Distributed Computation for Swapping a Failing Edge*

Linda Pagli[1], Giuseppe Prencipe[1], and Tranos Zuva[2]

[1] Dipartimento di Informatica, Università di Pisa, Italy
{pagli, prencipe}@di.unipi.it
[2] Department of Computer Science, University of Botswana, Gaborone
Zuvat@mopipi.ub.bw

Abstract. We consider the problem of computing the *best swap edges* of a shortest-path tree T_r rooted in r. That is, given a single link failure: if the path is not affected by the failed link, then the message will be delivered through that path; otherwise, we want to guarantee that, when the message reaches the edge (u, v) where the failure has occurred, the message will then be re-routed using the computed swap edge. There exist highly efficient serial solutions for the problem, but unfortunately because of the structures they use, there is no known (nor foreseeable) efficient distributed implementation for them. A distributed protocol exists only for finding swap edges, not necessarily optimal ones.

In [6], distributed solutions to compute the swap edge that minimizes the distance from u to r have been presented. In contrast, in this paper we focus on selecting, efficiently and distributively, the best swap edge according to an objective function suggested in [13]: we choose the swap edge that minimizes the distance from u to v.

Keywords: Fault-Tolerant Routing, Point of Failure Rerouting, Shortest Path Spanning Tree, Weighted Graphs, Distributed Algorithms, Data Complexity.

1 Introduction

Fault tolerance is a very important feature for distributed systems. When faults occur, programs may produce incorrect results or may stop before they have completed the intended computation. In many distributed systems the routing of messages is performed through a *shortest path* strategy. For this purpose, the shortest path trees (SPT's for short) starting from each node of the network, are computed in a preprocessing phase and stored in the so called *routing tables*. These tables specify, for each node in the network and for all possible destinations, the next hop that a message has to follow to reach its destination along the shortest path route; they contain also additional information such as the length of the path. The routing tables as a whole contain, in a distributed manner,

* This work has been supported in part by the University of Pisa and by "Progetto ALINWEB: Algoritmica per Internet e per il Web", MIUR Programmi di Ricerca Scientifica di Rilevante Interesse Nazionale.

A. Sen et al. (Eds.): IWDC 2004, LNCS 3326, pp. 28–39, 2004.

the shortest path trees rooted at each node; they can be computed by several distributed known algorithms with different degree of complexity and cost (e.g., see [3, 4, 7, 9]), starting from the distributed representation of the network, where the only knowledge of a node consists in its neighbors and their distance from it.

In these systems, a single link failure is enough to interrupt the message transmission by disconnecting one or more SPT's. Assuming that there should be at least two different routes between two nodes in the network[1] – otherwise nothing can be done – several approaches are known to recover from such situation.

One approach consists in recomputing the new shortest path trees from scratch and rebuilding the routing tables accordingly; clearly, this approach is rather expensive and causes long delays in the messages transmission [10, 15]. Another approach uses dynamic graph algorithms (e.g., those of [5]) but the difficulties arising in finding an efficient distributed approach have not yet been successfully overcome (e.g., see [14]).

A different strategy is suggested in [11]: k independent (possibly) edge-disjoint spanning trees are computed for each destination; hence at each entry of the routing table k additional links, specifying the next hop in each spanning tree, are inserted. To compute edge-disjoint spanning trees there exist also distributed algorithms (e.g. [12]). However, even if this strategy is k-fault tolerant, it is quite expensive in term of space requirements; in addition it is not shortest path.

The last approach, which will be also ours, starts from the observation that sooner or later each link will fail and from the idea of selecting for each link failure, a single non-tree link (the *swap edge*) able to reconnect the network [8, 13]. This approach does not compute the new shortest path tree, but the selection of the swap edge is done according to some optimization criteria and allows, with a single global computation, to know in advance how to recover from any possible failure. In addition, in [16] experimental results show that the tree obtained from the swap edge is very close to the new SPT computed from scratch.

