Computer Networks 112 (2017) 294–313

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

A neighbor detection algorithm based on multiple virtual mobile nodes for mobile ad hoc networks

Behnaz Bostanipour*, Benoît Garbinato

Distributed Object Programming Laboratory, Internef Building, University of Lausanne, CH-1015 Lausanne, Switzerland

ARTICLE INFO

Article history:
Received 4 April 2016
Revised 5 November 2016
Accepted 7 November 2016
Available online 9 November 2016

Keywords:
Neighbor detection
Virtual mobile node
MANET
Proximity-based mobile applications
Distributed algorithms
Smartphone

ABSTRACT

We introduce an algorithm that implements a time-limited neighbor detector service in mobile ad hoc networks. The time-limited neighbor detector enables a mobile device to detect other nearby devices in the past, present and up to some bounded time interval in the future. In particular, it can be used by a new trend of mobile applications known as proximity-based mobile applications. To implement the time-limited neighbor detector, our algorithm uses $n = 2^k$ virtual mobile nodes where k is a non-negative integer. A virtual mobile node is an abstraction that is akin to a mobile node that travels in the network in a predefined trajectory. In practice, it can be implemented by a set of real nodes based on a replicated state machine approach. Our algorithm implements the neighbor detector for real nodes located in a circular region. We also assume that each real node can accurately predict its own locations up to some bounded time interval $\Delta_{predict}$ in the future. The key idea of the algorithm is that the virtual mobile nodes regularly collect location predictions of real nodes from different subregions, meet to share what they have collected with each other and then distribute the collected location predictions to real nodes. Thus, each real node can use the distributed location predictions for neighbor detection. We show that our algorithm is correct in periodically well-populated regions. We also define the minimum value of $\Delta_{predict}$ for which the algorithm is correct. Compared to the previously proposed solution also based on the notion of virtual mobile nodes, our algorithm has two advantages: (1) it tolerates the failure of one to all virtual mobile nodes; (2) as n grows, it remains correct with smaller values of $\Delta_{predict}$. This feature makes the real-world deployment of the neighbor detector easier since with the existing prediction methods, location predictions usually tend to become less accurate as $\Delta_{predict}$ increases. We also show that the cost of our algorithm (in terms of communication) scales linearly with the number of virtual mobile nodes.

1. Introduction

The growing adoption and usage of mobile devices and particularly smartphones has caused the emergence of a new trend of distributed applications known as *Proximity-Based Mobile* (*PBM*) applications [3–6]. These applications enable a user to interact with others in a defined range and for a given time duration e.g., for social networking (WhosHere [52], LoKast [35], iGroups [25], LocoPing [34]), gaming (local multiplayer apps [36]) and driving (Waze [51]).

Discovering who is nearby is one of the basic requirements of PBM applications. It is the preliminary step for further interactions between users. It also enables users to extend their social network from the people that they know to the people that they might not know but who are in their proximity. For instance, with the social networking applications such as WhosHere [52] or LoKast [35], a user first discovers others in her proximity and then decides to view their profiles, start a chat with them or add them as friends. The discoverability, however, may not always be limited to the current neighbors. For instance, with the social networking applications such as iGroups [25] or LocoPing [34], a user can discover others who were in her vicinity during a past event (e.g., concert, tradeshow, wedding) or simply during a past time interval (e.g., the past 24 h). One can also think of applications that provide the user with the list of people who will be in her proximity up to some time interval in the future and thus create the potential for new types of social interactions [5].

In this paper, we present an algorithm that implements the *time-limited neighbor detector* service. This service enables a device to discover the set of its neighbors in the past, present and up to

* Corresponding author.
E-mail addresses: behnaz.bostanipour@unil.ch (B. Bostanipour), benoit.garbinato@unil.ch (B. Garbinato).

http://dx.doi.org/10.1016/j.comnet.2016.11.004
1389-1286/

some bounded time interval in the future in a mobile ad hoc network (MANET). It was first introduced in our previous work [5] to capture the requirements of neighbor detection in PBM applications.

Our algorithm implements the time-limited neighbor detector using $n = 2^k$ virtual mobile nodes where k is a non-negative integer.[1] A virtual mobile node is an abstraction that was already introduced in the literature and used for tasks such as routing or collecting data in MANETs [14,15]. It is akin to a mobile node that travels in the network in a predefined trajectory known in advance to all nodes. In practice a virtual mobile node is emulated by a set of real nodes in the network using a replicated state machine approach.

Our algorithm implements the neighbor detector for real nodes located in a circular region. We also assume that each real node can accurately predict its own locations up to some bounded time interval $\Delta_{predict}$ in the future. Thus, the region is divided into n equal subregions and each subregion is associated with one virtual mobile node. Each virtual mobile node regularly collects the location predictions from the real nodes in its subregion and meets other virtual mobile nodes to share what it has collected with them. After the sharing, each virtual mobile node has the location predictions collected from the entire region, which it distributes to the real nodes in its subregion. In this way, each real node can find its neighbors at current and future times based on the collected location predictions that it receives from a virtual mobile node. It can also store the collected location predictions so it can be queried about its past neighbors.

Main contributions. The main contributions of this paper are as follows. We introduce an algorithm that implements the time-limited neighbor detector service using $n = 2^k$ virtual mobile nodes where k is a non-negative integer. To guarantee the coordination between the virtual mobile nodes, we define a set of explicit properties for their trajectory functions and we show how such trajectory functions can be computed. We prove the correctness of the algorithm under certain conditions. In particular, we show that our algorithm is correct for a category of executions, called *nice executions*, which basically correspond to the executions of the algorithm in periodically well-populated regions such as main squares in a downtown area. We also define the minimum value of $\Delta_{predict}$ for which the algorithm is correct in different cases of nice executions.

This work relies on our previous work [5] for the general idea of using virtual mobile nodes and location predictions to implement the time-limited neighbor detector. However, contrary to the algorithm in [5] which uses only a single virtual mobile node and does not tolerate its failure, our algorithm can use multiple virtual mobile nodes and can tolerate the failure of one to all virtual mobile nodes. Due to the use of multiple virtual mobile nodes, our algorithm has a feature which did not exist in the previous solution: as the number of virtual mobile nodes grows, our algorithm remains correct with smaller values of $\Delta_{predict}$. This feature makes the real-world deployment of the neighbor detector easier. In fact, although there exist different approaches to predict the future locations of a real node, usually predictions tend to become less accurate as $\Delta_{predict}$ increases. We show that the cost of our algorithm (in terms of communication) scales linearly with the number of virtual mobile nodes. We also propose a set of optimizations which can be used for the real-world deployment of our algorithm.

To the best of our knowledge, this is the first work that uses multiple virtual mobile nodes to implement a neighbor detector service in MANETs. Moreover, this is the first work that defines a set of explicit properties for the trajectory functions of the virtual mobile nodes to guarantee the coordination between them.

Road map. The remainder of the paper is as follows. In Section 2, we describe our system model and introduce some definitions. In Section 3, we present a neighbor detector service for MANETs in two variants: the *perfect* variant, which corresponds to the ideal case of neighbor detection and is rather impractical and the *time-limited* variant, for which we propose an implementation in this paper. In Section 4, we present the implementation of the time-limited variant of the neighbor detector service based on virtual mobile nodes. In order to do so, we first describe what a virtual mobile node is and how it can be used for the implementation of the time-limited neighbor detector. We then add n virtual mobile nodes to the system model. Each virtual mobile node has a so called *scan path* through which it travels in its subregion. Thus, we define the properties of this path and we show how it can be computed in order to be useful for our algorithm. We then introduce a round-based algorithm that implements the time-limited neighbor detector in the new system model and prove the correctness of the algorithm under certain conditions. As we show in the proof, the algorithm can tolerate the failure of one to all virtual mobile nodes for a category of executions, called *nice executions*, which basically correspond to the executions of the algorithm in periodically well-populated regions. We also define the minimum value of $\Delta_{predict}$ for which the algorithm is correct in different cases of nice executions. Then, we show the evolution of this value as n grows. Based on this evolution, we deduce that as the number of virtual mobile nodes grows the algorithm requires smaller values of $\Delta_{predict}$ to correctly implement the time-limited neighbor detector. In Section 5, we discuss two topics related to the performance of the algorithm, namely its scalability with respect to the number of virtual mobile nodes and the optimizations which can improve its performance. In particular, we show that the communication cost of the algorithm, defined as the number of message broadcasts per round, has a complexity of $\mathcal{O}(n)$. In Section 6, we discuss the related work and in Section 7, we conclude and discuss future work. The paper has also an Appendix A, which is devoted to finding an upper bound for the scan path length of a virtual mobile node. This upper bound is used (directly or indirectly) in Sections 4.3, 4.6 and 5.1 to find other results.

2. System model and definitions

We consider a mobile ad-hoc network (MANET) consisting of a set P of processes that move in a two dimensional plane. A process abstracts a mobile device in a PBM application.[2] We use the terms *process*, *node* and *real node* interchangeably. Each process has a unique identifier. Processes can move on any continuous path, however there exists a known upper bound on their motion speed. A process is prone to *crash-reboot* failures: it can fail and recover at any time, and when the process recovers, it returns to its initial state. A process is *correct* if it never fails. Since we do not consider *Byzantine* behaviors, the information security and privacy issues are beyond the scope of this paper.

We assume the existence of a discrete global clock, i.e., the range T of the clock's ticks is the set of non-negative integers. We

[1] The present work is an extension of the work published as a short conference paper in [7].

[2] There are two main reasons behind our choice of a MANET as the underlying network architecture. Firstly, MANETs seem to be the most natural existing technology to enable PBM applications. In fact, similar to PBM applications, in a MANET two nodes can communicate if they are within a certain distance of each other (to have radio connectivity) for a certain amount of time. Secondly, mobile devices are increasingly equipped with ad hoc communications capabilities (e.g., WiFi in ad hoc mode or Bluetooth) which increases the chance of MANETs to be one of the future mainstream technologies for PBM applications [6].

also assume the existence of a known bound on the relative processing speed. Each process in the system has access to a *LocalCast service*, a *global positioning service* and a *mobility predictor* service. In the following, we first introduce some definitions that are used throughout the paper. We then present each of the above mentioned services and for each service, we describe how it can be implemented in the real world.

2.1. Definitions

Let p_i be a process in the network, we introduce the following definitions in order to capture proximity-based semantics.

- A *location* denotes a geometric point in the two dimensional plane and can be expressed as tuple (x, y).
- $loc(p_i, t)$ denotes the location occupied by process p_i at time $t \in T$.
- $Z(p_i, r, t)$ denotes all the locations inside or on the circle centered at $loc(p_i, t)$ with given radius r.
- r_d is called *the neighbor detection radius*. It is a constant known by all processes in the network. Thus, $Z(p_i, r_d, t)$ presents the *neighborhood region* of p_i at time t.

2.2. LocalCast service

This communication service was introduced in [14,15]. It allows a process to send messages to all processes located within a given radius around it. Formally, the LocalCast service exposes the following primitives:

- BROADCAST(m, r): broadcasts a message m *in* $Z(p_i, r, t_b)$, where p_i is the sender and t_b is the time when the broadcast is invoked.
- RECEIVE(m, p_i): callback delivering a message m broadcast by process p_i.

The service satisfies the following properties.

Reliable delivery. Assume that a process p_i performs a BROADCAST(m, r) action. Let d be a constant and $\Delta_{delivery} = [t_b;\ t_b + d]$. Then every process p_j delivers m in $\Delta_{delivery}$ if $\forall t \in \Delta_{delivery}, loc(p_j, t) \in Z(p_i, r, t)$ and p_j does not fail during $\Delta_{delivery}$.

Integrity. For any LocalCast message m and process p_i, if RECEIVE(m, p_j) event occurs at p_i, then a BROADCAST(m, r) event precedes it at some process p_j.

As stated in [14], sending a message using this service should be thought of as making a single wireless broadcast (with a small number of retries, if necessary, to avoid collisions). In practice, this service can be implemented with high probability by one of the existing single-hop wireless broadcast protocols as long as the broadcast radius is not too large [14,15].

2.3. Global positioning service

This service allows each mobile process p_i to know its current location and the current time via the following functions:

- GETCURRENTTIME: returns the current global time. Formally, this implies that each process p_i has access to the global clock modeled at the beginning of Section 2.
- GETCURRENTLOCATION: returns the location occupied by p_i at the current global time.

In this paper, we do not provide any formal properties for this service. However, we assume that the outputs of its functions are accurate. In an outdoor setting, this service can typically be implemented using NASA's GPS space-based satellite navigation technology. In an indoor environment, a MIT's Cricket device [39] may be more suitable to implement this service.

2.4. Mobility predictor service

This service allows each mobile process p_i to predict its future locations up to some bounded time $\Delta_{predict}$ via the following function:

- PREDICTLOCATIONS: returns a hash map containing the predicted locations for p_i at each time t in the interval $[t_c;\ t_c + \Delta_{predict}]$ where t_c is the time when PREDICTLOCATIONS is invoked.

The service satisfies the following property.

Strong accuracy. Let $t \in [t_c;\ t_c + \Delta_{predict}]$ and l be a location, if p_i is predicted to be at l at time t, then $loc(p_i, t) = l$.

In this paper we assume that in a PBM application a mobile device (abstracted by a process) is used only by one user. Therefore, to implement the mobility predictor service, a human mobility prediction method should be considered. The human mobility prediction methods are usually based on the fact that the human activities are characterized by a certain degree of regularity and predictability [40], thus, a person's future movement can be predicted using her prior movement history (e.g., previous locations, residence time at each previous location, etc...). In the literature, there exist various human mobility prediction methods [10,13,40,44,45], which are not all suitable for implementing the mobility predictor service. In fact, from a practical point of view, a prediction method should have the following characteristics to implement our mobility predictor service: (1) it should be able to predict not only the future locations of a user but also the time interval during which the user stays at each predicted location; (2) it should not require complex computations and large amount of memory space. The reason behind this requirement is that we consider a MANET where fixed infrastructures are lacking. Thereby, the predictions should be made by each device itself, which has limited resources in terms of battery life and computational capacity. The Markov-based method introduced in [45] and the method based on non-linear time series analysis introduced in [40] are examples of the prediction methods that satisfy the above mentioned requirements and can be used to implement our mobility predictor service.

Note that for simplicity, the mobility predictor service that we consider in this paper is an idealized predictor. In practice, the above mentioned prediction methods can implement it with high probability as long as $\Delta_{predict}$ is not large (i.e., less than or equal to five minutes). More precisely, the mobility predictor service has a *strong accuracy* property according to which the predictions are 100% accurate through the whole interval $\Delta_{predict}$. However, in reality the above mentioned methods can predict the next immediate location with a high accuracy (from 75% to 95% accuracy depending on the method) and this accuracy decreases in an almost linear way as $\Delta_{predict}$ increases. In general, we can say that with these prediction methods the predictions are still highly accurate for a $\Delta_{predict}$ equal to five minutes (for more details see the evaluations in [40]). Another characteristic of our mobility predictor service is that it makes the predictions for geometric locations. However, many of the existing mobility prediction methods make predictions for *symbolic* locations (e.g., rooms in a building, special areas in a map, etc...). In fact, since with symbolic locations, identifying a node's neighbors will depend directly on how symbolic locations are defined and since there exist different types of symbolic locations, in this paper for simplicity, we assume that the predictions are made for geometric locations. We believe that our neighbor detector algorithm can be adapted to the particular cases where symbolic locations are used.

3. The neighbor detector service

This service was first introduced in our previous work in [5]. Intuitively, the neighbor detector service allows a process to know its neighbors at a given time. Formally, it exposes the following primitive:

- PRESENT(t): returns $N(p_i, t)$ i.e., the set of processes detected as neighbors of p_i at time t, where p_i is the process that invokes PRESENT.

3.1. Neighbor detector variants

We present two variants of the neighbor detector service: *the perfect neighbor detector* and *the time-limited neighbor detector*. As we discuss in the following, *the perfect neighbor detector* presents an ideal case of neighbor detection and is rather impractical. The reason why we present this variant is to help the reader to better understand the properties of the other variant i.e., *the time-limited neighbor detector*. *The time-limited neighbor detector* is more practical and is the variant for which we propose an implementation in this paper.

3.1.1. Perfect neighbor detector

By querying this variant of neighbor detector service, a mobile process is able to know the set of its neighbors at any time in the past, present or the future.

