# Some Problems in Compact Message Passing

**Armando Castañeda**
Instituto de Matemáticas, UNAM, México
armando.castaneda@im.unam.mx

**Jonas Lefèvre**[1]
Computer Science, Loughborough University, UK
j.lefevre@lboro.ac.uk

**Amitabh Trehan**[1]
Computer Science, Loughborough University, UK
a.trehan@lboro.ac.uk

## Abstract

This paper seeks to address the question of designing distributed algorithms for the setting of *compact memory* i.e. sublinear (in $n$ – the number of nodes) bits working memory for connected networks of arbitrary topologies. The nodes in our networks may have much lower internal (working) memory (say, $O(poly \log n)$) as compared to the number of their possible neighbours ($O(n)$) implying that a node may not be even able to store the $ID$s of all of its neighbours. These algorithms may be useful for large networks of small devices such as the Internet of Things, for wireless or ad-hoc networks, and, in general, as memory efficient algorithms.

More formally, we introduce the *Compact Message Passing* ($CMP$) model – an extension of the standard message passing model considered at a finer granularity where a node can interleave reads and writes with internal computations, using a port only once in a synchronous round. The interleaving is required for meaningful computations due to the low memory requirement and is akin to a distributed network with nodes executing streaming algorithms. Note that the internal memory size upper bounds the message sizes and hence e.g. for $O(\log n)$ memory, the model is weaker than the CONGEST model; for such models our algorithms will work directly too.

We present some early results in the CMP model for nodes with $O(\log^2 n)$ bits working memory. We introduce the concepts of *local compact functions* and *compact protocols* and give solutions for some classic distributed problems (leader election, tree constructions and traversals). We build on these to solve the open problem of compact preprocessing for the compact self-healing routing algorithm *CompactFTZ* posed in *Compact Routing Messages in Self-Healing Trees* (Theoretical Computer Science 2017) by designing *local compact functions* for finding particular subtrees of labeled binary trees. Hence, we introduce the first fully compact self-healing routing algorithm. In the process, we also give independent fully compact versions of the Forgiving Tree [PODC 2008] and Thorup-Zwick's tree based compact routing [SPAA 2001].

**2012 ACM Subject Classification** E.1; H.3.4; C.2.1; C.2.4; G.2.2

**Keywords and phrases** Compact memory networks; Self-healing compact routing; Compact functions and protocols; Preprocessing; Tree algorithms and labellings

**Digital Object Identifier** 10.4230/LIPIcs.CVIT.2016.23

**Related Version** Full version of this paper at [5], `https://arxiv.org/abs/1803.03042`

**Acknowledgements** We would like want to thank Prof. Danny Dolev for his perpetual support

[1] This work supported by EPSRC grant EP/P021247/1: Compact Self-Healing and Routing Over Low Memory Nodes (COSHER)


42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

# 1 Introduction

Large networks of low memory devices such as the *Internet of Things (IOT)* are expected to introduce billions of very weak devices that will need to solve distributed computing problems to function effectively. In this paper, we attempt to formalise the development of distributed algorithms for such a scenario of large networks of low memory devices. We decouple the internal working memory of a node from the memory used by ports for ingress (receiving) and egress (transmitting) (e.g. the ingress queue (Rx) and the egress queue (Tx)) which cannot be used for computation. Thus, in an arbitrary network of $n$ nodes, nodes with smaller internal memory ($o(n)$ bits) may need to support a larger number of connections ($O(n)$). To enable this, we introduce the Compact Message Passing (CMP) model, the standard synchronous message-passing model at a finer granularity where each process can interleave reads from and writes to its ports with internal computation using its low ($o(n)$ bits) memory. We give the first algorithms for several classic problems in the CMP model, such as leader election (by flooding), DFS and BFS spanning tree construction and traversals and convergecast. We build on these to develop the first fully compact distributed routing, self-healing and self-healing compact routing algorithms. We notice that with careful construction, some (but not all) solutions incurred almost no additional time/messages overhead compared to regular memory message passing.

There has been intense interest in designing efficient routing schemes for distributed networks [51, 48, 3, 12, 53, 18, 28, 1, 7] with compact routing trading stretch (factor increase in routing length) for memory used. In essence, the challenge is to use $o(n)$ bits memory per node overcoming the need for large routing tables or/and packet headers. Our present setting has a similar ambition - *what all can be done if nodes have limited working memory even if they may have large neighbourhoods?* In fact, we define local memory as *compact* if it is $o(n)$ bits and by extension, an algorithm as compact if it works in compact memory.

We see two major directions for extending the previously mentioned works. Firstly, a routing scheme consists of two parts - a pre-processing algorithm (scheme construction) and a routing protocol [25]. The routing results mentioned above assume sequential centralized pre-processing. Since routing is inherently a distributed networks problem, it makes sense to have the scheme construction distributed too, and this has led to a recent spurt in designing efficient preprocessing algorithms for compact routing schemes [42, 27, 43, 19, 20]. These algorithms do not have explicit constraints on internal working memory, therefore, in essence, they choose to conserve space (for other purposes). Our interpretation is stricter and we develop a pre-processing scheme (for a routing scheme from [53]) assuming that nodes do not even have any excess space and therefore, have, to develop the whole solution in compact memory itself. Moreover, our solutions are deterministic unlike the solutions listed above, though they tackle a broader range of routing schemes than we do.

Secondly, deterministic routing schemes, in the preprocessing phase, rely on discovery and efficient distributed 'encoding' of the network's topology to reduce the memory requirement (a routing scheme on an arbitrary network with no prior topology or direction knowledge would essentially imply large memory requirements). This makes them sensitive to any topology change and, hence, it is challenging to design fault tolerant compact routing schemes. There has been some work in this direction e.g. in the dynamic tree model [36, 34] or with additional capacity and rerouting in anticipation of failures [9, 8, 11, 16, 24, 26]. Self-healing is a responsive fault-tolerace paradigm seeking minimal anticipatory additional capacity and has led to a series of work [55, 50, 32, 56, 45, 46, 31, 52] in the recent past for maintaining topological properties (connectivity, degrees, diameter/stretch, expansion etc.). Algorithms were also proposed to 'self-heal' computations e.g. [49]. Combining the above motivations, [6] introduced a fault-tolerant compact routing solution CompactFTZ in the (deletion only) self-healing model where an omniscient adversary attacks by removing nodes and the affected nodes distributively respond by adding back connections. However, as in previous routing schemes, CompactFTZ's pre-processing assumed large (not compact) memory. This paper addresses that important problem developing a compact pre-processing deterministic algorithm for CompactFTZ. We also develop a compact pre-processing deterministic algorithm for CompactFT (a compact version of *ForgivingTree* [32]). This leads to a fully compact (i.e. completely distributed and in compact memory) routing scheme, a fully compact self-healing routing scheme and a fully compact self-healing algorithm.

**Our model:** In brief (detailed model in Section 2), our network is an arbitrary connected graph over $n$ nodes. Each node has a number of uniquely identified communication ports. Nodes have $o(n)$ bits of working memory (We need only $O(\log^2 n)$ for our algorithms). However, a node may have $O(n)$ neighbours. Note that a node has enough ports for unicast communication with neighbours but port memory is specialised for communication and cannot be used for computation or as storage space. Also note that the size of the messages are upper bounded by the memory size (in fact, we only need $O(\log n)$ bits sized messages as in the CONGEST model [47]). In standard synchronous message passing, in every round, a node reads the messages of all its neighbours, does some internal computation and then outputs messages. Our nodes cannot copy all the messages to the working space, hence, in our model, nodes interleave processing with reads and writes as long as each port is read from and written to at most once in a round. Hence, a round may look like $pr_1pr_3pw_2pw_1pr_2\dots$ where $p$,$r$ and $w$ stand for processing, reading and writing (subscripted by port numbers). As in regular message passing, all outgoing messages are sent to the neighbours to be delivered by the end of the present round. The order of reads may be determined by the node ((self) deterministic reads) or by an adversary (adversarial reads) and the order of writes by the node itself. We call this the Compact Message Passing(CMP) model.

The model can also be viewed as a network of machines with each node locally executing a kind of streaming algorithm where the input (of at most $\delta$ items, where $\delta$ is the number of ports) is 'streamed' with each item seen at most once and the node computing/outputing results with the partial information. Our self-healing algorithms are in the *bounded memory deletion only self-healing model* [6, 55] where nodes have compact memory and in every round, an omniscient adversary removes a node but the nearby nodes react by adding connections to self-heal the network. However, their preprocessing requires only the CMP model.

