Practical Continuous Aggregation in Wireless Edge Environments *

Pedro Ákos Costa and João Leitão
NOVA LINCS & DI/FCT/NOVA University of Lisbon,
Lisboa, Portugal
pah.costa@campus.fct.unl jc.leitao@fct.unl.pt

Abstract

The edge computing paradigm brings the promise of overcoming the practical scalability limitations of cloud computing, that are a result of the high volume of data produced by Internet of Things (IoT) and other large-scale applications. The principle of edge computing is to move computations beyond the data center, closer to end-user devices where data is generated and consumed. This new paradigm creates the opportunity for edge-enabled systems and applications, that have components executing directly and cooperatively on edge devices.

Having systems' components, actively and directly, collaborating in the edge, requires some form of distributed monitoring as to adapt to variable operational conditions. Monitoring requires efficient ways to aggregate information collected from multiple devices. In particular, and considering some IoT applications, monitoring will happen among devices that communicate primarily via wireless channels. In this paper we study the practical performance of several distributed continuous aggregation protocols in the wireless ad hoc setting, and propose a novel protocol that is more precise and robust than competing alternative.

1 Introduction

Edge computing has emerged as a paradigm to overcome some of the scalability issues of cloud computing [1]. These limitations arise from the limited capability of cloud infrastructures to receive and process vast amounts of data in a timely fashion. This interest has been boosted by emerging new applications in the domain of Internet of Things (IoT) [2]. The rising popularity of these novel applications will lead to a tremendous growth in data production that will surpass the network's capacity for data producers and consumers to effectively transfer data to, and from cloud data centers [3].

In a nutshell, edge computing implies performing computations outside of the data center boundary, on devices closer to end clients of a system [4]. Therefore, edge computing can take many different forms depending on four main factors: *i*) the devices being leveraged to perform computations; *ii*) the communication medium used by those devices; *iii*) the interaction pattern of those devices with the remaining components of a system; and *iv*) the nature of the computations being performed at the edge of the system.

However, given the vast nature of the edge, we are particularly interested in a concrete edge scenario, that of multiple commodity devices being used to perform distributed computations over a given geographical area, where network infrastructure is not available. This can be the case for some applications in the domain of smart cities and smart spaces [5]. A concrete materialization of this could be, for instance, monitoring transit within a city enriched with local decisions regarding traffic signals. Alleviating areas of high traffic density in a timely fashion [6], without the need of time consuming centralized control.

The realization of this scenario would require a large number of devices with wireless capability. The absence of network infrastructure, and the non-negligible costs associated with its installation (i.e, adding access points),

*The work presented here was partially supported by the Lightkone European H2020 Project (under grant number 732505) and NOVA LINCS (through the FC&T grant UID/CEC/04516/2013).

motivates the need to have these devices create an infrastructure-less network, or ad hoc network [7]. In such a network, devices would interconnect forming a multi-hop network. This network can have routing capabilities (creating a Mesh network [8]) however, messages that transverse the network would be controlled by a routing protocol, instead of some application level protocol that is capable of modifying the contents of messages. This hinders the possibility to leverage the network's capacity to perform in-network computation. Furthermore, routing protocols have non-negligible overhead, which further reinforces the benefits of using a simple ad hoc network.

To allow these systems to gather relevant information regarding their operation, deployment and execution environment, or even application-level data, efficient and reliable distributed monitoring solutions are required. Key to the design of monitoring solutions is the capacity to aggregate information produced or managed independently by large numbers of devices using a distributed aggregation protocol [9].

In the particular case of monitoring operational aspects of distributed systems, the individual values owned by each node are not static. In fact, in many use cases the values being aggregated change over time. Hence, we need to consider a particular form of aggregation named *continuous aggregation* [10], where the value being computed by the distributed aggregation protocol is continually updated to reflect modifications in the individual input values, or changes in the system affiliation (i.e, no longer taking into consideration the value of a node that leaves the system or fails).

