Compact routing messages in self-healing trees

PLEASE CITE THE PUBLISHED VERSION

http://dx.doi.org/10.1016/j.tcs.2016.11.022

PUBLISHER

© Elsevier

VERSION

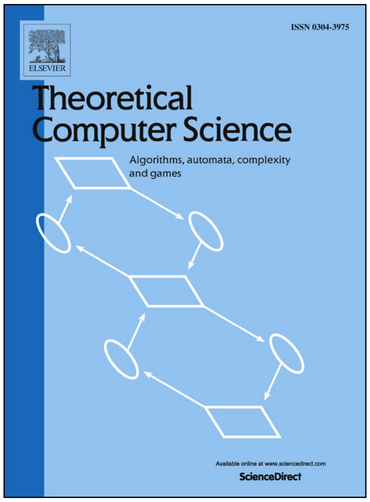
AM (Accepted Manuscript)

REPOSITORY RECORD

Castaneda, Armando, Danny Dolev, and Amitabh Trehan. 2016. “Compact Routing Messages in Self-healing Trees”. Loughborough University. https://hdl.handle.net/2134/24632.

Compact routing messages in self-healing trees

Armando Castañeda, Danny Dolev, Amitabh Trehan


ISSN 0304-3975
ELSEVIER
Theoretical Computer Science
Algorithms, automata, complexity and games
Available online at www.sciencedirect.com
ScienceDirect

PII: S0304-3975(16)30681-8
DOI: http://dx.doi.org/10.1016/j.tcs.2016.11.022
Reference: TCS 10987

To appear in: *Theoretical Computer Science*

Received date: 7 May 2016
Revised date: 15 November 2016
Accepted date: 16 November 2016

Please cite this article in press as: A. Castañeda et al., Compact routing messages in self-healing trees, *Theoret. Comput. Sci.* (2016), http://dx.doi.org/10.1016/j.tcs.2016.11.022

Highlights

- First self-healing compact routing algorithm for low memory nodes.
- Introduces CompactFT: a compact (O(polylog n) local memory) version of ForgivingTree [PODC2008].
- Introduces CompactFTZ: self-healing compact routing scheme delivering messages despite node failures in a distributed network.

Compact Routing Messages in Self-Healing Trees

Armando Castañeda[a,1], Danny Dolev[b,2], Amitabh Trehan[c,3,*]

[a]Instituto de Matemáticas, UNAM, México
[b]The Hebrew University of Jerusalem, Israel.
[c]School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, 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 set up and started. This paper presents, to our knowledge, the first self-healing compact routing scheme. Besides, our schemes are developed for low memory nodes and are compact schemes, meaning they require only $O(\log^2 n)$ bits memory.

We introduce two algorithms of independent interest: The first is *CompactFT*, a novel compact version of the self-healing algorithm Forgiving Tree of Hayes et al. [PODC 2008] that uses only $O(\log n)$ bits local memory. The second algorithm (*CompactFTZ*) combines CompactFT with Thorup-Zwick's tree-based compact routing scheme [SPAA 2001] to produce a compact self-healing routing scheme. In the self-healing model, the adversary deletes nodes one at a time and the affected nodes self-heal locally by adding few edges. We introduce the *bounded-memory self-healing model*, where the memory each node need to use for the self-healing algorithm is bounded. 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 maxi-

---

[*] Corresponding author

Email addresses: `armando.castaneda@im.unam.mx` (Armando Castañeda), `danny.dolev@mail.huji.ac.il` (Danny Dolev), `amitabh.trehaan@gmail.com` (Amitabh Trehan)

[1] Supported partially by PAPIIT-UNAM IA101015. This research was partially done while the first author was at the Department of Computer Science of the Technion, supported by an Aly Kaufman post-doctoral fellowship.

[2] 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 work was supported by the HUJI Cyber Security Research Center in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office, and by The Israeli Centers of Research Excellence (I-CORE) program, (Center No. 4/11).

[3] Research partly done as an I-CORE fellow supported by The Israeli Centers of Research Excellence (I-CORE) program, (Center No. 4/11). This project has also received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671603 (AllScale).

Preprint submitted to Theoretical Computer Science *November 15, 2016*

mum 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 is removed from the network. CompactFTZ uses only $O(\log n)$ bits memory for local fields (such as routing tables) and $O(\log^2 n)$ bits for the routing labels, thus requiring $O(\log^2 n)$ bits overall.

Keywords: Self-Healing fault-tolerant P2P algorithm, Distributed message passing algorithm, Compact Thorup-Zwick routing, Low memory Compact algorithm, Compact Forgiving Tree, Internet of Things

---

1. Introduction

Routing protocols have been the focus of intensive research over the years. Efficient and robust routing is critical in current networks, and will be even more so in future networks. Routing is based on information carried by the traveling packets and data structures that are maintained at 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 makes the use of centralized decisions very difficult, and we are close to giving up on maintaining long distance routing decisions. We are a few years before a full-scale deployment of the Internet of Things (IOT), which will introduce billions of very weak devices that need to have routing capabilities. The size of the network and the dynamic structure that will evolve will force focusing on local decisions, pushing for the use of protocols that do not require maintaining huge routing tables.

Santoro and Khatib [1], Peleg and Upfal [2], and Cowen [3] 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 in the length of the routing. Several papers followed up with some improvements on the schemes (cf. Thorup and Zwick [4], Fraigniaud and Gavoille [5], and Chechik [6]). These efficient routing schemes remain stable as long as there are no changes to the network.

The scale and mobility of future networks such as IOT will necessarily lead to continuous changes to the network and regular (possibly isolated) failures of components/devices. Thus, maintaining connectivity and ensuring a smooth running of already running protocols will be important. This is where the self-healing capabilities of the network will be important for efficient, responsive repair of the network without hindering regular operation.

The target of the current paper is to introduce an efficient *compact* scheme that combines small local memory and packet headers with the ability to update the local data structures stored at each node in a response to a change of the network. Throughout this paper, when we say compact, we imply schemes that use $o(n)$ local memory per node and packet headers of size $o(n)$, where n is the number of nodes in the network. Our new scheme has similar cost as previous compact routing schemes. In this work, we will focus on node failures. In general, node failures may be considered more challenging to handle than edge/link failures, and are analysed less often.

Our algorithms work in the *bounded memory self-healing model* (Section 2). 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. After the preprocessing is complete, 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. The adversary can attack only after the system has self-healed itself from the previous attack. Thus, the self-healing process should be efficient enough so that the system recovers quickly after any possible attack.

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 a 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 is the maximum degree of the initial spanning tree built on the graph. Moreover, CompactFT needs only $O(\log n)$ local memory. Theorem 4.1 states the results formally.

CompactFTZ, our compact routing algorithm, is based on the compact routing scheme on trees by Thorup and Zwick [4], 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 $O(\log n)$ 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 6.2 states the results formally.

In CompactFTZ, we use a slight variant of Thorup and Zwick. This variant is for no particular reason but to exemplify that our construction to obtain CompactFTZ is not tailor-made for a particular routing protocol but for any *direct* compact routing protocol for trees based on a DFS labeling (as Thorup and Zwick is). A routing protocol is *direct* if the header of a packet is not modified during the routing. Instead of using a concrete protocol, we could have considered an abstract compact routing protocol for trees and derive our construction, however, we use a concrete example to make the exposition concise. The same approach applies to the compact routing algorithm for trees of Fraigniaud and Gavoille [5], or any DFS-based interval routing for trees (see for example [1]), although the latter does not necessarily achieve compactness.

