# Monitoring Churn in Wireless Networks

Stephan Holzer[1], Yvonne Anne Pignolet[2], Jasmin Smula[1], and Roger Wattenhofer[1]

[1] Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Switzerland
[2] IBM Research, Zurich Research Laboratory, Switzerland

**Abstract.** Wireless networks often experience a significant amount of churn, the arrival and departure of nodes. In this paper we propose a distributed algorithm for single-hop networks that detects churn and is resilient to a worst-case adversary. The nodes of the network are notified about changes quickly, in asymptotically optimal time up to an additive logarithmic overhead. We establish a trade-off between saving energy and minimizing the delay until notification for single- and multi-channel networks.

## 1 Introduction

In traditional (wired) distributed systems the *group membership problem* has been studied thoroughly (we refer to [6] for a survey). The basic premise of group membership is to know which other nodes are there, for instance to share the load of some task. Nowadays Wireless LAN or Bluetooth often replace large parts of wired networks since one does not have to build an expensive communication infrastructure first, but can communicate in "ad hoc" mode immediately. This motivates a revisit of the *group membership problem* in a wireless context: Imagine for example a bunch of wireless sensors, distributed in an area to observe that area. From time to time some of the nodes will fail, maybe because they run out of energy, maybe because they are maliciously destroyed. On the other hand, from time to time some more sensors are added. Despite this *churn* (nodes joining and leaving), all nodes should be aware of all present nodes, with small delay only. To account for the self-organizing flavor and the wireless context we decided to change the name from group membership to *self-monitoring* in this paper. We present an efficient algorithm for the self-monitoring problem in an adversarial setting.

Reducing the frequency of checking for changes, and thus the number of messages exchanged per time period, prolongs the time interval until every node is informed about changes. Since energy as well as communication channels are scarce resources for wireless devices, we evaluate a trade-off between energy and delay / runtime for single- and multi-channel networks. For single-channel networks, our algorithm can be applied to multi-hop networks using [2], which shows that algorithms designed for single-hop networks can be efficiently emulated on multi-hop networks.

## 2 Model

The network consists of a set of wireless nodes, each with a built-in unique ID. All nodes are within communication range of each other, i.e., every node can communicate with every other node directly (single-hop). New nodes may join the network at any time, and nodes can leave or crash without notice. This fluctuation of nodes in the network is called churn. We exclude Byzantine behavior and assume that as soon as a node crashes, it does not send any messages anymore. Due to the churn, the number of nodes in the network varies over time. To simplify the presentation of the algorithms and their analysis, we assume time to be divided into synchronized time slots. Messages are of bounded size, each message can only contain the equivalent of a constant number of IDs. We first assume that the number of properly divided communication channels is rather large, a requirement we drop later. In each time slot a node $v$ is in one of three operating states: `transmit` ($v$ broadcasts on channel $k$), `receive` ($v$ monitors channel $k$) or `sleep` ($v$ does not send or receive anything). A transmission is successful, if exactly one node is transmitting on channel $k$ at a time, and all nodes monitoring this channel receive the message sent. If more than one node transmits on channel $k$ at the same time, listening nodes can neither receive any message due to interference (called a *collision*) nor do they recognize any communication on the channel (this is known as *no collision detection*). The energy dissipation of $v$ is defined to be the sum of the energy for transmission and reception. Because in current embedded systems transmitting and receiving consumes several orders of magnitude more energy than sleeping or local computations, we set the energy consumption for being in state `transmit` or `receive` to unity and neglect the energy used in state `sleep` or for local computations. The nodes have sufficient memory and computational power to store an *ID table* containing all IDs of currently participating nodes and execute the provided algorithms. $n_t$ denotes the number of entries in the ID table at time $t$.

At any time, an adversary may select arbitrary nodes to crash, or it may let new nodes join the network. However, the adversary may not modify or destroy messages. Since messages have bounded size, nodes can learn at most a constant number of identifiers per message. As each node can only receive at most one message per time slot, any algorithm needs at least $c_{\min}$ time units on average (for some constant $c_{\min}$) to learn about *one* crash or join. In other words, if *on average* more than rate $r_{\max} := c_{\min}^{-1}$ nodes crash (or join) per time unit, no algorithm can handle the information (cf. [5] for the maximum tolerable average message rate in a dynamic broadcast setting). In the following, we will define an adversary and monitoring algorithm accordingly. Denote by $b$ the number of crashes/joins that happen in a maximal burst and by $\tilde{b}$ the maximal burst-size that an algorithm tolerates.

**Definition 1 ($c$-Adversary, $(c, \tilde{b})$-Adversary).** *We call an adversary a* $c$-adversary *if it lets nodes join and crash arbitrarily as long as: First, there remains at least one node knowing the ID table in the network at any time. Second, on average the number of adversarial joins/crashes is at most one node in* $c$ *time*

*slots. The adversary has full knowledge of the algorithm and can coordinate crash and join events with the aim of making the algorithm fail. A* $(c, \tilde{b})$-adversary *is a* $c$*-adversary whose churn is bounded by a constant* $\tilde{b}$ *during every period of* $c \cdot \tilde{b}$ *time slots.*

## 3 Monitoring Algorithm

The algorithm we propose is asymptotically optimal in the sense that it can survive in a setting where on average one crash or join occurs in $c$ time units, for a constant $c > c_{\min}$. We can tolerate bursty churn (a large number of nodes joining or leaving during a small time interval). Similarly to an optimal algorithm, we need time to recover from bursts since the number of newly joining (or crashed) nodes is bounded according to the message size. The algorithm can also tolerate churn while trying to recover from previous bursts; again the only limit is the learning rate of $r_{\max}$ IDs per time unit. Indeed, the adversary may crash all but one node at the same instant (killing all nodes is a special case, leading to an initialization problem, which we do not address here). Clearly, learning churn takes time, depending on the bursts. If there is a burst of $\beta$ joins or crashes, an optimal algorithm needs at least $\beta \cdot c_{\min}$ time until the corresponding information at all nodes is up-to-date. Similarly, our algorithm needs time $\beta \cdot c$. If bursts happen while recovering from previous bursts, delays will take longer due to the constant learning rate. Up to a logarithmic additive term, the learning delay of the algorithm is asymptotically optimal: the algorithm handles the maximum average rate of churn any algorithm can tolerate in this communication model.