In this paper we study the practicality of performing distributed continuous aggregation, paving the way for novel edge computing applications that can leverage on large amounts of commodity devices that seamlessly interconnected through a wireless ad hoc network. To do this, we start by discussing the requirements of a distributed aggregation protocol for ad hoc networks (Section 2). We then survey some of the most popular distributed aggregation protocols (Section 3). Propose a novel distributed continuous aggregation protocol that we named *MiRAge* (Section 4). The practical performance of our protocol is experimentally validated and compared with solutions found in the literature on a real deployment using a fleet of 24 Raspberry Pi3 - Model B (Section 5). Finally, we conclude this paper by identifying interesting venues for future research (Section 6).

2 Aggregation in the Edge

Aggregation is an essential building block in many distributed systems [11]. A distributed aggregation protocol coordinates the execution of an aggregation task among devices of a system. In short, a distributed aggregation protocol should compute, in a distributed fashion, an aggregate function over a set of input values, where each input value is owned by a node in the system. Typical aggregate functions include *count*, *sum*, *average*, *minimum*, and *maximum*.

Not all of these aggregate functions are equivalent regarding their distributed computation. Where count, sum, and average are sensitive to input value duplication; maximum and minimum are not. Distributed aggregation protocols must take measures to deal with these aspects, ensuring that the correct aggregate result is computed.

Distributed aggregation protocols for edge systems, and in particular for settings where edge nodes interact through a multi-hop wireless ad hoc network, should be designed to address some key challenging aspects of this environment:

Continuous aggregation: As discussed in the previous section, the input values used for computing the aggregate function may change over time. Consider the example where one is computing the average CPU utilization among a collection of edge nodes. The individual usage of CPU by each node will vary accordingly to the tasks being executed by that device. To cope with this aspect, we argue in favor of using protocols that can perform *continuous aggregation*. Continuous aggregation was initially coined in [10] as being a distributed aggregation problem where periodically, every node receives a new input value for the aggregation discarding the previous one. However, this notion implies that periodically all nodes restart the protocol [12]. In practice, not all input values change at the same time, and the change of a single value should not require an effort as significant as restarting the whole aggregation process. Instead the protocol should naturally incorporate input value modifications continually and with minimal overhead.

Fully decentralized: Distributed aggregation in the context of edge computing is paramount to enable the self-management of edge computing platforms. To promote timely management decisions and overall system availability, one should favor local reconfiguration decisions with minimal coordination among nodes. This has two relevant implications in the properties of aggregation protocols. First, every node in the system should have local

access to the result being produced by the aggregation protocol. The second, is that the algorithm should not depend on the activity of specialized nodes in (or outside) the system, and operate in a fully decentralized fashion.
Fault tolerance: Distributed algorithms should be tolerant to node failures, which are unavoidable in any realistic distributed setting. Since we are considering a particular edge scenario where nodes communicate through a shared multi-hop ad hoc network, external interference is inevitable, which can be a result from having other devices in close vicinity using the wireless medium. This implies that a successful distributed aggregation protocol for this setting should be highly robust to message loss.
Minimal overhead: Considering the particular case of supporting distributed monitoring schemes, the execution of the aggregation protocol should have minimal overhead. Hence, minimizing its impact on any other services or applications being executed on edge devices. In particular, communication should have a low cost regarding the number of messages exchanged among nodes.

3 Related Work

Distributed aggregation algorithms have been widely studied in the past, particularly in the context of sensor networks [13–15] and peer-to-peer systems [16]. In the context of sensor networks, most solutions strive to propagate partial aggregate results towards a special node in the network, called *sink*, potentially performing in-network computation on the nodes alongside the route to the sink. In peer-to-peer systems aggregation protocols were mostly dedicated to counting the number of nodes in the system.

There are different classes of distributed aggregation algorithms that differ on the precision of the computed aggregate result, the supported aggregate functions, and how they deal with duplicated input values.
Sampling Techniques: Some protocols are designed to minimize the overhead by exploiting sampling techniques that compute approximate values for an aggregate function by only gathering information from a small subset of nodes. Examples of this technique include Random Tour and Sample & Collide [16] as well as Randomized Reports [17]. Unfortunately, the precision of these solutions is highly sensitive to the distribution of input values among the nodes of the system.
Specialized Data Structures: Other distributed aggregation algorithms resort to somewhat complex data structures in order to collect more information from the system than simply computing an aggregate function. For instance, Q-Digest [18] and Equi-Depth [19] can compute histograms with distributions of input values in the network by having nodes exchange collections of values among them. These solutions are much more computational demanding and the distribution computed by these solutions can have errors. Hence, inhibiting the computation of precise aggregate results from the computed distribution. Additionally, Q-Digest can only compute the distribution at a special sink node. Extrema Propagation [20], on the other hand, transforms the problem of computing an average in computing a minimum vector among all nodes in the system. Nodes iteratively exchange information, which is not sensitive to input duplicates however, the solution can only compute an approximate result.
Iterative Approaches: Many solutions rely on having nodes continually exchange information among them to compute estimates of the aggregate result, that becomes increasingly precise with the number of iterations. In Distributed Random Grouping (DRG) [15] nodes iteratively form groups with a leader. The leader gathers the current estimates of the group members, computes a new estimate from those contributions and its own estimate, and propagates the new estimate to all elements of the group.

