Distributed Monitoring of Network Properties: The Power of Hybrid Networks[*]

Robert Gmyr[1], Kristian Hinnenthal[2], Christian Scheideler[3], and Christian Sohler[4]

1 **Paderborn University, Paderborn, Germany**
`gmyr@mail.upb.de`
2 **Paderborn University, Paderborn, Germany**
`krijan@mail.upb.de`
3 **Paderborn University, Paderborn, Germany**
`scheideler@mail.upb.de`
4 **TU Dortmund, Dortmund, Germany**
`christian.sohler@tu-dortmund.de`

Abstract

We initiate the study of network monitoring algorithms in a class of hybrid networks in which the nodes are connected by an *external network* and an *internal network* (as a short form for externally and internally controlled network). While the external network lies outside of the control of the nodes (or in our case, the monitoring protocol running in them) and might be exposed to continuous changes, the internal network is fully under the control of the nodes. As an example, consider a group of users with mobile devices having access to the cell phone infrastructure. While the network formed by the WiFi connections of the devices is an external network (as its structure is not necessarily under the control of the monitoring protocol), the connections between the devices via the cell phone infrastructure represent an internal network (as it can be controlled by the monitoring protocol). Our goal is to continuously monitor properties of the external network with the help of the internal network. We present scalable distributed algorithms that efficiently monitor the number of edges, the average node degree, the clustering coefficient, the bipartiteness, and the weight of a minimum spanning tree. Their performance bounds demonstrate that monitoring the external network state with the help of an internal network can be done much more efficiently than just using the external network, as is usually done in the literature.

1998 ACM Subject Classification C.2.1 [Network Architecture and Design] Distributed networks, G.2.2 [Graph Theory] Graph Algorithms

Keywords and phrases Network Monitoring, Hybrid Networks, Overlay Networks

Digital Object Identifier 10.4230/LIPIcs.ICALP.2017.137

1 Introduction

In this paper we propose a new model for the study of distributed algorithms for communication networks that is based on a class of hybrid networks that is becoming more and more important. In this class of hybrid networks, the nodes are connected by an *external*

[*] This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Center "On-The-Fly Computing" (SFB 901) and within Project A2 of the Collaborative Research Center "Providing Information by Resource-Constrained Analysis" (SFB 876).

network and an *internal network*. While the external network is not under the control of the nodes, the internal network is fully under their control. Such hybrid networks can be found at a physical as well as logical level. Consider, for instance, the case that we have a set of wireless devices with access to the cell phone infrastructure that are dispersed over a limited area like a city center so that they can form a connected network using their WiFi connections. The advantage of this type of network is that the devices would in principle be able to exchange information without the use of the cell phone infrastructure, which would save costs. However, this may come at the price of having large message delays and even being unable to handle certain tasks as there might be network partitions from time to time. Therefore, if it is possible to design protocols that only require a small amount of message exchanges via the cell phone infrastructure in order to solve certain tasks much faster and more reliably than via the WiFi network, users may find it acceptable to make use of the cell phone infrastructure. Another example is an expedition or a rescue team that is connected via satellite telephones, which nowadays can support both satellite as well as wireless communication. In the logical world, one can envision a peer-to-peer network formed by friendship links in a social network. Just communicating via these friendship links has the advantage that all interactions are trusted. However, due to the irregular structure of the social network it has the disadvantage that it might be hard to perform certain tasks like network monitoring or finding anyone efficiently. Therefore, it might also be useful to have a network of untrusted links on top of the social network in order to be able to quickly approximate certain properties of it or to find shortest paths. A common theme in all of these examples is having two communication modes that significantly differ concerning their control and in which control comes with costs like financial cost, acceptance, reliability, or integrity. There is already a large body of literature on network algorithms for the case of static or dynamic networks whose topology is not under the control of the nodes. On the other side, there also exists an abundance of network algorithms in which the topology is fully under the control of the nodes, like in peer-to-peer systems. However, to the best of our knowledge, nothing rigorous in the context of network monitoring has been shown yet for *combinations* of these networks, so this paper initiates the rigorous study of this direction.

1.1 Model and Problem Statement

We consider networks with a *static* node set and a *dynamic* edge set. Time proceeds in *synchronous rounds* and for each round i we are given a set of undirected edges E_i. The *external network* in round i is represented by the undirected graph $G_i = (V, E_i)$. We assume that the degree of G_i is polylogarithmic for all i. An algorithm has no control over the edges in E_i, however it can establish additional *overlay edges* to form an *internal network* or *overlay network*: Each node u has a unique *identifier* $\mathrm{id}(u)$ which is a bit string of length $O(\log n)$ where $n = |V|$. Let $D_i(u)$ be the set of identifiers stored by a node u in round i. We define the set of overlay edges in round i as $D_i = \{(u, v) \mid u \in V \text{ and } v \in D_i(u)\}$. A node has immediate access to the identifiers of its neighbors in G_i and can store such an identifier for future reference. In round i, a node u can send a distinct message to each node v such that $\{u, v\} \in E_i$ or $(u, v) \in D_i$. A message sent in round i arrives at the beginning of round $i + 1$. The local memory and computation of the nodes is unbounded. However, a node can send and receive at most polylogarithmically many bits in each round.

We investigate *monitoring problems*. In these problems, a designated node s that we call the *monitor node* or simply *monitor* has to continuously observe a property of the external network like the number of edges or the weight of a minimum spanning tree. Formally, a property p is a function from the set of undirected graphs into some set of property values.

Since the external network G_i is dynamic, the property value $p(G_i)$ can change from round to round. We say an algorithm *monitors a property* p with *setup time* i_0 and *delay* δ if for all rounds $i \geq i_0$ the monitor node outputs the property value $p(G_i)$ by round $i + \delta$. We refer to the first i_0 rounds of the execution of a monitoring algorithm as the *setup phase* and refer to the remaining rounds as the *monitoring phase*. Initially, the set of overlay edges D_0 is empty. An algorithm can use the setup phase to construct an initial internal network that supports the computation of the property value. It can continue to adapt the internal network during the monitoring phase. We assume the graph G_0 to be connected. Beyond this, we make no assumptions about the evolution of the edge set.