**General solution strategy and an example:** A general solution strategy in the CMP model is to view the algorithms as addressing two distinct (but related) challenges. The first is that the processing after each read is constrained to be a function of the (memory limited) node state and the previous read (as in a streaming or online fashion) and it results in an output (possibly NULL) which may be stored or output as an outgoing message. We will refer to such a function as a *local compact function*. The second part is to design, what we call, a *compact protocol* that solves the distributed problem of passing messages to efficiently solve the chosen problem in our model. We discuss local compact functions a bit further in our context. A simple compact function may be the $max(.)$ function which simply outputs the maximum value seen so far. A more challenging problem is to design a function that outputs the neighbourhood of a node in a labelled binary tree. Consider the following question: *Give a compact function that given as input the number of leaves $n$ and any leaf node $v$,* *returns the neighbourhood of $v$ for a balanced binary search tree of $n$ leaves with internal nodes repeated as leaves and arranged in ascending order from left to right.* Note that the function should work for a tree of any size without generating the whole tree (due to limited memory). Figure 1 illustrates the question (further background is in Section 3) – the solution to a similar question (solution in Section 5.1) forms the crux of our fully compact algorithms. It's also a question of interest whether this approach could be generalised to construct a generic function that when input a compact description of a structure (in our case, already encoded in the function) generates relevant compact substructures on demand when queried.

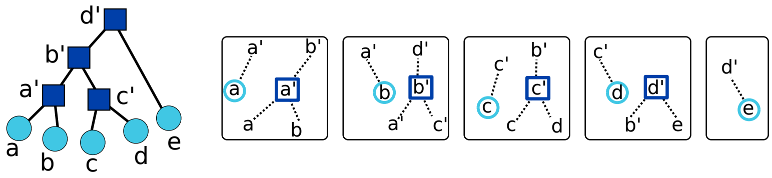


**Figure 1** *Compact function $f$ to query labeled BST trees/half-full trees (Section 5.1)*: On the left is such a tree with 5 leaves. $f(5, b)$ will return the second box (having the $O(\log n)$ sized subtrees of $b$ and $b'$)

**Our results:** Our results follow. We introduce the model CMP hoping it will provide a formal basis for designing algorithms for devices with low working memory in large scale networks. As mentioned, we introduce a generic solution strategy (compact protocols with compact functions) and in Section 5.1 (cf. Lemma 13), we give a compact function of independent interest that compactly queries a labelled binary tree. We give some deterministic algorithms in the CMP model as summarised in Table 1. We do not provide any non-obvious lower bounds but for some algorithms it is easy to see that the solutions are optimal and suffer no overhead due to the lower memory as compared to regular message passing (denoted with a '*' in Table 1). In general, it is easier to construct (by broadcast) and maintain spanning trees using a constant number of parent and sibling pointers, and effectively do bottom up computation, but unicast communication of parent with children may suffer due to the parent's lack of memory (with parent possibly resorting to broadcast). We solve preprocessing for the compact routing scheme $TZ$ ([6], based on [53]), compact self-healing scheme CompactFT ([6], based on [32]) and CompactFTZ as summarised in Theorem 1 leading to fully compact routing, self-healing and self-healing routing solutions (In conjunction with [6]) as Corollaries 2 to 4. Note that combining with the tree cover results from [54], our algorithms could be extended to obtain low stretch routing for general graphs.

▶ **Theorem 1.** *In the* Compact Message Passing *model, given a connected synchronous network $G$ of $n$ nodes and $m$ edges with $O(\log^2 n)$ bits local memory:*

1. *Algorithm* TZ *preprocessing (deterministic / adversarial reads) can be done using $O(\log n)$ bits size messages in $O(m)$ time and $O(mD)$ messages, or using $O(\log^2 n)$ bits size messages in $O(m)$ time and $O(m)$ messages.*
2. *Algorithm* CompactFT *preprocessing can be done using $O(\log n)$ bits size messages in $O(D)$ time and $O(m)$ messages for deterministic reads, or $O(D+\Delta)$ time and $O(m+n\Delta)$ messages for adversarial reads.*
3. *Algorithm* CompactFTZ *preprocessing can be done using $O(\log n)$ bits size messages in $O(m)$ time and $O(mD)$ messages for deterministic reads or $O(m+\Delta)$ time and $O(mD+n\Delta)$ messages for adversarial reads.*

| Algorithm | Internal memory | Time (# rounds) | # messages | In Paper |
|---|---|---|---|---|
| * Leader Election by flooding | $O(\log n)$ | $O(D)$ | $O(m)$ | Section 2.1 |
| * BFS Spanning Tree Construction | $O(\log n)$ | $O(D)$ | $O(m)$ | Section 4.1 |
| * (regular) Convergecast | $O(\log n)$ | $O(D)$ | $O(n)$ | Section 4.2 |
| Weight labelling with convergecast | $O(\log n)$ | $O(D)$ | $O(nD)$ | Section 4.2 |
| Compact DFS relabelling of a tree | $O(\log n)$ | $O(m)$ | $O(m)$ | Section 4.3 |
| * further Compact DFS walks | $O(\log n)$ | $O(n)$ | $O(n)$ | Section 4.3 |
| 'Light' Path by BFS construction | $O(\log^2 n)$ | $O(D)$ | $O(m)$ | Section 4.3 |
| Wills (w/ half-full tree labelling) setup | | | | |
| * ...with deterministic reads | $O(\log^2 n)$ | $O(1)$ | $O(n)$ | Section 5.2 |
| ...with adversarial reads | $O(\log n)$ | $O(\Delta)$ | $O(n\Delta)$ | Section 5.3 |
| TZ preprocessing | $O(\log^2 n)$ | $O(m)$ | $O(m)$ | Th.1.1 |
| CompactFT preproc. with adversarial reads | $O(\log n)$ | $O(D+\Delta)$ | $O(m+n\Delta)$ | Th.1.2 |
| with deterministic reads | $O(\log n)$ | $O(D)$ | $O(m)$ | Th.1.2 |
| CompactFTZ preproc. with adversarial reads | $O(\log^2 n)$ | $O(m+\Delta)$ | $O(m+n\Delta)$ | Th.1.4 |
| with deterministic reads | $O(\log^2 n)$ | $O(m)$ | $O(m)$ | Th.1.3 |

†) $\Delta$: Upper bound on number of ports of a node
*) No additional overhead in comparison with regular message passing.

**Table 1** Summary of the algorithms in the CMP model in this paper. Results apply to both deterministic and adversarial reads unless otherwise indicated.

4. *Algorithm* CompactFTZ *preprocessing can be done using $O(\log^2 n)$ bits size messages in $O(m)$ time and $O(m)$ messages for deterministic reads, or $O(m+\Delta)$ time and $O(m+n\Delta)$ messages for adversarial reads.*

*where $\Delta$ is an upper bound on the number of ports of a node*

▶ **Corollary 2.** *There is a fully compact and distributed variant of the routing scheme* TZ *for a network of nodes with $O(\log^2 n)$ bits memory, $O(\log n)$ bits routing tables, and $O(\log^2 n)$ bits labels and message size.*

▶ **Corollary 3.** *There is a fully compact and distributed variant of the self-healing algorithm* CompactFT *for a network of nodes with $O(\log n)$ bits memory and message size.*

▶ **Corollary 4.** *There is a fully compact and distributed variant of the self-healing routing algorithm* CompactFTZ *for a network of nodes with $O(\log^2 n)$ bits internal memory.*

## 2 Model

We assume a connected synchronous network of arbitrary topology represented by an undirected graph $G = (V, E)$ with $|V| = n$ and $|E| = m$ for $n$ nodes and $m$ bidirectional links. Every node has *compact* internal memory (of size $k = o(n)$), a *unique id* and a collection of *ports* (interfacing with the bidirectional links) each with a a locally unique *port-id*. Each port has an *in-buffer* that can be read from and an *out-buffer* that can be written to. Note that the ports need not be physical devices but even uniquely identifiable virtual interfaces e.g. unique frequencies in wireless or sensor networks. The neighbours need not be on contiguous ports i.e. there may be 'dead' ports interspersed with live ones. Even starting from contiguous ports, this can happen due to node deletions, or due to subnetworks generated (e.g. spanning trees). Thus, our algorithms have to be aware of such 'dead' ports.

In standard message passing, nodes are assumed to have unlimited space/computation. In each round, a node reads all its ports, copies received messages to internal memory, processes the inputs and prepares messages which are then sent to the ports for transmission. However, our compact memory nodes cannot store the inputs (or even $IDs$) of all their neighbours. Hence, we propose the following model with a streaming style computation.

