Stabilizing Voronoi Diagrams for Sensor Networks

Jorge A. Cobb

Department of Computer Science
The University of Texas at Dallas
Richardson, Texas 75080
Email: cobb@utdallas.edu

***Abstract*—Wireless sensor networks are characterized by their lack of physical resources, such as memory, battery power, and communication bandwidth. For this reason, every protocol in the network should be as efficient as possible. For scalability, and given that many sensor networks are deployed to cover a large area, the paradigm of geographical routing has been proposed in the literature. In particular, the Voronoi diagram, where the sensor locations act as generator points in the two-dimensional plane, serve as the foundation of some of these routing protocols. Existing protocols for creating the Voronoi diagram are either not fault-tolerant or not fully distributed. In this paper, we present the first protocol that is fully distributed and resilient to a wide variety of faults. In particular, the protocol is *stabilizing*, i.e., it will converge to a normal operating state regardless of the initial value of its variables.**

Keywords–Stabilizing systems; Voronoi diagram; Delaunay triangulation; Sensor networks.

I. INTRODUCTION

Consider a wireless network consisting of a large number of sensor nodes distributed over a geographical area. Each sensor has limited resources, such as memory and battery lifetime, and is capable of sensing its surroundings up to a certain distance. Due to the limited resources, it is crucial that each task performed by the sensor nodes consumes the least possible amount of memory and energy [1].

Greedy routing protocols have been proposed as a scalable solution for routing in large-scale wireless networks, such as large deployments of sensor networks [2]–[5]. In greedy routing, the routing state needed per node is independent of network size. This makes greedy routing attractive for the resource-starved sensor networks. Greedy routing is also known as geographic routing because, for a packet with destination d, a node u selects as the next hop to d a neighbor that minimizes the physical distance from u to d.

For nearly a century, the Voronoi diagram, and its dual, the Delaunay triangulation [6], have had a strong impact on various fields of science and engineering. In the particular context of network routing, Delaunay triangulations are well suited for greedy routing [7]. In general, greedy routing on an arbitrary graph may become trapped at a local minimum and not reach the destination. However, on a Delaunay triangulation, greedy routing is guaranteed to reach the destination.

In this paper, we develop a distributed protocol where each node can compute its Voronoi region, and thus, is able to support greedy routing. Given that the objective is to support greedy routing, the protocol does not require an additional routing mechanism that can be used to aid in node communication. The only assumption is that each node can communicate with other nodes within its wireless transmission radius.

In addition to being distributed, our solution is *stabilizing* [8]–[11], i.e., starting from *any* state, a subsequent state is reached and maintained where the sensors become aware of their Voronoi region. A system that is stabilizing is resilient against transient faults, because the variables of the system can be corrupted in any way (that is, the system can be moved into an arbitrary configuration by a fault), and the system will naturally recover and progress towards a normal operating state. Thus, stabilizing systems are resilient against node failures, node additions, undetected corrupted messages, and improper initialization states.

Distributed protocols exist in the literature that allow each node to obtain its Voronoi region. However, they do not exhibit all our desired features. Algorithms such as those in [12] are fully distributed, but they are not fault tolerant, and they assume an underlying routing protocol exists. Works designed for wireless greedy routing make no such assumption [13] [14], but they have limited fault-tolerance, and in particular, are not stabilizing. Solutions that are distributed and stabilizing exist [15], but they also assume an underlying routing protocol, and are thus not suitable for greedy routing.

The paper is organized as follows. Section II presents a review of Voronoi diagrams and Delaunay triangulations. Section III discusses our approach to find paths between Voronoi neighbors. Section IV discusses the information that must be exchanged between nodes in order to find paths between Voronoi neighbors. Section V presents a more detailed specification of the basic protocol. This protocol is not fault-tolerant, and thus, Section VI presents enhancements to enforce stabilization, and argues their correctness. Conclusions and future work are presented in Section VII.

II. VORONOI DIAGRAMS AND NETWORK MODEL

In this section, we review Voronoi diagrams and Delaunay triangulations. In addition, we present our network model and its relationship to Delaunay triangulations.

A. Voronoi Diagrams and Delaunay Triangulations

As shown in Figure 1(i), consider two points, a and x, in the two-dimensional Euclidean plane. The line segment from a to x is shown with dots, and the solid line corresponds to the perpendicular bisector of this line segment. Observe that any point below the bisector is closer to a than to x. Similarly, any point above the bisector will be closer to x than to a.

 ISBN: 978-1-61208-753-5

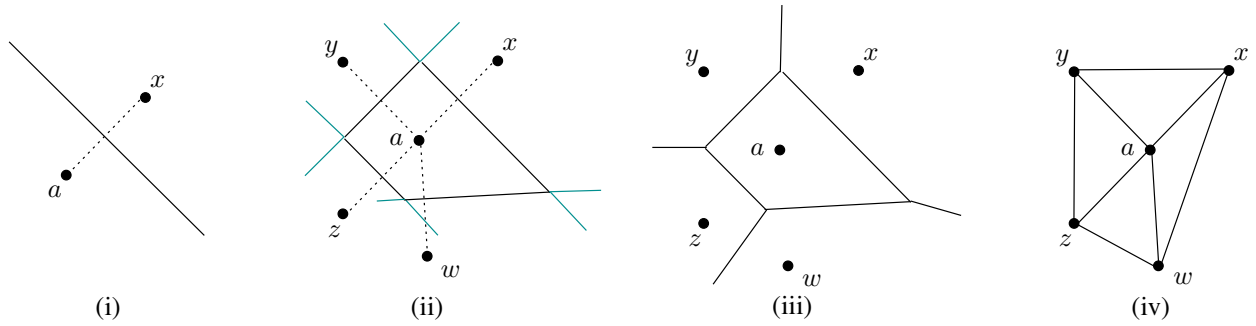

Figure 1. Voronoi diagram.

A *Voronoi diagram* (VD) consists of a set of *generator points* $P = p_1, p_2, \ldots, p_n$ and a set of regions $R = R_1, R_2, \ldots, R_n$. Each R_i consists of all points on the plane that are closer to p_i than to any other generator point in P. In Figure 1(i), $P = \{a, x\}$, R_a are all points below the bisector, and R_x are points above the bisector.

Figure 1(ii) shows the region R_a after a few more generator points are added. Region R_a becomes the convex hull obtained from the intersection of all the bisectors with all other generator points. Finally, Figure 1(iii) shows the regions of all five generator points.

An equivalent structure to the VD is the *Delanuay triangulation* (DT), shown in Figure 1(iv). Here, there is an edge between a pair of generator points p_i and p_j iff R_i and R_j share a face. E.g., point x has three edges: (x, y), (x, a), (x, w), because R_x shares a face with each of the regions R_a, R_y, and R_w. Thus, both the VD and the DT have the same information, but presented in different form.

