicating them in the SSD. In Figure 5 each entry is linked to the others as they all come from the same PROFILEADV advertised by node N8. *DecPath* and *IncPaths* contain routing information to direct the search process, as described in Section 3.2. Finally, each entry in an SSD is associated with a lease (not shown), whose expiration causes the removal of the entry not refreshed by a new PROFILEADV.

Upon receiving a PROFILEADV message, a node first updates the cost field in the message by adding its own sending cost, obtained by evaluating the expression in the

Id	Attribute	Value	Cost	Links	DecPath	IncPaths	Source
1	`Function`	*sensor*	5	2,3	N37	N98, N99	N8
2	`Type`	*acoustic*	5	1,3	N37	N98, N99	N8
3	`Location`	*room123*	5	1,2	N37	N98, N99	N8

Fig. 5: *An example of* State Space Descriptor (SSD).

`use cost` statement described in Section 2. Then, it compares each attribute-value pair in the message against the content of the local SSD. A modification (entry insertion or update) of the SSD is performed if an attribute-value pair: i) does not exist in the local SSD, or ii) it exists with a cost greater than the one in the message (after the local update above). The update (or insertion) of an SSD entry involves establishing the proper values in the *Links* field to keep track of the rest of the PROFILEADV message, updating the *DecPath* field with the identifier of the physical neighbor that sent the PROFILEADV, and setting the *Source* field to the identifier of the node whose information has been inserted in the PROFILEADV. For instance, assume the node storing the SSD in Figure 5 has a sending cost of 1, and receives the PROFILEADV in Figure 4. Its local SSD is then updated as described in Figure 6 (changes shown in bold). Note how the *Links* fields are updated so that only the minimum cost to reach an entry is kept, and yet the information about which entry came with which profile is not lost.

After a PROFILEADV has been processed locally, it is rebroadcast *only* if at least one SSD entry was inserted or updated, to propagate the state change. An example is shown in Figure 7(a). The PROFILEADV is rebroadcast as received, except for the updated *Cost* and *Source* fields. Interestingly, the propagation of PROFILEADV messages enables a node to determine if it lies, for some attribute-value pair, on a path where costs are increasing. This occurs when a PROFILEADV is overheard, through passive listening, with a cost greater than the corresponding pivot entry in the SSD. In this case, the identifier of the broadcasting node is inserted in the *IncPaths* field of the pivot entry.

Thus far, we assumed that PROFILEADV messages contain the whole node profile. Nevertheless, if some dynamic attribute changes frequently, there is a trade-off between the network load necessary to refresh the advertisements and the accuracy of the information being propagated. A straightforward alternative approach is to disseminate only part of the profile (e.g., static attributes) and perform additional matching at the receiver. These trade-offs are ultimately solved based on the characteristics of the deployment scenario, e.g., by considering information about the size of the logical neighborhood or the network density.

Finally, note how, as shown in Figure 7(a), profile advertisements do *not* flood the entire network, as a PROFILEADV is rebroadcast only upon an SSD update. Flooding occurs only for the first advertisement, or more generally when only one node contains a given attribute-value pair—a rather unusual case in the scenarios we target. Instead,

Id	Attribute	Value	Cost	Links	DecPath	IncPaths	Source
1	`Function`	*sensor*	**3**	3,**4**	**N77**	N98, N99	**N54**
2	`Type`	*acoustic*	5	1,3	N37	N98, N99	N8
3	`Location`	*room123*	**3**	1,**4**	**N77**	N98, N99	**N54**
4	**`Type`**	***temperature***	**3**	**1,3**	**N77**	-	**N54**

Fig. 6: *The SSD of Figure 5 at a node with a sending cost of* 1*, after receiving the* PROFILEADV *message in Figure 4.*

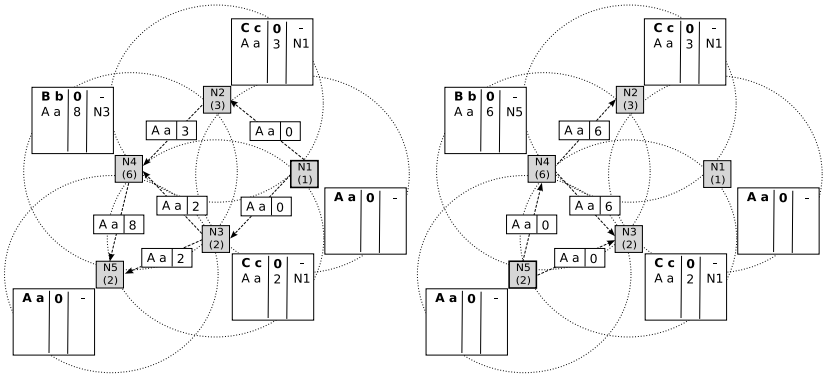
for a given set of attribute-value pairs, the state space generation builds a set of non-overlapping *regions*, each containing a node with the considered information. Within a region, each node knows how to route a message addressed to a neighborhood template that includes attributes matching those of a node, along the routes stored in *DecPath*. Each region can be regarded as a "concavity" defined by costs in SSDs, with the target node at the bottom (cost to reach it is zero) and nodes with increasing costs around it. This is illustrated in Figure 7(b), where we show the SSDs after all the nodes performed at least one profile advertisement. Next we describe how this distributed state space is exploited for routing.