1.1. Related Work

Algorithm		Over Complete Run		Per Healing Phase		
	Local Memory	Diameter (Orig: D)	Degree (Orig: d)	Parallel Repair Time	Msg Size	# Msges
Forg. Tree [7]	$O(n)$	$D\log\Delta^\dagger$	$d+3$	$O(1)$	$O(\log n)$	$O(1)$
CompactFT	$O(\log n)$	$D\log\Delta^\dagger$	$d+3$	$O(1)$	$O(\log n)$	$O(\delta)^\ddagger$

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

Table 1: Comparing CompactFT with Forgiving Tree

CompactFT uses ideas from the *Forgiving Tree* [7] (FT, in short) 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 between the algorithms.

Several papers have studied the routing problem in arbitrary networks (e.g. [8, 9, 3, 6]) and with the help of geographic information (e.g. [10, 11, 12]), 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. [13] presents labelling schemes for weighted dynamic trees where node weights can change. In a labelling scheme, each node has a label and from every two labels, the distance between the corresponding nodes can be easily computed. However, they do not deal with node deletions nor do they claim to deal with routing. In [14], 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. 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 [5, 15, 4]. These schemes have a multiplicative overhead factor of $\Omega(\log n)$ on the label sizes of the static schemes. In [16], Korman improves the results in [14], presenting a labelling scheme in the same model that allows computing any function on pairs of vertices with smaller labels, at the cost, in some cases, of communication. Our work differs from the previous papers 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 [17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27]. 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 [7, 28]. A variety of self-healing topology maintenance algorithms have been devised in the self-healing models [29, 30, 31, 32, 33, 34]. One of the central techniques used in these topological self-healing algorithms is the concept of a reconstruction structure (also used in this paper and described in Section 3). In previous work, reconstruction structures have been balanced binary search trees (BBST) as in the present paper and [7], half-full trees (a special kind of bbst) in [31], random r-regular expanders as in [30], or p-cycle deterministic expanders in [29].

Our paper moves in the direction of self-healing computation/routing along with topology, which is attempted in other papers, e.g., [35] (though in a different model). Finally, dynamic network topology and fault tolerance are core concerns of distributed computing [36, 37] and various models, e.g., [38], and topology maintenance and self-* algorithms abound [39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50].

2. Model of Computation

This section describes our model, which can be seen as a combination of the self-healing (Section 2.1) and the routing models (Section 2.2). Though the models are described independently, the final model should be seen as a union of both the models.

2.1. Compact Self-Healing Model

Let $G = G_0$ be an arbitrary connected graph on n nodes, which represent processors in a distributed network. Each node has a unique and private ID. The edges of G_0 represent direct communication lines between nodes: if there is an edge between u and v in G_0, there is a bidirectional communication line between u and v. The port numbers of each processor v are labeled $0, \ldots, \deg(v) - 1$. Initially, each node only knows the number of neighbors it has in G_0 as it knows the number of ports it is connected to, however, it does not know the ID of its neighbors. Namely, for each port, it knows it has a neighbor at the other extreme of that communication line but it does not know the ID of that node. Moreover, each node is unaware of the structure of the rest of G_0.

In the model, any algorithm proceeds in a sequence of phases where each phase begins with a powerful, *omniscient* (aware of all information including

Model 2.1 The memory-bound Self-healing Delete and Repair Model.

```
for $t := 1$ to $n$ do
    Adversary deletes a node $v_t$ from $G_{t-1}$, forming $H_t$.
    All neighbors of $v_t$ are informed of the deletion. Other nodes may be unaware of the deletion unless explicitly informed by another node.
    Recovery phase:
    Nodes of $H_t$ may communicate (asynchronously and concurrently) with their immediate neighbors. These messages are never lost or corrupted, and may contain the names of other vertices.
    During this phase, each node may insert edges joining it to any other node 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 phase ends.
end for
Success metrics: Minimize the following "complexity" measures:
    1. Graph properties/Invariants:
        (a) Degree increase: $\max_{t<n} \max_v(\deg(v, G_t) - \deg(v, G_0))$
        (b) Diameter stretch: $\max_{t<n} \mathrm{Dia}(G_t)/\mathrm{Dia}(G_0)$
    2. Communication per node: The maximum number of bits sent by a single node in a single recovery phase
    3. Recovery time: The maximum total time for a recovery phase; the number of (synchronous or asynchronous) rounds required for a recovery phase.
    4. Local Memory: The amount of memory a single node needs to run the algorithm.
```

the algorithm and node's private data) adversary deleting a node (i.e. crashing a process) and the network reacting in a distributed manner by individual nodes (processors) communicating and adding edges (connections) between themselves. This is described in Model 2.1.

Communication between nodes is by sending messages. Messages are not lost. The network is *phasewise asynchronous*, *i.e.,* each message is delivered in some finite but arbitrary time but all messages of a phase are received and processed before the next phase begins, i.e., before the next deletion occurs. For ease of description, we shall initially describe our algorithms in a *synchronous* manner. Each synchronous *round* consists of nodes processing messages received in previous rounds, generating and sending messages to their neighbours, which are received without loss by the end of the round. It is assumed that the adversary knows the algorithm.

We allow a certain amount of preprocessing to be done before the first deletion occurs. This is, for instance, used by the nodes to gather topological information about G_0, and to setup the data structures for the self-healing routing algorithm. Nodes are aware of the end of the preprocessing. For this work, we do not assume any constraints on the resources for preprocessing and do not count the cost of it towards our algorithm performance. It seems reasonable to assume that the devices and network using these algorithms will have access to more computing resources initially when they are being deployed. This can enable the preprocessing to be done quickly or even centrally and externally. Thus, this allows us to concentrate on the 'complexity' of self-healing, as follows.

With reference to Model 2.1, the objective is to minimize the following "complexity" measures (once the preprocessing stage is over):

- **Degree increase:** $\max_{t<n} \max_v (\deg(v, G_t) - \deg(v, G_0))$. This corresponds to the additional load a node needs to take due to self-healing.
- **Diameter stretch:** $\max_{t<n} \mathrm{Dia}(G_t)/\mathrm{Dia}(G_0)$. This measure corresponds to the increase in latency the self-healing algorithm will enforce upon the network. Note that it may be simple to contain the increase in node degrees or a distance bound but challenging to do both simultaneously. For instance, imagine connecting all the neighbors of deleted node as a clique.
- **Communication per node:** The maximum number of bits sent by a single node in a single recovery phase.
- **Recovery time:** The number of (synchronous or asynchronous) rounds required for a recovery phase.
- **Local Memory:** The amount of memory a single node needs to run the algorithm. As stated, we would like this to be $o(n)$ (to be able to claim our algorithms to be compact). In fact, we claim the algorithms we introduce in this paper to need $O(\log^2 n)$ local memory.

Notice that, in our model, the adversary is restricted to act (and begins a new phase) only after the algorithm finishes the repair. This makes it incumbent for the self-healing algorithm to finish the repair in a reasonable time to be useful in a realistic setting. An alternate and almost similar model would be to state a parameter, say τ, and state that we consider only algorithms which have repair time at most τ. In either case, we are looking for algorithms which minimise the self-healing repair time while fulfilling the other requirements.

We assume that port numbers are *reusable*, meaning that, in a healing phase, if there is no live node attached to a port #*port* of a live node v, then v is free to create a new edge with another live node and to assign to this edge the port number #*port*.

2.2. Compact Routing Model

As in the compact self-healing model, the routing algorithm is allowed a preprocessing phase, e.g., to run a distributed DFS on the graph. Each message has a *label* that may contain the node ID and other information derived from the preprocessing phase. Every node stores some local information, i.e. *routing tables*, for routing. The preprocessing is not allowed to change the original port assignment at any node (however, node deletions may force the self-healing algorithm to do a simple ports' reassignment). We are interested in minimizing the sizes of the label and the local information at each node. As is standard in routing models, we assume a centralised entity that has the labels of all nodes and if node v would like to send a message to another node w, it obtains the label of w from this global entity. Thus, locally, each node of the routing algorithm only stores its routing tables.