B. Network Model and Connectivity

We consider a two-dimensional Euclidean space in which a total of n sensor nodes have been placed. Each sensor is assumed to have a transmission radius r. Thus, if the distance between any pair of sensors is less than r, then the pair is able to exchange data messages.

As discussed earlier, sensor nodes correspond to point generators, and each sensor node has the objective of identifying each of its neighbors in the DT (equivalently, the VD). I.e., each sensor node must learn the location of all other sensor nodes with whom it shares a DT edge. Throughout the paper, we use DT and VD interchangeably.

Let $T(u)$ be the set of nodes that are within transmission range of u, i.e., *distance*$(w, u) \leq r$ iff $w \in T(u)$. Let $V(u)$ be the set of neighbors of u in the DT. These are referred to as the *Voronoi neighbors* of u. In Figure 1(iv), $V(x) = \{a, w, y\}$. Some of the nodes in $V(u)$ will be within transmission range of u, and thus, also in $T(u)$, while others will be farther away. The nodes in $V(u) \cap T(u)$ are said to be the *direct* Voronoi neighbors of u.

We assume that the sensor network is connected. I.e., for every pair of nodes u and v, there is a path of nodes $w_1, w_2, \ldots, w_k$, such that $w_1 = u$, $w_k = v$, and for each i, $1 \leq i < k$, $w_{i+1} \in T(w_i)$.

Note that it is possible for $w \in T(u)$ but $w \notin V(u)$. This is because other nodes can be in between u and w, and thus, the Voronoi region of u does not overlap that of w. I.e., the fact that nodes can communicate directly does not imply that they are Voronoi neighbors, and vice versa.

In order to learn its set of Voronoi neighbors, each node must be able to communicate with each of them, often indirectly via direct neighbors. For efficiency, we expect each node u to keep as little information as possible, in particular, in the order of $|V(u)|$, which is much smaller than the number of nodes in the network. To do so, we restrict ourselves to *only communicate via direct Voronoi neighbors*, that is, only with nodes in $V(u) \cap T(u)$. Thus, between any pair of nodes in the network, there must exist a path using only direct Voronoi neighbors. Otherwise, the network becomes, in effect, disconnected. We show below that such a path always exists. Before doing so, we present some definitions.

Consider a pair of Voronoi neighbors (u, v). A *Voronoi path* is a sequence of edges in the DT starting at u and ending in v. A *direct Voronoi path* from u to v is a Voronoi path where all the edges are direct edges. That is, there is a sequence of nodes, $w_1, w_2, \ldots, w_k$, such that $w_1 = u$, $w_k = v$, and for each i, $1 \leq i < k$, $w_{i+1} \in V(w_i) \cap T(w_i)$.

Theorem 1: (*Connectivity*) For every pair of nodes, u and v, there exists a direct Voronoi path from u to v.

Proof:
The proof is by contradiction. We assume that no such path exists. Therefore, as shown in Figure 2(i), there must exist a cut of the VD such that every pair of Voronoi neighbors have a distance greater than r between them.

From our network model, the sensor network is connected. Thus, there must exit a node a on one side of the cut and another node b on the other side of the cut such that *distance*$(a, b) \leq r$. From the definition of the cut, a and b are not Voronoi neighbors. This pair of nodes is shown in Figure 2(ii). The vertical dashed line is the bisector between a and b.

Let t be the neighbor of a such that the face that R_a and R_t share crosses the line segment between a and b. In order for the face of R_t to prevent a and b from being Voronoi neighbors, it must be that *distance*$(a, t) <$ *distance*(a, b).

Consider then the cut of the VD. Recall that a and b are on opposite sides. We have two cases to consider: the cut crosses the line segment between a and t, or it crosses the line segment between t and b.

The former case is not possible. This is because a and t are Voronoi neighbors, and, being in opposite sides of the cut,

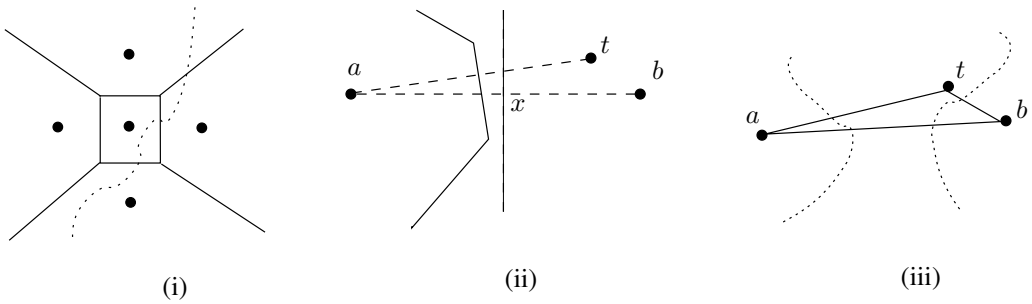

Figure 2. Voronoi path connectivity.

it must be that $distance(a,t) > r$. However, this contradicts what we have shown above, i.e., that

$$distance(a,t) < distance(a,b) \leq r.$$

In the latter case, we have a pair of nodes, t and b, on opposite sides of the cut, such that

$$distance(t,b) < distance(a,b) \leq r.$$

If t and b are Voronoi neighbors, then this is also a contradiction by the definition of the cut. If they are not, then we have found a pair of nodes, t and b, on opposite sides of the cut, such that their distance is smaller than the distance between a and b. Thus, the same argument can be applied again for t and b. I.e., either a contradiction is reached, or we obtain another pair of nodes with lesser distance. Since the different distances between nodes is finite, a contradiction must be reached. ■

III. Routing along Triangulation Edges

Recall that our objective is for each node to be aware of its Voronoi neighbors. However, some of those neighbors may not be direct neighbors. For example, consider Figure 1(ii). Assume that a has both x and w as direct neighbors. However, although x and w are Voronoi neighbors, they are not direct neighbors, due to their large distance between them. For x and w to learn about each other, it must be done through a.

In general, if Voronoi neighbors are not direct neighbors, they learn about each other via an intermediate node. As a node learns about its neighbors, it is then in a position to be an intermediate node and inform two of its neighbors about each other, and the process continues until all nodes are aware of all their Voronoi neighbors.

In this section, we show how to obtain a path between any pair of Voronoi neighbors, where the path consists only of direct edges in the DT. As shown earlier, such a path must exist. We begin by assigning a label to each edge in the DT, and use these labels to obtain the desired path.

A. Edge Labels and Neighbor Paths

Each edge in the DT can take part in at most two triangles. For example, in Figure 3, edge (b,h) takes part in triangle (i,b,h) and triangle (b,c,h). On the other hand, edge (i,g) belongs only to triangle (i,h,g).

