Stabilizing Voronoi Diagrams for Sensor Networks with Hidden Links

Jorge A. Cobb

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

***Abstract*—We present an efficient and self-stabilizing protocol for computing the Voronoi region of a sensor node in a large wireless sensor network deployed in the two dimensional plane. This protocol surpasses the preceding ones in that it is fully distributed, is self-stabilizing, and in particular, it moves away from the unit-disk transmission model. That is, the topology induced by the wireless communication links is assumed to be arbitrary. This naturally incorporates the practical case of obstacles interfering in the communication of some pairs of sensor nodes that are close to each other. Due to being self-stabilizing, the protocol converges to a normal operating state regardless of the initial value of its variables. Because faults can be modeled as having variable values that do not properly reflect the state of the network, the protocol is resilient against all types of transient faults, provided the network does not become partitioned.**

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

I. INTRODUCTION

Wireless sensor networks are characterized by having a limited number of resources. In particular, they operate on battery power, and have reduced processing and transmission capabilities. In order to preserve these critical resources, it is of paramount importance that every task performed by the sensors consumes the least amount of memory and energy [1].

Routing between nodes is a fundamental aspect of computer networks. Due to the limited resources in a wireless sensor network, it is desirable to use a routing protocol where the routing state stored at each node is independent of the network size; this is particularly important in a large-scale sensor network. One such approach is greedy routing [2]–[5]. 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.

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. Hence, in the particular context of network routing, Delaunay triangulations, and their dual, the Voronoi diagram, are well suited for greedy routing [6].

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 communication between nodes. The only assumption is that each node is initially only aware of those nodes with whom it can communicate in a single transmission hop. This is an extension to our earlier work [7]. In [7], we assumed the unit-disk transmission model. That is, there is a transmission radius r such that, if any pair of nodes are within a distance of r of each other, then they can communicate directly. In this paper, we relax this model, and assume that the network topology induced by the transmission links is arbitrary. This naturally incorporates the practical case of obstacles interfering in the communication of some pairs of sensor nodes that are close to each other.

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, Delanuay triangulations, and also our network model. In Section III, we present the local information that each node maintains about its Voronoi region, and how this information can be used to forward messages between distant Voronoi neighbors. In Section IV, we describe the adaptations that have to be made in order to deal with our general communication model. Section V presents the different types of messages used in the protocol and how they are used to route between nodes. Our protocol notation is presented in Section VI. The techniques that make the protocol stabilizing are reviewed in Section VII, and the specification of the protocol itself is given in Section VIII. We argue the correctness of the protocol in Section IX. Concluding remarks and possible future work are given in Section X.

 ISBN: 978-1-61208-828-0

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. This is similar to the review we presented in [7].

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.

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 *Delaunay 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. In [7], sensors are assumed to have the same transmission radius, and hence, if the distance between any pair of sensors is less than this radius, then the pair is able to directly exchange data messages. In this paper, we relax this assumption. We assume that obstacles could exist between nodes, and thus, nodes may not be able to communicate directly even though they are within transmission range. We thus assume a very general and arbitrary topology, in which the fact that a pair of nodes u and v can communicate directly is independent of whether u can communicate directly with another node w. This will have significant impacts on the algorithm, as discussed in later sections.

Nodes u and v are joined by a *physical link*, $\langle u, v \rangle$, if they can directly exchange messages. The set of nodes with whom u has a physical link is denoted by $L_{phy}(u)$. 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, $1 \leq i < k$, $w_{i+1} \in L_{phy}(w_i)$.

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 $V(u)$ be the set of neighbors of u in the DT. These are referred to as the *Voronoi neighbors* of u. For each v in $V(u)$, we refer to pair (u, v) as a *Voronoi edge*.

In Figure 1(iv), $V(x) = \{a, w, y\}$. Some of the nodes in $V(u)$ will be able to exchange messages directly with u, i.e., they are also contained in $L_{phy}(u)$. The nodes in $V(u) \cap L_{phy}(u)$ are said to be the *physical* Voronoi neighbors of u.

Note that it is possible for $w \in L_{phy}(u)$ 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.

Without obstacles in the network, such as the model we used in [7], between every pair of nodes there must exist a path such that each hop along the path consists of physical Voronoi neighbors. This allowed each node u to learn about its Voronoi neighbors by only exchanging messages with their physical Voronoi neighbors, i.e., nodes in $V(u) \cap L_{phy}(u)$. As we will show below, this is no longer the case if obstacles are present in the network.

III. Region Construction

We next describe the details of the information that a node maintains about its region, and how it provides assistance in building the regions of its neighbors.

A. Region Anatomy

Figure 2 depicts the region of u, consisting of eight Voronoi neighbors. Of these, neighbors i, m, and o are physical, i.e., they are contained in $L_{phy}(u)$. The set of physical Voronoi neighbors of a node u will be denoted by $core(u)$. The figure consists of only the region of u; the whole network is not shown.

Node u is aware of its core neighbors because it can communicate directly with them. On the other hand, consider node n. It must be that either o, m, or perhaps both, are able to communicate with n (recall that the network is connected), and inform u about n. Node u will remember which node informed it of the Voronoi edge (u, v). We will refer to this node as the *origin* of the edge.