Consider in particular a message with destination the root r of the SPT, arriving to node u where the link to the next hop v (as specified by the routing table) has just failed. The SPT is now divided into two disconnected subtrees, one of root r, say T'_r, and one of root u, T'_u. The swap edge can be selected, among the possible ones reconnecting the tree, to minimize different functions. For instance, it can be the one minimizing the distance from u to r, or the distance from u to v, (called *one-to-one* problems in [13]) or the total or the average distances from each node in T'_u to r (called *one-to-many* problems) in the tree obtained by substituting the failed edge (u, v) with the chosen swap edge. In [13] efficient sequential algorithms solving different one-to-one and one-to-many problems are given. In [8] the complexity of the selection of the swap edge minimizing the average distance, called there *average stretch factor*, is improved. In [6] it has been shown how the computation of swap edges can be efficiently performed in a distributed way. The routing table stored at a node is then designed to contain the new information needed to bypass any failed link.

[1] That is, the underlying graph must be 2−connected.

However, the only problem considered in [6] is the one-to-one swap problem which minimizes the distance from u to r, called there *point-of-failure shortest-path rerouting* strategy. This corresponds to the situation in which, after a failure, the message in u must be delivered as soon as possible to the root of the SPT (or,viceversa, from r to u).

Different situations can suggest a different selection of the swap edge. In particular, as suggested in [13], in this paper we consider the following problem (Best Near Swap, shortly BNS):

> Select any swap edge such that the distance from u to v is minimized, with v the parent of u in the original SPT.

This problem is of type one-to-one: it can be useful when the node u, once the failure has been detected, needs to deliver the message as soon as possible to its parent v in the original SPT [13]. In this paper, given an SPT T_r, we propose an efficient distributed algorithm which determines, for the failure of each possible edge e, a swap edge *optimal* for the BNS problem.

The solution we propose uses as starting point the structure of one of the algorithms presented in [6]. In addition, as will be observed later, the time needed to compute all the swap edges of an SPT, is less than the time required by the distributed computation of the SPT itself.

The paper is organized as follows. In the next section we give some definitions, terminology and we recall the computing paradigm. In Section 3, we propose and analyze the specific solutions for the considered problem. The concluding remarks are in Section 4.

2 Basic Definitions and Previous Results

Let $G = (V, E)$ be an undirected graph, of $n = |V|$ vertices and $m = |E|$ edges. A *label* $l(x)$ of length $|l(x)| \leq \log n$ is associated to each vertex of G. A *subgraph* $G' = (V', E')$ of G is any graph where $V' \subseteq V$ and $E' \subseteq E$. If $V' \equiv V$, G' is a *spanning* subgraph. A *path* $P = (V_p, E_p)$ is a subgraph of G, such that $V_p = \{v_1, \ldots, v_s\} | v_i \neq v_j$, for $i \neq j$, and $(v_i, v_{i+1}) \in E_p$, for $1 \leq i \leq s-1$. If $v_1 = v_s$ then P is a *cycle*. A graph G is *connected* if, for each pair $\{v_i, v_j\}$ of its vertices, there exists a path connecting them. A graph G is *biconnected* if, after the removal of anyone of its edges it remains connected. A *tree* is a connected graph with no cycles. A non negative real value called *weight* (or *length*) and denoted by $w(e)$ is associated to each edge e in G. Given a path P, the length of the path is the sum of the lengths of its edges. The *distance* $d_G(x, y)$ between two vertices x and y in a connected graph G, is the length of the shortest path from x to y in G – computed according to the weights of the edges in the path. In the following we will denote $d_G(x, y)$ by $d(x, y)$, when it is clear from the context to which graph is referred.

For a given vertex r, called *root*, the *shortest path tree* (SPT) of r is the spanning tree T_r rooted at r such that the path in T_r from r to any node v is the shortest possible one; i.e., $\forall v \in V$, $d_{T_r}(v, r) = d(v, r)$ is minimum.