1.2 Related Work

In the networking community, the name "hybrid network" has been used in the context of networks containing equipment from multiple vendors, consisting of different physical networks or communication modes, or networks incorporating both peer-to-peer and client-server approaches. These topics are not related to our work, so we do not consider them.

There is a large body of literature on overlay networks, especially in the context of peer-to-peer systems. Whereas most of the proposed overlay networks do not worry about the underlying network, there is also a number of proposals for so-called locality-aware overlays, with prominent examples like Tapestry [37] and Pastry [34]. However, these constructions are only concerned about adapting or optimizing the overlay to the underlying network and do not aim at monitoring properties of the underlying network with the help of the overlay.

The dynamic external network assumed by our model is related to the *dynamic graph model* introduced by Kuhn et al. [24], in which an adversary changes the edge set of a graph in every round. Kuhn et al. [24] focus on solving the counting and the token dissemination problem in that model, which has been further considered, for example, in [12, 15] (see [5] for an overview). Abshoff and Meyer auf der Heide study how to perform continuous aggregation in these networks [1]. However, like the other works in this area, they do not consider establishing additional overlay edges. Another approach related to ours is the work by Michail and Spirakis [28], which extends the population protocol model to a model in which nodes can decide whether to keep connections proposed to them or not, but there is no underlying network to monitor.

Some of our algorithms make use of techniques particularly known from the field of parallel computation. For example, it is well-known how to use *pointer jumping* [18] in order to perform rapid tree traversals in PRAMs (see e.g. [3, 19, 35]). Furthermore, there exists an abundance of parallel algorithms computing MSTs in such models, the best of which achieve a runtime of $O(\log n)$ (see [16] for an overview). The algorithm presented in Section 2 has some similarities with [19]. However, we are not aware of any distributed implementation of such an algorithm with runtime $o(\log^2 n)$ that does not cause high node congestion. This is also the problem with the various algorithms proposed for the congested clique model, which has recently received a considerable amount of attention (e.g., [7, 11, 17, 25, 27]).

Our algorithms make extensive use of a subroutine for the construction of overlay networks that we present in Section 2. This subroutine transforms a given graph into a rooted tree of constant degree and depth $O(\log n)$. Angluin et al. [2] proposed a similar subroutine that achieves the same result and that even works in an asynchronous setting. However, the subroutine of Angluin et al. is randomized while our subroutine is deterministic. As a consequence, all monitoring algorithms presented in this work are also fully deterministic. Furthermore, our subroutine can be used for the efficient construction of a spanning tree of a given graph, which cannot directly be achieved using the approach by Angluin et al.

Table 1 This table summarizes the results of this work. W is the maximum weight of an edge in the graph. The algorithm for monitoring the exact weight of a minimum spanning tree requires integral edge weights while the approximation algorithm has no such requirement. The latter algorithm approximates the weight M up to an additive factor of $\pm\varepsilon M$.

Monitoring Problem	Setup Time	Delay	Section
Number of Edges	$O(\log^2 n)$	$O(\log n/\log\log n)$	3
Average Node Degree	$O(\log^2 n)$	$O(\log n/\log\log n)$	3
Clustering Coefficient	$O(\log^2 n)$	$O(\log n/\log\log n)$	3
Bipartiteness	0	$O(\log^2 n)$	4
Exact MST Weight	0	$O(W+\log^2 n)$	5.1
Approximate MST Weight	$O(\log^2 n)$	$O(\log(W)/\varepsilon \cdot \log^2(W/\varepsilon) + \log n/\log\log n)$	5.2

In the algorithms presented in Section 3 the monitor continuously collects data from the nodes of the network by performing *aggregation*. There is a huge amount of work on aggregation in the context of sensor networks, but research in this area has focused on monitoring environmental properties, the state of systems (like bridges or airplanes) or facilities (like warehouses). Distributed aggregation has also been studied extensively for conventional, static networks (see, e.g., [4, 22, 23] or [26] for a comprehensive overview), but not for hybrid forms as considered in this paper.

One of the network properties considered in this work is the weight of a minimum spanning tree (or MST), see Section 5. The problem of computing an MST in a distributed manner is well studied, see for example [13, 14, 32, 33]. The problem of computing only the *weight* of an MST instead of the MST itself has been studied in the area of sequential sublinear algorithms. This line of research was initiated by Chazelle et al. [8] and continued in [6, 9, 10]. Our algorithms for monitoring the weight of an MST apply the ideas of Chazelle et al. [8] in a distributed context and also incorporate some ideas from [10].

1.3 Our Contribution

We initiate the study of hybrid networks consisting of externally and internally controlled edges and present *deterministic* algorithms for *monitoring network properties* in such networks. Our results are summarized in Table 1. As a byproduct of the algorithms for monitoring the weight of a minimum spanning tree, we also present algorithms for the distributed computation of minimum spanning trees in hybrid networks. Since the delays trivially increase to $\Omega(n)$ in the worst case when just using an external network, our results demonstrate that with the help of hybrid networks monitoring can be done exponentially faster compared to just having an external network.

2 Setup Phase

All monitoring algorithms presented in this work that rely on a dedicated setup phase use a common algorithm for the construction of the initial overlay network. This algorithm organizes the nodes into a tree T of polylogarithmic degree and depth $O(\log n/\log\log n)$ that is rooted at the monitor. In this section, we first present a more general algorithm that we refer to as the *Overlay Construction Algorithm*. This algorithm shares some similarities with an algorithm of Angluin et al. [2]. The algorithm is also frequently used as a subroutine throughout the remainder of this work. Based on this algorithm, we describe how the desired tree T can be constructed at the end of the section. For simplicity, we assume that every node knows the total number of nodes n. The algorithms can be modified to remove this assumption.