Perfect completeness. Let p_i and p_j be two correct processes, if $loc(p_j, t) \in Z(p_i, r_d, t)$, then $p_j \in N(p_i, t)$.

Perfect accuracy. Let p_i and p_j be two correct processes, if $p_j \in N(p_i, t)$, then $loc(p_j, t) \in Z(p_i, r_d, t)$.

Roughly speaking, the *perfect completeness* property requires a neighbor detector to detect any node that is in the neighborhood region at any time in the past, present or future. At the same time, the *perfect accuracy* property guarantees that no false detection occurs. Since in practice implementing the *perfect completeness* property requires an infinite knowledge of nodes' locations in the future, we consider a more practical variant of the neighbor detector service called *the time-limited neighbor detector*. We introduce this variant hereafter.

3.1.2. Time-limited neighbor detector

Compared to the *perfect neighbor detector*, this variant has a different *completeness* property. However, its *accuracy* property is the same. We define its properties, below.

Time-limited completeness. Let p_i and p_j be two correct processes and Δ_{future} be a bounded time interval such that $\Delta_{future} > 0$, if $loc(p_j, t) \in Z(p_i, r_d, t)$ and $t \leq t_c + \Delta_{future}$, then $p_j \in N(p_i, t)$, where t_c is the time when PRESENT is invoked at p_i.

Perfect accuracy. Let p_i and p_j be two correct processes, if $p_j \in N(p_i, t)$, then $loc(p_j, t) \in Z(p_i, r_d, t)$.

Similar to the *perfect completeness* property, the *time-limited completeness* property requires a neighbor detector to detect any node that is in the neighborhood region at any time in the past or present. However, its ability to detect future neighbors is limited by a bounded time duration Δ_{future}. More precisely, it only detects a node that is in the neighborhood region at any time from the time when PRESENT is invoked up to Δ_{future}.[3] The *perfect accuracy* property also guarantees no false detection.

[3] For simplicity, we do not assume a time bound on the availability of the past neighborhood information at this point. We will discuss this further in Section 5.2.2.

As already stated, in this paper we propose an implementation for the time-limited neighbor detector variant. Thus, henceforth whenever we use the term *the neighbor detector service*, we actually refer to the time-limited neighbor detector.

4. Implementing the time-limited neighbor detector

To implement the time-limited neighbor detector, our intuition is as follows: since each node knows its own locations up to $\Delta_{predict}$ in the future, we can think of a moving entity that travels through the network, collects the location predictions of all nodes, and then distributes all the collected location predictions to the nodes. In this way, each node can find its neighbors at current and future times based on the collected location predictions. It can also store the collected location predictions so it can be queried about its past neighbors. In our solution, we consider a *virtual mobile node* (first introduced in [14]) to be used as the moving entity. Moreover, to simplify the problem, we perform the neighbor detection only for real nodes which are in a circular region R of the two dimensional plane. However, using only one virtual mobile node to implement the neighbor detector has a main disadvantage: as the size of the region R grows, the virtual mobile node spends more time to travel through the network. This can cause the collected location predictions to expire before they can be used for neighbor detection. One way to overcome this problem is to increase $\Delta_{predict}$ of the mobility predictor. However, as discussed in Section 2.4, with the existing mobility prediction methods, predictions usually tend to become less accurate as $\Delta_{predict}$ increases.

Another way to deal with this problem is to decrease the traveling time of the virtual mobile node. In order to do so, our solution consists of using more than one virtual mobile node. In fact, our solution can work with $n = 2^k$ virtual mobile nodes where k is a non-negative integer. Thus, the region is divided into n equal subregions and each subregion is associated with one virtual mobile node. Virtual mobile nodes collect simultaneously the location predictions from the real nodes in their subregions and meet at the center of R to share what they have collected with each other. After the sharing, every virtual mobile node has the location predictions collected from the entire R. Then, the virtual mobile nodes simultaneously distribute the collected location predictions to the real nodes in their corresponding subregions. As we further show, as n grows, our solution can correctly implement the neighbor detector with smaller values of $\Delta_{predict}$. Intuitively, this is because as n grows, R is divided into more and consequently smaller subregions and each virtual mobile node spends less time to travel through its subregion.

In the following, we first describe what a virtual mobile node is and we add n virtual mobile nodes to the system model. We also define the properties of the *scan path* i.e., the path through which a virtual mobile node travels in its subregion and we show how it can be computed. We then introduce an algorithm that implements the neighbor detector in the new system model and we prove the correctness of the algorithm. As we show in the proof, the algorithm can tolerate the failure of the virtual mobile nodes under certain conditions. We also define the minimum value of $\Delta_{predict}$ for which the algorithm is correct. We then show the evolution of this value as n grows. Based on this evolution, we deduce that as the number of virtual mobile nodes grows the algorithm requires smaller values of $\Delta_{predict}$ to correctly implement the neighbor detector.

4.1. Virtual mobile node

A virtual mobile node (also referred to as a *virtual node*) is an abstraction that is akin to a mobile node that travels in the network in a predefined trajectory. It was first introduced by Dolev

et al. in [14] to simplify the task of designing algorithms for mobile ad hoc networks. In fact, Dolev et al. consider two main reasons behind the difficulty of designing algorithms for mobile ad hoc networks: (1) the movement of a mobile node is unpredictable; (2) mobile nodes are unreliable i.e., they can continuously join and leave the system, they may fail or recover or be turned on and off by the user or may sometimes choose to sleep and save power. Thus, a virtual mobile node is designed so that it can execute any distributed algorithm that a real node can execute, however, its movement can be predefined and known in advance to all real nodes in the network. Moreover, a virtual mobile node is reliable (also called *robust*). Roughly speaking, this means that a virtual mobile node does not fail as long as it travels through well-populated areas of the network [14].

In [14] an algorithm called *Mobile Point Emulator* (*MPE*) is introduced, which implements the virtual mobile node abstraction in a system model equivalent to the system model defined in this paper. The implementation of the virtual mobile node is based on a replicated state machine technique similar to the one originally presented in [33]. In fact, in order to achieve the robustness of the virtual mobile node in spite of the failure of the real nodes, the algorithm replicates the state of the virtual mobile node at the real nodes which travel near the location of the virtual mobile node. More precisely, the algorithm defines a *mobile point* to be a circular region of a radius r_{mp}, which moves according to the predefined path of the virtual mobile node, i.e., at time t the center of the mobile point coincides with the preplanned location of the virtual mobile node at time t. The MPE replicates the state of the virtual mobile node at every real node within the mobile point's region, modifying the set of replicas as the real nodes move in and out of the mobile point's region. MPE uses a total-order broadcast service to ensure that the replicas are updated consistently. The total order broadcast service is built using the LocalCast communication service (defined in Section 2.2) and synchronized clocks which are obtained by using a service equivalent to our global positioning service. Note that the real nodes are only used by the MPE algorithm to assist in emulating the virtual mobile node. Thereby, the motion of a virtual mobile node may be completely uncorrelated with the motion of the real nodes i.e., even if all the real nodes are moving in one direction, the virtual mobile node may travel in the opposite direction.

Similar to a real node, a virtual mobile node can communicate with other virtual or real nodes using the LocalCast service. Also, a virtual mobile node is prone to *crash-reboot* failures. It can crash if and only if its trajectory takes it into a region unpopulated by any real nodes (i.e., where there are no real nodes to act as replicas), however, it recovers to its initial state as soon as it renters a dense area. A virtual mobile node is *correct* if it never fails, i.e., $\forall t \in T$, at least one correct real node resides in the circular region of radius r_{mp} around the preplanned location of the virtual mobile node at time t.

4.2. Adding virtual mobile nodes to the system model

In this section, we add a set of n virtual mobile nodes $V = \{v_1, \ldots, v_n\}$ to the system model where $n = 2^k$ and k is a non-negative integer. Each virtual node is assigned a unique identifier. Note that we do not provide an implementation for the virtual nodes, however, we assume that they can be implemented by the MPE algorithm sketched in Section 4.1.

Let region R be a closed disk of radius r_{map}, centered at location $l_{map-center}$ which is the origin of the two dimensional plane. Each virtual node v_i is associated with a subregion R_i of R. The subregion R_i is a sector of R (in the shape of a pizza slice) enclosed by two radii and an arc, where the arc subtends an angle $\frac{2\pi}{n}$ (See Fig. 1.a). All subregions have the same area and $\bigcup_{i=1}^{i=n} R_i = R$.[4]

A virtual node can communicate with other virtual nodes or the real nodes using the LocalCast service (defined in Section 2.2) where the broadcast radius equals to a constant non-negative integer r_{com} known globally. This constant is defined by the virtual node implementation (see [14]). Moreover, similar to a real node, a virtual node has access to the global positioning service (defined in Section 2.3).

The movement of a virtual node v_i is defined by a predetermined trajectory function $loc(v_i, t)$, which maps every t in T to a location. This function is known to all virtual nodes and real nodes in the network. The average speed of v_i's movement is equal to a constant v_{avg}. This constant is defined by the speed of the real nodes (that emulate v_i) and the speed of *the join protocol* (a sub-protocol of the MPE algorithm, which enables a real node to join *the emulator set* i.e., the set of real nodes that emulate v_i). Roughly speaking, this means that v_i must move slowly enough so that new real nodes can join the emulator set before the old real nodes leave the emulator set [14].

The trajectory function of v_i is defined such that it can be used by our algorithm for the implementation of the neighbor detector. According to the trajectory function, v_i continuously scans the subregion R_i. The scans are arranged in the form of collect-distribute. More precisely, let $l_{init}(v_i)$ be a location different from $l_{map-center}$. Then, a collect scan starts at $l_{init}(v_i)$ and ends at $l_{map-center}$ and a distribute scan starts at $l_{map-center}$ and ends at $l_{init}(v_i)$ (See Fig. 1.b). The first scan starts at time $t = 0$ and is a collect scan. Collect and distribute scans alternate and v_i uses exactly the same path in the collect and the distribute scans. This path is called the *scan path* of v_i and its length is denoted by $L_{scan-path}(v_i)$. The amount of time that v_i spends in a collect scan is equal to the amount of time that it spends in a distribute scan. This time duration is denoted by $\Delta_{scan}(v_i)$.

In order to be useful for our neighbor detector algorithm, the scan path of v_i should satisfy the three following properties:

Scan completeness. Let s be a scan (collect or distribute) and let t_{begin} be the time when s begins, then the path traversed by v_i during s is such that $\forall location \in R_i, \exists t \in [t_{begin};\ t_{begin} + \Delta_{scan}(v_i) - 1]$ such that $distance(loc(v_i, t), location) \leq r_{com}$.

Equal scan path lengths. Let v_j be a virtual node different from v_i, then $L_{scan-path}(v_i) = L_{scan-path}(v_j)$.

Proportional scan path length. $L_{scan-path}(v_i)$ is an inverse function of n.

The *scan completeness* property guarantees that a scan covers the entire subregion R_i in terms of r_{com}. With regard to the *equal scan path lengths* property, it has a direct result, that is, the value of Δ_{scan} is the same for all virtual nodes (recall that all virtual nodes have the same average speed v_{avg}). Since all virtual nodes start their scanning at $t = 0$ and with a collect scan, this guarantees that all virtual nodes meet at the end of each collect scan at $l_{map-center}$. Finally, *proportional scan path length* guarantees that as n (i.e., the number of virtual nodes) grows, the scan path length and consequently Δ_{scan} of each virtual node decreases.

In next section, we define the scan path that satisfies these properties and is used by the trajectory functions of the virtual nodes.

[4] In general, a disk can be divided using straightedge and compass into n equal parts if $n = 2^k m$ where k is a non-negative integer and m is either equal to 1 or else m is a product of different Fermat primes [27].

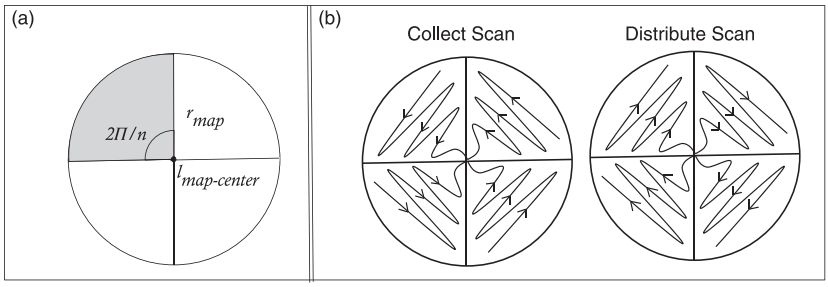

Fig. 1. The subfigures correspond to the case where the number of virtual mobile nodes (denoted by n) is equal to four. (a) Disk R is presented where the grey area corresponds to a subregion R_i. (b) Each virtual mobile node scans its associated subregion in the form of collect and distribute scans. The arrows indicate the direction of motion.

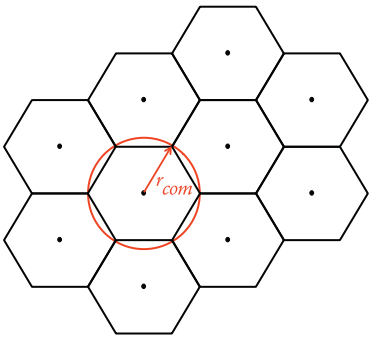

Fig. 2. Hexagonal tessellation of the surface of a subregion R_i. Each hexagon approximates a circle of radius r_{com} and hence its circumradius is equal to r_{com}. In the figure, we present the circle and its radius only for one hexagon.

4.3. The scan path of a virtual mobile node

As described in Section 4.2, the scan path of a virtual node v_i should satisfy a set of properties. We start by finding the optimal path that satisfies the *scan completeness* property. We then show that this path also satisfies the *equal scan path lengths* and the *proportional scan path length* properties.

The optimal path that satisfies the *scan completeness* property is the shortest possible path that goes through a set of locations that we call *covering centers*. Roughly speaking, the covering centers are such that if v_i broadcasts a message at all covering centers then the message is disseminated at all locations in R_i. Thus, covering centers are centers of disks of radius r_{com} that cover the whole surface of R_i such that the number of disks is minimum. Note that a part of the surface of some of these disks can be located out of R_i, thereby, some of the covering centers can be situated at the boundary or even out of R_i.

Finding covering centers is a NP hard problem [29]. It can be approximately solved by applying *hexagonal tessellation* (or so called *hexagonal tiling*) [21]. More precisely, the surface of R_i is tessellated using the regular hexagons of circumradius r_{com} (See Fig. 2). Since for all virtual nodes the scan path goes through $l_{map-center}$, the tessellation made by the tessellation algorithm is such that one of the hexagons is centered at $l_{map-center}$. The algorithm also ensures that the number of hexagons covering R_i is minimum. Once the tessellation is made, the centers of the hexagons are identified as the covering centers. Thus, the scan path can be found as the shortest possible route that visits the center of each hexagon exactly once. This is a variant of a famous algorithmic problem known as the *Travelling Salesman Problem* (*TSP*). In this case, the problem can be easily solved thanks to the properties of the hexagonal tessellation. In fact, in the hexagonal tessellation, the distance between the centers of any two adjacent hexagons is equal to $\sqrt{3}r_{com}$. Therefore, the scan path can be found as the path that connects the centers of each pair of adjacent hexagons exactly once.

The scan path that we have found satisfies the *equal scan path lengths* property since the tessellation of every subregion is made by applying the same algorithm and all subregions have the same shape and area. We would like also to show that the scan path satisfies the *proportional scan path length* property. In fact, the scan path length of v_i can be found as:

$$L_{scan-path}(v_i) = (NOC(R_i) - 1) \times \sqrt{3}r_{com} \tag{1}$$

where $NOC(R_i)$ denotes the number of covering centers for subregion R_i and can be calculated by the tessellation algorithm. Without applying the tessellation algorithm, we can still find an upper bound on $NOC(R_i)$ and consequently on $L_{scan-path}(v_i)$ using lattice theory (see Appendix A on how to find the upper bound). So, we have:

$$L_{scan-path}(v_i) < c_1 + \frac{c_2}{n} \tag{2}$$

where c_1 and c_2 are two constants defined in terms of r_{map} and r_{com} and whose values are defined by Eqs. (A.3) and (A.4) in Appendix A. Thus, the scan path that we have found also satisfies the *proportional scan path length* property. As described in Section 4.2, a direct consequence of this property is that as n grows, Δ_{scan} of each virtual node decreases. In fact, Δ_{scan} of a virtual node v_i can be calculated as below:

$$\Delta_{scan}(v_i) = \frac{L_{scan-path}(v_i)}{v_{avg}} \tag{3}$$

Considering Eqs. (3) and (2) above, we find:

$$\Delta_{scan}(v_i) < \frac{1}{v_{avg}} \times \left(c_1 + \frac{c_2}{n}\right) \tag{4}$$

As we discuss in detail in Section 4.6, Eq. (4) plus the correctness conditions of our neighbor detector algorithm imply that as n (i.e., the number of virtual nodes) grows, our neighbor detector algorithm remains correct with smaller values of $\Delta_{predict}$. Moreover, in Section 5.1, by using Eq. (4) we show that the communication cost of our neighbor detector algorithm scales linearly with the number of virtual mobile nodes.