After the removal of an edge $e = (u, v)$, T_r will be disconnected in two components T'_r and T'_u, rooted in r and u, respectively. Since G is biconnected,

there will always be at least an edge $e' \in E(G) \setminus E(T_r)$ that will join the two disconnected components. An edge is called a *feasible swap edge* (or simply a *swap edge*) for a node x if after the failure of edge $e = (u, v) \in T_r$, it can be utilized to reconnect T'_u to the other disconnected component rooted at r, thus forming a new spanning tree T' of G. It is easy to see that an edge $(x, y) \in E \setminus E(T_r)$ is feasible for u *if and only if* only one of x and y is a descendant of u. In the following, we will denote by $S(x)$ the set of swap edges for x, and by $InS(x) \subseteq S(x)$ the set of edges incident in x that are also swap edges for x.

We will assume that a node x knows the weight of all its incident links, and can distinguish those that are part of the spanning tree T_r from those that are not; moreover, among the links that are part of T_r, x can distinguish the one that leads to its parent $p(x)$ from those leading to its children. We assume also that each node knows its distance from the root and the distance from the root of its adjacent nodes in G.

The considered system is *distributed* with communication topology G. Each process is located at a node of G, knows the weight of its incident edges, and can communicate with its neighboring processes. A node, an edge, a label, a weight, and a distance are all unit of data. The system is *asynchronous*, and all the processes execute the same protocol. In the following, the term node or vertex will be also used to indicate a process and the term edge or link to indicate a communication line.

Algorithm 1 Overall Structure of the Algorithm

[Preprocessing]

1. DFS for labelling the tree according to [2].

[Broadcast.]

1. Each child x of the root starts the broadcast by sending to its children a list containing its name and its distance from the root.
2. Each node y, append x to the received list and sends it to its children.

[Convergecast.]

1. Each leaf z first computes its best swap. It then computes the best feasible swap edge for each of its ancestors, and sends the list of those edges to its parent (if different from r).
2. An internal node y waits until it receives the list of best swap edges from each of its children. Based on the received information and on $InS(y)$, it computes its best swap edge. It also computes the best feasible swap edge for each of its ancestors, and sends the list of those edges to its parent (if different from r).

As already mentioned, we will utilize as general paradigm, one of the algorithms of [6] for the *point-of-failure shortest path problem*. The algorithm consists of two phases: *broadcast* and *convergecast*. With the first phase (up-down in the SPT T_r) each node x receives the list of its ancestors along with their distances

from the root. The phase is started by the children of the root because no swap edge is computed for the root r. In the second phase (bottom-up) each node computes the best swap edge for itself and the best swap edge, among the edges examined so far, for each of its ancestors. The general structure of this algorithm is shown in Algorithm 1.

From [6], it derives that the message complexity of the above algorithm is $O(n)$, if long (that is $O(n)$ unit of data) messages are allowed. Otherwise, it becomes $O(n_r^*)$, where n_r^* is the size of the transitive closure of $T_r \setminus \{r\}$ and $0 \leq n_r^* \leq (n-1)(n-2)/2$. Clearly, swap edges must be selected in order to be optimal with respect to the BNS problem; this will be the focus of next section.

3 The Algorithm

In the problem we consider, the *optimal* swap edge e' for $e = (u, v)$ is any swap edge for e such that the distance from u to node v in the new tree $T' = T \setminus \{e\} \cup \{e'\}$ is minimized. More precisely, the optimal swap edge for $e = (u, v)$ is a swap edge $e' = (u', v')$ such that $d_{T'}(u, v) = d_{T_r}(u, u') + w(u', v') + d_{T_r}(v', v)$ is minimum.

As an example, consider the biconnected weighted graph G shown in Figure (1), with the minimum SPT (marked by a thick line) rooted in A. Consider vertex D after the failure of edge (D, B): the best swap edge for the BNS problem is (F, B).

