Compact Routing Messages in Self-Healing Trees

[Extended Abstract]

Armando Castañeda[*]
Instituto de Matemáticas,
UNAM
Mexico
armando.castaneda@im.unam.mx

Danny Dolev [†]
The Hebrew University of
Jerusalem
Israel
danny.dolev@mail.huji.ac.il

Amitabh Trehan [‡]
School of Electronics,
Electrical Engineering and
Computer Science
Queen's University Belfast, UK
a.trehan@qub.ac.uk

ABSTRACT

Existing compact routing schemes, e.g., Thorup and Zwick [SPAA 2001] and Chechik [PODC 2013], often have no means to tolerate failures, once the system has been setup and started. This paper presents, to our knowledge, the first self-healing compact routing scheme. Besides, our schemes are developed for low memory nodes, i.e., nodes need only $O(\log^2 n)$ memory, and are thus, compact schemes.

We introduce two algorithms of independent interest: The first is *CompactFT*, a novel compact version (using only $O(\log n)$ local memory) of the self-healing algorithm Forgiving Tree of Hayes et al. [PODC 2008]. The second algorithm (*CompactFTZ*) combines CompactFT with Thorup-Zwick's tree-based compact routing scheme [SPAA 2001] to produce a fully compact self-healing routing scheme. In the self-healing model, the adversary deletes nodes one at a time with the affected nodes self-healing locally by adding few edges. CompactFT recovers from each attack in only $O(1)$ time and Δ messages, with only $+3$ degree increase and $O(log\Delta)$ graph diameter increase, over any sequence of deletions (Δ is the initial maximum degree).

Additionally, CompactFTZ guarantees delivery of a packet sent from sender s as long as the receiver t has not been deleted, with only an additional $O(y \log \Delta)$ latency, where y is the number of nodes that have been deleted on the path between s and t. If t has been deleted, s gets informed and the packet removed from the network.

CCS Concepts

•**Mathematics of computing** → **Trees;** *Paths and connectivity problems; Graph algorithms;* Network flows; •**Networks** → **Overlay and other logical network structures;** •**Networks** → **Peer-to-peer networks;** *Network protocol design; Network algorithms; Network reliability;* Mobile networks; Ad hoc networks; •**Theory of computation** → **Graph algorithms analysis; Routing and network design problems; Distributed algorithms;** *Data structures design and analysis;* •**Computing methodologies** → **Distributed algorithms;** •**Computer systems organization** → *Fault-tolerant network topologies;* •**Information systems** → Data structures;

Keywords

Self-Healing, Distributed Algorithm, Compact Routing, Low Memory, Compact Algorithm, Forgiving Tree, Internet of Things, P2P

1. INTRODUCTION

Efficient routing is becoming critical in current networks, and more so in future networks. Routing protocols have been the focus of intensive research over the years. Routing is based on information carried by the traveling packets and data structures that are maintained at the intermediate nodes. The efficiency parameters change from time to time, as the network use develops and new bottlenecks are identified. It is clear that the size of the network eliminates the ability to use any centralized decisions, and we are close to giving up on maintaining long distance routing decisions. We are a few years before a full scale deployment of IOT (Internet of Things) that will introduce billions of very weak devices that need to be routed. The size of the network and the dynamic structure that will evolve will force focusing on local decisions. The weakness of future devices and the size of the network will push for the use of protocols that do not require maintaining huge routing tables.

Santoro and Khatib [44], Peleg and Upfal [40], and Cowen [12] pioneered the concept of compact routing that requires only a minimal storage at each node. Moreover, the use of such routing protocols imposes only a constant factor increase on the length of the routing. Several papers followed

[*] Supported by UNAM-PAPIIT IA101015.

[†] Danny Dolev is Incumbent of the Berthold Badler Chair in Computer Science, The Rachel and Selim Benin School of Computer Science and Engineering Edmond J. Safra Campus, This research project was supported in part by The Israeli Centers of Research Excellence (I-CORE) program, (Center No. 4/11) and by grant 3/9778 of the Israeli Ministry of Science and Technology.

[‡] Research partly done as an I-CORE fellow supported by The Israeli Centers of Research Excellence (I-CORE) program, (Center No. 4/11)

ICDCN '16, January 04-07, 2016, Singapore, Singapore
© 2015 ACM. ISBN 978-1-4503-4032-8/16/01…$15.00
DOI: http://dx.doi.org/10.1145/2833312.2833328

up with some improvements on the schemes (cf. Thorup and Zwick [45], Fraigniaud and Gavoille [19], and Chechik [8]). These efficient routing schemes remain stable as long as there are no changes to the network.

The target of the current paper is to introduce an efficient *compact* scheme that combines compact routing with the ability to correct the local data structure stored at each node in a response to the change. Throughout this paper, when we say compact, we imply schemes that use $o(n)$ local memory (in our case, we actually only use $O(\log^2 n)$ local memory) per node. Our new scheme has similar cost as previous compact routing schemes. We will focus on node failures, since that is more challenging to handle.

Our algorithms work in the *bounded memory self-healing mode*l (Section 1.1). We assume that the network is initially a connected graph over n nodes. All nodes are involved in a preprocessing stage in which the nodes identify edges to be included in building a spanning tree over the network and construct their local data structures. The adversary repeatedly attacks the network. The adversary knows the network topology and the algorithms, and has the ability to delete arbitrary nodes from the network. To enforce a bound on the rate of changes, it is assumed the adversary is constrained in that it deletes one node at a time, and between two consecutive deletions, nodes in the neighbourhood of the deleted node can exchange messages with their immediate neighbours and can also request for additional edges to be added between themselves.

Our self-healing algorithm CompactFT ensures recovery from each attack in only a constant time and Δ messages, while, over any sequence of deletions, taking only constant additive degree increase (of 3) and keeping the diameter as $O(D \log \Delta)$, where D is the diameter of the initial graph and Δ the maximum degree of the initial spanning tree built on the graph. Moreover, CompactFT needs only $O(\log n)$ local memory (where n is the number of nodes originally in the network). Theorem 3.1 states the results formally.

CompactFTZ, our compact routing algorithm, is based on the compact routing scheme on trees by Thorup and Zwick [45], and ensures routing between any pair of existing nodes in our self-healing tree without loss of any message packet whose target is still connected. Moreover, the source will be informed if the receiver is lost, and if both the sender and receiver have been lost, the message will be discarded from the system within at most twice of the routing time. Our algorithm guarantees that after any sequence of deletions, a packet from s to t is routed through a path of length $O(d(s,t) \log \Delta)$, where $d(s,t)$ is the distance between s and t, and Δ is the maximum degree of any node in the initial tree. Though CompactFTZ uses only $\log n$ local memory, the routing labels (and, hence, the messages) are of $O(\log^2 n)$ size, so nodes may need $O(\log^2 n)$ memory to locally process the messages. Theorem 4.2 states the results formally.