**Compact Message Passing (CMP):** Communication proceeds in synchronous rounds. Messages generated in a round are assumed to reach the neighbour by the end of the round. In every round, every node $v$ executes a *sweep* of its ports fulfilling the following conditions:

1. *Mutable reads condition:* If a read is executed on an in-buffer, the value in that buffer is cleared i.e. not readable if read again.
2. *Fair interleaving condition:* In a sweep, $v$ can read and write to its ports in any order interleaving the reads and writes with internal processing i.e. $pr_ipw_{i'}r_jpw_{j'}p\ldots$, where $p$,$r$ and $w$ stand for processing (possibly none), reading and writing (subscripted by port numbers ( $i, i', j, j', \ldots$). For example, $pr_1pr_2pw_2pw_1p\ldots$ are $pr_1pw_2pr_3pw_1\ldots$ are valid orders. Note that the memory restriction bounds the local computation between reads and writes in the same round. We say that such computations are given by *locally compact functions* where a locally compact function takes as input the previous read and the node state to produce the next state and message(s).
   a. *(self) deterministic reads:* $v$ chooses the order of ports to be read and written to provided that in a sweep, a port is read from and written to at most once. Note that the node can adaptively compute the next read based on previous reads in that sweep.
   b. *adversarial reads:* An adversary decides the order of reads i.e. it picks one of the possible permutations over all ports of the node. The order of writes is still determined by the node $v$ itself (otherwise an adversary could schedule all writes before reads etc. forcing dropped outgoing messages).

We define the following primitives: "**Receive** $M$ from $P$" reads the message $M$ from in-buffer $P$ to the internal memory; "**Send** $M$ via $P$" will write the message $M$ to the out-buffer of port $P$ and "**Broadcast** $M$" will write the message $M$ on every port of the node for transmission. Since condition 2 limits writes to one per port, we also define a primitive "**Broadcast** $M$ except $listP$" which will 'send' the message on each port except the ones listed in $listP$. This can be implemented as a series of Sends where the node checks $listP$ before sending the message. Notice that $listP$ has to be either small enough to fit in memory or a predicate which can be easily computed and checked. For ease of writing, we will often write the above primitives in text in a more informal manner in regular English usage i.e. receive, send, broadcast, and 'broadcast except to ...' where there is no ambiguity. A message is of the form $<$ Name of the message, Parameters of the message $>$.

## 2.1 Warm up: Leader Election by Flooding

As a warm up, let us implement flooding and use it to elect a leader. Assume that a node $v$ has a message $M$ in its memory that needs to be flooded on the network. Node $v$ executes the primitive *Broadcast M* (Sec. 2): $v$ sweeps through its ports in order copying $M$ to every port to be sent out in the next round. In the next round, all nodes, in particular, the neighbours of $v$ read through their ports in deterministic or adversarial order and receive $M$ from $v$. $M$ is copied to the main memory and subsequently broadcast further. To adapt the flooding algorithm for leader election, assume for simplicity that all nodes wake up simultaneously, have knowledge of diameter ($D$) and elect the highest $ID$ as leader. Since every node is a contender, it will broadcast its own ID: say, $v$ broadcasts $M_v$ (message with $ID$ $v$) in the first round. In the next round, every node will receive a different message from its neighbours.

Since a node may have a large number of neighbours, it cannot copy all these $IDs$ to the main memory (as in standard message passing) and deduce the maximum. Instead, it will use the interleaved processing in a streaming/online manner to find the maximum ID received in that round. Assume that a node $v$ has a few neighbours $\{a, b, d, f, \dots\}$ and the reads are executed in order $r_b, r_d, r_a, \dots$ and so on. To discover the maximum $ID$ received, $v$ simply compares the new $ID$ read against the highest it has: let us call this function max (this is a *locally compact* function). Therefore, $v$ now executes in an interleaved manner $r_b$ max $r_d$ max $r_a$ max . . . . At the end of the round, $v$ has the maximum $ID$ seen so far. Every node executes this algorithm for $D$ synchronous rounds to terminate with the leader decided. Note the algorithm can be adapted to other scenarios such as non-simultaneous wakeup and knowledge of $n$ (not $D$) with larger messages or more rounds. Without knowledge of bounds of $n$ or $D$, an algorithm such as ( [39], Algorithm 2) can be adapted (not discussed here).

## 3 Background (CompactFTZ) and paper layout

**Algorithm 3.1** CompactFTZ with preprocessing: A high level view

CompactFTZ*: Compact Preprocessing followed by Compact self-healing routing*

Given a distinguished node $v$ (e.g. by compact leader election (Sec. 2.1))
$T_a \leftarrow$ A BFS spanning tree of graph $G_0$ (Sec. 4.1)
$T_b \leftarrow$ Setup of TZ *heavy arrays* by *compact convergecast* of $T_a$ (Sec. 4.2)
$T_c \leftarrow$ DFS traversal and labelling (renaming) of $T_b$ (Sec. 4.3)
$T_d \leftarrow$ Setup of TZ *light levels* by BFS *traversal* of $T_c$ (Sec. 4.4)
$T_e \leftarrow$ Setup of CompactFT Wills (Sec. 5)
**while** true **do**
    **if** a vertex $x$ (with parent $p$) is deleted **then** {CompactFT *Self-healing [6]*}
        **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; $x$'s *heir* takes over $x$'s duties.
            All affected Wills are updated by simple update of relevant Willportions.
        **else** {Fix leaf deletion}
            **if** $p$ is *real*/alive **then** {Update Wills by simulating the deletion of $p$ and $x$}
                $p$ informs children about deletion; they update Leafwillportions exchanging messages via $p$
            **else** {$p$ had already been deleted earlier}
                Let $y$ be $x$'s *leafheir*; $y$ executes $x$'s Will and affected nodes update Willportions.
    **if** A message headed for $w$ is received at node $v$ **then** {*Compact Self-Healing Routing*}
        **if** $v$ is a real node **then** {Deliver over regular network via compact routing scheme TZ}
            If $(v = w)$ message has reached else if $w \notin [d_v, v]$ forward to parent else if $w \in [c_v, v]$ forward to *light* node through port $L(w)[\ell_v]$ else forward to a *heavy node* through port $P_v[i]$
        **else** {$v$ is a virtual helper node ($=$ *helper*$(v)$)}
            If $(v = w)$ message has reached else traverse RT in a binary search manner

Here, we give a brief background of TZ, CompactFT and CompactFTZ referring to the relevant sections for our solutions. Note that some proofs and pseudocodes have been omitted from the paper due to the lack of space. Algorithm 3.1 captures essential details of CompactFTZ (and of TZ and CompactFT). These algorithms, like most we referred to in this paper, have a distinct preprocessing and main (running) phase. The data structures are setup in the preprocessing phase to respond to events in the main phase (node deletion or message delivery). First, let us consider the intuitive approach. CompactFTZ is designed to deliver messages between sender and receiver (if it hasn't been adversarially deleted) despite node failure (which is handled by self-healing). Self-healing (CompactFT) works by adding virtual nodes and new connections in response to deletions. Virtual nodes are simply logical nodes simulated in real (existing) nodes' memories. Thus, the network over time is a patchwork of virtual and real nodes. It is now possible (and indeed true in our case) that the routing scheme TZ may not work over the patched (self-healed) network and the network information may be outdated due to the changes. Thus, the composition CompactFTZ has two distinct routing schemes and has to ensure smooth delivery despite outdated information. Nodes then respond to the following events: i) *node deletion (line 8):* self-heal using CompactFT moving from initial graph $G_0$ to $G_1$ and so on (the $i^{th}$ deletion yielding $G_i$), or ii) *message arrival (line 17):* Messages are forwarded using TZ or the second scheme (which is simply binary search tree traversal in our case).

