A Distributed Algorithm for Finding All Best Swap Edges of a Minimum Diameter Spanning Tree

Beat Gfeller, Nicola Santoro, and Peter Widmayer

***Abstract*—Communication in networks suffers if a link fails. When the links are edges of a tree that has been chosen from an underlying graph of all possible links, a broken link even disconnects the network. Most often, the link is restored rapidly. A good policy to deal with this sort of *transient* link failures is *swap rerouting*, where the temporarily broken link is replaced by a single *swap* link from the underlying graph. A rapid replacement of a broken link by a swap link is only possible if all swap links have been precomputed. The selection of high quality swap links is essential; it must follow the same objective as the originally chosen communication subnetwork. We are interested in a minimum diameter tree in a graph with edge weights (so as to minimize the maximum travel time of messages). Hence, each swap link must minimize (among all possible swaps) the diameter of the tree that results from swapping. We propose a distributed algorithm that efficiently computes all of these swap links, and we explain how to route messages across swap edges with a compact routing scheme. Finally, we consider the computation of swap edges in an arbitrary spanning tree, where swap edges are chosen to minimize the time required to adapt routing in case of a failure, and give efficient distributed algorithms for two variants of this problem.**

***Index Terms*—Fault-tolerant routing, swap edges, minimum diameter spanning tree, distributed algorithms.**

I. Introduction

For communication in computer networks, often only a subset of the available connections is used to communicate at any given time. If all nodes are connected using the smallest number of links, the subset forms a spanning tree of the network. This has economical benefits compared to using the entire set of available links, assuming that merely keeping a link active for potentially sending messages induces some cost. Furthermore, as only one path exists between any communication pair, a spanning tree simplifies routing and allows small routing tables. Depending on the purpose of the network, there is a variety of desirable properties of a spanning tree. We are interested in a *Minimum Diameter Spanning Tree* (MDST), i.e., a tree that minimizes the largest distance between any pair of nodes, thus minimizing the worst case length of any transmission path, even if edge lengths are not uniform. The importance of minimizing the diameter of a spanning tree has been widely recognized (see e.g. [2]); essentially, the diameter of a network provides a lower bound (and often even an exact one) on the computation time of most algorithms in which all nodes participate.

One downside of using a spanning tree is that a single link failure disconnects the network. Whenever link failures are transient, i.e., a failed link soon becomes operational again, the momentarily best possible way of reconnecting the network is to replace the failed link by a single other link, called a *swap* link. Among all possible swap links, one should choose a *best swap* w.r.t. the original objective [3], [4], [5], [6], that is in our case, a swap that minimizes the diameter of the resulting *swap tree*. Note that the swap tree is different from a minimum diameter spanning tree of the underlying graph that does not use the failed link. The reason for preferring the swap tree to the latter lies in the effort that a change of the current communication tree requires: If we were to replace the original MDST by a tree whose edge set can be very different, we would need to put many edges out of service, many new edges into service, and adjust many routing tables substantially — and all of this for a transient situation. For a swap tree, instead, only one new edge goes into service, and routing can be adjusted with little effort (as we will show). Interestingly, this choice of swapping against adjusting an entire tree even comes at a moderate loss in diameter: The swap tree diameter is at most a factor of 2.5 larger than the diameter of an entirely adjusted tree [4].

In order to keep the required time for swapping small, we advocate to precompute for each edge of the tree a best swap edge. We show in the following that the distributed computation of *all best swaps* has the further advantage of gaining efficiency (against computing swap edges individually), because dependencies between the computations for different failing edges can be exploited.

Related Work: Nardelli et al. [4] describe a centralized (i.e., non-distributed) algorithm for computing all best swaps of a MDST in $O(n\sqrt{m})$ time and $O(m)$ space, where the given underlying communication network $G = (V, E)$ has $n = |V|$ vertices and $m = |E|$ edges. For shortest paths trees (as opposed to minimum diameter spanning trees), an earlier centralized algorithm [5] has been complemented by a distributed algorithm using techniques for finding all best swap edges for several objectives [7], [8]. Using techniques that are quite different from the techniques we propose in this paper, these algorithms require either $O(n)$ messages of size $O(n)$ (i.e., a message contains $O(n)$ node labels, edge weights, etc.)

We gratefully acknowledge the support of the Swiss SBF under contract no. C05.0047 within COST-295 (DYNAMO) of the European Union and the support of the Natural Sciences and Engineering Research Council of Canada. An earlier version of parts of this paper was presented at the 21st International Symposium on Distributed Computing (DISC 2007) [1].

B. Gfeller and P. Widmayer are with ETH Zurich, Switzerland.
N. Santoro is with the Carleton University, Canada.

each, or $O(n^*)$ short messages with size $O(1)$ each, where n^* denotes the size of the transitive closure of the tree if edges are viewed to be directed away from the root. In a so-called preprocessing phase of this algorithm, some information is computed along with the spanning tree construction using $O(m)$ messages. A distributed algorithm for computing a MDST in a graph $G(V, E)$ in an asynchronous setting has $O(n)$ time complexity (in the standard sense, as explained in Section III) and uses $O(nm)$ messages [2]. However, no efficient distributed algorithm to compute the best swaps of a MDST had been found to date.

Our Contribution: In this paper, we propose a distributed algorithm for computing all best swaps of a MDST using no more than $O(n^* + m)$ messages of size $O(1)$ each. The *size of a message* denotes the number of atomic values that it contains, such as node labels, edge weights, path lengths etc., and n^* is the size of the transitive closure of the MDST with edges directed away from a center of the tree (i.e., n^* is essentially the same as above). Both n^* and m are very natural bounds: When each subtree triggers as many messages as there are nodes in the subtree, the size of the transitive closure describes the total number of messages. Furthermore, it seems inevitable that each node receives some information from each of its neighbors in G, taking into account each potential swap edge, totalling to $\Omega(m)$ messages. Our algorithm runs in $O(\|\mathcal{D}\|)$ time (in the standard sense, as explained in Section III), where $\|\mathcal{D}\|$ is the hop-length of the diameter path of G; note that this is asymptotically optimal. The message and time costs of our algorithm are easily subsumed by the costs of constructing a MDST distributively using the algorithm from [2]. Thus, it is cheap to precompute all the best swaps in addition to constructing a MDST initially.

Just like the best swaps algorithms for shortest paths trees ([7], [8]), our algorithm (like many fundamental distributed algorithms) exploits the structure of the tree under consideration. The minimum diameter spanning tree, however, is substantially different from shortest paths trees in that it requires a significantly more complex invariant to be maintained during the computation: We need to have just the right collection of pieces of information available so that on the one hand, these pieces can be maintained efficiently for changing failing edges, and on the other hand, they can be composed to reveal the diameter at the corresponding steps in the computation.

To complement our distributed algorithm, we propose a compact routing scheme for trees which can quickly and inexpensively adapt routing when a failing edge is replaced by a best swap edge. Notably, our scheme does not require an additional full backup table, but assigns a label of $c \log n$ bits to each node (for some small constant c); a node of degree δ stores the labels of all its neighbors (and itself), which amounts to $\delta c \log n$ bits per node, or $2mc \log n$ bits in total[1]. We will show how given this labeling, knowledge of the labels of both adjacent nodes of a failing edge and the labels of both adjacent nodes of its swap edge is sufficient to adjust routing.

Motivated by this routing scheme, we further consider a different variant of the swap edge computation problem, where instead of optimizing the quality of the resulting tree, the time required for the routing adaptation is minimized. This is useful whenever recovery of a failed edge is so quick that the speed of adjusting the routing tables takes priority. This boils down to replacing each failing edge by a swap edge whose endpoints are close to the endpoints of the failing edge. For two different variants of this problem (depending on whether an edge failure is detected at both of its endpoints or only at one), we give a distributed algorithm with running time $O(\|\mathcal{D}\|)$ and message complexity $O(n^* + m)$.

In Section II, we formally define the *distributed all best swaps* problem. Section III states our assumptions about the distributed setting and explains the basic idea of our algorithm. In Section IV, we study the structure of diameter paths after swapping, and we propose an algorithm for finding best swaps. The algorithm uses information that is computed in a preprocessing phase, described in Section V. Our routing scheme is presented in Section VI, and Section VII contains the algorithms for computing swaps closest to the failing edges. Section VIII concludes the paper.

