Distributed Calculation of Edge-Disjoint Spanning Trees for Robustifying Distributed Algorithms against Man-in-the-Middle Attacks

Gabriele Oliva†,§, ×, Sebastian Cioaba‡ and Christoforos N. Hadjicostis‡

Abstract—In this paper we provide a distributed methodology to allow a network of agents, tasked to execute a distributed algorithm, to overcome Man-in-the-middle attacks that aim at steering the result of the algorithm towards inconsistent values or dangerous configurations. We want the agents to be able to restore the correct result of the algorithm in spite of the attacks. To this end, we provide a distributed algorithm to let the set of agents, interconnected by an undirected network topology, construct several edge-disjoint spanning trees by assigning a label to their incident edges. The ultimate objective is to use these spanning trees to run multiple instances of the same distributed algorithm in parallel, in order to be able to detect Man-in-the-middle attacks or other faulty or malicious link behavior (e.g., when the instances yield different results) and to restore the correct result (when the majority of instances is unaffected). The proposed algorithm is lightweight and asynchronous, and is based on iterated depth-first visits on the graph. We complement the paper with a thorough analysis of the performance of the proposed algorithms.

Keywords: Distributed Algorithms, Man-in-the-middle attacks, Tree Packing Problem, Edge Disjoint Spanning Trees.

I. INTRODUCTION

Distributed algorithms, such as consensus algorithms [1], [2], have been proven quite effective in allowing a set of interconnected agents to perform complex global computations, optimizations, or other desirable operations, by means of iterative, simple and local interactions among neighboring agents. These algorithms, however, are often prone to failures and malicious data manipulations, and there is a need to provide reliable methodologies to overcome such issues.

Several solutions have been provided in the literature. For example, in [3] a distributed Byzantine consensus algorithm that tolerates message omissions by a subset of the agents is provided; in [4] a related methodology for directed graphs is developed; in [5] a framework that places emphasis on the trust among the nodes is given. More recently, an adaptive finite-time approach has been provided in [6], while a methodology that can overcome actuator faults is presented in [7]; moreover, in [8] an approach that can handle a variable network structure is proposed.

The above approaches focus on the agent’s behavior, while malicious attacks affecting the links are typically neglected.

Among other issues, Man-in-the-middle (MITM) attacks [9]–[11] represent an effective way to alter the data transmitted by the agents, potentially steering the network towards inconsistent results or dangerous configurations.

As shown in Figure 1, MITM attacks consist of an attacker (Mallory) becoming the proxy for the communication between two victim nodes (Alice and Bob). Specifically, Mallory intercepts all messages between Alice and Bob and it sends maliciously altered messages to Alice pretending to be Bob, and vice-versa. In this way, Alice and Bob believe they are directly communicating with each other, and are unable to detect the attack.

A. Related Work: Fault-Tolerant Distributed Algorithms

In the literature there have been attempts to develop distributed algorithms and protocols having some resistance to MITM attacks. In [12] Chiang et al. develop a methodology for distributed time synchronization that has some robustness to MITM attacks, in that only limited damage can be dealt by malicious attackers, which can cause only constant delays. In [13] a secure time synchronization protocol was developed, based on pairing and identity-based cryptography. In [14] RSSI information is used to spot spoofed messages. Such approaches, however, require the implementation of sophisticated countermeasures, which are often tailored to the specific application (e.g., time synchronization).

Note that fault-tolerant message delivery protocols are often implemented by constructing multiple paths on the network [15]–[17], by requiring the existence of such disjoint paths [18], or by building independent subnetworks [19], [20].

In this paper we follow such a perspective while facing MITM attacks; our focus, therefore, is not on preventing the occurrence of MITM attacks, but on guaranteeing that the distributed algorithm executed by the agents achieves its purpose, in spite of successful MITM attacks. To achieve this result, we partition the network into several Edge-Disjoint Spanning
Subgraphs (EDSSGs), i.e., connected subgraphs spanning all the nodes and not having common edges. Once these EDSSGs are defined, the agents run independent instances of the same distributed algorithm over each EDSSG. By comparing the results obtained by the different instances of the algorithm, which are supposed to yield the same result under nominal conditions, the agents are able to detect the attack (e.g., when the results obtained are not all the same) and to restore the correct value (when the majority of instances are unaffected).

In order to achieve this result in a distributed way, we develop a methodology to let the agents in the network partition their incident links in order to construct several Edge-Disjoint Spanning Trees (EDSTs) over the network, with the aim to use these EDSTs as the backbone for the EDSSGs.

B. Related Work: Edge-Disjoint Spanning Trees

The problem of finding the maximum number of EDSTs over a graph, often referred to as the Tree Packing Problem, has been faced since the beginning of the 1960’s when mathematicians like Tutte [21] and Nash-Williams [22] provided the same combinatorial conditions to calculate the maximum number of EDSTs. Other theoretical results include a shorter proof of the Tutte-Nash-Williams condition [23], and a characterization of the number of spanning trees based on the eigenvalues of the adjacency or Laplacian matrices of regular graphs [24], further extended to general graphs in [25]. In [26] the authors characterize the expected minimum total weight of a number of EDSTs over complete weighted graphs.