3.2 Finding the Members of a Logical Neighborhood

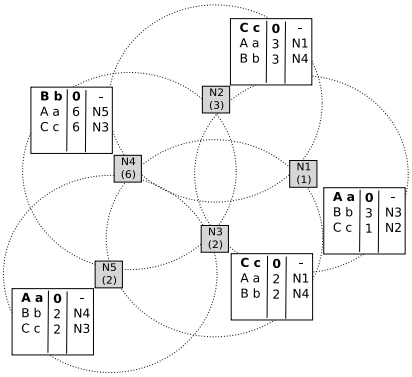
Local search procedures proceed step by step with subsequent *moves* exploring the state space [12]. At each step, a set of further local moves is available to proceed in the search process. Among them, some moves are accepted and generate further moves, while the remaining ones are simply discarded. In general, accepting moves depends on the heuristics one decides to employ given the particular problem tackled. In our case, a *move* is simply the sending of an application message containing the neighborhood template. Upon receiving a message, the move is accepted and further send operations are performed if the maximum number of hops, if any, has not been reached (as per the **`hops`** construct), and either i) the move proceeds along a decreasing path, or ii) enough unreserved credits are available on an exploring path. The notions of *decreasing path*, *exploring path* and *credit reservation* are at the core of our routing solution and are described next.

Decreasing paths. A path is *decreasing* if it gets the message closer to nodes whose profile matches the neighborhood template. To do so, message proceeds towards minima of the state space by traversing nodes that report an always smaller cost to reach a potential neighborhood member.

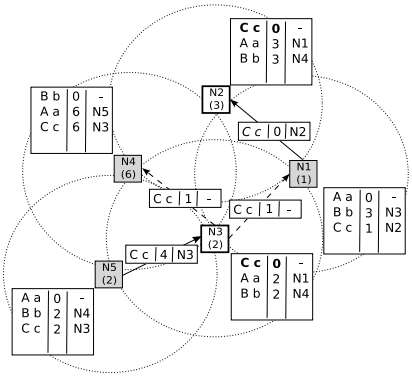
To determine decreasing paths, a node must identify the state space projection determined by a neighborhood template. To this end, the node finds in the local SSD the entry matching the neighborhood template with the greatest cost, if any. This entry is called *pivot*. If a pivot exists and is associated, via the SSD *Links* field, to a set of other entries matching the neighborhood template, the cost associated to the pivot represents the number of credits needed to reach the closest matching node via the path indicated by the *DecPath* field. For instance, imagine the application issues a `send(m,n)` operation through our enhanced communication API, to send the application message `m` to the neighborhood `n`, and assume `n` is defined to address all acoustic sensors. This neighborhood has its *pivot* in entry 2 of the SSD in Figure 6, and its predicate (`Function` $=$ $sensor$ **and** `Type` $=$ $acoustic$) is matched via the link pointing from entry 2 to entry 1. Consequently, the node evaluates the cost to reach the closest acoustic sensor as 5 and forwards the message towards N37. Due to the state space generation process, messages following a decreasing path are certainly forwarded towards nodes matching the neighborhood template. Indeed, these paths simply follow the reverse paths previously setup by PROFILEADV messages originating from nodes whose profile contains information matching the neighborhood template. Additionally,



(a) Building the state space (time goes from left to right). Arrow labels denote sending of PROFILEADV messages, showing only the attribute-value (e.g., A a), and Cost fields. SSDs are shown with only attribute-value, Cost and DecPath fields. After N1 disseminated its profile, N5's PROFILEADV need *not* be propagated system-wide, but only where updates in SSDs are needed to make its presence known.



(b) After all the nodes performed at least one profile advertisement, the SSDs contain the costs to reach the closest node with a given attribute-value pair.



(c) A message navigating the state space: dashed lines represent exploring directions, solid lines denote decreasing paths. Arrow labels represent application messages showing only the unreserved credits and the intended recipient.

