Self-Stabilizing Routing Protocol for General Networks

Doina Bein  
School of Computer Science,  
University of Nevada Las Vegas,  
USA  
siona@cs.unlv.edu

Ajoy K. Datta  
School of Computer Science,  
University of Nevada Las Vegas,  
USA  
datta@cs.unlv.edu

Vincent Villain  
LaRIA Université de Picardie  
Jules Verne, France  
villain@laria.u-picardie.fr

Abstract

Given an asynchronous network with at most \( n \) nodes we present a self-stabilizing distributed algorithm for routing (nodes can be added or can crash at any time, so their number can vary up to the upper bound \( n \)). It starts in some arbitrary state, with no knowledge of the network architecture and eventually builds in each node a correct routing table regarding the \( t \) closest neighbors (\( t \) depends on the network needs: it can be \( n \) when each node needs to know shortest paths to all other nodes, or less when we need only partial knowledge). The size of the table in any node \( v \) is \( O((t + \Delta_v) \log(n)) \) bits (\( \Delta_v \) is the degree of node \( v \)), and a total of \( O(n \ t \log(n)) \) bits per network. The stabilization time of the algorithm is \( O(d+c) \) time units (\( d \) is the maximum diameter of the network, and \( c \) is a large constant depending on the local computation time of a node).

Keywords: Asynchronous network, distributed algorithm, fault tolerance, routing, self-stabilization.

1. Introduction

Routing schemes implemented in point-to-point communication networks deliver messages between nodes ([1,2]). Each node maintains a routing table and it is important to update the routing scheme dynamically in case of network change (nodes can be added or conceivably crash.) When topology changes occur frequently in the network, a cold restart can become every expensive in terms of time and resources.

BGP (Border Gateway Protocol) provides the routing protocol that supports the Internet backbone. BGP servers must maintain routing tables that include all of the external addresses on the Internet! Routers use BGP to communicate with their intermediate neighbors to exchange their "routing tables" in order to inform each other about which IP ranges the router can forward.

The most general technique of designing a system to tolerate arbitrary transient faults is self-stabilization ([3]). A self-stabilizing system is guaranteed to converge to the intended behavior in finite time, regardless of the initial state of the nodes and initial messages on the links. In a distributed self-stabilizing routing algorithm, a node, with no initialization code and having only local information, has to achieve a global objective, to build a correct routing table with limited information regarding routing toward its closest nodes.

1.1 Related Work

It is known ([4,5,6]) that the memory requirements of a routing scheme are related to the worst case stretch factor the routing scheme guarantees. Peleg ([6]) showed that any universal routing strategy that can achieve a stretch factor \( s \geq 1 \) must use a total of \( \Omega(n^{1+1/s}) \) bits of routing information in the network.

Several routing strategies have been proposed which achieve an almost optimal efficiency-space relation. Specifically, Peleg ([6]) proved that for every graph and every integer \( k \geq 1 \) it is possible to construct a hierarchical routing scheme with stretch factor \( O(k) \) which uses a total of \( \Omega(k^{3}n^{1+1/k} \log n) \) bits and labels each node with \( O(\log^{2} n) \) bits. The scheme has a few drawbacks: it is not name-independent (it relabels the nodes with new names), it does not bound the local memory requirement of a node, and finally, it assumes a unit cost on the links of the network. Other hierarchical routing methods ([4,5]) avoid these problems but at the price of non-optimal efficiency-space. But the major disadvantage of all proposed hierarchical routing strategies is a complex decision function at the nodes, which becomes a bottleneck in the case of high-speed networks.

In 1973, Dijkstra introduced the notion of self-stabilization in the context of distributed systems ([3,7]). He defined a system to be self-stabilizing when, "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps". A system, which is not self-stabilizing, may stay in an illegitimate state forever.

Fault-tolerance is an important issue in designing network routing protocols since the topology changes due to the link/node failure or recovery. Self-stabilizing topology-update problems are discussed in [8,9].
1.2 Contributions

In this paper we propose a fault-tolerant distributed algorithm that can work in a general asynchronous network. The algorithm starts with no knowledge of the network architecture and progressively builds a correct routing table with information regarding the closest nodes that can be used further for different types of routing (hierarchical, compact, interval etc).

It supports fault causing nodes and link failures and additions of nodes and/or links, and it is guaranteed that it will reach a correct state in finite time (it is self-stabilizing). We assume that the maximum number of nodes in the network is \( n \) (nodes can be added or can crash at any time, so the number of nodes can vary but \( n \) is the upper bound).