Consider CompactFT. CompactFT seeks to limit diameter increase while allowing only constant (+3) node degree increase over any sequence of node deletions. Starting with a distinguished node (line 1), it constructs a BFS spanning tree in the preprocessing (line 2) and then sets up the healing structures as follows. A central technique used in topological self-healing is to replace the deleted subgraph by a reconstruction subgraph of its former neighbours (and possibly virtual nodes simulated by them). These subgraphs have been from graph families such as balanced binary search trees [32], half-full trees [31], random r-regular expanders [46], and p-cycle deterministic expanders [45]. Figure 2 illustrates this for CompactFT where the star graph of deleted node $v$ is replaced by the *Reconstruction Tree*(RT) of $v$. In preprocessing (line 6), every node constructs its RT (also called its Will) in memory and distributes the relevant portions (called Willportion) to its neighbours so that they can form the RT if it is deleted. However, since nodes do not have enough memory to construct their RT, they rely on a compact function to generate the relevant will portions. Referring back to Figure 1, the tree in the figure can be thought of as a RT of a deleted node (or its Will before demise) and the subgraphs in the boxes as the Willportions (one per neighbour). The node now queries the compact function **SearchHT**(Algorithm 5.1) to generate Willportions. Once these structures have been setup in preprocessing, the main phase consists of 'executing' the Will i.e. making the new edges upon deletion and keeping the Willportions updated. The actions differ for internal and leaf nodes – cf. [6] for details.

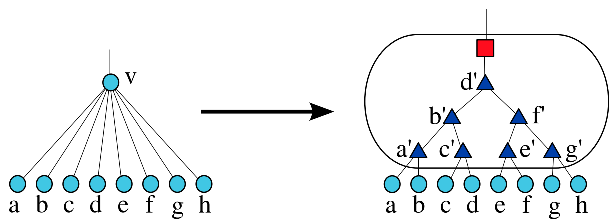


**Figure 2** CompactFT [6, 32]: The deleted node $v$ is replaced by a reconstruction tree (RT) with $v$'s ex-neighbours forming the leaves and simulating the internal virtual nodes (e.g. $d'$ is simulated by $d$).

Now, consider the routing scheme TZ. TZ is postorder variant of the tree routing scheme of [53]. The scheme is wholly constructed in the preprocessing phase - the original paper does not give a distributed construction. Here, we give a compact distributed construction. On a rooted spanning tree (the BFS tree obtained for CompactFT above), every node is marked either *heavy* if it heads a subtree of more than a $b^{th}$ (a constant) fraction of its parent's descendants else *light*. Reference to (at most $b$) heavy children is stored in an array $H$ with corresponding ports in an array $P$ making a routing table. We do this by a *compact* *convergecast* (Line 3). A DFS traversal prioritised by heavy children follows; nodes now get relabeled by their DFS numbers (line 4). Lastly, for every node, its path from the root is traced and the light nodes on the way (which are at most $O(\log n)$) are appended to its new label (line 5). Every node now gets a 'light level' as the number of light nodes in its path. Note that the label is of $O(\log^2 n)$ bits requiring our algorithms to use $O(\log^2 n)$ bits memory. All other parts (including CompactFT) require only $O(\log n)$ bits. This yields a compact setup of TZ. When a packet arrives, a real node checks its parent and array $H$ for the DFS interval failing which it uses its light level and the receiver's label to route the packet through light nodes. If a packet comes to a virtual node, binary search traversal is used since our RTs are binary search trees. Interestingly, even though the arrays and light levels etc. get outdated due to deletions, [6] shows routing continues correctly once set up.

# 4 Some Basic Tree Algorithms and $TZ$ Preprocessing

We present here three distributed algorithms related to trees (deferring some proofs/pseudo-codes to a complete version of this paper): (1) BFS traversal and spanning tree construction, (2) convergecast and (3) DFS traversal, tree construction and renaming. We present these independently and also adapting them in the context of TZ, CompactFT and CompactFTZ preprocessing. The general algorithms can be easily adapted for other problems, for example, the BFS construction can be adapted to compute compact topdown recursive functions, convergecast for aggregation and bottom-up recursive functions and DFS to develop other priority based algorithms.

## 4.1 Breadth First Traversal and Spanning Tree Construction

We assume the existence of a Leader (Section 2.1). Namely each agent has a boolean variable called $isLeader$ such that this variable is set to $False$ for each agent except exactly one. This Leader will be the root for our tree construction. The construction follows a classic Breadth First Tree construction. The root broadcasts a $JOIN$ message to all its neighbours. When receiving a $JOIN$ message for the first time a node *joins* the tree: it sets its *parent* and *parent_port* variables with the node and port ID from the sender, it answers $YES$ and broadcasts $JOIN$ further. It will ignore all next $JOIN$ messages. To ensure termination, each node counts the number of $JOIN$ and $YES$ messages it has received so far, terminating the algorithm when the count is equal to the number of its neighbours.

▶ **Lemma 5.** *Our algorithm constructs a BFS tree in $O(D)$ rounds using $O(m)$ messages.*

## 4.2 Convergecast

We present a distributed convergecast algorithm assuming the existence of a rooted spanning tree as before with every node having a pointer to its parent. We adapt it to identify heavy and light nodes for TZ preprocessing. The *weight* of a node $v$ is 1 if $v$ is a leaf, or the sum of the weight of its children, otherwise. For a given constant $b \geq 2$, a node $v$ with parent $p$ is *heavy* if $wt(v) \geq \frac{wt(p)}{b}$, else $v$ is *light*.

Algorithm 4.1 computes the weight of the nodes in the tree while also storing the IDs and ports of its heavy children in its lists $H$ and $P$. It is easy to see that a node can have at most $b$ heavy children, thus $H$ and $P$ are of size $O(\log n)$. To compute if the node is a heavy child, it has to wait for its parent to receive the weight of all its children. The parent could then broadcast or the child continuously sends messages until it receives an answer (message

**Algorithm 4.1** Weight Computation by Convergecast - Rules :

**Init** :
  **if** $n\_child = 0$ **then**
    $wt \leftarrow 1$ ; $continue \leftarrow$ **true**
    **send** $<WT, 1>$ via *parent*
  **else** $wt \leftarrow 0$ ; $continue \leftarrow$ **false**

**BeginRound** :
  **if** *continue* **then**
    **send** $<WT2, wt>$ via $parent\_port$

**Receive** $<WT', h>$ from $X$:
  $IsHeavy \leftarrow h$ ; $continue \leftarrow$ **false**
  **Terminate**

**Receive** $<WT, z>$ from $X$:
  $n\_wt{+}{+}$ ; $wt \leftarrow wt + z$
  **if** $n\_wt = n\_child$ **then**
    **send** $<WT, wt, myId>$ via $parent\_port$
    $continue \leftarrow$ **true**

**Receive** $<WT2, pwt, Id>$ from $X$:
  **if** $n\_wt = n\_child$ **then**
    **if** $pwt \geq \frac{wt}{b}$ **then**
      **send** $<WT', \textbf{true}>$ via $X$
      insert $Id$ in $H(v)$ ; insert $X$ in $P(v)$
    **else send** $<WT', \textbf{false}>$ via $X$

type $WT2$ in Algorithm 4.1). Note the broadcast version will accomplish the same task in $O(D)$ rounds with $O(n\Delta)$ messages, so either could be preferable depending on the graph.

▶ **Lemma 6.** *Algorithm 4.1 computes the weights in $O(D)$ rounds with $O(nD)$ messages.*

## 4.3 Depth First Walk And Node Relabelling

The next step in the preprocessing of TZ is to relabel the nodes using the spanning tree computed in the previous section. The labels are computed through a post-order DFS walk on the tree, prioritizing the walk towards heavy children. In the algorithm, the root starts the computation, sending the token with the ID set to 1 to its first heavy child. Once a node gets back the token from all its children, it takes the token's ID as its own, increments the token's ID and sends to its parent. Note that in our algorithm, each node $v$ has to try all its ports when passing the token (except the port connected to its parent) since $v$ cannot 'remember' which ports connect to the spanning tree. Our solution to this problem is to "distribute" that information among the children. This problem is solved while performing the DFS walk. Each node $v$, being the $l$-th child of its parent $p$, has a local variable $nxt\_p$, which stores the port number of $p$ connecting it with its $(l+1)$-th child. This *compact* representation of the tree will allow us to be round optimal in the next section.

▶ **Lemma 7.** *The relabeling of the nodes using a pre-order DFS walk can be performed in $O(m)$ rounds, using $O(m)$ messages overall.*

## 4.4 Computing Routing Labels

We now have enough information (a leader, a BFS spanning tree, node weights, DFS labels) to produce routing labels in TZ, and hence, to complete the preprocesing. For a node $v$, its *light path* is the sequence of port numbers for light nodes in the path from the root to $v$. The routing label of $v$ in TZ is the pair $(NewId, LightPath)$, where $NewId$ is its DFS label and $LightPath$ its light path. The second routing table entry for the root is empty.