3. The Algorithms: High Level

As stated, CompactFT (Algorithm 4.1) is an adaptation of FT [7] for low memory. CompactFTZ (Algorithm 6.2) 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 a careful setup of CompactFT and CompactFTZ fields (Tables 2 and 4). For CompactFT, every node sets up and distributes a *will* (Section 4), which is the blueprint of edges and virtual (*helper*) nodes to be constructed in case of the deletion of that node.
- After each deletion, the repair maintains the spanning tree of *helper* (virtual) 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:

  (i) *non-leaf deletion, i.e., v_i is not a leaf in T_{i-1}* (Section 4.1): The neighbours of v_i 'execute' v_i's *will*, leading v_i to be replaced by a *reconstruction tree* (RT(v_i)). As mentioned, reconstruction structures are a central technique behind topological self-healing and are introduced in FT [7], and described in more detail in [31], Section 3. RT(v_i), in the present algorithm as in CompactFT, is a balanced binary search tree (*BBST*) with an additional node (Figure 1). The children of v_i (which are real nodes) form the leaves. The helper nodes form the internal nodes of the tree and are simulated by v_i's neighbours. The labelling of the internal nodes (which is the ID of the leaf node simulating them) is done so that the entire tree forms a BBST. The 'rightmost' node among the leaves is a special node called the *heir* of v and contributes another internal node. As the name suggests, the heir is responsible for taking over v's simulations (if any) when v is deleted (Section 4).

  (ii) *leaf deletion, i.e., v_i is a leaf in T_{i-1}* (Section 4.2): 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

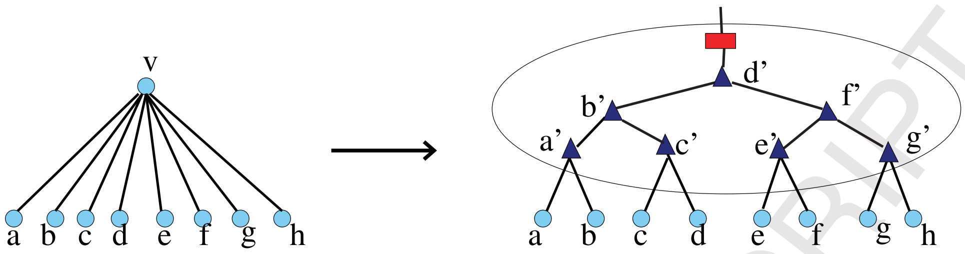


Figure 1: Deleted node v replaced by Reconstruction Tree (RT(v)). Nodes in oval are virtual helper nodes. The circles are regular helper nodes and the rectangle is an 'heir' helper node. The 'Will' of v is RT(v), i.e., the structure that replaces the deleted v.

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 what is 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 would make it to the right node at the end of RT(w) (either a leaf node or the root, which are real nodes). The message eventually reaches r, but if r is dead, the message is 'returned to sender' using s's label.

4. 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 4.1.

As stated, in CompactFT, a deleted node v is replaced by a RT(v) 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., 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 of RT(v) and is distributed among v's children. Each part of a node's *will* stored in another node is called a *willportion.* We denote *willportion*(v, w) to be the part of *will*(v) that involves w, i.e., the relevant subgraph, and is stored by node w. When v is a leaf node, however, *will*(v) differs in not being an 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*. The *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*.

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 $nextparent(v), <nextparent(v)>$.

The fields used by a node for executing CompactFT are given in Table 2. Unlike FT, a node cannot have either the list of its children or its own RT (since these can be as large as $\Omega(n)$). 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 4.1 gives a high level overview of CompactFT. The detailed algorithms are presented in Section 5. Also, Table 3 describes some of the special messages used by CompactFT. The algorithm begins with a preprocessing phase (Algorithm 4.1 line 1) in which a rooted BFS spanning tree of the network from an arbitrary node is computed. CompactFT 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

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 *willportion*) at v's port $<x>$, node x with node h.
PtWillConnection $((y, <y>), (z, <z>))$	Node v asks receivers (in their $v.\textit{willportion}$) 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).

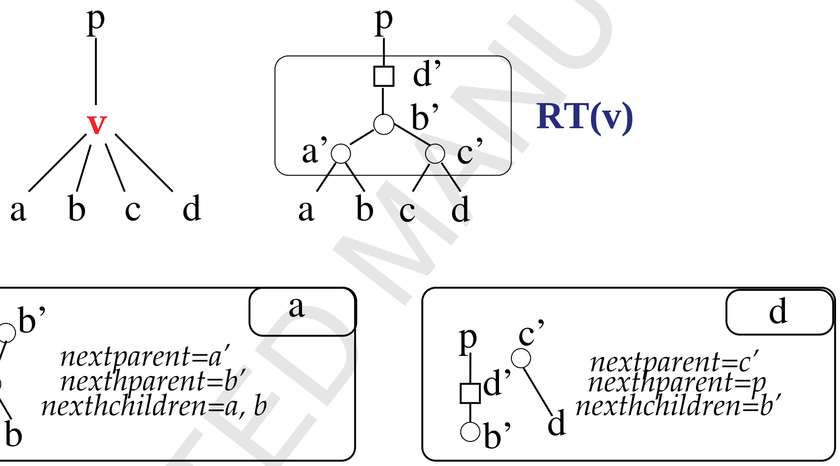


Figure 2: A node v and its neighbourhood. v's *will* RT(v) and the *willportions* for its children, a and d are shown. Note that d is the *heir* of v.

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 largest degree in the original graph (ref: Theorem 4.1). 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 4.1 CompactFT(Graph G): High level view

```
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).
while true do
  if a vertex $x$ is deleted then
    if $x$ was not a leaf (i.e., had any children) then // Fix non leaf deletion.
      $x$'s children execute $x$'s Will using $x$'s willportions they have; heir($x$) takes over $x$'s Will/duties.
      All affected Wills (i.e. neighbours of $x$ and of helper($x$)) are updated by simple update of relevant willportions.
    else // Fix leaf deletion.
      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
      if $p$ is real/alive then // Update Wills by simulating the deletion of $p$ and $x$
        if $x$ was $p$'s only child then
          $p$ computes its leafheir and leafwill and forwards it. // $p$ has become a leaf
        else
          $p$ informs all children about $x$'s deletion.
          $p$'s children update $p$'s willportions using $x$'s leafwillportions.
          Children issue updates to $p$'s willportions and other leafwillportions via $p$.
          $p$ forwards updates via broadcast or point-to-point messages, as required.
          $p$'s neighbours receiving these messages update their data structures.
        end if
      else // $p$ had already been deleted earlier.
        Let $y$ be $x$'s leafheir.
        $y$ executes $x$'s will.
        Affected nodes update their and their neighbour's willportions.
      end if
    end if
    if $x$ was node $z$'s leafheir then
      $z$ sets a new neighbour as leafheir following a simple rule.
    end if
  end if
end while
```

4.1. Deletion of a Non-Leaf Node:

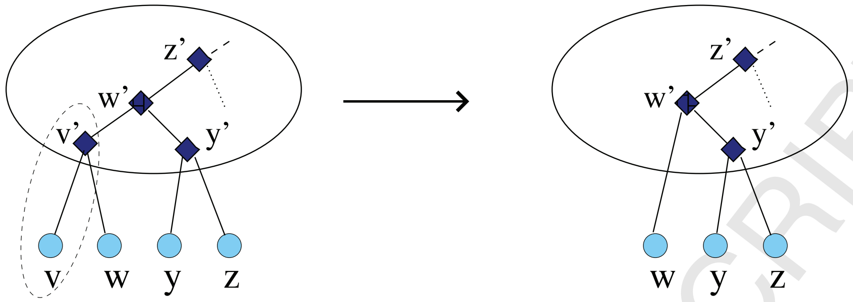
Assume that a node x is deleted. If x was not a leaf node (Algorithm 4.1 lines 5 - 6), it's neighbours simply execute x's *will*. One of x's children (by default the rightmost child) is a special child called the *heir* (say, h) and it takes over any virtual node (i.e., *helper*(x)) that x may have been 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).