We define a *Voronoi neighbor path* of an edge (a,z), denoted $VNP(a,z)$, as follows. If a and z are direct neighbors, then $VNP(a,z)$ is just the edge itself. Assume instead that they are not direct neighbors, and consider Figure 1(iv).

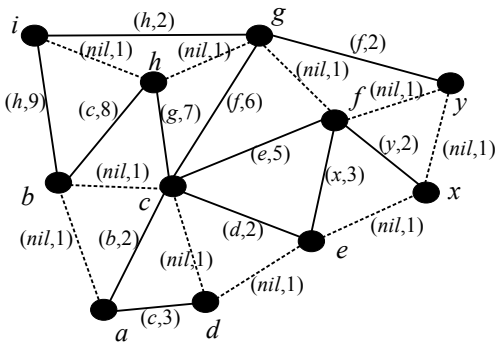

Figure 3. Edge label example.

Edge (a,z) takes part in two triangles: (a,y,z) and (a,w,z). Then, $VNP(a,z)$ is the concatenation of $VNP(a,y)$ with $VNP(y,z)$, or it is the concatenation of $VNP(a,w)$ with $VNP(w,z)$, whichever of these two yields the least number of direct edges.

Each edge is also considered to have a positive integer label that corresponds to the length, in direct Voronoi edges, of the Voronoi neighbor path. Labels can thus be defined recursively to be the smallest value that satisfies the following.

- If u and v are direct neighbors, then $label(u,v) = 1$.
- If u and v are not direct neighbors, and edge (u,v) takes part in triangles (u,x,v) and (u,y,v), then,

$$label(u,v) = min \begin{cases} label(u,x) + label(x,v) \\ label(u,y) + label(y,v) \end{cases}.$$

A simple induction proof can show that the label of each edge is well defined, and that edges with label $h+1$ can be computed once all edges with labels h or less have been computed.

Each non-direct edge is associated with a *hinge* node. The hinge node is the neighbor that defines the Voronoi neighbor path of the edge. In Figure 1(iv), the hinge node of (a,z) will be either y or w, in particular, it will be y if $VNP(a,y) : VNP(y,z)$ is shorter than $VNP(a,w) : VNP(w,z)$, where ':' denotes concatenation. In the event that both y and w provide the same length, we break ties alphabetically.

Figure 3 provides an example of the labels assigned to the DT of a group of nodes. Dashed edges indicate direct edges. All direct edges have no hinge and a label equal to one. Edge (c,a) has a label of two and its hinge is b, because b has a direct edge to each of c and a. Similarly, edges (c,e) and (i,g)

have labels equal to two. Edge (c, g) has a label of six with f as the hinge, because the labels of its edges (f, c) and (f, g) are five and one, respectively.

Finally, note that, in any triangle (u, v, w), there is one and only one node that can be the hinge of some edge in the triangle. For example, consider the triangle (f, c, g) in Figure 3. We have that $f = \mathit{hinge}(g, c)$, but $c \neq \mathit{hinge}(f, g)$ and $g \neq \mathit{hinge}(c, f)$. In general, we define the hinge of a triangle to be the node that is the hinge of the edge consisting of the other two nodes.

B. Finding Paths to Neighbors

We next address how to find a direct Voronoi path between any pair of Voronoi neighbors u and v. This path can be obtained recursively from the definition of edge labels as follows.

$$\mathit{path}(u, v) = \begin{cases} (u, v) & \text{if } \mathit{label}(u, v) = 1 \\ \mathit{path}(u, w) : \mathit{path}(w, v) & \text{if } w = \mathit{hinge}(u, v) \end{cases}$$

Consider again Figure 3, and finding a path from e to f. From the definition of $path(e, f)$, we have:

$$\begin{array}{rl} & \mathit{path}(e, f) \\ = & \mathit{path}(e, x) : \mathit{path}(x, f) \\ = & (e, x) : \mathit{path}(x, f) \\ = & (e, x) : \mathit{path}(x, y) : \mathit{path}(y, f) \\ = & (e, x) : (x, y) : \mathit{path}(y, f) \\ = & (e, x) : (x, y) : (y, f) \end{array}$$

A node does not need to know the entire topology in order to communicate with its neighbors. We show below that the only required information is the list of neighbors forming its Voronoi region, $R(u)$, and the label of each. Note that $R(u)$ is the same as the neighbors of u in the DT. E.g., in Figure 3, $R(h)$ consists of nodes i, b, c, and g. $R(f)$ consists of nodes c, g, y, x and e, while $R(a)$ consists of nodes b, c, and d.

Before discussing how neighboring nodes communicate, we begin by dividing a node's region into disjoint *segments*.

C. Segments of a Region

The set of direct neighbors of u will be denoted by $\mathit{core}(u)$. Node u can obtain this set from the convex-hull of nodes within transmission range of u.

$$\mathit{core}(u) = \mathit{convex\text{-}hull}(T(u))$$

That is, it is the subset of $T(u)$ obtained from the convex-hull of the bisectors from u to each element in $T(u)$. These nodes form the foundation for $R(u)$, as follows.

Lemma 1: (*Core in region*) For all u, $\mathit{core}(u) \subseteq R(u)$.

Proof:
If $v \in \mathit{convex\text{-}hull}(T(u))$, it implies that no node within transmission range can block the face of v in the convex-hull of $T(u)$. In order for v not to be in $R(u)$, the face that it provides to $R(u)$ must be blocked by other neighbors whose distance to u is closer than v's. Hence, these nodes must also be in $T(u)$. However, since v is also in the convex hull of $T(u)$, no node in $T(u)$ can block v. Hence, no node can block v from being in $R(u)$. ■

For terseness, we use the terms *right* and *left* instead of clockwise and counter-clockwise, respectively. Additionally, we use *dir* to represent either *right* or *left*, and $\neg$*dir* to represent the opposite direction of *dir*.

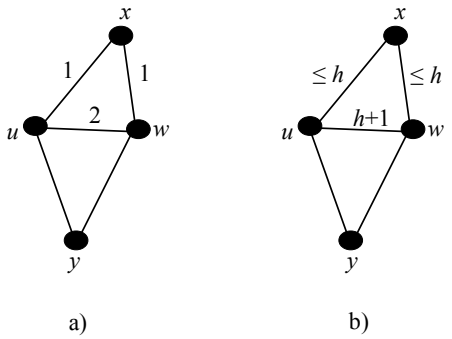

Figure 4. Segment structure induction.

Let v be any Voronoi neighbor of u. Let $\mathit{next}(u, v, \mathit{dir})$ be the next node along direction *dir* on region $R(u)$. For example, in Figure 3, $\mathit{next}(h, i, \mathit{right}) = g$, and $\mathit{next}(h, g, \mathit{left}) = i$. Also, for any pair of neighbors v and w of u, let $\mathit{bet}(u, v, w, \mathit{dir})$ denote the sequence of nodes found in $R(u)$ along direction *dir* starting from v and ending in w.