In addition, each node keeps track of the number of transmission hops necessary to cross the link; this is known as the *label* of the link. E.g., if the origin of (u, n) is m, then $label(u, n) = label(m, n) + 1$. Both o and m report to u the expected number of hops to cross edge (u, v) through them. Node u chooses as origin the neighbor providing the least number of hops (a final tie-breaker is made using the node identifier). In the figure, o is the origin of (u, n), as indicated by the small arrow.

Consider now core nodes i and m, and the nodes in between them, j, k, and l. Node i is the origin of edge (u, j), and j is the origin of (u, k). We denote by *segment* the sequence of nodes starting at a core node where each node is the origin of the previous edge. The clockwise segment starting at i is (i, j, k). The counter-clockwise segment starting at m is (m, l), while the counter-clockwise segment starting at i is simply i itself.

Consider next nodes k and l. They are not aware of each other, and thus they do not report each other to u. In this case,

 ISBN: 978-1-61208-828-0

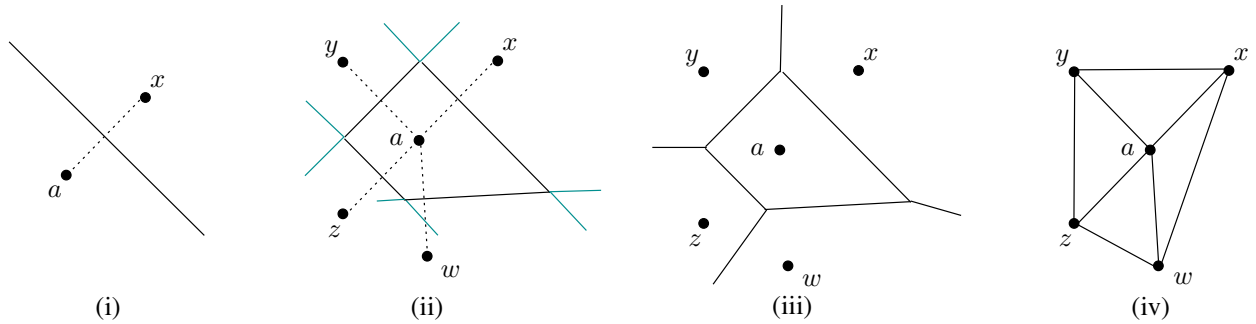

Figure 1. Voronoi diagram.

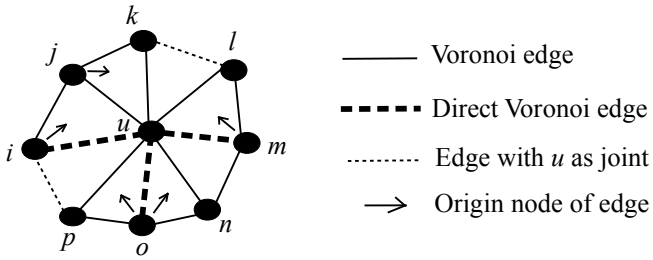

Figure 2. Segment construction.

u must introduce them to each other, and u becomes the origin of edge (k, l). Similarly, it introduces i and p to each other, and becomes the origin of edge (i, p).

We next describe the notion of a segment more formally [7]. 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*.

Let v be any Voronoi neighbor of u. Let *next*$(u, v,$ *dir*$)$ be the next node along direction *dir* on region $R(u)$. For example, in Figure 2, *next*$(u, i,$ *right*$) = j$, and *next*$(u, m,$ *left*$) = l$. Also, for any pair of neighbors v and w of u, let *bet*$(u, v, w,$ *dir*$)$ denote the sequence of nodes found in $R(u)$ along direction *dir* starting from v and ending in w.

Let *segment*$(u, v,$ *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, $v \in$ *core*(u), such that:

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

In addition, node w_j is denoted by *last*$(u, v,$ *dir*$)$.

B. Forwarding of Control Messages

In [7], we adopted the following general strategy. There is a single type of control message, namely *edge*, whose purpose is to inform a node that it has a Voronoi neighbor. Consider node u: it has to inform k and l of each other. To send an *edge* message to k, the destination field in the message is set to k, the direction is set to *right*, and the message is given to i. The message will then traverse *the entire segment* (reaching i, j, and k).

The reason the whole segment is traversed is the way in which nodes forward messages that are not addressed to them: the message is forwarded to the adjacent core node. For example, assume node u receives an *edge* message from m and the destination is not u. If the direction is *right*, it forwards the message to i, and if the direction is *left*, it forwards the message to o.

As another example, assume node l wants to send an *edge* message to k to inform it of a potential neighbor. Because u is the origin of (k, l), the message will arrive to u (via m). Node u is oblivious to the source, and it simply forwards it to the adjacent core node, i.e., i, and the message traverses the segment and arrives at k. Even further, the message may not even originate at l, it may simply need to traverse edge (l, k), and the source is not in the region of u. Nonetheless, since the message is received from m, it is forwarded to i, and the message arrives at k, as desired.

C. Obstacle Pitfalls

As discussed in Section II-B, we assume that, due to obstacles, the physical links form an arbitrary graph. Consider Figure 3(a), where the physical links are shown as dashed lines. Without obstacles, there would be a physical link between every pair of nodes. However, due to obstacles, the physical links $\langle t, v\rangle$, $\langle t, w\rangle$, and $\langle v, w\rangle$ are not present.