From an algorithmic point of view, there are several approaches for obtaining EDSTs in the literature, and they are intrinsically centralized. The centralized algorithm in [15] calculates two EDSTs; the algorithm presented in [27] is based on the work of Edmonds on matroid theory [28] and is able to construct the maximum number of EDSTs in a graph in polynomial time. Available distributed approaches are limited to extremely particular structures, such as hypercubes [29] or twisted cubes [30].

C. Contribution and Paper Outline

In this paper we provide a lightweight and asynchronous, yet suboptimal, algorithm that iteratively lets the agents label their incident links as part of different EDSTs. The key idea behind the algorithm is that, in order to increase the chances to have several edge-disjoint spanning trees, a good strategy is to resort to iterated depth-first visits of the graph, each time removing edges that have been labeled in previous iterations.

The procedure is iterated as long as it is possible, i.e., until the remaining links form a disconnected graph, as shown in the left plot of Figure 2, where 2 EDSTs are found. Other approaches, such as a breadth-first visit (see the right plot in Figure 2) are not suitable for the problem at hand, as in the breadth-first visit all links that are incident to the starting node are used for constructing the first EDST, thus preventing the construction of additional EDSTs.

The proposed approach, apart from constructing several EDSTs, also allows the nodes to calculate useful meta-information such as the diameter of each EDST and the number of nodes in the network.

The outline of the paper is as follows: Section II provides some preliminary definitions; Section III introduces the problem at hand and discusses the motivations underlying the proposed approach; Section IV collects some classical theoretical and algorithmic results related to finding the maximum number of EDSTs in a graph; Sections V and VI introduce the proposed algorithms for finding a single spanning tree, and several EDSTs, respectively; Section VII provides a simulation campaign aimed at assessing the performances of the proposed approach, while some conclusions and future work directions are collected in Section VIII.
is connected and undirected. Invariant with respect to the underlying graph \( G \)
steady-state result (asymptotically or in finite time) for each algorithm
exchange information in order to implement a distributed
disjoint spanning trees
As a consequence of the attack, algorithm \( A \) hijack the communication between one or more pairs of
edge-disjoint trees (EDST). Similarly, we refer to two connected and
graph composed of one or more connected components, each
\( V \), \( E \)
\( i \)
\( d \)
The Laplacian matrix is \( L = D - A \), while the signless Laplacian matrix is \( L_s = D + A \). Given a graph \( G \), we denote
by \( \lambda_i(G) \), \( \mu_i(G) \) and \( \theta_i(G) \) the \( i \)-th smallest eigenvalue of \( A \), \( L \) and \( L_s \), respectively\(^1\).
A tree \( T \) is a connected acyclic graph. We say \( T \) is rooted
at a node \( v_i \) if \( v_i \) has been designated the root of \( T \). With
the root node \( v_i \), a leaf is a node \( v_j \neq v_i \) such that
\( d_j = 1 \), while the depth of a node \( v_j \) is the length of the path
over \( T \) from the root node \( v_i \) to \( v_j \). Given a node \( v_j \) in a tree
the root \( v_i \), the branch of the tree for node \( v_j \) is the set
of nodes that are in a path \( P \), from \( v_j \) to a leaf node, such
that \( v_i \notin P \). A spanning tree of a graph \( G = \{V, E\} \) is a
connected acyclic subgraph \( T = \{V, E_T\} \), where \( E_T \subseteq E \). A
graph composed of one of more connected components, each
being a tree, is referred to as a forest.
Two subgraphs \( G_1 = \{V, E_1\} \) and \( G_2 = \{V, E_2\} \) of a graph
\( G = \{V, E\} \) are edge-disjoint if \( E_1 \cap E_2 = \emptyset \). In particular,
if \( G_1 \) and \( G_2 \) are connected they are edge-disjoint spanning
subgraphs (EDSSG). Similarly, we refer to two connected and
edge-disjoint trees \( T_1 = \{V, E_{T_1}\} \) and \( T_2 = \{V, E_{T_2}\} \) as edge-
disjoint spanning trees (EDST).

III. Problem Statement and Motivation
Consider a set of \( n \) agents interconnected by an undirected
and connected graph \( G = \{V, E\} \) and suppose the agents need
to exchange information in order to implement a distributed
algorithm \( A \) that satisfies the following assumption.

Assumption 1: The distributed algorithm \( A \) achieves a steady-state result (asymptotically or in finite time) for each
agent (not necessarily the same result for all agents) which is
invariant with respect to the underlying graph \( G \), as long as it
is connected and undirected.

Note that algorithm \( A \) might be prone to data manipulations
such as man-in-the-middle attacks, where malicious attackers
hitjack the communication between one or more pairs of
nodes, modifying the information sent, without being noticed.
As a consequence of the attack, algorithm \( A \) might yield
completely different results at each agent, preventing them
from performing their intended functions.

Our objective is therefore to endow the agents with the
ability to detect MITM attacks and recover the correct result
of the algorithm in spite of the attack. To this end, we note
at least one final value is different
\( E \)
\( m \) links in \( E \) are attacked.

Remark 1: As long as at least one final value is different
from the others, within the proposed strategy the agents are
able to detect a man-in-the-middle attack, even if they might
not be able to retrieve the correct result for Algorithm \( A \). In
other words, the agents are guaranteed to be able to spot an
attack if at most \( m - 1 \) links in \( E \) are attacked.