A few modifications are needed to [45] to achieve our compact fault-tolerant scheme. We have to ensure that the packets are routed despite the deletions, subsequent self-healing, and, thus, loss and obsolescence of some information, i.e., we would like to continue without updating outdated routing tables and labels. This is partly achieved by using a post-order DFS labelling that allows the self-healing nodes to do routing using just simple binary search in the affected areas ([45] uses pre-order DFS labels). Other modifications to the algorithm and labels modify the routing according to the self-healing state of the nodes, ensure packets are not lost while self-healing and allow undelivered packets to dead targets to be returned to alive senders.

Related Work.

Algorithm		Over Complete Run	
	Local Memory	Diameter (Orig: D)	Degree (Orig: d)
Forgiving Tree [24]	$O(n)$	$D \log \Delta$†	$d+3$
CompactFT	$O(\log n)$	$D \log \Delta$†	$d+3$
	Per Healing Phase		
	Parallel Repair Time	Msg Size	# Msges
Forgiving Tree [24]	$O(1)$	$O(\log n)$	$O(1)$
CompactFT	$O(1)$	$O(\log n)$	$O(\delta)$‡

† Δ: Highest degree of network.

‡ δ: Highest degree of a node involved in repair (at most Δ).

Table 1: Comparison of CompactFT with Forgiving Tree

CompactFT uses ideas from the *Forgiving Tree* [24] (FT, in short) approach in order to improve compact routing. The main improvement of CompactFT is that no node uses more than $O(\log n)$ local memory and thus, CompactFT is compact. CompactFT achieves the same bounds and healing invariants as FT, however, taking slightly more messages (at most $O(\Delta)$ messages as opposed to $O(1)$ in FT) in certain rounds. Table 1 compares both algorithms.

Several papers have studied the routing problem in arbitrary networks (e.g. [2, 3, 12, 8]) and with the help of geographic information (e.g. [7, 21, 31]), but without failures. These papers are interested in the trade-off between the size of the routing tables and the stretch of the scheme: the worst case ratio between the length of the path obtained by the routing scheme and the length of the shortest path between the source and the destination. Here we are mainly interested in preserving compactness under the presence of failures.

An interesting line of research deals with labelling schemes. [29] presents labelling schemes for weighted dynamic trees where node weights can change. However, it does not deal with node deletions nor does it claim to deal with routing. In [30], Korman et al. present a compact distributed labelling scheme in the dynamic tree model: (1) the network is a tree, (2) nodes are deleted/added one at a time, (3) the root is never deleted and (4) only leaves can be added/deleted. In a labelling scheme, each node has a label and from every two labels, the distance between the corresponding nodes can be easily computed. The fault-tolerant labelling scheme is obtained by modifying any static scheme. Using the previous, they get fault-tolerant (in the same model) compact versions of the compact tree routing schemes of [20, 19, 45]. These schemes have a multiplicative overhead factor of $\Omega(\log n)$ on the label sizes of the static schemes. In [27], Korman improves the results in [30], presenting a labelling scheme in the same model that allows to compute any function on pairs of vertices with smaller labels, at the cost, in some cases, of communication. Our work differs from the previous in the sense that though we use a spanning tree of the network, our network can be arbitrary and any node can be deleted by the adversary.

There have been numerous papers that discuss strategies for adding additional capacity and rerouting in anticipation of failures [10, 9, 11, 16, 18, 22, 26, 36, 47, 48, 49]. In each of these solutions, the network is fixed and either redundant information is precomputed or routing tables are updated when a failure is detected. In contrast, our algorithm runs in a dynamic setting where edges are added to the network as node failures occur, maintaining connectivity and preserving compactness at all time. Our bounded memory self-healing model builds upon the model introduced in [24]. A variety of self-healing topology maintenance algorithms have been devised in the self-healing models [38, 39, 25, 46, 43]. Our paper moves in the direction of self-healing computation/routing along with topology which is attempted in other papers e.g. [42] (though in a different model). Finally, dynamic network topoplogy and fault tolerance are core concerns of distributed computing [1, 35] and various models, e.g., [32], and topology maintenance and Self-* algorithms abound [6, 13, 14, 15, 28, 33, 41, 23, 5, 17, 4, 34].

1.1 Bounded Memory Self-healing Model

Let $G = G_0$ be an arbitrary connected (for simplicity) graph on n nodes, which represent processors in a distributed network. Each node has a unique ID. Initially, each node only knows its neighbors in G_0, and is unaware of the structure of the rest of G_0.

We allow a certain amount of pre-processing to be done before the adversary is allowed to delete nodes. In the preprocessing stage nodes exchange messages with their neighbors and setup data structures as required.

The adversarial process can be described as deleting some node v_t from G_{t-1}, forming H_t. All neighbors of v_t are informed of the deletion. In the healing stage nodes of H_t communicate (concurrently) with their immediate neighbors. Nodes may insert edges joining them to other nodes they know about, as desired. Nodes may also drop edges from previous phases if no longer required. The resulting graph at the end of this phase is G_t. Nodes are not explicitly informed when the healing stage ends. We make no synchronicity assumption except that all messages after the deletion of a node, i.e., in a healing phase, are safely received and processed before the adversary deletes the next node.

The objective is minimizing the following "complexity" measures (excluding the preprocessing stage):

- **Deg. increase:** $\max_{t<n} \max_v (\deg(v, G_t) - \deg(v, G_0))$
- **Diameter stretch:** $\max_{t<n} \mathrm{Dia}(G_t)/\mathrm{Dia}(G_0)$
- **Communication:** The maximum number of bits sent by a single node in each recovery phase
- **Recovery time:** The maximum total time for a recovery phase, assuming it takes a message no more than 1 time unit to traverse any edge
- **Local Memory:** The amount of memory a single node needs to maintain to run the algorithm.

1.2 Compact Routing Model

The algorithm is allowed a pre-processing phase, e.g., to run a distributed DFS on the graph. Each message has a label which may contain the node ID and other information derived from the pre-processing phase. Every node stores some local information for routing. The routing algorithm does not change the original port assignment at any node (however, node deletions may force the self-healing algorithm to do simple port re-assignments). We are interested in minimizing the sizes of the label and the local information at each node.

2. THE ALGORITHMS: HIGH LEVEL

As stated, CompactFT (Algorithm 3.1) is an adaptation of FT [24] for low memory. CompactFTZ (Algorithm 4.2) then conducts reliable routing over CompactFT. At a high level, the following happens:

- *Preprocessing:* A BFS spanning tree T_0 of graph G_0 is derived followed by DFS traversal and labelling and careful setup of CompactFT and CompactFTZ fields (Tables 2 and 4). For CompactFT, every node sets up and distributes a Will (Section 3), which is the blueprint of edges and virtual (helper) nodes to be constructed upon the node's deletion.
- After each deletion, the repair maintains the spanning tree of helper and real (original) nodes, i.e., the i^{th} deletion (say, of node v_i) and subsequent repair yields tree T_i. The helper nodes are simulated by the real nodes and only a real node can be deleted. The two main cases are as follows:
  - *non-leaf deletion, i.e., v_i is not a leaf in T_{i-1}* (Section 3.1): The neighbours of v_i 'execute' v_i's will leading v_i to be replaced by a *reconstruction tree* (RT(v_i)) which is a balanced binary search tree ($BBST$) having v_i's neighbours as the leaves and helper nodes as the internal nodes simulated by v_i's neighbours (Figure 1).
  - *leaf deletion, i.e., v_i is a leaf in T_{i-1}* (Section 3.2): This case is more complicated in the low memory setting since a node (in particular, v_i's parent p) cannot store the list of its children nor recompute its Will. If p was dead, v_i's siblings essentially deletes a redundant helper node while maintaining the structure. If p is alive, no new edges are made but p orchestrates a distributed update of its will while being oblivious of the identity of its children. Thus, when p is eventually deleted, the right structure gets put in place.
- *Routing:* Independent of the self-healing, a node s may send a message to node r (along with s's own label) using the CompactFTZ protocol. The label on the message (for r) along with the local routing fields at a real node tells the next node, say w, on the path. If, however, w had been deleted earlier, there could be a helper node on that port which is part of RT(w). Now, the message would be routed using the fact that the RT(w) is a BBST and make it to the right node at the end of RT(w). The message eventually reaches r, but if r is dead, the message is 'returned to sender' using s's label.