A simple variant of the algorithm for the BFS tree construction computes the routing labels if $O(\log^2 n)$ sized messages are permitted, otherwise a slower variant can do the same with $O(\log n)$ messages. For the $O(\log^2 n)$ size variant, the root begins by sending its path(*empty*) to each port $X$ along with the port number $X$. When a node receives a message $< RL, path, X >$ from its parent, it sets its light path to $path{\cdot}X$, if it is light, otherwise to $path$ only, producing its routing label. Then, for each port $X$, it sends its light path together with the port number $X$. For the $O(\log n)$ size variant, every light node receives from its parent the port number it is on (say, port $X$) and then does a broadcast labeled with $X$. The root also broadcasts a special message. Ever receiving node appends a received $X$ to its *path* incrementing its light level and terminating when receiving the root's message.

▶ **Lemma 8.** *The routing labels of* TZ *can be computed in $O(D)$ rounds using $O(m)$ messages of $O(\log^2 n)$ size or $O(mD)$ messages of $O(\log n)$ size.*

## 5 Compact Forgiving Tree

Section 3 gives an overview of CompactFT. As it stated, the central idea is a node's Will (its RT) which needs to be pre-computed before an adversarial attack. [6] has only a distributed non-compact memory preprocessing stage[2] in which, in a single round of communication, each node gathers all IDs from its children, locally produces its Will, and then, to each child, sends a part of its Will, called Willportion or *subwill*, of size $O(\log n)$. Computing the Will with compact memory is a challenging problem as a node might have $\Omega(n)$ neighbours making its Will of size $\Omega(n \log n)$. Thus, to compute this information in a compact manner, we need a different approach, possibly costing more communication rounds. Remarkably, as we show below, one round is enough to accomplish the same task in the CMP model, with deterministic reads. The solution is made of two components: a *local compact function* in Section 5.1 that efficiently computes parts of labelled half-full trees of size $O(n \log n)$ using only $O(\log n)$ memory, and a *compact protocol* in Section 5.2 that solves the distributed problem of transmitting the Willportions to its children in a single round.

### 5.1 Computing Half-Full Trees with Low Memory

Half-full trees [31] (which subsume balanced binary trees), redefined below, are the basis for computing the Will of a node in CompactFT. At the core of the construction is a labelling of the nodes with good properties that allows to have Willportions of size $O(\log n)$ to the children of a node. Roughly speaking, a half-full tree is made of several full binary trees, each with a binary search tree labelling in its internal nodes. In what follows we show how to compute properties of that labelling using low memory.

**Computing labels of full binary trees:** Given a power of two, $2^x$, consider the full binary tree with $2^x$ leaves defined recursively as follows. The root of the tree is the string 0, and each node $v$ has left child $v\,0$ and right child $v\,1$. It is easy to see that the nodes at height $h$ are the binary representation of $0, \ldots, 2^{x-h} - 1$. We write $\tilde{v}$ the integer represented by the chain $v$. Moreover, for any node $v$, its left and right children represent the number $2\tilde{v}$ and $2\tilde{v} + 1$, respectively. Let $B(2^x)$ denote the previous tree. We now define a function $\ell$ used in CompactFT that labels the nodes of $B(2^x)$ in the space $[0, 2^x - 1]$. Of course the labelling is not proper but it has nice properties that will allow us to compute it using low memory.

Consider a node $v$ of $B(2^x)$. Let $h_v$ denote the height of $v$ in $B(2^x)$. Then, we define $\ell$ as follows: if $h_v = 0$, $\ell(v) = \tilde{v}$, otherwise $\ell(v) = 2^{h_v - 1} - 1 + \tilde{v}\, 2^{h_v}$.

In words, if $v$ is of height 0, its label is simply $\tilde{v}$, otherwise its label is computed using a base number, $2^{h_v - 1} - 1$, plus $\tilde{v}$ times an offset, $2^{h_v}$. Figure 3 (left) depicts the tree $B(2^3)$ and its labelling $\ell$. Note that the internal nodes have a binary search tree labelling.

[2] Once the non-compact memory preprocessing stage is completed, each process uses only compact memory during the execution of the algorithm.

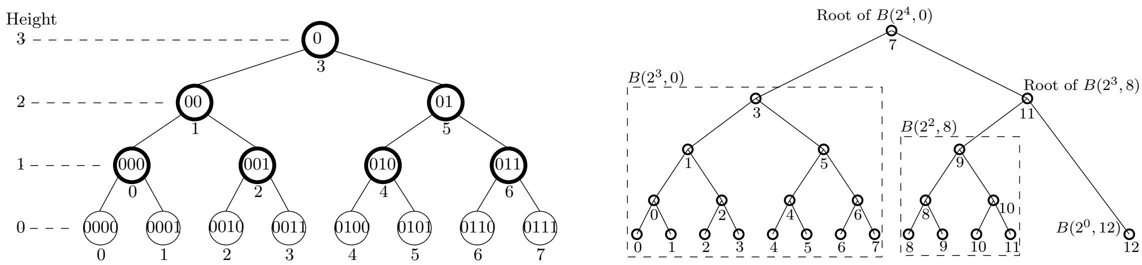

**Figure 3** The tree at the left is $B(2^3)$ with its labeling $\ell$. Each circle shows in its interior the binary identifying the vertex and its decimal value. Near each node appears its label $\ell$. Non-leaf nodes correspond to bold line circles. The tree at the right is the half-full tree $HT([0,12])$ with its labeling $\ell'$.

▶ **Lemma 9.** *Let $B(2^x)$ be a non trivial tree –with $x > 0$. For every vertex $v$, $\ell(v) \in [0, 2^x - 1]$. For the root $r$, $\ell(r) = 2^{x-1} - 1$. Consider any $y \in [0, 2^x - 1]$. There is a unique leaf $v$ of $B(2^x)$ such that $\ell(v) = y$. If $y \leq 2^x - 2$, there is a unique non-leaf $u$ of $B(2^x)$ such that $\ell(u) = y$, and there is no non-leaf $u$ of $B(2^x)$ such that $\ell(u) = 2^x - 1$.*

**Proof.** Let $v$ be a node of $B(2^x)$. As explained above, $\tilde{v} \in \left[0, 2^{x-h_v} - 1\right]$. It is clear that $\ell(v) \geq 0$. If $h_v = 0$, then $\ell(v) = \tilde{v} \leq 2^x - 1$. Else $\ell(v) = 2^{h_v-1} - 1 + \tilde{v}\, 2^{h_v} \leq 2^x - 1 - 2^{h_v-1} < 2^x - 1$.

The root $r$ of $B(2^x)$ has height $h_r = x$ and $\tilde{r} = 0$, hence, by definition, $\ell(r) = 2^{x-1} - 1$. Now, consider any $y \in [0, 2^x - 1]$. Since all leaves are at height 0, there is a unique leaf $v$ with $\ell(v) = y$. Suppose that $y \leq 2^x - 2$. There exists a unique interger factorization of $y + 1$ then there exists unique $h \geq 0$ and $p \geq 0$ such that $y + 1 = 2^h(2p + 1)$. This decomposition can be easily obtained from the binary representation of $y + 1$. By construction, we have $h \leq \log(y + 1) < \log(2^x) = x$ then $h + 1 \leq x$ and we have $2p < 2^{x-h}$ then $p \leq 2^{x-h-1} - 1$. Let consider the unique (non-leaf) node $u$ such that $\tilde{u} = p$ and $h_u = h + 1 \geq 1$. It means that $u$ is the unique node such that $\ell(u) = y$. Finally, there is no non-leaf $u$ of $B(2^x)$ such that $\ell(u) = 2^x - 1$ because we just proved that each element of $[0, 2^x - 2]$ has a unique inverse image under $\ell$. Since the number of non-leaf node is exactly $2^x - 1 = |[0, 2^x - 2]|$, there is no non leaf node $u$ such that $\ell(u) = 2^x - 1$. ◀

By Lemma 9, when considering the labelling $\ell$, each $y \in [0, 2^x - 1]$ appears one or two times in $B(2^x)$, on one leaf node and at most on one non-leaf node. Thus, we can use the labelling $\ell$ to unambiguously refer to the nodes of $B(2^x)$. Namely, we refer to the leaf $v$ of $B(2^x)$ with label $\ell(v) = y$ as *leaf* $y$, and, similarly, if $y \leq 2^x - 2$, we refer to the non-leaf $u$ of $B(2^x)$ with label $\ell(u) = y$ as *non-leaf* $y$. By abuse of notation, in what follows $B(2^x)$ denotes the tree itself and its labelling $\ell$ as defined above. The following lemma directly follows from the definition of $\ell$.