Remark 2: Within the proposed strategy, Algorithm \( A \) is
robust with respect to man-in-the-middle attacks leaving the
majority of the EDSSGs unaffected. In other words, the agents

\(^1\)Since these matrices are symmetric, their eigenvalues are real-valued.
are guaranteed to be able to compute the correct result if at most $\zeta$ links in $E$ are attacked, with

$$\zeta = \left\lfloor \frac{m}{2} \right\rfloor - 1.$$

**Remark 3:** Note that, when at least $\zeta + 1$ links are affected, there is the possibility that the majority of the results is faulted, preventing the agents from calculating the right values. However, attacking more than $\zeta$ links is not sufficient to have a guarantee to affect the final result. This happens because the attacker needs to attack at least $\zeta + 1$ links, each belonging to a different EDSSG. To succeed in this task, the attacker needs either prior knowledge on the network or enough resources to deal a large-scale attack.

**Remark 4:** In order to drive the agents towards incorrect final values, the altered values must all be the same. This, again, implies that, depending on the particular algorithm being executed, the attacker may need to spend nontrivial effort in order to successfully steer the algorithm towards a different result.

For the reasons collected in Remarks 1–4, in this paper we are interested in solving the following problem.

**Problem 1:** Given a connected and undirected graph $G = \{V,E\}$ find a maximum number of EDSSGs.

In this paper we are interested in solving the problem in a distributed way, i.e., we want each agent to interact with its neighbors in a distributed way in order to label their incident edges with different labels, so that links with the same label belong to the same EDSSG.

Note that finding a maximum number of EDSSGs essentially coincides with finding a maximum number of EDSTs. Moreover, solving exactly Problem 1, especially in a distributed way, might be impractical because, as discussed in the next section, available solutions in the literature are essentially centralized. In the remainder of this paper, therefore, we adopt a strategy which is suboptimal but implementable in a distributed context. Specifically, we provide an approximate solution to Problem 1 by iteratively performing depth-first visits of the graph and by removing the visited links from consideration, thus building multiple EDSTs.

**IV. Maximum Number of EDSTs**

In this section we discuss some results on the maximum number $\tau(G)$ of edge-disjoint spanning trees of a graph $G$.

**A. Combinatorial Condition**

The following theorem, proved independently by Tutte and Nash-Williams [21], [22], provides a combinatorial condition for finding $\tau(G)$.

**Theorem 1:** An undirected multi-graph $G = (V,E)$ contains $k$ edge-disjoint spanning trees iff for every partition $\Pi$ of $V$ into $l$ sets, $V_1, V_2, \ldots, V_l$, the number $q$ of edges whose endpoints are in different sets of the partition $\Pi$ is at least $q = k(l - 1)$.

**Remark 5:** The direct application of the above condition to calculate $\tau(G)$ is rather impractical, because of the need to inspect a vast number of partitions. In fact, for $n$ nodes, the total number of partitions of the nodes is given by the Bell number $B_n$, which can be obtained by taking $B_0 = B_1 = 1$ and by applying the recursive rule, for $m \geq 2$

$$B_{m+1} = \sum_{k=0}^{m} \binom{m}{k} B_k.$$

In [34] it is shown that

$$B_m \approx \frac{1}{\sqrt{m}} \lambda(m)^{m+1/2} e^{\lambda(m) - m - 1},$$

where $\lambda(m)$ is such that $\lambda(m) \log(\lambda(m)) = m$.

**B. Optimal but Centralized Algorithm**

In [27], Roskind and Tarjan provide an efficient, yet centralized, algorithm to find a maximum number of EDSTs in a graph, based on matroid theory.

Let a set $X$ of $n$ elements and define some of the subsets of $X$ as independent. The set $X$ is a matroid if the independence of any $S \subseteq X$ implies the independence of any subset of $S$. Matroids have the interesting property that the largest independent subset can be found by means of greedy algorithms [28]. In [27] the authors cast the problem at hand in terms of matroids: given a graph $G = \{V,E\}$, they choose $X = E$ and they say $E' \subseteq E$ is independent with respect to an integer $k$ if $G' = \{V,E'\}$ can be partitioned into $k$ edge-disjoint forests.

The main idea of the algorithm in [27] is, therefore, to iteratively grow a set of $k$ edge-disjoint forests $F = \{F_1, \ldots, F_k\}$, considering a link $e \in E$ at each time and checking the independence of $F \cup \{e\}$. If $F \cup \{e\}$ is independent, then the algorithm modifies the assignment of some of the links in $F$ and adds $e$ to $F$. If, conversely, $F \cup \{e\}$ is dependent, then the algorithm increases the number of forests (initially, $k = 1$).

The above algorithm, therefore, invokes $|E|$ times an oracle, that is, a function that is able to tell the independence of $F \cup \{e\}$ (and to reorganize the links in the forests to avoid cycles). The oracle is the bottleneck of the approach, and in [27] an efficient implementation is provided, which has a computational complexity $O(|E|^2)$; the implementation is based on the fast disjoint set union algorithm [35]. Such an approach, however, is centralized and appears hard to translate in a distributed context.