Fig. 7: *Building and navigating the state space.*

note how the reply feature provided by our communication API can be implemented trivially by keeping track of the reverse path along which a message is received.

Exploring paths. If a message were to follow decreasing paths only, it would easily get trapped into local minima of the state space. To avoid this, we allow messages to be propagated also along *exploring paths*, i.e., directions where the cost to reach the closest node with a particular attribute-value pair is non-decreasing. Exploring paths include directions where the cost does not change (e.g., at the border between two regions)

or where it increases. The latter are stored in the *IncPaths* SSD field, as discussed in Section 3.1.

Activating multiple exploring paths at each hop is ineffective, as it is likely to generate many routes that are shortly after rejoined. Therefore, exploration proceeds along a single increasing path, if available. Exploration on multiple paths, achieved through physical broadcast, is activated only when the message reaches a neighborhood member (i.e., a minima of the state space), or after the message has travelled for E hops, with E being a tunable protocol parameter. This design choice stems from the observation that increasing paths are key in enabling the message to "escape" local minima by directing it towards the boundary where a region confines with a different one, and a different decreasing path may become available.

Credit reservation. The instantiation of a neighborhood template may specify the credits to be spent for communicating with neighborhood members, as discussed in Section 2. To support this feature, the number of credits is appended by the sender to every application message sent to a given neighborhood. A node may decide to split these credits in two: one part *reserved* to be spent along decreasing paths and the other along exploring ones. The splitting occurs at the first node that identifies a decreasing path for the message being routed, and is effected by removing the reserved credits from the amount in the message, therefore effectively reserving the credits along the entire decreasing path. For instance, Figure 7(c) shows a message sent by N5 with 6 credits, targeting a neighborhood defined by a single predicate $\texttt{C} = c$. Neighborhood members are shown in white. As the pivot in N5's SSD reports a cost of 2 to reach the node N3 matching the predicate, the message is forwarded to N3 with 4 unreserved credits.

To deal with credit reservation, a node checks whether its identifier is inserted in the message by the sender node as the next hop along a decreasing path towards a matching node. If so, the node simply forwards the message to the next hop on the decreasing path (found in its SSD) without modifying the credit field, since the necessary credits have already been reserved by the first node on the decreasing path. Otherwise, if exploring paths are to be followed, the node "charges" the message for the number of credits associated to the node sending cost, as per the **`use cost`** declaration. The remaining (unreserved) credits are assigned to the exploring paths the local node decides to proceed on. Normally, all these credits are assigned to the single message forwarded along the increasing path. However, if multiple paths are explored in parallel through broadcast, according to the heuristics described above, the unreserved credits are divided by the number of neighbors before broadcasting the message. In Figure 7(c), N3 receives a message with 4 remaining credits. Since it is a neighborhood member, the message must be broadcast along all the available exploring paths. Therefore, N3 charges the message for its own sending cost (2) and divides the remaining credits by the number of its physical neighbors. This results in activating two exploring directions, each with a 1-credit budget.

4 Evaluation

This section reports about an evaluation of our routing protocol for logical neighborhoods. To this end, we implemented it on top of TinyOS [6] and evaluated it using the

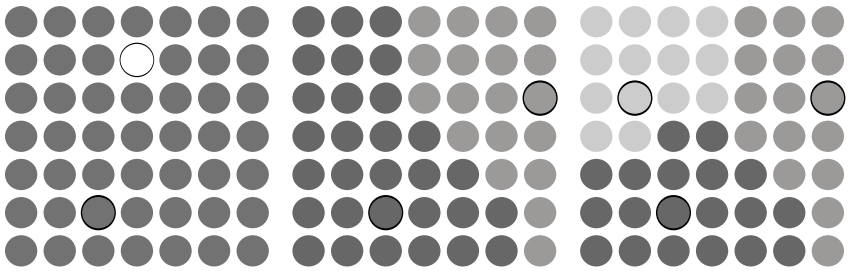


Fig. 8: *State space generation. The first* PROFILEADV *message spreads throughout the system as no node disseminated its profile yet. Profiles advertised by other nodes propagate only until a smaller cost is encountered, partitioning space in regions centered on neighborhood members. Note how the white node does not receive the message in the first propagation—due to collisions—but eventually receives it in later retransmissions.*

TOSSIM [13] simulator. Our goals were to verify that the protocol behaved as expected for what concerns the generation of the state space and the cost-aware routing of messages, and to characterize its performance. Clearly, this is key to assess the feasibility of our approach and abstraction. The deployment scenario we simulated is a grid where each node can communicate with its four neighbors. This choice not only simplifies the interpretation of results by removing the bias induced by more unstructured scenarios, but also models well some of the settings we target, e.g., indoor WSN deployments [14].

