Distributed Disaster Disclosure

Bernard Mans[1], Stefan Schmid[2], and Roger Wattenhofer[3]

[1] Department of Computing, Macquarie University, Sydney, NSW 2109, Australia
bmans@ics.mq.edu.au
[2] Institut für Informatik, Technische Universität München, 85748 Garching, Germany
schmiste@in.tum.de
[3] Computer Engineering and Networks Laboratory (TIK), ETH Zurich,
8092 Zurich, Switzerland
wattenhofer@tik.ee.ethz.ch

Abstract. Assume a set of distributed nodes which are equipped with a sensor device. When nodes sense an event, they want to know (the size of) the connected component consisting of nodes which have also sensed the event, in order to raise—if necessary—a disaster alarm. This paper presents distributed algorithms for this problem. Concretely, our algorithms aim at minimizing both the response time as well as the message complexity.

1 Introduction

Governments and organizations around the world provide billions of dollars each year in aid to regions impacted by disasters such as tornadoes, flooding, volcanos, earthquakes, bush-fires, etc. In order to recognize disasters early and in order to limit the damage, endangered environments are often monitored by a large number of distributed sensor devices. The idea is that when these devices sense an event, an alarm should be raised, e.g., to inform helpers in the local community. Unfortunately, in practice, the sensor devices may sometimes wrongfully sense events, and of course false alarms can be quite costly as well. Therefore, nodes sensing an event should make sure that there are other nodes in their vicinity which have sensed the same event. Clearly, as sensor nodes may only be equipped with a limited energy-source (e.g., a small battery), the *number of messages* transmitted by a distributed alarming protocol should be minimized. As a second objective, the algorithm should have a *small latency*: If there is a disaster, it is of prime importance that the alarm is raised as soon as possible.

This paper investigates protocols for distributed disaster detection and alarming. We speak of a *disaster* when more than a given number of nodes is involved, and assume that the more nodes sensing an event the more severe the potential damage. For example, in a sensor network application, an alarm should be raised when more than a given number of sensor nodes detects a certain event, and the alarm message should include the *magnitude* of the disaster.

Apart from wireless systems, the disclosure of disasters is important in wired systems as well, for instance, to respond fast to worm propagations through the

J. Gudmundsson (Ed.): SWAT 2008, LNCS 5124, pp. 246–257, 2008.

Internet and trigger appropriate defense mechanisms when many machines show signs of infection. Disasters with a large impact do not necessarily have to be globally distributed, but are often *local* in nature. For example, a bush-fire, or the emission of toxic chemicals, or even a computer virus, may mostly impact a certain region of the world.

In this paper we tackle the disclosure of such disasters from a viewpoint of *distributed computing*. Our goal is to minimize the communication overhead for computing the disaster's dimension, and the time until detection. Concretely, we consider a network $G = (V, E)$ of n sensor nodes. There may be several events going on simultaneously in the network. However, although our algorithms allow to detect them individually, for ease of presentation we will assume here that there is just *one event* which affects an arbitrary set of nodes $V' \subseteq V$.

When a node senses an event (*event-node*), it seeks to find out how many of the nodes in its vicinity sensed it as well; more concretely: a node aims at aggregating information about the connected component of event-nodes it is in, e.g., at computing the component's size. If the component's size exceeds a certain threshold, at least one node of the component should raise a disaster alarm and report the component's magnitude. In this paper, we assess the quality of a distributed algorithm using the classic quality measures *time and message complexity*, that is, the running time of the algorithm, and the total number of messages transmitted.

There are two major algorithmic challenges. The first challenge we call the *neighborhood problem*: After a node has sensed an event, it has no clue which of its neighbors (if any) are also event-nodes. Distributed algorithms where event-nodes simply ask all their neighbors already leads to a costly solution: If G is the star graph S_n and the star's center node is the only node in V', the message complexity is $\Theta(n)$ while the size of the disaster component is one. Observe that the simple trick to let nodes only ask the neighbors of higher degree does not work either: While it would clearly be a solution for the star graph, it already fails for dense graphs such as the clique graph K_n. Indeed, it may at first sight seem that $\Theta(n)$ is a lower bound for *any* algorithm for K_n, as an event-node has no information about its neighbors! We will show, however, that this (naive) intuition is incorrect.