Push-Sum [21] is a very well known protocol that operates by having each node iteratively exchange their input value and an additional parameter called *mass*. At each communication step, a node splits its local value and mass, transfers one half to a random peer, that incorporate them into its local value and mass, respectively. At any moment, a node can compute its local estimation of the aggregate result by dividing its current value by the locally stored mass, being only able to compute the average or sum/count aggregates. The correctness of the protocol depends on no mass being lost from the system, which implies that it is not robust neither to message loss nor node crashes. There are however, variants of this protocol that try to deal with this issue. LiMoSense [22] employs the same principles of Push-Sum, but stores the value and mass received, and sent to each peer, being then able to restore both in case of failure (through a compensation mechanism). Additionally in LiMoSense, nodes maintain a copy of their input value, allowing them to modify their input during the execution of the protocol, an aspect not explicitly supported by Push-Sum.

Flow-Updating [23] is another iterative approach where, contrary to Push-Sum and variants, nodes exchange and maintain flows to all their neighbors. Flows encode the difference between the local estimation of the aggregate

result of a node and that of its neighbor. These are continually updated to reflect changes in the computed local estimate. Due to the maintenance of state for each neighbor, this protocol is robust to both message loss and node crashes. Unfortunately, this protocol can only compute precise results of the average aggregate function, being unclear how to generalize this approach to other aggregate functions.

Tree Topologies: Finally, some aggregation protocols leverage on tree topologies to enable efficient aggregation, while avoiding the duplication of input values. The most well known of these solutions is the Tiny AGgregation (TAG) protocol [13], that was developed as part of TinyOs [24] to enable efficient aggregation in sensor networks using a sink node (usually not a sensor). The tree is built by having the sink node broadcast a message to the sensor nodes. These retransmit the message and set as their parent in the tree the node from whom they received the broadcast for the first time. In TAG the tree is constructed by having nodes that are connected in the tree to schedule their radios to be active in overlapping periods (which saves energy for resource constrained sensors). Aggregation happens by having nodes report to their tree parent the result of a partial aggregate with their own input value and the partial aggregate results of all their tree children. TAG also features a fault-tolerance mechanism that relies on a per node cache with previous values received from their children, that can be re-used in case of failure.

The Directed Acyclic Graph [14] protocol, further improves the fault tolerance of TAG by building a multi-path tree rooted in the sink node. This is achieved by assigning a grandparent node to every node, and leveraging on these grandparent nodes to effectively compute partial aggregate values. This allows computations to proceed even if some nodes fail during the propagation of input values and partial aggregates towards the sink node.

The Generic Aggregation Protocol [25] is another algorithm that relies on a tree topology however, contrary to TAG and DAG, the management of the tree in GAP happens naturally with the exchange of values among nodes to compute the aggregate (i.e, without needing the broadcast from a sink node). The process to build the tree is governed by an additional parameter maintained by each node called its *level*, which is initially set to an arbitrary large value. The tree is formed by an appointed root (that operates like a sink node) that has a virtual neighbor named *virtual root* with a constant level of -1. Each node maintains a set with information about all nodes with whom they exchange information (either received or sent). This set contains, for each other node, the current level of the node, its relative relation with the local node in the tree that can be either `PARENT`, `CHILD`, or `PEER`, and the last aggregate value observed from that node. Each message sent by a node contains information that enables the receiver to update this data structure and its local perception of the (current) tree topology (e.g, who is the current parent in the tree of a node). This is achieved by enforcing a set of invariants, for instance, a node will always consider as its parent in the tree, the node that has the lowest level. Additionally, if a node receives information from a node that believes to be its child, it marks the node accordingly in its local data structure. The aggregate value of a node is computed by combining the aggregate values of all its children in the tree and its own input value. Due to the decentralized nature of the tree management algorithm employed by GAP, the protocol is fault tolerant as long as the root of the tree does not fail.