Let $\mathit{segment}(u, v, \mathit{dir})$ be the longest sequence of nodes, $w_0, w_1, \ldots, w_j$, along the periphery of $R(u)$, starting from core node v, $v \in \mathit{core}(u)$, such that:

- $w_0 = v$,
- for each i, $0 \leq i < j$, $w_{i+1} = \mathit{next}(u, w_i, \mathit{dir})$, and
- for each i, $0 \leq i < j$, $\mathit{hinge}(u, w_{i+1}) = w_i$.

As an example, consider node f in Figure 3. We have:

$$\begin{array}{rcl} \mathit{segment}(e, x, \mathit{left}) & = & \langle x, f \rangle \\ \mathit{segment}(e, d, \mathit{right}) & = & \langle d, c \rangle \\ \mathit{segment}(f, y, \mathit{left}) & = & \langle y \rangle \\ \mathit{segment}(f, y, \mathit{right}) & = & \langle y, x, e, c \rangle \end{array}$$

Note that all nodes in $R(f)$ are contained in some segment of f. Also, the segments that make up $R(f)$ do not overlap with each other, other than at their starting core node or their last node. We argue below that this is true for all nodes.

From the definition of a segment, a few intuitive definitions follow. Given a neighbor w of u, we define the root, $\mathit{root}(u, w)$, to be the core neighbor v of u such that w is contained in $\mathit{segment}(u, v, \mathit{dir})$, for some $\mathit{dir} \in \{\mathit{left}, \mathit{right}\}$. We argue below that all neighbors must have a root. In the figure, $\mathit{root}(e, f) = x$, $\mathit{root}(e, c) = d$, and $\mathit{root}(e, d) = d$. Finally, let $\mathit{last}(u, v, \mathit{dir})$ be the final element in $\mathit{segment}(u, v, \mathit{dir})$. Thus, $\mathit{last}(e, x, \mathit{left}) = f$, and $\mathit{last}(e, d, \mathit{right}) = c$.

Theorem 2: (Segment structure): For every non-core node w in $R(u)$, there exists a core node v of u and a direction *dir*, such that:

- $w \in \mathit{segment}(u, v, \mathit{dir})$.
- all nodes in $\mathit{segment}(u, v, \mathit{dir})$, other than v, are not core nodes.
- Let p be the node previous to w in $\mathit{segment}(u, v, \mathit{dir})$, i.e., $p = \mathit{next}(u, w, \neg\mathit{dir})$. Then,
  - $\mathit{hinge}(u, w) = p$.
  - $\mathit{label}(u, w) = \mathit{label}(u, p) + \mathit{label}(p, w)$.

Proof:
The proof is by induction over the labels associated with the edges between u and neighbors in $R(u)$, that is, between u and w in the statement of the theorem.

Consider an arbitrary Voronoi edge, (u, w), with $\mathit{label}(u, w) = 2$, shown in Figure 4(a). There are only two possible nodes (one on each side) that can be the hinge, namely, x and y. Let x be the hinge node. Then, by the definition of a label, $\mathit{label}(u, x) = \mathit{label}(w, x) = 1$. Thus, x is a core node, and w belongs to the segment of x.

Assume now that the label of (u, w) is $h+1$, and all edges (u, v) in $R(u)$ with label at most h satisfy the theorem. Again, there are only two possible nodes that can be the hinge of this edge, as depicted in Figure 4(b). Without loss of generality, let x be the hinge node. From the definition of edge labels, the labels of (x, u) and (x, w) are both at most h.

From the induction hypothesis, there is a segment of u that contains x. This segment cannot begin from the direction of w and y, because all edges from any node in the segment to node u must have a label no greater than h, and edge (w, u) has label $h + 1$. Hence, the segment for x begins along the direction of x, and extending this segment by edge (w, u) with label $h + 1$ satisfies the theorem. ■

IV. Segment Construction

The objective of the protocol is for each node u to become aware of its region $R(u)$. A consequence of Theorem 2 is that $R(u)$ is divided into disjoint segments. Consider Figure 5(a). It shows $R(u)$ and the different segments it comprises. Bold dashed edges belong to core nodes of $R(u)$, and thin dashed edges separate one segment from another. For example, $\mathit{segment}(u, i, \mathit{right}) = \langle i, j, k\rangle$, while $\mathit{segment}(u, m, \mathit{left}) = \langle m, l\rangle$. These two segments do not end at the same node; there is an edge, (k, l), along the rim of $R(u)$, that does not belong to either segment. From the theorem, neither k nor l can be the hinges of trianlge (k, u, l). Otherwise, the segments would both end at either k or at l. Thus, it must be that $\mathit{hinge}(u, k, l) = u$. Similarly, $\mathit{hinge}(u, i, p) = u$ and $\mathit{hinge}(u, m, n) = u$.

Consider $\mathit{segment}(u, i, \mathit{right})$. From the theorem, $\mathit{hinge}(u, i, j) = i$ and $\mathit{hinge}(u, j, k) = k$. Because u is not the hinge of any of these triangles, information about the existence of these triangles is expected to be received from the root of the segment, i.e., from core node i. Similarly, information about the existence of triangle (u, l, m) is expected to be received from core node m. Once u learns of these segments, it makes the assumption that the hinge of (k, u, l) is itself. It is thus *its responsibility* to inform both k and l of the triangle (k, u, l). In this case, we say that u *is joining* nodes k and l.

To do this join, u will communicate with k and l via their root core nodes i and m, respectively. Node u must inform k that it has a neighbor l, and in addition, what u believes is the label of (k, l). This is represented by the following tuple:

$$\langle k, l, \mathit{label}(k, l)\rangle.$$

Node u must ensure this tuple reaches k. Similary, u sends the tuple

$$\langle l, k, \mathit{label}(k, l)\rangle$$

to node l. In general, joining tuples are of the form

$$\langle \mathit{destination}, \mathit{neighbor}, \mathit{edge\text{-}label}\rangle.$$

Tuple $\langle k, l, \mathit{label}(k, l)\rangle$ has to be routed towards k. To do so, u forwards it to the root node, i. Recursively, node i forwards it in the direction of k, in particular, first in the direction of j. Each of edges (i, j) and (j, k) may correspond to simply a direct edge (label one), or to a longer transmission path requiring crossing multiple direct Voronoi edges.

Note, however, that l may also desire to join a pair of nodes in its region $R(l)$, such as σ and ρ in Figure 5(b), and this join needs to be sent to σ. If u is in the segment of $R(l)$ that contains σ, then this join tuple created by l will eventually reach u. The task of u is to forward this join towards k, which is the next node along the segment of $R(l)$ containing σ. Thus, u has to forward *two* tuples to k (via core node i): a tuple created by u, and a tuple created by l (that was received via core node m). This argument can be extended further, because ρ may also be joining two nodes, r and s in the figure, and the join tuple may need to reach σ. Thus, the tuple is sent to l, who in turn has to send it to u, who in turn has to send it to k (via core node i).