4.2. Deletion of a Leaf Node:

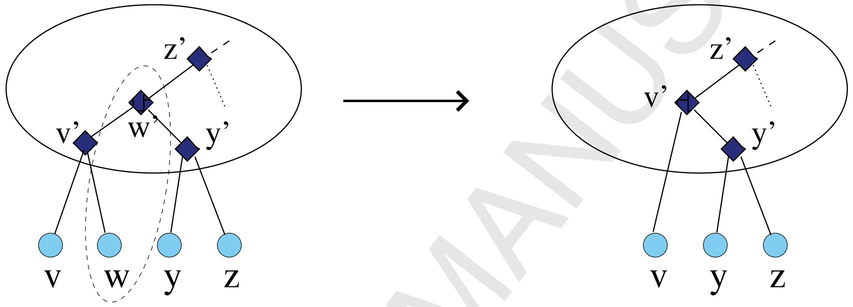
If the deleted node x was, in fact, a leaf node, the situation is more involved. There are two cases to consider: whether the parent p of x is a helper node (implying the original parent had been deleted earlier) or a real node. The second case, though trivial in FT (with $O(n)$ memory) is challenging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

bypass(x): (from [7]) *Precondition*: $|hchildren(x)| = 1$, *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.
$hparent(x) \leftarrow EMPTY$; $hchildren(x) \leftarrow EMPTY$.

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



(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.

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 having node p send to its ports $<a>$ and $<b>$ a PtWillConnection($(a, <a>), (b, <b>)$) message. 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 establish 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.* 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 4.1 summarises the properties of CompactFT. The detailed algorithms (pseudocodes) and proofs are deferred to Section 5.

Theorem 4.1. *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.*

5. Compact Forgiving Trees - Detailed Algorithms and Proofs

In this section we give the detailed pseudocodes for CompactFT (Algorithm 5.1 to Algorithm 5.4) and proofs for the main theorem (Theorem 4.1) and of the required lemmas.

Algorithm 5.1 CompactFT(Graph G): Main function

```
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 port number at which $x$ is connected to its parent (if any).
Each node only has a willportion and/or leafwillportions (constant sized (in
number of node $IDs$) portion of parent's will or sibling's will respectively).
The neighbours (parent and children) of $v$ are attached at numchildren($v$)
ports from port 0 to port maxportnumber($v$)
while true do
  if a vertex $x$ is deleted then
    if numchildren($x$) $> 0$ then
      FixNonLeafDeletion($x$)
    else
      FixLeafDeletion($x$)
    end if
  end if
end while
```

Lemma 5.1. *In* CompactFT*, a real node simulates at most one helper node at a time.*

Proof. This follows from the construction of the algorithm. If deleted node x was a non-leaf node, it is substituted by RT(x) (Figure 1). RT(x) is like a balanced binary tree such that the leaves are the real children of x and each internal node is a virtual helper node. Also, there are exactly the same number of internal nodes as leaf nodes and each leaf node simulates exactly one helper node. This is the only time in the algorithm that a helper node is created. At other times, such as during leaf deletion or as a heir, a node may simulate a

Algorithm 5.2 FixNonLeafDeletion(x): Self-healing on deletion of internal node

```
while true do
    for every child $v$ of $x$ (if exists) do
        Execute the *will* of $x$ using the $O(\log n)$ sized *willportion*$(d, v)$. // i.e., make the connections given by *willportion*$(x, v)$
    end for
    for parent $p$ of $x$ (if it exists) do
        $p$ will be contacted by heir of $x$, say $h$ to open a new connection to $h$ at port of deleted node, port $<x>$.
        $p$ will be informed by $h$ that the ID of the deleted node was $d$
        Send BrNodeReplace$((x, <x>), (h, <x>))$ message to every neighbour.
    end for
    if node $v$ receives message BrNodeReplace$((x, <x>), (h, <x>))$ from parent $p$ then
        $v$ replaces every occurrence of node $x$ with node $h$ in its *willportion*$(x, v)$.
    end if
end while
```

different node but this only happens if the node relinquishes its previous helper node. Thus, a real node simulates at most one helper node at any time. □

Lemma 5.2. *The* CompactFT *increases the degree of any vertex by at most 3.*

Proof. Since the degree of any node in a binary tree is at most 3, this lemma follows from Lemma 5.1. □

Lemma 5.3. *The* CompactFT*'s diameter is bounded by $O(D \log \Delta)$, where D is the diameter and Δ the highest degree of a node in the original graph (G_0).*

Proof. This also follows from the construction of the algorithm. The initial spanning tree T_0 is a BFS spanning tree of G_0; thus, the diameter of T_0 may be at most 2 times that of G_0. Consider the deletion of a non-leaf node x of degree d. x is replaced by RT(x) (Figure 1). Since RT(x) is a balanced binary tree (with an additional node), the largest distance in this tree is $\log d$. Two RTs never merge, thus, this RT cannot grow. A leaf deletion can only reduce the number of nodes in a RT, thus, reducing distances. Consider a path which defined the diameter D in G_0. In the worst case, every non-leaf node on this path was deleted and replaced by a RT. Thus, this path can be of length at most $O(D \log \Delta)$ where Δ is the maximum possible value of d. □

Lemma 5.4. *In* CompactFT*, a node may be* leafheir *for at most two leaf nodes.*

Proof. For contradiction, consider a node y that is *leafheir*(u), *leafheir*(v) and *leafheir*(w) for three leaf nodes u, v and w all of whose parent is node p. From the construction of the algorithm, v's *leafheir* can either be the parent of *real*(v)

Algorithm 5.3 FixLeafDeletion(x): Self-healing on deletion of leaf node

```
Let $p = parent(x)$
if $p$ is a real node then // $p$ has not been deleted yet
    $p$ broadcasts BrLeafLost(<$x$>) // $p$ does not know $ID$ of $d$, only the port number <$d$>
    if node $v$ receives a BrLeafLost(<$x$>) message then
        $v$.UpdateLeafWillPortion($p$, <$x$>)// Use <$x$>'s leafwill to update $p$'s will by updating willportions using broadcast($Br*$ messages) and/or point-to-point($Pt*$ messages)
    end if
else // The real parent of $x$ was already deleted earlier
    if node $v$ is <$x$>'s leafheir then // Note: nodes $v$ and $d$ were neighbours
        Execute <$x$>'s leafwill // A leafwill has a Bypass and/or a node takeover action
    end if
    if node $x$ was <$v$>'s leafheir then // Note: nodes $v$ and $d$ were neighbours
        if $parent(v) = helper(v)$ then // $v$'s helper node is real $v$'s parent
            $v$ designates the other child (i.e., not $v$) of $parent(v)$ as <$v$>'s leafheir // Each node in the RThas two children.
        else
            $v$ designates $parent(v)$ as new leafheir
        end if
        $v$ sends $leafwill(v)$ to $leafheir(v)$
    end if
end if
```

or a child of *helper*(v) in RT(p). If node y is *real*, it cannot have a child in RT(p), therefore it can be *leafheir* for only one of u, v or w (by the first rule). However, if y is a helper node, the following case may apply: y is the parent of both *real*(u) and *real*(v) and child of *helper*(w) (wlog). However, since the RT is a balanced binary search tree ordered on the IDs of the leaf children (the *real* nodes), one of y's children (the left child) must be *real*(y). Therefore, *helper*(y) can be *leafheir* for either u or v, and w. □