▶ **Lemma 10.** *Let $B(2^x)$ be a non trivial tree ($x \geq 1$). Consider any $y \in [0, 2^x - 1]$. The parent of the leaf $y$ is the non-leaf $2\lfloor \frac{y}{2} \rfloor$. If $y \leq 2^x - 2$, then let $y = 2^i - 1 + z\, 2^{i+1}$. If $i \leq x - 2$, the parent of the non-leaf $y$ is the non-leaf $2^{i+1} - 1 + \lfloor \frac{z}{2} \rfloor 2^{i+2}$. If $i \geq 1$, the left and right children of the non-leaf $y$ are the non-leafs $2^{i-1} - 1 + 2z\, 2^i$ and $2^{i-1} - 1 + (2z + 1)\, 2^i$, respectively. If $i = 0$, the left and right children of the non-leaf $y$ are the leafs $y$ and $y + 1$.*

The proof of Lemma 9 shows how to quickly represent a non-leaf node $v$ with $\tilde{v} \in [0, 2^x - 2]$ in its form $\tilde{v} = 2^i - 1 + \lfloor \frac{\tilde{u}}{2} \rfloor 2^{i+1}$ so that one can easily compute its parent and children, using Lemma 10. Given a value $y \in [0, 2^x - 1]$, function **SearchBT** in Algorithm 5.1 returns the parent of the leaf node $v$ of $B(2^x)$ and the parent and children of the non-leaf node $v'$ of $B(2^x)$ such that $y = \tilde{v} = \tilde{v'}$. The correctness of the function directly follows from Lemma 10.

Note that the function uses $o(x)$ memory: $O(\log x)$ bits to represent $y$ and a constant number of variables with $O(\log x)$ bits.

Let $B(2^x, a)$ denote the tree $B(2^x)$ together with the labelling $\ell'(v) = \ell(v) + a$. Clearly, $\ell'$ labels the nodes of $B(2^x, a)$ in the space $[a, a + 2^x - 1]$. For ease of representation, we use $B(2^x)$ to represent $B(2^x, 0)$ (i.e. $B(2^x)$ with labelling $\ell(v)$) in the discussion that follows.

**Computing labels of half-full trees:** Here, we give a compact function to return the requisite labels from a half-full tree.

▶ **Definition 11.** [31] Consider an integer interval $S = [a, b]$. The *half-full tree with leaves in* $S$, denoted $HT(S)$, is defined recursively as follows. If $|S|$ is a power of two then $HT(S)$ is $B(|S|, a)$. Otherwise, let $2^x$ be the largest power of two smaller than $|S|$. Then, $HT(S)$ is the tree obtained by replacing the right subtree of the root of $B(2^{x+1}, a)$ with $HT([a + 2^x, b])$. The nodes of $HT(S)$ have the induced labelling $\ell'$ of every $B(*, *)$ recursively used for defining the half-full tree.

.

Figure 3 (right) depicts the half-full tree $HT([0, 12])$ and its induced $\ell'$ labelling. The following lemma states some properties of the $\ell'$ labelling of a half-full tree.

▶ **Lemma 12.** *Consider a half-full tree $HT([a, b])$. For every node $v$ of $HT([a, b])$, $\ell'(v) \in [a, b]$. For the root $r$ of $HT([a, b])$, $\ell'(r) = 2^x - 1 + a$, where $2^x$ is the largest power of two smaller than $b - a + 1$. For every $y \in [a, b]$, there is a unique leaf $v$ of $HT([a, b])$ such that $\ell'(v) = y$, and if $y \in [a, b - 1]$, there is a unique non-leaf $v$ of $HT([a, b])$ such that $\ell'(v) = y$.*

As with full binary trees, by Lemma 12, when considering the labelling $\ell'$ of $HT([a, b])$, each $y \in [a, b]$ appears one or two times in $HT([a, b])$, on one leaf node and at most on one non-leaf node. Therefore, those two nodes in the half-full tree can be unambiguously referred as *leaf node* $y$ and *non-leaf node* $y$. The very definition of half-full trees, Definition 11, and Lemma 12 suggest a natural low memory (sublinear on the size of the interval) recursive function that obtains the parent and children of a node in $HT([a, b])$. Such a function, **SearchHT**, appears in Algorithm 5.1.

▶ **Lemma 13.** *Function **SearchHT**$(y, a, b)$ computes the parent and children of leaf and non-leaf $y \in [a, b - 1]$ in $HT([a, b])$ using $O(\log b)$ space.*

## 5.2 Computing and distributing will portions in one round

Among other things, in Section 4, we have computed a spanning tree $T$ of the original graph $G$. Here we present a one-round compact protocol that, for any node $x$, computes and sends to each child of $x$ in $T$ its corresponding will portion. Let $\delta$ denote the number of children of $x$ in $T$. The will of $x$ is the half-full tree $HT([0, \delta - 1])$, where each label $l$ is replaced with the ID of the $l$-th child of $x$ in $T$. Let $RT(x)$ denote this tree with the IDs at its nodes. Thus each child of $x$ with ID $y$ appears two times in $y$, one as leaf node and one as a non-leaf node, and the subwill of $y$ in $RT(x)$ is made of the parent of the leaf $y$ and the parent and children of the non-leaf $y$ in $RT(x)$. This is the information that $x$ has to compute and send to $y$. We can efficiently compute the subwill of a child using a slight adaptation of the function **SearchHT** defined in the previous subsection.

The representation of $T$ is compact: $x$ only knows its number of children in $T$ and the port of its first children (the ports of its children do not have to be contiguous). Additionally, the $l$-th child of $x$ has the port number of $x$, *nxt_port*, that is connected $(l + 1)$-th child of $x$.

In our solution, shown in Algorithm 5.2, $x$ first indicates to all its children to send its ID and *nxt_port* so that this data is in the in-buffers of $x$. Then, with the help of the *nxt_port*, $x$ can sequentially read a collect the IDs of its children, and in between compute and send will portions. In order to be compact, $x$ has to "forget" the ID of a child as soon as it is not needed anymore for computing the will portion of a child (possibly the same or a different one). For example, if $\delta = 13$, then $x$ uses the half-full tree $HT([0, 12])$ in Figure 3, and the label $l$ in $HT([0, 12])$ denotes to the $l$-th child of $x$ in $T$. After reading and storing the IDs of its first four children (corresponding to $0, 1, 2, 3$), $x$ can compute and send the subwill of its first and second children (0 and 1). The leaf 0 in $HT([0, 12])$ has parent non-leaf 0 while the non-leaf 0 has parent non-leaf 1 and children leaf 0 and leaf 1. Similarly, the leaf 1 has parent non-leaf 0 while the non-leaf 1 has parent non-leaf 3 and children non-leaf 0 and non-leaf 2. Moreover, at that point $x$ does not need to store anymore the ID of its first child because (leaf or non-leaf) 0 is not part of any other will portion. An invariant of our algorithm is that, at any time, $x$ stores at most four IDs of its children.

The rules appear in Algorithm 5.2. The algorithm uses function **SubWill**, which computed the subwill of a child node using function the compact function **SearchHT**.

---

**Algorithm 5.1** Calculates the parent and children of leaf and non-leaf $y \in [0, 2^x - 1]$ in $B(2^x)$ and in $HF([a, b])$ .

---

**Function SearchBT**$(y, 2^x)$
  **if** $x = 1$ **then**
    **return** $\langle 0, \perp, 0, 1\rangle$
  **else**
    **if** $y = 2^{x-1} - 1$ **then**
      $\langle P, P', L', R'\rangle \leftarrow \langle 2^{x-1} - 2, \perp, 2^{x-1} - 1 - 2^{x-2}, 2^{x-1} - 1 + 2^{x-2}\rangle$ {$y$ is the root}
    **else if** $y < 2^{x-1} - 1$ **then**
      $\langle P, P', L', R'\rangle \leftarrow$ **SearchBT**$(y, 2^{x-1})$ { $y$ is in the left subtree}
    **else**
      $\langle P, P', L', R'\rangle \leftarrow$ **SearchBT_shift**$(y, 2^{x-1}, 2^{x-1})$ { $y$ is in the right subtree}
  **if** $P' = \perp$ **then** $P' \leftarrow 2^{x-1} - 1$ { $y$ is in the root of one of the two subtrees}
  **return** $\langle P, P', L', R'\rangle$

**Function SearchBT_shift**$(y, 2^x, a)$
  $\langle P, P', L', R'\rangle \leftarrow$ **SearchBT**$(y - a, 2^x)$ ; **return** $\langle P + a, P' + a, L' + a, R' + a\rangle$