II. Problem Statement and Terminology

A communication network is an undirected graph $G = (V, E)$, with $n = |V|$ vertices and $m = |E|$ edges. Each edge $e \in E$ has a non-negative real *length* $l(e)$. The length $|\mathcal{P}|$ of a path $\mathcal{P} = \langle p_1, \ldots, p_r \rangle$ is the sum of the lengths of its edges, and the *distance* $d(x, y)$ between two vertices x, y is the length of a shortest path between x and y. Note that throughout the paper, we measure distances in the given spanning tree T, not in the underlying graph G itself. The *hop-length* $\|\mathcal{P}\| := r - 1$ of a path $\mathcal{P}$ is the number of edges that $\mathcal{P}$ contains. Throughout the paper, we are only dealing with *simple* paths. Given a spanning tree $T = (V, E(T))$ of G, let

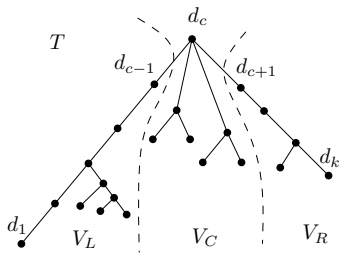

Fig. 1. A minimum diameter spanning tree T.

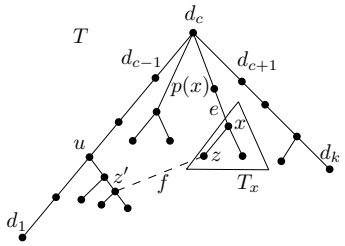

Fig. 2. A swap edge $f = (z, z')$ for $e = (x, p(x))$.

$\mathcal{D}(T) := \langle d_1, d_2, \ldots, d_k \rangle$ denote a *diameter* of T, that is, a longest path in T (see Fig. 1). Where no confusion arises, we

[1] As customary, we measure the memory requirement of the routing scheme in bits.

abbreviate $\mathcal{D}(T)$ with $\mathcal{D}$. Furthermore, define the *center* d_c of $\mathcal{D}$ as a node such that the lengths of $\mathcal{D}_L := \langle d_1, d_2, \ldots, d_c\rangle$ and $\mathcal{D}_R := \langle d_c, d_{c+1}, \ldots, d_k\rangle$ satisfy $|\mathcal{D}_L| \geq |\mathcal{D}_R|$ and have the smallest possible difference $|\mathcal{D}_L| - |\mathcal{D}_R|$. The set of neighbors of a node z (excluding z itself) in G and in T is written as $N_G(z)$ and $N_T(z) \subseteq N_G(z)$, respectively.

Let T be rooted at d_c, and let, for each node $x \neq d_c$, node $p(x)$ be the *parent* of x and $C(x)$ the set of its *children*. Furthermore, let $T_x = (V(T_x), E(T_x))$ be the subtree of T rooted at x, including x. Let V_L (L stands for "left") be the set of nodes in the subtree rooted at d_{c-1}, V_R the set of nodes in the subtree rooted at d_{c+1}, and V_C all other nodes.

Now, the removal of any edge $e = (x, p(x))$ of T partitions the spanning tree into two trees T_x and $T\backslash T_x$ (see Fig. 2), where $T\backslash T_x$ denotes the graph with vertex set $V(T)\backslash V(T_x)$ and edge set $E(T)\backslash E(T_x)\backslash\{(x, p(x))\}$. Note that $T\backslash T_x$ does not contain the node x. A *swap edge* f for e is any edge in $E\backslash E(T)$ that (re-)connects T_x and $T\backslash T_x$, i.e., for which $T_{e/f} := (V(T), E(T)\backslash\{e\} \cup \{f\})$ is a spanning tree of $G\backslash\{e\} := (V, E\backslash\{e\})$.

Let $S(e)$ be the set of swap edges for e. A *best swap edge* for e is any edge $f \in S(e)$ for which $|\mathcal{D}(T_{e/f})|$ is minimum. A *local* swap edge of node z for some failing edge e is an edge in $S(e)$ adjacent to z. The *distributed all best swaps* problem for a MDST is the problem of finding for every edge $e \in E(T)$ a best swap edge (with respect to the diameter), or to determine that no swap edge for e exists. Throughout the paper, let $e = (x, p(x))$ denote a failing edge and $f = (z, z')$ a swap edge, where z is a node inside T_x, and z' a node in $T\backslash T_x$.

III. Algorithmic Setting and Basic Idea

In our setting, nodes have unique identifiers that possess a linear order. Each node knows its own neighbors in T and in G, and for each neighbor the length of the corresponding edge. At the end of the distributed computation, for every edge $e = (x, p(x))$ of T, the selected best swap edge f (if any exists) must be known to the nodes x and $p(x)$ (but not necessarily to any other nodes). We assume port-to-port communication between neighbouring nodes. The distributed system of nodes is totally asynchronous. Each message sent from some node to one of its neighbors arrives eventually (there is no message loss). As usual, we define the asynchronous time complexity of an algorithm as the longest possible execution time assuming that sending a message requires at most one time unit. Furthermore, nodes do not need to know the total number of nodes in the system (although it is easy to count the nodes in T using a convergecast).

A. The Basic Idea

Our goal is to compute, for each edge of T, a best swap edge. A swap edge for a given failing edge $e = (x, p(x))$ must connect the subtree of T rooted at x to the part of the tree containing $p(x)$. Thus, a swap edge must be adjacent to some node inside T_x. If each node in T_x considers its own local swap edges for e, then in total all swap edges for e are considered. Therefore, each node inside T_x finds a best *local* swap edge, and then participates in a *minimum finding process* that computes a (globally) best swap edge for e. The computation of the best local swap edges is composed of three main phases: In a first (preprocessing) phase, a root of the MDST is chosen, and various pieces of information (explained later) are computed for each node. Then, in a second (top-down) phase each node computes and forwards some "enabling information" (explained later) for each node in its own subtree. This information is collected and merged in a third (bottom-up) phase, during which each node obtains its best local swap edge for each (potentially failing) edge on its path to the root. The efficiency of our algorithm will be due to our careful choice of the various pieces of information that we collect and use in these phases.

To give an overview, we now briefly sketch how each node computes a best local swap edge. First observe that after replacing edge e by f, the resulting diameter is longer than the previous diameter only if there is a path through f which is longer than the previous diameter, in which case the path through f is the new diameter. In this case, the length of the diameter equals the length of a longest path through f in the new tree. For a local swap edge $f = (z, z')$ connecting nodes $z \in V(T_x)$ and $z' \in V\backslash V(T_x)$, such a path consists of

(i) a longest path inside $T\backslash T_x$ starting in z',
(ii) edge f, and
(iii) a longest path inside T_x starting in z.

Part (i) is computed in a preprocessing phase, as described in Section V. Part (ii) is by assumption known to z, because f is adjacent to z. Part (iii) is inductively computed by a process starting from the root x of T_x, and stopping in the leaves, as follows. A path starting in z and staying inside T_x either descends to a child of z (if any), or goes up to $p(z)$ (if $p(z)$ is still in T_x) and continues within $T_x\backslash T_z$. For the special case where $z = x$, node x needs to consider only the weighted heights of the subtrees rooted at its children, where the weighted height of a subtree denotes the maximum length of any path from the subtree root to a leaf in the subtree. All other nodes z in T_x additionally need to know the length of a longest path starting at $p(z)$ and staying inside $T_x\backslash T_z$. This additional *enabling information* will be computed by $p(z)$ and then be sent to z.

Once the best local swap edges are known, a best (global) swap edge is identified by a single minimum finding process that starts at the leaves of T_x and ends in node x. To compute all best swap edges of T, this procedure is executed separately for each edge of T. This approach will turn out to work with the desired efficiency:

Main Theorem. *All best swap edges of a MDST can be computed in an asynchronous distributed setting with $O(n^* + m)$ messages of constant size, and in $O(\|\mathcal{D}\|)$ time.*

Note that a naive algorithm, in which each node sends its topology information to the root, where the solution is computed and then broadcast to all nodes, would be a lot less efficient than the above bounds in several respects: Since the root must receive $\Theta(m)$ edge weights in this naive algorithm, possibly through only a constant number of edges (e.g. if the MDST has constant degree), its running time would be

$\Omega(m)$, possibly much higher than $O(\|D\|)$ (which is $O(n)$). Furthermore, sending information about the m edges to the root would require $\Omega(mn)$ constant size messages in some cases, as the information might have to travel through $\Omega(n)$ intermediate nodes. Reducing the number of messages to $O(n)$ would be possible by increasing the message size from constant to $O(m)$, which however does not seem practical.

We will prove the above theorem in the next sections, by proving that the preprocessing phase can be realized with $O(m)$ messages, and after that the computation of all best swap edges requires at most $O(n^*)$ additional messages.

Our algorithm requires that each node knows which of its neighbors are children and which neighbor is its parent in T. Although this information is not known a priori, it can be easily computed in a preprocessing phase, during which a diameter and a root of T are selected.

IV. How to Pick a Best Swap Edge

In our distributed algorithm, we compute for each (potentially) failing edge the resulting new diameter for each possible swap edge candidate. This approach can be made efficient by exploiting the structure of changes of the diameter path, as described in the following.

A. The Structure of Changes of the Diameter Path

For a given failing edge e, let $\mathcal{P}_f$ be a longest path in $T_{e/f}$ that goes through swap edge f for e. Then, we have the following:

Lemma 1: The length of the diameter of $T_{e/f}$ is $|\mathcal{D}(T_{e/f})| = \max\{|\mathcal{D}(T)|, |\mathcal{P}_f|\}$.