In summary, u does not send an individual tuple to its neighbor k (via core neighbor i), it sends a *stack of tuples*. In this stack, the tuple generated by u is at the top. The remainder of the stack consists of tuples that l wants to forward to k. This in turn contains the tuple from l joining σ and ρ, plus the stack of tuples that ρ wants to send to σ, etc..

Finally, by symmetry, u has to forward to l (via core neighbor m) a stack of tuples that it received from k (via core neighbor i).

V. Hull Construction Protocol

Before presenting our method in more detail, we first give a brief overview of the notation.

A. Protocol Notation

The notation used originates from [10] [11], and is typical for specifying stabilizing systems. The behavior of each node is specified by a set of inputs, a set of variables, a set of parameters, and a set of actions.

The inputs declared in a node can be read, but not written, by the actions of that node. The variables declared in a node can be read and written by the actions of that node. For simplicity, a shared memory model is used, i.e., each node u is able to read the variables of nodes in $T(u)$. To maintain a low atomicity, and thus an easier transition to a message-passing model (see Section VII), each action is able to read the variables of a *single* neighbor.

Every action in a node u is of the form:

$$<\text{guard}> \rightarrow <\text{statement}>.$$

The <guard> is a boolean expression over the inputs, variables, and parameters declared in the node, and also over the variables declared in a single node in $T(u)$. The <statement> is a sequence of assignment, conditional, and iteration statements that change some of the variables of the node.

The parameters declared in a node are used to write a set of actions as one action, with one action for each possible value of the parameters. For example, if the following parameter definition is given,

par g : 1 .. 2

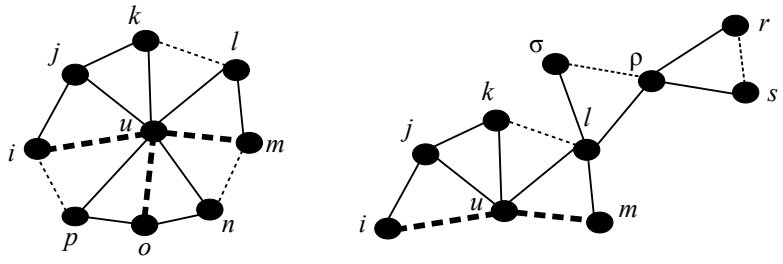

Figure 5. Segment construction.

then the following action

$$x = g \quad \rightarrow \quad x := x + g$$

is a shorthand notation for the following two actions.

$$\begin{array}{lll} x = 1 & \rightarrow & x := x + 1 \\ x = 2 & \rightarrow & x := x + 2 \end{array}$$

An execution step consists in evaluating the guards of all the actions of all nodes, choosing an action whose guard evaluates to true, and executing the statement of this action. An execution consists of a sequence of execution steps, which either never ends, or ends in a state where the guards of all the actions evaluate to false. All executions are assumed to be weakly fair, that is, an action whose guard is continuously true must be eventually executed.

To distinguish between variables of different nodes, the variable name is prefixed with the node name. For example, variable $x.v$ corresponds to variable v in node x. If no prefix is given, then the variable corresponds to the node whose code is being presented.

B. Method

We next present our protocol in more detail. In particular, we present the specification of an arbitrary node u.

Node u is aware of nodes in $T(u)$, and thus also of *core*(u), because they are within transmission range. We thus assume the core nodes are simply an input to node u.

We represent region $R(u)$ by the two-dimensional array E.

$$E \ : \ \textbf{array}[core][\textit{left} \ . \ . \ \textit{right}] \ \textbf{of sequence of ID}$$

The first index corresponds to the core nodes. Each core node can be the root of at most two segments: one in each direction. Thus, the second index is the direction. The element stored in $E[i][dir]$ corresponds to the sequence of nodes in *segment*(u, i, dir). For example, in Figure 5, *segment*$(u, i, right) = \langle i, j, k \rangle$, while *segment*$(u, i, left) =$ **nil** because there is no segment counter-clockwise starting at i. Note, however, that some core neighbors may be the root of both a *left* and a *right* segment, such as core neighbor o.

The labels of each neighbor of u are stored in array L. This is a one-dimensional array, with one element per neighbor.

As mentioned in Section IV, each node may have to send a stack of tuples to each of its core neighbors. These stacks are stored in array $send$.

$$send \ : \ \textbf{array}[core][\textit{left} \ . \ . \ \textit{right}] \ \textbf{of stack of} \ (\textbf{ID}, \textbf{ID}, \textbf{integer})$$

The second dimension of the array is necessary because, as mentioned above, a core node can be the root of a segment in each direction. The complete algorithm is shown below.

node u
inp
  $core$: **set of ID** {core neighbors}
var
  E : **array**[$core$][*left* . . *right*] **of sequence of ID** {region edges}
  L : **array**[**ID**] **of integer** {edge labels}
  $send$: **array**[$core$][*left* . . *right*] **of stack of** (**ID**, **ID**, **integer**) {forwarded edges}
  $rcvd$: **stack of** (**ID**, **ID**, **integer**)
  k, l : **ID**
par
  dir : *left* . . *right*
  i : **ID**
begin
  $i \in core \ \rightarrow$
    $rcvd := i.send[u][dir]$;
    build-segment$(rcvd, i, dir)$;
    $m :=$ *next-core*(i, dir);
    if *last*$(i, dir) \neq$ *last*$(m, \neg dir)$ **then** {join segments}
      $k :=$ *last*$(E[i][dir])$;
      $l :=$ *last*$(E[m][\neg dir])$;
      $send[m][dir] := (l, k, L[l] + L[k]) : rcvd$;
    else
      $send[m][dir] :=$ **nil**
    end if
end

Node u consists of a parameterized action. Since the action has two parameters, i and dir, it is a shorthand for many actions: one action for every combination of a core neighbor of u and a value from $\{left, right\}$. Node u reads the tuples that a neighbor i is sending to u along direction dir. These tuples are stored in a temporary array $rcvd$. Then, several steps are taken to build the segment whose root is i. These steps are captured in *build-segment*$(rcvd, i, dir)$ as shown in Figure 6.