For a given bidirected connected graph G of polylogarithmic degree, the Overlay Construction Algorithm arranges the nodes of G into a tree of constant degree and depth $O(\log n)$. On a high level, the algorithm works as follows. It operates on *supernodes* which are groups of nodes that act in coordination. Let the identifier of a supernode be the highest identifier of the nodes it contains. Define two supernodes u, v to be adjacent if there are nodes x, y that are adjacent in G such that x is in u and y is in v. Initially, each node forms a supernode on its own. The algorithm alternatingly executes a *grouping step* and a *merging step*. In the grouping step, each supernode u determines the neighboring supernode v with the highest identifier. If $\mathrm{id}(v) > \mathrm{id}(u)$ then u sends a *merge request* to v. Consider the graph whose node set is the set of all supernodes and that contains a directed edge (u, v) if u sent a merge request to v. Since each supernode sends at most one merge request to a supernode of higher identifier, this graph is a forest. During the merging step, each tree of this forest is merged into a new supernode. Before we describe how this high-level algorithm can be implemented by the nodes, we analyze the number of iterations of consecutive grouping and merging steps until only a single supernode remains. The following lemma can be shown by observing that each supernode merges with another supernode within at most 2 iterations.

▶ **Lemma 1.** *After $O(\log n)$ iterations only a single supernode remains.*

At the beginning of every *grouping step*, the following *invariant* holds: Each supernode is internally organized in an overlay forming a tree of constant degree and depth $O(\log n)$ that is rooted at the node with the highest identifier and each node knows the identifier of its supernode. The nodes cooperatively simulate the behavior of their respective supernodes during the grouping step as follows. Consider a supernode u and the corresponding internal tree T_u. First, every node of u sends $\mathrm{id}(u)$ along every incident edge in the original graph G. Thereby, every node learns the identifiers of its neighboring supernodes. Then, the nodes of u use a convergecast along T_u to determine the identifier of the supernode v with the highest identifier among the neighbors of u. This convergecast also collects the identifier of the node x with the highest identifier in u that is adjacent to a node in v. Once this convergecast is complete, the root of T_u knows both $\mathrm{id}(v)$ and $\mathrm{id}(x)$. If $\mathrm{id}(v) > \mathrm{id}(u)$ then the root of u sends a message to x. Upon receiving this message, x sends a merge request to a neighboring node in v and sends a broadcast through T_u to establish itself as the new root of T_u. The nodes in G wait with starting the merging step until $O(\log n)$ rounds have passed to guarantee that the above operations are completed in all supernodes and all nodes start the merging step at the same time.

At the beginning of every *merging step*, we have the following situation. Consider the graph consisting of the internal trees of all supernodes together with all edges along which a merge request has been sent. This graph is a forest and the trees of this forest form the new supernodes resulting from the merging step. Therefore, the nodes of each new supernode v are already arranged into a tree T_v. Furthermore, v contains exactly one former root node that did not instruct a node to send a merge request. It is not hard to see that this node has the highest identifier in v and therefore becomes the root of T_v. In its current state, T_v can have up to polylogarithmic degree and linear depth. To restore the invariants required at the beginning of a grouping step, we have to transform T_v into a tree of constant degree and depth $O(\log n)$. Furthermore, we have to make sure that all nodes in v know the identifier $\mathrm{id}(v)$ of the root of T_v.

First, we reorganize T_v into a *child-sibling tree*. For this, each inner node y arranges its children into a path sorted by increasing identifier and only keeps the child with the lowest identifier. Each former child of y changes its parent to be its predecessor on the path and stores its successor as a *sibling*. In the resulting child-sibling tree, each node stores at most three identifiers: a parent, a sibling, and a child. By interpreting the sibling of a node as a second child, we get a binary tree. This transformation of T_v into a binary tree takes $O(1)$ rounds.

Based on this binary tree, we construct a *ring* of *virtual nodes* as follows. Consider the depth-first traversal of the tree that visits the children of each node in order of increasing identifier. A node occurs at most three times in this traversal. Let each node act as a distinct virtual node for each such occurrence and let $k \leq 3n$ be the number of virtual nodes. A node can locally determine the predecessor and successor of its virtual nodes according to the traversal. Therefore, the nodes can connect their virtual nodes into a ring in $O(1)$ rounds.

Next, we use pointer jumping to quickly add *chords* (i.e., shortcut edges) to the ring. The virtual nodes execute the following protocol for $\lfloor \log n \rfloor + 1 \geq \lfloor \log k \rfloor - 1$ rounds. Each virtual node y learns two identifiers ℓ_t and r_t in each round t of this protocol. Let ℓ_0 and r_0 be the predecessor and successor of y in the ring. In round t, y sends ℓ_t to r_t and vice versa. At the beginning of round $t+1$, y receives one identifier from ℓ_t and r_t, respectively. It sets ℓ_{t+1} to the identifier received from ℓ_t and r_{t+1} to the identifier received from r_t. It then proceeds to the next round of the protocol. In every round of this protocol each virtual node adds a new chord to the ring by introducing its latest neighbors to each other. The distance between these neighbors w.r.t the ring doubles from round to round up to the point where the distance exceeds the number of virtual nodes k. Based on this observation, it is not hard to show that after the specified number of rounds, the diameter of the graph has reduced to $O(\log n)$ while the degree has grown to $O(\log n)$. Once the protocol finished, the root of v initiates a broadcast from one of its virtual nodes followed by a convergecast to determine the number of virtual nodes k.

Finally, we use the chords to construct a binary tree of depth $O(\log n)$. For this, the root of v initiates a broadcast by sending a message to its neighbors $\ell_{t'}$ and $r_{t'}$ where $t' = \lfloor \log k \rfloor - 1$. A node that receives the broadcast after t steps forwards it to $\ell_{t'}$ and $r_{t'}$ where $t' = \max\{\lfloor \log k \rfloor - t - 1, 0\}$. It is not hard to see that the binary tree constructed by this broadcast has depth $O(\log n)$ and contains all nodes of the ring. At this point, the nodes discard all overlay edges constructed so far and only keep the edges of the binary tree. We then merge the virtual nodes back together such that each node adopts the edges of its virtual nodes. This results in a graph of degree at most 6 and diameter $O(\log n)$. Note that this graph is not necessarily a tree. To construct a tree that satisfies the invariants for the grouping step, the root of v sends another broadcast through the resulting graph to construct a breadth-first search tree that has constant degree and diameter $O(\log n)$. This broadcast also informs all nodes in v about $\text{id}(v)$. The operations described above take $O(\log n)$ rounds overall. As before, all nodes in G wait for $O(\log n)$ rounds to pass so that they enter the next grouping step at the same time.