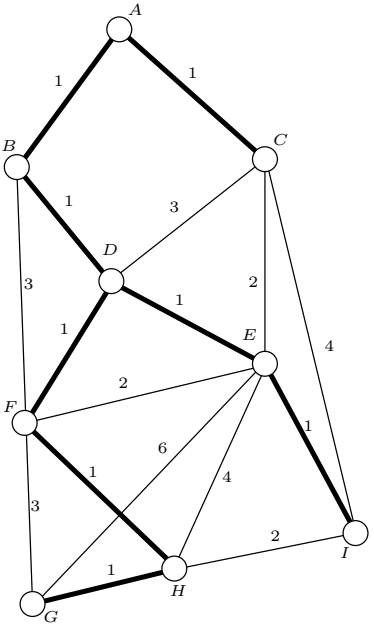

Fig. 1. An example: the thick line represents the starting SPT, rooted in A

Solving the BNS problem for a given T_r, means determining an optimal swap edge for each edge in T_r. We will design a distributed solution for the above problem, for which sequential solutions have been already studied [8, 13]. Starting from the overall structure described in Algorithm 1, we need to specify how the convergecast part is done. In particular, we will detail

(i) the computation of the best swap edge in the convergecast phase, and
(ii) the additional information, of constant size, to be communicated to the ancestors together to the swap edge.

All techniques described here are new and totally different from those adopted to design sequential solutions; furthermore, to our knowledge, this is the first distributed solution for the BNS problem.

Let us denote by $u_j, 1 \leq j \leq h$ the children of node u, and by $ANC(u)$ the set of ancestors of u in the original spanning tree T_r. Moreover, let $SL(u)$ be the list containing the set of pairs *(edge, distance)* of the feasible swap edges for u and their distance values, and $ASL(u)$ be the list of triples *(edge, distance, node)* indicating for each node $a_k \in ANC(u)$ the best feasible swap edge for a_k and the distance between a_k and $p(a_k)$ via the specified swap edge.

The details of the operations executed by node u are reported in Algorithm 2 (to compute its best swap edge) and Algorithm 3 (to compute its ancestors' swap edges). In particular, node u computes its best swap edge by considering all its feasible swap edges; that is, $InS(u)$ and the swap edges transmitted to it from its children. Then it computes, among the ones in T'_u, the best feasible swap edge for each one of its ancestors. Note that, the swap edges it computes for its ancestors can be worse than the final swap edges computed by its ancestors when *they* execute Algorithm 2.

Algorithm 2 Compute My Best Swap Edge

The protocol is described with respect to node u, with (u, v) the edge that fails.

1. Determine which of u's incident edges are feasible for u; i.e., u constructs the set $InS(u)$.
2. For each swap edge $s_i = (u, y_i) \in InS(u)$, compute the value of the distance $d_{i_{T'}}$ between u and v in T' via s_i, and insert it in $SL(u)$ the pair $(s_i, d_{i_{T'}})$.
3. If u is not a leaf, from each $ASL(u_j)$ received from child u_j, extract (s_i, d_i, u) (or NIL), and insert (s_i, d_i) in $SL(u)$ (or NIL, if no such pair exists).
4. Sort $SL(u)$ in non decreasing order of d_i. The minimal element of $SL(u)$ gives one of the best swap edges for u and the value of the minimal distance.

In the next section, we will introduce some properties that will be needed in order to show how node u can locally efficiently compute the operations in Algorithms 2 and 3.

3.1 Basic Properties

The first thing a node has to be able to do locally is to check the *feasibility* of an edge, i.e. if an edge can be considered a swap edge: this operation can be

Algorithm 3 Compute My Ancestors' Best Swap Edge

The protocol is described with respect to node u.
For each ancestor node $a_k \in ANC(u)$:

1. Consider the swap edge $s_i \in SL(u)$ feasible for a_k, with the minimal value of d_i, if any. If such an edge exists, compute the new value of the distance d_i for a_k, otherwise set it to NIL.
2. For all $1 \leq j \leq h$, let $\{(s_j, d_j, a_k)\}$ be the set of triples from $ASL(u_j)$, and consider the set $\{(s_j, d_j, a_k) \cup (s_i, d_i, a_k)\}$, $1 \leq j \leq h$, where (s_i, d_i, a_k) is the triple computed in Step 1. Select from this set the triple $(\overline{s}, \overline{d}, a_k)$ such that the distance $\overline{d}$ between a_k and $p(a_k)$ is minimal, if any, and insert it, in $ASL(u)$ (to be sent to u's parent); if no triple can be selected, insert NIL in $ASL(u)$.