All the solutions based on trees discussed above require the existence of a sink to build and manage the tree supporting the execution of the aggregation. If the sink becomes unavailable, the protocols are no longer capable of operating and computing an aggregate result. Furthermore, in these solutions only the sink node becomes aware of the aggregate result as part of the execution of the protocol. If this result is relevant to the remainder nodes of the system, it has to be broadcasted by the sink node after its computation.

4 The Design of MiRAge

In this section we discuss the design of our own distributed aggregation protocol. Our protocol is inspired in the design of GAP [25] discussed previously, but generalizes its design to remove the dependence of a single root. To this end, our protocol leverages on a self-healing spanning tree to *support* efficient continuous aggregation. In our protocol, that we named Multi Root Aggregation, or simply MiRAge, all nodes compete to build a tree rooted on themselves. This competition is controlled via the identifier of each node (a large random bit string) and a monotonic sequence number (i.e, a timestamp) controlled by the corresponding root node. Additionally, our protocol was designed to ensure that all nodes in the system are able to continually compute and update the result of the aggregate function.

In the following, we discuss the system model assumed in the design and implementation of MiRAge. We then discuss in detail the design of our algorithm. We start by explaining how the aggregate value is computed by each

node. Then we explain how the natural evolution of the protocol via the exchange of messages between nodes, allows to maintain the self-healing spanning tree that support the aggregation.

4.1 System Model

We assume a distributed system where nodes communicate via message exchange. Furthermore, we assume that devices are equipped with a WiFi radio capable of operating in ad hoc mode. Each node is pre-configured to join a single ad hoc network. We do not assume any routing algorithm or infrastructure access. Devices can transmit messages using one-hop broadcast, where the message can be received (with some probability) by all, or a subset of devices in the transmission range of the sender.

No node is aware of the total number of nodes in the system. However, we assume that each node has a unique identifier (this can be achieved by having each node generate a large random bit string at bootstrap). We do not make any assumption regarding clock synchronization, although, we assume that each node perceives the passing of time at a similar (albeit, not necessarily equal) rate.

Finally, we assume that each device runs a discovery protocol, where periodically the node transmits (in one-hop broadcast) an announce containing its own identifier. The period of this transmission is controlled via a parameter ΔD. This protocol is also used by each node as an unreliable failure detector, where if the announce of a known node is not received for a (large enough) consecutive number of transmission periods, the node becomes suspected of having failed, generating a notification to the aggregation protocol. The number of transmissions that a node can miss before suspecting another one is a parameter denoted K_{fd}. This is an assumption made by many other aggregation protocols including GAP [25], LiMoSense [22], Flow-Updating [23], among others.

4.2 Aggregation Mechanism

Algorithm 1 describes the local state maintained by each node executing MiRAge and the components of the protocol related with computing the aggregate function at each node.

The intuition of MiRAge is quite simple. Nodes organize themselves in a fault tolerant *support* spanning tree that is used to guide the aggregation mechanism. Nodes periodically propagate to their neighbors their local estimate of the aggregate result, that is obtained by applying the aggregate function over their own input value and the (latest) estimates received from neighbors with whom they share a tree link.

In MiRAge, each node owns a unique node identifier (Alg. 1 line 2) and stores its own input value for the aggregation (Alg. 1 line 3). Additionally, each node maintains its current estimate of the aggregate result (Alg. 1 line 4) and a set of known neighbors, where the following information is maintained for each neighbor: *i*) its node identifier; *ii*) the latest received estimation; *iii*) its current status in the support spanning tree, being `Active` if the neighbor shares a tree link with the local node and `Passive` otherwise; *iv*) the identifier of the tree that the neighbor is connected to; and *v*) its level in that tree (Alg. 1 line 5).