Once only a single supernode u remains, which is the case if during a grouping step no node reports the identifier of a neighboring supernode, T_u covers all nodes of G and has the desired properties. Since the algorithm runs for $O(\log n)$ iterations and each iteration takes $O(\log n)$ rounds, we have the following theorem.

▶ **Theorem 2.** *Given any bidirected connected graph G of n nodes and polylogarithmic degree, the Overlay Construction Algorithm constructs a constant degree tree of depth $O(\log n)$ that contains all nodes of G and that is rooted at the node with the highest identifier. The algorithm takes $O(\log^2 n)$ rounds.*

Theorem 2 directly implies the following corollary.

▶ **Corollary 3.** *Consider a bidirected graph G of n nodes and polylogarithmic degree. For each connected component C of G, the Overlay Construction Algorithm constructs a constant degree tree of depth $O(\log |C|)$ that contains all nodes of C and that is rooted at the node with the highest identifier in C. The algorithm takes $O(\log^2 |C|)$ rounds in each component and $O(\log^2 n)$ rounds overall.*

Finally, note that the algorithm only sends merge requests along edges of G. This gives rise to the following observation.

▶ **Observation 4.** *For a bidirected connected graph G, the edges along which the Overlay Construction Algorithm sends the merge requests form a spanning tree of G.*

So by letting the nodes locally mark the edges that carry a merge request, the algorithm can be used for the distributed construction of a spanning tree of G in $O(\log^2 n)$ time. While this observation is not immediately relevant for the setup phase, it will be useful in later sections.

Based on the Overlay Construction Algorithm, it is easy to achieve the goal for the setup phase of organizing the nodes into a tree T of polylogarithmic degree and depth $O(\log n/\log\log n)$ that is rooted at the monitor s. At the beginning of the setup phase, each node stores the identifiers of its neighbors in the given graph G_0 that represents the external network. This effectively creates a bidirected overlay network that directly corresponds to the undirected graph G_0. Note that G_0 is connected by assumption. Therefore, we can use the Overlay Construction Algorithm to construct a tree of constant degree and depth $O(\log n)$ that contains all nodes in the network. Once the algorithm terminates, s broadcasts a message through the resulting tree to establish itself as the new root. This does not increase the asymptotic depth of the tree. We then decrease the depth to $O(\log n/\log\log n)$ as follows. Each node x broadcasts its identifier down the tree up to a distance of $\lceil \log\log n \rceil$. Every node that receives the broadcast of x establishes an edge to x. It is not hard to see that this creates a graph of at most polylogarithmic degree and diameter $O(\log n/\log\log n)$. Finally, s sends a broadcast through this graph to create a breadth-first search tree that has the desired properties. We have the following theorem.

▶ **Theorem 5.** *A setup time of $O(\log^2 n)$ rounds is sufficient to organize the nodes of the network into a tree T of polylogarithmic degree and depth $O(\log n/\log\log n)$.*

Unless otherwise stated, we assume in the following sections that the setup phase is executed as described above.

3 Three Simple Monitoring Problems

In order to introduce some basic concepts that underlie all monitoring algorithms presented in this work, we first consider three simple monitoring problems. Specifically, we show how to monitor the number of edges, the average node degree, and the clustering coefficient of the network by performing *aggregation* on the tree T constructed during the setup phase.

Consider the problem of monitoring the number of edges. We first present an algorithm that efficiently determines the number of edges in a graph G and then show how this algorithm can be used to continuously monitor the number of edges. It is well known that the number of edges in a graph is $|E| = 1/2 \cdot \sum_{u \in V} \deg(u)$ where $\deg(u)$ is the degree of a node u. Therefore, we can compute $|E|$ by aggregating the sum of all node degrees in the following way. In the first round, each leaf node u in T sends $\deg(u)$ to its parent. Once an inner node

u has received a value x_j from each of its children, it sends $\deg(u) + \sum_j x_j$ to its parent. After $O(\log n / \log\log n)$ rounds, the monitor s has received a value from each of its children and can use these values together with its own degree to compute $|E|$ as described above.

To continuously monitor $|E_i|$ for every $i \geq i_0$, the above algorithm is executed in a *pipelined fashion*: In each round $i \geq i_0$ a new instance of the algorithm is started. The instances run in parallel and do not interact with each other. At the beginning of a round i, each node stores the identifiers of its neighbors in the graph G_i that represents the external network to create a copy of the graph in form of an overlay network that the algorithm can operate on. This copy is discarded once the corresponding instance of the algorithm terminates. The messages sent by an instance of the algorithm are labeled with the round number in which the instance was started so that received messages can be correctly assigned. Note that the number of bits a node sends and receives per round in the given algorithm is polylogarithmic. Since the delay of the algorithm is also polylogarithmic, the number of bits a node sends and receives in the pipelined execution is polylogarithmic as well. We have the following theorem.

▶ **Theorem 6.** *The number of edges can be monitored with setup time $O(\log^2 n)$ and delay $O(\log n / \log\log n)$.*

In the remainder of this work, we only present algorithms that compute the value of a network property for a single graph G and implicitly assume that the respective algorithm is executed in a pipelined fashion to solve the monitoring problem under consideration.

Based on the ideas of the algorithm above, it is easy to solve a number of monitoring problems that can be reduced to aggregation. For example, one can monitor the average node degree by letting s multiply the result of the given algorithm with $2/n$ before outputting it. This gives us the following corollary.