The tasks are fairly distributed among the nodes and each node builds its own routing table based on the information gathered online up to the current moment. So for a node \( v \), the routing table size is \( O(n + \Delta \log(n)) \) bits (\( \Delta \) is the degree of node \( v \)). The value chosen for \( t \) depends on the network needs: it can be \( n \) when the node needs to know shortest paths to all other nodes, or less when we need only partial knowledge). The total amount of information stored in all the nodes in the graph is \( O(n t \log(n)) \). The stabilization time of the algorithm is \( O(d+c) \) time units, where \( d \) is the maximum diameter of the network, and \( c \) is a large constant depending on the local computation time of a node.

1.3 Outline of the Paper

We start section 2 by giving general definitions regarding distributed systems and self-stabilization, and continue with our main contribution, the self-stabilizing distributed routing algorithm. We then prove the correctness of the algorithm in Section 3 and we give some concluding remarks in section 4.

2. Self-Stabilizing Distributed Routing Algorithm (SRS)

In this section we define the self-stabilizing routing algorithm. We present some general notions, and continue with the self-stabilizing algorithm.

2.1 Definitions

Distributed systems are a class of multiprocessor systems, where the nodes have own memory. The nodes communicate by messages, with two actions: send(message) and receive(message). If nodes \( p \) and \( p \) need to communicate, they must send/receive messages from each other; a communication link (bi-directional channel) must exist between them. Messages sent by a node can be either fixed or variable size. Our algorithm is asynchronous, which means that is guaranteed to run correctly in networks with arbitrary timing guarantees. A very common assumption is to bound the interval of time for transmitting a message, called timeout, after which the message is considered lost.

Each node starts with a unique ID and initially knows only its direct neighbors. Edges are labeled by distance values. Every node \( p \) can distinguish its entire links. The variable \( N_p \) refers to the set of the direct neighbors of \( p \), arranged in some arbitrary order \( \prec_p \). The number of neighbors of \( p \), \( |N_p| \), is called the degree of \( p \) and is denoted by \( \Delta_p \). We assume that \( N_p \) is maintained by an underlying local topology maintenance protocol that it can alter its values in case of changes in the network (failures of nodes, or links, or both.)

We can order all the other nodes with respect to the distance relation and choose the set \( t \)-ball \( B_v(t) \) as the first \( t \) nodes according to the node ascending ordering \( ([10]) \). The \( t \)-ball defines the closer nodes, and does not always contain all the neighbors of the current node.

We cannot bound the moment of time when a message can be received (since the system is asynchronous), and we cannot wait forever to receive all the messages sent by other nodes in order to construct a correct \( t \)-ball. So we relax the definition of \( t \)-ball to fit to an asynchronous algorithm:

Definition 1 A partial \( t \)-ball for a node \( v \), \( B_v(t) \), is a set of \( t \) nodes, with the length of the path toward \( v \) within the lowest values received by the node until a certain condition becomes true.

In the description of the algorithm, we use the word \( t \)-ball instead of partial \( t \)-ball, by a slight abuse in notation.

The program consists of a set of global variables and a finite set of actions. Each action is uniquely identified by a label and is part of a guarded command: \( <label> :: <guard> \rightarrow <action> \)

The guard of an action is a Boolean expression involving the global variables and/or local variables. The action can be executed only if its guard evaluates to true. We assume that the actions are atomically executed: the evaluation of a guard and the execution of the corresponding action, if it is selected for execution, are done in one atomic step.

In the system, one or more nodes execute an action and a node may take at most one action. This execution model is known as the distributed daemon. We assume a weakly fair daemon, meaning that if node \( p \) is continuously enabled, \( p \) will be eventually chosen by the distributed daemon to execute an action. A network protocol is a set of node programs, one for each node.

Each component of a system (node or link) has a local state, which is the ID of the node and the values of the program variables. We define the global state of a system as the union of the local state of its components as well as the messages on links.

A self-stabilizing system \( S \) guarantees that, starting from an arbitrary global state, it reaches a legal global state within a finite number of state
transitions, and remains in a legal state unless a change occurs. In a non-self-stabilizing system, the system designer needs to enumerate the accepted kinds of faults, such as node/link failures, and he must add special mechanisms for recovery. Generally, not all types of faults are taken in consideration, and an obscure error such as a memory corruption can provoke a general reset of the entire system. Ideally, a system should continue its work by correctly restoring the state of the system whenever a fault occurs ([11],[2]).