4.4. Neighbor detector algorithm

The algorithm includes two parts: a part that is executed on each real node p_i (Algorithm 1) and a part that is executed on each virtual node v_i (Algorithm 2). The algorithm relies on the movement of the virtual nodes. Thus, it divides time into rounds of duration Δ_{scan}, where Δ_{scan} is calculated by Eq. (5) and is globally known.

$$\Delta_{scan} = \Delta_{scan}(v_i) \quad \textit{where } v_i \in V \tag{5}$$

Algorithm 1 Neighbor detector algorithm at real node p_i.

initialisation:
  $round \leftarrow$ getRound(getCurrentTime) {assigns to round its value at current time. The first round is a collect round}
  $noMsgSentInThisRound \leftarrow true$
  $networkLocs \leftarrow \perp$ {creates hash map networkLocs to store the location predictions of real nodes in the network}

present(t)
  $N \leftarrow \emptyset$ {creates set N to store the neighbors of p_i at time t}
  if $networkLocs(p_i, t) \neq \perp$ **then** {checks whether a location prediction for p_i at time t exists in networkLocs}
    for all $p_j \in networkLocs$ **do**
      if $p_j \neq p_i \wedge$ distance$(networkLocs(p_j, t), networkLocs(p_i, t)) \leq r_d$ **then**
        $N \leftarrow N \cup p_j$
  return N

upon distance(getCurrentLocation, $loc(v_i$, getCurrentTime$) \leq r_{com}$**such that** $v_i \in V$ **do**
  if $round = collect \wedge noMsgSentInThisRound$ **then**
    $realmsg \leftarrow \perp$ {creates realmsg to encapsulate the hash map locs}
    $realmsg.locs \leftarrow$ predictLocations {hash map locs stores the output of the mobility predictor service}
    trigger broadcast$(realmsg, r_{com})$
    $noMsgSentInThisRound \leftarrow false$

upon roundIsOver(getCurrentTime) **do**
  $noMsgSentInThisRound \leftarrow true$
  if $round = collect$ **then**
    $round \leftarrow distribute$
  else if $round = distribute$ **then**
    $round \leftarrow collect$

upon receive$(virtmsg, v_i)$ **do** {receives virtmsg from virtual mobile node v_i}
  for all $(p_k, t) \in virtmsg.collectedLocs$ **do**
    if $networkLocs(p_k, t) = \perp$ **then** {checks if a location prediction for p_k at time t does not exist in networkLocs}
      $networkLocs(p_k, t) \leftarrow virtmsg.collectedLocs(p_k, t)$ {adds the location prediction for p_k at time t from collectedLocs to networkLocs}

Algorithm 2 Neighbor detector algorithm at virtual mobile node v_i.

initialisation:
  $round \leftarrow$ getRound(getCurrentTime) {assigns to round its value at current time. The first round is a collect round}
  $coveringCenters \leftarrow \{l_1, \ldots, l_{NOC(R_i)}\}$ { Set coveringCenters contains the covering centers of subregion R_i}
  $collectedLocs \leftarrow \perp$ {creates hash map collectedLocs to store the collected location predictions}

upon receive$(realmsg, p_i)$ **do** {receives realmsg from real node p_i}
  for all $t \in realmsg.locs$ **do**
    $collectedLocs(p_i, t) \leftarrow realmsg.locs(t)$ {adds the location prediction for p_i at time t from locs to collectedLocs}

upon roundIsOver(getCurrentTime) **do**
  if $round = collect$ **then**
    $intervirtmsg \leftarrow \perp$ {creates intervirtmsg to encapsulate the hash map collectedLocs}
    $intervirtmsg.collectedLocs \leftarrow collectedLocs$
    trigger broadcast$(intervirtmsg, r_{com})$
    $round \leftarrow distribute$
  if $round = distribute$ **then**
    $collectedLocs$.clear() {clears the content of hash map collectedLocs at the end of each distribute round}
    $round \leftarrow collect$

upon receive$(intervirtmsg, v_j)$ **do** {receives intervirtmsg from virtual mobile node v_j}
  $collectedLocs$.combine$(intervirtmsg.collectedLocs)$ {combines v_i's collectedLocs with collectedLocs of intervirtmsg}

upon getCurrentLocation $= l_i$**such that** $l_i \in coveringCenters$ **do** {v_i is at a covering center of subregion R_i}
  if $round = distribute$ **then**
    $virtmsg \leftarrow \perp$ {creates virtmsg to encapsulate the hash map collectedLocs}
    $virtmsg.collectedLocs \leftarrow collectedLocs$
    trigger broadcast$(virtmsg, r_{com})$

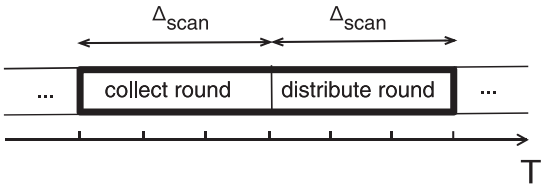

Fig. 3. A phase in the neighbor detector algorithm, composed of a collect and a distribute round. Since the duration of each round is equal to Δ_{scan}, the duration of a phase is equal to $2 \times \Delta_{scan}$.

In Eq. (5) above, the value of $\Delta_{scan}(v_i)$ can be found by Eq. (3) of Section 4.3. Since the value of $\Delta_{scan}(v_i)$ is the same for all virtual nodes, in Eq. (5) there is no difference which virtual node v_i is used for calculation of Δ_{scan}.

There exist two types of rounds: collect and distribute rounds, which alternate. The first round is a collect round. Given this fact and since the execution of the algorithm starts at $t = 0$ (i.e., when the virtual nodes start their movement by a collect scan), the collect and distribute rounds coincide with the collect and distribute scans of virtual nodes, respectively.

The algorithm proceeds in *phases*. Each phase comprises a collect and a distribute round (See Fig. 3). In the collect round, every virtual node scans its subregion and collects the location predictions sent to it by real nodes. Then, the virtual nodes share their collected location predictions with each other when the collect round terminates (i.e., when they meet at $l_{map-center}$). In the distribute round, each virtual node distributes the collected location predictions to real nodes in its subregion. Every real node stores the collected location predictions that it receives to use them for neighbor detection. In the following, we discuss the algorithm in more detail and whenever necessary, we refer to the lines in the algorithm.

The algorithm keeps informed each real and virtual node of round changes via two functions GETROUND and ROUNDISOVER. Function GETROUND is called at initialization (lines 2 and 29). It returns the round type (collect or distribute) at current time. This value is stored in variable *round*. As stated previously, the first round is a collect round, thereby, at $t = 0$, GETROUND returns collect.[5] Function ROUNDISOVER takes current time as parameter and returns a boolean. Thus, when a round terminates ROUNDISOVER returns true, so that the value of variable *round* is changed from collect to distribute and vice-versa.

Since the trajectory function of all virtual nodes are globally known, each real node p_i can calculate its distance to every virtual node at any time. Thus, at each collect round p_i waits until its distance to a virtual node v_i becomes less than or equal to r_{com} (note that v_i can be any virtual node in V) (line 12). Then, if p_i has not already sent a message to any virtual node in that round, it creates a message *realmsg* to send to v_i (line 14). This message encapsulates a hash map *locs* which is used to store the output of PREDICTLOCATIONS primitive of the mobility predictor service (line 15). To store each location prediction of p_i, the hash map *locs* uses one key which is the time instant for which the location is predicted. For instance, *locs*(t) returns the predicted location at time t. Once *locs* is assigned its value, *realmsg* is broadcast within the radius r_{com}, so it can be received by v_i (line 16).

Each virtual node has a hash map *collectedLocs*. It is used to store the location predictions that the virtual node collects. When v_i receives *realmsg* from p_i, it stores every location prediction that exists in *locs* in its *collectedLocs* map (lines 32–34). For this storage, two keys are used where one key is the name of the real node for which the prediction is made and the other key is the time instant for which the prediction is made. For instance, *collectedLocs*(p_i, t) returns the predicted location of p_i at time t.

When a collect round terminates (i.e., when all virtual nodes are at $l_{map-center}$), v_i creates *intervirtmsg* to share its *collectedLocs* with other virtual nodes (lines 35–38). It broadcasts *intervirtmsg* within the radius r_{com}, so it can be received by all virtual nodes (line 39). When a virtual node receives *intervirtmsg*, it combines its own *collectedLocs* with *collectedLocs* of *intervirtmsg*, so that at the next distribute round, all virtual nodes have the same location predictions in their *collectedLocs* maps (lines 44–45).

In a distribute round, v_i encapsulates its *collectedLocs* in a *virtmsg* and broadcasts it whenever it is on a covering center of its subregion R_i (lines 46–50). The set of covering centers denoted by *coveringCenters* is defined at the initialization (line 30) and can be found by the tessellation algorithm discussed in Section 4.3. Thus, v_i broadcasts *virtmsg* on covering centers so that they are disseminated in the whole R_i.

Each real node has a hash map called *networkLocs* that is used to store the location predictions of all real nodes in the network. Similar to *collectedLocs*, *networkLocs* has two keys to store a location prediction: one key is the name of the real node for which the prediction is made and the other key is the time instant for which the prediction is made. For instance, *networkLocs*(p_i, t) returns the predicted location of p_i at time t. The *networkLocs* map is augmented in distribute rounds, i.e., when new location predictions are received in *collectedLocs* of a *virtmsg* (lines 24–27). Thus, whenever primitive PRESENT(t) is invoked at p_i, the map lookups on *networkLocs* as well as distance comparisons are performed to find the real nodes which are in the neighborhood region of p_i at time t (lines 5–9). The names of real nodes found in this way, are stored in set N which is returned as the result (lines 10–11).

4.5. Proof of correctness

In this section we present a proof of correctness for the algorithm. As we show, under certain conditions, the algorithm correctly implements the time-limited neighbor detector abstraction (defined in Section 3) and can tolerate the failure of one to all virtual mobile nodes. Thus, we start by describing the intuition behind the proof and some of the conditions required to guarantee the correctness of the algorithm. We also introduce some preliminary notations, definitions and lemmas that are used throughout the proof. Then, we present the proof. In particular, we define the minimum $\Delta_{predict}$ for which the algorithm is correct. This value is then used in Section 4.6, where we study the impact of increasing the number of virtual mobile nodes on it.

4.5.1. Intuition behind the proof

As we show in the proof, the algorithm can tolerate the failure of one to all virtual nodes under certain conditions. Recall that the failure of a virtual node is of a *crash-reboot* type: it crashes when the area around its trajectory becomes unpopulated and it recovers to its initial state as soon as it reenters a dense area. Since the algorithm proceeds in phases, to guarantee its correctness in spite of the failure of the virtual nodes, our intuition is as follows. We prove the correctness of the algorithm for a category of the executions called *nice executions*, which basically correspond to the executions in periodically well-populated regions such as main squares in a downtown area. In a nice execution, virtual nodes can fail. However, there exist time periods during a nice execution where the whole region R is populated well enough so that all virtual nodes are up. Each of such periods is long enough to contain

[5] Note that the failure of a real or virtual node is of a *crash-reboot* type i.e., if it crashes it recovers to its initial state. Therefore, calling GETROUND at initialization, enables a real or virtual node to know the round type not only at $t = 0$ but also after each recovery.

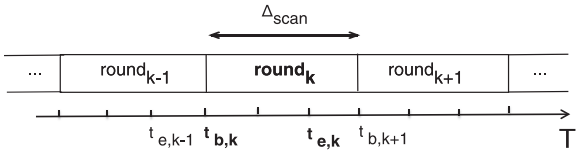

Fig. 4. $t_{b,k}$ and $t_{e,k}$ of $round_k$.

at least one phase that is entirely executed in it.[6] A phase that is executed while all virtual nodes are up is called an *atomic phase*. In an atomic phase, no virtual node crashes, thereby, no location prediction is lost by a virtual node during the collection, sharing and distribution of location predictions. Thus, for neighbor detection, a real node p_i should rely on the location predictions that it receives during the distribute round of each atomic phase. Intuitively, this means that the location predictions that p_i receives during the distribution in an atomic phase should be long enough so that p_i can use them for current and future neighbor detection at least until the distribution in the next atomic phase (as for the past neighbor detection, p_i can use the location predictions that it has received and stored in all previous atomic phases). Note that, in the distribute round of an atomic phase, p_i may receive the location predictions at the latest at the end of the round. Therefore, to guarantee the correctness of the algorithm, $\Delta_{predict}$ should be long enough to ensure the current and future neighbor detection by p_i, at least, at each time instant between the end of the distribute rounds of two consecutive atomic phases. As we further show, such $\Delta_{predict}$ can be found based on the maximum time duration between the end of the distribute rounds of two consecutive atomic phases.

4.5.2. Preliminaries

Before beginning the main part of the proof, we present some preliminary notations, definitions and lemmas that are used throughout the proof. In particular, we focus on formally defining a nice execution and highlight those characteristics of a nice execution that can be used for the proof.

In the following, we first present the preliminary notations and definitions. We then present the preliminary lemmas.

Preliminary notations and definitions. Here we present the preliminary notations and definitions.

- Given two sets A and B, $A \subseteq B$ indicates that A is a subset of B.
- P_R denotes a subset of P (recall that P is the set of all real nodes) such that $\forall p_i \in P_R$, p_i never leaves region R and the movement of p_i during Δ_{scan} is negligible.
- $round_k$ denotes the kth round of the algorithm, where k (also called *the index of the round*) is an integer such that $k \geq 1$.
- ϕ_i denotes the ith phase of the algorithm, where i is an integer such that $i \geq 1$.
- $d(\phi_i)$ returns the index of the distribute round of a phase ϕ_i. For instance, if $round_k$ is the distribute round of ϕ_i, then $d(\phi_i) = k$.
- $t_{b,k}$ and $t_{e,k}$ refer to the first clock tick and the last clock tick in $round_k$, respectively (See Fig. 4). Note that, $t_{e,k} = t_{b,k} + \Delta_{scan} - 1$. We call $t_{b,k}$, *the beginning time* of $round_k$ and $t_{e,k}$ *the end time* of $round_k$.
- *Global System State.* The local state of a (virtual or real) node is a tuple that contains the value of its variables. In particular, among these variables there is a variable which indicates whether the node is up or down. The global system state is a vector σ whose elements are the local states of all virtual and real nodes in the system.
- *Execution.* An execution E of neighbor detector algorithm is an infinite sequence that maps every time instant in T to a global system state. Formally, $E := (\sigma_t)_{t=0}^{+\infty}$ where σ_t is the global state at time $t \in T$.
- *Stable and unstable periods.* Let E be an execution, then E could include two types of time periods called *stable* and *unstable* periods. A stable period is a period during which all virtual nodes are up. On the other hand, an unstable period is a period during which at least one virtual node is down.
- *Nice execution.* Let E be an execution. Let Δ_{stable}^{min} and $\Delta_{unstable}^{max}$ be two non-negative integers such that $\Delta_{stable}^{min} = 4 \times \Delta_{scan}$. We say that E is *nice* if the duration of each stable period in E is at least equal to Δ_{stable}^{min} and the duration of each unstable period in E is at most equal to $\Delta_{unstable}^{max}$. Moreover, the first stable period in E starts at $t = 0$.[7]
- *Atomic phase.* Let E be an execution. Let ϕ_i be a phase in E. Then ϕ_i is *atomic* if it entirely occurs in a stable period of E, that is, while all virtual nodes are up.
- *Nonatomic phase.* Let E be an execution. Let ϕ_i be a phase in E. Then ϕ_i is *nonatomic* if at least a part of it occurs in an unstable period of E.