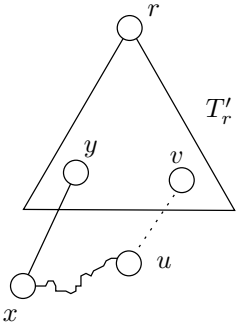

Fig. 2. Property 1

easily done during the convergecast phase, through the information collected in the broadcast phase. We state the following

Property 1 . *A swap edge $(x, y) \in E \setminus E(T_r)$ with $x \in T'_u$ and $y \in T'_r$ is* feasible *for node u if y does not belong to the path connecting x to u.*

Property 1, derives immediately from the fact that an edge (x, y) is feasible for u *if and only if* only one of its endpoints is a descendant of u (refer to Figure 2). Furthermore, we have

Property 2 . *Feasibility of swap edge $(x, y) \in E \setminus E(T_r)$ with $x \in T'_u$ and $y \in T'_r$ for node u can be checked at node x, and no communication is needed.*

Property 2, immediately derives from the fact that node x is descendant of u and neighbor of y in $\in E \setminus E(T_r)$, and that after the broadcast phase x has locally the list of all nodes between the root and itself, hence it knows which nodes are its ancestors. Therefore, the feasibility test of Step 1 in both Algorithm 2 and 3 can be locally performed. Note that, even if the global complexity does not change, this feasibility test is simpler than that used in [6], which required additional labeling of the tree nodes, with two labels to be transmitted in the two phases.

We also observe that if an edge is not feasible for x, it is not feasible for none of its ancestors.

In order to solve BNS problem, we need to compute in the convergecast phase the *nearest common ancestor* of pairs of nodes $x, y \in T_r$ (called $nca(x, y)$). Recall that the $nca(x, y)$ is the common ancestor of x and y, whose distance to x and y is smaller than the distance of any other common ancestor of x and y. In a recent work [2], it has been shown that this information can be locally computed in constant time, through a proper labeling of the tree that requires labels of $O(\log n)$ bits, denoted in the following as $l(x)$, that can be p recomputed by a depth first traversal of the tree. Therefore, Algorithm 1, needs to be slightly modified to transmit, for each node x, $l(x)$ instead of its name x. When such a labeling is computed for T_r, each node can be distinguished by its label. Therefore, we can state that

Property 3 . *Let $(x, y) \in E \setminus E(T_r)$, with $x \in T'_u$, and $y \in T'_r$ be a swap edge for u after the failure of edge (u, v). Then, $nca(y, v)$ can be computed at x, and no communication is needed.*

Property 3 follows from the results shown in [2], and noting that $l(v) \in ANC(u)$, hence $l(v) \in ANC(x)$, and that $l(y)$ is accessible at x since x is directly connected to y.

3.2 Correctness

In the BNS problem, the *optimal swap edge* for the failure of the edge $e = (u, v)$ is an edge which minimizes the distance $d'_T(u, v)$ from u to v in the new spanning tree T' obtained after the removal of the failed edge. In this section, we show that the computation of the best swap edge can be accomplished by each node, in the convergecast phase, without requesting additional information to any other node in the SPT, which is not a neighbor; that is without additional message complexity, obtaining the same complexity of Algorithm 1.

First of all we have to define how the distance $d_{T'}(u, v)$ along a given swap edge $e = (u, y) \in InS(u)$ is computed (Step 2 of Algorithm 2). The possible cases that we can have are:

- v and y lay on the same path from the root to a leaf, hence one node is ancestor of the other one (Figure 3.a, and nodes B an F in Figure 1), or
- v and y have a nearest common ancestor (Figure 3.b), such as nodes H and I in Figure 1, having D as nca.