▶ **Corollary 7.** *The average node degree can be monitored with setup time $O(\log^2 n)$ and delay $O(\log n / \log\log n)$.*

As a final example, we consider the *clustering coefficient* of a network [36]. Intuitively, the clustering coefficient reflects the relative number of triangles in a graph G. It is particularly relevant in the context of biological and social networks [29, 30, 31, 36]. Formally, the clustering coefficient of a node u is defined as

$$C(u) = \frac{2 \cdot |\{v, w \in N(u) \mid \{v, w\} \in E\}|}{\deg(u) \cdot (\deg(u) - 1)},$$

where $N(u)$ is the set of neighbors of u. The clustering coefficient of a graph G is defined as $C(G) = 1/n \cdot \sum_{u \in V} C(u)$. Each node u can compute $C(u)$ in constant time by communicating with its neighbors. Therefore, $C(G)$ can be computed by aggregating the sum of all $C(u)$ along T and dividing the result by n at the monitor. We have the following theorem.

▶ **Theorem 8.** *The clustering coefficient can be monitored with setup time $O(\log^2 n)$ and delay $O(\log n / \log\log n)$.*

4 Bipartiteness

In this section, we consider the problem of monitoring whether the network forms a *bipartite* graph. Our algorithm is based on the following commonly known approach (see e.g. [20]). Given a connected graph G, compute a rooted spanning tree of G and assign a color from $\{0, 1\}$ to each node that corresponds to the parity of its depth in the spanning tree. We say an edge is *valid* if it connects nodes of different colors and *invalid* otherwise. G is bipartite if and only if all edges are valid. It remains to show how this approach can be implemented efficiently in our framework.

According to Observation 4, we can use the Overlay Construction Algorithm to mark the edges of a spanning tree S in G. Define the monitor s to be the root of S. Each node has to determine the parity of its depth in S to set its color. Since S might have linear depth, a simple broadcast from s does not constitute an efficient solution to this problem. Instead, we use pointer jumping along a depth-first traversal of S to determine the colors of the nodes. We define the traversal of S as follows. The traversal starts at s and moves to the neighbor of s in S with the lowest identifier. For a node u, let $u_0, \ldots, u_{\deg(u)-1}$ be the neighbors of u in S arranged by increasing identifier. When the traversal reaches u from a node u_i, it continues on to node $u_{(i+1) \bmod \deg(u)}$. The traversal finishes when it reaches s from the neighbor of s in S with the highest identifier. Define the *traversal distance* $d(u)$ to be the number of steps required to reach u for the first time in this traversal. As we will show in Lemma 9, the parity of $d(u)$ equals the parity of the depth of u in S. For now, we focus on computing $d(u)$ efficiently for all nodes.

Each node u simulates one virtual node for each occurrence of u in the traversal, and the nodes connect these virtual nodes into a ring. More specifically, each node u simulates virtual nodes $v_0, \ldots, v_{\deg(u)-1}$ such that v_i is the successor of a virtual node of u_i and the predecessor of a virtual node of $u_{(i+1) \bmod \deg(u)}$ in the ring. Note that the resulting ring consists of $2(n-1)$ virtual nodes. We use pointer jumping to add chords to the ring following the same protocol we used during the merging step of the Overlay Construction Algorithm. We define ℓ_0 to be the successor of a virtual node and r_0 to be the predecessor of a virtual node. We execute the protocol for $t = \lfloor \log(2(n-1)) \rfloor$ rounds so that each node constructs chords ℓ_i and r_i for each $1 \leq i \leq t$. Each chord ℓ_i (resp. r_i) bridges a distance of exactly 2^i along the ring. The chords allow us to efficiently compute the values $d(u)$ in the following way. Let v^* be the virtual node simulated by s that precedes a virtual node of the neighbor of s with the lowest identifier. v^* stores the value 0 and initiates a broadcast by sending a message with value 2^i along each chord ℓ_i for $0 \leq i \leq t$. Consider a virtual node that receives a broadcast message and that has not yet stored a value. Let x be the value associated with the received message. The virtual node stores x and sends a message containing the value $x + 2^i$ along each chord ℓ_i for $0 \leq i \leq t$. It is not hard to see that this broadcast reaches all virtual nodes within $O(\log n)$ rounds and the value stored at a virtual node v after the broadcast finishes corresponds to the length of the path from v^* to v along the ring. Therefore, each node u can determine the value $d(u)$ by taking the minimum of the values stored at its virtual nodes. Once all nodes computed their traversal distance in this way, each node u determines whether it is incident to an invalid edge by checking for each neighbor w in G whether $d(u) \equiv d(w) \mod 2$. Then, the nodes use a convergecast to inform s whether there is an invalid edge. If so, s outputs that G is not bipartite. Otherwise, s outputs that G is bipartite.

To establish the correctness of the algorithm, we show the following lemma.

▶ **Lemma 9.** *For each node u, the parity of the depth of u equals the parity of $d(u)$.*

Proof. Note that the tree is traversed in a depth-first order. Therefore, the traversal takes an even number of steps between any two visits of the same node. Let P be the shortest path from s to u in S. The length of P equals the depth of u. The traversal follows P but takes a detour whenever it explores a branch outside of P. By the argument above, each such detour has even length. Therefore, the parity of the depth of u equals the parity of $d(u)$. ◀

We can monitor bipartiteness for a disconnected graph by performing the above algorithm on its connected components and aggregating the results on the tree T built in the setup phase. Furthermore, we can reduce the setup time to 0 by delaying the first monitoring phase until T is constructed, which does not asymptotically increase the total delay. This implies the following theorem.

▶ **Theorem 10.** *The bipartiteness of a graph can be monitored with setup time* 0 *and delay* $O(\log^2 n)$*.*

5 Minimum Spanning Tree

We now turn to the problem of monitoring the *weight* of a *minimum spanning tree* (or *MST*). We assume that the edge set changes from round to round but the external network always stays connected. Additionally, we assume that each edge has a weight that can also change every round. We present an algorithm that monitors the exact MST weight in Section 5.1 and an algorithm that monitors an approximation of the MST weight with a shorter delay in Section 5.2. Both algorithms are based on a sequential approximation algorithm by Chazelle et al. [8]. As a byproduct, we describe in Section 5.3 how the algorithms for computing the MST weight can be adapted for the distributed computation of an actual MST.