Lemma 5.5. CompactFT *requires only* $O(\log n)$ *memory per node.*

Proof. Here, we analyse the memory requirements of a real node v in CompactFT. As mentioned before, v does not store the list of its neighbours or its RT (these can be of $\Omega(n)$ size). However, v uses the $O(\log n)$ sized fields **numchildren** and **maxportnumber** to keep track of the number of children it presently has and the maximum port number it uses. CompactFT uses the property that the initial port assignment does not change. If v is a non-leaf node, it does not store any piece of *will*(v) that is distributed among v's children. If v is a leaf node, it has only a $O(\log n)$ sized will (*leafwill*(v)), which it stores with its *leafheir*

Algorithm 5.4 UpdateLeafWillPortion($p, <x>$): Node v updates leaf wills by 'simulation'. The identity of any node a is available as $(a, <a>)$ in the wills.

```
if Node $v$ is $x$'s leafheir then // Simulate execution of $x$'s leafwill
    Let $x'$ be the parent of helper($x$) in will($p$)
    if helper($v$) is parent of real($x$) in will($p$) then // Case 1: helper($x$) was not the parent of real($x$) in will($p$)
        for will($p$) do
            Let $w$ be other child (i.e., not $x$) of $v$.
            Let $u$ be the parent of helper($v$).
            Let $l'$ be the left child of helper($x$)
            Let $r'$ be the right child of helper($x$) // $NULL$ if $x$ was an heir.
        end for
        $p$.PtWillConnection((real($w$), $<w>$), $(u, <u>)$) // Simulate Bypass($<x>$)
        $p$.PtWillConnection((helper($v$), $<v>$), $(x', <x'>)$) // $v$ sends messages through parent $p$
        $p$.PtWillConnection((helper($v$), $<v>$), $(l', <l'>)$)
        $p$.PtWillConnection((helper($v$), $<v>$), $(r', <r'>)$)
    else
        $p$.PtWillConnection((helper($v$), $<v>$), $(x', <x'>)$) // Case 2: helper($x$) was the parent of real($x$); Only Bypass($x$) required.
    end if
else if node $x$ was $<v>$'s leafheir then
    if parent($v$) = helper($v$) in will($p$) then
        $v$ designates the other child (i.e., not $v$) of parent($v$) as $<v>$'s leafheir.
    else
        $v$ designates parent($v$) as new leafheir
    end if
    $p$.PtNewLeafWill(($v, <v>$), (leafheir($v$), $<$leafheir($v$)$>$), leafwill($v$)) // $v$ sends leafwill($v$) to leafheir($v$).
end if
```

(though v can also store the *leafwill*(v)). Let p be the parent of v. If p is *real*, node v will store *willportion*(p, v), *i.e.,* the portion of *will*(p) (*i.e.,* RT(p)) in which v is involved. From Lemma 5.1, there can only be two occurrences of v (one as *real* and one as *helper*). Since RT(p) is a binary tree, *helper*(v) can have at most 3 neighbours and *real*(v) at most 1 (as real nodes are leaves in a RT). Therefore, the total number of IDs in *leafwillportion*(p, v) cannot exceed 6 and its size is $O(\log n)$. By the same logic, if v hosts a *helper* node, by Lemma 5.1, it only requires $O(\log n)$ memory. v may also have *leafwill*s for its siblings. By Lemma 5.4, *real*(v) and *helper*(v) may store at most 2 of these wills each; this takes only $O(\log n)$ storage. Every node in a *will* is identified by both its ID and port number. However, this only doubles the memory requirement. Finally, all the messages exchanged (Table 3) are of $O(\log n)$ size. □

Theorem 4.1: *The* CompactFT *has the following properties:*

1. CompactFT *increases the degree of any vertex by only 3.*
2. CompactFT *always has diameter bounded by* $O(D \log \Delta)$*, 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.*

Proof. Part 1 follows from Lemma 5.2 and Part 2 from Lemma 5.3. Part 3 follows from Lemma 5.5. Part 4 follows from the construction of the algorithm. Since the virtual helper nodes have degree at most 3, healing one deletion results in at most O(1) changes to the edges in each affected reconstruction tree. As argued in Lemma 5.5, both the memory and any messages thus constructed are $O(\log n)$ bits. Any message is required to be sent only $O(1)$ hops away. Moreover, all changes can be made in parallel. Only the broadcast messages, $Br*$ messages, are broadcasted by a *real* node to all its neighbours and thus $O(\Delta)$ messages (sent in parallel) may be used. □

6. 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 [4] (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 4).

The variant of Thorup and Zwick we use in our construction is for no particular reason but to show that the construction is not tailor-made for a particular routing protocol but for any direct compact routing tree protocol based on a DFS labelling. Recall that a routing protocol is *direct* if the header of a packet is not modified during the routing.

6.1. 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.

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 *tzheavy* and a light node as *tzlight*.

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	
$H_v[0]$ $H_v[1,\ldots,H_v[0]]$	number of tzheavy children of v tzheavy children of v
P_v: array with $b+1$ elements	
$P_v[0]$ $P_v[1,\ldots,H_v[0]]$	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

We first enumerate the nodes of T in *DFS post-order manner*, with the tzheavy nodes traversed before the tzlight 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 6.1.

Note that here we describe the scheme for the static case where the tree does not change over time. In Section 6.2 we show that it can be adapted to the dynamic self-healing model by initially setting up the data structure in exactly the same way as the static case during preprocessing. The classification of tzheavy and tzlight nodes are irrelevant when the deletions start happening (and actually might not be true anymore in the current virtual tree) as they were useful only in the initial setting up to compute compact routing tables and labels.

Algorithm 6.1 The TZ scheme. Code for node v for a message sent to node w.

```
operation $\mathsf{TZ}_v(w, L(w))$:
if $v = w$ then
    The message reached its destination
else if $w \notin [d_v, v]$ then
    Forward to the parent through port $P_v[0]$
else if $w \in [c_v, v]$ then
    Forward to a tzlight node through port $L(w)[\ell_v]$
else
    Let $i$ be the index s.t. $H_v[i]$ is the smallest tzheavy child of $v$ greater than or equal to $w$
    Forward to a heavy node through port $P_v[i]$
end if
end operation
```

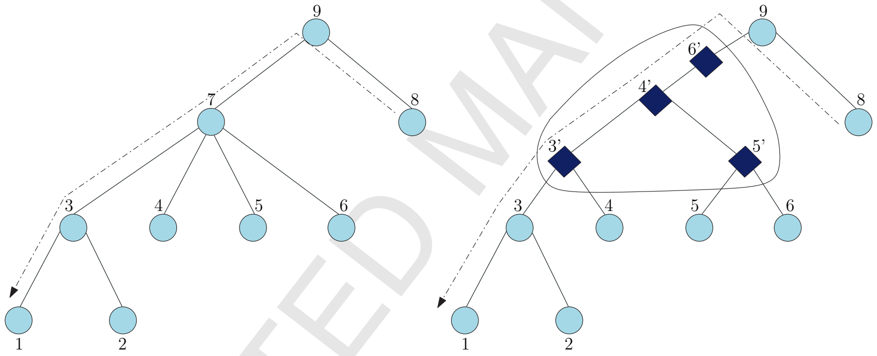


Figure 4: The left side shows the tree before any deletion with the path a message from 8 to 1 will follow. The right side shows the tree obtained after deleting 7. The nodes enclosed in the rectangle are virtual helper nodes replacing 7. To route a message from 8 to 1, virtual nodes perform binary search,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

6.2. *The CompactFTZ scheme*

CompactFTZ (Algorithm 6.2) is a fault tolerant adaptation of TZ that runs in CompactFT. The initialization phase 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 6.1, 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).