Proof: Let T_1 and T_2 be the parts into which T is split if e is removed. It is easy to see that

$$|\mathcal{D}(T_{e/f})| = \max\{|\mathcal{D}(T_1)|, |\mathcal{D}(T_2)|, |\mathcal{P}_f|\}. \quad (1)$$

Since T is a MDST, we have

$$|\mathcal{D}(T_{e/f})| \geq |\mathcal{D}(T)|. \quad (2)$$

Because T_1 and T_2 are contained in T,

$$|\mathcal{D}(T_1)| \leq |\mathcal{D}(T)| \quad \text{and} \quad |\mathcal{D}(T_2)| \leq |\mathcal{D}(T)|. \quad (3)$$

If $|\mathcal{P}_f| \geq |\mathcal{D}(T)|$, it is clear that $|\mathcal{P}_f|$ is a largest term in (1), so the claim holds. On the other hand, if $|\mathcal{P}_f| < |\mathcal{D}(T)|$, then either T_1 or T_2 must contain a diameter of length exactly $|\mathcal{D}(T)|$ (otherwise, either (2) or (3) would be violated). Thus, the claim holds also in this case. ∎

That is, for computing the resulting diameter length for a given swap edge $f = (z, z')$ for e, we only need to compute the length of a longest path in $T_{e/f}$ that goes through f. For node z in the subtree T_x of T rooted in x, and z' outside this subtree, such a path $\mathcal{P}_f$ consists of three parts. To describe these parts, let $\mathcal{L}(H, r)$ denote a longest path starting in node r and staying inside the graph H. The first part is a longest path $\mathcal{L}(T\backslash T_x, z')$ in $T\backslash T_x$ that starts in z'. The second part is the edge f itself. The third part is a longest path $\mathcal{L}(T_x, z)$ starting in z and staying inside T_x. This determines the length of a longest path through f as $|\mathcal{P}_f| = |\mathcal{L}(T_x, z)| + l(f) + |\mathcal{L}(T\backslash T_x, z')|$.

B. Distributed Computation of $|\mathcal{L}(T_x, z)|$

For a given failing edge $e = (x, p(x))$, each node z in T_x needs its $|\mathcal{L}(T_x, z)|$ value to check for the new diameter when using a swap edge. This is achieved by a distributed computation, starting in x. As x knows the heights of the subtrees of all its children (from the preprocessing), it can locally compute the height of its own subtree T_x as $|\mathcal{L}(T_x, x)| = \max_{q \in C(x)}\{l(x, q) + \text{height}(T_q)\}$, where $C(x)$ is the set of children of x. For a node z in the subtree rooted at x, a longest simple path either goes from z to its parent and hence has length $|\mathcal{L}(T_x\backslash T_z \cup \{(z, p(z))\}, z)|$, or goes into the subtree of one of its children and hence has length $|\mathcal{L}(T_z, z)|$ (see Fig. 3). The latter term has just been described, and the former can be computed by induction by the parent r of z and can be sent to z. This inductive step is identical to the step just described, except that z itself is no candidate subtree for a path starting at r in the induction. In total, each node r computes, for each of its children $q \in C(r)$, the value of

$$|\mathcal{L}(T_x\backslash T_q \cup \{(q, r)\}, q)| = l(q, r) + \max\Big\{ |\mathcal{L}(T_x\backslash T_r \cup \{(r, p(r))\}, r)|, \max_{s \in C(r), s \neq q}\{l(r, s) + \text{height}(T_s)\} \Big\},$$

and sends it to q, where we assume that the value $|\mathcal{L}(T_x\backslash T_r \cup \{(r, p(r))\}, r)|$ was previously sent to r by $p(r)$.

A bird's eye view of the process shows that each node z first computes $|\mathcal{L}(T_x, z)|$, and then computes and sends $|\mathcal{L}(T_x\backslash T_q \cup \{(q, z)\}, q)|$ to each of its children $q \in C(z)$. Computation of the $|\mathcal{L}(T_x, z)|$ values finishes in T_x's leaves. Note that a second value will be added to the enabling information if $(x, p(x)) \in \mathcal{D}$, for reasons explained in the next section.

C. Distributed Computation of $|\mathcal{L}(T\backslash T_x, z')|$

In the following, we explain how z can compute $|\mathcal{L}(T\backslash T_x, z')|$ for a given swap edge $f = (z, z')$. In case the failing edge $e = (x, p(x)) \notin \mathcal{D}$, we show below that the information obtained in the preprocessing phase is sufficient.

For the sake of clarity, we analyze two cases separately, starting with the simpler case.

Case 1: The removed edge e is not on the diameter. For this case, we know from [4] that at least one of the longest paths in $T\backslash T_x$ starting from z' contains d_c. If $z' \in V_L$, we get a longest path from z' through d_c by continuing on the diameter up to d_k, and hence we have $|\mathcal{L}(T\backslash T_x, z')| = d(z', d_c) + |\mathcal{D}_R|$. If z' is in V_C or V_R, some longest path from z' through d_c continues on the diameter up to d_1, yielding $|\mathcal{L}(T\backslash T_x, z')| = d(z', d_c) + |\mathcal{D}_L|$. Remarkably, in this case $|\mathcal{L}(T\backslash T_x, z')|$ does not depend on the concrete failing edge $e = (x, p(x))$, apart from the fact that (z, z') must be a swap edge for e.

Case 2: The removed edge e is on the diameter. We analyze the case $e \in \mathcal{D}_L$, and omit the symmetric case $e \in \mathcal{D}_R$. If $z' \in V_L$ or $z' \in V_C$, we know from [4] that again, one of the longest paths in $T\backslash T_x$ starting at z' contains d_c. Thus, for $z' \in V_L$ we are in the same situation as for the failing edge not on the diameter, leading to $|\mathcal{L}(T\backslash T_x, z')| = d(z', d_c) + |\mathcal{D}_R|$. For $z' \in V_C$, after d_c a longest path may continue either on

$\mathcal{D}_R$, or continue to nodes in V_L[2]. In the latter case, the path now cannot continue on $\mathcal{D}_L$ until it reaches d_1, because edge e lies on $\mathcal{D}_L$. Instead, we are interested in the length of a longest path that starts at d_c, proceeds into V_L, but does not go below the parent $p(x)$ of x on $\mathcal{D}_L$; let us call this length $\lambda(p(x))$. As announced before, we include the $\lambda(p(x))$ value as a second value into the *enabling information* received by $p(x)$; then, we get $|\mathcal{L}(T\backslash T_x, z')| = d(z', d_c) + \max\{|\mathcal{D}_R|, \lambda(p(x))\}$. The remaining case is $z' \in V_R$. For this case (see Fig. 4), we know (from [4]) that at least one of the longest paths in $T\backslash T_x$ starting at z' passes through the node u' closest to z' on $\mathcal{D}(T)$. After u', this path may either

(i) continue on $\mathcal{D}_R$ up to d_k, or
(ii) continue through d_c going inside V_C, or
(iii) continue through d_c going inside V_L (without crossing $e = (x, p(x))$), or
(iv) continue towards d_c only up to some node d_i on $\mathcal{D}_R$, going further on non-diameter edges inside V_R.

Option (i) yields a length of $d(z', d_k) = d(z', u') + d(u', d_k) = d(z', u') + (|\mathcal{D}_R| - d(d_c, u'))$. Option (ii) requires the term γ, denoting the length of a longest path starting in d_c and consisting only of nodes in V_C. The length of the path using this option is then $d(z', d_c) + \gamma$. Option (iii) yields the length $d(z', d_c) + \lambda(p(x))$.

It remains to show how the length of a longest path of the last type (Option (iv)) can be found efficiently. We propose to combine three lengths, in addition to the length of the path from z' to u'. The first is the length of a longest path inside V_R that starts at d_k; let us call this length μ_R. In general, this path goes up the diameter path $\mathcal{D}_R$ for a while, and then turns down into a subtree of V_R, away from the diameter, at a diameter node that we call ρ_R (see Fig. 4). Given μ_R, the distance from u' to ρ_R, and the distance from ρ_R to d_k, the desired path length of an upwards turning path inside V_R is $d(z', u') + d(u', \rho_R) + \mu_R - d(d_k, \rho_R)$. Note that while it may seem that ρ_R needs to lie above u' on $\mathcal{D}_R$, this is not really needed in our computation, because the term above will not be larger than Option (i) if ρ_R happens to be at u' or below. Furthermore, in this case Option (iv) cannot be better than Option (i) and thus need not be considered.