Let \( X \) be a set. \( x \mapsto Q \) means that an element \( x \in X \) satisfies the predicate \( Q \) defined on the set \( X \). We define a special predicate \( \text{true} \) as follows:

for any \( x \in X \), \( x \mapsto \text{true} \).

Let \( P \) be a distributed system and \( R \) and \( S \) predicates on the states of \( P \). \( R \) is closed if every state of the computation of \( P \) that starts in a state satisfying \( P \) also satisfies \( R \). \( R \) converges to \( S \) in \( P \) if \( R \) is closed in \( P \), \( S \) is closed in \( P \), and any computation starting from a state satisfying \( R \) contains a state satisfying \( S \).

**Definition 2** \( P \) stabilizes to \( R \) iff \( \text{true} \) converges to \( R \) in \( P \).

### 2.2 Routing Algorithm

The purpose of the algorithm is to construct a correct routing table in each node with information regarding routing to the closest nodes in the network. For each node \( v \) it selects in the \( t \)-ball \( B \), the \( t \) closest nodes (\( t \) is decided in advance), and also it considers all the direct neighbors of \( v \). The routing table called \( H \) will contain at most \( t + \Delta \) entries.

Each node \( v \) maintains several global variables of different types. The underlying layer of topological maintenance protocol computes the variable \( N_v \), the set of the neighbors’ IDs of the node \( v \) (set of integers). The others are calculated and used by the layers of the algorithm:

- \( B \) = the list of nodes IDs situated in the \( t \)-ball \( B \), of the node \( v \)
- \( \text{updated} = \text{true} \) when the \( t \)-ball is updated (Boolean)
- \( \text{RxId}_\text{ds} \) = the set of IDs of other nodes known by the current node
- \( H \) = a linked list with information regarding the nodes from \( B \). An element has three fields:
  - \( \text{dest} \) = destination ID (integer)
  - \( \text{neighbor} \) = the neighbor which is the first node on the path to \( \text{dest} \) (integer)
  - \( \text{distance} \) = the distance from \( \text{dest} \) or 0 (int)
  - \( \text{direct} = \text{true} \) if \( \text{dest} \) is a direct neighbor and the direct link is the shortest (Boolean)

The list \( H \) is maintained in ascending order of the distance value and has several functions that help us to retrieve information from it:

- \( \text{Give}_\text{Ids}(H) \) = returns all the IDs (field \( id \)) in \( H \), or \( \text{null} \) if \( H \) is \( \emptyset \)
- \( G(H, id) \) = returns the element of \( H \) with the given \( id \) if it exists, or \( \text{null} \) otherwise

We consider the following notations:

- \( v \in H \) means \( v \in \text{Give}_\text{Ids}(H) \) (e.g. \( H \subseteq B \) means \( \text{Give}_\text{Ids}(H) \subseteq B \))
- \( H[id] \) means \( G(H,u) \) (e.g. \( H[u].\text{neighbor} \) means \( (G(H,u)).\text{neighbor} \)).

There are other functions that we use:

- \( \text{Remove}_\text{Id}(id, H, B) \) = remove the element with given \( id \) from \( B \) and \( H \)
- \( \text{Maximum}_\text{Distance}(B, H) \) = selects the \( id \) of the node which is in \( B \) that has the maximum distance and also is not a direct neighbor
- \( \text{NewCell}(B, H, \ldots) \) = creates a new cell for \( id \) in \( H \) with the fields’s values specified, and adds \( id \) to \( B \)

The general algorithm has two layers:

**Algorithm 2.1 SRS Self-Stabilizing Routing Scheme**

| A.01 | Error Correction |
| A.02 | Calculate Ball |

### 2.3 Error_Correction

**Error_Correction** has the role to broadcast periodically (a \( \text{timeout} \) is given) a message \( \text{DIST} \) with the node ID and the distance to that node, initially 0, to all its neighbors. These messages will be forwarded to other nodes, if they satisfy some distance criteria (such that the network doesn’t get flooded) to help them calculate the distance to the current node. Eventually discrepancies in the global variables are detected and then the entire construction of the SRS scheme must start from scratch. In this case, all the global variables are reset to \( \text{null} \) or \( \text{false} \), in order to start a fresh phase.