Analyzing the Routing Behavior. Before characterizing the performance of our routing protocol, we analyze whether its behavior matches our design criteria. First, we verify separately the two basic mechanisms underlying our routing, i.e., the state space generation and its "navigation" by applicative messages addressed to a logical neighborhood. As for the former, the key property we want to verify is that the propagation of PROFILEADV messages is localized and partitions the system in non-overlapping regions, each with routing information towards a neighborhood member.

To simplify the analysis of results we developed a simple visualization tool that, given a simulation log and a neighborhood template, displays the propagation of PROFILEADV as well as applicative messages. Figure 8 shows a sample output of our tool where the logical neighborhood we consider selects three members (represented as circled nodes) based on their profiles, and the node sending cost is equal for all devices. The three snapshots correspond to the points in time when a given PROFILEADV, generated by one of the neighborhood members, has ceased to propagate. As it can be observed, the first PROFILEADV is propagated in the whole system, as no other profile information exists yet. However, when the second member propagates its profile, this is spread only until it reaches a node where the cost is less than the one in PROFILEADV. This process partitions the state space in two non-overlapping regions. Eventually, the system reaches a stable situation where the number of regions is equal to the number of neighborhood members, as shown in Figure 8—right.

For what concerns routing of applicative messages, Figure 9 shows the output of our visualization tool when a message is sent to the same neighborhood of Figure 8. The credits associated to the neighborhood are set as an over-approximation of the

credits needed to reach the same three nodes along the shortest path. (More details about setting credits are reported later.) Note how the one in the picture is a worst-case scenario where the sender belongs to the same neighborhood the message is addressed to. In this situation, the message starts from a minimum of the state space, i.e., without any decreasing path. Therefore, the initial moves must be exploring ones, until a region different from the one where the message originated is reached. Despite this unfavorable initial situation, the message reaches all the intended recipients by alternating moves along decreasing paths with exploration steps.

The effectiveness of our mechanisms in reducing communication costs is unveiled when heterogeneous devices with different sending costs are deployed. Figure 10 shows a situation with a single neighborhood member and a message sender placed at the opposite corners of the grid, and where sending costs are assigned according to an integer approximation of a bi-dimensional Gaussian distribution. The figure shows the message dutifully steering away from the network center, where sending costs are higher, and striking a balance between the length of its route and the sending cost of the nodes on it. Thanks to the way our state space is generated through profile advertisements and SSD updates, this path is guaranteed to be, within a region, the one with the minimum cumulative sending cost.

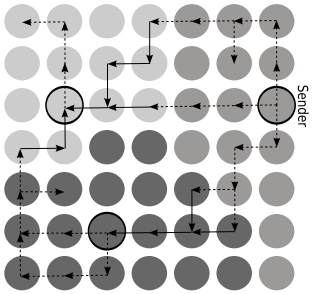


Fig. 9: *An applicative message navigates the state space. Solid lines are decreasing paths, dashed lines are exploring paths.*

Performance Characterization and Comparative Evaluation. Next, we wanted to study the performance of our protocol. Therefore, we defined a set of synthetic scenarios with a variable number of nodes placed 35 ft apart and with a communication range[1] of 50 ft. Each run lasted 1000 s—a value for which we verified all the measures exhibit a variance less than 1%. In dynamic scenarios, this approach provides more precise results than only averaging over multiple runs [15].

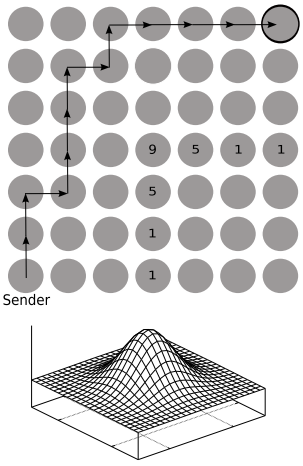


Fig. 10: *A message navigating a state space where sending costs follow the distribution at the bottom.*

Each node is configured with a single (static) attribute whose value is randomly chosen from a predefined set $\mathcal{A}$ at system start-up. This profile is disseminated by PROFILEADV messages once every 15 s. A single sender node is placed in the center of the grid, generating applicative messages at the rate of 1 msg/s towards a single neighborhood defined with an equality predicate over the node profiles. In this setting, the number of receivers is determined by $|\mathcal{A}|$, and in our case yields a number of neighborhood members of about 10% of the nodes in the system. The node sending cost is constant and identical throughout the system.