**C. Lower bound on $\tau(G)$**

The following Theorem provides a lower bound on $\tau(G)$ [25], which extends the result given in [24]. We will take advantage of this lower bound in Section VII while discussing the performances of the proposed distributed algorithms.
**Theorem 2 (Liu et al., 2014 [25]):** Let $G$ be a graph of minimum degree $d_{\text{min}} \geq 2\tau_b(G) \geq 4$; the following conditions hold true:

1. If $\lambda_{n-1}(G) < d_{\text{min}} - \frac{2\tau_b(G)-1}{d_{\text{min}}+1}$, then $\tau(G) \geq \tau_b(G)$.
2. If $\theta_{n-1}(G) < 2d_{\text{min}} - \frac{2\tau_b(G)-1}{d_{\text{min}}+1}$, then $\tau(G) \geq \tau_b(G)$.
3. If $\mu_2(G) > \frac{2\tau_b(G)-1}{d_{\text{min}}+1}$, then $\tau(G) \geq \tau_b(G)$.

In [24] it is shown that the above bounds are tight in the case of regular graphs with $\tau_b(G) = 2$ or $\tau_b(G) = 3$, while the result is extended to graphs with $\tau_b(G) \geq 4$ in [36].

V. DISTRIBUTED CONSTRUCTION OF EDSTS

As discussed above, the algorithm from Roskind and Tarjan is quite hard to implement in a distributed fashion. In this paper we present a distributed methodology for obtaining EDSTS based on repeated depth-first visits of the graph; in this section we develop an algorithm to construct a single spanning tree, while in the next section we discuss the construction of multiple EDSTS.

Note that, the algorithm we present in the remainder of this section exploits the typical token-passing approach that implements a depth-first visit in a graph, which is quite diffused in the literature of distributed algorithms [37]-[40]. With respect to the state of the art, the main improvement here is the ability to calculate the diameter of the tree while it is being built. As discussed in the conclusive section, this feature can be the basis to guide the construction of EDSSGs with small diameter; specifically, it might be useful to sacrifice some of the EDSTS in order to have more “spare links” to be assigned to the remaining EDSTS, with the aim to reduce their diameter and boost the convergence time of the distributed algorithms being executed.

A. Distributed Spanning Tree Construction Algorithm

In this section we develop a distributed algorithm to let a set of agents, interconnected by an undirected and connected graph, find a spanning tree by labeling some of the edges; we call this algorithm the Distributed Spanning Tree Construction Algorithm (DSTC). The pseudocode of the proposed DSTC Algorithm is reported in Algorithm 1, while an example of its execution is shown in Figure 3. The algorithm amounts to a depth-first visit of the graph, starting from a leader node $v_{1,i}$, which can be elected via several techniques (e.g., see [41] and references therein). Specifically, we assume the depth-first visit is implemented in an asynchronous way, by letting the nodes pass each other a single token (initially the leader has the token). The node holding the token passes it to a neighbor node which has not yet been visited (if any), otherwise it returns the token to its father (i.e., to the node from which it received the token in first place). The spanning tree is constructed by selecting the edges through which the token is passed among the nodes. In the process, each node receiving a token also updates some internal variables, which are used to keep track of the links in the spanning tree and to calculate useful meta-information such as the number of nodes and the diameter of the spanning tree that is being constructed. The procedure