Our algorithm is partially randomized. However, randomness is only required for detecting new nodes since this part cannot be done in a deterministic fashion. All other parts of the algorithm are deterministic, which might be of interest in a setting where only updates on crashed nodes are needed and no nodes join the network.

**Main Theorem 1** *We construct a* monitoring algorithm *that tolerates* $c$*-adversaries with maximum burst size* $b$ *with logarithmic additive overhead:* $O(b+\log n_t)$ *time slots after an event all nodes have updated the corresponding entries in their ID tables.*

*Proof.* From a family of FBMAs $\{A_{\tilde{b}}\}_{\tilde{b}\in\mathbb{N}}$ that tolerate $(c/4, \tilde{b})$-adversaries we construct a monitoring algorithm $B$. Each $A_{\tilde{b}}$ might fail if the churn is too large – since we do not know $b$ beforehand, we derive an algorithm $B$ that adapts to the bursts by searching for a good value for $\tilde{b}$ with a binary doubling search procedure. It executes algorithms $A_{\tilde{b}_i}$ from the above family with estimated values $\tilde{b}_i$ for $b$, starting with $\tilde{b}_1 := \log n_t$ (we do not start with $\tilde{b}_1 = 1$ because the running time of $A$ always exceeds $\log n_t$ due to the dissemination step). If algorithm $A$ detects its failure, we know that an algorithm $A$ tolerating $(\frac{c}{4}, \tilde{b}_i)$-adversaries is not sufficient and $B$ doubles the estimated value of $b$ to $\tilde{b}_{i+1} := 2\tilde{b}_i = 2^i \log n_t$.

Let the adversary's maximal burst be $b$. After at most $\log(b/\log n_t)+1$ repetitions, the algorithm $A$ succeeds and so does $B$. The total time needed by $B$ is at most

$$\sum_{i=1}^{\log\left(\frac{b}{\log n_t}\right)+1} \frac{c}{4}\cdot(\tilde{b}_i+\log n_t) \;<\; \sum_{i=0}^{\log\left(\frac{b}{\log n_t}\right)} 2^{i-1}c\log n_t \;\leq\; \frac{cb}{\log n_t}\log n_t \;=\; c\cdot b.$$

*Remark 1 (Adaptivity).* After a maximal burst of size $b$ happened, the above procedure always needs as much time as it needed for the big burst for all later bursts. The algorithm can be modified to update a network the quicker the smaller the current burst is by setting the estimate $\tilde{b}$ to $\tilde{b}_1$ after every successful update (proofs would need to be adjusted slightly in a few spots).

From now on, we will use the term "FBMA" as an abbreviation for "fixed burst monitoring algorithm". Let us now consider the FBMA $A_{\tilde{b}}$, ALGORITHM 1. In order to work correctly if the bursts are smaller than anticipated and to detect its failure it requires the following invariant.

**Invariant 2** *All nodes that have been in the network for $\Theta(n_t)$ time slots have the same view of the network, i.e., their* ID table *always contains the same entries. Nodes that joined more recently know their position in the* ID table.

**Algorithm 1** $A_{\tilde{b}}$ for a fixed $\tilde{b}\in\mathbb{N}$

**loop forever** // nodes have same ID table (INVARIANT 2)
1: partition nodes into sets of size $O(\min(\tilde{b}, n_t))$;
2: detect crashed nodes in each set on separate channels in parallel;
3: detect joined nodes;
4: disseminate information on crashed and joined nodes to all nodes;
5: stop if burst too large;
6: all nodes update their *ID table*;

To ensure that this invariant holds when starting the algorithm, we may assume that at time 0 there is only a single designated node active, and all other nodes still need to join. This leads to the same sorted *ID table* at all nodes.

**Theorem 3.** *If* INVARIANT 2 *holds at the start, then for all $\tilde{b}\in\mathbb{N}$,* FBMA $A_{\tilde{b}}$ *(*ALGORITHM 1*) tolerates $(c,\tilde{b})$-adversaries for a constant $c$. Furthermore each node detects if the algorithm failed $c\cdot(\tilde{b}+\log n_t)$ time slots after a stronger adversary caused a burst larger than $2\tilde{b}$. The energy consumption and the time for detection is asymptotically optimal.*

*Proof.* In brief, ALGORITHM 1 repeats a loop consisting of six steps to maintain up-to-date information in the *ID tables* of the nodes. Each step is fully distributed and does not need a central entity to control its execution. Subsequently, we call one execution of the loop of the FBMA $A_{\tilde{b}}$ a *round*.

**Step 1 – partition nodes into sets:** Nodes are divided into $N\in O(1+n_t/\tilde{b})$ sets $V:=\{S_1,\ldots,S_N\}$. Based on the information in their *ID table*, the nodes can determine which set they belong to by following a deterministic procedure. Each set appoints nodes as representatives of the set and designates their replacements in case they crash. Time $O(1)$.

**Step 2 – detect crashed nodes in each set on separate channels:** Each set $S_I \in V$ executes an algorithm to detect its crashed nodes. No communication between sets takes place. To avoid collisions each set carries out its intra-set communication on a separate channel. To find out if any of the set members in $S_I$ have crashed, each node sends a "hello" message in a designated time slot. All other nodes of the set detect who did not send a message and generate the information to disseminate: a list of so-called update items $U_I$ (details in SECTION 3.2). Time $O(\min(\tilde{b}, n_t))$.