3. CompactFT: DETAILED DESCRIPTION

CompactFT maintains connectivity in an initially connected network with only a total constant degree increase per node and $O(\log \Delta)$ factor diameter increase over any sequence of attacks while using only $O(\log n)$ local memory (where n is the number of nodes originally in the network). The formal theorem statement is given in Theorem 3.1.

As stated, in CompactFT, a deleted node is replaced by a RT formed by (virtual) helper nodes simulated by its children (siblings in case of a leaf deletion) (Figure 1). This healing is carried out by a mechanism of wills:

Will Mechanism: A Will(v) is the set of actions, i.e.,

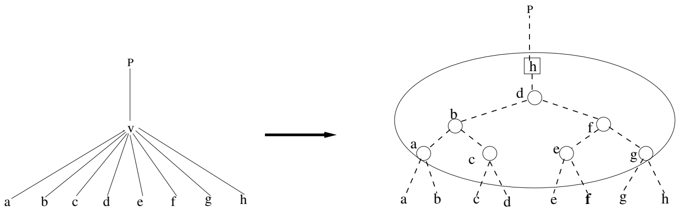


Figure 1: Deleted node v replaced by Reconstruction Tree (RT(v)). Nodes in oval are virtual helper nodes. The circles are regular helper nodes and the rectangle is 'heir' helper node. The 'Will' of v is RT(v), i.e., the structure that replaces the deleted v.

Current fields	Fields having information about a node's current neighbors
`parent(v)`	Parent of v
`parentport(v)`	Port at which $parent(v)$ is attached
`numchildren(v)`	Number of children of v
`maxportnumber(v)`	Maximum port number used by v
`heir(v)`,<$heir(v)$>	The heir of v and its port
Helper fields	Fields for a helper node v may be simulating. (Empty if none)
`hparent(v)`	Parent of the helper node v simulates
`hchildren(v)`	Children of helper node v simulates.
Reconstruction fields / WillPortion /LeafWillPortion	Fields used by v to reconstruct its connections when its neighbor is deleted.
`nextparent(v)`	Node which will be next Parent(v)
`nexthparent(v)`	Node which will be next hparent(v)
`nexthchildren(v)`	Node(s) that will be next hchildren(v)
Flags	Boolean fields telling node's status.
`hashelper(v)`	True if v is simulating a *helper* node

Table 2: The fields maintained by a node v for Compact FT. Each reference to a sibling is tagged with the port number at which it is attached to parent (not shown above for clarity) e.g. $nextparent(v)$ is $nextparent(v)$, <$nextparent(v)$>.

Message	Description
BrLeafLost (<x>)	Node v broadcasts, informing that the leaf node at v's port <x> has been deleted.
BrNodeReplace ((<x>, <x>), (h, <x>))	Node v broadcasts, asking receivers to replace (in their *willportions*) at v's port <x>, node x with node h.
PtWillConnection ((y, <y>), (z, <z>))	Node v asks receivers (in their v.Willportion) to make an edge between node y and node z.
PtNewLeafWill ((y, <y>), (z, <z>), $W(y)$)	Node v informs node z that it is the new LeafHeir(y) and gives it $W(y)$ (= LeafWill(y)).

Table 3: Messages used by *CompactFT* (sent by a node v).

the subgraph to be constructed on the deletion of node v. When v is a non-leaf node, this is essentially the encoding of the structure RT(v) and is distributed among v's children. Each part of a node's Will stored with another node is called a Willportion. We denote Willportion(p, v) to be the part of Will(p) that involves v, i.e., the relevant subgraph, and is stored by node v. When v is a leaf node, however, Will(v) differs in not being a RT(v) and is stored with siblings of v. For clarity, we call this kind of Will a LeafWill, the Willportions as LeafWillportions and a leaf node's heir as LeafHeir. Thus, a Will of a node is distributed among the node's neighbours such that the union of those Willportions makes the whole Will. Note that a Willportion (or LeafWillportion) is of only constant size (in number of node IDs). Figure 2 shows the Will of a node v and the corresponding Willportions.

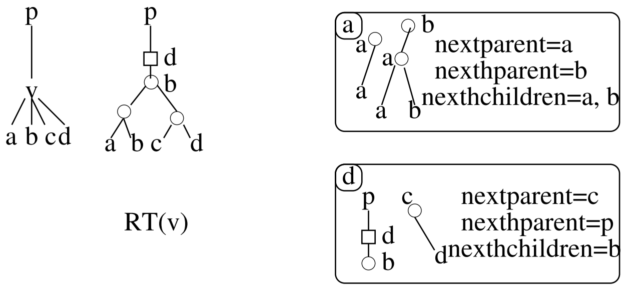


Figure 2: The network of a node v, v's Will = RT(v) and Willportions for its children a and d.

The fields used by a node for executing CompactFT are given in Table 2. Unlike FT, the node cannot have either the list of its children or its own RT (since these can be as large as $O(n)$). Rather, a node v will store the number of its children (*numchildren(v)*), highest port number in use (*maxportnumber*) and will store every node reference in the form (*nodeID*, *port*), e.g., in Willportion(p, v), a reference to a node x will be stored as (x, <x>), where <x> is the port of p at which x is connected.