**Function SearchHT**$(y, a, b)$
  **if** $b - a = 2^x$ for some $x$ **then**
    **return** **SearchBT_shift**$(y, 2^x, a)$ {The HT is actually a BT}
  **else**
    $\langle P, P', L', R'\rangle \leftarrow \langle \perp, \perp, \perp, \perp\rangle$
    $x = \lfloor \log_2(b - a)\rfloor$ { let $2^x$ be the largest power of two smaller than $b - a + 1$}
    $z = \lfloor \log_2(b - a - 2^x)\rfloor$ {let} $2^z$ be the largest power of two smaller than $b - a - 2^x$
    **if** $y = a + 2^x - 1$ **then**
      $\langle P, P', L', R'\rangle \leftarrow \langle 2^x - 2, \perp, 2^x - 1 - 2^{x-1}, 2^x - 1 + 2^z\rangle$ { $y$ is the root}
    **else if** $y < a + 2^{x-1} - 1$ **then**
        $\langle P, P', L', R'\rangle \leftarrow$ **SearchBT_shift**$(y, 2^{x-1}, a)$ {$y$ is in the left subtree}
      **else**$\langle P, P', L', R'\rangle \leftarrow$ **SearchHT**$(y, a + 2^x, b)$ { $y$ is in the right subtree}
  **if** $P' = \perp$ **then** $P' \leftarrow 2^{x-1} - 1$ { $y$ is in the root of one of the two subtrees}
  **return** $\langle P, P', L', R'\rangle$

---

▶ **Lemma 14.** *All the subwills are correctly computed and sent in 1 round. Moreover, at any time, the memory contains at most $5 \log \delta$ Ids (node or port) and subwills.*

**Algorithm 5.2** Wills

**Init** :
  **if** DFS walk is over : **then**
    $current \leftarrow fst\_p$ ; $k \leftarrow 0$
    **if** not $IsLeader$ **then**
      **send** $< MYId, myId, nxt\_p, ichild >$
      via $parent\_port$

**Receive** $<WILL, p, p_h, c_l, c_r, bool>$from$X$:
  $nxtparent \leftarrow p$ ; $nxthparent \leftarrow ph$
  $nxtchild_l \leftarrow c_l$ ; $nxtchild_r \leftarrow c_r$
  $heir \leftarrow bool$
  **Terminate**

**Receive**$< MYId, z, nxtPort, _ >$from $current$:
  $Node[k] \leftarrow [0, z, current]$
  $Will[k] \leftarrow$ **SubWill**$(k, n\_child, parent)$
  **for all** $j \in Will[k] \cup \{k\}$ **do**
    **if** $\max(Will[j]) \leq k$ **then**
      $p, p_h, c_l, c_r = Will[j]$
      **send** $<WILL, Node[p][1], Node[p_h][1],$
      $Node[c_l][1], Node[c_r][1], [n\_child-1=k]>$
      via $Node[j][2]$
      $free(Will[j])$
      **for all** $x \in \{p, p_h, c_l, c_l\}$ **do**
        $Node[x][0]++$
        **if** $Node[x][0] = 4$ **then** $free(Node[x])$
  $current \leftarrow nxtPort$ ; $k++$

### 5.3 Computing and distributing will parts with adverserial reads

The previous protocol works only in the deterministic reads case. However, it can be adapted to the adversarial reads case at the cost of some more rounds. Instead of computing all subwills in one round, we now compute one subwill per round. At round $k$, a node computes the subwill of its $k$-th child. To do so, it reads all the ports and stores only the needed IDs for computing the subwill of its $k$-th child, and then it sends it to the child.

## 6 Conclusion and Related works

In this work, we formalise and give algorithms for networks of low memory machines executing streaming style algorithms developing the first fully compact self-healing routing algorithm. The power of the CMP model needs to be studied in more detail and lower bounds developed. Algorithms in the asynchronous CMP model and more efficient/optimal versions should be developed. Earlier works have looked at various memory settings in distributed networks. In the *network finite state machine* model [21], weak computational devices with constant local memory communicate in an asynchronous network. Any node only broadcasts symbols from a constant size alphabet and each time it reads its ports (all of them) can only distinguish up to a constant number of occurrences. They show probabilisitic solutions to MST and 3-coloring of trees in this model. In the *beeping* model of communication [13], nodes execute synchronous rounds. Every round, a node either "beeps" and sends or stays silent and listens. A listening node obtains a single bit encoding if one or more of its neighbours beeped. [30] have shown that there are probabilistic solutions to the leader election problem in this model for complete graphs where each node is a state machine with constant number of states. These solutions imply compact probabilistic solution in our CMP model.

As far as we know, the computational power of the CONGEST models ($O(poly \log n)$ sized messages) [47] has never been studied when the local memory of the nodes too is restricted. However, [17] has studied the difference between nodes performing only broadcasts or doing unicast, showing that the unicast model is strictly more powerful. [4] studied the general case where nodes are restricted to sending some number of distinct messages in a round. It'll be interesting to see where CMP fits. Finally, dynamic network topology and fault tolerance are core concerns of distributed computing [2, 44] and various models (e.g. [37]) and topology maintenance and self-* algorithms abound [14, 15, 35, 38, 40, 10, 41, 33, 23].

## References


1. Ittai Abraham, Cyril Gavoille, and Dahlia Malkhi. Routing with improved communication-space trade-off. In Rachid Guerraoui, editor, *Distributed Computing*, pages 305–319, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
2. Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* John Wiley & Sons, 2004.
3. Baruch Awerbuch, Amotz Bar-Noy, Nathan Linial, and David Peleg. Improved routing strategies with succinct tables. *J. Algorithms*, 11(3):307–341, 1990. URL: `http://dx.doi.org/10.1016/0196-6774(90)90017-9`.
4. Florent Becker, Antonio Fernández Anta, Ivan Rapaport, and Eric Rémila. Brief announcement: A hierarchy of congested clique models, from broadcast to unicast. In Georgiou and Spirakis [29], pages 167–169. URL: `http://doi.acm.org/10.1145/2767386.2767447`, doi:10.1145/2767386.2767447.
5. Armando Castañeda, Jonas Lefèvre, and Amitabh Trehan. Self-healing routing and other problems in compact memory. *CoRR*, abs/1803.03042, 2018. URL: `http://arxiv.org/abs/1803.03042`, arXiv:1803.03042.
6. Armando Castañeda, Danny Dolev, and Amitabh Trehan. Compact routing messages in self-healing trees. *Theoretical Computer Science*, 709:2 – 19, 2018. Special Issue of ICDCN 2016 (Distributed Computing Track). URL: `http://www.sciencedirect.com/science/article/pii/S0304397516306818`, doi:https://doi.org/10.1016/j.tcs.2016.11.022.
7. Shiri Chechik. Compact routing schemes with improved stretch. In Fatourou and Taubenfeld [22], pages 33–41. URL: `http://doi.acm.org/10.1145/2484239.2484268`.
8. Shiri Chechik. Fault-tolerant compact routing schemes for general graphs. *Inf. Comput.*, 222:36–44, 2013. URL: `http://dx.doi.org/10.1016/j.ic.2012.10.009`.
9. Shiri Chechik, Michael Langberg, David Peleg, and Liam Roditty. f-sensitivity distance oracles and routing schemes. *Algorithmica*, 63(4):861–882, 2012. URL: `http://dx.doi.org/10.1007/s00453-011-9543-0`.
10. Zeev Collin and Shlomi Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49(6):297 – 301, 1994. URL: `http://www.sciencedirect.com/science/article/pii/0020019094901031`.
11. Bruno Courcelle and Andrew Twigg. Compact forbidden-set routing. In Wolfgang Thomas and Pascal Weil, editors, *STACS 2007, 24th Annual Symposium on Theoretical Aspects of Computer Science, Aachen, Germany, February 22-24, 2007, Proceedings*, volume 4393 of *Lecture Notes in Computer Science*, pages 37–48. Springer, 2007. URL: `https://doi.org/10.1007/978-3-540-70918-3`, doi:10.1007/978-3-540-70918-3.
12. Lenore Cowen. Compact routing with minimum stretch. *J. Algorithms*, 38(1):170–183, 2001. URL: `http://dx.doi.org/10.1006/jagm.2000.1134`.
13. Julius Degesys, Ian Rose, Ankit Patel, and Radhika Nagpal. DESYNC: self-organizing desynchronization and TDMA on wireless sensor networks. In Tarek F. Abdelzaher, Leonidas J. Guibas, and Matt Welsh, editors, *Proceedings of the 6th International Conference on Information Processing in Sensor Networks, IPSN 2007, Cambridge, Massachusetts, USA, April 25-27, 2007*, pages 11–20. ACM, 2007. URL: `http://doi.acm.org/10.1145/1236360.1236363`, doi:10.1145/1236360.1236363.
14. Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, November 1974. URL: `http://dx.doi.org/10.1145/361179.361202`.
15. Shlomi Dolev. *Self-stabilization.* MIT Press, Cambridge, MA, USA, 2000.
16. Robert D. Doverspike and Brian Wilson. Comparison of capacity efficiency of dcs network restoration routing techniques. *J. Network Syst. Manage.*, 2(2), 1994.