**Step 3 – detect joined nodes:** New nodes listen to learn the tolerated burst size $\tilde{b}$ and when to try joining. They send requests to join to $S_1$ with probability $1/\tilde{b}$. In expectation at least one node can join in a constant number of rounds if the estimate $\tilde{b}$ is in $\Omega(b)$. Detected joiners are added to $U_1$ together with a note that they joined. After $O(\tilde{b} + \log n_t)$ time slots $S_1$ decides whether the estimate $\tilde{b}$ needs to be doubled due to too many joiners. Its decision is correct with high probability (whp), that is with probability greater than $1 - n_t^{-\gamma}$ for any but fixed constant $\gamma$ (details in SECTION 3.3). Time $O(\tilde{b} + \log n_t)$.

**Step 4 – disseminate information on crashed and joined nodes to all nodes:** Now every set $S_I$ has a list $U_I$ of update items containing the IDs of crashed and joined nodes in the set. To distribute this information, each set becomes a vertex of a balanced binary tree and the representative nodes communicate with the representatives of neighboring vertices in the tree according to a pre-computed schedule. If a representative crashes, there are $\tilde{b}$ replacements to take over its job. No collisions occur due to the schedule (details in SECTION 3.4). Time $O(\tilde{b} + \log n_t)$.

**Step 5 – stop if burst too large:** If the adversary is too strong, information on some of the sets is missing, or more than $\tilde{b}$ nodes crashed or tried to join. In this case, all nodes are notified and the execution of the algorithm stops (details in SECTION 3.5). Time $O(\tilde{b} + \log n_t)$.

**Step 6 – all nodes update their *ID table*:** If the algorithm did not stop, every node now has the same list $U = \bigcup_{I=1}^{N} U_I$ and can update its *ID table*. INVARIANT 2 holds. Time $O(1)$.

Newly arrived nodes do not know the *ID table* yet and have to learn the IDs of all present nodes in asymptotically optimal time, described in SECTION 3.5. However, even with incomplete *ID tables* they can participate in the algorithm, see SECTION 3.6.

While steps 1 and 6 are executed locally and hence the time complexity is constant, steps 2–5 require communication between nodes. The following sections describe the steps in more detail and examine their time complexity as well as prove that the INVARIANT 2 at the beginning of the loop holds (as long as $b$ is bounded by $\tilde{b}$ – else the algorithm will detect that it failed). We focus on $n_t \geq 2\tilde{b} + 2$, as in the case $n_t < 2\tilde{b} + 2$ the statements hold due to simple facts like "there will always remain at least one node in the network".

### 3.1 Partition Nodes into Sets (Step 1)

**Compute sets:** If $\tilde{b} \geq n_t/6 - 6$, the network forms one large set. If $\tilde{b} < n_t/6 - 6$, let $s := 2\tilde{b} + 2$ and partition the $n_t$ nodes into $N := \lceil \frac{n_t}{s} \rceil - 1$ sets $S_1, \ldots, S_N$. Each set is of size $s$, except $S_N$ which contains between $s$ and $2s - 1$ nodes. The nodes are assigned to the sets in a canonical way, based on their ID's position in the sorted *ID table* $\{id_1 < id_2 < \cdots < id_N\}$. Set $S_I$ is the set $S_I := \{id_{(I-1)\cdot s+1}, \ldots, id_{I\cdot s}\}$ for $1 \leq I \leq N - 1$ and $S_N = \{id_{(N-1)\cdot s+1}, \ldots, id_{N\cdot s}, \ldots, id_{n_t}\}$. We denote the index of $S_I$ by a capital $I$ and call it the ID of the set. Let us denote the set of all sets $\{S_1, \ldots, S_N\}$ by $V$ (since the sets will be the vertices of a communication graph in the dissemination step 4). Note that there is no ambiguity in the mapping of nodes to sets.

**Compute representatives:** In the subsequent steps, the sets will communicate with each other. To this end, representative senders and receivers are chosen to act on behalf of the set. Moreover, for each representative, the set appoints $\tilde{b}$ replacement nodes to monitor the representative and take over if it crashes. Each set $S_I$ designates two sets of nodes $R^{sender} := \{id_{(I-1)\cdot s}, \ldots, id_{(I-1)\cdot s+\tilde{b}}\}$ and $R^{receiver} := \{id_{(I-1)\cdot s+\tilde{b}+1}, \ldots, id_{I\cdot(i-1)+|S_I|-1}\}$, each consisting of $\tilde{b} + 1$ nodes. In each set we appoint the node with smallest ID to be the representative sender/receiver of $S_I$, denoted by $r_I^{sender}$, $r_I^{receiver}$. Its replacements are the other $\tilde{b}$ nodes in $R^{receiver}$ and $R^{sender}$. The $i^{th}$ replacement node of a representative (which is the node with $i^{th}$-smallest ID of the corresponding set) will take over the role of the representative in case the representative as well as the replacement nodes 1 to $i - 1$ crashed. After computing $S_{I_v}$ each $v$ can check easily if it is its set's representative sender/receiver or the $i^{th}$ replacement by looking at its position in the sorted ID table. The replacement nodes listen in all time slots whether their representative is sending or receiving messages in order to detect its failure and have the same knowledge as the representative. Thus they are able to take over the representative's role immediately. To keep things simple we often write that "$S_I$ sends an update item to $S_J$" instead of "the representative sender $r_I^{sender}$ of $S_I$ sends information on some crashed or new node to the representative receiver $r_J^{receiver}$ of $S_J$". In some cases the introduced notation of representatives is used to clarify what exactly the algorithm does. As no communication is necessary for this step, the time complexity is $O(1)$.

### 3.2 Detect Crashed Nodes in each Set in Parallel (Step 2)

Let the time slot in which the current round of the algorithm starts be $t_0$. All nodes that crash in time slot $t_0 + 1$ or later might not necessarily be detected during this execution of the loop but in the next one, i.e. at most $O(\tilde{b} + \log n_t)$ time slots later. Each set $S_I$ detects separately, which of its members crashed. Set $S_I$ uses the channel $I$ for communication among its set members to avoid collisions with other sets.