The second challenge concerns the coordination of the nodes during the exploration of the component. In a distributed algorithm where all nodes start exploring the component independently at the same time, a lot of redundant information is collected, resulting in a too high message complexity. As a lower bound, we know that the time required to compute the disaster component's size is at least linear in the component's diameter d, and the number of messages needed by any distributed algorithm is linear in the component's size s. We are hence striving for distributed algorithms which are output-sensitive and thus *competitive* to these lower bounds.

2 Model

We consider arbitrary undirected graphs $G = (V, E)$ where the nodes V have unique identifiers. We assume that an arbitrary subset of nodes $V' \subseteq V$ senses

an event. The nodes V' are called *event-nodes*, and the nodes $V \setminus V'$ are called *non-event nodes*. We are interested in the subgraph induced by the nodes in V', that is, in the subgraph $H = (V', E')$ with $E' := \{\{u, v\} | u, v \in V', \{u, v\} \in E\}$. The subgraph H consists of one or more connected *components* C_i. The total number of nodes in component C_i will be referred to by $size(C_i)$. When the component is clear from the context, we will simply use s for $size(C_i)$. Note that in the following, for ease of presentation, we will often assume that there is only one type of event. However, all our algorithms can also handle concurrent events of different types.

After an event has hit a subset of nodes V', at least one node in each event component C_i is required to determine $size(C_i)$. This paper studies distributed algorithms which try to minimize the message and time complexities. Thereby, we allow the algorithm designer to *preprocess* the graph, e.g., to decompose the network into clusters with desired properties, i.e., to pre-compute *network decompositions* [12] (or, more specifically, sparse neighborhood-covers) of the graph. Note, however, that in this preprocessing phase, it is not clear yet which nodes will be affected by an event, i.e., V' is unknown. Also note that this preprocessing is done *offline* and its resulting structure can be reused for all future events.

During the *runtime phase*, an arbitrary number of events will hit the nodes, and each node $v \in V'$ first has to figure out which of its neighbors also belong to V' (*neighborhood problem*). In Section 3, we will allow non-event nodes to participate in the distributed algorithm as well. We will refer to this model as the *on-duty model*. It is suited for larger sensor nodes which are attached to a constant (infinite) energy supply. For smaller (wireless) nodes which rely on a limited battery, this model may not be appropriate: Typically, in order to save energy, such nodes are in a parsimonious sleeping mode. Only an event will trigger these nodes to wake up and participate in the distributed computation. We will refer to the latter model as the *off-duty model*. It will be discussed quickly in Section 4.

This paper assumes a *synchronous* environment in the sense that events are sensed by all nodes simultaneously and that there is an upper bound (known by all nodes) on the time needed to transmit a message between two nodes. The algorithms are presented in terms of communication *rounds*.

3 The On-Duty Model

In this section, the model is investigated where the non-event nodes are also allowed to participate in the distributed computations during runtime.

3.1 A Simple Solution for the Tree

Before discussing the general problem, we quickly review a simple special graph to acquaint the reader with our problem. Concretely, we look at *undirected trees*.

Consider an event component C_i of (unknown) size in a tree. If we let all s nodes start exploring the component, the message complexity grows quickly and

the overhead is large. In contrast, the following ALG_{TREE} algorithm helps to organize the nodes in a simple preprocessing phase, such that component detection at runtime is efficient. Concretely, in the preprocessing phase, ALG_{TREE} makes the entire tree graph directed and rooted, i.e., each node (except the root) is assigned a *parent node*. See Figure 1 (*left*).

During runtime, when a node senses an event, it will *immediately* notify its parent using a dummy packet. This is necessary in order to ensure fast termination. The computation of the component's size then works by an aggregation algorithm on the tree: Leaf nodes—nodes which have not received a notification from their children—inform their parents that they are the only event-node in the corresponding subtrees. Inner nodes wait until the sizes of all their children's subtrees are known, and then propagate this result to their parent node. After $O(d)$ many rounds, the root of the component knows the exact value.

Obviously, Algorithm ALG_{TREE} is asymptotically optimal for trees both in terms of time and message complexity: The time and message complexities for exploring an event component are $O(d)$ and $O(s)$, respectively, where d is the diameter of the (event) component, and s is the component's size.