5.1 Exact MST Weight

The main idea behind the algorithm is to reduce the computation of the weight of an MST in a graph G to counting the number of connected components in certain subgraphs of G. This idea was first introduced by Chazelle et al. [8]. We assume that the edge weights are taken from the set $\{1, 2, \ldots, W\}$ for some given $W \in \mathbb{N}$. Define the *threshold graph* $G^{(\ell)}$ to be the subgraph of G consisting of all edges with weight at most ℓ, and define $c^{(\ell)}$ to be the number of connected components in $G^{(\ell)}$. The MST weight M can be computed from the values $c^{(\ell)}$ as shown in the following lemma.

▶ **Lemma 11** (Chazelle et al. [8]). *In a graph with edge weights from* $\{1, 2, \ldots, W\}$*, the MST weight is* $M = n - W + \sum_{i=1}^{W-1} c^{(i)}$*.*

Based on Lemma 11, the monitor can compute the MST weight as follows. Consider the threshold graph $G^{(\ell)}$ for some $\ell \in \{1, 2, \ldots W - 1\}$. According to Corollary 3, executing the Overlay Construction Algorithm on $G^{(\ell)}$ creates an overlay network in which each connected component of $G^{(\ell)}$ is spanned by a rooted tree of overlay edges. Each node knows whether it is a root of one of these trees. Therefore, we can determine $c^{(\ell)}$ by counting the number of roots, which can easily be achieved using aggregation along the tree T that results from the setup phase. By iterating this process, the monitor learns the value $c^{(\ell)}$ for each $\ell \in \{1, 2, \ldots W - 1\}$. It can then use the equation given in Lemma 11 to compute the MST weight. Since T is only used after the algorithm already ran for $O(\log^2 n)$ rounds, we can construct T during the monitoring phase and therefore skip the setup phase. Furthermore, we can reduce the delay by computing up to $\log^2 n$ different $c^{(i)}$'s in parallel. This implies the following theorem.

▶ **Theorem 12.** *For edge weights from* $\{1, 2, \ldots, W\}$*, the MST weight can be monitored with setup time* 0 *and delay* $O(W + \log^2 n)$*.*

5.2 Approximate MST Weight

Next, we present an algorithm that monitors the MST weight with a significantly shorter delay at the cost of a small approximation error. The algorithm is less restrictive in that it allows the edge weights to be real numbers from the interval $[1, W]$ for a given $W \in \mathbb{R}$. It is based on the same general idea as the algorithm from the previous section but additionally incorporates ideas from the work of Czumaj and Sohler [10].

First, each node rounds up the edge weight of each incident edge to a power of $(1+\varepsilon)$ for a fixed ε with $0 < \varepsilon \le 1$. In the resulting graph G', each edge weight is of the form $(1+\varepsilon)^i$ where $0 \le i \le \log_{1+\varepsilon} W$. Let M' be the MST weight in G'. We have the following lemma, which is analogous to Lemma 11 from the previous section.

▶ **Lemma 13** (Czumaj and Sohler [10]). *In a graph with edge weights of the form $(1+\varepsilon)^i$ for $0 \le i \le \log_{1+\varepsilon} W$, the MST weight is $M' = n - W + \varepsilon \cdot \sum_{i=0}^{\log_{1+\varepsilon} W - 1} (1+\varepsilon)^i \cdot c^{((1+\varepsilon)^i)}$.*

Based on Lemma 13, we can compute M' by determining the number of connected components $c^{((1+\varepsilon)^i)}$ in $\log_{1+\varepsilon} W$ many threshold graphs. While this already implies an improvement over the algorithm from the previous section, we can further reduce the delay by ignoring large components in the threshold graphs.

Consider some threshold graph $G^{((1+\varepsilon)^i)}$. We execute the Overlay Construction Algorithm as in the previous section but we stop its execution after $O(\log^2(2W/\varepsilon))$ rounds. By Corollary 3, the algorithm is guaranteed to finish its computation in each connected component of size at most $2W/\varepsilon$. In larger connected components, the algorithm may finish but is not guaranteed to do so. It is easy to modify the algorithm such that all nodes of a connected component know whether the algorithm finished its computation for that connected component. This allows us to ignore root nodes in connected components for which the algorithm did not finish. Thereby, the algorithm establishes a unique root node for each connected component of size at most $2W/\varepsilon$ while in each larger connected component either a unique root node is established or no root is established. Let $\hat{c}^{((1+\varepsilon)^i)}$ be the number of root nodes established in this way. The nodes determine the value of $\hat{c}^{((1+\varepsilon)^i)}$ using aggregation along the tree T constructed in the setup phase.

As in the previous section, the nodes iteratively execute this process for each i such that $0 \le i \le \log_{1+\varepsilon} W$. After the Overlay Construction Algorithm finishes for an iteration, the nodes start the aggregation for counting the number of roots. The nodes do not wait for this aggregation to finish but rather execute it in parallel to the next iteration. Thereby, we slightly interleave consecutive iterations which reduces the overall delay. After the monitor has learned the values $\hat{c}^{((1+\varepsilon)^i)}$, it computes and outputs $\hat{M} = n - W + \varepsilon \cdot \sum_{i=0}^{\log_{1+\varepsilon} W - 1} (1+\varepsilon)^i \cdot \hat{c}^{((1+\varepsilon)^i)}$. We have the following theorem.