Each node $v$ is assigned a unique time slot to inform the other set members of its state (ALGORITHM 2, lines 4–5). In all other time slots, $v$ listens to the other set members to determine crashed nodes, i.e., when $v$ does not receive a

message in the time slot corresponding to a certain ID (line 6) it assumes that the node with this ID has crashed and adds it to $U_I$ (line 7).

**Theorem 4.** *When repeating* ALGORITHM 2 *continuously, crashed nodes are detected at most two rounds ($O(\tilde{b})$ time slots) if $b < \tilde{b}$.*

*Proof.* There are $O(\tilde{b})$ nodes in each set, thus each set can complete the crash detection in $O(\tilde{b})$ time slots. If there are $N$ channels available, all sets can execute this algorithm simultaneously. If a node crashes after sending its "I'm here!" message, its failure is detected the next time ALGORITHM 2 is executed.

**Corollary 1.** *The monitoring algorithm detects crashes in $O(b+\log n_t)$ if $b < \tilde{b}$.*

### 3.3 Detect Joined Nodes (Step 3)

**Algorithm 2** Crash Detection

```
1: compute index i_v of v's ID and I_v of v's set S_{I_v};
2: U_{I_v} := ∅
3: for k := 0, ..., |S_{I_v}| − 1 do
4:    if i_v == I_v · |S_{I_v}| + k then
5:       send "Im here!" on channel I_v;
6:    else if no message received on channel I_v then
7:       U_{I_v} := U_{I_v} ∪ {id_{I_v·|S_{I_v}|+k}};
```

Apart from detecting nodes that have disappeared, the network needs to be able to integrate new nodes. ALGORITHM 4 describes the behavior of nodes of the network and ALGORITHM 3 the behavior of nodes eager to join the network. Let $j \leq \tilde{b}$ be the number of such joiners. They listen on channel 1 for the representative of the corresponding set $S_1$ to announce the current number of nodes $n_t$ and the estimated $\tilde{b}$. When they have received such a message, they wait for a time slot and then try to join by sending a message with their ID with probability $p := 1/\tilde{b}$ on channel 1. If there has not been a collision, the representative sender of the set $S_1$ replies to the successful joiner with a welcome message. Otherwise each unsuccessful joiner repeats sending messages with this probability followed by listening for a reply or a stop message in the next time slot. The representative sender transmits a stop message after $d \cdot \max(\log n_t, \tilde{b})$ time slots for some constant $d$. The probability that a joiner is successful is constant if $j < \tilde{b}$ and hence the joiners attach to the network in a constant number of rounds in expectation.

**Lemma 1.** *In expectation a node attaches to the network within 4 rounds if $j < \tilde{b}$.*

*Proof.* Since $j < \tilde{b}$ the probability that a joiner is successful in a certain time slot is at least $1/\tilde{b}(1-1/\tilde{b})^{j-1} \geq \frac{1}{e\tilde{b}}$. Thus the probability that a joiner is the only sender at least once during $\tilde{b}$ time slots is greater than $1-(1-\frac{1}{e\tilde{b}})^{\tilde{b}} > 1-e^{-e^{-1}} > 0.3$. Hence the expected number of rounds until a node has joined is less than 4.

The set $S_1$ is able to detect if the current estimate for $\tilde{b}$ is in the correct order of magnitude by letting the representative sender transmit messages every second time slot reserved for the joiners until it tried $d' \log n_t$ times for some constant $d'$ to be defined later. Hence, every second opportunity for new nodes

to join is blocked $d' \log n_t$ times. The other nodes in $S_1$ count the number of times the representative sender of $S_1$ transmits successfully. If this number is less than a threshold $\tau = 2d' \log n_t \cdot e^{-2} \cdot (1 - 2/\tilde{b})$, the set decides that $\tilde{b}$ is too small for the current number of joiners and lets the other sets know about this in the next step. To this end, all nodes in $S_1$ insert an additional update item to $U_1$ which has highest priority to be forwarded to all other nodes.

Using Chernoff bounds we can show that this decision is correct w.h.p. (see the technical report of this paper [8] for a proof). This procedure only prolongs the period until nodes are detected by a constant factor.

*Remark 2.* As discussed in Section 5, there exist energy-efficient size approximation algorithms. However, letting an unknown number of nodes join cannot be solved with the help of these algorithms, since they do not handle node failures and they do not give high probability results for a small number of joining nodes.

After joining, the new nodes listen on channel 1 until the end of the current loop. In addition, they (and the old nodes in the network) have to execute the algorithm described in Section 3.6 to get to know all the nodes that are currently in the network.

**Algorithm 3** Join Algorithm

For new nodes that want to join

```
1:  while attached == false do
2:    repeat
3:      listen on channel 1;
4:    until received message "b̃ bursts"
5:    p := 1/b̃;
6:    loop
7:      send message "hello, id" on channel 1;
8:      listen on same channel;
9:      if received welcome message then
10:       attached := true;
11:     else if received "stop joining" then
12:       break;
```

**Algorithm 4** Join Detection

For nodes in the network

```
1:  count := 0;
2:  for k := 0, ..., d · max(log n_t, b̃) do
3:    if (I_v == 1 and k mod 4 == 0 and
        k < d' log n_t and i_v == r_v^sender) then
4:      send message "b̃ bursts";
5:    else if received message from r_v^sender then
6:      count := count + 1;
7:    else if message from joiner id_j received then
8:      if i_v == r_v^sender then
9:        send message "welcome";
10:     U_{I_v} := U_{I_v} ∪ {id_j};
11: if count ≥ (d' log n_t / 2e)(1/2 + 1/e) then
12:   U_{I_v} := U_{I_v} ∪ {"b̃ too small"};
13: if i_v == r_v^sender then
14:   send message "stop joining";
```

*Remark 3.* We could use more sets than $S_1$ to listen to joining nodes. As we only need to make sure that new nodes can join in a constant number of rounds and that the error probability is low, we use only the set $S_1$ for simplicity's sake.