Each node also owns a set of local variables that capture its current position and configuration in the support spanning tree. This includes: the identifier of the tree to which the node is currently connected (tree identifiers are the identifier of the tree root node), its level (the root of the tree has level zero), the highest timestamp observed for the current tree, and the identifier of the parent node (Alg. 1 lines $6-9$). In addition, each node stores a map that associates tree identifiers to the highest observed timestamp for that tree (Alg. 1 line 10).

When a node is initialized (Alg. 1 line 11) it has no knowledge regarding existing neighbors. Due to this, it assumes that the result of the aggregate function is its own value, and initializes the state related with the support spanning tree to reflect a tree rooted on itself (the tree identifier being its own identifier). Additionally, the node setups a periodic function named `Beacon` that is executed every ΔT, which corresponds to the main aggregation logic of our algorithm. A typical value for ΔT is one or two seconds.

When the Beacon procedure is executed, a node will start by updating its local estimate. This is achieved through the execution of the `updateAggregation` procedure, which applies the aggregate function (operator $\oplus$ in Alg. 1 line 32) to the input value of the node with the received estimates of neighbors whose link with the local node has been marked as belonging to the node's current tree (i.e, status = `Active`).

After updating its local estimate, the node will prepare a message to be disseminated through one-hop broadcast. This message contains, for each known neighbor, a tuple containing the neighbor identifier and the locally computed aggregate value without the effects of the last contribution received from that neighbor (operator $\ominus$ in Alg. 1 line 26).

Algorithm 1: MiRAge: Aggregate Function Computation

1: **Local State:**
2: N_{id} `//Node identifier`
3: Value `//current input value`
4: Aggregation `//Current result of aggregation`
5: Neighs `//Set:` (N_{id}, value, status, T_{id}, T_{lvl})
6: T_{id} `//Unique identifier of the tree to which the node is currently attached (`N_{id} `of tree root)`
7: T_{lvl} `//Level of the node in its current tree`
8: T_{ts} `//Higher timestamp of current tree`
9: P_{id} `//Identifier of current tree parent`
10: Trees `//Map:` $Tress[T_{id}] \rightarrow$ `Timestamp`

11: **Upon Init ( ) do:**
12: Value $\longleftarrow initValue()$ `//initial input value`
13: $T_{id} \longleftarrow N_{id}$
14: $T_{lvl} \longleftarrow 0$
15: $T_{ts} \longleftarrow now()$ `//now() = current time`
16: $P_{id} \longleftarrow N_{id}$
17: Neighs $\longleftarrow \{\}$
18: Aggregation $\longleftarrow$ Value
19: **Setup Periodic Timer** `Beacon` **(**ΔT**)**

20: **Upon** `Beacon` **do:** `//every` ΔT
21: **Call** updateAggregation()
22: **if** ($T_{id} = N_{id}$) **then**
23: $T_{ts} \longleftarrow now()$
24: msg $\longleftarrow \{\}$
25: **foreach** (id, val, $stat$, tid, $tlvl$) $\in$ Neighs $\wedge T_{id} = tid$ **do**
26: msg $\longleftarrow$ msg $\cup$ (id, Aggregation $\ominus$ val)
27: **Trigger** `OneHopBCast` ($< N_{id}$, T_{id}, T_{lvl}, T_{ts}, P_{id}, Value, msg$>$)

28: **Procedure** updateAggregation()
29: Aggregation $\longleftarrow$ Value
30: **foreach** ($Neigh$, V_{neigh}, $Status$, T_{neigh}, L_{neigh}) $\in Neighs$ **do**
31: **If** ($Status =$ `Active`) **then** `//Active` $\rightarrow T_{id} = T_{neigh}$
32: Aggregation $\longleftarrow$ Aggregation $\oplus$ V_{neigh}

33: **Upon** `Receive` ($<$ id, tid, $tlvl$, tts, pid, val, msg $>$) **do:**
34: **if** $\exists(N_{id}, RecvVal) \in$ msg **then**
35: **Call** updateNeighborEntry(id, tid, $tlvl$, $RecvVal$)
36: **Call** updateTree(tid, tts, $tlvl$, pid, id, val)

37: **Procedure** updateNeighborEntry(id, tid, $tlvl$, val)
38: **if** ($\nexists$ (id', _, $stat$, _, _) $\in$ Neighs : $id' = id$) **then**
39: Neighs $\longleftarrow$ Neighs $\cup$ (id, val, `Passive`, tid, $tlvl$)
40: **else**
41: Neighs $\longleftarrow$ Neighs $\setminus$(id, _, $stat$, _, _) $\cup$ (id, val, $stat$, tid, $tlvl$)

This tuple is computed independently of the status of that neighbor. The message is then tagged with the local node identifier, and the information on the support tree that the node is currently attached to, including the tree identifier, the level of the node, the highest tree timestamp observed, and the identifier of the node's parent (Alg. 1 line 27).