3.2 The Neighborhood Problem

The neighborhood problem is a first key challenge in distributed disaster disclosure. While for special graphs, e.g., trees, the solution can be straight-forward, the situation for general graphs is less clear. In this section, we present a *network decomposition approach* [1] for the neighborhood problem.

Broadly speaking, the idea of our decomposition is to divide the nodes into different, overlapping sets or *clusters* with corresponding *cluster heads* (e.g., the node with the largest ID in the cluster). These cluster heads provide a local coordination point, where nodes can learn which of their neighbors sensed the event as well.

Before defining our decomposition more formally, we need to introduce the following definition. Two different types of diameters of node sets are distinguished: the *weak* and the *strong* diameters.

Definition 1 (Weak and Strong Diameters). *Given a set S of nodes $S \subseteq V$ of a graph $G = (V, E)$, we call the maximum length of a path between any two nodes $v, u \in S$ the* weak diameter $diam(S) := max_{u,v \in S}(dist_G(u, v))$, *if the path is allowed to include nodes from the entire node set V. On the other hand, for the* strong diameter $Diam(S)$ *of a set S, $Diam(S) := max_{u,v \in S}(dist_S(u, v))$, paths are allowed to use nodes from S only. It thus holds that $diam(S) \leq Diam(S)$. Henceforth, when the set or cluster S is clear from the context, we will just write d and D for $diam(S)$ and $Diam(S)$, respectively.*

We can now define the notion of a (k,t)-*neighborhood cover*—a special form of a network decomposition [12]. In such a cover, each node belongs to at least one, but at most to k sets or *clusters*. The overlap of the clusters guarantees that there is at least one cluster containing the entire t-neighborhood of a node.

Definition 2 (Sparse (k,t)-neighborhood Cover). *[1] A* (k,t)-neighborhood cover *is a collection of sets (or* clusters*) of nodes $S_1, ..., S_r$ with the following properties: (1) $\forall\ v$, $\exists\ i$ such that $N_t(v) \subseteq S_i$, where $N_t(v) = \{u | dist_G(u, v) \leq t\}$, and (2) $\forall i$, $Diam(S_i) \leq O(kt)$.*

A (k,t)-neighborhood cover is said to be sparse *if each node is in at most $kn^{1/k}$ sets. Finally, we will refer to the node with the largest ID in a given set S as the* cluster head *of S. In the following, we will sometimes denote a sparse (k,t)-neighborhood cover by (k,t)-$\mathcal{NC}$.*

We will propose a solution to the neighborhood problem which—in the preprocessing phase—decomposes the network with such a neighborhood cover. Thereby, we will make use of the following result.

Theorem 1. *[1] Given a graph $G = (V, E)$, $|V| = n$, and integers k, $t \geq 1$, there is a deterministic (and distributed) algorithm which constructs a t-neighborhood cover in G where each node is in at most $O(kn^{1/k})$ clusters and the maximum cluster diameter is $O(kt)$.*

The idea for solving the neighborhood problem is to compute a ($\log n$,1)-$\mathcal{NC}$ in the preprocessing phase. At runtime, in the first round, each event-node v sends a message to all cluster heads of the clusters it belongs to. The cluster head of one of those clusters will then reply in the second round with the set of v's neighbors which are also event-nodes. This algorithm has the following properties.

Theorem 2. *The ($\log n$,1)-$\mathcal{NC}$ algorithm solves the neighborhood problem for any component in time $O(\log n)$ and requires $O(s \log n)$ many messages, where n is the total number of nodes in the network, and s is the event component's size.*

Proof. The time complexity is due to the fact that messages have to be routed to the cluster heads and back, and that—according to Theorem 1—the diameter of clusters in the ($\log n$,1)-$\mathcal{NC}$ is bounded by $O(kt) = O(\log n)$.