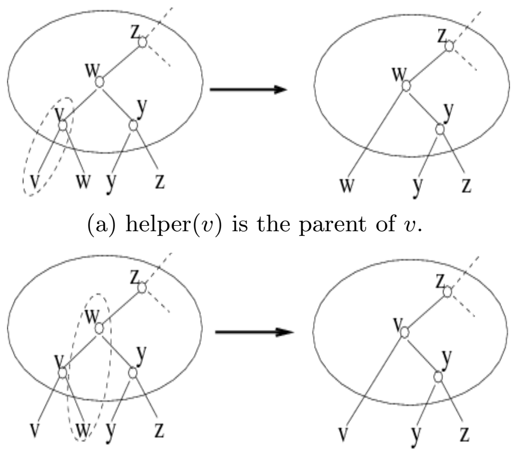
Algorithm 3.1 gives a high level overview of CompactFT. The detailed algorithms are presented in the full version of this paper. Also, Table 3 describes some of the special messages used by CompactFT. CompactFT begins with a preprocessing phase (Algo 3.1 line 1) in which a rooted BFS spanning tree of the network from an arbitrary node is computed. The algorithm will then maintain this tree in a self-healing manner. Each node sets up the CompactFT data structures including its Will. We do not count the resources involved in the preprocessing but note that at the end of that phase all the CompactFT data structures are contained within the $O(\log n)$ memory of a node. As stated, the basic operation is to replace a (non-leaf) node by a RT. A leaf deletion, however, leads to a reduction in the number of nodes in the system and the structure is then maintained by a combination of a 'short circuiting' operation and a helper node reassignment (this is also encoded in the leaf node's LeafWill and is discussed later). An essential invariant of CompactFT is that *a real node simulates at most one* helper *node* and since each helper node is a node of a binary tree, the degree increase of any node is restricted to at most 3. Similarly, since RTs are balanced binary trees, distances and, hence, the diameter of the CompactFT, blows up by at most a $\log \Delta$ factor, where Δ is the degree of the highest node (ref: Theorem 3.1). In the following description, we sometimes refer to a node v as Real(v) if it is real, or helper(v) if it is a helper node, or by just v if it is obvious from the context.

Algorithm 3.1 CompactFT(Graph G): High level view

```
1: Preprocessing and INIT: A rooted BFS spanning tree $T(V, E')$ of $G(V, E)$ is computed. For every node $v$, its Will (non-leaf or leaf as appropriate) is computed. Every node $x$ in a Will is labeled as $(x, <x>)$, where $x$ is $x$'s ID and $<x>$ is $x$'s parent's port number at which $x$ is connected (if it exists). Each node only has a Willportion and/or LeafWillportions ($O(\log n)$ sized portion of parent's or sibling's Will, respectively).
2: while true do
3:   if a vertex $x$ is deleted then
4:     if $x$ was not a leaf (i.e., had any children) then // Fix non leaf deletion.
5:       $x$'s children execute $x$'s Will using $x$'s Willportions they have; Heir($x$) takes over $x$'s Will/duties.
6:       All Affected Wills (i.e. neighbours of $x$ and of helper($x$)) are updated by simple update of relevant Willportions.
7:     else // Fix leaf deletion.
8:       Let node $p$ (if it exists) be node $x$'s parent // If $p$ does not exist, $x$ was the only node in the network, so nothing to do
9:       if $p$ is real/alive then // Update Wills by simulating the deletion of $p$ and $x$
10:        if $x$ was $p$'s only child then
11:          $p$ computes its LeafHeir and LeafWill and forwards it. // $p$ has become a leaf
12:        else
13:          $p$ informs all children about $x$'s deletion.
14:          $p$'s children update $p$'s Willportions using $x$'s LeafWillportions.
15:          Children issue updates to $p$'s Willportions and other LeafWillportions via $p$.
16:          $p$ forwards updates via broadcast or point-to-point messages, as required.
17:          $p$'s neighbours receiving these messages update their data structures.
18:        end if
19:      else // $p$ had already been deleted earlier.
20:        Let $y$ be $x$'s LeafHeir.
21:        $y$ executes $x$'s Will.
22:        Affected nodes update their and their neighbour's Willportions.
23:      end if
24:    end if
25:    if $x$ was node $z$'s LeafHeir then
26:      $z$ sets a new neighbour as LeafHeir following a simple rule.
27:    end if
28:  end if
29: end while
```

3.1 Deletion of a Non-Leaf Node:

Assume that a node x is deleted. If x was not a leaf node (Algorithm 3.1 lines 5 - 6), it's neighbours simply execute x's Will. One of x's children (by default the rightmost child) is a special child called the Heir (say, h) and it takes over any virtual node (i.e., helper(x)) that x may have been simulating, otherwise it is the one that connects the rest of the RT to the parent of x (say, p). This past action may lead to changes in the Wills of other live nodes. In particular, p will have to tell its children to replace x by h in p's Will. Due to the limited memory, p does not know the identity of x. However, when h contacts p, it will inform p that x has been deleted and p will broadcast a message BrNodeReplace($(x, < x >), (h, < x >)$) asking all neighbours to replace x by h in their Willportions at the same port (Table 3).



(a) helper(v) is the parent of v.

(b) helper(w) is not the parent of w.

Figure 3: Deletion of a leaf node whose parent is a helper **node: two cases.**

3.2 Deletion of a Leaf Node:

If the deleted node x was, in fact, a leaf node, the situation is more involved. There are two cases to consider: whether the parent p of x is a helper node (implying the original parent had been deleted earlier) or a real node. The second case, though trivial in FT (with $O(n)$ memory) is challenging in CompactFT. Before we discuss the cases, we introduce the 'short-circuiting' operation used during leaf deletion:

bypass(x): (from [24]) *Precondition*: $|\text{hchildren}(x)| = 1$, *i.e.,* the helper node has a single child.
Operation: *Delete* helper(x), i.e., Parent of helper(x) and child of helper(x) remove their edges with helper(x) and make a new edge between themselves.
hparent(x) ← *EMPTY*; hchildren(x) ← *EMPTY*.

1. *Parent p of x is a* helper *node*

   If p is a helper node, this implies that the original parent of x (in G_0) had been deleted at some stage and x has exactly one helper node in one of the RTs in the tree above. Since x has been deleted, p has only one child now and Bypass(p) can now be executed. There are two further cases:

   (a) helper(x) *is parent of x (Figure 3(a))*: In this case, the only thing that needs to be done is Bypass(x) since this bypasses the deleted nodes and restores connectivity. However, the issue is that helper(x) has already been deleted, so how is Bypass(x) to be executed? For this, we use the mechanism of a LeafWill. Assume helper(x) had two children, x and y. When x sets up its LeafWill (which consists only of Bypass(x)), it designates y as its LeafHeir and sends its LeafWill. In Figure 3(a), the LeafHeir of node v is w and the LeafWill(v) consists of the operation Bypass(v).

   (b) helper(x) *is not the parent of x (Figure 3(b))*: Let p be the parent of the deleted node x. Since p now has only one child left, it will have to be short-circuited by Bypass(p). However, the node helper(x) has also been lost. Therefore, if we don't fix that, we will disconnect the neighbours of helper(x). However, since p has been bypassed, Real(p) is not simulating a helper node anymore and, thus, Real(p) will take over the slot of helper(x) by making edges between its ex-neighbours. In this case, x simply designates p as its LeafHeir and leaves LeafWill(x) (which is of only $O(\log n)$ size) with p. In Figure 3(b), node w is deleted, its parent and LeafHeir is helper(v) and, thus, when w is deleted, following LeafWill(w), Bypass(v) is executed and v takes over helper(w).

The only situation left to be discussed is when x was a LeafHeir of another node. In this case, the algorithm follows the rules apparent from the cases before. Let v be the node that had x as its LeafHeir. Assume that after healing, p is the parent of Real(v) and assume for now that p is a helper node (the real node case is discussed later). Then, if p is helper(v), v makes the other child of p (i.e., v's sibling) as v's LeafHeir, otherwise v sets p as its LeafHeir and hands its Will over to the LeafHeir.