The following lemma states how correctly compute this distance.

Lemma 1. *Let $(u, y) \in InS(u)$. We have:*

(i) $d_{T'}(u, v) = w(u, y) + |d_T(v, r) - d_T(y, r)|$, *if* $nca(y, v) = v$ *or* $nca(y, v) = y$.
(ii) $d_{T'}(u, v) = w(u, y) + d_T(v, r) - d_T(z, r) + d_T(y, r) - d_T(z, r)$, *if* $nca(y, v) = z$.

Proof. First note that $d_{T'}(u, v) = w(u, y) + d_T(y, v)$. We distinguish the two possible cases.

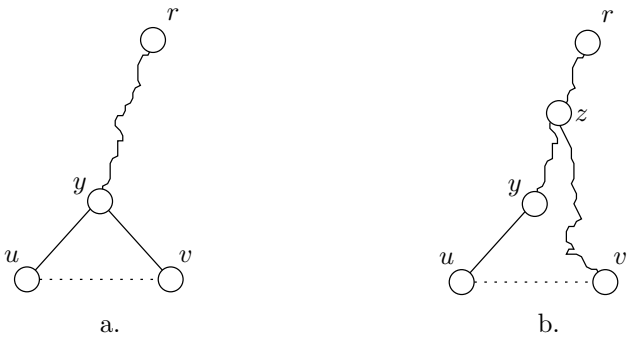

Fig. 3. (a) Case (i) of Lemma 1.(b) Case (ii) of Lemma 1

(i) y and v lay on the same path from the root to a leaf; hence, $d_{T'}(y, v)$ can be computed as the difference of their distances from the root.
(ii) In this case y and v lay on different paths which intersect in z. Their distance is easily computed by their distances from the root and from the distance of z from the root, possibly equal to 0.

Note that, each node knows its distance from r and the distance from r of each of its neighbors in G (this information can be obtained during the broadcast phase); hence, $d_{T'}(u, v)$ in the above lemma can be locally computed at u.

In the convergecast phase of Algorithm 1, a node (either a leaf or an internal node) has to be able to locally compute the best feasible swap for an ancestor. The following lemma states how distance in Step 1 of Algorithm 3 can be computed (refer to Figure 4.a and 4.b).

Lemma 2. *Consider a subset $a_1, \ldots, a_l$ of the ancestors of u, with $p(u) = a_1$, and a_k adjacent to a_{k+1}, $1 \leq k < l$. Furthermore, let (x, y) be a feasible swap edge for (u, a_1) and for (a_k, a_{k+1}), $1 \leq k < l$. Consider the two trees $T' = ((T_r \setminus (u, a_1) \cup (x, y))$ and $T'' = ((T_r \setminus (a_k, a_{k+1}) \cup (x, y))$; then, $d_{T''}(a_k, a_{k+1}) = d_{T'}(u, a_1) + w(u, a_1) - w(a_k, a_{k+1})$.*

Proof. Since (x, y) is feasible for a_k, by Property 1 it follows that y is not descendant of a_k, and that $nca(y, a_1) = nca(y, a_k)$. Thus, in T', the path p' from u to a_1 along the swap edge (x, y) includes $a_k, ..., a_2$ (see Figure 4.a). Moreover, the path p'' from a_k to a_{k+1} in T'' results to be the same as p', except for edge (u, a_1), that is substituted with (a_k, a_{k+1}) (Figure 4.b), and the lemma follows.

Also in this case, similarly to Lemma 1, the computation of $d_{T''}$ can be performed locally at u, because of the information retrieved during the broadcast phase of Algorithm 1. Finally, we can state that problem BNS is correctly solved by Algorithm 2 and 3.