▶ **Theorem 14.** *For edge weights from $[1, W]$, the MST weight M can be monitored up to an additive term of $\pm\varepsilon M$ for any $0 < \varepsilon \le 1$ with setup time $O(\log^2 n)$ and delay $O\left(\frac{\log W}{\varepsilon} \cdot \log^2\left(\frac{W}{\varepsilon}\right) + \frac{\log n}{\log\log n}\right)$.*

Proof. We first show the approximation factor. Rounding up the edge weights to a power of $(1+\varepsilon)$ increases the MST weight by a factor of at most $(1+\varepsilon)$. Therefore, we have $M \le M' \le (1+\varepsilon) \cdot M$. When computing the values $\hat{c}^{((1+\varepsilon)^i)}$ the algorithm potentially ignores all connected components of size larger than $2W/\varepsilon$. In each threshold graph there are at most $\varepsilon n/(2W)$ such connected components. The algorithm cannot overestimate the number of connected components. Therefore, we have $c^{((1+\varepsilon)^i)} - \varepsilon n/(2W) \le \hat{c}^{((1+\varepsilon)^i)} \le c^{((1+\varepsilon)^i)}$. For the upper bound on the output $\hat{M}$ of the algorithm, the equations above together with the definitions of M' and $\hat{M}$ directly imply $\hat{M} \le M' \le (1+\varepsilon) \cdot M$. For the lower bound on $\hat{M}$, we have $\hat{M} \geq n - W + \varepsilon \cdot \sum_{i=0}^{\log_{1+\varepsilon} W-1} (1+\varepsilon)^i \cdot \left(c^{((1+\varepsilon)^i)} - \frac{\varepsilon n}{2W}\right) = M' - \frac{\varepsilon^2 n}{2W} \cdot \sum_{i=0}^{\log_{1+\varepsilon} W-1} (1+\varepsilon)^i \geq M' - \frac{\varepsilon}{2} \cdot n \geq (1-\varepsilon) \cdot M$, where we assume $n \geq 2$ so that $n \leq M+1 \leq 2M$ for the last inequality.

We now turn to the delay of the algorithm. The algorithm iteratively computes the values $\hat{c}^{((1+\varepsilon)^i)}$ for $\log_{1+\varepsilon} W = O(\log(W)/\varepsilon)$ threshold graphs. In each of these iterations the modified Overlay Construction Algorithm is executed for $O(\log^2(W/\varepsilon))$ rounds. After the Overlay Construction Algorithm finishes in the last iteration, the nodes have to wait for the final aggregation to complete. This takes an additional $O(\log / \log\log n)$ rounds. ◀

Finally, if $W = n^{O(1)}$ we can execute $\log W$ iterations of the algorithm in parallel which gives us the following corollary.

▶ **Corollary 15.** *For edge weights from $[1, W]$ where $W = n^{O(1)}$, the MST weight M can be monitored up to an additive term of $\pm\varepsilon M$ for any $0 < \varepsilon \leq 1$ with setup time $O(\log^2 n)$ and delay $O\left(\frac{1}{\varepsilon} \cdot \log^2\left(\frac{W}{\varepsilon}\right) + \frac{\log n}{\log\log n}\right)$.*

5.3 Distributed Computation of MSTs

Based on the ideas of the previous two sections, we can devise algorithms that mark the edges of an MST instead of just computing the MST weight. While these algorithms do not fit into the monitoring context, they represent natural extensions of the given algorithms and demonstrate that the underlying ideas might be useful outside of network monitoring.

First, we run the Overlay Construction Algorithm on the graph $G^{(1)}$ and mark all edges along which a merge request is sent. We then add the edges of weight 2 and run the Overlay Construction Algorithm again to further merge the supernodes while still marking edges as before. We keep adding edges of increasing weight in this way until we reach the threshold graph $G^{(W)}$ in which only a single supernode remains. By Observation 4, the marked edges form a spanning tree of G. Furthermore, at any given time the algorithm only adds edges of minimal weight to the spanning tree. Therefore, the algorithm is essentially a distributed implementation of Kruskal's Algorithm [21] so that the spanning tree computed by the algorithm is in fact an MST. We have the following theorem.

▶ **Theorem 16.** *Consider a network that initially forms a bidirected connected graph G with edge weights from $\{1, 2, \ldots, W\}$ for some $W \in \mathbb{N}$. There is an algorithm that marks the edges of an MST in G in $O(W \log^2 n)$ rounds.*

Combining this approach with the ideas from Section 5.2 gives us the following theorem.

▶ **Theorem 17.** *Consider a network that initially forms a bidirected connected graph G with edge weights from $[1, W]$ for some $W \in \mathbb{R}$. Let M be the weight of an MST in G. There is an algorithm that marks the edges of a spanning tree in G with weight M' such that $M \leq M' \leq (1+\varepsilon) \cdot M$ in $O(1/\varepsilon \cdot \log W \cdot \log^2 n)$ rounds.*

6 Future Work

We were only able to present a small number of monitoring problems in hybrid networks in this work. However, there is an abundance of classical problems in the literature that can be newly investigated under this model. As we tried to demonstrate in Section 5.3, the idea of using the ability to establish new edges to speed up computation can also be applied outside of network monitoring. We would be very interested in seeing further applications of this idea in other contexts.

References

1 Sebastian Abshoff and Friedhelm Meyer auf der Heide. Continuous aggregation in dynamic ad-hoc networks. In Magnús M. Halldórsson, editor, *Structural Information and Communication Complexity - 21st International Colloquium, SIROCCO 2014, Takayama, Japan, July 23-25, 2014. Proceedings*, volume 8576 of *Lecture Notes in Computer Science*, pages 194–209. Springer, 2014. doi:10.1007/978-3-319-09620-9_16.

2 Dana Angluin, James Aspnes, Jiang Chen, Yinghua Wu, and Yitong Yin. Fast construction of overlay networks. In *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'05, page 145, 2005. doi:10.1145/1073970.1073991.

3 Mikhail Atallah and Uzi Vishkin. Finding euler tours in parallel. *Journal of Computer and System Sciences*, 29(3):330–337, 1984. doi:10.1016/0022-0000(84)90003-5.

4 Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley and Sons, Inc., 2nd edition, 2004.