2. *Parent p of x is a real node*

   This case is trivial in FT as all that p needs to do is remove x from the list of its children (children(p) in FT), recompute its Will and distribute it to all its children. However, in CompactFT, p cannot store the list of its children and thus, update its Will. Therefore, we have to find a way for the Will to be updated in a distributed manner while still taking only a constant number of rounds. This is accomplished by using the facts that the Willportions are already distributed pieces of p's Will and each leaf deletion affects only a constant number of other nodes allowing us to update the Willportions Notice that since p is real, nodes cannot really execute x's Will as in case 1. However, Will(p) is essentially the blueprint of RT(p). Hence, what p and its neighbours do is execute Will(x) on Will(p): this has the effect of updating Will(x) to its correct state and when ultimately p is deleted, the right structure is in place.

   This 'simulation' is done in the following manner: p detects the failure of x and informs all its neighbours by a BrLeafLost(<x>) message (Table 3). The node that is LeafHeir(x), say v, will now simulate execution of LeafWill(x). As discussed in Case 1, a LeafWill has two parts: a Bypass operation and a possible helper node takeover by another node. Suppose the Bypass operation is supposed to make an edge between nodes a and b. Node v simulates this by asking p to send a PtWillConnection((a, <a>), (b, <b>)) message to its ports <a> and <b>. This has the effect of node a and b making the appropriate edge in their Willportion(p). Similarly, for the node take over of helper(x), v asks p to send PtWillConnection messages to make edges (in Willportions) between the node taking over and the previous neighbours of helper(x) in Will(p).

   Another case is when x was the LeafHeir of another node, say w. Since LeafHeir(x) has already done the healing, the Willportions are now updated and it is easy for w to find another LeafHeir. This is straightforward as per our previous discussion. The new LeafHeir will either be Real(w)'s Parent or (if Parent(w) = helper(w)) Parent(w)'s other child. Notice this information is already present in Willportion(p, w). The new LeafWill(w) is also straightforward to calculate. As stated earlier, every LeafWill has a Bypass and/or a node takeover operation. All the nodes involved are neighbours of w in Willportion(p, w). Therefore, this information is also available with w enabling it to reconstruct its new LeafWill which it then sends to the new LeafHeir via p using the PtNewLeafWill() message (Table 3). Finally, there is a special case:

   - *x was the only child of (real) parent p*:
     Finally, there is also the possibility of node x being the only child of its parent p in which case p will become a leaf itself on x's deletion. Node p can only be a Real node (a helper node cannot have one child) and since x does not have any sibling, x will not have any LeafHeir or LeafWill (rather, these fields will be set to NULL). Thus, when x will be deleted, there will be no new edges added. However, p will detect that it has become a leaf node and using p's parent's Willportion, it will designate a new LeafHeir, compute a new LeafWill (as discussed previously) and send it to its LeafHeir by messages (if p's parent is Real) or directly.

Theorem 3.1 (proof deferred to the full paper) summarises the properties of CompactFT.

THEOREM 3.1. *The* CompactFT *has the following properties:*

1. CompactFT *increases degree of any vertex by only 3.*
2. CompactFT *always has diameter $O(D \log \Delta)$, where D is the diameter and Δ the maximum degree of the initial graph.*
3. *Each node in* CompactFT *uses only $O(\log n)$ local memory for the algorithm.*
4. *The latency per deletion is $O(1)$ and the number of messages sent per node per deletion is $O(\Delta)$; each message contains $O(1)$ node* ID*s and thus $O(\log n)$ bits.*

4. A COMPACT SELF HEALING ROUTING SCHEME

In this section, we present CompactFTZ, a fault tolerant, self-healing routing scheme. First, we present a variant of the compact routing scheme on trees of Thorup and Zwick [45] (which we refer to as TZ in what follows), and then we make this algorithm fault-tolerant in the self-healing model using CompactFT (Section 3).

4.1 Compact Routing on Trees

We present a variant of TZ that mainly differs in the order of DFS labelling of nodes. The local fields of each node are changed accordingly. This variant allows us to route even in the presence of adversarial deletions on nodes when combined with CompactFT (Lemma 4.1).

Let T be a tree rooted at a node r. Consider a constant $b \geq 2$. The *weight* s_v of a node v is the number of its descendants, including v. A child u of v is *heavy* if $s_u \geq s_v/b$, and *light* otherwise. Hence, v has at most $b-1$ heavy children. By definition, r is heavy. The *light routing index* ℓ_v of v is the number of light nodes on the path from r to v, including v if it is light. We label a heavy node as

v	DFS number (post-order)
d_v	smallest descendent of v (in the original scheme, this is f_v, the largest descendant of v)
c_v	smallest descendent of first tzlight child of v, if it exists; otherwise $v{+}1$ (in the original scheme, this is h_v, the first tzheavy child of v)
H_v: array with $b+1$ elements	
$H_v[0]$ $H_v[1,\dots,H_v[0]]$	number of tzheavy children of v tzheavy children of v
P_v: array with $b+1$ elements	
$P_v[0]$ $P_v[1,\dots,H_v[0]]$	port number of the edge from v to its parent. port numbers from v to its tzheavy children
ℓ	light routing index of v

Table 4: Local fields of a node v: Locally, each node v stores the above information

tzheavy and a light node as *tzlight*. Note that here we are describing the scheme for the static case where the tree does not change over time. However, it is easy to extend this to the dynamic case (Section 4.2) by initially setting up the data structure in exactly the same way as the static case during preprocessing. Later on the classification into heavy and light type remains as it was set initially and need not be updated.

We first enumerate the nodes of T in *DFS post-order manner*, with the heavy nodes traversed before the light nodes. For each node v, we let v itself denote this number. This numbering gives the IDs of nodes (in the original scheme, the nodes are labelled in a pre-order manner and the light nodes are visited first). For ease of description, by abuse of notation, in the description and algorithm, we refer interchangeably to both the node itself and its ID as v.

Note that each node has an ID that is larger than the ID of any of its descendants. Moreover, given a node and two of its children u and v with $u < v$, the IDs in the subtree rooted at u are strictly smaller than the IDs in the subtree rooted at v. With such a labelling, routing can be easily performed: if a node u receives a message for a node v, it checks if v belongs to the interval of IDs of its descendants; if so, it forwards the message to its appropriate children, otherwise it forwards the message to its parent. Using the notion of tzlight and tzheavy nodes, one can achieve a compact scheme. The local fields for a node are given in Table 4. Note that each node v locally stores $O(b \log n)$ bits. The *label* $L(v)$ of v is defined as follows: an array with the port numbers reaching the light nodes in the path from r to v. The definition of tzlight nodes implies that the size of $L(v)$ is $O(\log^2 n)$, hence the size of the header $(v, L(v))$ of a packet to v is $O(\log^2 n)$. The scheme TZ is described in Algorithm 4.1.