In Figure 3(b), the Voronoi edges computed are shown as solid lines. Node u is aware of its neighbors v and w, and thus it considers them part of its region, and sends an *edge* message to each of them, which makes them aware of each other. However, since t is not in the region of u, u ignores t. Thus, node t is not aware of v and w, and it thus cannot compute its own region. We address a mechanism to correct this below.

IV. Extended Links

In order to fix the problem described above, node t must be made aware that, if it were not for the obstacles, it would actually have a physical link with v and w. We can accomplish this by node u *extending* the link $\langle u, t\rangle$ to v and w. That is, to make v and w aware that they should have a link with t.

To become aware of this need, node u notices that its physical link $\langle u, t\rangle$ intersects its Voronoi edge (v, w). Thus, u notifies both v and t that they have an *extended link* between them. This link is shown as a gray dotted line in Figure 3 (c). Thus, v and t consider this link as an ordinary physical link, and add it to their set L of links. The extended link $\langle t, v\rangle$ is

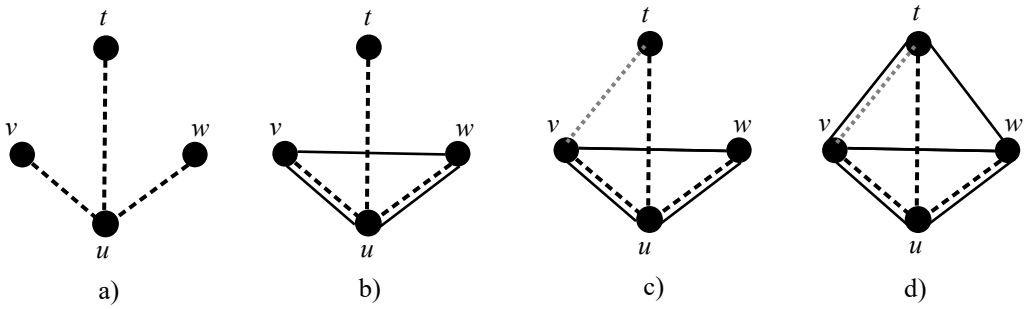

Figure 3. Extended links.

treated in the algorithm like any other link, e.g., it could be part of the core of t and the core of v. The difference only lies in that for t to send a message to v (and viceversa) the message has to be sent through u.

Once v and t are aware of each other, they add link $\langle t, v\rangle$ to their region. Node v then becomes the origin of edge (t, w), and it sends an *edge* message to both t and w making them aware of edge (t, w). The final results is shown in Figure 3(d).

Note that u does not need to extend the link $\langle u, t\rangle$ to w. Although possible, it is not necessary, since v will become the origin of the edge and join t with w. Thus, u will extend the link towards the neighbor that is closest of the two.

The extension of a link can become more complex, requiring the same link to be extended *multiple times*. This is illustrated in Figure 4. Part (a) of the figure indicates the physical links as dashed lines, and the solid lines correspond to the desired Voronoi edges. Without extending physical links, the Voronoi edges found are in part (b) of the figure. Part (c) shows the extension of edge $\langle p, t\rangle$ into edge $\langle q, t\rangle$. Node t disregards this edge since it is not part of its region (the Voronoi face with q is blocked by the faces of nodes s and u). Node q, however, will try to join t and r. This may be accepted by r, but t will reject it since edge (t, r) is not part of its region. The nodes will not learn the correct edges shown in part (a).

Note, however, that the extended link $\langle q, t\rangle$ crosses the Voronoi edge (s, u) of the region of t. Thus, t could extend it even further, by notifying s that it should have an extended link $\langle s, q\rangle$, as shown in part (d). Node s thus adds this link to its region, and joins q and u. This makes node q desist in attempting to join t with r, and instead joins u and r, yielding the correct result.

V. Message Types and Routing

We discussed above the need for extended links. In this section, we discuss the method for creating them and for sending messages across them. Our objective is to modify the method we presented in [7] the least possible, and still account for extended links. For example, consider again Figure 2, and assume node u receives a message from core node m, which is not destined for u. Thus, u forwards it to i. This should remain in effect even if the link $\langle i, u\rangle$ is an extended link, as opposed to a physical link. Thus, note that the core of a node may now consist of a mixture of physical links plus extended links.

Before discussing creating an extended link, we recall that in [7] we had only one message type, *edge*, to inform two nodes that there should be joined by a Voronoi edge. For example, in Figure 2, node u would send the following message to node k via link $\langle i, u\rangle$.

$$(ttl, \textit{edge}, dir, dst, src, nbr, lbl)$$

where dir = *right*, $dst = k$, $src = u$, $nbr = l$, and $lbl = \textit{label}(u, k) + \textit{label}(u, l)$. Also, the time to live, ttl, is the number of physical links traversed by the message. It is initially equal to one, and is incremented by one per physical link traversed. The message is discarded if it reaches an upper bound discussed in Section VII-A.

A. Creating an Extended Link

Consider now creating an extending a link, such as Figure 3(c), where node u wants to extend its physical link $\langle u, t\rangle$ to create the link $\langle v, t\rangle$. To do so, it sends a *link* message to both v and t, as follows:

$$(ttl, \textit{link}, dir, dst, src, nbr, lbl).$$

For v, $dst = v$, $src = u$, $nbr = t$, and $lbl = \textit{label}(u, v) + \textit{label}(u, t)$. Similarly for t, we have $dst = t$ and $nbr = v$.