### 3.4 Disseminate Crash/Join-Information to all Nodes (Step 4)

In the previous sections we discussed how each set $S_I$ detects crashed nodes and accepts new nodes that want to join the network. This information is stored in a (possibly empty) list $U_I$ of *update items*, where each update item consists of the ID of the node it refers to and whether the node has crashed or joined the network. This list $U_I$ needs to be distributed to all other sets. To this end, the representatives of each set communicate with representatives of other sets to compute the set $U = \bigcup_{I=1}^{N} U_I$ of all changes in the network.

**Theorem 5.** *If $b \leq \tilde{b}$ (otherwise the algorithm stops in* STEP 5*), the update items are disseminated within time $O(\tilde{b} + \log n_t)$ with* ALGORITHM 5.

*Idea:* First, the sets are mapped to vertices of a communication graph $G$ (in our case this will be a tree[3]). This can be done deterministically within each node and no messages need to be exchanged. Second, neighboring sets exchange information repeatedly until the information reaches all sets. See ALGORITHM 5 for a description in pseudo-code.

**Definition 2 (Family of communication graphs).** *Let $\mathcal{C}$ be an infinite family of communication graphs $C_N = (V_N, E_N)$ over $N$ vertices which have the property, that the in-degree and the out-degree of each vertex are bounded by $d_N$, each. Furthermore we require that each $C_N$ can be computed deterministically only from knowledge of $N$, as well as a schedule $s_N$ of length $l_N$, where $s_N : V_N \times \{1,\dots,l_N\} \longrightarrow \{1,\dots,N\} \times \{1,\dots,N\}, (v,t) \longmapsto (\kappa_{send}, \kappa_{receive})$ that tells each vertex $v \in V$ that it should send in time slot $t \in \{1,\dots,l_N\}$ on channel $\kappa_{send} \in \{1,\dots,N\}$ and receive on channel $\kappa_{receive} \in \{1,\dots,N\}$ respectively – in such a way that within $l_N$ time slots all neighbors of $G$ are able to exchange exactly one message containing one update item without collisions.*

**Definition 3 (Trees).** *Let $\mathcal{C} := \{C_N \mid N \in \mathbb{N}\}$ be the family of balanced binary trees over $N$ nodes. In $C_N := (V_N, E_N)$ we have the vertices $V_N := \{1,\dots,N\}$ and for each vertex $v \in V_N \setminus \{1\}$ there are directed edges $(v, \lfloor v/2 \rfloor)$ and $(\lfloor v/2 \rfloor, v)$ connecting $v$ to its parent $\lfloor v/2 \rfloor$.*

**Lemma 2.** *A schedule $s_N$ of length 4 can be computed deterministically for any member $C_N$ of the above tree family.*

*Proof.* Each node $v$ in odd levels of the tree (that is $\lfloor log_2(v) \rfloor$ is odd) will exchange one message (both ways) with child $2v$ in the first time slot and with child $2v+1$ in the second time slot – observe that children are in even levels. Then each node $v$ in even levels of the tree will exchange one message (both ways) with child $2v$ in the third time slot and with child $2v+1$ in the fourth time slot. Every node $u$ will send only on its own channel $u$ to avoid collisions – receivers will tune to this channel. The complete schedule is given by

$$s_N(v,1) = \begin{cases} (v, 2v) & : \lfloor \log_2(v) \rfloor \text{ is odd} \\ (v, \lfloor v/2 \rfloor) & : \lfloor \log_2(v) \rfloor \text{ is even} \end{cases} \quad s_N(v,2) = \begin{cases} (v, 2v+1) & : \lfloor \log_2(v) \rfloor \text{ is odd} \\ (v, \lfloor v/2 \rfloor) & : \lfloor \log_2(v) \rfloor \text{ is even} \end{cases}$$

$$s_N(v,3) = \begin{cases} (v, 2v) & : \lfloor \log_2(v) \rfloor \text{ is even} \\ (v, \lfloor v/2 \rfloor) & : \lfloor \log_2(v) \rfloor \text{ is odd} \end{cases} \quad s_N(v,4) = \begin{cases} (v, 2v+1) & : \lfloor \log_2(v) \rfloor \text{ is even} \\ (v, \lfloor v/2 \rfloor) & : \lfloor \log_2(v) \rfloor \text{ is odd} \end{cases}$$

If a channel (vertex) on (to) which a node $v$ should send or listen is not in the range of $\{1,\dots,N\}$, then $v$ can be sure that the corresponding node does not exist and just sleeps in this slot – this will happen for the root and the leaves.

[3] We decided to present the algorithm in this slightly more general way such that it will be easy to replace the family of communication graphs. This is useful to handle unreliable communication where information being transported from a leaf to the root is very unlikely. Using expander graphs might help in this case, since they also have logarithmic diameter and constant degree but are more robust: after a short time (say $f(n)$) the information will be copied to $2^{f(n)/O(1)}$ nodes with not too small a probability. Compared to the tree, it is more likely that at least one of the many copies of the information will reach the destination.

**Corollary 2.** *The family of trees* $\mathcal{C} := \{C_N \mid N \in \mathbb{N}\}$ *from* DEFINITION 3 *combined with the schedules* $s_N$ *from* LEMMA 2 *is a family of communication graphs, where the diameter of* $C_N$ *is* $2 \cdot \lceil \log n_t \rceil$*, the in-degree as well as the out-degree of each node are bounded by* $d_N = 3$ *and the length of any schedule* $s_N$ *is* 4.