4.2 The CompactFTZ scheme

CompactFTZ (Algorithm 4.2) is fault tolerant adaptation of TZ that runs in CompactFT. The *initialisation* phase (Algorithm 4.2 line 4.2) performed during preprocessing sets up the data structures for CompactFTZ in the following order: A BFS spanning tree of the network is constructed rooted at an arbitrary node, then a DFS labelling and TZ setup is done as in Section 4.1, followed by CompactFT data structures setup using the previously generated DFS numbers as node IDs. The underlying layer is aware of the node IDs, DFS number IDs and node labels to be used for sending messages (as in TZ).

Algorithm 4.1 The TZ scheme. Code for node v for a message sent to node w.

```
operation TZ_v(w, L(w)):
 1: if $v = w$ then
 2:     The message reached its destination
 3: else if $w \notin [d_v, v]$ then
 4:     Forward to the parent through port $P_v[0]$
 5: else if $w \in [c_v, v]$ then
 6:     Forward to a tzlight node through port $L(w)[\ell_v]$
 7: else
 8:     Let $i$ be the index s.t. $H_v[i]$ is the smallest tzheavy
        child of $v$ greater than or equal to $w$
 9:     Forward to a heavy node through port $P_v[i]$
10: end if
end operation
```

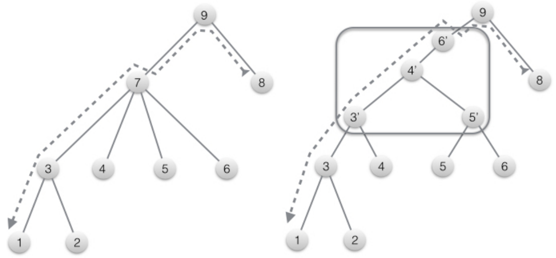


Figure 4: The left side shows the tree before any deletion with the path a message from 8 to 1 will follow. The right side shows the tree obtained after deleting 7. The nodes enclosed in the rectangle are virtual helper nodes replacing 7. To route a message from 8 to 1, virtual nodes perform binary search, while real nodes follow TZ.

Algorithm 4.2 The CompactFTZ scheme. Code for node v for a message sent to node w.

```
Preprocessing: Construct a BFS spanning tree of the network from an arbitrary node. Do a DFS labelling and TZ setup followed by CompactFT data structures setup using TZ DFS numbers as node $ID$s.
 1: $v$ runs CompactFT at all times.
 2: if $v$ is a real node then
 3:     Invoke $TZ_v(w, L(w))$
 4: else // $v$ is a virtual helper node (= helper($v$))
 5:     if $v = w$ then
 6:         The message has reached its destination
 7:     else if $w \notin [d_v, v]$ then
 8:         Forward to the parent of helper($v$) in the current
            virtual tree.
 9:     else if $w < v$ then
10:         Forward to the left child of helper($v$) in the current
            virtual tree.
11:     else
12:         Forward to the right child of helper($v$) in the current
            virtual tree.
13:     end if
14: end if
```

Recall that, in our model, if there is no edge between u and v, and port numbers x and y of u and v, respectively, are not in use, then u or v can request an edge (u, v) attached to these ports. In what follows, we assume that in CompactFT, when a child x of p is deleted and a child w of x creates an edge (p, w), such an edge will use the port of p used by (p, v) and any available port of w.

Every node runs CompactFT at all time. For routing, a real node just follows TZ (Algorithm 4.2, Line 3), while a virtual node first checks if the packet reached its destination (Line 5), and if not, it performs a binary search over the current virtual tree (Lines 7 to 12). As mentioned earlier, though we use the notion of light and heavy nodes in the initial setup and use it to compute routing tables and labels, we do not maintain this notion as the algorithm progresses but just use the initially assigned labels throughout. Further, following CompactFT, if a node x is deleted, it is replaced by $\mathrm{RT}(x)$. If a packet traverses $\mathrm{RT}(x)$, the virtual nodes ignore the heavy/light classification and just use the IDs to perform binary search. Figure 4 illustrates CompactFTZ in action. In the figure, node 8 sends a packet for node 1. If there is no deletion in the tree, the packet will simply follow the path via the root 9, node 7, node 3 to node 1. Recall that at each node, the node checks if the packet destination falls in the intervals given by its heavy node, otherwise, it uses the light routing index to pick the correct port to forward the message from the label of the destination node given in the message. However, if node 7 is deleted by the adversary, using CompactFT, the children of 7 construct RT(7) (recall this is also done in a compact manner). Since node 9 has helper node helper(6) at the port where it had node 7 earlier, the packet gets forwarded to node helper(6). Since 1 is less than 6, the packet traverses the left side of RT(7) and eventually reaches node 3. Node 3 applies the TZ routing rules as before and the packet reaches node 1.

We use the following notation: Let T_t be the CompactFTZ tree after t deletions. For a vertex v, let $T_t(v)$ denote the subtree of T_t rooted at v. The set with the children of v in T_t is denoted as $children_t(v)$, while $parent_t(v)$ is the parent of v in T_t. The set of IDs in $T_t(v)$ is denoted as $\mathrm{ID}(T_t(v))$. If v has two children, $left_t(v)$ $(right_t(v))$ denotes the left (right) child of v, and $L_t(v)$ $(R_t(v))$ denote the left (right) subtree of v. Given two nodes u and v, we write $u < \mathrm{ID}(T_t(v))$ if $\mathrm{ID}(u)$ is smaller than any ID in $\mathrm{ID}(T_t(v))$, and similarly, we write $\mathrm{ID}(T_t(u)) < \mathrm{ID}(T_t(v))$ if every ID in $\mathrm{ID}(T_t(u))$ is smaller than any ID in $\mathrm{ID}(T_t(v))$. The definitions naturally extends to $>, \leq$ and $\geq$.

Theorem 4.2 states the routing properties of CompactFTZ. Lemma 4.1 is the key to proving Theorem 4.2 (proofs deferred to the full version). Lemma 4.1 basically states that, after any sequence of deletions and subsequent self-healing, real nodes maintain the TZ properties and the helper nodes (i.e. the RTs) the BST properties, allowing routing to always function.

Lemma 4.1. *At every time t, the* CompactFTZ *tree T_t satisfies the following two statements:*

1. *For every real node $v \in T_t$, for every $c \in children_t(v)$, $v > ID(T_t(c))$, and for every $c, d \in children_t(v)$ with $c < d$, $ID(T_t(c)) < ID(T_t(d))$.*
2. *For every virtual node* helper$(v) \in T_t$, *$v \geq ID(L_t(v))$ and $v < ID(R_t(v))$.*

Theorem 4.2. *For each T_t, for each two real nodes $u, w \in T_t$,* CompactFTZ *successfully delivers a message from u to w through a path in T_t of size at most $\delta(u, w) + y(\log \Delta - 1)$, where $\delta(u, w)$ is the distance between u and w in T_0 and $y \leq t$ is the number of non-leaf nodes deleted to get T_t.*

Lemma 4.3 states the memory usage of CompactFTZ leading to the final correctness theorem (Theorem 4.4).