In general, one of the endpoints (i.e., v) is a Voronoi neighbor, and the other (i.e., t) is a physical link. To send the message to the Voronoi neighbor, we simply send it as before, by sending it to the core neighbor of the segment containing the destination, and the message will traverse the whole segment until it is removed at the destination. In the case of v, v itself is the core node, and the message arrives in just one hop. For t, it is a physical link, so the message is sent directly.

A more interesting scenario occurs Figure 4(d), where node t is extending its link, $\langle t, q\rangle$ to create the link $\langle s, q\rangle$. In this case, neighbor s is a Voronoi neighbor, as expected, but node q is not a physical link, it is an *extended* link. Thus, the *link* message has to be routed in a manner that is different from routing around a segment of the region of node t. This is because extended links bypass Voronoi edges, in particular, link $\langle t, q\rangle$ crosses over Voronoi edges (s, u), (q, u), and (q, r). We discuss this next.

B. Transferring Messages Across an Extended Link

As mentioned at the beginning of the section, we need a general mechanism by which *any* message can be *tunneled* across an extended link. To send an arbitrary message over the extended link $\langle t, q\rangle$, node t must remember the *source* of

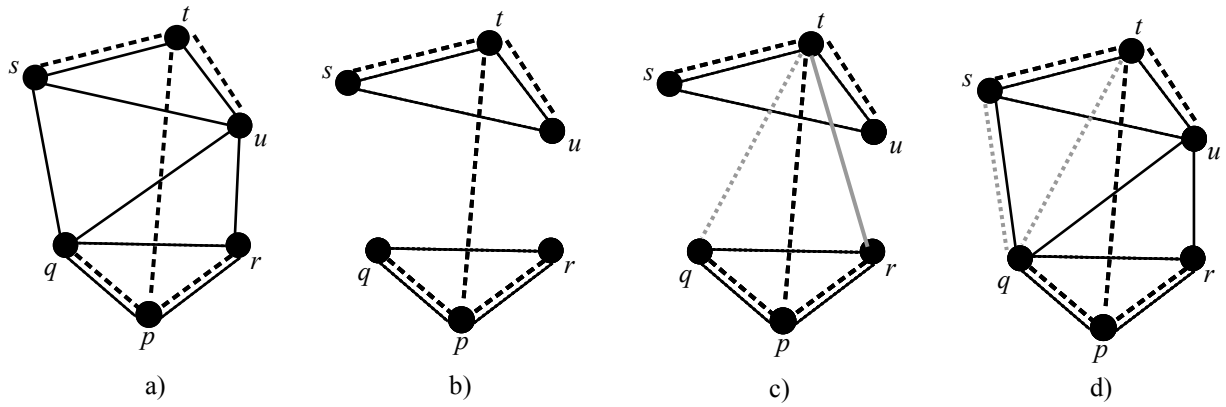

Figure 4. Link extended multiple times.

the link, in this case node p, and send the message to it. Node p cannot handle this message in the normal way, since the message is not being routed along Voronoi edges. We thus need a new message type to indicate to p that the message is being *thrown* across network, and that p should then find a way to forward it to the other end point q.

We introduce two additional message types, *throw* and *catch*, to tunnel the message across the extended link. For the specific case above, t sends the following message to p:

$$(ttl, \textit{throw}, dir, dst, src, nbr, (msg)),$$

where $dst = p$ (p should process and not just forward this message), $src = t$, and $nbr = q$. Node p then retrieves the encapsulated message msg, notices that it should be sent to q, and sends the following message to q:

$$(ttl, \textit{catch}, dir, dst, src, (msg))$$

where $dst = q$, and $src = t$, which allows q to learn that msg originated at t.

Note that, given that q is a Voronoi neighbor of p, p will use normal Voronoi routing to route the *catch* message. I.e., it will find the core node of the segment containing q, and forward the catch message to this node.

C. Generalized Send Operation

Sending a message across a link can become more complex, and it requires an iterative approach as follows. Consider Figure 4(d) again, and assume node q needs to route a message msg to node s along the extended link. To do so, it has to encapsulate msg in a *throw* message and send it to the source of the link, i.e., t. To send the message to t it has to encapsulate it in another *throw* message and send it to the source of that link, i.e., p. In the figure, p is a physical neighbor, so the message could be sent directly. However, it is easy to extend the network so that p is a Voronoi neighbor of q, but not a physical neighbor. This would then require q to find the core node of the segment of p, and then send the message to this core neighbor, etc..

VI. Protocol Notation

Before presenting the protocol in detail, we overview our 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 $L_{phys}(u)$. To maintain a low atomicity, and thus a possible transition to a message-passing model, 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 of the node, and also over the variables declared in a single node in $L_{phy}(u)$. 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

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.

In order to simplify the presentation and proofs, we assume a *shared memory model with low atomicity*. In particular, a

 ISBN: 978-1-61208-828-0

node is able to read the variables of its neighbors. However, in any action, it is only allowed to read the variables of a single neighbor. This will allow for an easy transformation into the message passing model.

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

The main inputs and variables of each node are as follows. Each node has a unique identifier from the set **ID**. Each node u receives as input the set L_{phys} consisting of the identifiers of all neighbors with whom it shares a physical link.