**Algorithm 2.2 Error_Correction**

**Messages**

- \( \text{DIST} \) sender: the ID of the sender
dist: the length of the path the message went through

- \( \text{LOST} \) id: the sender ID

**Local variables**

- \( id, nb, nhr: \text{int} \)

**Predicate**

\( \text{error} = (B \setminus H \neq \emptyset) \)

**Macro**

\( \text{RESTART} = \text{reset} \text{Calculate Ball} \) and set the global variables to their default values

**Actions:**

1.01 \( \text{timeout} \) \( \longrightarrow \)

/* \( v \) broadcasts the message \( \text{DIST} \) */

1.02 \( \text{SEND \text{DIST}(ID,0)} \) TO all \( nb \in N_v \)

1.03 \( \text{error} \) \( \longrightarrow \) \( \text{RESTART} \)

1.04 \( \exists \ id \in H \colon H[id].\text{direct} = \text{true} \land id \in N_v \land \text{length(link to id)} \neq H[id].\text{distance} \)

1.05 \( \text{RESTART} \)
2.4 Calculate_Ball

The algorithm Calculate_Ball gathers data about the neighborhood. The DIST messages from the other nodes are processed and the first \( t \) lowest distances, breaking ties by increasing node ID, are stored in the data structure.

\( H \) together with the node IDs (stored also in the set \( B \)). So far, \( B \), is computed in the set \( B \) and the corresponding links are labeled dynamically with the IDs from the set \( B \).

The set \( B \) should contain \( t \) nodes with the lowest \( t \) distances to \( v \). Whatever is stored in \( B \), is stored also in \( H \), with additional data regarding the distance to those nodes and the neighbors of \( v \) toward them. So we make further tests on \( H \) instead of \( B \). Besides the nodes in \( B \), \( H \) contains also information regarding the direct neighbors of \( v \).

In order to detect eventual discrepancies, each node sends its t-ball \( B \) to each neighbor. From [10], we know that: if \( u \in B \), then for every node \( x \) on the shortest path from \( v \) to \( u \), \( u \in B \).

The way a node \( v \) selects its nodes in \( B \) is by comparing different distances received from all the neighbors that also includes those nodes in their t-balls. Checking the other t-balls help us to eliminate wrong nodes and cycles in delivering messages.

On receiving a message DIST from a neighbor \( nbr \), if the message contains its own ID (DIST.sender = IDv), discard it. Otherwise, process the message and eventually broadcast it to the other neighbors (if any). Message processing means:
- add the length(link to \( nbr \)) to the field distance in the message
- if the updated distance is within the top of the \( t \) lowest distances, breaking ties by increasing node ID, the ID is stored in \( B \) and \( H \), and the message will be broadcast to all the other neighbors.
- otherwise, discard the message.

Algorithm 2.3 Calculate_Ball

<table>
<thead>
<tr>
<th>Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>BALL: Sender: the ID of the sender ( B ) : the set ( B ) ( dest ) : the ID of the destination ( sender ) : the sender ID, a neighbor of the current node ( BN ) : the t-ball of the sender ( HN ) : the data structure ( H ) of the sender</td>
</tr>
</tbody>
</table>

DIST: sender: the ID of the sender \( dist \) : the length of the path the message went through

LOST: id: the sender ID \( id2 \) : the ID of the other node adjacent to the crashed link

Local variables \( id, u, nb, nbr : int /* elements in \( N \) */ 
updated, to_send : Boolean

Macros

REMOVE = eliminate a wrong node
input id: int /* wrong node to be removed */ 
\( N \) : set of int /* the set of neighbors to be warned about */ 
UPDATE NBRS = send DIST messages to a set of neighbors and update some local variables
UPDATE = update the data structure \( B \) and \( H \)

input id, dist, nbr: int

Actions

2.01 \( B \cup ID, \setminus Rcvd_IDs \not= \emptyset \longrightarrow \text{RESTART} \)

2.02 \( \exists id \in H.(id \not= N \land H[id].direct = \text{true}) \lor \)
\( H[id].neighbor \not= \emptyset \longrightarrow \)
\( /* \text{id is a wrong node and remove it from } B \) \)
and \( H \) */

2.03 REMOVE (id, \( N \))

2.04 \( B = \emptyset \longrightarrow \)

2.05 Rcvd_IDs := \{ ID \}

2.06 \( H := \emptyset \)