Consider Figure 5 as an example. The stack of tuples expected to be received are, first, a tuple from i joining u and j, followed by a tuple from j joining u and k. These tuples are removed from $rcvd$ one at a time and the appropriate edges and labels are added to E and L (we denote concatenation via ':'). The remaining tuples do not have u as the destination,

build-segment$(rcvd, i, dir)$
$E[i][dir] :=$ **nil**; $L[i] := 1$;
$(dst, neigh, label) :=$ **top**$(rcvd)$;
while $(dst = u)$ **do**
$E[i][dir] := E[i][dir] : neigh$;
$L[neigh] := label$;
pop$(rcvd)$;
$(dst, neigh, label) :=$ **top**$(rcvd)$;
end while

next-core$(v, dir) = w \Leftrightarrow$
$\langle \forall x : (x \in bet(u, v, w, dir) \wedge x \notin \{v, w\}) \Rightarrow x \notin core(u) \rangle$

Figure 6. Auxiliary definitions.

and these are not processed by *build-segment*$(rcvd, i, dir)$.

The action continues by finding the next core neighbor along direction dir. Let m be this node (potentially i itself). The action checks if the segments of i and m need to be joined. If so, it sets $send[m][dir]$ to the single tuple $(l, k, L[l] + L[k])$. This tuple will be propagated by m along the segment until it reaches its destination l. In addition, the tuples remaining in $rcvd$ correspond to the tuples that k needs to forward to l via u. These tuples (if any left) are forwarded via core neighbor m. Thus, they are concatenated to the end of $send[m][dir]$.

VI. Stabilization

We next describe the changes that are necessary to strengthen our protocol and achieve stabilization. We begin with a formal definition of stabilization.

A *predicate* P of a network is a boolean expression over the variables in all nodes of the network. A network is called *P-stabilizing* iff every computation has a suffix where P is true at every state of the suffix [9] [11].

Stabilization is a strong form of fault-tolerance. Normal behavior of the system is defined by predicate P. If a fault causes the system to a reach an abnormal state, i.e., a state where P is false, then the system will converge to a normal state where P is true, and remain in the set of normal states as long as the execution remains fault-free.

We will add stabilization in two steps. First, local sanity checks ensure that the data currently available to a node meets the criteria of predicate P. Data that does not meet the sanity checks is simply discarded. Second, a method is introduced to ensure that information being propagated from node to node has a limit on the distance it can propagate from its source. In this way, incorrect information has a limit on its propagation. This, in combination with the sanity checks, ensures that the system returns to its normal operating state defined by P.

The variables of the updated protocol remain as before. The updated actions are given below. The specific predicate P and a proof of stabilization is given in Section VI.

begin
{core sanity }
$core \neq$ *convex-hull*$(T) \quad \rightarrow$
$core :=$ *convex-hull*(T)

{region sanity}
$\neg$*region-sanity*$(E, L) \quad \rightarrow$
for $k \in core, d \in$ *left .. right*, **do**
$E[k][d] :=$ **nil**

{receive edges from neighbor}
$i \in core \quad \rightarrow$
$rcvd := i.send[u][dir]$;
build-segment$(rcvd, i, dir)$;
$m :=$ *next-core*(i, dir);
if *last*$(i, dir) \neq$ *last*$(m, \neg dir)$ **then**
$k :=$ *last*$(E[i][dir])$;
$l :=$ *last*$(E[m][\neg dir])$;
$send[m][dir] :=$
$(l, k, L[l] + L[k], L[k]) :$ *hops-1*$(rcvd)$;
$send[m][dir] :=$ *filter*$(send[m][dir])$;
else
$send[m][dir] :=$ **nil**
end if
end

A. New Actions

The first action is for sanity of the set of core nodes. Because nodes may fail and new nodes may join the network, the set of core neighbors of a node u becomes a variable, rather than an input. In addition, node u is aware of $T(u)$ because it is in direct communication with these nodes. Thus, $T(u)$ becomes an input to u. The core sanity action simply ensures the correct membership of the core set.

The second action is for sanity on the region formed by the segments stored in $u.E$. Node u can perform local tests on $u.E$ to ensure its values are consistent. Once this action is executed, and $u.E$ is consistent, the remaining actions are written so that if $u.E$ is in a consistent state before the action, it will remain in a consistent state after the action. Therefore, $u.E$ will be in an inconsistent state only immediately after a fault.

Above, predicate *region-sanity*(E, L) is the conjunction of the following four conditions.

1) Let E^* be the union of all nodes contained in any segment of u. Then,
$$convex\text{-}hull(T \cup E^*) = E^*.$$
That is, the segments themselves form a convex hull containing all the core nodes.
2) Segments have no nodes in common, except that adjacent segments may have the same last node.
$\langle \forall x, (c, d) \neq (c', d') :$
$x \in (E[c][d] \cap E[c'][d']) \Rightarrow$
$(x =$ *last*$(E[c][d]) \wedge (x =$ *last*$(E[c'][d']) \wedge$
next-core$(c, d) = c' \wedge d \neq d')\rangle$
3) Nodes within the same segment should be unique.
$\langle \forall c, d, m, n :$
$(m < n \wedge E[c][d](m) = E[c][d](n)) \Rightarrow$
$E[c][d](n) =$ **nil**$\rangle$
Above, the m^{th} element in the sequence $E[c][d]$ is denoted by $E[c][d](m)$.
4) Labels should be increasing from one node to the next.
$\langle \forall c, d, n > 0 : E[c][d](n) \neq$ **nil** $\rightarrow$
$L(E[c][d](n)) > L(E[c][d](n-1))\rangle$

```
build-segment(rcvd, i, dir)
    E[i][dir] := nil; L[i] := 1;
    (dst, neigh, label, hops) := top(rcvd);
    while (dst = u) do
        E[i][dir] := E[i][dir] : neigh;
        L[neigh] := label;
        for each x, x ∈ E ∧ x ∉ convex-hull(E) do
            E := E − x;
        end for
        if ¬region-sanity(E, L) then
            E[i][dir] := E[i][dir] − neigh;
        end if
        pop(rcvd);
        (dst, neigh, label, hops) := top(rcvd);
    end while
```

Figure 7. Modified *build-segment* routine.

```
filter(stack)
    temp := stack;
    stack := nil;
    hops := 1;
    label := 2;
    while temp ≠ nil
        (d, n, l, h) := top(temp);
        if h ≥ hops ∧ l ≥ label then
            stack := stack : (d, n, l, h);
            hops := max(hops, h);
            label := max(label, l) + 1;
        end if;
        pop(temp);
    end while
```

Figure 8. Ensuring sanity in $send$ array.

B. Strengthening Existing Actions

Although some of the information that a node u maintains can be checked locally for consistency, u cannot determine if the tuples that it forwards from one node to another are consistent. To ensure that faulty information is not propagated indefinitely, each tuple is assigned a *hop count*, that is decremented each time the tuple is forwarded from one node to the next, and it is discarded if it reaches zero. Because the label of an edge (u, v) corresponds to the number of direct Voronoi edges for u to reach v, this label can be used as an initial hop count for tuples generated by u and destined for v.