In order to represent the exchange of messages, each node has the following array.

$$send \quad : \quad \textbf{array}[L_{phy}] \textbf{ of set of } (\textbf{ID}, message) \qquad (*)$$

Element $send[v]$ contains the set of messages that node u wishes v to read. The purpose of the identifier attached to the message will be made clear in the next section. Each node u also has a variable, $rcvd$, as follows.

$$rcvd \quad : \quad \textbf{set of } message$$

To read the messages of a neighbor v, node u copies $v.send[u]$ into $u.rcvd$ (discarding the identifiers coupled with each message).

VII. 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.

In a stabilizing system, we assume each node can be in an arbitrary initial state, where its variables can have corrupt and misleading information. To ensure stabilization, the system must eliminate information which is not consistent with the current network state. Below, we describe how to eliminate non-existent nodes, stale edges, and stale messages.

A. Eliminating Ghost Nodes

Due to faults, information about nodes that have died may still exist in the data structures of a node, or perhaps a fault introduced information about non-existent nodes. We discuss below how this extraneous information is eliminated, provided nodes are aware of the number of nodes, N, that are in the system.

Obtaining N can be easily achieved by running a self-stabilizing leader election protocol, such as the one in [16], which builds a spanning tree in the network without knowledge of the network size. A simple diffusing computation allows the root to compute the number of nodes in the network, and report this value to all other nodes. This can be accomplished by periodically exchanging a constant-sized packet over each physical link.

Armed with the knowledge of N, no node will forward a message whose ttl is greater than N. Thus, any message in the network with a non-existing source will be removed from the network within N execution rounds.

Consider again Figure 2. Assume u receives an *edge* message from j joining k and u. If node j does not exist, as mentioned above, due to the ttl, messages from j will disappear and never be reintroduced. On the other hand, if j does exist, does k exist? Note that k may not be a physical neighbor of j, and hence, does not have immediate access to it. Nonetheless, the label assigned to (k, u) by j will be less than the label of (j, k), and as we approach closer to node k (by following the origin node of edge (j, k), which is not drawn in the figure), the labels continue to decrease. Since the smallest possible label is one, which is the case of a physical link, node k must exist.

B. Eliminating Stale Edges, Links, and Messages

Voronoi edges and extended links are created at a node after receiving a message of the appropriate type. If these messages cease to exist, then the corresponding edge/link is stale, and should be removed. Thus, we maintain the following array:

$$inc \quad : \quad \textbf{array}[L_{ext} \bigcup V_{join}] \textbf{ element of } \ L_{phy}$$

Entry $u.inc[v]$ stores the incoming physical link over which the message that created the edge/link (u, v) was received. Whenever u copies into $u.rcvd$ the messages from $v.send[u]$, all edges and links in $inc[v]$ are marked as stale. If an edge or link is not refreshed by any of the messages in $u.rcvd$, then it is discarded.

Similarly, as mentioned in Section III-B, node u forwards messages that are not destined for it, and places them in the appropriate entry in array $send$. Note that, as indicated in (*), each message is associated with an ID. This is the ID of the physical link over which the message was received. Thus, to prevent stale messages, whenever node u copies into $u.rcvd$ the messages from $v.send[u]$, it removes all messages in array $send$ whose ID is v.

VIII. Protocol Specification

We next present the specification of our protocol for an arbitrary node u. As discussed before, its inputs consists of the total number of nodes in the system, N, and the set of physical links, L_{phy}. It also contains several parameters to expand an action into multiple actions.

In terms of variables, L_{ext} contains the set of extended links. At all times, $L_{ext} \cap L_{phy} = \emptyset$. For conciseness, we define $L = L_{ext} \bigcup L_{phy}$. In addition, V contains the Voronoi neighbors found thus far.

We reuse the definitions related to a region $R(u)$, such as *segment*, *last*, and *next*, but instead based on the neighbors V found so far by node u, rather than the actual region $R(u)$ defined by the network topology.

Also, since node u is understood, we omit it from some definitions. E.g., $core$ is the set of core nodes of u, which we

define as $core = V \cap L$, and ($last(v, dir)$ is the last node of the segment of u starting at core node v in the direction *dir*. Finally, we define $V_{join} = V - core$.

The node contains three arrays not yet discussed. The first is *origin*, in which the origin of each link in L_{ext} and each edge V_{join} is stored. As discussed in Section VII, stale edges are marked in array *stale*. Finally, array *label* stores the label of each edge in V that is not a direct edge. By definition, $label[v] = 1$ for all v, $v \in L_{phy}$.

The complete specification is below, followed by a description of each action.

node u
inp
N : **integer** {number of nodes}
L_{phys} : **set of ID** {physical links}
param
t, t', i : **ID** {any node}
dir : **element of** *left* . . *right* {direction}
var
L_{ext} : **set of ID** {extended links}
V : **set of ID** {Voronoi neighbors}
inc : **array** $[L_{ext} \cup V_{join}]$ **element of** L_{phy}
{incoming physical link announcing the link/edge}
$origin$: **array** $[L_{ext} \cup V_{join}]$ **element of** V
{Voronoi neighbor announcing the link/edge}
$stale$: **array** $[L_{ext} \cup V_{join}]$ **of boolean**
{stale links/edges}
$label$: **array** $[L_{ext} \cup V_{join}]$ **of** $2 \; . \; . \; N$
{hops needed to traverse edge/link}
$rcvd$: **set of** *message*
$send$: **array** $[L_{phy}]$ **of set of**
(**element of** $L_{phy} \cup u$, *message*)
begin
{extend a link}
$t \in L \wedge t \notin V \wedge outside(t, V) \;\; \rightarrow$
$(q, r) := closest(t, V)$;
$dir := direction(core(q), q)$;
$msg := (1, link, dir, q, u, t,$
$max(label[q], label[r]) + label[t])$;
$sendmsg(u, core(q), msg)$