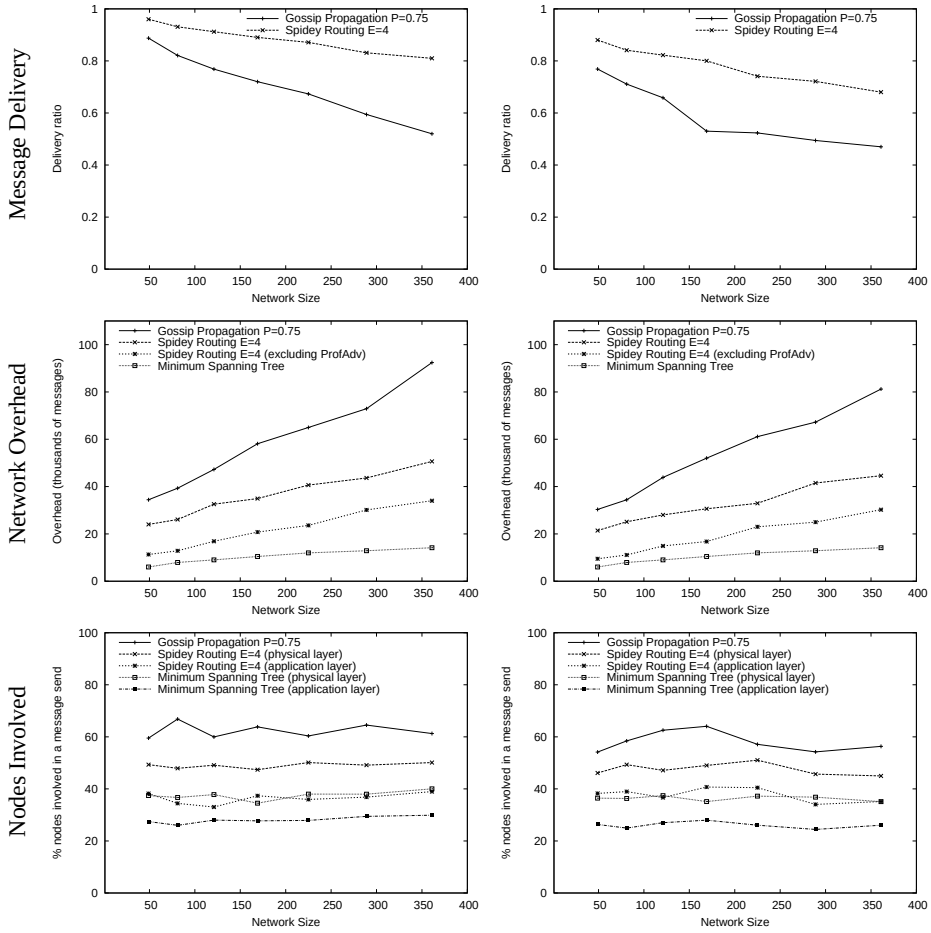
[1] We used the TinyOS' `LossyBuilder` to generate topology files with transmission error probabilities taken from real testbeds.

Credits are assigned by computing the average cost to reach each node in the system along the shortest path and weighing this value by the probability of the node being a receiver. Then, we increased this minimal value by about one third, to give each message some extra credits to spend on exploratory paths. This approach clearly overestimates the actual cost to reach a receiver, e.g., because it does not consider that two receivers may share part of the path from the sender. The definition of a model supporting fine-tuning of credit assignment to neighborhoods deserves further investigation based on the large body of literature on ad-hoc network density and random graph theory, and is our immediate research goal.

In the absence of directly applicable solutions to compare against, we chose a gossip approach as a baseline, because it is general enough to address the characteristics of our scenarios (e.g., lack of knowledge about the nature of applicative data) and yet generates less traffic than a straightforward flooding protocol. We set the protocol parameters so that gossip rebroadcasts a packet received for the first time with a probability $P = 0.75$, and our solution triggers new exploring directions once every $E = 4$ hops. This latter choice is a reasonable trade-off between generating too many redundant exploratory paths (E too small) and never activating exploratory paths within a region ($E > d$, with d the region diameter).

We based our evaluation on three metrics, namely i) the *message delivery ratio*, defined as the ratio between the messages received by neighborhood members and those that have actually been sent; ii) the *network overhead*, defined as the overall number of messages exchanged at the MAC layer, thus including PROFILEADV messages; and (iii) the average number of *nodes involved* in routing. This figure is further divided into the nodes processing a message at the MAC layer, and those processing one at the application layer. Message delivery is a measure of how effectively a protocol steers messages towards the intended recipients. On the other hand, in the absence of a precise model to evaluate a node's power consumption, ii) and iii) provide a sense of how a protocol exploits communication and computational resources, respectively.