Preliminary Lemmas. We prove seven preliminary lemmas where two lemmas, i.e., Lemmas 3 and 6, have each an associated corollary. Lemmas 1–5 are straightforward and mainly used to prove (directly or indirectly) Lemmas 6 and 7. Lemmas 6 and 7 are important results which are used (with Lemma 3 and its associated corollary) for the main proof in the next section. In particular, Lemma 6 proves that there exists a maximum, denoted by Δ_{gap}, for the time duration between the end of the distribute rounds of two consecutive atomic phases in a nice execution. It also defines the value of Δ_{gap}. Lemma 7 shows that in a nice execution the time duration between the end of a round $round_k$ such that $k \geq 3$ and the end of the distribute round of the last atomic phase before $round_k$ has a maximum which is equal to Δ_{gap}.

In the following, we prove the lemmas and whenever necessary, we give some additional information regarding their use and the intuition behind them.

Lemma 1. *Let E be an execution. If E is nice, then every stable period in E contains at least one atomic phase.*

Proof. If E is nice, then the duration of every stable period in E is greater than or equal to $4 \times \Delta_{scan}$. Therefore, regardless of how the rounds occur in a stable period, the stable period contains at least one phase which is entirely executed in it. Hence, in this case each stable period contains at least one atomic phase. □

Lemma 2. *Let E be an execution. If all virtual nodes are correct, then E is nice.*

Proof. If all virtual nodes are correct, then they are always up. This means that there exists only one stable period in E which has an infinite duration. Therefore, the duration of the stable period is

[6] The existence of nice executions is realistic considering the variation of population density in a periodically well-populated urban region (e.g., a public square) during a time interval (e.g., a working day). In fact, in a periodically well-populated urban region, there are periods of time where the population density becomes so low so that the virtual nodes that scan the region become unstable (i.e., they crash and recover many times while traveling through their preplanned trajectory). However, after some bounded time duration, the population density increases high enough to guarantee that the virtual nodes remain up for at least some period of time.

[7] As we further discuss in Section 4.5.3, one of the conditions for the correctness of the algorithm is Condition C6 according to which if PRESENT(t) of the neighbor detector is called, then $t \geq t_{e,1}$ and $t_c \geq t_{b,3}$ where t_c is the time when PRESENT(t) is called. As we show in the proof, the lower bounds for t and t_c in Condition C6 are found based on the beginning time of the first stable period in a nice execution. Thus, the assumption that the first stable period in a nice execution starts at $t = 0$ is in fact a convention which simplifies the computation of these lower bounds.

greater than $\Delta^{min}_{stable} = 4 \times \Delta_{scan}$ and the duration of any unstable period is zero and consequently less than or equal to $\Delta^{max}_{unstable}$. □

Lemma 3. *Let E be an execution. If E is nice, then the first phase or ϕ_1 of the algorithm is also the first atomic phase in E.*

Proof. The first phase of the algorithm denoted by ϕ_1 comprises $round_1$ and $round_2$. By definition, the first stable period of a nice execution begins at $t = 0$. Moreover, every stable period of a nice execution lasts at least $4 \times \Delta_{scan}$. Therefore, if E is nice, then the first four rounds of E are in the first stable period. Accordingly, phase ϕ_1 occurs entirely in the first stable period and thus, it is atomic. Therefore, ϕ_1 is also the first atomic phase. □

Corollary 1. *Let E be an execution and $round_k$ be a round in E such that $k \geq 3$. If E is nice, then there exists at least one atomic phase in E which occurs before $round_k$.*

Proof. The proof follows directly from Lemma 3. □

Lemma 4. *Let E be an execution and S_i be a stable period in E. If there exists more than one atomic phase in S_i, then there exists no nonatomic phase in S_i that occurs between the atomic phases.*

Proof. Let ϕ_i and ϕ_j be two atomic phases in S_i such that ϕ_j is the next atomic phase after ϕ_i. Assume for contradiction that there exists a nonatomic phase $\phi_i + 1$ after ϕ_i and before ϕ_j in S_i. Since $\phi_i + 1$ is nonatomic, a part of it should occur in an unstable period. This suggests that an unstable period should exist between ϕ_i and ϕ_j, which implies that ϕ_j should occur in a stable period different than S_i which is impossible. □

Lemma 5. *Let E be a nice execution. Then, every round in E is either in an atomic phase or between two atomic phases.*

Proof. Let $round_k$ be a round in E. From Lemma 3, we get that the first phase in E is atomic, which means that $round_1$ and $round_2$ are in an atomic phase. Thus, the lemma holds for $k < 3$. To show that the lemma also holds for $k \geq 3$ we proceed as follows. By Corollary 1 we know that if $k \geq 3$, there exists at least one atomic phase before $round_k$. Thus, to prove that the lemma holds for $k \geq 3$, we should show that $round_k$ is either in an atomic phase or there exists an atomic phase after $round_k$ so that based on Corollary 1, we can conclude that $round_k$ is between two atomic phases. In the following, we prove the lemma for $k \geq 3$ by considering two possible cases.

Case 1: $round_k$ is in an unstable period of E. Since an unstable period of E lasts at most $\Delta^{max}_{unstable}$, we know that there exists a stable period after the unstable period, which according to Lemma 1 contains at least one atomic phase. Therefore, in this case there exists an atomic phase after $round_k$ and the lemma holds.

Case 2: $round_k$ is in a stable period of E. Let S_i denote the stable period where $round_k$ occurs. By Lemma 1, we know that S_i contains at least one atomic phase. So there exist two subcases: (1) $round_k$ is in an atomic phase of S_i and the lemma holds; (2) $round_k$ is outside of any atomic phases of S_i, which means that $round_k$ is in a nonatomic phase that partly occurs in S_i. By Lemma 4, we know that the nonatomic phase that contains $round_k$ cannot occur between two atomic phases of S_i. Therefore, $round_k$ is either outside and before any atomic phases of S_i or outside and after any atomic phases of S_i. We show that in both cases the lemma holds. In fact, if $round_k$ is outside and before any atomic phases of S_i, then it means that there exists an atomic phase after $round_k$ and the lemma holds. If $round_k$ is outside and after any atomic phases of S_i, then it means that there exists an unstable period just after S_i (otherwise, $round_k$ should be in an atomic phase). Since an unstable period of E lasts at most $\Delta^{max}_{unstable}$, we know that there exists a stable period after the unstable period, which according to Lemma 1 contains at least one atomic phase. Therefore, there exists an atomic phase after $round_k$ and the lemma holds. □

Now that we have proved Lemmas 1–5, we can present more important results based on them. The next lemma proves that there exists a maximum, denoted by Δ_{gap}, for the time duration between the end of the distribute rounds of two consecutive atomic phases in a nice execution. It also defines the value of Δ_{gap}.

Lemma 6. *Let E be an execution. Let ϕ_i and ϕ_j be two atomic phases in E such that ϕ_j is the next atomic phase after ϕ_i. If E is nice, then the time duration between $t_{e,d(\phi_i)}$ and $t_{e,d(\phi_j)}$ has a maximum denoted by Δ_{gap} such that $\Delta_{gap} = 6 \times \Delta_{scan} - 2 + \Delta^{max}_{unstable}$.*

Proof. We assume that E is nice and we consider the two possible cases below.

Case 1: ϕ_i and ϕ_j are in the same stable period. Since ϕ_j is the next atomic phase after ϕ_i and since ϕ_i and ϕ_j are in the same stable period, by Lemma 4, we know that there exists no nonatomic phase between ϕ_i and ϕ_j. Therefore, ϕ_j can only be the phase just after ϕ_i (see Fig. 5.a). Thus, in this case the time duration between $t_{e,d(\phi_i)}$ and $t_{e,d(\phi_j)}$ is always equal to $2 \times \Delta_{scan}$ and the maximum time duration between $t_{e,d(\phi_i)}$ and $t_{e,d(\phi_j)}$ is also equal to $2 \times \Delta_{scan}$.

Case 2: ϕ_i and ϕ_j are in two different stable periods. Let S_i and S_j be two different stable periods such that ϕ_i is in S_i and ϕ_j is in S_j. Since ϕ_j is the next atomic phase after ϕ_i, we know that there is no atomic phase between ϕ_i and ϕ_j. Moreover, since E is nice by Lemma 1 we know that every stable period in E contains at least one atomic phase. Therefor, there exists no stable period between S_i and S_j. Accordingly, there exists only one unstable period between S_i and S_j. Thus, in this case the maximum time duration between $t_{e,d(\phi_i)}$ and $t_{e,d(\phi_j)}$ corresponds to the following situation: the unstable period between S_i and S_j lasts $\Delta^{max}_{unstable}$. In addition, there exist a nonatomic phase ϕ_{i+1} just after ϕ_i and a nonatomic phase ϕ_{j-1} just before ϕ_j such that one time unit of ϕ_{i+1} and one time unit of ϕ_{j-1} occur in the unstable period between S_i and S_j and $2 \times \Delta_{scan} - 1$ time units of ϕ_{i+1} occur in S_i and $2 \times \Delta_{scan} - 1$ time units of ϕ_{j-1} occur in S_j (see Fig. 5.b).[8] Thus, the maximum time duration between $t_{e,d(\phi_i)}$ and $t_{e,d(\phi_j)}$ in this case is equal to $2 \times (2 \times \Delta_{scan} - 1) + \Delta^{max}_{unstable} + 2 \times \Delta_{scan} = 6 \times \Delta_{scan} - 2 + \Delta^{max}_{unstable}$.

In each case described above, the time duration between $t_{e,d(\phi_i)}$ and $t_{e,d(\phi_j)}$ has a maximum. In Case 1, the maximum is equal to $2 \times \Delta_{scan}$. In Case 2, it is equal to $6 \times \Delta_{scan} - 2 + \Delta^{max}_{unstable}$. Hence, Δ_{gap} is equal to $6 \times \Delta_{scan} - 2 + \Delta^{max}_{unstable}$. □

The value of Δ_{gap} defined by Lemma 6 corresponds to nice executions in general. The following corollary of Lemma 6 defines the value of Δ_{gap} in a special case of nice executions, i.e., where all virtual nodes are correct. This value of Δ_{gap} is smaller than the value defined in Lemma 6. We will use this value in Section 4.5.3 to define the minimum $\Delta_{predict}$ required for the correctness of the algorithm in the special case where all virtual nodes are correct.

Corollary 2. *If all virtual nodes are correct, then $\Delta_{gap} = 2 \times \Delta_{scan}$.*

Proof. Let E be the execution considered in Lemma 6. Let ϕ_i and ϕ_j be the two atomic phases considered in Lemma 6. From Lemma 2, we know that if all virtual nodes are correct, then E is nice. Moreover, if all virtual nodes are correct, E contains only one stable period and no unstable period. Therefore, there exists only one case, i.e., ϕ_i and ϕ_j are in the same stable period. By proof of Lemma 6, we know that the maximum time duration between

[8] Recall that the duration of a phase is equal to $2 \times \Delta_{scan}$ time units.

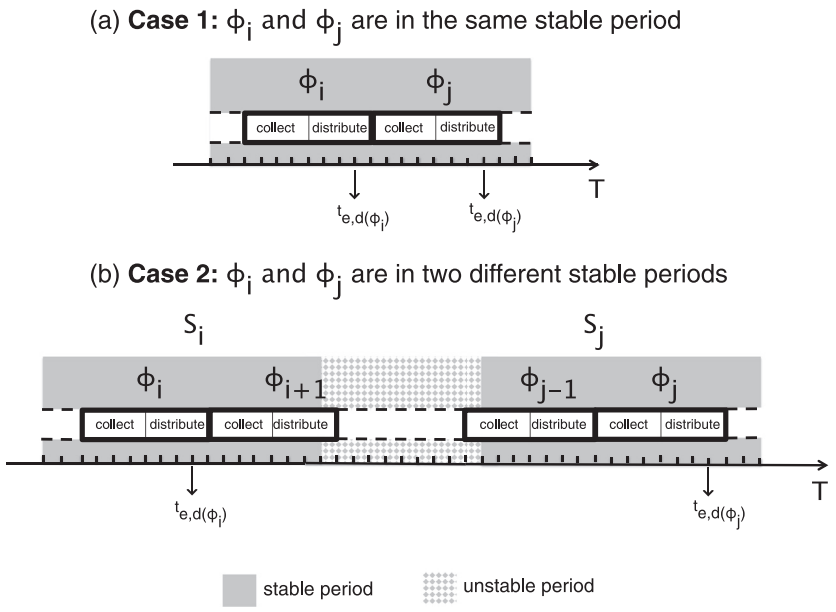

Fig. 5. Examples of occurrence of two consecutive atomic phases ϕ_i and ϕ_j in a nice execution. The cases correspond to the cases of the proof of Lemma 6.

$t_{e,d(\phi_i)}$ and $t_{e,d(\phi_j)}$ in this case is equal to $2 \times \Delta_{scan}$. Hence, Δ_{gap} is equal to $2 \times \Delta_{scan}$. □

Our final preliminary lemma shows that in a nice execution the time duration between the end of a round $round_k$ such that $k \geq 3$ and the end of the distribute round of the last atomic phase before $round_k$ has a maximum. This maximum is equal to the time duration Δ_{gap} which is already defined by Lemma 6. Note that the following lemma is defined for $k \geq 3$ since by Corollary 1 we know that there exists at least one atomic phase before $round_k$ if $k \geq 3$.

Lemma 7. *Let E be a nice execution. Let $round_k$ be a round in E such that $k \geq 3$. Let ϕ_i be the last atomic phase before $round_k$. Then, the maximum time duration between $t_{e,d(\phi_i)}$ and $t_{e,k}$ is equal to Δ_{gap}.*

Proof. By Lemma 5, $round_k$ occurs either in an atomic phase or between two atomic phases. Thus, let ϕ_j be the atomic phase just after ϕ_i, we know that $round_k$ occurs either between ϕ_i and ϕ_j or in ϕ_j. Therefore, we can say that $round_k$ can be, at the latest, the distribute round of ϕ_j. From Lemma 6, we get that the maximum time duration between $t_{e,d(\phi_i)}$ and $t_{e,d(\phi_j)}$ is equal to Δ_{gap}. Hence, the lemma holds. □

4.5.3. The proof

We prove the correctness of the algorithm under a set of conditions. In particular, we define the minimum $\Delta_{predict}$ for which the algorithm is correct in different cases of nice executions i.e., in the general case as well as in the special case where all virtual mobile nodes are correct. Thus, in the following, we first introduce the conditions under which the algorithm is correct and formally describe the meaning and the implication of each condition. We then introduce the theorems and the lemmas that are used for the proof.

Note that in this section, $\Delta_{predict}$ refers to the prediction interval of the mobility predictor service (defined in Section 2.4), Δ_{future} refers to the time duration defined in the *time-limited completeness* property of the neighbor detector (stated in Section 3.1.2) and Δ_{gap} refers to the time duration defined in Lemma 6.

Conditions. We prove that the algorithm is correct under the conditions listed hereafter.

C1. We consider a nice execution for the proof.

C2. The value of the constant d defined in the *reliable delivery* property of the LocalCast service (stated in Section 2.2) is negligible.

C3. The execution time of lines 36–40 and line 45 of the algorithm is negligible.

C4. $\Delta_{predict} = 2 \times \Delta_{scan} - 1 + \Delta_{gap} + \Delta_{future}$.

C5. Let p_i and p_j be the processes defined in the *time-limited completeness* property of the neighbor detector (stated in Section 3.1.2), then $p_i, p_j \in P_R$.

C6. If PRESENT(t) of the neighbor detector is called, then $t \geq t_{e,1}$ and $t_c \geq t_{b,3}$ where t_c is the time when PRESENT(t) is called.