{join an edge}
$t \in core \wedge t' = next\text{-}core(t, dir) \;\; \rightarrow$
$x := last(t, dir)$;
$y := last(t', \neg dir)$;
$msg := (1, edge, dir, x, u, label[x] + label[y])$;
$sendmsg(u, t, msg)$;
$msg := (1, edge, \neg dir, y, u, label[x] + label[y])$;
$sendmsg(u, t', msg)$

{receive a message}
$i \in L_{phy} \;\; \rightarrow$
$rcvd := i.send[u]$;
$clear(send, i)$;
for each $v \in (L_{ext} \cup V_{join})$ **do**
if $inc[v] = i$ **then**
$stale[v] := true$;
for each $msg \in rcvd$ **do**
$process\text{-}msg(msg)$;
for each $v \in (L_{ext} \cup V_{join})$ **do**
if $stale[v]$ **then**
$V := V - \{v\}$;
$L_{ext} := L_{ext} - \{v\}$;
$clean(V)$;
end

In the first action, a link to a node t is extended via a Voronoi neighbor q. Node t should be outside of the region defined by V. If so, the Voronoi edge in V, namely (q, r), crossed by t, such that t is closer to q than r, is found, and a *link* message is sent to the core node of the segment containing q. Note that the label in the message is $max(label[q], label[r]) + label[t]$ rather than simply $label[q] + label[t]$. The reason for this will be made clear in Section IX.

In the second action, the nodes at the end of two segments are joined together by sending an *edge* message to them. The core nodes are t and t', without any core nodes between them. An *edge* message is sent to the two endpoints of the corresponding segments to indicate to them that they are potential neighbors. Function *next-core* is defined as follows:

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

Also, the *sendmsg* routine, as described in Section V-C, is as follows.

$sendmsg(in, out, msg)$
$h := msg.ttl + 1$;
if $h > N$ **then** *return*;
if $out \in L_{phys}$ **then**
$send[out] := send[out] \cup (in, msg)$
if $out \in L_{ext}$ **then**
$or := origin[out]$;
$out' := core(or)$;
$dir := direction(out', or)$;
$msg' := (h, throw, dir, or, u, out', msg)$;
$send(in, out', msg')$

In the third action, a message is received from a physical link, by copying the appropriate contents of the $send$ array of physical neighbor i. Then, the $send$ array is cleared of all earlier messages that were received from neighbor i, and every edge and link whose message that created them was received along link i are marked stale. If these edges and links are refreshed by messages from i, then their staleness is removed at the end of the action. In addition, routine $clean(V)$ removes from V edges that are inconsistent with each other. It's objective is to remove all nodes in V that violate the following *clean* consistency requirement between successive nodes of a segment:

$\langle \forall x, dir, v, w,$
$(x \in core \wedge w \in segment(x, dir) \wedge v \in segment(x, dir) \wedge$
$v \neq w \wedge w = next(v, dir))$
$\Rightarrow$
$(label[w] > label[v] \wedge origin[w] = v)$
$\rangle$

That is, every neighbor should have as origin the previous neighbor on its segment, and furthermore, its label should be also greater than that of the previous neighbor. Routine $clean(V)$ is as follows.

```
clean(V)
    V := convex-hull(V ∪ L_phy);
    clean := core;
    for each i ∈ L_phy do
      label[i] := 1;
    for each i ∈ core do
      for dir ∈ {left, right}
        v := i;
        while (next(v, dir) ≠ nil, ∧
               origin(next(v, dir)) = v ∧
               label(v) < label(next(v, dir))) do
            clean := clean ∪ {v};
    V := clean
```

Each received message is processed according to its message type. The code for routine *process-msg* is as follows.

```
process-msg(msg)
    if msg.dst ≠ u ∧ i ∈ V ∧ msg.ttl < N then
        dir := msg.dir;
        sendmsg(i, next-core(i, dir)), dir);
    if msg.dst = u ∧ msg.ttl < N
        if msg.type = edge then
            process-edge-msg;
        if msg.type = link then
            process-link-msg;
        if msg.type = throw then
            process-throw-msg;
        if msg.type = catch then
            process-catch-msg
```

We next discuss the processing of each message type. The processing of an *edge* message is as follows.

```
process-edge-msg
    nbr := msg.nbr;
    src := msg.src;
    l := msg.lbl;
    if src ∈ V ∧ nbr ∉ L_phy ∧
      nbr ∈ convex-hull(V ∪ {nbr}) ∧
      (nbr ∉ V ∨ l < label[nbr] ∨
      (l = label[nbr] ∧ src < origin[nbr])) then
        V := V ∪ nbr;
        origin[nbr] := src;
        inc[nbr] := i;
        label[nbr] := l;
        stale[nbr] := false
```

An edge is added to V under certain conditions. First, the origin of the edge, i.e., the source of the message, must already be in V. Next, it should not be a physical neighbor because these neighbors are always present and considered for V. Also, the node will only be added to V if it takes part in the convex-hull, i.e., the region, of the node. Finally, either the node is a new addition to V, or it provides a better label than before (ties broken in favor of smaller origin ID).