Figure 11 illustrates our simulation results along the aforementioned metrics and w.r.t. the network size. Each chart is the average result of 5 different runs. As it is clear from the figures, our protocol outperforms gossip in all metrics. Message delivery is consistently higher than in gossip, and is even significantly less sensitive to an increase of the network size. As for network overhead, we provide additional insights by showing the results for our protocol with and without PROFILEADV advertisements, and by comparing against the ideal lower bound provided by routing along the minimum spanning tree rooted at the sender and connecting all neighborhood members (computed with a global knowledge of network topology). The chart evidences that we generate almost half of the overhead of gossip and yet deliver significantly more messages. The gap between the two is even more evident in the curve without the PROFILEADV messages, which essentially highlights how efficient is the pure routing mechanism, once the routing information is in place. This is particularly significant because the dissemination of PROFILEADV during state space generation is a fixed cost that is paid once and for all. In other words, adding another sender—regardless of the neighborhood it addresses—does *not* increment the overhead due to state space generation. In addition, the chart shows how the performance of our protocol in this case is closer to the ideal



(a) Static Network (b) Dynamic Network (failures)

Fig. 11: *Evaluation against gossip and ideal multicast, in static and dynamic scenarios.*

lower bound than to gossip. Finally, for what concerns the nodes involved in processing, Figure 11 shows that our performance at MAC layer is in between gossip and the minimum spanning tree, while at the application layer our routing requires only about half of the nodes exploited by gossip to process application messages and exhibits a performance closer to the minimum spanning tree. Therefore, our protocol is likely to provide a considerably longer network lifetime, although a precise characterization of the energy consumption is beyond the scope of this paper. This result is due to our guided exploration process, which privileges unicast messages (that, unlike broadcast, do not reach the application layers at all nodes in range), thus saving processing. In contrast, gossip explores the system in a completely "blind" way.

As shown in the right column of Figure 11, the evaluation was carried out also in a more dynamic scenario where 10% of the nodes are randomly turned off for 30 s and

then reactivated without allowing any settling time in between. Clearly, we excluded from this random selection the intended message recipients, as this would irremediably impact the message delivery ratio. A similar setting has already been used in existing works on routing for WSNs (e.g. [7]) to simulate node failures or the addition of new nodes. As Figure 11 shows, our protocol still provides higher delivery than gossip at lower communication and computational costs, despite node failures. In particular, although nodes joining or leaving the system generate additional profile advertisements to change the shape of the state space, the network overhead remains far from the one of gossip. This result is due to the ability of the state space to change its shape very rapidly in response to network topology changes. For instance, a single PROFILEADV message dissemination among nodes in close proximity to the changing one is usually all it is needed to restore a stable situation.

Finally, the results illustrated in this section should be regarded as worst-case. Indeed, not only the credit assignment can likely be fine-tuned to waste less resources, but also our choice of neighborhood predicates (single disjuncts) is restrictive. Indeed, it forces each message to follow at most a single decreasing path at a time: neighborhood templates containing multiple elementary disjuncts instead can be routed more accurately by exploiting multiple decreasing paths, therefore further increasing delivery. Moreover, setting uniform costs throughout the system does not leverage the ability of our protocol to route in a cost-aware fashion. Nevertheless, we chose these settings to be fair to gossip, which does not provide these advanced capabilities.

5 Related Work

Only few works propose distributed abstractions for WSNs that support some notion of scoping. Moreover, unlike the strongly decentralized scenarios we target in this work, many assume a single data sink.

The work closer to ours is the neighborhood abstraction described in Hood [16], where each node has access to a local data structure where attributes of interest provided by (physical) neighbors are cached. However, only homogeneous nodes are assumed. Moreover, data collection is built into the constructs and therefore, as stated in Section 3, communication is expected to flow only according to a many-to-one paradigm. Finally, the current implementation considers only 1-hop neighbors and is mainly based on broadcasting all attributes and performing filtering on the receiver's side. Clearly, our framework is much more flexible as it provides a general application-defined neighborhood abstraction, which is decoupled from the application functionality and therefore can be used for purposes other than data collection (e.g., network reprogramming), as well as in conjunction with it to support efficiently heterogeneous scenarios.