In total, we get

$$|\mathcal{L}(T\backslash T_x, z')| = \max\Big\{d(z', d_k), \quad d(z', d_c) + \gamma, \\ d(z', d_c) + \lambda(p(x)), \\ d(z', u') + d\big(u', \rho_R\big) + \mu_R - d\big(d_k, \rho_R\big)\Big\}.$$

All of these path length computations can be carried out locally with no message exchanges, if the constituents of these sums are available locally at a node. We will show in the next section how to achieve this in an efficient preprocessing phase.

D. The BESTDIAMSWAP Algorithm

For a given edge $e = (x, p(x))$ that may fail, each node z in the subtree T_x rooted at x executes the following steps:

[2]The option of going back into V_C can be ignored because it cannot yield a path longer than $\mathcal{D}_R$.

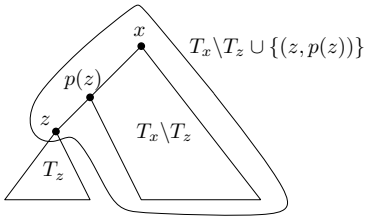

Fig. 3. Illustration of the tree $T_x\backslash T_z \cup \{(z, p(z))\}$.

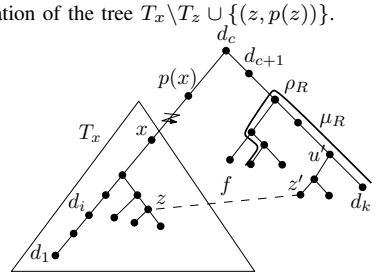

Fig. 4. Computing $|\mathcal{L}(T\backslash T_x, z')|$ if $e \in \mathcal{D}_L$, $z \in V_L$ and $z' \in V_R$.

(i) Wait for the enabling information from the parent (unless $x = z$), and then compute $|\mathcal{L}(T_x, z)|$. Compute the enabling information for all children and send it.
(ii) For each local swap edge $f = (z, z')$, compute $|\mathcal{L}(T\backslash T_x, z')|$ as described in Section IV-C.
(iii) For each local swap edge $f = (z, z')$, locally compute

$$|\mathcal{D}(T_{e/f})| = \max\Big\{|\mathcal{D}(T)|, \\ |\mathcal{L}(T_x, z)| + l(f) + |\mathcal{L}(T\backslash T_x, z')|\Big\}.$$

Among these, choose a best swap edge f^*_{local} and store the resulting new diameter as $|\mathcal{D}(T_{e/f^*_{local}})|$. If no local swap edge exists, then create a "dummy" candidate whose diameter length is ∞.
(iv) From each child $q \in C(z)$, receive the node label of a best swap edge candidate f^*_q and its resulting diameter $|\mathcal{D}(T_{e/f^*_q})|$. Pick a best swap edge candidate f^*_b among these, i.e., choose $b := \arg\min_{q \in C(z)} |\mathcal{D}(T_{e/f^*_q})|$. Compare the resulting diameter of f^*_b and f^*_{local}, and define f_{best} as the edge achieving the smaller diameter (or any of them if their length is equal), and its diameter as $|\mathcal{D}(T_{e/f_{best}})|$.
(v) Send the information f_{best}, $|\mathcal{D}(T_{e/f_{best}})|$ to the parent.

The above algorithm computes the best swap edge for one (potentially) failing edge e, based on the information available after the preprocessing phase. In order to compute all best swap edges of T, we execute this algorithm for each edge of T independently. A pseudocode description of Algorithm BESTDIAMSWAP is given in Fig. 5. BESTDIAMSWAP in turn uses Algorithm LONGEST, which is described in Fig. 6.

Analysis of the Algorithm: We now show that the proposed algorithm indeed meets our efficiency requirements:

Theorem 1: After preprocessing, executing the BESTDIAMSWAP algorithm independently for each and every edge $e \in E(T)$ costs at most $O(n^*)$ messages of size $O(1)$ each, and $O(\|\mathcal{D}\|)$ time.

Proof: Correctness follows from the preceding discussion. Preprocessing ensures that all precomputed values defined for

Algorithm: BESTDIAMSWAP(x, z).
Describes how node z computes a best swap for $e = (x, p(x))$. Note that BESTDIAMSWAP(x, z) is executed for each tree edge $e \in E(T)$ separately, using a farthest-first contention resolution policy.

if $z = x$ **then**
  $m = 0$
else
  $\{\ z \neq x\ \}$
  Wait until information $m = \mathcal{L}(T_x \backslash T_z, z)$ is received from parent.
  if $x \in \mathcal{D}$ **then** wait for the information $\lambda(p(x))$ from parent.
end
$\mathcal{L}(T_x, z) = \max\left\{m, \max_{q \in C(z)}\{l(z,q) + \text{height}(T_q)\}\right\}$
for *each local swap edge* $f = (z, z')$ **do**
  $|\mathcal{P}_f| = \mathcal{L}(T_x, z) + l(f) +$ LONGEST(e, f)
  $|\mathcal{D}(T_{e/f})| = \max\{|P_f|, |\mathcal{D}(T)|\}$
end
for *each child* $q \in C(z)$ **do**
  compute the enabling information:
  $\mathcal{L}(T_x \backslash T_q \cup \{(q,z)\}, q) = l(z,q) +$
  $\max\left\{m, \max_{s \in C(z), s \neq q}\{l(z,s) + \text{height}(T_s)\}\right\}$
  if $x \in \mathcal{D}$ **then** append $\lambda(p(x))$ to the enabling information and
  send it to q.
end
wait until all children have sent back their best swap candidate
select one best of these and the local swap candidates
if *there is no swap candidate* **then**
  $\{$ all children have sent a "dummy" candidate, and there are no local swap candidates $\}$
  create a "dummy" candidate whose diameter length is ∞.
if $z = x$ **then**
  store that swap edge as the best swap edge for e
  inform $p(x)$ about the best swap edge.
else
  $\{\ z \neq x\ \}$
  send that swap edge to $p(z)$
end

Fig. 5. Algorithm BESTDIAMSWAP(x, z).

Algorithm: LONGEST$(e = (x, p(x)), f = (z, z'))$.
Returns the length $|\mathcal{L}(T \backslash T_x, z')|$ of a longest path in $T \backslash T_x$ that starts in z'.

Input: an edge $e = (x, p(x))$ whose best swap edge shall be computed, and a local swap edge $f = (z, z')$.

if e *is on the diameter (i.e.,* $x \in \mathcal{D}(T)$*)* **then**
  if $x \in V_L$ **then** $\{\ e \in V_L\ \}$
    if $z' \in V_L$ **then**
      $\{$ since one longest path through f goes through d_c: $\}$
      return $d(z', d_c) + |\mathcal{D}_R|$
    else if $z' \in V_C$ **then**
      $\{$ since one longest path through f goes through d_c: $\}$
      return $d(z', d_c) + \max\{|\mathcal{D}_R|, \lambda(p(x))\}$
    else if $z' \in V_R$ **then**
      $\{$ Let u' be the nearest ancestor of z' on the diameter. One longest path from z' must go through u'. $\}$
      $d(d_k, \rho_R) = |\mathcal{D}_R| - d(\rho_R, d_c)$
      $d(u', \rho_R) = |d(u', d_c) - d(\rho_R, d_c)|$
      $d(u', d_k) = |\mathcal{D}_R| - d(u', d_c)$
      $d(z', u') = d(z', d_c) - d(u', d_c)$
      $from\text{-}u' = \mu_R - d(d_k, \rho_R) + d(u', \rho_R)$
      return $d(z', u') + \max\{d(u', d_k), d(u', d_c) + \lambda(p(x)), from\text{-}u'\}$
    end
  else $\{x \in V_R$: symmetric to $x \in V_L\ \}$
    $\{$ code omitted because it is completely symmetric to the above case $\}$
else $\{$ e not on the diameter $\}$
  if $z' \in V_L$ **then**
    return $d(z', d_c) + |\mathcal{D}_R|$
  else $\{$ $z' \in V_C$ or V_L, and $|\mathcal{D}_L| \geq |\mathcal{D}_R|$ $\}$
    return $d(z', d_c) + |\mathcal{D}_L|$
  end

Fig. 6. Algorithm LONGEST$(e = (x, p(x)), f = (z, z'))$.

the other end z' of a candidate swap edge are available locally at z (these values are required to compute, e.g., $|\mathcal{L}(T \backslash T_x, z')|$). As to the message complexity, consider the execution of the BESTDIAMSWAP algorithm for one particular edge $e = (x, p(x))$. Starting in node $x \in V \backslash \{d_c\}$, each node in T_x sends a message containing the "enabling information" (i.e., $\mathcal{L}(T_x \backslash T_q, q)$ and possibly $\lambda(p(x))$) containing $O(1)$ items to each of its children. Furthermore, each node in T_x (including finally x) sends another message with size $O(1)$ up to its parent in the minimum finding process. Hence, two messages of size $O(1)$ are sent across each edge of T_x, and one message is sent across e. Thus, the computation of a best swap for e requires $2 \cdot |E(T_x)| + 1 = 2 \cdot |V(T_x)| - 1$ messages. The number of messages exchanged for computing a best swap edge for *each and every* edge $(x, p(x))$ where $x \in V \backslash \{d_c\}$ is $\sum_x (2 \cdot |V(T_x)| - 1) = 2n^* - (n-1)$.