As for the message complexity, observe that each of the s nodes in the component sends a message to at most $O(kn^{1/k}) = O(\log n \cdot n^{1/\log n}) = O(\log n)$ cluster heads (Theorem 1). The cluster head's replies add at most a constant factor to the complexity, and hence we have $O(s \log n)$ message transmissions. □

3.3 Hierarchical Network Decomposition

In this section we propose the distributed algorithm ALG_{DC} for exploring the event components. ALG_{DC}'s running time is linear in the diameter of the component, and the message complexity is linear in the component's size (both up to polylogarithmic factors). Obviously, this is asymptotically optimal up to polylogarithmic factors, since the exploration of a graph requires at least d time and requires s messages.

ALG_{DC} makes again use of the sparse (k,t)-$\mathcal{NC}$ of Definition 2. However, instead of using just one decomposition as in the neighborhood problem, we

build a *hierarchical* structure for exponentially increasing neighborhood sizes, i.e., for $t = 1, 2, 4, 8$, etc.

The detailed preprocessing and runtime phases are now described in turn (see also Algorithm 1).

***ALG$_{DC}$* Preprocessing Phase.** In the preprocessing phase, ALG_{DC} constructs a hierarchy of sparse $(\log n,t)$-$\mathcal{NC}$s (Definition 2) for exponentially increasing neighborhood sizes, that is, the decompositions $\mathcal{D}_0 := (\log n, 1)$-$\mathcal{NC}$, $\mathcal{D}_1 := (\log n, 2)$-$\mathcal{NC}$, $\mathcal{D}_2 := (\log n, 4)$-$\mathcal{NC}$, $\mathcal{D}_3 := (\log n, 8)$-$\mathcal{NC}$, ..., $\mathcal{D}_i := (\log n, 2^i)$-$\mathcal{NC}$, ..., $\mathcal{D}_{\log \Delta} := (\log n, \Delta)$-$\mathcal{NC}$, are constructed, where Δ is the diameter of the graph G.[1] Moreover, each node computes the shortest paths to its cluster heads (e.g., using Dijkstra's single-source shortest path algorithm [4]). These paths are allowed to include nodes outside the clusters.

***ALG$_{DC}$* Runtime Phase.** At runtime, initially, all event-nodes are in the *active* state. The event-nodes then contact their cluster heads to learn about their neighbors which are also event-nodes.

ALG_{DC} then starts with decomposition $\mathcal{D}_0$, switches to the level $\mathcal{D}_1$ afterwards, then to level $\mathcal{D}_2$, and so on, until level $\mathcal{D}_{\log d}$. On a general level i, ALG_{DC} does the following: All event-nodes which are still active inform their cluster heads in the $\mathcal{D}_i$ decomposition about the parts of their component which they already know. Each cluster head h of the clusters C in $\mathcal{D}_i$ then looks at each event component it hears about and performs the checks described next: If a component K is completely contained in C, h computes K's size and informs all nodes in K about s. Thereafter, all corresponding nodes are told to change to the passive state. If, on the other hand, the component K hits the boundary of C, h determines the node v_{max} with the largest ID it sees in the component, and tests whether v_{max}'s entire 2^i-neighborhood is contained in C. If this is the case, h tells v_{max} to remain active and provides it with all the event-nodes in v_{max}'s component which h knows. If not, v_{max} does not need to be notified by this cluster head. All other nodes are told to become passive. Figure 1 (*right*) depicts the situation. This scheme is applied recursively for increasingly larger neighborhood covers.

Theorem 3. *ALG_{DC} always terminates with the correct solution.*

Proof. In the $(\log n,d)$-$\mathcal{NC}$ (the weak diameter is used as clusters may include nodes outside the disaster component), there are definitively no active event-nodes left, and ALG_{DC} terminates. It remains to prove that there will always be at least one active event-node in each component K until a cluster contains the component completely. To see this, consider the (globally) largest ID node v in K. According to Theorem 1, there is always a cluster which completely contains v's neighborhood. This cluster will instruct v to continue, unless K is covered completely. □

[1] Note that $\log \Delta$ does not have to be integer. However, in this paper, we simplify the description by omitting corresponding $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ operations.