Lemma 4.3. CompactFTZ *uses only $O(log^2 n)$ memory per node to route a packet.*

Proof. First, CompactFT uses only $O(\log n)$ local memory (Theorem 3.1). The local fields of a node for routing have at most a constant number $(O(b))$ fields which are node references $(\log n)$ size, thus, using $O(\log n)$ memory. The label of a node (which is the 'address' on a packet) is, however, of $O(\log^2 n)$ size (since there can be $O(\log n)$ light nodes on a source-target path) and therefore, a node needs $O(\log^2 n)$ bits to process such a packet. □

Ignoring congestion issues, Lemma 4.3 implies that a node can store and route unto x packets using $x. \log^2 n$ local memory.

Theorem 4.4. CompactFTZ *is a self-healing compact routing scheme.*

Proof. The theorem follows directly from Lemma 4.3, Theorems 4.2 and 3.1. □

4.3 Reporting Non-delivery (deleted receivers and sources)

Contrary to what happens in static schemes such as Thorup-Zwick [45], we now have the issue that a node might want to send a packet to a node that has been deleted in G_t, hence we need a mechanism to report that a packet could not be delivered. To achieve that, the header of a packet now is defined as follows: when a node s wants to send a packet to t, it sends it with the header $((t, L(t) \cdot (s, L(s)))$. When running CompactFTZ, each node considers only the first pair.

When a node v receives a message M with a header containing two pairs, it proceeds as follows to detect an error i.e. a non-deliverable. The following conditions suggest to v that the receiver t has been deleted and the packet is non-deliverable:

1. *If v is a leaf (real node) and $v \neq t$:* This is a dead end since the packet cannot traverse further. This implies that t must have been in the subtree of v but the subtree of v is now empty.

2. *If v is a non-leaf node but there is no node at the port it should forward to:* Similar to above, it indicates that v's subtree involving t is empty.

3. *If u sent the packet to v but according to the routing rules, v should send the packet back to u:* This happens when v is a helper node which is part of $\mathrm{RT}(t)$ or $\mathrm{RT}(x)$ where x was not on the path $s - t$ in T_0. Node v will receive the message either on the way up (towards the root) or way down (from the root). In either case, if v is part of $\mathrm{RT}(t)$, due to the dfs numbering, it would have to return M to u. Another possibility is that due to a number of deletions $\mathrm{RT}(t)$ has disappeared but then x would either be an ascendent (if M is on the

way up) or x would be a descendent of t (if M is on the way down). Either way, the DFS numbering would indicate to v that it has to return the message to u.

If a target deletion has been detected due to the above rules, v removes the first pair of the header and sends back M to the node it got M from (with the header now only having $(s, L(s))$. When a node v receives a message M with a header containing only one pair, it proceeds as before and applies the same rules discussed previously. This time, a non-delivery condition however implies that the source has been removed too, and, therefore M can be discarded from the system. This ensures that 'zombie' or undeliverable messages do not clog the system.

4.3.1 About stretch

The stretch of a routing scheme A, denoted $\lambda(A, G)$, is the minimum λ such that $r(s,t) \leq \lambda\, dist(s,t)$ for every pair of nodes s, t, where $dist(s,t)$ is the distance between s and t in the graph G and $r(s,t)$ is the length of the path in G the scheme uses for routing a message from s to t.

The stretch $\lambda(\text{CompactFTZ}, T_0)$ is 1: for any pair of nodes, TZ routes a message through the unique path in the tree between them. Similarly, the stretch $\lambda(\text{CompactFTZ}, T_t)$ is 1: each node that is deleted is replaced with a binary tree structure R, and the nodes in it perform a binary search, hence a message passing through R follows the shortest path from the root to a leaf, or vice versa.

The stretch of CompactFTZ is different when we consider G_t. First note that the stretch $\lambda(\text{CompactFTZ}, G_0)$ might be of order $\Theta(n)$ since a spanning tree of a graph may blow up the distances by that much. Since $\lambda(\text{CompactFTZ}, T_0) = 1$, it follows that $\delta_T(u,w) \leq \lambda(\text{CompactFTZ}, G_0) \cdot \delta_G(u,w)$, where $\delta_T(u,w)$ is the distance between u and w in T_0 and $\delta_G(u,w)$ is the distance between u and w in G_0. Theorem 4.2 states that, for routing a message from u to w, CompactFTZ uses a path in T_t of size at most $\delta_T(u,w)+y(\log\Delta-1)$, where $y \leq t$ is the number of non-leaf nodes deleted to get T_t. The $y(\log\Delta - 1)$ additive factor in the expression is because each deleted non-leaf node is replaced with a binary tree, whose height is $O(\log\Delta)$. In the worst case, that happens for all y binary trees for a given message, which implies that $\lambda(\text{CompactFTZ}, G_t) \leq y(\log\Delta - 1) \cdot \lambda(\text{CompactFTZ}, G_0)$ (since CompactFTZ only uses the tree for routing).

5. EXTENSIONS AND CONCLUSION

This paper presented, to our knowledge, the first compact self-healing algorithm and also the first self-healing compact routing scheme. We have not considered the memory costs involved in the preprocessing but we believe that it should be possible to set up the data structures in a distributed compact manner: this needs to be investigated. The current paper focuses only on node deletions,. Can we devise a self-healing compact routing scheme working in a fully dynamic scenario with both (node and edge) insertions and deletions? The challenges reside in dealing with the expanding out-degree efficiently.

The current paper allows to add additional links to nearby nodes in an overlay manner. What should the model be of losing links without losing nodes? How will it affect the algorithms appearing in this paper?

6. REFERENCES

[1] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* John Wiley & Sons, 2004.

[2] B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg. Improved routing strategies with succinct tables. *J. Algorithms*, 11(3):307–341, 1990.

[3] B. Awerbuch, O. Goldreich, D. Peleg, and R. Vainish. A trade-off between information and communication in broadcast protocols. *J. ACM*, 37(2):238–256, 1990.

[4] S. Baswana and S. Sen. A simple linear time algorithm for computing a (2k-1)-spanner of o($n^{1+1/k}$) size in weighted graphs. In *ICALP*, pages 384–296, 2003.

[5] J. Beauquier, J. Burman, and S. Kutten. A self-stabilizing transformer for population protocols with covering. *Theor. Comput. Sci.*, 412(33):4247–4259, 2011.

[6] A. Berns and S. Ghosh. Dissecting self-* properties. *Self-Adaptive and Self-Organizing Systems, International Conference on*, 0:10–19, 2009.

[7] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. *Wireless Networks*, 7(6):609–616, 2001.

[8] S. Chechik. Compact routing schemes with improved stretch. In *ACM Symposium on Principles of Distributed Computing, PODC '13*, pages 33–41, 2013.

[9] S. Chechik. Fault-tolerant compact routing schemes for general graphs. *Inf. Comput.*, 222:36–44, 2013.

[10] S. Chechik, M. Langberg, D. Peleg, and L. Roditty. f-sensitivity distance oracles and routing schemes. *Algorithmica*, 63(4):861–882, 2012.

[11] B. Courcelle and A. Twigg. Compact forbidden-set routing. In *STACS 2007, Proceedings*, pages 37–48, 2007.