---

**Algorithm 1 Distributed Spanning Tree Construction Algorithm (DSTC)**

```
procedure INITIALIZE\(ID,\)is-\(\text{leader}\)
\(v_i^{\text{father}} \leftarrow \emptyset;\)
\(\mathcal{M}_i \leftarrow \mathcal{N}_i;\)
\(\Delta_i \leftarrow 0;\) // biggest \(\Delta\) from \(v_i\) to a leaf
\(\Delta^\uparrow_i \leftarrow 0;\) // second biggest \(\Delta\) from \(v_i\) to a leaf
\(\delta^T_i \leftarrow 0;\) // estimate of the diameter
\(\text{label} (v_i, v_j)\) with \(-\infty\) for all \(v_j \in \mathcal{N}_i;\)
if is-\(\text{leader}\) then
\(\text{visited}_i \leftarrow 1;\)
send \(<i,\text{visited}_i>\) to all neighbors;
\(\Delta_i \leftarrow 0;\)
select random \(v_j \in \mathcal{M}_i;\)
send token \(<ID, 1, 0, 0, 0>\) to \(v_j;\)
\(\text{label} (v_i, v_j)\) with ID;
else
\(\text{visited}_i \leftarrow 1;\)
\(\Delta_i \leftarrow \infty;\) // depth of \(v_i\) in the tree
```

```
procedure ON\(\text{RECEIVE}||\text{VISITED}||<j,\text{visited}_j>\)
\(\mathcal{M}_i \leftarrow \mathcal{M}_i \setminus \{v_j\};\)
```

```
procedure ON\(\text{RECEIVE}||\text{TOKEN}||<s,\text{ID}, m, \Delta, \Delta^\uparrow, \delta^T>\)
lable \((v_s, v_i)\) with ID:
if \(\Delta_i = \infty\) then // update \(\Delta_i\) on first token received
\(\Delta_i \leftarrow \Delta_s + 1;\)
\(v_i^{\text{father}} \leftarrow v_s;\)
if \(\Delta > \Delta^\downarrow_i\) then // update \(\Delta^\downarrow_i\) and \(\Delta^\uparrow_i\)
\(\Delta^\downarrow_i \leftarrow \Delta_i;\)
\(\Delta^\uparrow_i \leftarrow \Delta^\downarrow_i;\)
else if \(\Delta > \Delta^\uparrow_i\) then
\(\Delta^\downarrow_i \leftarrow \Delta_i;\)
if \(\delta^T > \delta^T_i\) then // update \(\delta^T_i\)
\(\delta^T_i \leftarrow \delta^T;\)
\(\delta^T \leftarrow \max\{\delta^T_i, \Delta^\downarrow_i + \Delta^\uparrow_i - 2\Delta_i\};\)
/* Send token */
if \(\mathcal{M}_i \neq \emptyset\) then //send towards leaves
select random \(v_j \in \mathcal{M}_i;\)
if is-\(\text{leader}\) then
\(\text{send} <ID, m, 0, 0, 0>\) to \(v_j;\)
else if not visit\(\text{ed}_i\), then
\(\text{visited}_i \leftarrow 1;\)
\(\text{send} <i,\text{visited}_i>\) to all neighbors;
\(\text{send} <ID, m + 1, \Delta + 1, \Delta_i, 0>\) to \(v_j;\)
else
\(\text{send} <ID, m, \Delta_i, \Delta_i, 0>\) to \(v_j;\)
else if not \(v_i^{\text{father}} = \emptyset\) then // send towards root
if not visit\(\text{ed}_i\), then
\(\text{visited}_i \leftarrow 1;\)
\(\text{send} <i,\text{visited}_i>\) to all neighbors;
\(\text{send} <ID, m + 1, \Delta + 1, \Delta_i, \delta^T>\) to \(v_i^{\text{father}};\)
else
\(\text{send} <ID, m, \Delta^\uparrow_i, \Delta_i, \delta^T>\) to \(v_i^{\text{father}};\)
stop;
else // \(v_i\) is the leader and visit complete
\(\delta \leftarrow \delta^T;\)
stop;
```
where the token are modified at each passage. Specifically, the token is a tree that is being constructed, and some of the contents of the neighbors have been visited.

Fig. 3. Example of execution of the proposed DSTC algorithm. Red dotted links are labeled as belonging to a spanning tree.

is completed when the token returns to the leader and all its neighbors have been visited.

The token contains some information about the spanning tree that is being constructed, and some of the contents of the token are modified at each passage. Specifically, the token is the 5-tuple

\[ <\text{ID}, m, \Delta, \Delta_s, \delta_T> \]

where

- \text{ID} is the identifier of the spanning tree being constructed;
- \( m \) is the current value of the number of nodes visited so far;
- \( \Delta \) is the depth of the spanning tree so far (i.e., the maximum length of a path connecting a visited node to the leader);
- \( \Delta_s \) is the depth of the node \( v_s \) that sends the token;
- \( \delta_T \) is the current estimate of the diameter of the spanning tree.

In addition to passing the token, each node \( v_i \) maintains and updates the following parameters during the execution of the algorithm:

- a binary label \text{visited}, that is used to determine if \( v_i \) has been visited (we initialize \text{visited}_i = 0 \) for all nodes);
- the identifier of its father over the tree (initialized to zero);
- an integer \( \Delta_i \) that represents the depth of \( v_i \) over the spanning tree–initially \( \Delta_i = \infty \) for all nodes but the leader, which sets \( \Delta_i = 0 \);
- the set \( \mathcal{M}_i \) of not visited neighbors, which is initially equal to its neighborhood \( \mathcal{N}_i \);
- the labels associated to the links \( (v_i, v_j) \) towards its neighbors \( v_j \) (initialized to \(-\infty\));
- two integers \( \Delta^\dagger \) and \( \Delta^\ddagger \), which are the biggest and the second biggest depths of the nodes that branch from \( v_i \) along the tree (both initialized to zero for all nodes);
- an integer \( \delta^\dagger \) which is an estimate of the diameter of the tree (initialized to zero for all nodes).

During the initialization phase, the leader \( v_{i^*} \) sends the token

\[ <\text{ID}, 1, 0, 0, 0> \]

to a random node \( v_j \in \mathcal{M}_{i^*} \), and it labels \( (v_{i^*}, v_j) \) with the identifier ID of the current spanning-tree; then the leader becomes visited and it provides such an information to all its neighbors, by broadcasting the message

\[ <i^*, \text{visited}, \ast> \]

Every time a node \( v_i \) receives information about a neighbor \( v_j \) that becomes visited, \( v_j \) is removed from \( \mathcal{M}_i \).