Informally speaking, Condition C1 enables us to use the characteristics of a nice execution (stated in the preliminary lemmas of Section 4.5.2) to prove the correctness of the algorithm. Condition C2 plus the *reliable delivery* property of the LocalCast service (stated in Section 2.2) ensure that a node which remains for a negligible time within the broadcast radius of the sender, will deliver the broadcast message with negligible delay. Condition C3 guarantees that the creation of a *intervirtmsg* message (i.e., the message used by a virtual node to share its collected location predictions with other virtual nodes) and the combination of the location predictions collected by different virtual nodes takes a negligible time. This condition plus some other properties imply that the sharing of collected location predictions between virtual nodes takes a negligible time. Condition C4 defines a value of $\Delta_{predict}$ for which the algorithm is correct. This value is long enough to ensure the current and future neighbor detection by a real node at each time instant between the end of the distribute rounds of two consecutive atomic phases. As we show in the proof, the value defined in Condition C4 is in fact the minimum value of $\Delta_{predict}$ for which the algorithm is correct. Condition C5, assumes that processes p_i and p_j defined in the *time-limited completeness* property of the neighbor detector are in set P_R. This means that p_i and p_j are always in region R and their movements are negligible during Δ_{scan}. As we show in the poof, this condition plus some other properties ensure that in the collect as well as in the distribute round of an atomic phase, there exists a time when p_i and p_j are within distance r_{com} to a virtual node and thus, can communicate with it. Finally, Condition C6 implies that the neighbor detection is not guaranteed for time instants before $t_{e,1}$ (i.e., the end time of $round_1$) and PRESENT(t) is called in $round_k$ where $k \geq 3$.

Theorems and Lemmas. We prove four theorems. The main theorem, which proves the correctness of the algorithm, is Theorem 3. The proof of Theorem 3 relies on Theorems 1 and 2. Theorem 1 proves that the algorithm satisfies the *time-limited completeness* property of the time-limited neighbor detector abstraction (stated in Section 3.1.2). To prove Theorem 1, we use three helper Lemmas, that is, Lemmas 8, 9 and 10. Each helper lemma also relies for its proof on some of the preliminary lemmas introduced in the previous section. Theorem 2 proves that the algorithm satisfies the *perfect accuracy* property of the time-limited neighbor detector abstraction (stated in Section 3.1.2). The proof of this theorem is straightforward and relies on no lemma. Our final theorem, Theorem 4, proves that the minimum $\Delta_{predict}$ for which the algorithm is correct is equal to the one defined in Condition C4. The proof of this theorem relies on Theorems 3, 1 and Corollary 4, which is the associated corollary of Lemma 10.

In the following, we first prove the helper lemmas for Theorem 1. We then prove Theorems 1–4, respectively. Note that throughout the proof, whenever necessary we refer to the lines of the algorithm.

Before presenting the helper lemmas for Theorem 1, we describe the key idea behind the proof of Theorem 1. As previously stated according to Theorem 1, the algorithm satisfies the *time-limited completeness* property. Roughly speaking, this property requires a process p_i to detect any process p_j that is in the neighborhood region of p_i at any time in the past, present and up to interval Δ_{future} in the future. In the algorithm a process uses the location predictions that it stores in its *networkLocs* for neighbor detection. Thereby, in order to prove Theorem 1, by using the helper lemmas we basically prove that: (1) at the distribute round of each atomic phase, p_i receives accurate location predictions for both p_i and p_j and stores the predictions in its *networkLocs*; (2) the location predictions stored in *networkLocs* are long enough so that at any time instant in a $round_k$ such that $k \geq 3$, p_i has enough predictions to detect past, present and future neighbors. The reason why $k \geq 3$, is that in a nice execution, the first distribute round which occurs in an atomic phase is $round_2$. The helper lemmas 9 and 10 each have a corollary. These corollaries are later used by Theorem 4 to prove that the value of $\Delta_{predict}$ defined in Condition C4, is also the minimum $\Delta_{predict}$ for which the algorithm is correct.

Lemma 8. *Let p_i and p_j be the processes defined in the time-limited completeness property of the neighbor detector. Let $round_k$ be the distribute round of an atomic phase. Then, in $round_k$, process p_i receives a virtmsg from a virtual mobile node with collectedLocs map which contains accurate location predictions for both p_i and p_j. Moreover, all location predictions are defined for the time interval $[t_{e,k-1};\ t_{b,k-1} + \Delta_{predict}]$.*

Proof. If $round_k$ is a distribute round in an atomic phase, then $round_{k-1}$ is a collect round in the same atomic phase. According to Condition C5, $p_i, p_j \in P_R$. Therefore, p_i and p_j are always in region R and their movements are negligible during Δ_{scan}. Moreover, the scan path of each virtual node guarantees the *scan completeness* property (stated in Section 4.2) which implies that in each round, for each location l in a subregion R_i scanned by virtual node v_i, there exists a time when l is within distance r_{com} to v_i. Considering these facts and since in $round_{k-1}$ all virtual nodes are up, then in $round_{k-1}$ there exists a time when the distance of p_i and p_j to a virtual node becomes less than or equal to r_{com}. According to the algorithm, in a collect round, as soon as a real node realizes that is within a distance r_{com} to a virtual node, it sends its location predictions in *locs* map to the virtual node (lines 12–16). Therefore, in $round_{k-1}$ both p_i and p_j send their *locs* maps to a virtual node. The *strong accuracy* property of the mobility predictor service (stated in Section 2.4) guarantees that the location predictions of p_i and p_j in their *locs* maps are accurate. Moreover, in $round_{k-1}$, if a real node is within a distance r_{com} to a virtual node at the earliest possible time (i.e., at $t_{b,k-1}$ or the beginning time of the round), its *locs* map is defined for time interval $T_1 = [t_{b,k-1};\ t_{b,k-1} + \Delta_{predict}]$. On the other hand, if in $round_{k-1}$ a real node is within a distance r_{com} to a virtual node at the latest possible time (i.e., $t_{e,k-1}$ or the end time of the round), its *locs* map is defined for time interval $T_2 = [t_{e,k-1};\ t_{e,k-1} + \Delta_{predict}]$. The intersection of T_1 and T_2 is $T_3 = [t_{e,k-1};\ t_{b,k-1} + \Delta_{predict}]$. Thus, regardless of the time when p_i and p_j are within a distance r_{com} to a virtual node in $round_{k-1}$, their *locs* maps contain location predictions for time interval T_3. Condition C2 plus the *reliable delivery* property of the underlying broadcast (stated in Section 2.2) ensure that a node which remains for a negligible time within the broadcast radius of the sender, will deliver the broadcast message with negligible delay. Thus, considering Condition C2, the *reliable delivery* property of the underlying broadcast and the fact that in $round_{k-1}$ all virtual nodes are up, we know that the communication between a virtual node and a correct real node in $round_{k-1}$ is reliable and takes negligible delay. Moreover, according to the algorithm, a virtual node stores all the location predictions received from the real nodes in its *collectedLocs* map (lines 32–34). Therefore, the location predictions sent by p_i and p_j in $round_{k-1}$ are received and stored by the virtual nodes which are in their proximity in $round_{k-1}$.

According to the algorithm, when $round_{k-1}$ terminates, virtual nodes share their own *collectedLocs* with each other by broadcasting *intervirtmsg* messages (lines 36–40). Then, they combine the received *collectedLocs* with their own *collectedLocs* (line 45). Considering Condition C2, the *reliable delivery* property of the underlying broadcast and the fact that all virtual nodes are up and meet when $round_{k-1}$ (which is a collect round) terminates, we know that the communication between virtual nodes is reliable and takes negligible time. Condition C3 also guarantees that the creation of a *intervirtmsg* message and the combination of *collectedLocs* maps takes a negligible time. As a result, at the beginning of $round_k$, which is a *distribute* round, all the virtual nodes have the location predictions of p_i and p_j in their *collectedLocs*. According to the algorithm, in $round_k$ each virtual node v_i encapsulates its *collectedLocs* map in a *virtmsg* and broadcasts it at the covering centers of its subregion R_i (lines 46–50). As previously stated, according to Condition C5, $p_i \in P_R$, which means that p_i is always in R and its movement during Δ_{scan} is negligible. We also know that in $round_k$, all virtual nodes are up. Therefore, regardless of the subregion where p_i is found, there exists a time in $round_k$ when p_i is within the broadcast radius of a virtual node which broadcasts *virtmsg*. Condition C2 and the *reliable delivery* property of the underlying broadcast, also guarantee that *virtmsg* will be received by p_i in a negligible time. Therefore, we know that regardless of the subregion where p_i is found in $round_k$, it receives a *virtmsg* encapsulating the *collectedLocs* and broadcast by a virtual node, in $round_k$. As we have just shown, the *collectedLocs* map contains accurate location predictions for both p_i and p_j and all the location predictions are defined for time interval T_3. Hence, the Lemma holds. □

Lemma 9. *Let p_i and p_j be the processes defined in the time-limited completeness property of the neighbor detector. Let phase ϕ_i be an atomic phase. Then, once ϕ_i terminates, networkLocs of p_i contains accurate location predictions for both p_i and p_j, which are all defined for the time interval $[t_{e,1};\ t_{e,d(\phi_i)} + \Delta_{gap} + \Delta_{future}]$.*

Proof. For the proof we use induction.

Base Case: the first atomic phase. From Condition C1, we get that the execution that we consider for the proof is nice. Thus, from Lemma 3, we get that in a nice execution, the first phase or ϕ_1 is also the first atomic phase. According to the algorithm, ϕ_1 comprises $round_1$ and $round_2$ where $round_1$ is a collect round and $round_2$ is a distribute round. Since $round_2$ is a distribute round of an atomic phase, by Lemma 8 we know that

in $round_2$, p_i receives the accurate location predictions for both p_i and p_j and the predictions are all defined for time interval $T_1 = [t_{e,1};\ t_{b,1} + \Delta_{predict}]$. By replacing $\Delta_{predict}$ by its value defined in Condition C4, we have $T_1 = [t_{e,1};\ t_{b,1} + 2 \times \Delta_{scan} - 1 + \Delta_{gap} + \Delta_{future}] = [t_{e,1};\ t_{e,2} + \Delta_{gap} + \Delta_{future}]$. Since $t_{e,2} = t_{e,d(\phi_1)}$, we have $T_1 = [t_{e,1};\ t_{e,d(\phi_1)} + \Delta_{gap} + \Delta_{future}]$. According to the algorithm, p_i stores all the received location predictions in *networkLocs* and uses them for neighbor detection (lines 24–27). Therefore, once the first atomic phase terminates, process p_i has the accurate location predictions for both p_i and p_j which are all defined for the time interval T_1. Hence, the lemma holds in this case.

Inductive Step. We assume that the lemma holds for an atomic phase ϕ_i which is either the first atomic phase or after the first atomic phase. Then, we wish to show that the lemma holds for ϕ_j which is the next atomic phase after ϕ_i. By inductive hypothesis we know that once ϕ_i terminates, *networkLocs* of p_i contains accurate location predictions for both p_i and p_j, which are all valid for the time interval $T_1 = [t_{e,1};\ t_{e,d(\phi_i)} + \Delta_{gap} + \Delta_{future}]$. According to Condition C1, the execution that we consider is nice, therefore, by Lemma 6, we know that the maximum time duration between $t_{e,d(\phi_i)}$ and $t_{e,d(\phi_j)}$ is equal to Δ_{gap}. Thus, let $T_2 = [t_{e,1};\ t_{e,d(\phi_j)} + \Delta_{future}]$, we know that once ϕ_i terminates, *networkLocs* of p_i contains accurate location predictions for both p_i and p_j, which are valid for the time interval T_2 since $T_2 \subseteq T_1$. Moreover, by Lemma 8, we know that in the distribute round of ϕ_j, the process p_i receives the accurate location predictions for both p_i and p_j and the predictions are defined for the time interval $T_3 = [t_{e,d(\phi_j)-1};\ t_{b,d(\phi_j)-1} + \Delta_{predict}]$. By replacing $\Delta_{predict}$ by its value defined in Condition C4, we have $T_3 = [t_{e,d(\phi_j)-1};\ t_{b,d(\phi_j)-1} + 2 \times \Delta_{scan} - 1 + \Delta_{gap} + \Delta_{future}] = [t_{e,d(\phi_j)-1};\ t_{e,d(\phi_j)} + \Delta_{gap} + \Delta_{future}]$. According to the algorithm, p_i stores all the received location predictions in *networkLocs* and uses them for neighbor detection (lines 24–27). Thus, when ϕ_j terminates, *networkLocs* of p_i contains accurate location predictions for both p_i and p_j, which are defined for time interval $T_4 = T_2 \cup T_3 = [t_{e,1};\ t_{e,d(\phi_j)} + \Delta_{gap} + \Delta_{future}]$. Hence, the lemma holds in this case.

Since the base case holds and since the inductive step holds, the lemma holds. □

Corollary 3. *Let $\Delta^{min}_{predict}$ be the minimum value of $\Delta_{predict}$ for which Lemma 9 holds, then $\Delta^{min}_{predict} = 2 \times \Delta_{scan} - 1 + \Delta_{gap} + \Delta_{future}$ (i.e., the value defined in Condition C4).*

Proof. For the proof we show that if $\Delta^{min}_{predict} = 2 \times \Delta_{scan} - 1 + \Delta_{gap} + \Delta_{future}$, then Lemma 9 holds for $\Delta_{predict} \geq \Delta^{min}_{predict}$ and it does not hold for $\Delta_{predict} < \Delta^{min}_{predict}$. Thus, we assume that $\Delta_{predict} = \Delta^{min}_{predict} + \Delta_{diff}$ where $\Delta_{diff} = \Delta_{predict} - \Delta^{min}_{predict}$ and we consider the two following cases:

Case 1 $\Delta_{predict} \geq \Delta^{min}_{predict}$. Consider the proof of Lemma 9. Then, in Base Case of the induction to calculate T_1, replace $\Delta_{predict}$ by $\Delta^{min}_{predict} + \Delta_{diff}$. If $\Delta^{min}_{predict} = 2 \times \Delta_{scan} - 1 + \Delta_{gap} + \Delta_{future}$, then $T_1 = [t_{e,1};\ t_{e,2} + \Delta_{gap} + \Delta_{future} + \Delta_{diff}]$. As $t_{e,2} = t_{e,d(\phi_1)}$, we have $T_1 = [t_{e,1};\ t_{e,d(\phi_1)} + \Delta_{gap} + \Delta_{future} + \Delta_{diff}]$. Since in this case $\Delta_{diff} \geq 0$, then $[t_{e,1};\ t_{e,d(\phi_1)} + \Delta_{gap} + \Delta_{future}] \subseteq T_1$ and the lemma holds for Base Case of the induction. Also, in Inductive Step of the induction, to calculate T_3, replace $\Delta_{predict}$ by $\Delta^{min}_{predict} + \Delta_{diff}$. If $\Delta^{min}_{predict} = 2 \times \Delta_{scan} - 1 + \Delta_{gap} + \Delta_{future}$, then $T_3 = [t_{e,d(\phi_j)-1};\ t_{e,d(\phi_j)} + \Delta_{gap} + \Delta_{future} + \Delta_{diff}]$ and $T_4 = T_2 \cup T_3 = [t_{e,1};\ t_{e,d(\phi_j)} + \Delta_{gap} + \Delta_{future} + \Delta_{diff}]$. Since in this case $\Delta_{diff} \geq 0$, then $[t_{e,1};\ t_{e,d(\phi_j)} + \Delta_{gap} + \Delta_{future}] \subseteq T_4$ and the lemma holds for Inductive Step of the induction. So, Lemma 9 holds in this case.