[12] L. Cowen. Compact routing with minimum stretch. *J. Algorithms*, 38(1):170–183, 2001.

[13] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, November 1974.

[14] S. Dolev. *Self-stabilization.* MIT Press, Cambridge, MA, USA, 2000.

[15] S. Dolev and N. Tzachar. Empire of colonies: Self-stabilizing and self-organizing distributed algorithm. *Theor. Comput. Sci.*, 410(6-7):514–532, 2009.

[16] R. D. Doverspike and B. Wilson. Comparison of capacity efficiency of dcs network restoration routing techniques. *J. Network Syst. Manage.*, 2(2), 1994.

[17] M. Elkin. A near-optimal distributed fully dynamic algorithm for maintaining sparse spanners. In I. Gupta and R. Wattenhofer, editors, *PODC*, pages 185–194. ACM, 2007.

[18] X. Feng and C. Han. A fault-tolerant routing scheme in dynamic networks. *J. Comput. Sci. Technol.*, 16(4):371–380, 2001.

[19] P. Fraigniaud and C. Gavoille. Routing in trees. In *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Proceedings*, pages 757–772, 2001.

[20] P. Fraigniaud and C. Gavoille. A space lower bound for routing in trees. In H. Alt and A. Ferreira, editors,

STACS 2002, Proceedings, volume 2285 of *Lecture Notes in Computer Science*, pages 65–75. Springer, 2002.

[21] M. Fraser, E. Kranakis, and J. Urrutia. Memory requirements for local geometric routing and traversal in digraphs. In *Proceedings of the 20th Annual Canadian Conference on Computational Geometry, Montréal, Canada, August 13-15, 2008*, 2008.

[22] T. Frisanco. Optimal spare capacity design for various protection switching methods in ATM networks. In *Communications, 1997. ICC 97 Montreal, 'Towards the Knowledge Millennium'. 1997 IEEE International Conference on*, volume 1, pages 293–298, 1997.

[23] D. Ghosh, R. Sharman, H. Raghav Rao, and S. Upadhyaya. Self-healing systems - survey and synthesis. *Decis. Support Syst.*, 42(4):2164–2185, 2007.

[24] T. Hayes, N. Rustagi, J. Saia, and A. Trehan. The forgiving tree: a self-healing distributed data structure. In *PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 203–212, New York, NY, USA, 2008. ACM.

[25] T. P. Hayes, J. Saia, and A. Trehan. The forgiving graph: a distributed data structure for low stretch under adversarial attack. *Distributed Computing*, pages 1–18, 2012.

[26] R. R. Iraschko, M. H. MacGregor, and W. D. Grover. Optimal capacity placement for path restoration in STM or ATM mesh-survivable networks. *IEEE/ACM Trans. Netw.*, 6(3):325–336, 1998.

[27] A. Korman. General compact labeling schemes for dynamic trees. *Distributed Computing*, 20(3):179–193, 2007.

[28] A. Korman, S. Kutten, and T. Masuzawa. Fast and compact self stabilizing verification, computation, and fault detection of an MST. In *PODC*, pages 311–320, 2011.

[29] A. Korman and D. Peleg. Labeling schemes for weighted dynamic trees. *Inf. Comput.*, 205(12):1721–1740, 2007.

[30] A. Korman, D. Peleg, and Y. Rodeh. Labeling schemes for dynamic tree networks. *Theory Comput. Syst.*, 37(1):49–75, 2004.

[31] E. Kranakis, H. Singh, and J. Urrutia. Compass routing on geometric networks. In *Proceedings of the 11th Canadian Conference on Computational Geometry, UBC, Vancouver, British Columbia, Canada, August 15-18, 1999*, 1999.

[32] F. Kuhn, N. Lynch, and R. Oshman. Distributed computation in dynamic networks. In *Proceedings of the 42nd ACM symposium on Theory of computing*, STOC '10, pages 513–522, New York, NY, USA, 2010. ACM.

[33] F. Kuhn, S. Schmid, and R. Wattenhofer. A Self-Repairing Peer-to-Peer System Resilient to Dynamic Adversarial Churn. In *4th International Workshop on Peer-To-Peer Systems (IPTPS), Cornell University, Ithaca, New York, USA, Springer LNCS 3640*, February 2005.

[34] S. Kutten and A. Porat. Maintenance of a spanning tree in dynamic networks. In P. Jayanti, editor, *Distributed Computing*, volume 1693 of *Lecture Notes in Computer Science*, pages 846–846. Springer Berlin / Heidelberg, 1999.

[35] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[36] K. Murakami and H. S. Kim. Comparative study on restoration schemes of survivable ATM networks. In *INFOCOM*, pages 345–352, 1997.

[37] F. Orejas, P. G. Spirakis, and J. van Leeuwen, editors. *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings*, volume 2076 of *Lecture Notes in Computer Science*. Springer, 2001.

[38] G. Pandurangan, P. Robinson, and A. Trehan. Dex: Self-healing expanders. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 702–711, Washington, DC, USA, 2014. IEEE Computer Society.

[39] G. Pandurangan and A. Trehan. Xheal: a localized self-healing algorithm using expanders. *Distributed Computing*, 27(1):39–54, 2014.

[40] D. Peleg and E. Upfal. A tradeoff between space and efficiency for routing tables (extended abstract). In J. Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 43–52. ACM, 1988.

[41] R. Poor, C. Bowman, and C. B. Auburn. Self-healing networks. *Queue*, 1:52–59, May 2003.

[42] G. Saad and J. Saia. Self-healing computation. In P. Felber and V. K. Garg, editors, *Stabilization, Safety, and Security of Distributed Systems, SSS 2014, Proceedings*, volume 8756 of *Lecture Notes in Computer Science*, pages 195–210. Springer, 2014.

[43] J. Saia and A. Trehan. Picking up the pieces: Self-healing in reconfigurable networks. In *IPDPS. 22nd IEEE International Symposium on Parallel and Distributed Processing.*, pages 1–12. IEEE, April 2008.

[44] N. Santoro and R. Khatib. Labelling and implicit routing in networks. *The computer journal*, 28(1):5–8, 1985.

[45] M. Thorup and U. Zwick. Compact routing schemes. In *SPAA*, pages 1–10, 2001.

[46] A. Trehan. *Algorithms for self-healing networks.* Dissertation, University of New Mexico, 2010.

[47] B. van Caenegem, N. Wauters, and P. Demeester. Spare capacity assignment for different restoration strategies in mesh survivable networks. In *Communications, 1997. ICC 97 Montreal, 'Towards the Knowledge Millennium'. 1997 IEEE International Conference on*, volume 1, pages 288–292, 1997.

[48] J. Vounckx, G. Deconinck, R. Lauwereins, and J. A. Peperstraete. Fault-tolerant compact routing based on reduced structural information in wormhole-switching based networks. In *Structural Information and Communication Complexity, 1st International Colloquium, SIROCCO 1994, Proceedings*, pages 125–148, 1994.

[49] Y. Xiong and L. G. Mason. Restoration strategies and spare capacity requirements in self-healing ATM networks. *IEEE/ACM Trans. Netw.*, 7(1):98–110, 1999.