When a node \( v_j \) receives a token for the first time, it updates its depth setting \( \Delta_i = \Delta + 1 \), while each time a node receives a token, it updates the value of the biggest depth \( \Delta^\dagger \) and second biggest depth \( \Delta^\ddagger \) of one of its neighbors that branch from \( v_i \) along the tree; then, node \( v_i \) updates \( \delta^\dagger \) and it calculates

\[ \overline{\delta}_T = \max\{\delta^\dagger_T, \Delta^\dagger + \Delta^\ddagger - 2\Delta_i\} \]

The node \( v_j \) attempts to transmit the token downwards the tree to a not visited neighbor \( v_j \in \mathcal{M}_i \). If such a neighbor exists, we have three cases:

1) if \( v_i \) is the leader, then it sends to node \( v_j \) the token

\[ <\text{ID}, m, 0, 0, 0> \]

2) if \( v_i \) is not visited, then it becomes visited, it informs its neighbors and it sends to node \( v_j \) the token

\[ <\text{ID}, m+1, \Delta + 1, \Delta_i, 0> \]

3) otherwise \( v_i \) sends to node \( v_j \) the token

\[ <\text{ID}, m, \Delta_i, \Delta_i, 0> \]

If \( \mathcal{M}_i = \emptyset \), conversely, node \( v_i \) transmits the token upwards to its parent, unless \( v_i \) is the leader. Specifically, three cases are possible:

I) if \( v_i \) is not visited nor the leader, then it becomes visited, it informs its neighbors and it sends to its parent the token

\[ <\text{ID}, m + 1, \Delta + 1, \Delta_i, \overline{\delta}_T> \]

II) if \( v_i \) is visited and it is not the leader, it sends to its parent the token

\[ <\text{ID}, m, \Delta^\ddagger, \Delta_i, \overline{\delta}_T> \]
(recall that $\Delta^i$ is the biggest depth among those of the nodes that branch from $v_i$ along the tree);

III) if $v_i$ is the leader, then it calculates the diameter of the tree as $\delta = 2\Delta_T$, the number $m$ of nodes visited, and terminates the procedure.

In all the above cases, a node $v_i$ with $M_i = \emptyset$ is no more involved in the calculation of that particular tree, while it is involved in the calculation of successive spanning trees.

B. Discussion and Properties

Let us provide some results on the correctness of the above algorithm.

**Theorem 3:** Suppose $G$ is undirected and connected. The DSTC Algorithm is correct, that is, at its completion

a) it finds a spanning tree over $G$;

b) the leader node knows the number $n$ of nodes in $G$;

c) the leader node knows the diameter $\delta$ of the spanning tree.

**Proof:** The proof of point a) is trivial; since we execute a depth-first search of the nodes, we obtain a spanning tree.

As for point b), the estimate $m$ of the number of nodes is increased each time a not visited node receives the token, and the depth-first visit approach guarantees all nodes are visited.

Let us now prove point c). Note that, when a leaf node transmits the token to its parent, it transmits its depth over the tree, while non-leaf nodes $v_i$ transmit to their parents the value of the maximum depth $\Delta^i$ of one of the nodes in the branch of $v_i$. As a result, each node knows the correct value of $\Delta^i$ and $\Delta^i$.

Let us consider the generic node $v_i$ sending the token to its parent along the tree. If $v_i$ is a leaf, then it sends $\delta_T^i = \Delta^i$ to its parent. If it is not a leaf, nor the leader, it calculates $\delta_T^i$ as the maximum between $\delta_T^i$ and $\Delta^i + \Delta^i - 2\Delta_i$. The first term is the maximum length of a path from the root node to one of the leaves in the branch of $v_i$. As for the second term, we claim that it is the maximum length of a path connecting the 2 farthest leaves (if there is more than one) in the branch of node $v_i$; in fact, the distance between $v_i$ and its farthest leaf is $\Delta^i - \Delta_i$, while the distance from the second farthest one (if any) is $\Delta^i - \Delta_i$.

Therefore, it can be noted that, when a node $v_i$ transmits a token to its parent, all possible paths of maximum length involving $v_i$ and the nodes in the branch of $v_i$ have been inspected, and the maximum among the lengths of such paths is stored in $\delta_T^i$; hence, when the last token is received by the root node it must hold $\delta_T = \delta$.

Let us now discuss the computational characteristics of the proposed DSTC algorithm.

**Proposition 1:** The DSTC algorithm is such that:

a) it requires $2e + 2n - 2$ total messages and at most $2|N_i|$ messages for each node;

b) each message is $O(\log n)$ bits;

c) the total memory required at each node is $O(|N_i| \log n)$ bits.

**Proof:** a) If $G$ is connected, then exactly $2n - 2$ tokens are sent, because a token is sent twice along all the links assigned to the spanning tree, which are $n - 1$. Since eventually all nodes are visited, each node provides an additional message to all its neighbors, and for each edge two such messages are sent, hence the total number of messages sent (without counting the leader election) is $2e + 2n - 2$, while each node sends at most $2|N_i|$ messages (i.e., a token and a visited message to all its neighbors).

b) Each message requires $O(\log n)$ bits, as the quantities exchanged are Boolean variables or integers in the range $[0, n]$.

c) As for the memory requirements for each agent, they need to store a label for each neighbor, and using progressive integers for the ID of the trees each label requires $O(\log n)$ bits, hence the memory requirement is $O(|N_i| \log n)$ bits for each agent.

Note that, the DSTC algorithm can be implemented in an asynchronous fashion, as only one message is sent at a time in the network. Notice further that, at the end of the procedure, the leader node knows the number $n$ of nodes in the network and the diameter $\delta$ of the tree; this information can be provided to all other nodes by means of $n - 1$ more messages (i.e., along the spanning tree).

In the next section we provide a methodology to solve Problem 1 in an approximated manner.