The processing of a *link* message is as follows.

```
process-link-msg
    ngh := msg.ngh;
    src := msg.src;
    or := origin[ngh];
    if src ∈ V ∧
      (ngh ∉ L_ext ∨ label[src] < label[or] ∨
       (label[or] = label[src] ∧ src < or)
    then
        L_ext = L_ext ∪ {ngh};
        origin[ngh] := src;
        inc[ngh] := i;
        stale[ngh] := false
```

In order to add a node as an extended link, the origin of the link (i.e., the source of the message), must already be a Voronoi neighbor. Also, either it is a new extended link, or the origin of the link has a better label than the origin of the current extended link. If so, the node is added to the extended link set and the book-keeping arrays are updated accordingly.

The processing of a *throw* message is as follows.

```
process-throw-msg
    m'.type := catch;
    m'.ttl := msg.ttl;
    m'.msg := decap(msg);
    dst := msg.ngh;
    src := msg.src;
    if dst ∈ V_join ∧ src ∈ L_ext then
        msg.dir := direction(core(dst), dst);
        sendmsg(i, core(dst), m')
    if dst ∈ L_ext ∧ src ∈ V_join then
        dir := nil;
        sendmsg(i, dst, m')
```

The original message is retrieved from decapsulation of the catch message. If the destination of the message, i.e., the origin of the extended link, is a Voronoi neighbor, then the core node associated with this Voronoi neighbor is found, and the message is sent to it. On the other hand, if the destination of the message is the endpoint of an extended link, then the message is sent directly over the link.

The processing of a *catch* message is as follows. The original message is simply retrieved from the *catch* message, and then processed like a normal message.

```
process-catch-msg
    msg' := decap(msg);
    msg'.ttl := msg.ttl;
    process-msg(msg')
```

IX. CORRECTNESS

We show that regardless of the initial state of the system, $R(u) = V$ will hold permanently for every 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.

Note that after executing the third action to receive messages, the clean requirement on V is satisfied due to the *clean*(V) routine. We thus assume that this always holds in between action executions.

A. Eliminating Non-Existing Nodes

If there is a message msg in a send queue that is paired with a neighbor v in L_{phys}, then, the next time node u reads messages from v, msg is removed from the send queue. I.e., after each execution round, all messages disappear unless being received again from the corresponding neighbor.

Consider first all messages whose src value is a non-existent node. No new messages with a non-existing node as a src field can be created, because the value of src is set to the node creating the message, i.e., a valid live node. Any of these messages with $ttl = N - x$ will disappear within x execution rounds, because in a round all existing messages are deleted, and the forwarded messages have their ttl increased by one, and are discarded when it reaches N.

We next argue that all messages with a non-existent ngh disappear. We do so in conjunction with showing that these nodes disappear from $V_{join} \cup L_{ext}$. We do a combined induction over the label values and the ttl.

Consider first messages with $lbl = 1$. No new message can be created with this label since new messages have a label equal to the sum of two other labels, all of which are at least 1 (i.e., L_{phy} neighbors). Thus, by induction on the ttl, these messages will disappear. Note that $label[v]$ is defined to be at least 2, and thus any non-existent neighbor in $V_{join} \cup L_{ext}$ must have a label of at least 2.

Consider next that all messages with a non-existent ngh have a label of at least x, permanently, and all nodes in $V_{join} \cup L_{ext}$ have a label at least x, permanently. Similarly, any new message with label x is obtained by adding the labels of two nodes. Since non-existing nodes have a label of at least x, no such message can be created. Hence, by a simple induction on the ttl, all messages with non-existent ngh and a label of x will disappear permanently. Also, in the next execution round, edges and links in $V_{join} \cup L_{ext}$ with a non-existent ngh and a label of x will be marked as stale and not be refreshed, and thus removed. Hence, all messages with a non-existent ngh will have a label of at least $x + 1$, permanently, and all nodes in $V_{join} \cup L_{ext}$ will have a label at least $x + 1$, permanently. The desired result follows by induction.

B. Constructing $R(u)$

Due to lack of space, we present an overview of the proof. We begin by observing that if a node v is in $R(u)$, and if it is also in V, then $v \in$ *convex-hull*(V). That is, no existing node can block v. Since non-existing nodes have been shown to disappear, v cannot be blocked by any other node. Thus, the only reason v could be removed from V is if v does not satisfy the clean condition that is required of V. We will show by induction that this is not the case.

Also, similar to the first argument in Section IX-A, an induction argument can show that if a message is not recreated at its source node, then, due to the ttl, all copies of the message will be removed from the network. This observation will be used throughout.

We define the notion of the *DTlabel* of an edge (u, v) in the DT of the network in a similar way as was done in [7]. Our induction will be over the *DTlabel*. The *DTlabel* of edge (u, v) is the smallest positive integer such that:

(i) If (u, v) is a direct physical link between u and v, then *DTlabel*$(u, v) = 1$.

(ii) If (u, v) is not a physical or an extended link, then note that (u, v) can be involved it at most two triangles in the DT. Let those triangles be (x, u, v) and (y, u, v), i.e., either x or y is the origin of edge (u, v) ($x = y$ in the case of only one triangle). Then, *DTlabel*$(u, v) =$ *min*(*label*(x, u) + *label*(x, v), *label*(y, u) + *label*(y, v)).