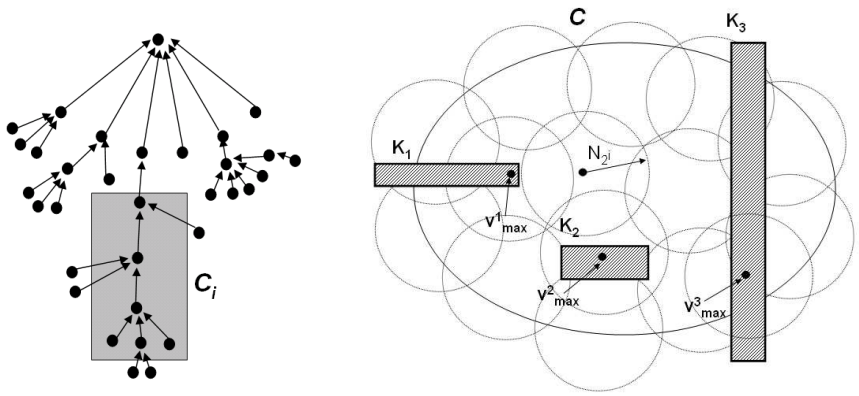

Fig. 1. *Left:* In the preprocessing phase, ALG_{TREE} makes the tree rooted and directed. Information about the event component (shaded) can then efficiently be aggregated at runtime. *Right:* Visualization for ALG_{DC}: Components K_1 and K_3 have nodes which are outside C, while K_2 is completely contained in C. The cluster head of C informs all nodes in K_2 about the component's size and deactivates them. In K_1, the N_{2^i}-neighborhood of the maximal node is completely contained, so v^1_{max} is told to remain active. In K_3, the cluster head instructs all nodes in $K_3 \cap C$ to deactivate, as v^3_{max} is too close to the boundary.

Theorem 4. *ALG_{DC} has a total running time of $O(d \log n)$, and requires at most $O(s \log d \log n)$ many messages, where n is the size of the network, s is the number of nodes in the component and d is the component's weak diameter.*

Proof. *Time complexity.* The execution of ALG_{DC} proceeds through the hierarchy levels up to level d for exponentially increasing decompositions. For each level, the active event-nodes are involved in a constant number of message exchanges with their cluster heads. On level i, according to Theorem 1, the cluster diameter is $O(2^i \log n)$, and hence the time required is $O(2^i \log n)$ as well. As $\sum_{i=0}^{\log d} 2^i = O(d)$, we have a total execution time of $O(d \cdot \log n)$.

Message complexity. Consider again the $O(\log d)$ many phases through which ALG_{DC} proceeds on the decomposition hierarchy. First, we show that the number of active nodes is at least cut in half after each phase. To see this, recall that according to ALG_{DC}, a node v with maximal identifier can only continue if it its 2^i-neighborhood is completely contained in a cluster, while the entire component v is in is not yet seen by any cluster head (e.g., component K_1 in Figure 1). This implies that for each node which remains active, at least 2^i nodes have to be passive. Consequently, the maximal number of active nodes is divided by two after each phase.

Now observe that in the first phase, all s nodes are active, sending $O(s \log n)$ many messages to their cluster heads. The cluster head's replies are asymptotically of the same order. In the second phase, the diameters of the clusters have doubled, but the number of active nodes is divided by two. Thus, again $O(s \log n)$ many messages are sent by ALG_{DC}. Generally, in phase i, the cluster's diameter

Algorithm 1 ALG_{DC}

```
1: (* Global Preprocessing *)
2: for i from 0 to log d:
3:    D_i := (log n, 2^i)-NC;
4: (* Runtime *)
5: i := 1;
6: ∀v ∈ V': v.active := true;
7: while (∃v : v.active = true)
8:    ∀ active v: notify v's cluster heads in D_i;
9:    for all clusters C
10:      let K := {K_1, ..., K_r} be C's components;
11:      ∀K ∈ K:
12:      if (K ⊆ C): output(size(K));
13:      else
14:         v_max := max{i|i ∈ (K ∩ C)};
15:         ∀v ∈ K: v.active := false;
16:         if (N_{2^i}(v_max) ⊆ C)
17:            v_max.active := true;
18:   i++;
```

is $O(2^i \log n)$, but only a fraction of $O(s/2^i)$ many nodes are active. Therefore, the message complexity is bounded by $O(\log d \cdot s \log n)$. □

While ALG_{DC} is asymptotically optimal up to polylogarithmic factors, the main term contains a factor which is a function of n. The subsequent section presents a different approach which aims at being more competitive in this respect. Moreover, ALG_{DC} needs large messages up to the size of the component; the message *sizes* of the algorithm of Section 3.4 are logarithmic in the number of nodes only.