Theorem 1. *Each node $u \neq r$:*

(i) *correctly computes its best swap edge;*

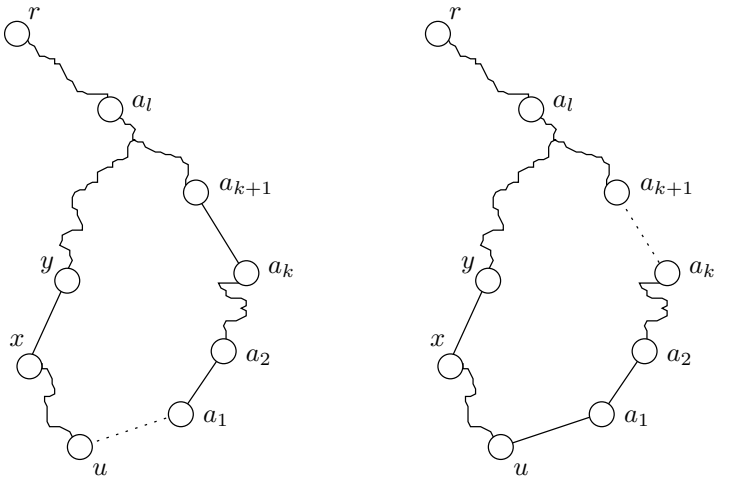

Fig. 4. Trees T and T' in Lemma 2

(ii) *correctly determines for each ancestor $a_k \neq r$ the best swap edge feasible for a_k in T'_u.*

Proof. First observe that, as result of the broadcast, every node receives the label of its ancestors (except r) and it can compute the feasibility of each available swap edge for itself and its ancestors (Property 1 and 2). The proof is by induction on the height $h(u)$ of the subtree T_u.

Basis. $h(u) = 0$; i.e., u is a leaf. In this case, one component contains only u, while the other contains all the other nodes. In other words, the only possible swap edges are incident on u. Thus, u can correctly compute its best swap edge computing the value of the distance as stated in Lemma 1, proving (i). It can also immediately determine the feasibility of any of those edges with respect to all its ancestors and, in case they are feasible, compute for them the value of the distance as stated in Lemma 2 and select, for each ancestor, the best feasible one.

Induction Step. Let the theorem hold for all nodes x with $k - 1 \geq h(x) \geq 0$; we will now show that it holds for u with $h(u) = k$. By inductive hypothesis, it receives from each child y the best feasible swap edge for each ancestor of y, including u itself. Hence, based on these lists and on the locally available set $InS(u)$, u can correctly determine its optimal swap edge, as well as its best feasible swap edge for each of its ancestors.

We can pose the following:

Theorem 2. *Problem* BNS *can be solved with the $O(n)$ message complexity and $O(n_r^*)$ data complexity.*

Proof. The theorem follows immediately from Properties 2 and 3, the computation of the *nca*, from Lemma 2, and from the fact that all the needed computation to determine the best swap edge of a node and of its ancestors can be done locally, thus not changing the complexity of Algorithm 1.

Example. Consider in the example of Figure 1, the computation of node D. Assume that nodes F and E have already correctly computed the lists $ASL(F) = \{((F,B),4,D),NIL\}$ and $ASL(E) = \{((E,C),5,D),((E,C),5,B)\}$ and sent them to their parent D. D has only one feasible swap edge (D,C), for which $d_{T'}(D,B) = 5$, then it receives $(F,B),4$ from F and $(E,C),5$ from E. Therefore (F,B) is the best swap edge for D. Now D selects the best feasible swap for its parent B, by considering the best one among its swap edges feasible for B, that is (D,C) and considering the edges coming from its children, in this case only (E,C) from E. The value $d_{T'}(B,A)$ via (E,C) is already known, while D has to compute $d_{T'}(B,A)$ via (D,C), which is equal to 5. $\{((D,C),5,B)\}$ is then transmitted to B.

4 Concluding Remarks

In this paper we have presented an efficient distributed algorithm to solve the BNS problem: given a shortest path tree T_r rooted in r, we want to select the swap edge e' for any edge (u,v) in T_r so that the distance between u and v is minimized in $T_r \setminus \{(u,v)\} \cup \{e'\}$. To make the routing table 1-fault tolerant, the computation of the swap edges must be repeated for all possible shortest path trees; that is, for all nodes of the graph.