Case 2 $\Delta_{predict} < \Delta^{min}_{predict}$. Consider the proof of Lemma 9. Then, in Base Case of the induction to calculate T_1, replace $\Delta_{predict}$ by $\Delta^{min}_{predict} + \Delta_{diff}$. If $\Delta^{min}_{predict} = 2 \times \Delta_{scan} - 1 + \Delta_{gap} + \Delta_{future}$, then $T_1 = [t_{e,1};\ t_{e,2} + \Delta_{gap} + \Delta_{future} + \Delta_{diff}]$. As $t_{e,2} = t_{e,d(\phi_1)}$, we have $T_1 = [t_{e,1};\ t_{e,d(\phi_1)} + \Delta_{gap} + \Delta_{future} + \Delta_{diff}]$. Since in this case $\Delta_{diff} < 0$, then T_1 is a subset of $[t_{e,1};\ t_{e,d(\phi_1)} + \Delta_{gap} + \Delta_{future}]$ such that T_1 is not equal to $[t_{e,1};\ t_{e,d(\phi_1)} + \Delta_{gap} + \Delta_{future}]$. Therefor, the lemma does not hold for Base Case of the induction. So, Lemma 9 does not hold in this case. □

Lemma 10. *Let p_i and p_j be the processes defined in the time-limited completeness property of the neighbor detector. Then, at every $round_k$ such that $k \geq 3$, networkLocs of p_i contains accurate location predictions for both p_i and p_j, which are defined for the time interval $[t_{e,1};\ t_{e,k} + \Delta_{future}]$.*

Proof. By Condition C1, we know that the execution that we consider for the proof is nice. By Corollary 1, which is a corollary of Lemma 3, we know that there exists at least one atomic phase which occurs before $round_k$. Let ϕ_i be the last atomic phase before $round_k$. By Lemma 9, we know that once ϕ_i terminates, *networkLocs* contains accurate location predictions for both p_i and p_j for the time interval $[t_{e,1};\ t_{e,d(\phi_i)} + \Delta_{gap} + \Delta_{future}]$. By Lemma 7, we know that the maximum time duration between $t_{e,d(\phi_i)}$ and $t_{e,k}$ is equal to Δ_{gap}. Thus, in $round_k$, *networkLocs* of p_i contains accurate location predictions for both p_i and p_j for the time interval $[t_{e,1};\ t_{e,k} + \Delta_{future}]$. □

Corollary 4. *The minimum value of $\Delta_{predict}$ for which Lemma 10 holds is $\Delta^{min}_{predict} = 2 \times \Delta_{scan} - 1 + \Delta_{gap} + \Delta_{future}$.*

Proof. Consider the proof of Lemma 10. This proof relies on Lemma 9 which states that once ϕ_i terminates, *networkLocs* of p_i contains accurate location predictions for both p_i and p_j, which are defined for the time interval $[t_{e,1};\ t_{e,d(\phi_i)} + \Delta_{gap} + \Delta_{future}]$. This time interval is the minimum time interval required to prove Lemma 10, mainly because Δ_{gap} is the maximum (and hence, the least upper bound) for the time duration between $t_{e,d(\phi_i)}$ and $t_{e,k}$. Moreover, By Corollary 3, which is a corollary of Lemma 9, we know that $\Delta^{min}_{predict} = 2 \times \Delta_{scan} - 1 + \Delta_{gap} + \Delta_{future}$, is the minimum value for which Lemma 9 holds. Therefore, based on the above arguments, $\Delta^{min}_{predict} = 2 \times \Delta_{scan} - 1 + \Delta_{gap} + \Delta_{future}$ is the minimum value for which Lemma 10 holds. □

Theorem 1. *The neighbor detector algorithm satisfies the time-limited completeness property.*

Proof. From Condition C6, we have $t_c \geq t_{b,3}$, which means that PRESENT(t) is called in $round_k$ such that $k \geq 3$. Also, $t \geq t_{e,1}$ implies that the neighbor detection is not guaranteed for time instants before $t_{e,1}$. Thus, considering Condition C6, we can reformulate the theorem as follows. Let p_i and p_j be the two correct processes defined in the *time-limited completeness* property. Let PRESENT(t) be invoked at p_i in $round_k$ such that $k \geq 3$. If $loc(p_j, t) \in Z(p_i, r_d, t)$ and $t_{e,1} \leq t \leq t_c + \Delta_{future}$, then $p_j \in N(p_i, t)$ where t_c is the time when PRESENT(t) is called at p_i. For the proof, we proceed as follows. By Lemma 10, we know that at every $round_k$ such that $k \geq 3$, *networkLocs* of p_i contains accurate location predictions for both p_i and p_j for the time interval $T_1 = [t_{e,1};\ t_{e,k} + \Delta_{future}]$. We know that in $round_k$, $t_c \in [t_{b,k};\ t_{e,k}]$. Thus, let $T_2 = [t_{e,1};\ t_c + \Delta_{future}]$ we have $T_2 \subseteq T_1$. Therefore, in $round_k$, *networkLocs* of p_i contains accurate location predictions for both p_i and p_j for T_2. Since the algorithm guarantees the correct neighbor matching based on calculating the distance between the predicted locations (line 9), if $loc(p_j, t) \in Z(p_i, r_d, t)$, then $p_j \in N(p_i, t)$ for $\forall t \in T_2$. So, the theorem holds. □

We next prove Theorem 2 according to which the algorithm correctly implements *the perfect accuracy* property. Roughly speaking, this property guarantees that no false neighbor detection occurs. Since for neighbor detection a process uses the location pre-

dictions stored in its *networkLocs*, to prove the theorem we basically show that: (1) the location predictions in *networkLocs* are indeed collected from the real nodes and are not created by the communication links; (2) the locations are predicted accurately and (3) the algorithm correctly detects the neighbors of a process by calculating the distance between the predicted locations.

Theorem 2. *The neighbor detector algorithm satisfies the perfect accuracy property.*

Proof. Let p_i and p_j be the processes defined in the *perfect accuracy* property, according to the algorithm, if $p_j \in N(p_i, t)$, there exists a round during which p_i has received from a virtual node a *virtmsg* with a *collectedLocs* map such that a location prediction for key pair (p_j, t) exists in *virtmsg.collectedLocs*. The *integrity* property of the underlying broadcast (stated in Section 2.2) guarantees that the *virtmsg* is indeed sent by a virtual node. According to the algorithm, the *collectedLocs* map of the *virtmsg* is created by combining the *collectedLocs* of all virtual nodes in the system (line 45). These *collectedLocs* are encapsulated in *intervirtmsg*s and received from virtual nodes when a collect round terminates. The *integrity* property of the underlying broadcast guarantees that each *intervirtmsg* is indeed sent by a virtual node. The *collectedLocs* of *intervirtmsg*s contain location predictions that are collected by virtual nodes during the collect round. In the collect round, the location predictions are sent by real nodes in *locs* maps of *realmsg*s (lines 32–34). The *integrity* property of the underlying broadcast guarantees that each *realmsg* received from a real node is indeed sent by that real node. The *strong accuracy* property of the mobility predictor service (stated in Section 2.4) guarantees that the location predictions in *locs* maps are accurate. Moreover, the algorithm only detects p_j as a neighbor of p_i at time t if the distance between their predicted locations at t is less than or equal to r_d (line 9). Hence, if $p_j \in N(p_i, t)$, then $loc(p_j, t) \in Z(p_i, r_d, t)$ and the theorem holds. □

Theorem 3. *The neighbor detector algorithm correctly implements the time-limited neighbor detector service.*

Proof. By Theorem 1, the algorithm guarantees the *time-limited completeness* property. By Theorem 2, the algorithm guarantees the *perfect accuracy* property. Hence, Theorem 3 holds. □

Theorem 4. *The minimum value of $\Delta_{predict}$ for which neighbor detector algorithm correctly implements the time-limited neighbor detector service is $\Delta^{min}_{predict} = 2 \times \Delta_{scan} - 1 + \Delta_{gap} + \Delta_{future}$.*

Proof. According to Theorem 3, the algorithm correctly implements the time-limited neighbor detector service. For the proof of Theorem 3 we used Theorem 1 which itself relies upon Lemma 10. By Corollary 4, which is a corollary of Lemma 10, we know that $\Delta^{min}_{predict} = 2 \times \Delta_{scan} - 1 + \Delta_{gap} + \Delta_{future}$ is the minimum value for which Lemma 10 holds. Hence, Theorem 4 holds. □

Corollary 5. *In the general case of nice executions $\Delta^{min}_{predict} = 8 \times \Delta_{scan} + \Delta^{max}_{unstable} + \Delta_{future} - 3$ and in a special case of nice executions where all virtual nodes are correct $\Delta^{min}_{predict} = 4 \times \Delta_{scan} + \Delta_{future} - 1$.*

Proof. In Theorem 4, $\Delta^{min}_{predict}$ is defined as a function of Δ_{gap}. From Lemma 6, we get that for nice executions in general $\Delta_{gap} = 6 \times \Delta_{scan} - 2 + \Delta^{max}_{unstable}$. However, by Corollary 2, which is a corollary of Lemma 6, we know that in a special case of nice executions where all virtual nodes are correct, $\Delta_{gap} = 2 \times \Delta_{scan}$. Thus, the corollary holds if in $\Delta^{min}_{predict}$ we replace Δ_{gap} with its value for each case. □

Since Δ_{scan} and Δ_{future} are positive integers and $\Delta^{max}_{unstable}$ is a non-negative integer, from Corollary 5, we conclude that $\Delta^{min}_{predict}$ in the case that all virtual nodes are correct is smaller than in the general case of nice executions. This result is very intuitive. In fact, in the general case of nice executions the location predictions should be long enough to do not expire during the unstable periods where virtual nodes crash. In the case where all virtual nodes are correct, there is no unstable period, therefore, the predictions do not need to be as long as in the general case.

4.6. Impact of increasing the number of virtual mobile nodes on $\Delta^{min}_{predict}$

As discussed at the beginning of Section 4, one of our motivations for designing a neighbor detector algorithm based on multiple virtual mobile nodes, was that by growing the number of virtual mobile nodes (denoted by n), we can reduce $\Delta_{predict}$ required for the correctness of the algorithm. Thus, here we discuss the impact of increasing n on $\Delta^{min}_{predict}$ (defined by Theorem 4). In order to do so, we first find an upper bound on $\Delta^{min}_{predict}$ where the upper bound is a function of n. To find the upper bound we proceed as follows. In Corollary 5, which is the associated corollary of Theorem 4, $\Delta^{min}_{predict}$ is expressed as a function of Δ_{scan} (both in the general case of nice executions as well as in the case where all virtual nodes are correct). By Eq. (5) of Section 4.4, we know that Δ_{scan} is equal to $\Delta_{scan}(v_i)$ where v_i can be any virtual node in the set of all virtual nodes in the system. Moreover, in Eq. (4) of Section 4.3 we have already defined an upper bound for $\Delta_{scan}(v_i)$ where the upper bound is a function of n. Considering these facts, we find the following upper bound for $\Delta^{min}_{predict}$:

$$\Delta^{min}_{predict} < c'_1 + \frac{c'_2}{n} \tag{6}$$

where c'_1 and c'_2 are two constants defined as follows. Let c_1 and c_2 be the constants in Eq. (4) and whose values are defined by Eqs. (A.3) and (A.4) in Appendix A. Then, for the general case of nice executions, we have:

$$c'_1 = \frac{8}{v_{avg}} \times c_1 - 3 + \Delta^{max}_{unstable} + \Delta_{future} \tag{7}$$

$$c'_2 = \frac{8}{v_{avg}} \times c_2 \tag{8}$$

And in the special case of nice executions where all virtual nodes are correct, we have:

$$c'_1 = \frac{4}{v_{avg}} \times c_1 - 1 + \Delta_{future} \tag{9}$$

$$c'_2 = \frac{4}{v_{avg}} \times c_2 \tag{10}$$

According to the Eq. (6), as n grows $\Delta^{min}_{predict}$ decreases. Roughly speaking, this means that as the number of virtual nodes grows the algorithm requires smaller values of $\Delta_{predict}$ to correctly implement the neighbor detector. However, note that as n approaches infinity, the right hand side of the equation approaches c'_1 which is a constant. In practice, this means that if the number of virtual nodes is very large, adding more virtual nodes does not reduce any more the minimum value of $\Delta_{predict}$ required for correctness of the algorithm.

5. Performance discussion

In this section we discuss two topics related to the performance of our algorithm, namely its scalability with respect to the number of virtual nodes and the optimizations which can improve its performance.

5.1. Scalability with respect to the number of virtual mobile nodes

As discussed in Section 4.6, one of the advantages of our neighbor detector algorithm is that as n (i.e., the number of virtual nodes) grows the algorithm requires smaller values of $\Delta_{predict}$ to correctly implement the neighbor detector. However, adding more virtual nodes also affects the consumption of resources such as energy and bandwidth. Communication is the main cause of energy and bandwidth consumption in a network executing the neighbor detector algorithm. Therefore, in this section we study the impact of increasing n on *the communication cost*. Note that since the algorithm is round-based and there exist an infinite number of rounds, we define the communication cost for one round, which can be measured by the number of broadcasts (via the underlying LocalCast service) in a round.

Let $NOB(R_i)$ denote the number of broadcasts in a subregion R_i during a round. Basically, $NOB(R_i)$ is an increasing function of $\Delta_{scan}(v_i)$ (recall that v_i is the virtual node associated to R_i). In fact, if $\Delta_{scan}(v_i)$ increases, v_i scans its subregion longer. Consequently, more real nodes emulate v_i, which results in more broadcasts. In addition, if the round is a collect round, more real nodes send their predictions to v_i and if the round is a distribute round, v_i broadcasts the location predictions longer. Let *maximum broadcast rate* (mbr) denote the maximum number of broadcasts per time unit in any subregion, we have:

$$NOB(R_i) \leq mbr \times \Delta_{scan}(v_i) \tag{11}$$

Note that mbr is a constant and independent of n. In fact, mbr is a function of the number of real nodes that are within a distance r_{mp} and/or r_{com} of the location of v_i per time unit. These are the real nodes that emulate v_i or send their location predictions to v_i. As in our study we assume that the real node density, r_{mp} and r_{com} are constant, then mbr is also a constant.

Let $NOB(R)$ denote the number of broadcasts in the entire region R during a round (recall that R is the region for which the neighbor detection is implemented). Based on Eq. (11) and since the value of Δ_{scan} is the same for all virtual nodes, we can find the following upper bound on $NOB(R)$:

$$NOB(R) \leq n \times mbr \times \Delta_{scan}(v_i) \tag{12}$$

Considering Eq. (12) and Eq. (4) of Section 4.3, which defines an upper bound for $\Delta_{scan}(v_i)$, we have:

$$NOB(R) < \frac{n \times mbr \times c_1}{v_{avg}} + \frac{mbr \times c_2}{v_{avg}} \tag{13}$$

where c_1 and c_2 are the constants of Eq. (4) and whose values are defined by Eqs. (A.3) and (A.4) in Appendix A. Based on Eq. (13), we know that the number of broadcasts in a round has a complexity of $\mathcal{O}(n)$. This means that the communication cost of the algorithm scales linearly with the number of virtual mobile nodes. Although there exists no widely-accepted definition of scalability in the literature, it is usually assumed that an algorithm is scalable if its cost is less than $\mathcal{O}(n^2)$ [47]. Therefore, we can say that the algorithm is scalable (in terms of communication cost) with respect to the number of virtual nodes.

5.2. Performance optimization

It is beyond the scope of this paper to present a deployment of the neighbor detector algorithm in a real network. Here, we only discuss some ways in which the neighbor detector algorithm can be optimized for deployment purposes. It would be interesting to experiment with a real deployment to determine the extent to which these optimizations can be applied and whether they can effectively improve the performance of the algorithm.

Since the neighbor detector algorithm relies on virtual mobile nodes, optimizing the implementation of the virtual mobile nodes indirectly optimizes the neighbor detector algorithm. Thus, in the following we first discuss ways in which the implementation of the virtual mobile nodes can be optimized. We then discuss ways in which the neighbor detector algorithm can be directly optimized.

5.2.1. Optimizing the implementation of the virtual mobile nodes

In this paper we assumed that the virtual mobile nodes are implemented by the MPE algorithm sketched in Section 4.1. The MPE algorithm has a number of limitations. In particular, it requires significant amounts of communication and power consumption [14]. Moreover, it relies on the LocalCast service, a powerful local communication service which is both reliable and synchronous (i.e., it delivers the message after a bounded time interval). Below, we suggest two ways to deal with these limitations.

Optimizing the MPE algorithm. In [14], Dolev et al. propose some optimizations for the MPE algorithm. These optimizations mainly reduce the number of message exchanges between the real nodes, which results in saving power as well. For instance, if a (temporary) leader is elected within a mobile point, and the leader initiates all the transitions for the replica, conflicting requests are avoided and power is saved. The interested reader can refer to [14] for more detail regarding these optimizations. Moreover, according to Dolev et al. the MPE algorithm can be correctly implemented using an underlying broadcast service that works in partially synchronous environments. They also propose some broadcast algorithms that can be used for implementing the MPE algorithm in such environments. The interested reader can refer to [15] for more detail.