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

Algorithm 6.2 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 IDs.

```
 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
```

CompactFT, when a child x of p is deleted and a child w of x creates an edge (p, w), such an edge will reuse the port of p used by (p, v) and any available port of w. As we shall see, this will make that the local TZ protocol running in p is oblivious to the deletion of x.

Every node runs CompactFT at all times. For routing, a real node just follows TZ (Algorithm 6.2, 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 tzlight and tzheavy 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 RT(x). If a packet traverses RT(x), the virtual nodes ignore the tzheavy/tzlight 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 each node checks if the packet destination falls in the intervals of one of its tzheavy nodes, otherwise, it uses its light routing index to pick the correct port to forward the message to in the label of the destination node carried 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.

In what follows 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 by $children_t(v)$, while $parent_t(v)$ is the parent of v in T_t. The set of IDs in $T_t(v)$ is denoted by $\mathrm{ID}(T_t(v))$. If v has two children, $left_t(v)$ and $right_t(v)$ denote the left and right children of v, and $L_t(v)$ and $R_t(v)$ denote the left and 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 $\geq$.

Lemma 6.1, which is the key for proving the correctness CompactFTZ in Theorem 6.2, basically shows that, after any sequence of deletions and subsequent self-healing phases, real nodes maintain the TZ properties and the helper nodes (i.e., the RTs) the BST properties, allowing routing to work properly.

Lemma 6.1. *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 $c \in children_t(v)$, $v > \mathrm{ID}(T_t(c))$, and for every $c, d \in children_t(v)$ with $c < d$, $\mathrm{ID}(T_t(c)) < \mathrm{ID}(T_t(d))$.*
2. *For every virtual node* helper$(v) \in T_t$, *$v \geq \mathrm{ID}(L_t(v))$ and $v < \mathrm{ID}(R_t(v))$.*

Proof. We proceed by induction on t. For $t = 0$, T_0 satisfies property (1) because the properties of the initial DFS labelling, while property (2) is satisfied since there are no virtual nodes in T_0.

Suppose that T_t satisfies (1) and (2). Let $v \in T_t$ the node deleted to obtain T_{t+1}. We show that T_{t+1} satisfies (1) and (2), we only need to check the nodes that are affected when getting T_{t+1}. We identify two cases:

1. $\underline{v \text{ is not a leaf in } T_t}$. Let $c_1, \ldots, c_x$ be the children of v in T_t in ascending order. Each c_i might be real or virtual, and if it is virtual then it is denoted c'_i and is simulated by a real node $real(c'_i)$. Let $RT(v)$ be the reconstruction tree obtained from the children of v, as explained in Section 4. By construction, we have that (virtual) c'_x, the child of v with the largest ID, is the root of $RT(v)$ and it has no right subtree. Moreover, c'_{x-1} is the left child of c'_x. We now see what happens in T_{t+1}. We first analyze the case of the parent of v and then the case of the children.

   If $z = parent_t(v)$ is a real node, then z and c'_x create an edge between them, and hence $RT(v)$ is a subtree of z in T_{t+1} (see Figures 5 top-left and top-right where the node 9 is deleted). By induction hypothesis, the lemma holds for v in T_t and $RT(v)$ contains only the children of v, hence the lemma still holds for z in T_{t+1}. Now, if $z' = parent_t(v)$ is a virtual node, then it must be that there is a virtual node v' in T_t that is an ancestor of v (this happens because at some point the parent of v in $T = T_0$ was deleted, hence v created a virtual node v'; see Figure 5 bottom-left where the parent of 8 is virtual). For now, suppose c_x is a real node (below we deal with the other case). In T_{t+1}, c'_x replaces v', and z' and c'_{x-1} create an edge between them, hence in the end $left_{t+1}(c'_x) = left_t(v')$, $right_{t+1}(c'_x) = right_t(v')$ and $z' = parent_{t+1}(c'_{x-1})$ (see Figure 5 bottom-right where 7' replaces 8'). In other words, the root c'_x of $RT(v)$ takes over v' and the left subtree of c'_x in $RT(v)$ (whose root is c'_{x-1}) is connected to z'. We argue that the lemma holds for c'_x and z' in T_{t+1}. The case of z' is simple: by induction hypothesis, the lemma holds for z' and v in T_t, and $RT(v)$ is made of the children of v in T_t. The case of c'_x is a bit more tricky. First, since the lemma holds for v' in T_t, then it must be that $v \in L_t(v')$ and $c_x < \mathrm{ID}(R_t(v'))$. Also, we have that $v < c'_x$, hence $c'_x \geq \mathrm{ID}(L_{t+1}(c'_x))$ and $c'_x < \mathrm{ID}(R_{t+1}(c'_x))$.

Now, let's see what happens with a child c_i. If c_i is a real node, then virtual c'_i (which belongs to $RT(v)$) is an ancestor of c_i in T_{t+1}. The lemma holds for c_i because the subtrees of c_i in T_t and T_{t+1} are the same. Similarly, the lemma holds for c'_i because $RT(v)$ is a binary search tree and the lemma holds for each child of v in T_t, by induction hypothesis.

Consider now the case c_i is a virtual node, hence denoted c'_i. In this case, in T_t, (real) c_i is a descendant of c'_i. We have that c'_i appears in $RT(v)$ two times, and both of them as virtual nodes, one as a leaf and the other as an internal node (see Figure 5 top-right where 11 is deleted to get the tree at the bottom-left; $8'$ is virtual, is a child of 11 and appears two times as virtual node in $RT(11)$). Let c'_i denote the virtual leaf (this node also belongs to T_t) and c''_i denote the internal virtual node. So, by replacing v with $RT(v)$, it would not be true any more that every real node simulates at most one virtual nodes, since c_i would be simulating c'_i and c''_i. To solve this situation, $u' = left_t(c'_i)$ replaces c'_i, and c'_i replaces c''_i (in this case c'_i always has only one child in T_t, u', which is left). In other words, in $R(v)$, c'_i is moved up to the position of c''_i and u' is moved up to the position of c'_i (see Figure 5 bottom-left). Thus, $L_{t+1}(u') = L_t(u')$ and $R_{t+1}(u') = R_t(u')$. The lemma holds for u' because it was moved up one step and the lemma holds for it in T_t, by induction hypothesis. To prove that the lemma holds for c'_i, let us consider $RT(v)$ and c'_i, c''_i in it. By construction, we have that $c'_i \geq \mathrm{ID}(L(c''_i, RT(v)))$ and $c'_i < \mathrm{ID}(R(c''_i, RT(v)))$. Also, since the lemma holds for each child of v in T_t, for each child $c_j \neq c'_i$ of v in T_j, if $c_j < c'_i$, it must be that $c'_i > \mathrm{ID}(T_t(c_j))$, otherwise $c'_i < \mathrm{ID}(T_t(c_j))$. These two observations imply that $c'_i \geq \mathrm{ID}(L_{t+1}(c'_i))$ and $c'_i < \mathrm{ID}(R_{t+1}(c'_i))$. This completes the case.

2. $\underline{v \text{ is a leaf in } T_t}$. First consider the subcase when the $parent_t(v)$ is a real node. We have that T_{t+1} is T_t minus the leaf v, hence the lemma holds for T_{t+1}, by induction hypothesis.

   Now, if $u' = parent_t(v)$ is a virtual node then in T_t, there is a virtual node v', which is an ancestor of v (this happens because at some point the parent of v in $T_0 = T$ was deleted, hence v created a virtual node).