*In the first part* of the algorithm, all nodes start with the same *ID table*, what we can assume according to INVARIANT 2. From the information $n_t$ stored in the ID table, each set $v$ of the $N$ sets computes deterministically without communication (line 1) the communication graph $G := C_N$ as well as the schedule $S_N$ of length $l_N$.

*In the second part* of the algorithm, $O(diameter(G) + \tilde{b})$ phases, each of $l_N + d_N$ time slots, are executing. During each phase each vertex is able to send one update item to each of its (at most) $d_N$ out-neighbors and receive one update item from each of its (at most) $d_N$ in-neighbors. This communication takes place by adhering to the previously computed schedule $s_N$ of length $l_N$. Thus in each phase each vertex exchanges messages with its neighbors. The vertices maintain two lists of update items. In the first list $U$ are the items the set knows of, while the second list $U'$ contains the items it has forwarded already. In the first of all phases, the first list is set to $U := U_I$, the list of the IDs determined in the detection step, and the second list $U' := \emptyset$ is empty (line 3). After the completion of the second part, $U$ equals $U'$ and contains all items. In each phase, set $S_I$ sends the information of the lowest ID in $U \setminus U'$ to its (at most) $d_N$ out-neighbors and receives (at most) $d_N$ update items from its $d_N$ in-neighbors. Depending on the outcome of each phase, the lists $U$ and $U'$ are updated.

**Algorithm 5** Deterministic Dissemination

**Sender:**

1: compute schedule $s_N$ for $G := C_N := (\underbrace{\{S_1, \ldots, S_N\}}_{\text{vertices } V_N}, \underbrace{E_N}_{\text{edges}})$;
2: $U' := \emptyset$;
3: **for** $t = 1, \ldots, diameter(G) + \tilde{b}$ **do**
4: **for** $j = 1, \ldots, l_N$ **do**
5: $item_{send} := \min_{item \in U \setminus U'}\{D\}$ or "no news" if $U$ empty;
6: send $item_{sende}$ on channel $s_N(I_v, j)_1$;
7: $U' := U' \cup \{item_{send}\}$;
8: **for** $j = l_N + 1, \ldots, l_N + d_N$ **do**
9: receive item $item_{receive}$ on channel $I_v$;
10: $U := U \cup \{item_{receive}\}$;
11: send $U$ on channel $I_v$;

**Receiver**

1: compute schedule $s_N$ for $G := C_N := (\underbrace{\{S_1, \ldots, S_N\}}_{\text{vertices } V_N}, \underbrace{E_N}_{\text{edges}})$;
2: **for** $t = 1, \ldots, diameter(G) + \tilde{b}$ **do**
3: **for** $j = 1, \ldots, l_N$ **do**
4: receive $item_j$ on channel $s_N(I_v, j)_2$;
5: **for** $j = l_N + 1, \ldots, l_N + d_N$ **do**
6: send $item_j$ on channel $I_v$ unless it is "no news";
7: $U := U \cup \{item_j\}$;

First we show that exchanging messages with neighboring vertices is possible for two representatives in each set within time $l_N + d_N$ if none of them crashes (LEMMA 3). We argue later in LEMMA 5 that we can tolerate $\tilde{b}$ crashes during the execution and in SECTION 4 we establish a time/energy/channel trade-off for fewer channels.

**Lemma 3.** *All sets transmitting update items to their (at most)* $d_N$ *out-neighbors and receiving (at most)* $d_N$ *update items from their (at most)* $d_N$ *in-neighbors*

*takes time $l_N + d_N$ when the number of channels $N$ is equal to the number of sets and no node crashes.*

*Proof.* We adhere to the schedule $s_N$. As we noted before, all nodes computed the same graph $G$ and schedule $s_N$ such that all global communication activities are consistent with the local computation of $v$. This takes $l_N$ time. Afterwards the receiver $r_I^{receiver}$ reports the newly received update items (there are at most $d_N$, one from each neighbor) to $r_I^{sender}$ on the set's channel $I$ during time slots $l_N+1, \ldots, l_N+d_N$ of this phase (lines 5–6 of the receiver's part). $r_I^{sender}$ receives this information and adjusts $U$ and $U'$ accordingly (lines 8–10 of the sender's part). All these computations happen in a deterministic way based on the same information (stored in each node) and yield the same schedule for the whole graph in each node.

Observe that no set (vertex) crashes completely as the adversary is bounded to let at most $\tilde{b}$ nodes crash during the execution of the algorithm. Hence there are $\tilde{b}$ nodes ready to replace the representatives. In LEMMA 5 we prove that repeating the procedure from LEMMA 3 $O(diameter(G) + \tilde{b})$ times will lead to full knowledge of $U$. First we prove a weak version of this lemma (LEMMA 4). We extend this lemma to hold despite crashes during execution (LEMMA 5).

**Lemma 4.** *All vertices can learn the set $U$ that contains all update items after $O(diameter(G)+\tilde{b})\cdot(l_N+d_N))$ time slots if no nodes crash during the execution of this algorithm.*

*Proof.* W.l.o.g., let $U := \{item_1, \ldots, item_{\tilde{b}}\}$ be a sorted list of update items. By induction on $i$ we prove that $item_i$ is known to all vertices $S_I$ in $G$ after $O((diameter(G) + i) \cdot (l_N + d_N))$ time slots of executing ALGORITHM 5 if no nodes crash during the execution.

**Base case $i = 1$:** Any representative $v$ that receives $item_1$, will always immediately communicate $item_1$ to its neighbors in the next phase since $item_1$ is the first item in $v$'s sorted list $U \backslash U'$. Thus item $item_1$ will have been broadcast to all nodes after $diameter(G)+1$ phases if no nodes crash during this computation.