VI. CONSTRUCTION OF MULTIPLE EDSTS

As discussed above, in order to provide a suboptimal solution to Problem 1, we attempt to build multiple edge-disjoint spanning trees by iteratively executing the DSTC Algorithm and removing the edges in the found spanning tree, and so on until DSTC fails to construct a spanning tree; we refer to this algorithm as *Distributed Edge-Disjoint Spanning Trees Construction Algorithm* (DESTC), and we report its pseudocode in Algorithm 2.

Specifically, within DESTC the leader node starts a DSTC procedure, and when the procedure terminates each node discards the labeled edges. After the $t$-th run of DSTC, the leader starts a new DSTC if $t = 1$ (i.e., if only one DSTC has been performed) or if the number of visited nodes $m_t$ during the $t$-th DSTC is equal to the number of nodes visited at the previous run. When the condition is not met, the leader stops and the algorithm terminates.

Note that, over a connected graph, there is the guarantee to find at least one spanning tree, hence the leader node is able to detect the failure or success of subsequent executions of the DSTC Algorithm by comparing the estimate of the number of nodes obtained at each execution after the first estimate.

Note further that there is no need to elect a new leader at each execution of the DSTC Algorithm; it is however possible to further reduce the overall computations by selecting different leaders, as discussed in the next remark.

**Remark 6:** If the number of nodes does not change during the execution of the DESTC Algorithm, a simple improvement
consists in executing once the DSTC algorithm, and then simplifying the successive runs of DSTC. Specifically, when the \( n \)-th node is visited, there is no need to traverse the links in the spanning tree up to the leader. It is sufficient to let the \( n \)-th node become the new leader and begin a new depth-first visit. Such an approach would result in a reduction of half of the time steps, and would be more robust because it does not rely on a single leader. However, for simplicity, in the following we neglect this possibility.

**Algorithm 2** Distributed Edge-Disjoint Spanning Trees Construction (DESTC)

1. **procedure** DESTC(is-leader)
2. \[ if \text{is-leader} \] then
3. \[ t \leftarrow 0; \]
4. \[ \text{start first DSTC}; \]
5. **procedure** OnFINISHEDDSTC(is-leader)
6. remove labeled edges from consideration;
7. **if** is-leader then
8. \[ t \leftarrow t + 1; \]
9. \[ m_t \leftarrow \text{visited nodes in } t\text{-th run of DSTC}; \]
10. **if** \( t = 1 \) or \( m_t = m_{t-1} \) then
11. \[ \text{start new DSTC}; \]
12. **else** // DSTC failed to find a spanning tree
13. stop;

**A. Computational Properties**

The following proposition characterizes the number of messages exchanged during the execution of the DESTC algorithm.

**Proposition 2:** The overall number \( \phi \) of messages exchanged within the DESTC Algorithm is such that

\[
\phi \in \left[k\psi, (k+1)\psi\right],
\]

where

\[
\psi = 2e + 2(n-1) \left[1 - \frac{(k-1)(k-2)}{2}\right]
\]

and \( k \) is the number of EDSTs found by Algorithm DESTC. Moreover, each node sends at most \( 2(k+1)|N_i| \) messages.

**Proof:** The number of messages required for the first execution is \( 2e + 2(n-1) \), while for each execution \( t \) we need to discard

\[
2 \sum_{i=1}^{k} (t-1)(n-1)
\]

links, because at each execution \( n-1 \) links are removed from the graph. Hence, the total number of messages required for \( k \) executions is

\[
\phi' = k[2e + 2(n-1)] - 2k \sum_{i=1}^{k} (t-1)(n-1)
\]

or, after some algebra, \( \phi' = k\psi \). The root node may have some links left after \( k \) executions, hence the agents attempt to execute the DSTC Algorithm once more; the total number of messages exchanged is, therefore, \( \phi < (k+1)\psi \).

**Remark 7:** The memory requirements and the size of the messages within DESTC Algorithm coincide with those of DSTC Algorithm.

**VII. SIMULATIONS**

In this section we analyze the performance of the proposed DSTC and DESTC algorithms and we provide examples of applications that cope with MITM attacks.

**A. Experimental Setting**

In the following, we consider a **random geometric graph**, i.e., we place the nodes in uniformly distributed random positions in the unit square \([0,1] \times [0,1]\) and we connect them by an edge when their Euclidean distance is less than or equal to a threshold \( \rho \).

**B. DSTC and DESTC: Computational Properties**

Figure 4 shows the average of the total number of messages exchanged by the agents during the DESTC algorithm (magenta squares) and for the DSTC algorithm (green triangles). As discussed in Section VII-A, we consider random geometric graphs with \( n = 30 \) and different values of \( \rho \). The plot shows the average of the results for \( m = 500 \) trials.

**Fig. 4.** Average number of messages for the DESTC algorithm (magenta squares) and for the DSTC algorithm (green triangles). As discussed in Section VII-A, we consider random geometric graphs with \( n = 30 \) and different values of \( \rho \). The plot shows the average of the results for \( m = 500 \) trials.

**C. DESTC versus Optimal but Centralized Approach**

We now evaluate the results of DESTC in terms of the number of found EDSTs, depending on the communication radius \( \rho \). Specifically, we compare the number of EDSTs found by the proposed algorithm (DESTC) against the algorithm
from Roskind and Tarjan (R&T) [27] and against the lower bound (LB) discussed in Section IV-C.