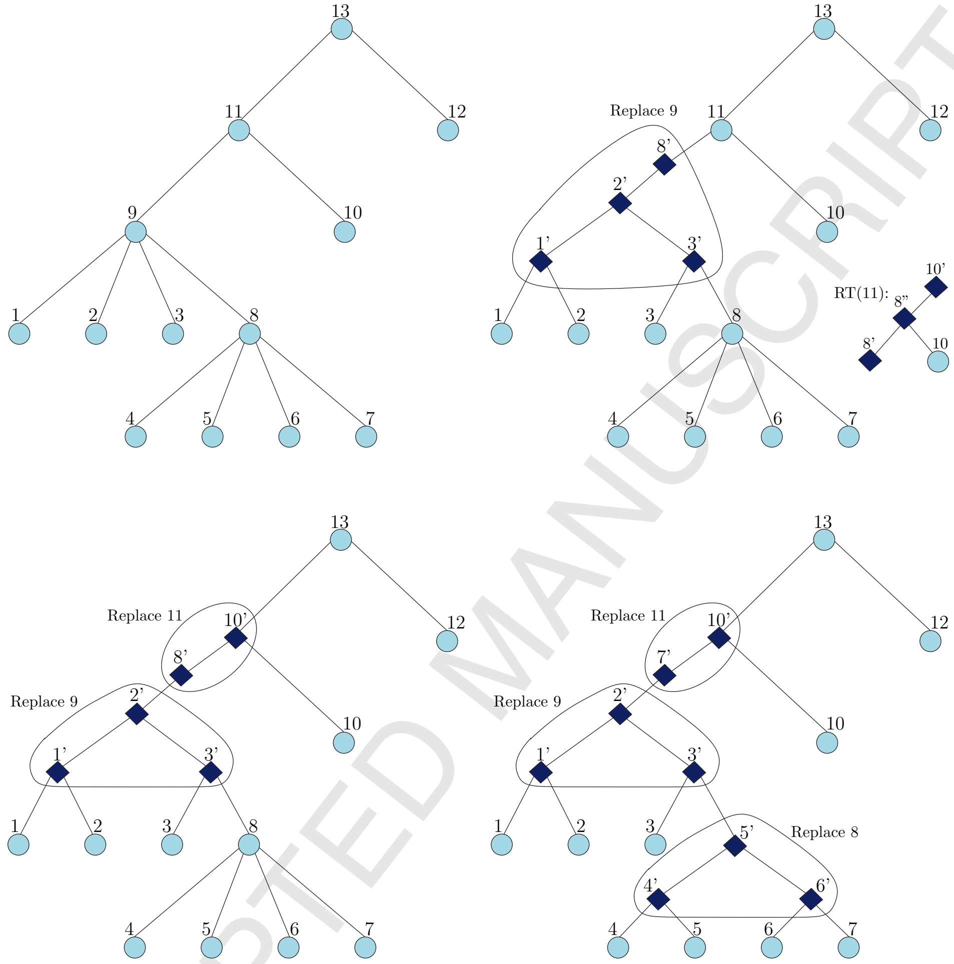


Figure 5: A sequence of deletions. The up-left side shows the tree before any deletion. The up-right shows the tree after the deletion of node 9, which is replaced with a binary tree with its children; after this deletion the reconstruction tree of 11 is shown in the figure. The bottom-left depicts the tree obtained after deleting 11; note 8' simulates 9 before the deletion but 8' participates in the simulation of 10' after the deletion. Finally, the bottom-right shows the tree after the deletion of 8; observe that 7', the largest child of 8, took the place of 8', i.e. its responsibility in the simulation of 11.

In T_t, there is a descending path from v' to v, that passes through virtual nodes until reaches v. Let us denote this path $P = v', u'_1, \ldots, u'_x, v$, for some $x \geq 0$. We have the following three subcases.

If $x = 0$, then v' is the parent of v, and actually $left_t(v') = v$, by the induction hypothesis. Thus, v' and v are just removed, and $R_t(v')$ replaces v' in T_{t+1}, namely, $parent_t(v')$ is connected the root of $R_t(v')$. It is easy to check that the lemma holds for T_{t+1}.

If $x = 1$ then u'_1 replaces v' and $L_{t+1}(u'_1) = L_t(u'_1)$ and $R_{t+1}(u'_1) = R_t(v')$. Namely, v and v' are removed and u'_1 is moved up one step. Again, it is not hard to see that the lemma holds for T_{t+1}.

The last subcase is $x > 1$. In T_{t+1}, u'_x replaces v' and $R_{t+1}(u'_{x-1}) = L_t(u'_x)$. Clearly, the lemma holds for u'_{x-1} because $u'_{x-1} < \mathrm{ID}(L_t(u'_x))$, by induction hypothesis. To prove that the lemma holds for u'_x, we observe the following about the path $P = v', u'_1, \ldots, u'_x, v$, $x > 1$ (e.g., the path from 7' to 7 in Figure 5 bottom-right). The lemma holds for T_t, hence $v \in L_t(v')$, which implies that $u'_1 = left_(v')$, and actually $u'_i < v' = v$, for each v_i. Also observe that the induction hypothesis implies that $v \in R_t(u_i)$, for each u'_i. Therefore, we have that $u'_x > \mathrm{ID}(T_t(u'_i))$, $1 \leq i \leq x-1$, which implies that $u'_x > \mathrm{ID}(L_{t+1}(u'_x))$. Also, since $u'_x \in L_t(v')$, it must be that $u'_x < \mathrm{ID}(R_t(v'))$, hence $u'_x < \mathrm{ID}(R_{t+1}(u'_x))$. The induction step follows.

□

Theorem 6.2. *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.*

Proof. The claim holds for $t = 0$ since CompactFTZ is correct before deletions (as TZ is correct). For $t > 0$, suppose a message M from u to w in T_t, reaches a real node x (possibly $x = u$). Note that x is oblivious to the t-th node deletion, namely, it does not change its routing tables in the healing-process, hence it makes the same decision as in T_{t-1}. Let *#portnumber* be the port through which x sends M in T_t, and let v be the vertex that is connected to x through port *#portnumber* in T_{t-1}. Lemma 6.1 (1) implies that if w is ancestor (descendant) of x in T_{t-1}, then it is ancestor (descendant) of x in T_t. Moreover, the path in T_t from x to w passes through port *#portnumber*. If v is not the t-th vertex deleted, then, v is in T_t, and it gets M from x, which implies that M gets closer to w. Otherwise, v is the t-th vertex deleted. In this case, in T_t, a virtual node y' is connected to x through port *#portnumber*, thus, y' gets M from x. By Lemma 6.1 (2), y' necessarily sends M to a virtual or real node that is closer to w. Thus, we conclude that M reaches w.

For the length of the path, note that after t deletions, from which y are non-leafs, at most y nodes in the path from u to w in T_0 are replaced in T_t with y binary trees of depth $O(\log\Delta)$ each of them. Then, the length of the path from u to w in T_t is at most $\delta(u,w) + y(\log\Delta) - y = \delta(u,w) + y(\log\Delta - 1)$, in case the path has to pass through all these trees from the root to a leaf, or vice versa. □

Lemma 6.3 states the memory usage of CompactFTZ leading to the final correctness theorem (Theorem 6.4).

Lemma 6.3. 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 4.1). 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 6.3 implies that a node can store and route up to x packets using $O(x \log^2 n)$ local memory.