As to the time complexity, note that the best swap computation of a *single* edge according to the BESTDIAMSWAP algorithm requires at most $O(\|\mathcal{D}\|)$ time. Now note that this algorithm can be executed independently (and thus concurrently) for each potential failing edge: In this fashion, each node x in T sends exactly one message to each node in T_x during the top-down phase. Symmetrically, in the bottom-up phase, each node u in T sends exactly one message to each node on its path to the root. The crucial point here is to avoid that some of these messages block others for some time (as only one message can traverse a link at a time). Indeed, one can ensure that each message reaches its destination in $O(\|\mathcal{D}\|)$ time as follows. A node z receiving a message with destination at distance d from z forwards it only after all messages of

the protocol with a destination of distance more than d from z have been received and forwarded. By induction over the distance of a message from its destination, this "farthest-first" contention resolution policy (see also [9]) allows each message to traverse one link towards its destination after at most one time unit of waiting. Thus, the $O(\|\mathcal{D}\|)$ time complexity also holds for the entire algorithm. ■

Instead of sending many small messages individually, we can choose to sequence the process of message sending so that messages for different failing edges are bundled before sending (see also [7], [8] for applications of this idea). This leads to an alternative with fewer but longer messages:

Corollary 1: After preprocessing, the *distributed all best swaps* problem can be solved using $O(n)$ messages of size $O(n)$ each, and in $O(\|\mathcal{D}\|)$ time.

V. The Preprocessing Phase

The preprocessing phase serves the purpose of making the needed terms in the sums described in the previous section available at the nodes of the tree. Details of this phase can be found in the pseudocode given as Algorithm Preprocessing 1 and Algorithm Preprocessing 2 in Figs. 7, 8 and 9.

A. Algorithms

In the preprocessing phase, a diameter $\mathcal{D}$ of T is chosen, and its two ends d_1 and d_k as well as its center d_c are identified. This can be done essentially by a convergecast, followed by a broadcast to distribute the result (see e.g. [10]); the details are standard and therefore omitted. After preprocessing exchanges $O(n)$ messages, each node knows the information that is requested in (A) and (C) below. It is crucial that during preprocessing, each node obtains enough information to later carry out all computational steps to determine path components (i), (ii) and (iii). More precisely, each node gets the following global information (the same for all nodes):

(A) The endpoints d_1 and d_k of the diameter, the length $|\mathcal{D}|$ of the diameter, and the lengths $|\mathcal{D}_L|$ and $|\mathcal{D}_R|$.

(B) The length μ_R of a longest path starting in d_k that is fully inside $T_{d_{c+1}}$, together with the node ρ_R on $\mathcal{D}$ where this path leaves the diameter, and the distance $d(\rho_R, d_c)$. Fig. 10 illustrates such a longest path μ_R. Symmetrically, the values μ_L, ρ_L and $d(\rho_L, d_c)$ are also required.

In addition, each node z obtains the following information that is specific for z:

(C) For each child $q \in C(z)$ of its children, the weighted height of q's subtree T_q.

(D) Whether z is on the diameter $\mathcal{D}$ or not.

(E) The distance $d(z, d_c)$ from z to d_c.

(F) The identification of the parent $p(z)$ of z in T.

(G) To which of V_L, V_C and V_R does z belong.

(H) If $z \notin \mathcal{D}$, the closest ancestor u of z on the diameter; the distance $d(u, d_c)$ from u to d_c.

(I) If z is on the left (right) diameter $\mathcal{D}_L$ ($\mathcal{D}_R$), with $z = d_i$, the length $\lambda(d_i)$ of a longest path in T starting at d_c and neither containing the node d_{c+1} (d_{c-1}) nor the node d_{i-1} (d_{i+1}), nor any node from V_C (see Fig. 10).

Algorithm: Preprocessing 1 for node z: Finding a diameter.

{ Let $\tilde{T}_s$ be the subgraph of T that would stay connected to s if edge (s, z) were removed. Each message $M_s = (deepestnode, height, source, diamLen, d_1, d_k)$ from neighbor s contains the identifier of a deepest node in $\tilde{T}_s$, height$(\tilde{T}_s) + l(s, z)$, the neighbor s that sent the message, the length of a diameter of $\tilde{T}_s$, and its two endpoints d_1 and d_k. }

$Diams = \{\}$; $Heights = \{\}$
if *z is a leaf* **then**
  $last =$ the only node in $N_T(z)$
  Send $(z, l(z, last), z, 0, z, z)$ to $last$.
else { z not a leaf }
  Wait until at least $|N_T(z)| - 1$ neighbors' messages have been received.
  $last =$ node in $N_T(z)$ whose message has not yet been received, or was received last.
  for *each message M_s from neighbor $s \in N_T(z) \backslash \{last\}$* **do**
    $(deepestnode, height, s, diamLen, d_1, d_k) = M_s$
    Insert $(deepestnode, height, s)$ into $Heights$.
    Insert $(diamLen, d_1, d_k)$ into $Diams$.
  end
  $(diamLen^*, d_1^*, d_k^*) =$ Update($Heights$, $Diams$)
  $(a, height_a, s_a) =$ a tuple in $Heights$ with highest $height_a$.
  Send $(a, height_a + l(z, last), s_a, diamLen^*, d_1^*, d_k^*)$ to $last$.
end
Wait until a message (from $last$) is received.
if *the message is M_{last} from $last$ and $id(z) < id(last)$* **then**
  { Locally compute the *global* diameter of T: }
  Insert the $(deepestnode, height)$ tuple from M_{last} into $Heights$
  Insert the $(diamLen, d_1, d_k)$ tuple from M_{last} into $Diams$
  $(diamLen^*, d_1^*, d_k^*) =$ Update($Heights$, $Diams$)
  Send "$\mathcal{D} = (d_1^*, d_k^*, diamLen^*)$" to all neighbors.
else if *the message is "$\mathcal{D} = (d_1, d_k, |\mathcal{D}|)$"* **then**
  Forward the message "$\mathcal{D} = (d_1, d_k, |\mathcal{D}|)$" to all other neighbors.
end
Start Preprocessing 2.

Fig. 7. Algorithm Preprocessing 1

Procedure: Update($Heights$, $Diams$)

$(a, height_a, s_a) =$ a tuple in $Heights$ with highest $height_a$.
$(b, height_b, s_b) =$ a tuple in $Heights \backslash \{(a, height_a, s_a)\}$ with highest $height_b$.
Insert $(height_a + height_b, a, b)$ into $Diams$.
return $(diamLen^*, d_1^*, d_k^*) =$ a triple in $Diams$ with highest $diamLen^*$.

Algorithm: Preprocessing 2 for node z: Computing information about the diameter.

{ Determine if z is itself on the diameter $\mathcal{D}$:
If no message containing d_1 (d_k) as the deepest node was received, then d_1 (d_k) must be in $\tilde{T}_{last}$: }
Find $(d_1, height_1, s_1)$ tuple in $Heights$, set $s_1 = last$ if none found.
Find $(d_k, height_k, s_k)$ tuple in $Heights$, set $s_k = last$ if none found.
if $(s_1 == s_k)$ **then** { z is not on $\mathcal{D}$ }
  Upon receiving
  $M_* = (\mu_L, \rho_L, d(\rho_L, d_c), \mu_R, \rho_R, d(\rho_R, d_c), |\mathcal{D}_L|, |\mathcal{D}_R|, V_*, d(z, d_c), u, d(u, d_c))$ from neighbor q, where $V_* \in \{V_L, V_C, V_R\}$, set $parent = q$, and send
  $M_* = (\mu_L, \rho_L, d(\rho_L, d_c), \mu_R, \rho_R, d(\rho_R, d_c), |\mathcal{D}_L|, |\mathcal{D}_R|, V_*, d(z, d_c) + d(z, r), u, d(u, d_c))$ to every neighbor $r \in N_T(z) \backslash \{parent\}$.
  { Send the message M' across each non-tree edge: }
  Send $M' = (d(z, d_c), V_*, u', d(u', d_c))$ to all neighbors $z' \in N_G(z) \backslash N_T(z)$.
  return
end
{ z is on $\mathcal{D}$. In this case, z knows at least one of the distances $d(z, d_1) = \text{height}(\tilde{T}_{d_1})$ and $d(z, d_k) = \text{height}(\tilde{T}_{d_k})$. }
if $s_1 == last$ **then** $height_1 = |\mathcal{D}| - height_k$
if $s_k == last$ **then** $height_k = |\mathcal{D}| - height_1$
{ $height_1 = d(z, d_1)$ and $height_k = d(z, d_k)$. }

Fig. 8. Algorithm Preprocessing 2

(J) For each of the neighbors z' of z in G, which of V_L, V_C and V_R contains z'; the distance $d(z', d_c)$ from z' to d_c; the nearest ancestor u' of z' on $\mathcal{D}$, the distance $d(u', d_c)$.