2.07 Upon RECEIPT of DIST(s, dist, \( s \)) FROM neighbor \( nbr \) \( \longrightarrow \)

2.08 if \( (s \not= ID) \)

2.09 then

2.10 \( \text{dist} := \text{dist} + \text{length(link to } \text{nbr}) \)
\( /* \text{update the information in } B \) \)
\( H \) */

2.11 UPDATE (s, dist, \( s \), \(_nbr \))

2.12 endif

2.13 Upon RECEIPT of LOST(id1, id2) FROM neighbor \( nbr \) \( \longrightarrow \)

2.14 if \( (id2 \in H \land H[id2].neighbor = \text{id1}) \)

2.15 then

2.16 REMOVE(id2, \( N \setminus \{ nbr \} \))

2.17 endif

2.18 Upon RECEIPT of CHECK(s, \( B \)u, \( H \)) FROM neighbor \( nbr \) \( \longrightarrow \)

2.19 if \( (s = \text{nbr}) \)

2.20 then

2.21 for all \( (u \in B \land u \not= s \land H[u].neighbor = s) \)

2.22 if \( (u \not= \text{B}) \land (u \in \text{B} \land \)
\( H[u].distance \not= H[u].distance + \text{length(link to } \text{nbr}) \)

2.23 then

2.24 \( REMOVED(u, \text{N}) \)

2.25 endif

2.26 endfor

2.27 endif
Macros

**REMOVE** (id, N)
- R.01 Remove_ID (id, H, B)
- R.02 SEND LOST(IDs_id) TO all nb ∈ N
- R.03 Revd_IDs := Revd_IDs \ { id }

**SEND_NBRS**
- /* the message DIST is forwarded to the other neighbors */
- S.01 SEND DIST(s, dist_s) TO all nb ∈ N \ { nb }  
- S.02 updated := true

**UPDATE** (id, dist, nbr)
- Local variable
  - ID_max_dst : int
- U.01 Revd_IDs := Revd_IDs \ { s }
- U.02 if (id ∈ B) then
- U.03 endif
- U.04 if (id ∈ H ∧ (H[id].distance > dist ∨ (H[id].distance < dist ∧ H[id].neighbor = nbr))) then
- U.05 endif
- U.06 H[id].distance := dist
- U.07 H[id].neighbor := nbr
- U.08 H[id].direct := false
- U.09 SEND_NBRS
- U.10 endif
- U.11 else
- U.12 if (|B| < t) then
- U.13 endif
- U.14 NewCell (H, B, id, dist, nbr, false)
- U.15 SEND_NBRS
- U.16 else
- U.17 ID_max_dst := Maximum_Distance (B, H)
- U.18 if (H[ID_max_dst].distance > dist) ∨ (H[ID_max_dst],distance = dist ∧ H[ID_max_dst].distance > id)) then /* id is inserted and ID_max_dst is removed */
- U.19 endif
- U.20 Remove_ID (ID_max_distance, H, B)
- U.21 NewCell (H, B, id, dist, nbr, false)
- U.22 SEND_NBRS
- U.23 endif
- U.24 endif
- U.25 endif

3. Correctness

All the proofs are made for a generic node v. We show that the partial t-ball B will eventually contain only correct IDs.

For updating, B, v receives only DIST messages with correct distances. Besides the removal of the wrong IDs from B (Properties 1 and 2), we have to show that B gets emptied at most once (by executing **RESTART**) in every execution of the algorithm (Lemma 1), such that the routing table H will eventually contain only correct information.

Using Theorem 1 and Lemma 2 we prove that the algorithm stabilizes in O(d + c) time units, where d is the diameter of the network and c a large constant. Therefore **SRS** constructs a routing scheme in polynomial time and it is self-stabilizing also.

3.1 A Correct T-ball

To calculate the t-ball B for an arbitrary node v in the network, we use guarded commands in the algorithm **Calculate_Ball** and some guards in **Error_Correction**.

Adding nodes to B is done automatically, and the macro **UPDATE** in **Calculate_Ball** takes care of it. The main concern is to remove the “bad” nodes, with invalid information in H and/or B. First, we show that the partial t-ball B will contain only correct IDs, and the wrong IDs from B are removed (Properties 1 and 2). Next, we prove that B can become ∅ at most once, so B converges to a correct t-ball (Lemma 1).

We have the following observations, most of them referring to macro **UPDATE** in **Calculate_Ball**.