Theorem 6.4. CompactFTZ *is a self-healing compact routing scheme.*

Proof. The theorem follows directly from Lemma 6.3 andTheorems 4.1 and 6.2. □

7. Reporting Non-delivery (deleted receivers and sources)

Contrary to what happens in static schemes such as Thorup-Zwick [4], 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 RT(t) or RT(x) where x was not on the $s - t$ path in T_0. Node v will receive the message either on the way up (towards the root) or on the way down (from the root). In either case, if v is part of 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 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.

8. 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 6.2 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).

9. 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 the out-degree efficiently.

The current paper allows adding 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?

References

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

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

[3] L. Cowen, Compact routing with minimum stretch, J. Algorithms 38 (1) (2001) 170–183.
URL `http://dx.doi.org/10.1006/jagm.2000.1134`

[4] M. Thorup, U. Zwick, Compact routing schemes, in: SPAA, 2001, pp. 1–10.
URL `http://doi.acm.org/10.1145/378580.378581`

[5] P. Fraigniaud, C. Gavoille, Routing in trees, in: Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Proceedings, 2001, pp. 757–772.

[6] S. Chechik, Compact routing schemes with improved stretch, in: ACM Symposium on Principles of Distributed Computing, PODC '13, 2013, pp. 33–41.
URL `http://doi.acm.org/10.1145/2484239.2484268`

[7] T. Hayes, N. Rustagi, J. Saia, 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, ACM, New York, NY, USA, 2008, pp. 203–212.

[8] B. Awerbuch, A. Bar-Noy, N. Linial, D. Peleg, Improved routing strategies with succinct tables, J. Algorithms 11 (3) (1990) 307–341.
URL `http://dx.doi.org/10.1016/0196-6774(90)90017-9`

[9] B. Awerbuch, O. Goldreich, D. Peleg, R. Vainish, A trade-off between information and communication in broadcast protocols, J. ACM 37 (2) (1990) 238–256.
URL `http://doi.acm.org/10.1145/77600.77618`

[10] P. Bose, P. Morin, I. Stojmenovic, J. Urrutia, Routing with guaranteed delivery in ad hoc wireless networks, Wireless Networks 7 (6) (2001) 609–616.
URL `http://dx.doi.org/10.1023/A:1012319418150`

[11] M. Fraser, E. Kranakis, 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.

[12] E. Kranakis, H. Singh, 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.
URL `http://www.cccg.ca/proceedings/1999/c46.pdf`

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

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

[15] P. Fraigniaud, C. Gavoille, A space lower bound for routing in trees, in: H. Alt, A. Ferreira (Eds.), STACS 2002, Proceedings, Vol. 2285 of Lecture Notes in Computer Science, Springer, 2002, pp. 65–75.

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

[17] S. Chechik, Fault-tolerant compact routing schemes for general graphs, Inf. Comput. 222 (2013) 36–44.
URL `http://dx.doi.org/10.1016/j.ic.2012.10.009`

[18] S. Chechik, M. Langberg, D. Peleg, L. Roditty, f-sensitivity distance oracles and routing schemes, Algorithmica 63 (4) (2012) 861–882.
URL `http://dx.doi.org/10.1007/s00453-011-9543-0`

[19] B. Courcelle, A. Twigg, Compact forbidden-set routing, in: STACS 2007, Proceedings, 2007, pp. 37–48.

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

[21] X. Feng, C. Han, A fault-tolerant routing scheme in dynamic networks, J. Comput. Sci. Technol. 16 (4) (2001) 371–380.
URL `http://dx.doi.org/10.1007/BF02948985`

[22] T. Frisanco, Optimal spare capacity design for various protection switching methods in ATM networks, in: Communications,1997 IEEE International Conference on, Vol. 1, 1997, pp. 293–298.
URL `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?isnumber=13277&arnumber=605267&count=107&index=57`

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

[24] K. Murakami, H. S. Kim, Comparative study on restoration schemes of survivable ATM networks, in: INFOCOM, 1997, pp. 345–352.
URL `citeseer.ist.psu.edu/murakami97comparative.html`

[25] B. van Caenegem, N. Wauters, 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, Vol. 1, 1997, pp. 288–292.
URL `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?isnumber=13277&arnumber=605255&count=107&index=56`

[26] J. Vounckx, G. Deconinck, R. Lauwereins, 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, 1994, pp. 125–148.

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

[28] A. Trehan, Self-healing using virtual structures, CoRR abs/1202.2466.

[29] G. Pandurangan, P. Robinson, A. Trehan, Dex: Self-healing expanders, in: Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS '14, IEEE Computer Society, Washington, DC, USA, 2014, pp. 702–711.
URL `http://dx.doi.org/10.1109/IPDPS.2014.78`

[30] G. Pandurangan, A. Trehan, Xheal: a localized self-healing algorithm using expanders, Distributed Computing 27 (1) (2014) 39–54.
URL `http://dx.doi.org/10.1007/s00446-013-0192-1`

[31] T. P. Hayes, J. Saia, A. Trehan, The forgiving graph: a distributed data structure for low stretch under adversarial attack, Distributed Computing (2012) 1–18.
URL `http://dx.doi.org/10.1007/s00446-012-0160-1`

[32] A. Trehan, Algorithms for self-healing networks, Dissertation, University of New Mexico (2010).
URL `http://proquest.umi.com/pqdlink?did=2085415901&Fmt=2&clientId=11910&RQT=309&VName=PQD`

[33] A. D. Sarma, A. Trehan, Edge-preserving self-healing: keeping network backbones densely connected, in: Workshop on Network Science for Communication Networks (NetSciCom 2012), IEEE InfoComm, 2012, iEEE Xplore.

[34] J. Saia, A. Trehan, Picking up the pieces: Self-healing in reconfigurable networks, in: IPDPS. 22nd IEEE International Symposium on Parallel and Distributed Processing., IEEE, 2008, pp. 1–12.
URL `http://arxiv.org/pdf/0801.3710`

[35] G. Saad, J. Saia, Self-healing computation, in: P. Felber, V. K. Garg (Eds.), Stabilization, Safety, and Security of Distributed Systems, SSS 2014, Proceedings, Vol. 8756 of Lecture Notes in Computer Science, Springer, 2014, pp. 195–210.

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

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

[38] F. Kuhn, N. Lynch, R. Oshman, Distributed computation in dynamic networks, in: Proceedings of the 42nd ACM symposium on Theory of computing, STOC '10, ACM, New York, NY, USA, 2010, pp. 513–522.
URL `http://doi.acm.org/10.1145/1806689.1806760`

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

[40] E. W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM 17 (11) (1974) 643–644.
URL `http://dx.doi.org/10.1145/361179.361202`

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

[42] A. Korman, S. Kutten, T. Masuzawa, Fast and compact self stabilizing verification, computation, and fault detection of an MST, in: PODC, 2011, pp. 311–320.

[43] F. Kuhn, S. Schmid, 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, 2005.

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

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

[46] M. Elkin, A near-optimal distributed fully dynamic algorithm for maintaining sparse spanners, in: I. Gupta, R. Wattenhofer (Eds.), PODC, ACM, 2007, pp. 185–194.

[47] S. Baswana, 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, 2003, pp. 384–296.

[48] S. Kutten, A. Porat, Maintenance of a spanning tree in dynamic networks, in: P. Jayanti (Ed.), Distributed Computing, Vol. 1693 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 1999, pp. 846–846.
URL `http://dx.doi.org/10.1007/3-540-48169-9_{24}`

[49] Z. Collin, S. Dolev, Self-stabilizing depth-first search, Information Processing Letters 49 (6) (1994) 297 – 301.
URL `http://www.sciencedirect.com/science/article/pii/0020019094901031`

[50] S. Kutten, C. Trehan, Principles of Distributed Systems: 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings, Springer International Publishing, Cham, 2014, Ch. Fast and Compact Distributed Verification and Self-stabilization of a DFS Tree, pp. 323–338.