**Inductive step $i \to i + 1$:** Let us assume the induction hypothesis for $i$. Item $item_{i+1}$ can only be delayed (in line 5 of the sender's part) by items with smaller indices. Let $item_j$ be the item with the largest index that delays $item_{i+1}$ on any of the shortest paths to any of the vertices in $G$. Then $item_{i+1}$ is known by all vertices in $G$ one phase after $item_j$. By the induction hypothesis, this is after $diameter(G) + j + 1$ phases. We remember $j \leq i$ to obtain the induction hypothesis for $i + 1$.

**Lemma 5.** LEMMA 4 *holds for up to $\tilde{b}$ nodes crashing during the execution.*

*Proof.* If a representative crashes during the dissemination step (either a sender or a receiver), a replacement node realizes the crash of its representative at most one phase later since the replacement is listening to all actions of the representatives and thus detects whether it sent all messages it was supposed to send. If it did not send a message during a phase it must have crashed and

the next replacement node steps up to be the new representative (in case no information needs to be sent by a representative in a time slot it does not matter whether it crashed). This is possible since the replacement nodes have exactly the same information as the representative and know when the representative should send what message. For the same reason they are able to know how many replacements happened before and thus when it is their turn to jump in to retransmit the necessary message in the next phase after the crash. Since at most $\tilde{b}$ nodes can crash, there will never be more than $\tilde{b}$ retransmissions necessary. This can lead to a delay of at most $\tilde{b}$ phases and the statement follows.

*Proof (Proof of* THEOREM 5*).* We combine LEMMA 4 and LEMMA 5 as well as use the fact that in the communication graphs provided by the tree family from COROLLARY 2, for all values of $N \in \mathbb{N}$ we have $diameter(C_N) = O(\log n_t), d_N = 3$ and that the schedule-length of $s_N$ is $l_N = 4$.

As a consequence, all representatives and replacements of $S_I$ know all the update items available after $O(\log n_t + \tilde{b})$ time slots. Thus all nodes in any set $S_I$ are aware of all crashed and new nodes at the time when the algorithm started (and also of some crashes/joins that happened during the algorithm's execution, but not necessarily all of those). This proves LEMMA 3.

### 3.5 Stop if Burst too Large (Step 5)

In this step, the sets determine whether the algorithm failed due to too large a burst – that is more than $\tilde{b}$ nodes joined or crashed (within time $c\tilde{b}$). To distinguish sets that do not have any information to forward from sets of which all members crashed, we let each set $S_I$ send "I'm here!" in its scheduled time slots without new information to be sent.

**Theorem 6.** *If $b > \tilde{b}$ then $O(\log n_t + \tilde{b})$ time slots after the dissemination step all nodes have the same information: Either they have noticed that the burst is too large and stopped the execution or all have the same information on network changes.*

*Proof.* Set $S_1$ knows with high probability if too many nodes tried to join and forwarded this information in the dissemination step. Thus all nodes are aware of this event at the end of step 4 of the FBMA $A_{\tilde{b}}$ if it occurs: if the decision of $S_1$ is wrong, the algorithm still works properly, it just takes longer until all nodes which to join the network are included, however, all nodes receive the same information.

If one or more sets $S_I$ completely crash before or during step 4, its neighbors immediately know that more than $\tilde{b}$ nodes crashed and the algorithm might fail (e.g. the communication graph might be disconnected and not all nodes will have been delivered the same information). The neighbors of $S_I$ then broadcast this information through the communication graph with highest priority. Even if further sets crash completely and the failure message originated by the neighbors of $S_I$ does not reach all sets, the neighbors of the other crashed sets will start

propagating such a message through the network as well. After $O(\log n_t + \tilde{b})$ phases all representatives are informed if one or more sets did not receive all the information: If no set crashed then after $\log n_t + \tilde{b}$ phases all sets have all the update items. If a set crashes before all sets have this information, then $\log n_t + \tilde{b}$ phases later all sets are informed of a failure, no matter how many sets crash now. If a set crashes afterwards, the update information has reached all sets already and thus all surviving sets can continue with this information.

The last possibility of an adversary to disturb the self-monitoring process consists in letting more than $\tilde{b}$ nodes crash even though all sets survive. By extending the dissemination phase by a constant number of time slots, we can ensure that all sets notice if more than $\tilde{b}$ update items have been disseminated and conclude that the adversary exceeds the bound of $\tilde{b}$. Therefore, also in this case a potential failure of the algorithm is known to all nodes after the dissemination step. Thus, the algorithm guarantees that all sets have the same set of update items at the end of a successful round if it did not stop the execution.

### 3.6 Participating Without Complete *ID Table*

Note that the joiners can already participate in the information-dissemination algorithm without knowing the complete ID table: When a new node $v$ is detected by the network, the node that is the oldest in the network according to the timestamp (ties are broken by ID) tells $v$ the smallest ID of a node $w$ in the network that is larger than $v$'s ID. This is possible since the oldest node is guaranteed to have a complete ID table. Joiner $v$ now assumes to have this position in the ID table. After the dissemination step has finished, node $v$ determines the number $c^<$ of crashed nodes with IDs smaller than $v$ and subtracts $c^<$ from its assumed position. Then $v$ counts $j^<$, the number of nodes that joined the network with an ID smaller than itself and adds $j^<$ to its assumed position. Thus there is only one node in the network assigned to a position in the ID table after updating the ID tables based on the information gathered in the dissemination step. Knowing this position in the ID table allows the joiner to participate in all the necessary algorithms: partition / crash and join detection / information dissemination.

In order to allow new nodes to learn the IDs of the nodes that are already in the network, the existing nodes alternately transmit their IDs and the time slot when they arrived on channel 1. This process can be interleaved with the execution of the monitoring algorithm, i.e., odd time slots can be used for the monitoring algorithm while even time slots are reserved for getting to know all existing nodes.

## 4 Energy and Trade-offs

So far, we assumed that there are as many channels available as there are sets. However, our algorithm can be modified such that it works for networks with a bounded number of $k$ channels. It can also be adapted to dissipate only a

limited amount of energy $e$. See the technical report of this paper [8] for a detailed explanation.