3.4 Forests and Pointer Jumping

This section presents an alternative distributed algorithm ALG_{FOREST} for disaster detection. It is based on the *merging forests paradigm* (e.g., [8,10]), and makes use of *pointer jumping techniques* [2] in order to improve performance—both techniques are known, e.g., from *union-find data structures* [4].

***ALG*$_{FOREST}$ Preprocessing Phase.** ALG_{FOREST} solves the neighborhood problem by a sparse $(\log n,1)$-$\mathcal{NC}$. No additional decompositions are required for ALG_{FOREST}.

***ALG*$_{FOREST}$ Runtime Phase.** First, event-nodes perform a lookup operation at the cluster heads of the $(\log n, 1)$-$\mathcal{NC}$ in order to find out their neighbors which are also event-nodes. Then, each node v selects the node with the largest ID among its neighbors to become its *parent*; in case this ID is smaller than the ID of v itself, no parent is chosen. As cycles are impossible in parental relationships, the relationships define a *forest* among the event-nodes.

The idea of ALG_{FOREST} is to merge these trees efficiently to form one single tree on which all information about the component can be aggregated. However, before merging the trees, each tree is transformed to a *logical star graph*, that is, each node in the tree will learn about the tree's root (i.e., the star's center). This is achieved by the following *randomized pointer jumping technique* (cf. Algorithm 2): First, each node in the tree tosses a fair coin resulting in a bit 0 or 1 with probability 1/2 each. Parents then inform their children about their bit. Let IS

be the set of nodes consisting of all children having a 0-bit and whose parent has a 1-bit. The set IS of nodes forms a random *independent set* on the tree. The nodes in the IS will then establish a (logical) link to their parent's parent. This procedure is repeated until all nodes in the tree have a logical link to the root. Termination follows immediately from the fact that nodes arriving at the root will stop.

By this pointer jumping technique, trees become rooted stars. From now on, the roots then become the coordinators of the tree: First, they perform a *converge cast operation* [12] to learn the size of the tree. Then a root informs its children about its ID and the tree size. Subsequently, the root tells its children to determine in which trees their neighboring nodes are by performing a lookup in the $(\log n, 1)$-$\mathcal{NC}$. Information about the sizes and root IDs of the neighboring trees is then aggregated to the root. A tree seeks to join the largest neighboring tree, where "large" is defined with respect to the number of nodes in the tree, and in case of a tie, with respect to the roots' IDs. If a tree has no larger tree in its neighborhood, it will not send any join requests.

Basically, the rooted stars then become the virtual nodes of the new graph, where the corresponding roots are their coordinators, and the pointer jumping and merging techniques are applied recursively (cf. Algorithm 3; for simplicity, although the algorithm is of course distributed as described in the text, it is here presented in *global* pseudo-code). The algorithm terminates when stars do not have any neighboring stars anymore. Moreover, note that the phases of the trees need not to be synchronized, that is, some trees can be performing pointer jumping operations while other trees are in a converge cast phase.

Algorithm 2 describes the pointer jumping sub-routine for a tree T.

Algorithm 2 ALG_{PJ}

```
1: while (∃v s.t. v.parent ≠ root)
2:   ∀v ∈ T:
3:     with prob = 1/2 v.bit := 0, else v.bit := 1;
4:   ∀v ∈ T:
5:     if (v.bit = 0 ∧ v.parent.bit = 1)
6:       IS := IS ∪ {v};
7:   ∀v ∈ IS:
8:     v.parent = v.parent.parent;
```

Algorithm 3 ALG_{FOREST}

```
1: ∀v ∈ V: define v.parent;
2: let 𝒯 := {T_1, ..., T_f} be set of resulting trees;
3: while (|𝒯| > 1) do
4:   ∀T ∈ 𝒯: ALG_PJ(T);
5:   ∀T ∈ 𝒯:
6:     T_m := max{X|X ∈ 𝒯, adjacent(X, T)};
7:     if (T < T_m): merge T ▷ T_m;
8:   update 𝒯: set of resulting trees;
```

Lemma 1. *Let T be a tree, let h be its height, s its size, and d the weak diameter of the underlying graph. Applying ALG_{PJ} to T requires expected time $O(d \log h)$, and $O(sd \log h)$ many messages on average.*

Proof. *Time Complexity.* Consider an arbitrary node v, and consider its path to the root. In each round, the length of this path is reduced by a factor 3/4 in expectation. From this it follows that $O(\log h)$ many iterations are enough to find the root. Moreover, as the virtual links span at most d hops in the underlying graph, the claim follows.