**Observation 1** For ∀ u ∈ B, H[u].distance contains the lowest distance received toward u (lines U.10-13). Starting from an arbitrary configuration, after a finite time H[u].distance is the lowest distance between v and u.

Consider m as the initial value of H[u].distance. If m ≥ some distance from u to v received in some message, m gets later overwritten by that distance and H[u].distance converges to the shortest distance (condition H[id].distance > dist in line U.04 where id = u and dist is the value of the distance received in that message).

If m is smaller than any possible path length from v toward u, we show next that m gets replaced with a correct value and later H[u].distance converges to the shortest distance.

The value m is stored as the distance from u to v through a neighbor H[u].neighbor. This neighbor keeps also u in its t-ball, otherwise it would not have forwarded the message. If that neighbor forwards to v another distance to u, this value replaces m. This action does not affect the process of selecting the shortest distance to u, because the overwrite is done only in case a new distance is received from the neighbor toward u on the shortest path known up to this point. Take a particular example in Figure 1:

![Figure 1. A correct distance replaces the old one](image)

Assume now that v knows that u is at the distance 10 and this is the shortest distance to u, so it is stored in H[u].distance=10. Node v receives the distance from u through its neighbor on the path stored as shortest up to this point to be 17. Then v
replaces 10 by 17 and maybe a better value will overwrite later 17.

**Observation 2** $B$ contains the first $t$ nodes in increasing order of the distance. If $B$ has less than $t$ values, an incoming node is simply added to $B$ (lines U.14-15). If $B$ has exactly $t$ elements and a new node should be added, the one with the longest distance, breaking ties by increasing node ID, is removed (lines U.17-22).

Another situation regards “down” nodes. When we say that node $u$ is down, we see this from the point of view of $v$: either $u$ fails, or on the path from $u$ to $v$ some link is down, so $v$ does not “see” $u$ as an up node. If it is only a link failure, $v$ will probably “see” $u$ through another path.

For any node $u$, $u \neq v$, we have the following observations:

**Observation 3** If $u$ is stored in $H_v$ as a neighbor of $v$, but it is not a current neighbor ($u \notin N_v$), $u$ gets removed from $H_v$ (the guard 2.02 becomes true in node $v$ and it is executed for $id = u$).

**Observation 4** If $u$ is stored in $H_v$ as reaching $v$ through a non-existing neighbor, $u$ gets removed also (the guard 2.02 becomes true).

**Observation 5** If $u$ has reached $v$ through a neighbor $w$ of $v$ ($w \in N_v$) (see Figure 2) and the path between $u$ and $w$ does not exist anymore (we received a message LOST($w$, $u$)), then the path between $u$ and $v$ is removed from $H_v$ (also (lines 2.13-17).

![Figure 2. The node $u$ has reached $v$ through $w$ but the path $u \rightarrow w$ is disconnected](image)

**Observation 6** From (10) we know that: $u \in B_v \Rightarrow \forall w$ on the shortest path from $u$ to $v$, $u \in B_v$.

Thus $v$ checks this property with its direct neighbors by receiving their $t$-balls and sending its $t$-ball, whenever a change occurs in $B$.

Based on these observations, we prove further properties. The first property shows the removal of non-existing nodes. A non-existing node is a node, which either has failed, or it was never an up node in the network.

**Property 1** Starting from an arbitrary configuration, if a node $u$ fails, or some links are down, or $u$ does not exist, all nodes $v$, which have $u$ included in $B_v$ as reachable through those links, will remove $u$ from their $B$ and $H$ data structure in finite time.

**Proof** The proof is by induction on the number of hops from $u$ to an arbitrary $v$, such that $u \in B_v$.

- case i) If $u$ is stored as a direct neighbor of $v$ ($H_v[u].\text{direct} = \text{true}$) then by Observation 3, $u$ is removed and the information is broadcast to the other nodes, as we remarked in Observation 6.
- case ii) If $H_v[u].\text{direct} = \text{false}$ but $H_v[u].\text{neighbor} = v$, by either Observation 3 or 4, $u$ is removed and the information is broadcast to the other nodes.