Consider for example Figure 5. Node u creates a tuple that is to be sent to neighbor k via core neighbor i. This tuple is given a hop count of $L[k]$. Similarly, the tuple u creates for l and sent via core neighbor m is given a hop count of $L[l]$. In addition, when u forwards tuples from l to i, it decrements the hop count in each of them by one. The same is true for tuples from k to m.

There are two main changes in the action that receives edges from neighbors. The first consists of strengthening routine *build-segment*, as shown in Figure 7. The segment is constructed one node at a time, as before. However, there might be some nodes in E that do not belong to the convex-hull, i.e., they are covered by the new nodes being added. These nodes are removed from E. Another change to *build-segment* is that a node is not added to E if in doing so the region-sanity predicate is violated. This ensures that once region-sanity holds (by the earlier action), it will continue to hold.

The second change to the action is to check the stack in $send[m][dir]$ for sanity, before making it available to neighbor m. This is done by the *filter* routine in Figure 8. This routine ensures that the hops remaining in the tuples of the stack are in non-increasing order towards the top of the stack. In addition, the labels have to be in strictly decreasing order towards the top of the stack. Finally, no label can be less than two.

C. Convergence

We show that regardless of the initial state of the system, the following predicate will hold permanently for every u,

$$R(u) = E^*$$

where E^* is the union of all nodes in any segment of u.

We define an *execution round* to be a subsequence of an execution in which every action of every node has either been executed or its guard is not enabled. A round captures the notion of taking enough execution steps guaranteeing that every node makes progress.

We begin with some observations. The core neighbors of node u cannot change unless there is a fault. Thus, after one round, the core sanity action ensures that $core$ is correct. Note that no other action affects this set, and hence, it continues to have the correct values (unless a fault occurs).

After one execution round, the region sanity action ensures that predicate *region-sanity* holds. However, routine *build-segment* affects array E, and thus it affects *region-sanity*. The removal of a node from E does not affect the truth value of *region-sanity*. Furthermore, when a node is added to E, *region-sanity* is checked. Thus, *region-sanity* continues to hold.

Note that the action to receive edges is parameterized. Thus, every segment in E is rebuilt at least once in each round. Because of this, every stack in array $send$ is also rebuilt. As it is rebuilt, routine *filter* ensures that $send$ has sanity values.

Finally, after an execution round, $L[i] = 1$ for every core neighbor i, and due to sanity on the send array, $L[v] \geq 2$ for every node in E.

1) Eliminating Non-Existing Nodes: If a node u is aware of all nodes in the network, simply taking the convex-hull of these nodes will suffice to compute its Voronoi region. Our objective is to have each node learn the least possible number of other nodes and thus minimize communication. However, due to faults, this communication may contain nodes that no longer exist due to failures, or simply values that have been corrupted due to communication errors. We next argue that such non-existing nodes will disappear from arrays E and $send$. We do so by induction on the label.

Basis: Within one round, any node v in a segment of E with $L[v] = 1$ is a real node, and all tuples in the $send$ array with a label value of one also correspond to real nodes.

As argued above, $L[v] \geq 2$ for any non-core node in E, and furthermore, $L[c] = 1$ for all core nodes c, and from the core-sanity, the core nodes are real-nodes (due to having direct communication with them) and immutable. From sanity of array $send$, no tuple has a label less than two.

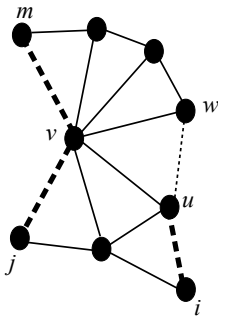

Figure 9. Stabilization induction step.

Induction Step: Assume now all nodes v in E with $L[v] \leq k$ are real nodes, and all tuples in array $send$ with *label* $\leq k$ are real nodes. We argue the same holds for $k+1$.

Consider first array $send$, and consider tuples with label $k+1$. In each round, *build-segment* completely rebuilds array $send$. Any tuples created with label $k+1$ must be real because the nodes being joined have labels at most k. Tuples added to array $send$ that are being forwarded from a neighbor have a hop count decreased by one. Thus, the maximum hop count of all tuples in the network that are not real nodes and have $k+1$ will decrease by one after each round. As this reaches zero, the tuples are eliminated by the *filter* routine. Thus, eventually all tuples with label $k+1$ correspond to real nodes.

For array E, once all tuples with label $k+1$ are real, the next time a segment is rebuilt, the nodes and labels come from the tuples of the send arrays. Hence, in one more round all nodes v in E with $L[v] = k+1$ are real nodes.

2) Constructing the Region: We next argue that within a bounded number of rounds $R(u) = E^*$. We assume that we have reached a state where all known nodes are real. We argue by induction on the label of edges in $R(u)$ that these edges are added to and preserved in E. Since the convex-hull of a set of nodes S does not change if we add to S nodes that are not in the convex-hull, then by the *region-sanity* predicate we will have that $R(u) = E^*$.

Recall that when a node processes the tuples from a neighbor, some are used to build the corresponding segment, and some are forwarded to the adjacent core node. The former are said to be *consumed* by the node. Note also that when forwarding tuples to a core neighbor, the first tuple is created by the node, and the remaining tuples are simply forwarded.

We use Figure 9 as reference for the induction, consisting of an arbitrary node u and neighboring core nodes i and m. Dashed lines correspond to core edges. The induction is based on the label of edges in $R(u)$, as follows.

a) For any node u and any core neighbor i, the sequence of nodes in $E[i][dir]$ with label at most l corresponds to the nodes in *segment*(u, i, dir) of $R(u)$ with label at most l.
b) If u has an edge of label l with a neighbor w, and u sends a tuple to core neighbor i of its segment containing w, and the tuple has a label of at least $l+1$, a hop count of at least l, and w as the destination, then the tuple reaches w and w consumes this tuple.
c) If u has an edge of label l with a neighbor v, and u sends a tuple to core neighbor i of its segment containing v, and the tuple has a label of at least $l+1$, a hop count of at least $l+1$, and a destination not equal to v, then v forwards the tuple to the core node of the adjacent segment to the one received.

We begin the induction with $l = 2$. Consider Figure 4, where x is the hinge of (u, w), and (u, w) has a label of two. Part a) requires that this edge will be added to the segment $u.E[x]$. Because u and w are core edges of x, from region-sanity, x is aware of them, and there can be no other nodes in x's segments $x.E[u]$ and $x.E[w]$. When x reads the tuples from w, it creates the tuple $(x, u, w, 2, 2)$ and sends it to u. Since this is the top tuple, it passes the send-sanity test at u, and u adds edge (u, w) to $u.E[x]$. Because (u, w) is in $R(u)$, no other node can block this edge, and the region sanity test is satisfied, making the edge permanent in $u.E[x]$.