(iii) If (u, v) is an extended link formed by extending another link (p, v) that crosses an edge (u, w) in $R(p)$, then *DTlabel*(u, v) = *max*(*label*(p, u), *label*(p, w)) + *label*(p, v).

The reason for (iii) above being *max*(*label*(p, u), *label*(p, w)) rather than simply *label*(p, u) is that our induction will be based on *DTlabel*, and *both* edges/links (p, u) and (p, w) must exist and be stable before the link (u, v) can be created (or considered stable).

We require one additional finding. By following the routing along Voronoi edges as described in Section III-B, if u sends a message to a neighbor v, $v \in V$, then the message will only traverse along edges that have a *DTlabel* value smaller than that of (u, v), as shown earlier in [7].

We need to show by induction that, for all h, $1 \leq h \leq N$,

(i) For every message of type *edge* or *link* with label h, the message will arrive at its destination and always be available at the destination.

(ii) For every message of type *catch* or *throw* that encapsulates a message with label h or less, the message will arrive at its destination.

(iii) For every node u and every v, $v \in u.V$, if $u.label[v] \leq h$, then *DTlabel*$(u, v) = u.label[v]$, and $v \in R(u)$.

(iv) For every node u and every v, $v \in R(u)$, if $DTlabel(u, v) \leq h$, then $u.label[v] =$ *DTlabel*(u, v), and $v \in V$.

(v) For each extended link (u, v), if *DTlabel*$(u, v) \leq h$, then $v \in u.L_{ext}$, $u.label[v] =$ *DTlabel*(u, v), and $u.origin[v] =$ *origin*(u, v).

(vi) For every node u and every node v, $v \in u.L_{ext}$, if $u.label[v] \leq h$, then *DTlabel*$(u, v) = u.label[v]$ and $u.origin[v] =$ *origin*(u, v).

Consider first $h = 1$. Extended links and non-direct Voronoi edges have a $DTlabel$ greater than one. Only direct physical links can have a label of one, which is hard-coded in the protocol. Also, routine $clean(V)$ ensures that the physical links in $R(u)$ are included in V. The origin value of a physical link is *nil* since it does not depend on other links. Furthermore, any message created has as label the sum of two other labels, and thus, its label is at least two. Any existing messages that have a label of one will not be recreated by their sources, and thus, by the ttl, they will be permanently removed.

Assume the induction hypothesis is correct for all labels at most h. We now consider labels at most $h + 1$. Consider first a message with label $h + 1$ whose (src, dst, nbr) do not correspond to a triangle in the DT. In order for a message of label $h+1$ to be recreated, both edges (src, dst) and (src, nbr) must have a label at node src of at most h. But, by the induction hypothesis, these would correspond to real edges in the DT, and hence, this message cannot be recreated. By the ttl, the message will disappear permanently from the network.

Consider now a Voronoi edge (u, v) in the DT with *DTlabel*$(u, v) = h + 1$ and origin x. This implies

DTlabel$(x, u) \leq h$ and *DTlabel*$(x, v) \leq h$. By the induction hypothesis, u and v are in $x.V$ with the correct label and origin, and also in $R(x)$. Thus, no node can be in between u and v in $x.V$, and in consequence, x will send an *edge* message of label $h + 1$ to both u and v. From the induction hypothesis, these messages will be delivered to u and v. Since edge (u, v) belongs to both $R(u)$ and $R(v)$, no node can block the edge from being added to $u.V$ and $v.V$, as desired.

The argument that extended links will be created from *link* messages follows a similar argument. Likewise, if a *throw* or *catch* message contains a message with label $h + 1$, it implies that both the link and the edge that join the extended link have labels at most h, and thus, the contained message will be delivered correctly.

X. Conclusion and Future Work

We presented an efficient and self-stabilizing protocol for computing the Voronoi region of a sensor node in a large wireless sensor network. In particular, it departs from the unit-circle communication model. Because faults can be modeled as making arbitrary changes to the network state, any self-stabilizing protocol is resilient against all type of transient faults, provided the network does not become partitioned. There is no assumption of having an underlying routing protocol to aid in routing control messages. Thus, the protocol can successfully communicate with any Voronoi neighbor without any additional aid, and can be used as the foundation for a geographic routing protocol.

If nodes are distributed in the plane according to a Poisson process with constant intensity, then each node in the DT of these nodes has on average six neighbors [17]. In general, a node u receives messages from a source s to destination d if s and d are Voronoi neighbors and their Voronoi path crosses u. Given the small number of surrounding neighbors, if the deployment area 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 $O(N)$. In our future work, we will perform simulations over randomly generated networks to measure the worst and average node overhead over various topologies and node densities.

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] 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.

[7] J. A. Cobb, "Stabilizing voronoi diagrams for sensor networks," in Proc. of The 14th Int. Conf. on Systems and Networks Communications, Valencia, Spain, 2019, pp. 7 – 13.

[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. Nnez-Rodrguez, 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] J. A. Cobb, "Preserving routes during fast convergence," in Proc. of the 13th IEEE International Workshop on Assurance in Distributed Systems and Networks (ADSN), Madrid, Spain, June 2014, pp. 125–132.

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

 ISBN: 978-1-61208-828-0