The work on Abstract Regions [17], instead, proposes a model where <*key,value*> pairs are shared among the nodes belonging to a given *region*. The span of a region is based mainly on physical characteristics of the network (e.g., physical or hop-count distance between sensors), and its definition requires a dedicated implementation. Therefore, each region is somehow separated from others, and regions cannot be combined. This results in a much lower degree of orthogonality and flexibility with respect to our approach. Moreover, the concept of *tuning interface* provides access to a region's

implementation, enabling the tweaking of low-level parameters (e.g., the number of retransmissions). Instead, our approach provides a higher-level, user-defined notion of cost that can be used to control resource consumption. In TinyDB [11], materialization points create views on a subset of the system. In this sense, common to our work is the effort in providing the application programmer with higher-level network abstractions. However, the approach is totally different, as TinyDB forces the programmer to a specific style of interaction (i.e., a data-centric model with SQL-like language) and targets scenarios where a single base station is responsible for coordinating all the application functionality. SpatialViews [18] is a programming language for mobile ad-hoc networks where *virtual networks* can be defined depending on the physical location of a node and the services it provides. Computation is distributed across nodes in a virtual network by migrating code from node to node. Common to our work is the notion of scoping virtual networks provides. However, SpatialViews targets devices much more capable than ours, focuses on migrating computation instead of supporting an enhanced communication facility as we do, and yet provides less general abstractions. Finally, in [19], the authors propose a language and algorithms supporting generic role assignment in WSNs with an approach that, in a sense, is dual to ours. In fact, their approach *imposes* certain characteristics on nodes in the system so that some specified requirements are met, while in our approach the notion of logical neighborhood *selects* nodes in the system based on their characteristics.

As for our routing protocol, we were influenced by Directed Diffusion [7] in using a soft-state approach based on periodic refresh for storing routes. However, our solution is radically different as it targets much more general scenarios. We do not assume data collection as the main communication functionality, and therefore we cannot rely on any knowledge about message content, required in Directed Diffusion for interpolation along failing paths. Similarly, we take into account an explicit notion of communication cost without relying on an application-defined notion of data rate. Moreover, an important difference is that our profile advertisements do not propagate to the whole network, unlike interests in Directed Diffusion. Finally, routing in Directed Diffusion is entirely determined by gradients, while we make the system more resilient to changes by allowing exploratory steps, whose use is nevertheless under the control of the programmer through the credit mechanism.

6 Conclusions and Future Work

This paper presented the SPIDEY language and a routing protocol supporting logical neighborhoods, a novel programming abstractions for WSNs. Logical neighborhoods capture sets of nodes with functionally related characteristics. SPIDEY constructs enable the programmer to specify neighborhoods declaratively, and yet control the trade-off between accuracy and resource consumption using an application-defined notion of cost. This latter information is used by our dedicated routing protocol, which supports efficiently our abstraction.

The benefits of our proposal impact two orthogonal aspects. First, developers can concentrate on the actual application goals while relying on logical neighborhoods as a way to logically partition the system and interact with it. We conjecture that applications

built on top of our abstraction result in cleaner, simpler, and more reusable implementations. A qualitative and quantitative evaluation of the advantages our approach brings to the development task is currently being carried on. Second, our routing protocol achieves a longer system lifetime and a better resource utilization, by focusing only on the nodes that actually need to be involved.

In this paper, we coupled logical neighborhoods with the broadcast-based primitives typically provided by the operating system. As we pointed out, this choice simplifies the programmer's task, and opens up opportunities for adapting existing techniques by replacing physical with logical neighborhoods. Our future research goals involve the coupling of logical neighborhoods with different services (e.g., to support code deployment only in given portions of the system) as well as alternative communication paradigms. In particular, we plan to integrate logical neighborhoods with our tuple space middleware TINYLIME [20] supporting scenarios with multiple mobile sinks, to empower sinks with the ability to restrain data sharing to the desired set of nodes. Interestingly, this scenario is easily encompassed by our routing protocol, as routes are determined by the profiles of (static) sensors rather than the requests of (mobile) sinks. Finally, our immediate research goal is to devise an analytical model of our routing protocol, to provide the programmer with the ability to properly dimension the allocated credits based on the characteristics of the network, e.g., in terms of density and connectivity.

Acknowledgements. The work described in this paper is partially supported by the Italian Ministry of Education, University, and Research (MIUR) under the VICOM project, by the National Research Council (CNR) under the IS-MANET project, and by the European Union under the IST-004536 RUNES project.