Message Complexity. The message complexity follows immediately from the time complexity, as there are at most $O(d \cdot \log h)$ many rounds and at most s nodes. □

From the description of ALG_{FOREST} it follows that there will never be cycles in the pointer structure, and that all trees of a given component will eventually merge. In the following, the algorithm's performance is analyzed in detail.

Theorem 5. *ALG_{FOREST} has an expected total running time of $O(d \log s + \log s \log n)$, and requires at most $O(s \log s(d+\log n))$ many messages on average, where n is the network's size, s is the component's size, and d is the component's weak diameter.*

Proof. *Time Complexity.* The time to solve the neighborhood problem using the network decomposition is of course again $O(\log n)$.

By the description of ALG_{FOREST} it follows that a tree always joins a neighboring tree which is of the same size or larger. By a simple induction argument it can be seen that in phase i, the size of the minimal tree is at least 2^i: For $i = 0$, all trees have at least one node, and the claim follows. Now, by the induction hypothesis, assume that in phase i, indeed all trees are of size at least 2^i. Clearly, each tree will either join a neighbor, or will be joined by at least one neighbor, or both. In both cases, the new tree's size at least doubles. Consequently, ALG_{FOREST} will form a single tree after at most $\log s$ many such phases. In each phase, the tree has to be converted to a star by ALG_{PJ}, which—according to Lemma 1—requires expected time $O(d \cdot \log s)$. However, due to the exponentially growing tree sizes, a geometrically declining number of roots performing the pointer jumping operations exists, and hence the overall costs are $O(d \cdot \log s)$ as well. There are two more operations to be taken into account: First, in each phase, a constant number of aggregations or converge cast operations have to be performed in the tree, requiring time at most $O(d)$ per phase. This does not increase the execution time asymptotically. Second, according to ALG_{FOREST}, in each phase the root asks its children about the trees of their neighbors. This is done by a lookup operation in $(\log n, 1)$-$\mathcal{NC}$, which requires time $O(\log n)$ in each of the $\log s$ many phases. This gives the second summand in the formula: $O(\log s \log n)$.

Message Complexity. According to Lemma 1, the pointer jumping algorithm requires $O(s \cdot d \cdot \log d)$ many messages. Since the tree sizes at least double in each phase, the amortized amount of messages for the entire execution is $O(s \cdot d \cdot \log d)$ as well. The total cost for the $(\log n, 1)$-$\mathcal{NC}$ lookups are $O(s \log n)$ for each of the $\log s$ many phases. Finally, the aggregation costs are in $O(s \cdot d \log s)$. Since $s \geq d$, this supersedes the message cost of the pointer jumping. The claim follows. □

4 The Off-Duty Model

So far, we have assumed that both event and non-event nodes can participate in the component's exploration. While this assumption may be justified in certain systems, e.g., in wired networks, it may not be realistic for wireless networks where only the nodes which have sensed an event wake up from energy-saving mode. In the following, we will briefly discuss this *off-duty model*.

Clearly, if the number of messages does not matter, the event component can be explored in optimal time by using a simple flooding algorithm where each event-node floods the entire graph.

If we ignore the time complexity and only seek to minimize the total number of messages, the situation is different: Consider the clique K_n and assume that there are two event-nodes. Clearly, in order to find out about each other, at least $\Omega(n)$ messages need to be sent: Nodes cannot agree on local coordinators in the preprocessing phase, as these coordinators may be sleeping at runtime. On the other hand, for an optimal "offline" algorithm a constant number of messages is sufficient. Consequently, the message complexity of any distributed algorithm must be worse by a factor of at least $\Omega(n)$.

In contrast to the difficulty of the neighborhood detection, the component exploration is well understood. Depending on whether time or communication costs should be optimized, an appropriate distributed *leader election algorithm* can be applied to the resulting graph (e.g., [11] for time-optimality).