## 5 Related Work

Many algorithms have been designed for wireless networks under varying assumptions concerning the communication model (reception range, collision detection, transmission failures, etc.). There are many problems that are non-trivial even in single-hop networks. We focus here on networks where nodes cannot distinguish collisions from noise (no-collision detection model). The ability to detect collisions can lead to an exponential speed up, e.g., as shown in [12] for leader election. Moreover we consider the energy expenditure for transmission and listening. Basic algorithms for these networks can be used as services or building blocks for more complex algorithms and applications. Among them are initialization ($n$ nodes without IDs are assigned labels $1, \ldots, n$) [14], leader election [13,15], size approximation [3,9], alerting (all nodes are notified if an event happens at one or more nodes) [11], sorting ($n$ values distributed among $n$ nodes, the $i^{th}$ value is moved to the $i^{th}$ node) [10,17], selection problems like finding the minimum, maximum, median value [16] and computing the average value [13], and do-all (schedule $t$ similar tasks among $n$ nodes with at most $f$ failures) [4]. Note that in contrast to our work the adversary examined in these papers cannot let nodes join or crash. Moreover we cannot apply existing size approximation algorithms to estimate the number of newly joined nodes, since they do not handle node failures and they do not give high probability results for a small number of joining nodes. Our work can be seen as continuous initialization with the extension that more information is available. New nodes can join the network later and are given a label (position in the ID table). After each round of our self-monitoring algorithm, these labels are updated and in addition all nodes know which nodes have failed. Moreover, the ID table can be used to designate a leader and all nodes are aware of the current network size. In [1], a routing problem is studied in a multi-channel, single-hop, time slotted scenario and energy is considered as well. The algorithm they propose is not suitable for our application, since it requires a preprocessing phase of $O(n)$ time slots.

One of the problems underlying the monitoring problem is the dynamic broadcast problem, where an adversary can continuosly inject packets to be delivered to all participants of the network, see [5] for (im)possibility results and algorithms (nodes are assumed not to crash in this model).

The problem we solve can be viewed as a special case of the continuous gossip problem, introduced in [7] recently: an adversary can inject rumours as well as crash and restart participating nodes at any time, yet the rumours need to reach their destination before a deadline. The authors analyze the problem in a message passing model with unbounded message size and no collisions and devise an algorithm with a guaranteed per-round message complexity. Our update items can be viewed as rumours that directly depend on the crashes and restarts and the deadlines are related to the number of crashes and restarts in a time interval.

## References


1. A. Bakshi and V.K. Prasanna. Energy-Efficient Communication in Multi-Channel Single-Hop Sensor Networks. In *Conference on Parallel and Distributed Systems*, page 403. IEEE, 2004.
2. R. Bar-Yehuda, O. Goldreich, and A. Itai. Efficient Emulation of Single-Hop Radio Network with Collision Detection on Multi-Hop Radio Network with No Collision Detection. *Distributed Computing*, 5(2):67–71, 1991.
3. I. Caragiannis, C. Galdi, and C. Kaklamanis. Basic computations in wireless networks. *Lecture notes in computer science*, 3827:533, 2005.
4. B.S. Chlebus, D.R. Kowalski, and A. Lingas. Performing work in broadcast networks. *Distributed Computing*, 18(6):435–451, 2006.
5. B.S. Chlebus, D.R. Kowalski, and M.A. Rokicki. Maximum throughput of multiple access channels in adversarial environments. *Distributed Computing*, 22(2):93–116, 2009.
6. G.V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys (CSUR)*, 33(4):427–469, 2001.
7. Dariusz R. Kowalski Chryssis Georgiou, Seth Gilbert. Meeting the deadline: On the complexity of fault-tolerant continuous gossip. In *PODC*, 2010.
8. S. Holzer, Y.A. Pignolet, J. Smula, and R. Wattenhofer. Monitoring Churn in Wireless Networks. Technical report, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Switzerland, 2010. ftp://ftp.tik.ee.ethz.ch/pub/publications/TIK-Report-328.pdf.
9. J. Kabarowski, M. Kutylowski, and W. Rutkowski. Adversary immune size approximation of single-hop radio networks. *Lecture Notes in Computer Science*, 3959:148, 2006.
10. M. Kik. Merging and Merge-Sort in a Single Hop Radio Network. In *32nd conference on current trends in theory and practice of computer science (SOFSEM)*, page 341, 2006.
11. M. Klonowski, M. Kutyłowski, and J. Zatopianski. Energy Efficient Alert in Single-Hop Networks of Extremely Weak Devices. In *ALGOSENSORS*, 2009.
12. D.R. Kowalski and A. Pelc. Leader Election in Ad Hoc Radio Networks: A Keen Ear Helps. In *36th International Colloquium on Automata, Languages and Programming*, page 533, 2009.
13. M. Kutyłowski and D. Letkiewicz. Computing Average Value in Ad Hoc Networks. *Mathematical Foundations of Computer Science (MFCS2003)*, 2747:511–520, 2003.
14. M. Kutyłowski and W. Rutkowski. Adversary Immune Leader Election in Ad Hoc Radio Networks. In *European Symposium on Algorithms ESA*, pages 397–408. Citeseer, 2003.
15. C. Lavault, J.F. Marckert, and V. Ravelomanana. Quasi-optimal energy-efficient leader election algorithms in radio networks. *Information and Computation*, 205(5):679–693, 2007.
16. M. Singh and V.K. Prasanna. Optimal energy-balanced algorithm for selection in a single hop sensor network. In *IEEE Workshop on Sensor Network Protocols and Applications (SNPA)*, 2003.
17. Mitali Singh and Viktor K. Prasanna. Energy-optimal and energy-balanced sorting in a single-hop wireless sensor network. In *Pervasive Computing and Communications (PERCOM)*, 2003.