Using a simpler algorithm than the MPE algorithm to implement the virtual mobile nodes. In [14], in addition to the MPE algorithm, Dolev et al. also describe an agent-based algorithm to implement the virtual mobile node abstraction. The algorithm uses a mobile agent, which is a special program (or as called in [14], a *dynamic process*) that jumps from one real node to another, moving in the direction specified by the trajectory function of the virtual mobile node. An agent *hitches a ride* with a host that is near to the specified location of the virtual mobile node. Compared to the MPE algorithm, this algorithm is simpler and more efficient (i.e., it requires less message exchanges and power consumption). However, it only achieves one of the goals of the design of a virtual mobile node i.e., the movement of the virtual mobile node can be predefined. In fact, the host of the agent is a single point of failure and therefore a virtual mobile node implemented by the agent-based algorithm is not robust. It seems likely that our neighbor detector algorithm can be correctly implemented (with some probability and under some conditions) even if it uses the virtual nodes implemented by the agent-based algorithm. In this case the basic idea is that if the average time during which a host remains up after each recovery is known, then the agent can change its host just before the time when the host is likely to crash.

5.2.2. Optimizing the neighbor detector algorithm

In addition to the implementation of the virtual nodes, the neighbor detector algorithm can also be directly optimized in many ways. Below we discuss some of these methods.

Avoid unnecessary sharing and distribution of location predictions. In the current version of the neighbor detector algorithm, when a collect round terminates, each virtual mobile node shares the location predictions that it has collected with other virtual mobile nodes. However, sharing the location predictions may not be always necessary. In fact, users of mobile devices may remain for long periods

of time in a subregion depending on their work habits, their movement speed, etc. In this case, the predicted locations of a real node are all in the same subregion where the real node resides at the moment of collection. Thus, if at the end of a collect round, for every virtual node v_i, the predicted locations that it has collected are all in R_i (as a reminder, R_i denotes the associated subregion of v_i), then there is no need for virtual nodes to share what they have collected. Consequently, in the next distribute round, each virtual node only distributes the predictions that it has collected from its associated subregion. In the described example, the unnecessary distribution of location predictions is avoided indirectly, i.e., by avoiding the unnecessary sharing. However, it seems that the unnecessary distribution can also be avoided after the sharing. For instance, after the sharing, based on the location predictions, each virtual node v_i can identify the real nodes that will be in its associated subregion during the upcoming distribute round. Thus, if v_i can determine the location predictions which are never used for neighbor detection by these real nodes, it can avoid distributing them.

Defining a time bound for the detection of the past neighbors. In the definition of the *time-limited completeness* property of the neighbor detector service (stated in Section 3.1.2), for simplicity, we do not assume a time bound for the detection of the past neighbors. This implies that each real node should have an unbounded buffer to store all the location predictions that it receives from the virtual nodes in the distribute rounds. However, for real deployments, based on the application requirements and the availability of the resources, a time interval down to which the past neighbors can be detected should be considered. In this way, the location predictions stored by a real node can be erased as soon as they become too old to be used for the neighbor detection.

Defining clusters for real nodes. In the current version of the neighbor detector algorithm, each real node acts on its own and is responsible for sending and receiving messages to the virtual nodes. However, we can think of real nodes forming clusters such that within each cluster one or more real nodes, called the *gateway nodes*, are in charge of communicating with the virtual nodes. Based on how the clusters are defined, the other real nodes can communicate with these gateway nodes using the LocalCast service or a traditional MANET routing protocol. It seems that using clusters can reduce the scan path length of a virtual node (and consequently the scan duration or Δ_{scan}). In fact, in this case, during a scan a virtual node should only be in the transmission range of the gateway nodes instead of all real nodes. It remains an open question as to whether using clusters can also reduce the number of message exchange.

6. Related work

For the related work, we consider three categories of algorithms, which are *neighbor detection algorithms*, *mobility-assisted algorithms* and *virtual mobile node-based algorithms*. Note that these categories are not disjoint. In particular, all virtual mobile node-based algorithms can also be categorized as mobility-assisted algorithms and some of them can also be categorized as neighbor detection algorithms. Our motivation for considering a category for virtual mobile node-based algorithms is to be able to discuss in detail how these algorithms use virtual mobile nodes to achieve their goals and to compare their approach with the approach used by our algorithm.

6.1. Neighbor detection algorithms

In ad hoc networks neighbor detection is usually studied as a building block for applications such as routing, leader election, group management and localization. The majority of the existing neighbor detection algorithms belong to the *hello protocols* family [1,4,6,17,24,26,28,37,50]. They are based on the *basic hello protocol* first described in *Open Shortest Path First* (*OSPF*) routing protocol [38], which works as follows: each node in the network periodically sends *hello* messages to announce its presence to close nodes, and maintains a neighbor set. The sending frequency is denoted by f_{hello}. If a *hello* message is not received from a neighbor for a predefined amount of time, then that neighbor is discarded from the neighbor set. Thus, the optimal f_{hello} for this approach should be high enough to cause the neighbor set to remain up to date but not too high to cause the waste of communication bandwidth and energy [26]. However, finding the optimal f_{hello} is not obvious and the existing solutions cannot ideally solve this problem. Moreover, contrary to our neighbor detector algorithm, the *hello protocols* usually provide only the set of current neighbors and they do not satisfy any formal guarantees.

Although, the *hello protocols* comprise the majority of the existing neighbor detection algorithms for ad hoc networks, in the literature there exist also the schemes that use different approaches than the *hello* broadcast for neighbor detection [11,12]. For instance, in [12] Cornejo et al. define a reliable neighbor detection abstraction that establishes links over which message delivery is guaranteed. They present two region-based neighbor detection algorithms which implement the abstraction with different link establishment guarantees. The main idea behind the algorithms is that a node sends a *join* message some time after entering a new region to establish communication links. It also sends a *leave* message some time before leaving a region to inform the other nodes so that they can tear down their corresponding link with that node. Since a node should send a *leave* message some time before it actually leaves a region, the algorithm assumes that a node's trajectory function is known to that node with enough anticipation to communicate with other nodes before leaving the region. The approach applied in [12] for neighbor detection is interesting because it uses a relatively lower number of message broadcast compared to the *hello protocols*. Similar to our work, this approach also uses the knowledge of nodes about their future locations for the neighbor detection. However, contrary to our work, no future neighbor detection is defined and only the current neighbor detection is guaranteed.

The time-limited neighbor detector service implemented in this paper is first introduced in our previous work in [5]. To the best of our knowledge, it is the only neighbor detector service for ad hoc networks that detects not only the current neighbors of a node but also its future neighbors. In addition to the definition of the neighbor detector, in [5] we also proposed a simple but limited algorithm that implements the neighbor detector using a single virtual mobile node. In Section 6.3, which is dedicated to the virtual mobile node-based algorithms, we will describe this simple algorithm in more detail and compare it with the algorithm introduced in the present work.

6.2. Mobility-assisted algorithms

In the literature of mobile ad hoc networks, node mobility is leveraged for different purposes e.g., to improve security [8], increase network capacity [19] or locating nodes [20]. In particular, there exist various algorithms that take advantage of mobility for routing purposes in sparse mobile ad hoc networks [2,9,18,23,31,32,41–43,46,48,53–56].[9] In a sparse mobile ad hoc network, node deployment is sparse. Therefore, nodes may

[9] Since a sparse mobile ad hoc network is a specific type of delay tolerant networks (DTNs), in the literature, some of these algorithms are presented as routing algorithms for DTNs.

not be in the transmission range of each other for long periods of time. Several routing algorithms for this type of networks use the *store-carry-forward* model, according to which the nodes in the network forward a message from the source node to the destination node in one or many hopes, such that, once a relay node receives the message, it stores and carries the message until it has a chance to forward it to another node [53]. The famous examples of the algorithms that use the *store-carry-forward* model are the *epidemic routing* algorithms [18,23,32,42,43,46,48] which exploit the inherent node mobility [56]. In [22], Hatzis et al. introduce the concept of *compulsory* protocols, which require a subset of the mobile nodes to move in a *pre-specified* manner. The motivation behind the design of *compulsory* protocols is that if mobile nodes moved in a programmable way, algorithms could take advantage of motion, performing even more efficiently than in static networks [14]. In [22], Hatzis et al. present an efficient compulsory protocol for leader election. Furthermore, Chatzigiannakis et al. [9] and Li et al. [31] propose simple and efficient routing algorithms based on the idea of compulsory protocols.

Among the mobility-assisted algorithms that do not use virtual mobile nodes, the closest algorithms to our work are the *message ferrying* (MF) algorithms [2,53–55]. MF algorithms are the routing algorithms for sparse ad hoc networks, which use moving entities called *message ferries* (or *ferries* for short) for carrying messages between nodes. Ferries travel through the network and communicate with nodes using a single-hop broadcast scheme. Similar to a virtual mobile node's path, the route through which a ferry moves is programmed and usually known to all nodes in the network. However, contrary to a virtual mobile node, a ferry is, in fact, a real node which has no (or less) resource constraints (in terms of energy consumption or buffer size) compared to other nodes in the network. Ferries are used in different applications. For instance, in a disaster scene where the existing infrastructure is unusable, airplanes or vehicles can be used as ferries to transport data between users in separated areas [55].[10] The MF algorithms usually aim at satisfying some guarantees in terms of network throughput, average message delay or energy consumption. Therefore, the routes of the ferries should be designed so that such guarantees can be achieved. In [53,55] the ferry route design problem is considered for a network where regular nodes[11] are stationary and their locations are *a priori* known. Thus, the ferry route design problem is basically considered as a variation of the *traveling salesman problem* (TSP) and is solved by applying some TSP algorithms on the locations of the regular nodes. Obviously, the solutions proposed in [53,55] are limited since they can only be applied to stationary networks where the locations of regular nodes are *a priori* known. In [54], a network composed of mobile regular nodes is considered. Since regular nodes are mobile, to ensure the meeting between the ferries and the regular nodes, two approaches are proposed where each approach is presented by one MF algorithm. In the Node-Initiated MF (NIMF) algorithm, ferries move around the deployed area according to routes known to the regular nodes. When a regular node has a message to send or receive, it moves close to a ferry and communicates with it. The problem with this approach is that it disrupts the normal movement of the regular nodes for the purpose of communication with the ferries. In the Ferry-Initiated MF (FIMF) algorithm, the ferries move to meet the regular nodes at their requests. Thus, when a regular node wants to send a message to another regular node or receive a message, it generates a *service request* (which is a control message encapsulating the location of the regular node) and transmits it to a chosen ferry using a long-range radio. Upon the reception of a *service request*, the ferry will adjust its trajectory to meet up with the regular node and exchange messages using short range radios. The main problem with this approach is that it requires the use of long-range radio which might not always be feasible or desirable. In [2], a ferry route design algorithm called Optimized Way Points (OPWP) is proposed for a network where regular nodes are mobile. OPWP only guarantees probabilistic meetings between the ferries and the regular nodes, that is, it ensures that every time a ferry traverses its route, it meets every regular node with a certain minimum probability. A ferry route found by OPWP comprises an ordered set of way-points and waiting times at these way-points that are chosen carefully based on the mobility model of the regular nodes. More precisely, to choose the set of way-points for a ferry route, OPWP requires that for every regular node p_i and every way-point s in the deployment area, the probability of the meeting between the ferry and p_i when the ferry moves as well as the probability of the meeting between the ferry and p_i when the ferry waits at s to be known. Thus, the main problem with the approach used by OPWP is that to find the routes of the ferries, not only the mobility model of the regular nodes should be *a priori* known but also the mobility model should be such that the above mentioned probabilities can be determined from it.

As we have described, the MF algorithms require some nodes in the network (basically the ferries) to move in a controlled, programmed way (note that in some MF algorithms such as in the NIMF algorithm, not only the ferries but also the regular nodes should adjust their movement for the communication purposes) whereas our work is based on virtual mobile nodes which only use the real nodes to emulate the virtual mobile node and does not require them to move in a programmed way. Not requiring the real nodes to move in a programmed way is preferable especially in the PBM applications (i.e., the target applications for our work) where the wireless devices are used by ordinary people who are not amenable to following instructions as to where their devices may travel. Moreover, as described above, the approaches proposed in the literature to compute the ferry routes, are either limited (e.g., require stationary regular nodes with known locations) or complicated (e.g., require sending control messages using long range radio or require some probabilities to be determined from the mobility model of the regular nodes). On the contrary, our approach for finding the scan path of virtual mobile nodes does not require a stationary network and is simple, since it only requires finding the covering centers using a hexagonal tessellation algorithm.

In the next section we discuss a special type of mobility-assisted algorithms, i.e., *the virtual mobile node-based* algorithms and compare them to our algorithm.

6.3. Virtual mobile node-based algorithms

The idea of using virtual mobile nodes to facilitate the design of algorithms for mobile ad hoc networks was first introduced by Dolev et al. in [14]. The virtual mobile node abstraction design was inspired by idea of the compulsory protocols of Hatzis et al. [22] (we have already discussed the compulsory protocols in Section 6.2). Note however that, contrary to the compulsory protocols, the virtual mobile node-based algorithms do not require a set of real nodes to move in a programmable way but they rather require the virtual mobile nodes, emulated by the real nodes, to move in a programmable way.

In [15], several basic algorithms that use virtual mobile nodes to solve various problems are briefly presented. These algorithms address the problems such as routing, collecting and evaluating data, group communication and atomic memory in mobile ad hoc

[10] In the literature, there exists also the notion of *Data MULEs* (Mobile Ubiquitous LAN Extensions) [41]. Message ferries and Data MULEs are somehow synonymous. The main difference between them is that the movement of Data MULEs is random [2,43,54].

[11] In the MF algorithms, the nodes in the network which are not ferries are referred to as *regular nodes*. We also adopt this term while discussing MF algorithms.

networks. Similar to the present work, some of these algorithms use several virtual nodes which regularly exchange information between each other. However, contrary to the present work, no explicit properties for the trajectory functions of the virtual nodes are defined to guarantee the meeting and communication between them.

In our previous work [5], we presented an algorithm that implements the time-limited neighbor detector service with the same guarantees defined in the present work. Similar to the present work, real nodes have access to a mobility predictor service and can predict their locations up to $\Delta_{predict}$ in the future. However, the algorithm uses only a single virtual mobile node that travels through the network, collects location predictions and distributes neighbor detection-related information. Thus, in order to stay correct, the algorithm requires greater $\Delta_{predict}$ values as the map size grows. Another drawback of the algorithm presented in [5] is that it does not tolerate the failure of the virtual mobile node. These drawbacks are the main motivations for the present work. Finally, note that the present work is an extension of the work published as a short conference paper in [7].

7. Conclusion

We have introduced an algorithm that implements the time-limited neighbor detector service for MANETs using $n = 2^k$ virtual mobile nodes where k is a non-negative integer. We proved that our algorithm is correct under certain conditions. In particular, we showed that our algorithm is correct for a category of executions, called *nice executions*, which basically correspond to the executions of the algorithm in periodically well-populated regions. We also defined the minimum value of $\Delta_{predict}$ for which the algorithm is correct in different cases of nice executions.

Compared to the previously proposed algorithm [5], which uses a single virtual mobile node, our algorithm has two advantages: (1) it tolerates the failure of one to all virtual mobile nodes; (2) as the number of virtual mobile nodes grows, it remains correct with smaller values of $\Delta_{predict}$. This feature makes the real-world deployment of the neighbor detector easier since with the existing prediction methods, location predictions usually tend to become less accurate as $\Delta_{predict}$ increases. We showed that the cost of our algorithm (in terms of communication) scales linearly with the number of virtual mobile nodes. We also proposed a set of optimizations which can be used for the real-world deployment of our algorithm.

To the best of our knowledge, this is the first work that uses multiple virtual mobile nodes to implement a neighbor detector service in MANETs. Another novelty of our work, is the definition of explicit properties for the scan paths of virtual nodes and then presenting a way to compute the scan paths. As shown in the paper, the scan paths are defined so that they guarantee a full collection and distribution of predictions in each subregion as well as the coordination between the virtual mobile nodes. Thus, we believe that the approach used in this paper to define the scan paths can be used for implementing other virtual mobile node-based algorithms such as virtual mobile node-based routing algorithms.