- case iii) $\exists w$: $H_v[w].\text{direct} = \text{false} \land H_v[w].\text{neighbor} = w \land w \neq u$. It is compulsory for $w$ to be an up neighbor, otherwise the guard 2.02 becomes true in node $v$ for $id = w$, and $w$ and all the other nodes which reach $v$ through $w$ get removed (Observation 4). We have a situation like this (Figure 3):

![Figure 3. The down node $u$ reached $v$ through $w$](image)

For any node $w_k$ such that $u \in N_{w_k} \land H_{w_k}[u].\text{direct} = \text{true} \Rightarrow B_{w_k}$ and $H_{w_k}$ get updated. Recursively, $B_u$ and $H_u$ get updated.

**Property 2** Starting from an arbitrary configuration, the cycles in forwarding a message to any arbitrary node are eventually removed from $B$ and $H$.

**Proof** A cycle in this case means that a node forward a message to another node, that node to another one so on, until the message is forwarded back to the first node, without reaching the destination. Figure 4 shows an example of a cycle of dimension 3 with the following data:

- $H_v[x].\text{neighbor} = w : v$ knows that the best neighbor to reach $x$ is $w$
- $H_w[x].\text{neighbor} = u : w$ knows that the best neighbor to reach $x$ is $u$
- $H_u[x].\text{neighbor} = v : u$ knows that the best neighbor to reach $x$ is $v$

![Figure 4. A cycle of dimension 3](image)

Therefore a message sent to $x$, once it enters the cycle, it goes forever. Simply checking whether $x \in B_u \land H_u[x].\text{neighbor} = v \Rightarrow x \in B_v$ leaves a cycle undetected.

A stronger condition should be added such that, at some point, $x$ is removed from a set $B$ of a node along the cycle and recursively, $x$ gets removed from all the other nodes that form the cycle. The extra condition that removes the eventual cycles is $H_v[x].\text{distance} = \text{length}(v, w) + H_w[x].\text{distance}$.

To understand how it works, consider the following example (Figure 5):

![Figure 5. A correct situation](image)

Here, we see that $H_v[x].\text{distance}$ must be $2 + 3 = 5$. Suppose now $H_v[x].\text{distance} = 12 \Rightarrow H_u[x].\text{distance}$ must be $12 - 4 = 8 \Rightarrow H_w[x].\text{distance}$ must be $8 - 3 = 5 \Rightarrow H_x[x].\text{distance}$ must be $5 - 2 = 3$. This
contrasts the initial value of \( H_v[x].distance = 12 \).

Up to this point we have shown how to remove “bad” nodes from \( B \). Another way to remove elements from \( B \) is to set it to \( \emptyset \). We show that it is possible at most once in every execution of the algorithm SRS in node \( v \). In this way, \( B \) will contain, in infinite time, the \( t \) closest nodes to \( v \), so gradually, the cover will be calculated based on these values, and finally the labeling functions.

**Lemma 1** Starting from an arbitrary configuration, in any execution, \( v \) executes RESTART at most once.

**Proof** Suppose we have executed RESTART. We prove that further actions do not determine another RESTART, by looking at each guard from all the modules that have as action RESTART and we prove that they cannot become enabled again (see Properties 3, 4, and 5).

**Property 3** After RESTART is executed, in Error_Correction the predicate error remains false (so, its action RESTART is not executed anymore).

**Proof** After RESTART is executed, the data structure \( B \) and \( H \) are \( \emptyset \). In the algorithm Calculate_Ball, whenever \( B \) adds or removes an element, the same element is added/removed from \( H \). Also, whenever a crash occurs in the network, the node \( v \) does not become disconnected, so it has at least one up neighbor. So \( B \) does not become \( \emptyset \) because of Remove_ID executions. Therefore the condition \( B \setminus H \neq \emptyset \) is false from here on.

**Property 4** Once the macro RESTART is executed, the guard 1.04 of Error_Correction (keeping correct data about the neighbors) remains false.

**Proof** The field direct specifies whether a direct neighbor of \( v \) has the shortest path to \( v \) through the link between them.

\( H \) starts as \( \emptyset \). Whenever a node is added/updated in \( H \), the value of the field direct of that element is set to false (lines U.08, U.14, U.21, of the macro UPDATE in the algorithm Calculate_Ball).

The only statement that sets the value of the field direct for a node \( u \) to true is in the lines 1.06-1.08 of the algorithm Error_Correction. But this is done only if \( u \) is neighbor of \( v \) and has the direct link as the shortest path \( (H[u].neighbor = u) \) and, in this case the field distance is set to the correct value (the value of the length of the link).