5 John Augustine, Gopal Pandurangan, and Peter Robinson. Distributed algorithmic foundations of dynamic networks. *SIGACT News*, 47(1):69–98, 2016. doi:10.1145/2902945.2902959.

6 Petra Berenbrink, Bruce Krayenhoff, and Frederik Mallmann-Trenn. Estimating the number of connected components in sublinear time. *Information Processing Letter*, 114(11):639–642, 2014. doi:10.1016/j.ipl.2014.05.008.

7 Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic methods in the congested clique. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC'15, pages 143–152, 2015. doi:10.1145/2767386.2767414.

8 Bernard Chazelle, Ronitt Rubinfeld, and Luca Trevisan. Approximating the minimum spanning tree weight in sublinear time. *SIAM J. Comput.*, 34(6):1370–1379, 2005.

9 Artur Czumaj, Funda Ergün, Lance Fortnow, Avner Magen, Ilan Newman, Ronitt Rubinfeld, and Christian Sohler. Approximating the weight of the euclidean minimum spanning tree in sublinear time. *SIAM J. Comput.*, 35(1):91–109, 2005. doi:10.1137/S0097539703435297.

10 Artur Czumaj and Christian Sohler. Estimating the weight of metric minimum spanning trees in sublinear time. *SIAM J. Comput.*, 39(3):904–922, 2009. doi:10.1137/060672121.

11 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC'14, pages 367–376, 2014. doi:10.1145/2611462.2611493.

12 Chinmoy Dutta, Gopal Pandurangan, Rajmohan Rajaraman, Zhifeng Sun, and Emanuele Viola. On the complexity of information spreading in dynamic networks. In Sanjeev Khanna, editor, *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 717–736. SIAM, 2013. doi:10.1137/1.9781611973105.52.

13 Michael Elkin. Unconditional lower bounds on the time-approximation tradeoffs for the distributed minimum spanning tree problem. In *Proceedings of the 36th ACM Symposium on Theory of Computing*, STOC'04, pages 331–340, 2004. doi:10.1145/1007352.1007407.

14 Michael Elkin. A faster distributed protocol for constructing a minimum spanning tree. *Journal of Computer and System Sciences*, 72(8):1282–1308, 2006. doi:10.1016/j.jcss.2006.07.002.

15 Bernhard Haeupler and David Karger. Faster information dissemination in dynamic networks via network coding. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing*, PODC'11, pages 381–390, 2011. doi:10.1145/1993806.1993885.

16 Shay Halperin and Uri Zwick. Optimal randomized EREW PRAM algorithms for finding spanning forests. *Journal of Algorithms*, 39(1):1–46, 2001. doi:10.1006/jagm.2000.1146.

17 James W. Hegeman, Gopal Pandurangan, Sriram V. Pemmaraju, Vivek B. Sardeshmukh, and Michele Scquizzato. Toward optimal bounds in the congested clique: Graph connectivity and mst. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC'15, pages 91–100, 2015. doi:10.1145/2767386.2767434.

18 Joseph JaJa. *An Introduction to Parallel Algorithms*, volume 17. Addison Wesley, 1992.

19 Donald B. Johnson and Panagiotis Metaxas. A parallel algorithm for computing minimum spanning trees. *Journal of Algorithms*, 19(3):383–401, 1995. doi:10.1006/jagm.1995.1043.

20 Jon Kleinberg and Eva Tardos. *Algorithm Design: Pearson New International Edition*. Pearson Education Limited, 2013.

21 Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956. doi:10.2307/2033241.

22 Fabian Kuhn, Thomas Locher, and Stefan Schmid. Distributed computation of the mode. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing*, PODC'08, pages 15–24, 2008. doi:10.1145/1400751.1400756.

23 Fabian Kuhn, Thomas Locher, and Roger Wattenhofer. Tight bounds for distributed selection. In *Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'07, pages 145–153, 2007. doi:10.1145/1248377.1248401.

24 Fabian Kuhn, Nancy Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *Proceedings of the 42nd ACM Symposium on Theory of Computing*, STOC'10, pages 513–522, 2010. doi:10.1145/1806689.1806760.

25 Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In *Symposium on Principles of Distributed Computing*, PODC'13, pages 42–50, 2013. doi:10.1145/2484239.2501983.

26 Thomas Locher. *Foundations of aggregation and synchronization in distributed systems*. PhD thesis, ETH Zürich, 2009. doi:10.3929/ethz-a-005799819.

27 Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-weight spanning tree construction in O(log log n) communication rounds. *SIAM J. Comput.*, 35(1):120–131, 2005. doi:10.1137/S0097539704441848.

28 Othon Michail and Paul G. Spirakis. Simple and efficient local codes for distributed stable network construction. In *ACM Symposium on Principles of Distributed Computing*, PODC'14, pages 76–85, 2014. doi:10.1145/2611462.2611466.

29 Mark E. J. Newman. The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences*, 98(2):404–409, 2001. doi:10.1073/pnas.98.2.404.

30 Mark E. J. Newman, Steven H. Strogatz, and Duncan J. Watts. Random graphs with arbitrary degree distributions and their applications. *Physical Review E*, 64(2):26118, jul 2001. doi:10.1103/PhysRevE.64.026118.

31 Mark E. J. Newman, Duncan J. Watts, and Steven H. Strogatz. Random graph models of social networks. *Proceedings of the National Academy of Sciences*, 99(suppl 1):2566–2572, 2002. doi:10.1073/pnas.012582999.

32 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. Fast distributed algorithms for connectivity and mst in large graphs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'16, pages 429–438, 2016. doi:10.1145/2935764.2935785.

33 David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed mst construction. In *40th Annual Symposium on Foundations of Computer Science*, FOCS'99, pages 253–261, 1999. doi:10.1109/SFFCS.1999.814597.

34 Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329–350, 2001. `doi:10.1007/3-540-45518-3_18`.

35 Robert E. Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14(4):862–874, 1985. `doi:10.1137/0214061`.

36 Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998. `doi:10.1038/30918`.

37 Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004. `doi:10.1109/JSAC.2003.818784`.