Computing the Additional Information: Recall that the first preprocessing part ends with a broadcast that informs all nodes about the information described in (A) and (C). The second part of the preprocessing phase follows.

A node z receiving the message about $\mathcal{D}$ can infer from the previous convergecast whether it belongs to $\mathcal{D}$ itself by just checking whether the paths from z to d_1 and d_k go through the same neighbor of z.

Information (E) is obtained by having the center node send a "distance from d_c" message to both neighbors d_{c+1} and d_{c-1} on $\mathcal{D}$, which is forwarded and updated on the diameter[3]. This information is used by the diameter nodes for computing $\lambda(d_i)$, required in (I). The center initiates the inductive computation of $\lambda(d_i)$:

- $\lambda(d_c) = 0$.

[3]The message is updated as follows: when forwarded through an edge on the diameter, the length of this edge is added to the forwarded distance. This ensures that each node which receives the message obtains its own distance from d_c. Details are described in Figure 9.

if $\big(height_1 \geq height_k\big) \wedge \big(height_1 - l(z, s_1) < height_k + l(z, s_1)\big)$ **then** { z is d_c }
  $\lambda(d_c) = 0$
  Send $\big(\lambda(d_c), l(s_1, z)\big)$ to s_1 and $\big(\lambda(d_c), l(s_k, z)\big)$ to s_k
  $|\mathcal{D}_L| = height_1$; $|\mathcal{D}_R| = height_k$
  Receive $\big(\mu, \rho, d(\rho, d_c), d_1\big)$ from s_1 and $\big(\mu', \rho', d(\rho, d_c)', d_k'\big)$ from s_k, and set
  $(\mu_L, \rho_L, d(\rho_L, d_c) = \big(\mu, \rho, d(\rho, d_c), d_1\big)$,
  $(\mu_R, \rho_R, d(\rho_R, d_c) = \big(\mu, \rho, d(\rho, d_c), d_k\big)$.
  Forward $M_* = (\mu_L, \rho_L, d(\rho_L, d_c), \mu_R, \rho_R, d(\rho_R, d_c), |\mathcal{D}_L|, |\mathcal{D}_R|, V_L, 0)$ to d_{c-1}.
  Forward $M_* = (\mu_L, \rho_L, d(\rho_L, d_c), \mu_R, \rho_R, d(\rho_R, d_c), |\mathcal{D}_L|, |\mathcal{D}_R|, V_R, 0)$ to d_{c+1}.
  Forward $M_* = (\mu_L, \rho_L, d(\rho_L, d_c), \mu_R, \rho_R, d(\rho_R, d_c), |\mathcal{D}_L|, |\mathcal{D}_R|, V_C, 0)$ to all neighbors in $N_T(d_c) \backslash \{d_{c-1}, d_{c+1}\}$.
else
  Wait for the message containing λ and $d(z, d_c)$.
  $(i, height_i, s_i)$ = a tuple in $Heights \backslash \{(s_1, \cdot, \cdot), (s_k, \cdot, \cdot)\}$ with largest $height_i$.
  Compute $\lambda(z) = \max\{\lambda, d(z, d_c) + height_i\}$.
  Send $\big(\lambda(z), d(z, d_c) + l(z, s_1)\big)$ to s_1, and $\big(\lambda(z), d(z, d_c) + l(z, s_k)\big)$ to s_k.
  if z *is* d_1 **then** $\mu(d_1) = 0$; Send $\big(\mu(d_1), d_1, d(d_1, d_c), d_1\big)$ to s_k
  else if z *is* d_k **then** $\mu(d_k) = 0$; Send $\big(\mu(d_k), d_k, d(d_k, d_c), d_k\big)$ to s_1.
  else { z is on $\mathcal{D}$, but $z \notin \{d_1, d_c, d_k\}$ }
    Upon receiving $\big(\mu, \rho, d(\rho, d_c), d_*\big)$, where $d_* \in \{d_1, d_k\}$, compute
    $\mu(z) = \max\{\mu, d(d_*, z) + height_i\}$ and set
    $\rho(z) = z$ and $dist = d(z, d_c)$ if $\mu(z) > \mu$, and
    $\rho(z) = \rho$ and $dist = d(\rho, d_c)$ otherwise.
    Forward $\big(\mu(z), \rho(z), dist\big)$ along the diameter.
    Upon receiving
    $M_* = (\mu_L, \rho_L, d(\rho_L, d_c), \mu_R, \rho_R, d(\rho_R, d_c), |\mathcal{D}_L|, |\mathcal{D}_R|, V_*, d(z, d_c))$ from one neighbor on the diameter, where $V_* \in \{V_L, V_C, V_R\}$, send
    $M_* = (\mu_L, \rho_L, d(\rho_L, d_c), \mu_R, \rho_R, d(\rho_R, d_c), |\mathcal{D}_L|, |\mathcal{D}_R|, V_*, d(z, d_c) + d(z, q))$ to the other neighbor q on the diameter, and send
    $M_* = (\mu_L, \rho_L, d(\rho_L, d_c), \mu_R, \rho_R, d(\rho_R, d_c), |\mathcal{D}_L|, |\mathcal{D}_R|, V_*, d(z, d_c) + d(z, r), u, d(u, d_c))$ to all other neighbors $r \in N_T(z) \backslash \{q\}$
end
Send $M' = (d(z, d_c), V_*, u', d(u', d_c))$ to all neighbors $z' \in N_G(z) \backslash N_T(z)$.

Fig. 9. Algorithm Preprocessing 2, continued

- For each d_j, $1 \leq j < c$,

$$\lambda(d_j) = \max\{\lambda(d_{j+1}), d(d_c, d_j) + h_2(d_j)\},$$

  h_2 being the weighted height of a highest subtree of d_j apart from the diameter subtree.
- For each d_j, $c < j \leq k$,

$$\lambda(d_j) = \max\{\lambda(d_{j-1}), d(d_c, d_j) + h_2(d_j)\}.$$

In order to compute μ_L and μ_R as required in (B), we define $\mu(d_i)$ for each node d_i on $\mathcal{D}_L$ as the length of a longest path starting in d_1 that is fully inside T_{d_i}, together with the node $\rho(d_i)$ on $\mathcal{D}_L$ where such a path leaves the diameter. For d_i on $\mathcal{D}_R$, the definition is symmetric. We then have $\mu_L = \mu(d_{c-1})$ and $\mu_R = \mu(d_{c+1})$. The inductive computation of $\mu(d_i)$ is started by d_1 and d_k, and then propagated along the diameter:

- $\mu(d_1) = \mu(d_k) = 0$;
- for each d_j, $1 < j < c$,

$$\mu(d_j) = \max\{\mu(d_{j-1}), d(d_1, d_j) + h_2(d_j)\};$$

- for each d_j, $c < j < k$,

$$\mu(d_j) = \max\{\mu(d_{j+1}), d(d_k, d_j) + h_2(d_j)\}.$$

Along with $\mu(d_j)$, $\rho(d_j)$ and $d(\rho(d_j), d_c)$ can be computed as well. The computation stops in d_c, which receives the messages $(\mu(d_{c-1}), \rho(d_{c-1}), d(\rho(d_{c-1}), d_c)) = (\mu_L, \rho_L, d(\rho_L, d_c))$ and $(\mu(d_{c+1}), \rho(d_{c+1}), d(\rho(d_{c+1}), d_c)) = (\mu_R, \rho_R, d(\rho_R, d_c))$.
Altogether, this second preprocessing part operates along the diameter and takes $O(\|\mathcal{D}(T)\|) = O(n)$ messages.

Distributing the Information: When the computation of the two triples $(\mu_L, \rho_L, d(\rho_L, d_c))$ and $(\mu_R, \rho_R, d(\rho_R, d_c))$ completes in d_c, the center packs these values plus the values $|\mathcal{D}_L|$ and $|\mathcal{D}_R|$ into one message M_*. It adds the appropriate one of the labels "V_L","V_R" and "V_C" to M_*, before forwarding M_* to d_{c-1}, d_{c+1} and any other neighbor of d_c in T and then flooding the tree. Additionally, M_* contains the "distance from d_c" information which is updated on forwarding, such that all nodes know their distance to the center[4]. When M_* is forwarded from a node $u \in \mathcal{D}$ to a node not on $\mathcal{D}$, it is extended by the "distance from u" information, which is also updated on forwarding. In addition, $d(u, d_c)$ is appended to M_*. Finally, if node z receives M_* from node v, then z learns that v is its parent.