**17** Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 367–376. ACM, 2014. URL: `http://doi.acm.org/10.1145/2611462.2611493`, doi:10.1145/2611462.2611493.

**18** Tamar Eilam, Cyril Gavoille, and David Peleg. Compact routing schemes with low stretch factor. *J. Algorithms*, 46(2):97–114, February 2003. URL: `http://dx.doi.org/10.1016/S0196-6774(03)00002-6`, doi:10.1016/S0196-6774(03)00002-6.

**19** Michael Elkin and Ofer Neiman. On efficient distributed construction of near optimal routing schemes: Extended abstract. In George Giakkoupis, editor, *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 235–244. ACM, 2016. URL: `http://doi.acm.org/10.1145/2933057.2933098`, doi:10.1145/2933057.2933098.

**20** Michael Elkin and Ofer Neiman. Linear-size hopsets with small hopbound, and distributed routing with low memory. *CoRR*, abs/1704.08468, 2017. URL: `http://arxiv.org/abs/1704.08468`, arXiv:1704.08468.

**21** Yuval Emek and Roger Wattenhofer. Stone age distributed computing. In Fatourou and Taubenfeld [22], pages 137–146. URL: `http://doi.acm.org/10.1145/2484239.2484244`, doi:10.1145/2484239.2484244.

**22** Panagiota Fatourou and Gadi Taubenfeld, editors. *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*. ACM, 2013. URL: `http://dl.acm.org/citation.cfm?id=2484239`.

**23** Michael Feldmann and Christian Scheideler. A self-stabilizing general de bruijn graph. In Paul Spirakis and Philippas Tsigas, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 250–264, Cham, 2017. Springer International Publishing.

**24** Xiushan Feng and Chengde Han. A fault-tolerant routing scheme in dynamic networks. *J. Comput. Sci. Technol.*, 16(4):371–380, 2001. URL: `http://dx.doi.org/10.1007/BF02948985`.

**25** Pierre Fraigniaud and Cyril Gavoille. A space lower bound for routing in trees. In Helmut Alt and Afonso Ferreira, editors, *STACS 2002, 19th Annual Symposium on Theoretical Aspects of Computer Science, Antibes - Juan les Pins, France, March 14-16, 2002, Proceedings*, volume 2285 of *Lecture Notes in Computer Science*, pages 65–75. Springer, 2002.

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

**27** Cyril Gavoille, Christian Glacet, Nicolas Hanusse, and David Ilcinkas. On the communication complexity of distributed name-independent routing schemes. In Yehuda Afek, editor, *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, volume 8205 of *Lecture Notes in Computer Science*, pages 418–432. Springer, 2013. URL: `https://doi.org/10.1007/978-3-642-41527-2_29`, doi:10.1007/978-3-642-41527-$2_2$9.

**28** Cyril Gavoille and David Peleg. Compact and localized distributed data structures. *Distributed Computing*, 16(2):111–120, Sep 2003. URL: `https://doi.org/10.1007/s00446-002-0073-5`, doi:10.1007/s00446-002-0073-5.

**29** Chryssis Georgiou and Paul G. Spirakis, editors. *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*. ACM, 2015.

**30** Seth Gilbert and Calvin C. Newport. The computational power of beeps. In Yoram Moses, editor, *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo,* Japan, October 7-9, 2015, Proceedings*, volume 9363 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2015. URL: `https://doi.org/10.1007/978-3-662-48653-5_3`, doi:10.1007/978-3-662-48653-5$_3$.

**31** Thomas P. Hayes, Jared Saia, and Amitabh Trehan. The forgiving graph: a distributed data structure for low stretch under adversarial attack. *Distributed Computing*, pages 1–18, 2012. URL: `http://dx.doi.org/10.1007/s00446-012-0160-1`.

**32** Tom Hayes, Navin Rustagi, Jared Saia, and Amitabh 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.

**33** Sebastian Kniesburges, Andreas Koutsopoulos, and Christian Scheideler. Re-chord: a self-stabilizing chord overlay network. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 235–244, New York, NY, USA, 2011. ACM. URL: `http://doi.acm.org/10.1145/1989493.1989527`.

**34** Amos Korman. General compact labeling schemes for dynamic trees. *Distributed Computing*, 20(3):179–193, 2007.

**35** Amos Korman, Shay Kutten, and Toshimitsu Masuzawa. Fast and compact self stabilizing verification, computation, and fault detection of an MST. In Cyril Gavoille and Pierre Fraigniaud, editors, *PODC*, pages 311–320. ACM, 2011.

**36** Amos Korman, David Peleg, and Yoav Rodeh. Labeling schemes for dynamic tree networks. *Theory Comput. Syst.*, 37(1):49–75, 2004.

**37** Fabian Kuhn, Nancy Lynch, and Rotem 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. URL: `http://doi.acm.org/10.1145/1806689.1806760`.

**38** Fabian Kuhn, Stefan Schmid, and Roger 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.

**39** Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. On the complexity of universal leader election. *J. ACM*, 62(1):7:1–7:27, 2015. URL: `http://doi.acm.org/10.1145/2699440`.

**40** Shay Kutten and Avner Porat. Maintenance of a spanning tree in dynamic networks. In Prasad Jayanti, editor, *Distributed Computing*, volume 1693 of *Lecture Notes in Computer Science*, pages 846–846. Springer Berlin / Heidelberg, 1999. URL: `http://dx.doi.org/10.1007/3-540-48169-9_{24}`.

**41** Shay Kutten and Chhaya Trehan. Fast and compact distributed verification and self-stabilization of a dfs tree. In *International Conference on Principles of Distributed Systems*, pages 323–338. Springer, 2014.

**42** Christoph Lenzen and Boaz Patt-Shamir. Fast routing table construction using small messages: extended abstract. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 381–390. ACM, 2013. URL: `http://doi.acm.org/10.1145/2488608.2488656`, doi:10.1145/2488608.2488656.

**43** Christoph Lenzen and Boaz Patt-Shamir. Fast partial distance estimation and applications. In Georgiou and Spirakis [29], pages 153–162. URL: `http://doi.acm.org/10.1145/2767386.2767398`, doi:10.1145/2767386.2767398.

**44** N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Mateo, CA, 1996.

**45** Gopal Pandurangan, Peter Robinson, and Amitabh 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. URL: `http://dx.doi.org/10.1109/IPDPS.2014.78`.

**46** Gopal Pandurangan and Amitabh Trehan. Xheal: a localized self-healing algorithm using expanders. *Distributed Computing*, 27(1):39–54, 2014. URL: `http://dx.doi.org/10.1007/s00446-013-0192-1`.

**47** David Peleg. *Distributed Computing: A Locality Sensitive Approach*. SIAM, 2000.

**48** David Peleg and Eli Upfal. A tradeoff between space and efficiency for routing tables (extended abstract). In Janos 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.

**49** George Saad and Jared Saia. A theoretical and empirical evaluation of an algorithm for self-healing computation. *Distributed Computing*, 30(6):391–412, 2017. URL: `https://doi.org/10.1007/s00446-016-0290-y`, doi:10.1007/s00446-016-0290-y.

**50** Jared Saia and Amitabh 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. URL: `http://arxiv.org/pdf/0801.3710`.

**51** Nicola Santoro and Ramez Khatib. Labelling and implicit routing in networks. *The computer journal*, 28(1):5–8, 1985.

**52** Atish Das Sarma and Amitabh 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.

**53** Mikkel Thorup and Uri Zwick. Compact routing schemes. In *SPAA*, pages 1–10, 2001. URL: `http://doi.acm.org/10.1145/378580.378581`.

**54** Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, January 2005. URL: `http://doi.acm.org/10.1145/1044731.1044732`, doi:10.1145/1044731.1044732.

**55** Amitabh 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`.

**56** Amitabh Trehan. Self-healing using virtual structures. *CoRR*, abs/1202.2466, 2012.