We note that the proposed algorithm allows for the efficient construction of a rerouting service. To do so, the proposed computation must be carried out for the n shortest path trees, each having as root a different vertex of the graph G. In this regards, an interesting open problem is whether it is possible to achieve the same goal in a more efficient way than by performing n independent computations.

An immediate possible development of this study, would be to study how to recover from multiple link failures, following the same strategy of storing in the routing tables the information useful for finding alternative paths. Other possible studies involve the analysis of the *one-to-many* problems presented in Section 1, such as choosing the swap edge so to minimize the sum of the distances from each node in T'_u to r.

References

1. Y. Afek, M. Ricklin. Sparser: a paradigm for running distributed algorithms. *Journal of Algorithms*, 14:316-328, 1993.
2. S. Alstrup, C. Gavoille, H. Kaplan, T. Rauhe. Nearest Common Ancestor: A survey and a new distributed algorithms. *14th Annual ACM Symposium on Parallel Algorithms and Architecture (SPAA)*, 258–264, 2002.

3. B. Awerbuch, R. Gallager. A new distributed algorithm to find breadth first search trees. *IEEE Transactions on Information Theory, 33 (315–322) 1987.*
4. K. M. Chandy, J. Misra. Distributed computation on graphs: shortest path algorithms. *Communication of ACM, 25 (833–837) 1982.*
5. D. Eppstein, Z. Galil, G.F. Italiano. Dynamic graph algorithms. *CRC Handbook of Algorithms and Theory, CRC Press, 1997.*
6. P. Flocchini, A. Mesa Enriquez, L. Pagli, G. Prencipe, N. Santoro. Efficient protocols for computing the optimal swap edges of a shortest path tree. *Proc. of 3-th IFIP International Conference on Theoretical Computer Science* (TCS@2004), 153–166, 2004.
7. G. N. Frederikson. A distributed shortest path algorithm for planar networks. *Computer and Operations Research, 17 (153–151) 1990.*
8. A. Di Salvo, G. Proietti. Swapping a failing edge of a shortest paths tree by minimizing the average stretch factor. *Proc. of 10th Colloquium on Structural Information and Communication Complexity* (SIROCCO 2004), 99–104, 2004.
9. P. Humblet. Another adaptive distributed shortest path algorithm. *IEEE/ACM Transactions on Communications*, 39(6):995–1003, 1991.
10. H. Ito, K. Iwama, Y. Okabe, T. Yoshihiro. Polynomial-time computable backup tables for shortest-path routing. *Proc. of 10th Colloquium on Structural Information and Communication Complexity* (SIROCCO 2003), 163–177, 2003.
11. A. Itai, M. Rodeh. The multi-tree approach to reliability in distributed networks. *Information and Computation*, 79:43-59, 1988.
12. H. Mohanti, G. P. Batthacharjee. A distributed algorithm for edge-disjoint path problem. *Proc. of 6th Conference on Foundation of Software Technology and theoretical Computer Science*, (FSTTCS), 344-361, 1986.
13. E. Nardelli, G. Proietti, P. Widmayer. Swapping a failing edge of a single source shortest paths tree is good and fast. *Algoritmica*, 35:56–74, 2003.
14. P. Narvaez, K.Y. Siu, H.Y. Teng. New dynamic algorithms for shortest path tree computation. *IEEE Transactions on Networking*, 8:735–746, 2000.
15. L. L. Peterson, B. S. Davie. *Computer Networks: A Systems Approach, 3rd Edition.* Morgan Kaufmann, 2003.
16. G. Proietti. Dynamic maintenance versus swapping: An experimental study on shortest paths trees. *Proc. 3-rd Workshop on Algorithm Engineering (WAE 2000).* Lecture Notes in Computer Science, Springer, (1982) 207–217, 2000.