At the end of this second part of the preprocessing phase, each node z' sends a message M' to each of its neighbors z in $G \backslash T$. Note that this is the only point in our solution where messages need to be sent over edges in $G \backslash T$. M' contains $d(z', d_c)$ and exactly one of { "$z' \in V_L$", "$z' \in V_C$" , "$z' \in V_R$" }, whichever applies. Furthermore, let u' be the nearest ancestor of z' on $\mathcal{D}$; the distance $d(u', d_c)$ is also appended to M'.

As a consequence, after each node has received its version of the message M_*, the information stated in (B), (E), (F), (G), (H) is known to each node. Furthermore, each node that has received M' from all its neighbors in G knows the information stated in (J). The distribution of this information requires $O(\|\mathcal{D}(T)\|)$ time and $O(m)$ messages. Let us summarize.

Lemma 2: After the end of the two parts of the preprocessing phase, which requires $O(\|\mathcal{D}\|)$ time, all nodes know all information (A)–(J), and $O(m)$ messages have been exchanged.

Recognizing Swap Edges Using Labels: A node $v \in T_x$ must be able to tell whether an incident edge $f = (v, w)$ is a swap edge for $e = (x, p(x))$ or not. We achieve this by the folklore method of attaching two labels to each node: The first label is the node's number in a preorder traversal, while the second is its number in a reverse preorder traversal. For any two nodes, a simple comparison of both respective labels tells whether one node lies in the subtree of the other node (see, e.g., [7], [8]).

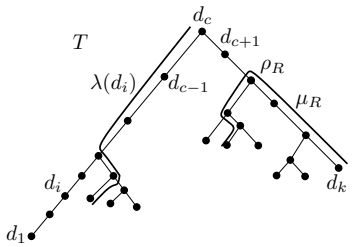

Fig. 10. Definition of $\lambda(d_i)$, μ_R and ρ_R.

VI. Routing Issues

A natural question arises concerning routing in the presence of a failure: After replacing the failing edge e by a best swap edge f, how do we adjust our routing mechanism in order to guide messages to their destination in the new tree $T_{e/f}$? And how is routing changed back again after the failing edge has been repaired? Clearly, it is desirable that the adaptation of the routing mechanism is as fast and inexpensive as possible.

Existing Approaches: The simplest routing scheme uses a routing table of n entries at each node, which contains, for each possible destination node, the link that should be chosen for forwarding. This approach can be modified to allow swaps by storing additional n entries for the swap links at each node [7]. In [3] a scheme is proposed that stores only one swap entry, at the cost of choosing suboptimal swap edges. All these approaches require $O(n^2)$ routing entries in total.

In the following, we propose to use a *compact* routing scheme for arbitrary trees (shortest paths, minimum diameter, or any other) which requires only δ entries, i.e. $\delta c \log n$ bits, at a node of degree δ, thus n entries or $2mc \log n$ bits in total, which is the same amount of space that the *interval routing* scheme of [11] requires. The header of a message requires $c \log n$ bits to describe its destination.

Our Routing Scheme: Our routing scheme for trees is based on the labeling $\gamma : V \to \{1, \ldots, n\}^2$ described at the end of Section V-A. Recall that γ allows to decide in constant time whether a is in the subtree of b (i.e., $a \in T_b$) for any two given nodes a and b.

[4]The nodes on $\mathcal{D}$ already have that information at this point, but all other nodes still require it.

Basic Routing Algorithm:
A node s routes message M with destination d as follows:

(i) If $d = s$, M has arrived at its destination.
(ii) If $d \notin T_s$, s sends M to $p(s)$.
(iii) Otherwise, s sends M to the child $q \in C(s)$ for which $d \in T_q$.

This algorithm clearly routes each message directly on its (unique) path in T from s to d. Before describing the adaptation in the presence of a swap, observe that a node s which receives a message M with destination d can locally decide whether M traverses a given edge $e = (x, p(x))$: edge e is used by M if and only if exactly one of s and d is in the subtree T_x of x, i.e., if $(s \in T_x) \neq (d \in T_x)$. Thus, it is enough to adapt routing if all nodes are informed about a failing edge (and later the repair) by two broadcasts starting at its two incident nodes (the points of failure). However, the following lemma shows that optimal rerouting is guaranteed even if only those nodes which lie on the two paths between the points of failure and the swap edge's endpoints are informed about failures.

Lemma 3: Let $e = (x, p(x))$ be a failing edge, and $f = (z, z')$ a best swap of e, where z is in T_x and z' in $T \backslash T_x$, as shown in Fig. 11. If all nodes on the path from x to z know that e is unavailable and that $f = (z, z')$ is a best swap edge, then any message originating in $s \in T_x$ will be routed on the direct path from s to its destination d. Symmetrically, if all nodes on the path from $p(x)$ to z' know about e and f, then any message originating in $s' \in T \backslash T_x$ will be routed on the direct path from s' to its destination d'.

Proof: Let M be any message with source $s \in T_x$. If $d \in T_x$, then trivially M will be routed on its direct path, because it does not require edge e. If $d \in T \backslash T_x$, consider the path $\mathcal{P}_T$ from s to d in T, and the path $\mathcal{P}_{T_{e/f}}$ from s to d in $T_{e/f}$. Consider the last common node i of $\mathcal{P}_T$ and $\mathcal{P}_{T_{e/f}}$ in T_x. The path composed of the paths $\langle x, \ldots, i \rangle$, $\langle i, \ldots, z \rangle$ is exactly the unique path in T from x to z, so node i lies on that path. Obviously, M will be routed on the direct path

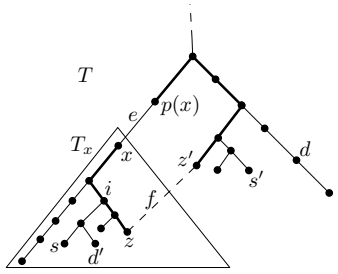

Fig. 11. Only some nodes need to know about failure of edge $e = (x, p(x))$.

towards d up to i. As i lies on the path from x to z, it knows about the failure and the swap, and will route M towards z. Because each node on the path from i to z also knows about the swap M will proceed on the direct path to z. At z, M will be routed over the swap edge f, and from z' onwards M is forwarded on the direct path from z' to d. ■

Given Lemma 3, we propose the following "lazy update" procedure for informing nodes about an edge failure:

Algorithm SWAP:
If an edge fails, no action is taken as long as no message needs to cross the failed edge. As soon as a message M which should be routed over the failing edge arrives at the point of failure, information about the failure and its best swap is attached to message M, and M is routed towards the swap edge. On its way, all nodes which receive M route it further towards the swap edge, and remember for themselves the information about the swap.

Observation (Adaptivity). *After one message M has been rerouted from the point of failure to the swap edge, all messages originating in the same side of T as M (with respect to the failing edge) will be routed to their destination on the direct path in the tree (i.e., without any detour via the point of failure).*

If a failing edge has been replaced by a swap edge, then all nodes which know about that swap must be informed when the failure has been repaired. Therefore, a message is sent from the point of failure to the swap edge (on both sides if necessary), to inform these nodes, and to deactivate the swap edge.

Remark: The above routing scheme has the disadvantage that each node must know the labels of all its neighbors. Thus, an individual node is potentially required to store much more than $O(\log n)$ bits. This drawback could be removed by combining the above scheme with a compact routing scheme for the designer-port model, see e.g. [12]: Such a routing scheme assigns a label of $O(\log n)$ bits to every node, such that the correct forwarding port for a given destination can be computed solely on the basis of the labels of the current position and of the destination. The labels we introduced in our scheme are then only used to determine whether a message needs to be rerouted (because it would otherwise need to use the failing edge). As this is possible solely on the basis of the labels of the message's current position and its destination, this combination yields a compact routing scheme which can efficiently adapt to swaps.

VII. FINDING SWAPS CLOSE TO THE FAILURE

In the routing scheme described in Section VI, the time required to adapt to an edge failure by activating a swap edge depends on the lengths of the paths between the two points of failure and the corresponding two endpoints of the swap edges. Two different possible models of failure detection seem reasonable:

1. The failure of an edge is detected at both of its endpoints concurrently.
2. The failure of an edge is detected at one of its endpoints only.

Of course, in some systems it might be unknown in advance whether one or both endpoints will detect a given failure. In such a system, the latter of the above variants would minimize the worst case time to adapt routing.

If the prime goal is to reconnect the network quickly after an edge failure, and the quality of the resulting tree is less important, then one should precompute swap edges which are "closest" to the failing edge. In the following, we present

two efficient distributed algorithms for computing such swap edges in both models of failure detection. Both of these algorithms employ the same basic principle as the algorithm BESTDIAMSWAP described in Section IV-D: for each failing edge $e = (x, p(x))$, first all nodes in the subtree T_x compute their locally best swap edge, and then a minimum finding process finds a globally best swap edge for e. The difference lies in the means of computing the quality of a given swap edge candidate f, given a failing edge e.