References

1. Mainwaring, A., Culler, D., Polastre, J., Szewczyk, R., Anderson, J.: Wireless sensor networks for habitat monitoring. In: Proc. of the 1^{st} ACM Int. Workshop on Wireless sensor networks and applications. (2002) 88–97
2. Akyildiz, I.F., Kasimoglu, I.H.: Wireless sensor and actor networks: Research challenges. Ad Hoc Networks Journal **2**(4) (2004) 351–367
3. Petriu, E., Georganas, N., Petriu, D., Makrakis, D., Groza, V.: Sensor-based information appliances. IEEE Instrumentation and Measurement Mag. **3** (2000) 31–35
4. Dermibas, M.: Wireless sensor networks for monitoring of large public buildings (2005) Tech. Report, University of Buffalo. Available at `www.cse.buffalo.edu/tech-reports/2005-26.pdf`.
5. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC language: A holistic approach to networked embedded systems. In: Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'03). (2003) 1–11
6. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. In: ASPLOS-IX: Proc. of the 9^{nt} Int. Conf. on Architectural Support for Programming Languages and Operating Systems. (2000) 93–104
7. Intanagonwiwat, C., Govindan, R., Estrin, D., Heidemann, J., Silva, F.: Directed diffusion for wireless sensor networking. IEEE/ACM Trans. Networking **11**(1) (2003) 2–16
8. Estrin, D., Govindan, R., Heidemann, J., Kumar, S.: Next century challenges: scalable coordination in sensor networks. In: Proc. of the 5^{th} Int. Conf. on Mobile computing and networking (MobiCom). (1999)

9. Qi, H., P.T. Kuruganti: The development of localized algorithms in wireless sensor networks. Sensors Journal **2**(7) (2002)
10. Mottola, L., Picco, G.: Programming Wireless Sensor Networks with Logical Neighborhoods. In: Proc. of the the 1^{st} Int. Conf. on Integrated Internet Ad hoc and Sensor Networks (InterSense 2006), Nice (France) (2006) (Short paper). To appear. Available at `www.elet.polimi.it/upload/picco`.
11. S.R. Madden, M.J. Franklin, J.M. Hellerstein, Hong, W.: TinyDB: an acquisitional query processing system for sensor networks. ACM Trans. Database Syst. **30**(1) (2005) 122–173
12. L.A. Wosley: Integer Programming. Wiley (1998)
13. Levis, P., Lee, N., Welsh, M., Culler, D.: Tossim: accurate and scalable simulation of entire tinyos applications. In: Proc. of the 1^{st} Int. Conf. on Embedded Networked Sensor Systems (SenSys). (2003) 126–137
14. Stoleru, R., J.A. Stankovic: Probability grid: A location estimation scheme for wireless sensor networks. In: Proc. of the 1^{st} Int. Conf. on Sensor and Ad-Hoc Communication and Networks (SECON). (2004)
15. Yoon, J., Liu, M., Noble, B.: Sound mobility models. In: Proc. of ACM MobiCom. (2003) 205–216
16. Whitehouse, K., Sharp, C., Brewer, E., Culler, D.: Hood: a neighborhood abstraction for sensor networks. In: Proc. of the 2^{nd} Int. Conf. on Mobile systems, applications, and services (MobiSys). (2004)
17. Welsh, M., Mainland, G.: Programming sensor networks using abstract regions. In: Proc. of the 1^{st} USENIX-ACM Symp. on Networked Systems Design and Implementation (NSDI04). (2004)
18. Ni, Y., Kremer, U., Stere, A., Iftode, L.: Programming ad-hoc networks of mobile and resource-constrained devices. In: Proc. of the ACM SIGPLAN Conf. on Programming language design and implementation. (2005) 249–260
19. Frank, C., Römer, K.: Algorithms for generic role assignment in wireless sensor networks. In: Proc. of the 3^{rd} ACM Conf. on Embedded Networked Sensor Systems (SenSys). (2005)
20. Curino, C., Giani, M., Giorgetta, M., Giusti, A., A.L. Murphy, G.P. Picco: TinyLime: Bridging Mobile and Sensor Networks through Middleware. In: Proc. of the 3^{rd} IEEE Int. Conf. on Pervasive Computing and Communications (PerCom). (2005) 61–72

A SPIDEY Grammar

```
<node_template> ::= node template <node_templ_id>
                              ({static | dynamic} <field_name>)+

<node_instance> ::= create node <node_id> from <node_templ_id>
                    (<field_name> as {<target_lang_expr> |
                              <function_name>(<nhood_id>) every <time_period>})+

<nhood_template> ::= neighborhood template <nhood_templ_id>
                              [(<par_name>(,<par_name>)*)]
                     [with <node_predicates>]
                     [[{min | max}] cardinality <integer_value>]
                     [{union | intersect | minus | on}
                              <nhood_templ_id> [<par_bindings>]]*

<nhood_instance> ::= create neighborhood <nhood_id>[<par_bindings>]
                              from <nhood_templ_id>
                     [[{min | max}] hops <integer_value>]
                     [credits <numeric_value>]

<par_bindings> :: = (<par_name>:<target_lang_expr>
                              (,<par_name>:<target_lang_expr>)*)

<cost_function> ::= use cost <target_lang_expr>
```