The field distance can change when a shorter distance is detected. But at that time, the field direct is set to false automatically (lines U.08 of the macro UPDATE in the algorithm Calculate_Ball).

Taking an example (Figure 6):

![Figure 6. A shorter distance through another neighbor](image)

Suppose that initially for the node \( w \) stored in \( H_w \), the values of the fields are:

- \( H[w].neighbor = w \)
- \( H[w].distance = 5 \)
- \( H[w].direct = true \)

If node \( v \) detects a shorter distance to node \( w \) through node \( u \), the values are changed in the macros UPDATE (lines U.06-08) to:

- \( H[w].neighbor = u \)
- \( H[w].distance = 3 \)
- \( H[w].direct = false \)

**Property 5** The guard 2.01 in the algorithm Calculate_Ball (keeping data about unknown nodes, whose DIST messages have not been yet received) remains false.

**Proof** The set Rcvd_IDs keeps all the nodes which have sent information regarding the distance toward the node \( v \). These distances are valid data.

Obviously, we cannot trust the information regarding a node in \( B \) whose message has not been yet received. Because of that we execute RESTART when such a node is detected. After RESTART gets executed and \( B \) becomes \( \emptyset \), \( \text{Rcvd\_IDs} := \{ ID_v \} \). From here on, \( \text{Rcvd\_IDs} \) will start storing only the IDs of the nodes that have sent messages and maybe have changed the set \( B \). So, \( B \subset \text{Rcvd\_ID} \), so the guard 2.01 remains false.

So we have at most one RESTART in every execution.

**Theorem 1** Starting from an arbitrary configuration, the partial t-ball \( B \), becomes in finite time the t-ball of node \( v \), \( B_v(t) \), required by SRS scheme. The routing table \( H \) has the size \( O((t + \Delta_v)\log(n)) \).

**Proof** Using the Properties 1 and 2, Lemma 1, and Observation 1.

We know, by Property 1, that \( B \) will contain only the up nodes in the network, and with the cycles removed (Property 2). The guards of the algorithm Calculate_Ball updates \( B \) in case of new distances or topology changes (Observation 1).

By Lemma 1, once a node starts executing the distributed algorithm, we can have at most one RESTART, which means that \( B \) can be reset to \( \emptyset \) at most once.

\( H \) will contain, besides the nodes in \( B \), the direct neighbors of \( v \), which makes a total of at most \( t + \Delta_v \) entries (some neighbors can be in \( B \) also, and we don’t have redundant entries). Each entry has \( O(\log(n)) \) data bits, so \( H \) will have \( O((t + \Delta_v)\log(n)) \) bits.

### 3.2 Self-Stabilization and Time Complexity

We consider \( d \) to be the diameter of the network. In case of network change, \( d \) can be modified, so, to be more precise, consider \( d \) to be the maximum diameter over all the diameters of the network in
extended to an unweighted network. It takes stabilizing routing algorithm for an asynchronous, purposes. The proposed algorithm is a self-stabilizing routing algorithm for an asynchronous, small amount of memory in the nodes for routing direct routing schemes, which require a relatively become larger and larger, it is essential to design routing functions use \( O((t + \Delta) \log(n)) \) bits in each node. It can be used further for different types of routing (hierarchical, compact, interval etc).

Routing algorithms can be used to design efficient solutions to some fundamental problems in distributed computing, such as broadcasting, mutual exclusion, BFS, and DFS. There already exist self-stabilizing solutions to the above problems ([12]). One interesting topic of future research is to find efficient self-stabilizing solutions (more efficient than the existing ones) to the above problems.

Theorem 2 The distributed algorithm constructs a SRS scheme in polynomial time. The routing table in node \( v \) is of size \( O((t + \Delta) \log(n)) \) bits and with a total of \( O(n t \log(n)) \) bits per network. The algorithm stabilizes in \( O(d+c) \) time units where \( c \) is a large constant and \( d \) is the diameter of the network, and the routing

4. Conclusions

In this paper, we have presented a self-stabilized routing scheme SRS. As high-speed networks become larger and larger, it is essential to design direct routing schemes, which require a relatively small amount of memory in the nodes for routing purposes. The proposed algorithm is a self-stabilizing routing algorithm for an asynchronous, arbitrary weighted network, and can easily be extended to an unweighted network. It takes \( O(d+c) \) time units to stabilize, where \( c \) is a large constant and \( d \) is the diameter of the network, and the routing

5. References