A. An edge failure is detected at both endpoints

If an edge failure is detected at both endpoints of the failing edge, then the fastest way of informing all nodes which need to be informed in order to readjust routing, as identified by Lemma 3, is to send two messages, each starting at one endpoint of the failing edge $e = (x, p(x))$, to the two respective endpoints of the swap edge $f = (z, z')$. Thus, the time until this message reaches both destinations is the maximum length of the two paths connecting f's endpoints with e's endpoints, i.e., $\max\{d(x, z), d(p(x), z')\}$ (see Fig. 12).

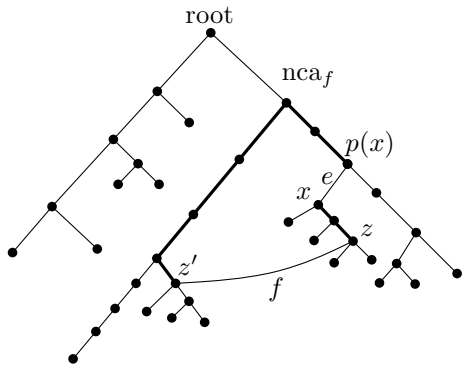

Fig. 12. Illustration of the distances $d(x, z)$ and $d(p(x), z')$.

In order to evaluate a local swap edge candidate $f = (z, z')$ for a given failing edge e, a node $z \in T_x$ must compute $\max\{d(x, z), d(p(x), z')\}$. For a given edge $f = (z, z')$, let nca_f denote the node closest to the root among all nodes on the path from z to z' in T. It is easy to see that $d(p(x), z') = d(p(x), \mathrm{nca}_f) + d(z', \mathrm{nca}_f)$. We propose to use again the folklore labeling described at the end of Section V. Using this labeling, there is a simple solution for providing z and z' with all the information required to compute the terms

$$d(z', \mathrm{nca}_f),\ d(p(x), \mathrm{nca}_f) \text{ and } d(x, z), \tag{4}$$

respectively: The root r sends its own label to all nodes in its subtree T_r. Each other node v sends the two labels of v and $p(v)$ and the distance $d(v, p(v))$ to all nodes in its subtree T_v. In total, each node in the tree then knows its entire path to the root, including the labels of all nodes and all distances. This process clearly terminates in $O(\|\mathcal{D}\|)$ time and requires $O(n^*)$ messages.

Given this information, a node can compute distances between two arbitrary ancestors of itself. Hence, node z can compute $d(x, z)$, and if nca_f were known to both z and z', then in the same way the other terms in (4) could also be computed: z computes $d(p(x), \mathrm{nca}_f)$, and z' computes $d(z', \mathrm{nca}_f)$.

It remains to show how nca_f can be computed locally. This is where we make use of the labeling scheme once again: as z' knows the label of $p(x)$, it can determine in constant time whether a given node t is an ancestor of $p(x)$. Since z' knows all nodes on the path from z' to the root (and all their labels), one of which is nca_f, it just needs to find the lowest node on its path to the root which is still an ancestor of $p(x)$. Similarly, z can also compute nca_f locally.

As a second step, each node z' incident to some swap edge candidate $f = (z, z')$ sends $d(z', \mathrm{nca}_f)$ to node z across the edge f (note that nca_f depends only on the swap edge, not on the failing edge). This step requires two messages per non-tree edge, thus $O(m)$ additional messages. Then, node z knows all the terms in (4), and can evaluate the quality of swap edge $f = (z, z')$ for the given failing edge $e = (x, p(x))$ as $\max\{d(z', \mathrm{nca}_f) + d(p(x), \mathrm{nca}_f), d(z, x)\}$. Let us summarize.

Theorem 2: In a network where the failure of an edge is detected at both endpoints concurrently, all closest swap edges of a spanning tree can be computed in an asynchronous distributed setting with $O(n^* + m)$ messages of constant size, and in $O(\|\mathcal{D}\|)$ time.

B. An edge failure is detected at one endpoint only

If the failure of an edge is detected at one of its endpoints only, then the fastest way of informing all nodes as identified by Lemma 3 is to send one message from the point where the failure is detected to the endpoint of its swap edge on the corresponding side. The message then must cross the swap edge and continue towards the other endpoint of the failing edge. Thus, the time used by the message to inform all required nodes is proportional to $d(x, z) + |f| + d(z', p(x))$, irrespective of which side of the failing edge detects the failure. Note that an algorithm for computing the distances $d(x, z)$ and $d(z', p(x)) = d(z', \mathrm{nca}_f) + d(p(x), \mathrm{nca}_f)$ has already been described in Section VII-A. A slight modification of this algorithm thus computes all best swap edges in this scenario, using $O(n^* + m)$ messages and $O(||\mathcal{D}||)$ time.

Theorem 3: In a network where the failure of an edge is only detected at one of its endpoints, all closest swap edges of a spanning tree can be computed in an asynchronous distributed setting with $O(n^* + m)$ messages of constant size, and in $O(\|\mathcal{D}\|)$ time.

VIII. DISCUSSION

We have presented a distributed algorithm for computing all best swap edges for a minimum diameter spanning tree. Our solution is asynchronous, requires unique identifiers from a linearly ordered universe (but only for tiebreaking to determine a center node), and uses $O(\|\mathcal{D}\|)$ time and $O(n^* + m)$ small messages, or $O(n)$ messages of size $O(n)$. It remains an open problem to extend our approach to subgraphs with other objectives; for instance, can we efficiently compute swap edges for failing edges in a spanner?

IX. Acknowledgements

We would like to thank the anonymous reviewers for their comments, which helped to improve the presentation of this work.

References

[1] B. Gfeller, N. Santoro, and P. Widmayer, "A distributed algorithm for finding all best swap edges of a minimum diameter spanning tree," in *Proceedings of 21st International Symposium on Discributed Computing (DISC)*, ser. Lecture Notes in Computer Science, vol. 4731, 2007, pp. 268–282.

[2] M. Bui, F. Butelle, and C. Lavault, "A Distributed Algorithm for Constructing a Minimum Diameter Spanning Tree," *Journal of Parallel and Distributed Computing*, vol. 64, pp. 571–577, 2004.

[3] H. Ito, K. Iwama, Y. Okabe, and T. Yoshihiro, "Single Backup Table Schemes for Shortest-path Routing." *Theoretical Computer Science*, vol. 333, no. 3, pp. 347–353, 2005.

[4] E. Nardelli, G. Proietti, and P. Widmayer, "Finding All the Best Swaps of a Minimum Diameter Spanning Tree Under Transient Edge Failures," *Journal of Graph Algorithms and Applications*, vol. 5, no. 5, pp. 39–57, 2001.

[5] ——, "Swapping a Failing Edge of a Single Source Shortest Paths Tree Is Good and Fast." *Algorithmica*, vol. 35, no. 1, pp. 56–74, 2003.

[6] A. D. Salvo and G. Proietti, "Swapping a Failing Edge of a Shortest Paths Tree by Minimizing the Average Stretch Factor," *Theoretical Computer Science*, vol. 383, no. 1, pp. 23–33, 2007.

[7] P. Flocchini, A. M. Enriques, L. Pagli, G. Prencipe, and N. Santoro, "Point-of-failure Shortest-path Rerouting: Computing the Optimal Swap Edges Distributively," *IEICE Transactions on Information and Systems*, vol. E89-D, no. 2, pp. 700–708, 2006.

[8] P. Flocchini, L. Pagli, G. Prencipe, N. Santoro, and P. Widmayer, "Computing All the Best Swap Edges Distributively," *J. Parallel Distrib. Comput.*, vol. 68, no. 7, pp. 976–983, 2008.

[9] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1992.

[10] N. Santoro, *Design and Analysis of Distributed Algorithms*, ser. Wiley Series on Parallel and Distributed Computing, A. Y. Zomaya, Ed. Wiley, 2007.

[11] N. Santoro and R. Khatib, "Labelling and Implicit Routing in Networks," *The Computer Journal*, vol. 28, no. 1, pp. 5–8, 1985.

[12] M. Thorup and U. Zwick, "Compact routing schemes," in *SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 2001, pp. 1–10.

B. Gfeller received the M.Sc. degree from ETH Zurich, where he is currently a Ph.D. student at the institute of Theoretical Computer Science.

N. Santoro (Ph.D., Waterloo) is professor of computer science at Carleton University. Dr. Santoro has been involved in distributed computing from the beginning of the field. He has contributed extensively on the algorithmic aspects, authoring many seminal papers. He is a founder of the main theoretical conferences in the field (PODC, DISC, SIROCCO), and he is the author of the book *Design and Analysis of Distributed Algorithms* (Wiley, 2007). His current research is on distributed algorithms for mobile agents, autonomous mobile robots, and mobile sensors.

P. Widmayer graduated from Karlsruhe University, spent a postdoc year at IBM T.J. Watson Research Center, received his habilitation in Karlsruhe, and held a position at Freiburg University before joining ETH Zurich, where he teaches in the Computer Science Department.