Figure 5 shows a comparison of DESTC (lower plots) and [27] R&T (upper plots) on a particular instance with \( n = 30 \) nodes and \( \rho = 0.65 \). As shown by the picture, DESTC Algorithm finds 6 EDSTs, while the algorithm in [27] finds \( \tau(G) = 10 \) EDSTs. Within DESTC Algorithm, instead, there are 122 spare links that constitute a disconnected graph, implying that a different choice of the links would have yielded a bigger number of EDSTs.

In Figure 6, we consider random geometric graphs with \( n = 30 \) nodes and for each choice of \( \rho \in [0.4, 0.95] \) we generate 500 connected graphs. According to the figure, we note that the median value for DESTC is above the one for LB when \( \rho \leq 0.8 \); moreover, it can be noted that the 5th percentile for DESTC is above the median of LB for \( \rho = 0.4 \), and it is above the 25th percentile of LB for \( \rho = 0.45 \). For higher values of \( \rho \), the results for DESTC and LB tend to coincide. As for the comparison with R&T, it can be noted that for \( \rho < 0.5 \) the results are comparable for the median values, although R&T has better results in terms of 75th and 95th percentile. For \( \rho \geq 0.5 \), instead, the 75th percentile for DESTC is below the 25th percentile for R&T, while for \( \rho \geq 0.75 \) the 75th percentile for DESTC is below the 5th percentile for R&T. The results suggest that the proposed algorithm has good performance for relatively sparse networks, while it gets increasingly closer to the lower bound as \( \rho \) grows; it should be noted, however, that the distributed setting is of particular interest in the case of sparse networks, therefore the proposed algorithm appears well justified. Note that, in the above simulation, the EDSTs found by DESTC have average diameter between 24 (\( \rho = 0.4 \)) and 25.67 (\( \rho = 0.95 \)), while R&T obtains considerably smaller average diameter, being between 8 and 10; in both cases, however, the average diameter is well above the diameter of the graph being partitioned, which in our trials is around 4 for \( \rho = 0.4 \) up to 2 for \( \rho = 0.95 \).

D. Coping with MITM Attacks

We now provide an example showing the potential of the proposed approach in terms of detection of MITM attacks and restoration of the correct result. We consider a random
geometric graph with \( n = 30 \) nodes and \( \rho = 0.7 \), resulting in \(|E| = 963\) links. By means of DESTC Algorithm, we find 9 EDSTSs and we analyze the effectiveness of MITM attacks affecting distributed algorithms running in parallel over the found EDSTSs.

Specifically, in Figure 7 we report the results obtained with respect to agents executing the max-consensus algorithm (upper plot, see [42] and references therein for details on max-consensus algorithms) and the average-consensus algorithm [1] (lower plot). Note that, since the average-consensus algorithm has asymptotical convergence, we implement a stopping criterion and we approximate the result of each instance. In each plot we consider MITM attacks affecting a given number of links (up to 30) selected at random and we plot, depending on the number of attacked links the percentage of successful detections of the attack (in red) and the percentage of successful restoration of the correct result (in blue). In the max-consensus case we assume the \( i \)-th agent starts with initial condition equal to \( i \) (so that the maximum is \( n = 30 \)) and the attackers replace the transmitted values with a value \( 2n = 60 \), so that by affecting one link the result of the corresponding EDSSG is 60 instead of 30. In the average-consensus case we do the same (hence the average is 15.5), but since we approximate the asymptotic result at a given point, we assume that solutions differing of less than 0.1 are counted as if they were the same result. According to the figure, the ability to cope with MITM attacks is radically different in the case of max-consensus and average-consensus. In the first case the agents are able to detect the attack with high probability in spite of the attacks; the detection percentage is around 100% if up to 13 links are attacked, while it degrades as more links are attacked (for 30 links attacked the detection percentage is around 20%). In the case of average-consensus, instead, the percentage is always 100%, since the attackers have no obvious way to guarantee that each attacked instance will yield the same result, hence in general the attacks yield different results, and the agents are able to spot the attack. As for the restoration probability, according to Remark 2, having 9 EDSTSs we are guaranteed to be able to restore attacks to no more than 4 links in all cases, while the percentage degrades as more links are attacked. However, the results for average-consensus are remarkably better than max-consensus; as discussed above, the attackers might not be able to steer the result of the different instances to the same result; hence, although not being the absolute majority, there might be a relative majority of instances that are not attacked (e.g., two instances having the same value while all other instances have erroneous but different values).

VIII. CONCLUSIONS

In this paper we provide a lightweight and asynchronous distributed methodology to construct a set of EDSTSs. Such EDSTSs can be used to execute in parallel several instances of the same distributed algorithm, so that the algorithm becomes robust to man-in-the-middle attacks. Future work will be aimed at extending the methodology to directed graphs and to increase the degree of parallelism in the construction of the EDSTSs. Moreover we will investigate strategies to construct a smaller set of EDSSGs with small diameter; to this end the ability of DSCT Algorithm to find the diameter of the EDSTSs it constructs will be highly beneficial. Another important research direction is the definition of performance bounds of the proposed approach with respect to the optimal but centralized algorithm.

REFERENCES