As a potential future work, we consider a real-world deployment of our neighbor detector algorithm. In fact, we are currently developing, *ManetLab*, a modular and configurable software framework for creating and running testbeds to evaluate MANET-specific protocols [49]. Once the neighbor detector algorithm is deployed on ManetLab, we can perform the following:

- compare our theoretical results with the results obtained from the real deployment. In fact, as shown by a quantitative analysis in the paper, when n (i.e., the number of virtual nodes) grows, the neighbor detector algorithm requires smaller values of $\Delta_{predict}$ to correctly implement the neighbor detector and at the same time its communication cost grows as $\mathcal{O}(n)$. Thus, it would be interesting to validate our theoretical results using the real deployment and determine up to which value of n the utility of the algorithm outweighs its cost;
- deploy some other famous neighbor detection algorithms (mentioned in Section 6.1) and compare their performance with the algorithm in the present work. Note that since other neighbor detection algorithms (except the algorithm in our previous work [5]) only detect current neighbors the comparison will only be limited to current neighbor detection;
- apply the optimizations proposed in Section 5.2 on the deployment and determine the extent to which these optimizations can be applied and whether they can effectively improve the performance of the algorithm.

The neighbor detector algorithm presented in this paper is designed for periodically well-populated regions. Thus, another issue which can be investigated as future work is to implement the neighbor detector for less populated regions. In order to do so, we can think of using another type of virtual nodes called *autonomous virtual mobile nodes* [16]. This type of virtual nodes can move autonomously, choosing to change their path based on their own state and inputs from the environment. For instance, if the area in their paths appears deserted, they can change their path to the more populated areas.

Acknowledgment

This research is partially funded by the Swiss National Science Foundation in the context of Project 200021-140762.

Appendix A. Finding an upper bound for the scan path length of a virtual mobile node

According to Eq. (1) of Section 4.3, the scan path length of a virtual node v_i or $L_{scan-path}(v_i)$ can be calculated as:

$$L_{scan-path}(v_i) = (NOC(R_i) - 1) \times \sqrt{3} r_{com}$$

where $NOC(R_i)$ denotes the number of covering centers for subregion R_i. We can find an upper bound on $NOC(R_i)$ by counting the number of lattice points that are associated to the tessellation of R_i. In mathematics and group theory, a two dimensional *lattice* is a discrete subgroup of $\mathbb{R}^2$ which spans the vector space of $\mathbb{R}^2$. In a regular hexagonal tessellation, the centers and the vertices of the hexagons form a two dimensional lattice called the *hexagonal lattice*. Let NOL_{disk} denote the number of hexagonal lattice points within a disk centered at the origin of the plane (which is $l_{map-center}$ in our case). Then, NOL_{disk} can be calculated using Eq. (A.1) where r is the radius of the disk and d is the distance between the closest lattice point pairs [30].

$$NOL_{disk}(r,d) = \sum_{x=-\lfloor \frac{r}{d\sqrt{3}} \rfloor}^{\lfloor \frac{r}{d\sqrt{3}} \rfloor} \left(2 \left\lfloor \sqrt{\frac{r^2}{d^2} - 3x^2} \right\rfloor + 1 \right) + \sum_{x=\frac{1}{2}-\lfloor \frac{r}{d\sqrt{3}}+\frac{1}{2} \rfloor}^{\lfloor \frac{r}{d\sqrt{3}}+\frac{1}{2} \rfloor - \frac{1}{2}} 2 \left\lfloor \sqrt{\frac{r^2}{d^2} - 3x^2} + \frac{1}{2} \right\rfloor \quad (A.1)$$

Obviously, a covering center is also a hexagonal lattice point. Thereby, if we want to calculate an upper bound for the number of covering centers of R_i using the number of the lattice points, we should take into account the limit cases i.e., where a covering center is at the boundary or out of R_i. Since the circumradius of a hexagon is r_{com}, we know that a covering center cannot be located at a distance greater than r_{com} from a boundary of R_i. Thus, we calculate the number of lattice points for an extended region R_i' made

from R_i. The extension is performed by increasing the arc of R_i by $2r_{com}$ (i.e., adding r_{com} at each end of the arc) and by increasing the radius of R_i by r_{com}. Therefore, R'_i has a radius of length $r_{map} + r_{com}$ and a central angle $\theta' = \frac{2r_{com}}{r_{map}+r_{com}} + \frac{2\pi}{n}$. Let $NOL_{region}(R'_i)$ denote the number of lattice points in R'_i, we have:

$$\begin{aligned} NOL_{region}(R'_i) &= NOL_{disk}(r_{map} + r_{com}, r_{com}) \times \frac{\theta'}{2\pi} \\ &= NOL_{disk}(r_{map} + r_{com}, r_{com}) \\ &\quad \times \left(\frac{2r_{com}}{r_{map} + r_{com}} + \frac{2\pi}{n} \right) \times \frac{1}{2\pi} \end{aligned} \tag{A.2}$$

As described before, $NOC(R_i) < NOL_{region}(R'_i)$, thus, considering this fact and Eq. (1) of Section 4.3 and Eq. (A.2), we have:

$$L_{scan-path}(v_i) < c_1 + \frac{c_2}{n}$$

where c_1 and c_2 are two constants defined below.

$$c_1 = \sqrt{3} r_{com} \times \left(\frac{r_{com} \times NOL_{disk}(r_{map} + r_{com}, r_{com})}{\pi \times (r_{map} + r_{com})} - 1 \right) \tag{A.3}$$

$$c_2 = \sqrt{3} r_{com} \times NOL_{disk}(r_{map} + r_{com}, r_{com}) \tag{A.4}$$

References

[1] M. Bakht, M. Trower, R. Kravets, Searchlight: helping mobile devices find their neighbors, ACM SIGOPS Oper. Syst. vol. 45 (no. 3) (2012) 71–76.
[2] M.M.B. Tariq, M. Ammar, E. Zegura, Message ferry route design for sparse ad hoc networks with mobile nodes, in: Proceedings of the ACM MobiHoc'06, 2006, pp. 37–48.
[3] B. Bostanipour, B. Garbinato, A. Holzer, Spotcast – a communication abstraction for proximity-based mobile applications, in: Proceedings of the IEEE NCA'12, 2012, pp. 121–129.
[4] B. Bostanipour, B. Garbinato, Improving neighbor detection for proximity-based mobile applications, in: Proceedings of the IEEE NCA'13, 2013, pp. 177–182.
[5] B. Bostanipour, B. Garbinato, Using virtual mobile nodes for neighbor detection in proximity-based mobile applications, in: Proceedings of the IEEE NCA'14, 2014, pp. 9–16.
[6] B. Bostanipour, B. Garbinato, Effective and efficient neighbor detection for proximity-based mobile applications, Elsevier Comput. Netw. J. vol. 79 (2015) 216–235.
[7] B. Bostanipour, B. Garbinato, Neighbor detection based on multiple virtual mobile nodes, in: Proceedings of the IEEE PDP'16, 2016, pp. 322–327.
[8] S. Capkun, J.-P. Hubaux, L. Buttyan, Mobility helps security in ad hoc networks, in: Proceedings of the ACM MobiHoc'03, 2003, pp. 46–56.
[9] I. Chatzigiannakis, S.E. Nikoletseas, P.G. Spirakis, An efficient communication strategy for ad-hoc mobile networks, in: Proceedings of the DISC'01, 2001, pp. 285–299.
[10] C. Cheng, R. Jain, E. van den Berg, Location prediction algorithms for mobile wireless systems, in: M. Illyas, B. Furht (Eds.), Handbook of Wireless Internet, CRC Press, 2003.
[11] A. Cornejo, S. Viqar, J.L. Welch, Reliable neighbor discovery for mobile ad hoc networks, in: Proceedings of the DIALM-PODC'10, 2010, pp. 63–72.
[12] A. Cornejo, S. Viqar, J.L. Welch, Reliable neighbor discovery for mobile ad hoc networks, in: Ad Hoc Networks. 12 (January 2014), 2014, pp. 259–277.
[13] T. Do, D. Gatica-Perez, Contextual conditional models for smartphone-based human mobility prediction, in: Proceedings of the ACM UbiComp'12, 2012, pp. 163–172.
[14] S. Dolev, S. Gilbert, N.A. Lynch, E. Schiller, A.A. Shvartsman, J.L. Welch, Virtual mobile nodes for mobile ad hoc networks, in: Proceedings of the DISC'04, 2004, pp. 230–244.
[15] S. Dolev, S. Gilbert, N.A. Lynch, E. Schiller, A.A. Shvartsman, J.L. Welch, Virtual Mobile Nodes for Mobile Ad Hoc Networks, Tech report lcs-tr-937, MIT, 2004.
[16] S. Dolev, S. Gilbert, E. Schiller, A.A. Shvartsman, J.L. Welch, Autonomous virtual mobile nodes, in: DIALM-POMC, 2005, pp. 62–69.
[17] P. Dutta, D. Culler, Practical asynchronous neighbor discovery and rendezvous for mobile sensing applications, in: Proceedings of the ACM SenSys'08, 2008.
[18] R. Groenevelt, P. Nain, G. Koole, The message delay in mobile ad hoc networks, Elsevier J. Perform. Eval. vol. 62 (2005) 210–228.
[19] M. Grossglauser, D. Tse, Mobility increases the capacity of ad hoc wireless networks, IEEE/ACM Trans. Netw. vol. 10 (no. 4) (2002) 477–486.
[20] M. Grossglauser, M. Vetterli, Locating nodes with EASE: mobility diffusion of last encounters in ad hoc networks, in: Proceedings of the IEEE INFOCOM'03, 2003.
[21] B. Grunbaum, G.C. Shephard, Tilings and Patterns, W. H. Freeman, New York, 1990.
[22] K.P. Hatzis, G.P. Pentaris, P.G. Spirakis, V.T. Tampakas, R.B. Tan, Fundamental control algorithms in mobile networks, in: Proceedings of the ACM SPAA'99, 1999.
[23] Z.J. Haas, T. Small, A new networking model for biological applications of ad hoc sensor networks, IEEE/ACM Trans. Netw. vol. 14 (2006) 27–40.
[24] D. He, N. Mitton, D. Simplot-Ryl, An energy efficient adaptive HELLO algorithm for mobile ad hoc networks, in: Proceedings of the ACM MSWiM'13, 2013, pp. 65–72.
[25] iGroups, http://tinyurl.com/y9cwvmx.
[26] F. Ingelrest, N. Mitton, D. Simplot-Ryl, A turnover based adaptive hello protocol for mobile ad hoc and sensor networks, in: Proceedings of the MASCOTS'07, 2007, pp. 9–14.
[27] I.M. Isaacs, Geometry for College Students, American Mathematical Society, 2009.
[28] A. Kandhalu, K. Lakshmanan, R.R. Rajkumar, U-connect: a low-latency energy-efficient asynchronous neighbor discovery protocol, in: IPSN'10, 2010, pp. 350–361.
[29] B. Korte, J. Vygen, Combinatorial Optimization: Theory and Algorithms, 5 ed., Springer, 2012.
[30] P. Lax, R. Phillips, The asymptotic distribution of lattice points in euclidean and non-euclidean spaces, J. Funct. Anal. 46 (3) (1982) 280–350.
[31] Q. Li, D. Rus, Sending messages to mobile users in disconnected ad-hoc wireless networks, in: Proceedings of the MobiCom'00, 2000.
[32] A. Lindgren, A. Doria, O. Schelen, Probabilistic routing in intermittently connected networks, ACM SIGMOBILE Mob. Comput. Commun. Rev. vol. 7 (2003) 19–20.
[33] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565.
[34] LocoPing, http://www.locoping.com/.
[35] LoKast, http://www.lokast.com/.
[36] Local multiplayer apps, http://appcrawlr.com/ios-apps/best-apps-local-multiplayer.
[37] M.J. McGlynn, S.A. Borbash, Birthday protocols for low energy deployment and flexible neighbor discovery in ad hoc wireless networks, in: Proceedings of the ACM MobiHoc'01, 2001, pp. 137–145.
[38] J. Moy, OSPF – open shortest path first, RFC 1583, 1994.
[39] N.B. Priyantha, A. Chakraborty, H. Balakrishnan, The cricket location-support system, in: Proceedings of the MobiCom'00, 2000.
[40] S. Scellato, M. Musolesi, C. Mascolo, V. Latora, A.T. Campbell, Nextplace: a spatio-temporal prediction framework for pervasive systems, in: Proceedings of the Pervasive'11, 2011, pp. 152–169.
[41] R. Shah, S. Roy, S. Jain, W. Brunette, Data MULEs: modeling a three-tier architecture for sparse sensor networks, Elsevier Ad Hoc Networks Journal vol. 1 (2003) 215–233.
[42] G. Sharma, R. Mazumdar, N. Shroff, Delay and capacity tradeoffs in mobile ad hoc networks: a global perspective, IEEE/ACM Trans. Netw vol. 15 (2007) 981–992.
[43] T. Small, Z.J. Haas, Resource and performance tradeoffs in delay-tolerant wireless networks, in: Proceedings of the ACM WDTN'05, 2005, pp. 260–267.
[44] L. Song, D. Kotz, R. Jain, X. He, Evaluating location predictors with extensive wi-fi mobility data, in: Proceedings of the IEEE INFOCOM'04, 2004, pp. 1414–1424.
[45] L. Song, U. Deshpande, U.C. Kozat, D. Kotz, R. Jain, Predictability of WLAN mobility and its effects on bandwidth provisioning, in: Proceedings of the IEEE INFOCOM'06, 2006.
[46] T. Spyropoulos, K. Psounis, C.S. Raghavendra, Spray and wait: an efficient routing scheme for intermittently connected mobile networks, in: Proceedings of the ACM WDTN'05, 2005, pp. 252–259.
[47] N. Suzuki, Shared Memory Multiprocessing, MIT Press, Cambridge, MA, USA, 1992.
[48] A. Vahdat, D. Becker, Epidemic routing for partially connected ad hoc networks, Tech report cs-200006, Duke University, 2000.
[49] F. Vessaz, B. Garbinato, A. Moro, A. Holzer, Developing, deploying and evaluating protocols with manetlab, in: Proceedings of the NETYS'13, 2013, pp. 89–104.
[50] G. Wattenhofer, G. Alonso, E. Kranakis, P. Widmayer, Randomized protocols for node discovery in ad hoc, single broadcast channel networks, in: Proceedings of the IPDPS'03, 2003.
[51] Waze, https://www.waze.com/en/.
[52] WhosHere, http://whoshere.net/.
[53] W. Zhao, M. Ammar, Message ferrying: proactive routing in highly-partitioned wireless ad hoc networks, in: Proceedings of the IEEE FTDCS'03, 2003, pp. 308–314.
[54] W. Zhao, M. Ammar, E. Zegura, A message ferrying approach for data delivery in sparse mobile ad hoc networks, in: Proceedings of the ACM MobiHoc'04, 2004, pp. 187–198.
[55] W. Zhao, M. Ammar, E. Zegura, Controlling the mobility of multiple data transport ferries in a delay-tolerant network, in: Proceedings of the IEEE INFOCOM'05, vol. 2, 2005, pp. 1407–1418.
[56] X. Zhang, G. Neglia, J. Kurose, D. Towsley, Performance modeling of epidemic routing, Elsevier Comput. Netw. J. vol. 51 (2007) 2867–2891.

Behnaz Bostanipour is a Ph.D. student in computer science at the University of Lausanne in the Distributed Object Programming (DOP) Lab under the supervision of Prof. Garbinato. Before joining the DOP Lab, she obtained a B.Sc. degree and a M.Sc. degree in Communication Systems from Swiss Federal Institute of Technology of Lausanne (EPFL). Her research interests focus on mobile ad hoc networks and distributed computing. In particular, she studies and designs new abstractions and algorithms for proximity-based mobile applications running on mobile devices and smartphones.

Benoit Garbinato is a professor in computer science at the University of Lausanne since 2004, where he leads the Distributed Object Programming (DOP) Lab. In the nineties, he contributed to the emerging research trend on separation concerns and protocol composition in fault-tolerant distributed systems, as part of his Ph.D. thesis. He then worked in the industry, first for the research lab of UBS in Zurich (Ubilab), where he lead the software engineering group, and later for Sun Microsystems professional services, as senior software architect. Since his return to the academic world, his research and teaching activities focus on the design and implementation of adequate programming abstractions for emerging distributed architectures, such as pervasive and mobile systems. He has over 50 publications in international conferences and journals, and is member of the ACM and the IEEE societies.