5 Related Work

Motivated by the at times tragic consequences of nature's moods, disaster disclosure is subject to a huge body of research, and it is impossible to provide a complete overview of all the proposed approaches. While many systems are based on (or complemented by) satellite techniques, e.g., for damage estimation of landslides in the *Shihmen Reservoir* in Thailand caused by heavy rainfalls, or for post-earthquake damage detection [6], there are also approaches which directly deploy sensor nodes in the region, e.g., for detecting the boundaries of a toxic leach [5]. Distributed event detection also appears in wired environments, e.g., in the defense against Internet worms [9]. Early warning systems are not only useful to react to natural catastrophes, but are also employed in international politics. Techniques to implement such indicators include expected utility models, artificial intelligence methods, or hidden Markov models [13].

This paper assumes an interesting position between local and global distributed computations, as our algorithms aim at being *as local as possible* and *as global as necessary*. While in the active field of local algorithms [12], algorithms are bound to perform their computations based only on the states of their immediate neighbors, many problems are inherently global, e.g., *leader election*. Only recently, there is a trend to look for local solutions for global problems, where the runtime depends on the *concrete problem input* [3,7], rather than considering the worst-case over all possible inputs: if in a special instance of a problem the input behaves well, a solution can be computed quickly. Similar concepts have already been studied outside the field of distributed computing, e.g., for sorting algorithms. Our paper is a new incarnation of this philosophy as performance mostly depends on the output only.

6 Conclusion

This paper has addressed the problem of distributed alarming and efficient disaster detection. We have presented first solutions for this problem by providing competitive distributed algorithms. We believe that there remain many interesting problems for future research. For instance, the question of fault-tolerance

has to be addressed: How can our algorithms be adapted for the case that nodes may be faulty, yielding disconnected components? Moreover, it would be interesting to investigate asynchronous environments. Finally, our model could also be extended to incorporate wireless aspects, such as interference.

References

1. Awerbuch, B., Berger, B., Cowen, L., Peleg, D.: Fast Network Decomposition. In: Proc. ACM PODC (1992)
2. Blelloch, G.E., Maggs, B.M.: Parallel Algorithms. In: Atallah, M.J. (ed.) Handbook of Algrorithms and Theory of Comptuation, CRC Press, Boca Raton (1998)
3. Birk, Y., Keidar, I., Liss, L., Schuster, A., Wolff, R.: Veracity Radius: Capturing the Locality of Distributed Computations. In: Proc. ACM PODC (2006)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press and McGraw-Hill (2001)
5. Dong, C.J.G., Wang, B.: Detection and Tracking of Region-Based Evolving Targets in Sensor Networks. In: Proc. 14th ICCCN (2005)
6. Eguchi, R.T., Huyck, C.K., Adams, B.J., Mansouri, B., Houhmand, B., Shinozuka, M.: Resilient Disaster Response: Using Remote Sensing Technologies for Post-Earthquake Damage Detection. In: Earthquake Engineering to Extreme Events (MCEER), Research Progress and Accomplishments 2001-2003 (2003)
7. Elkin, M.: A Faster Distributed Protocol for Constructing a Minimum Spanning Tree. In: Proc. 15th SODA (2004)
8. Gallager, R.G., Humblet, P.A., Spira, P.M.: A Distributed Algorithm for Minimum-Weight Spanning Trees. In: ACM TOPLAS (1983)
9. Kim, H.-A., Karp, B.: Autograph: Toward Automated, Distributed Worm Signature Detection. In: Proc. 13th Usenix Security Symposium (2004)
10. Lotker, Z., Patt-Shamir, B., Peleg, D.: Distributed MST for Constant Diameter Graphs. J. of Dist. Comp. (2006)
11. Peleg, D.: Time-optimal Leader Election in General Networks. J. Parallel Distrib. Comput. 8(1), 96–99 (1990)
12. Peleg, D.: Distributed Computing: A Locality-sensitive Approach. SIAM, Philadelphia (2000)
13. Schrodt, P.A.: Early Warning of Conflict in Southern Lebanon Using Hidden Markov Models. In: American Political Science Association (1997)