For part b), assume u were to send a tuple to x with label $l > 2$ and hop count $h = 2$ and destined to w. Due to the label and hop count, the tuple passes the send sanity test at x. Node x receives it, and not being the destination forwards it to w. This tuple is below the tuple that x created to inform w of u. Node w processes the first tuple (adding u to its segment), and then processes the tuple from u.

Part c) is similar, except that the hop count is greater than two, and the destination is not w. In this case, w forwards it to the core neighbor opposite to x (not shown in the figure).

For the induction step, assume the statement holds for all values of l, $l \leq k$, and we show that it will also hold and continue to hold for label $l = k+1$.

Consider Figure 9, where *label*$(u, w) = k+1$, and *hinge*$(u, w) = v$. Let j and m be the roots of the segments of v containing u and w, respectively. From Theorem 2, all labels in these two segments of v are at most k. In addition, let i be root of the segment of u containing edge (u, v). Again, from Theorem 2, the labels in this segment are at most k. From the induction hypothesis and the labels being at most k, arrays E and L of nodes v and u permanently have the correct values for these three segments.

For part a), when v reads the tuples from m, it creates a tuple $(u, w, k+1, L[u])$ and sends it to k. From the induction hypothesis, this tuple is received and consumed at u adding the edge (u, w). Because this edge is in $R(u)$, it passes the region sanity test and no other node can displace this edge.

For part b), consider the case of u sending to i a tuple destined to w with label greater than $k+1$ and hop count of $k+1$. From part (b) of the induction hypothesis, and *label*$(u, v) \leq k$, this tuple is received at v (via j) and forwarded to core node m.

Note that, if the destination of this tuple were v rather than w, then, by the induction hypothesis, this tuple would have been consumed at v. This implies the tuple destined to w is at the top of the stack that v forwards to m. In particular, it is next to the top of the stack, which consists of the tuple created by v and sent to m to join edge (u, w). By the induction hypothesis, the tuple joining (u, w) is consumed at w. Recall that our tuple of interest is immediately below this tuple on the stack. Hence, since its destination is also w, this tuple will also be consumed at w, as desired.

For part c), the argument is similar, except that the label and hop count in the tuple that u sends to i are both greater than

 ISBN: 978-1-61208-753-5

$k+1$, and w is not the destination. Thus, when w processes the tuple, it forwards it to the core node adjacent to the core node from where the tuple is received, as desired.

VII. Conclusion and Future Work

We have developed a distributed algorithm that allows sensor nodes to learn their Voronoi region in a two-dimensional field. The algorithm is shown to be stabilizing, and thus, it is resilient to a wide variety of faults. It has the advantage of not assuming that there is an underlying routing protocol, and thus, there is no hidden communication cost. The low level of atomicity consists of only reading the variables of a single neighbor at a time. This is similar to receiving a message from the neighbor containing a copy of the neighbor's variables. We will extend the algorithm to the message passing model in future work.

Regarding communication overhead, each node receives a stack of tuples from each core neighbor. Each stack is of size at most $O(N)$ due to the requirement that labels decrease towards the top of the stack. Assuming that on average each node has g neighbors, then the overhead is $O(g \cdot N)$. It is known that if nodes are distributed in the plane according to a Poisson process with constant intensity, then each node in the DT has on average six surrounding triangles [16]. Thus, on average, the overhead is $O(N)$.

In general, a node u receives tuples from a source s to destination d if s and d are Voronoi neighbors and their VNP crosses u. If the area where the sensors are deployed is regular, as opposed to a long linear shape, then we expect the number of such pairs to be small, even of constant size. Thus, the overhead in most networks will be small, even smaller than $O(N)$. We will investigate this in future work, along with several variations of the problem such as obstacles that interfere with communication and having nodes with different transmission radius.

References

[1] J. Yick, B. Mukherjee, and D. Ghosal, "Wireless sensor network survey," Computer Networks, vol. 52, no. 12, 2008, pp. 2292 – 2330.

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

[3] B. Karp and H. T. Kung, "Gpsr: Greedy perimeter stateless routing for wireless networks," in Proc. of the 6th Annual International Conference on Mobile Computing and Networking, ser. MobiCom '00. New York, NY, USA: ACM, 2000, pp. 243–254.

[4] S. S. Lam and C. Qian, "Geographic routing in d-dimensional spaces with guaranteed delivery and low stretch," SIGMETRICS Perform. Eval. Rev., vol. 39, no. 1, Jun. 2011, pp. 217–228.

[5] B. Leong, B. Liskov, and R. Morris, "Geographic routing without planarization," in Proc. of the 3rd Conf. on Networked Systems Design & Implementation, ser. NSDI'06. Berkeley, CA, USA: USENIX Association, 2006, pp. 25–25.

[6] S. Fortune, Voronoi Diagrams and Delaunay Triangulations, second edition, J. E. Goodman and J. O'Rourke, Eds. CRC Press, 2004.

[7] P. Bose and P. Morin, "Online routing in triangulations," in Proc. of the 10th International Symposium on Algorithms and Computation, ser. ISAAC '99. London, UK: Springer-Verlag, 1999, pp. 113–122.

[8] M. Schneider, "Self-stabilization," ACM Computing Surveys, vol. 25, no. 1, Mar. 1993, pp. 45–67.

[9] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," Commun. ACM, vol. 17, no. 11, 1974, pp. 643–644.

[10] S. Dolev., Self-Stabilization. Cambridge, MA: MIT Press, 2000.

[11] M. G. Gouda, "The triumph and tribulation of system stabilization," in Proc. of the 9th International Workshop on Distributed Algorithms (WDAG). London, UK: Springer-Verlag, 1995, pp. 1–18.

[12] Y. Núnez-Rodrıguez, H. Xiao, K. Islam, and W. Alsalih, "A distributed algorithm for computing voronoi diagram in the unit disk graph model," in Proc. of the 20th Canadian Conference in Computational Geometry, Quebec, Canada, 2008, pp. 199–202.

[13] D. Y. Lee and S. S. Lam, "Protocol design for dynamic delaunay triangulation," in 27th International Conference on Distributed Computing Systems (ICDCS '07), June 2007, pp. 26–26.

[14] ——, "Efficient and accurate protocols for distributed delaunay triangulation under churn," in 2008 IEEE International Conference on Network Protocols, Oct 2008, pp. 124–136.

[15] R. Jacob, S. Ritscher, C. Scheideler, and S. Schmid, "A self-stabilizing and local delaunay graph construction," in Algorithms and Computation, Y. Dong, D.-Z. Du, and O. Ibarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 771–780.

[16] R. A. Dwyer, "Higher-dimensional voronoi diagrams in linear expected time," Discrete & Computational Geometry, vol. 6, no. 3, Sep 1991, pp. 343–367.

 ISBN: 978-1-61208-753-5