Upon receiving one of these messages (Alg. 1 line 33), a node checks if there is a tuple in the message tagged with its own identifier. If so, then it uses the `updateNeighborEntry` procedure to either create or update the entry for that neighbor in its neighbor set. The information that is updated is the latest estimate received, the identifier of the tree to which the neighbor is currently attached, and its current tree level (the status of the node remains unchanged and is set to `Passive` if this was the first message received from that node).

4.3 Tree Management Mechanism

We now discuss how MiRAge manages the support spanning tree used by the aggregation strategy. As noted before, the support tree implicitly defines the data path used for propagating aggregation information. In MiRAge, there is no specialized sink node nor pre-configured root node. Instead, all nodes strive to become the root of a tree covering all nodes in the system, being a node attached to a single tree at each moment. Interestingly, the management of the support spanning tree in MiRAge does not require any additional exchange of information.

Each node has a level associated with its current position on the tree to which it is attached. We rely on the level to avoid the creation of cycles when managing the tree. The level of a node in a tree is defined as being the level of its parent plus one. The root of a tree has a level with a value of zero (and for convenience of notation, the parent of the root is defined as being the node itself). A tree is considered **correct** while its root is non-faulty, which can be verified by nodes through the observation of increasing timestamps for that tree.

When the root of a tree fails, maintaining that tree becomes impossible, since electing a new root requires synchronization among nodes, potentially impairing scalability and availability. Alternatively, in MiRAge, nodes actively compete to establish the tree rooted at themselves as the **dominating tree**. The dominating tree is the correct tree with the lowest identifier (i.e, whose root has the lowest identifier) at any given time. We further say that a tree a dominates over tree b, if the identifier of a is lower than b. Our tree stabilization mechanism allows nodes to change trees (and/or parent), as long as it ensures that the node is moving to a tree that dominates over its previous tree, or leaving a tree that is no longer correct.

Upon initialization, each node joins the tree rooted on itself (see Alg. 1 line 13). Whenever nodes exchange aggregation information, they also propagate information regarding their current tree and their position in that tree.

Whenever a node processes an aggregation message (Alg. 1 line 33), it takes advantage of this information to run a local stabilization mechanism to manage the support tree (Alg. 1 line 36). This is achieved through the procedure `updateTree`, presented in Alg. 2. In a nutshell, this procedure is responsible for ensuring three complementary goals: *i*) the correct dominating tree covers all nodes; *ii*) in presence of failures, the tree is repaired (potentially establishing a new dominating tree); and *iii*) avoid a node to connect to a tree whose root has failed (i.e, tree stability). We now explain how each of this goals is achieved in MiRAge.

Establishing Dominating Tree: When a node joins the system, it establishes itself as the root of its own tree. Since nodes exchange information regarding the tree to which they are connected, and since tree identifiers are unique, every node can compare the identifier of its current tree with the tree to which a neighbor is connected. This enables nodes to switch to a different tree if it dominates over their current tree, setting the node that sent information about the new dominating tree as their parent (Alg. 2 lines 12 and 18). This allows a single (correct) dominating tree to eventually be established among all nodes.

Additionally, and since each node only maintains a single parent, it is easy to ensure that no cycles exist in the tree. In particular, assume that a node a switches parent from node p to p'. The next message transmitted by a will report p' as being its current parent. This allows p to set the local status of a to `Passive`, effectively removing the link between these nodes from the tree (Alg. 2 line 20). Furthermore, p' will observe this message and set the status of a to `Active`, ensuring the link between them is now part of the tree (Alg. 2 line 8). This simple procedure allows a topology with no cycles and no redundant link to be established.

Tree Repair/Recovery: When a node fails, its neighbors will be eventually notified through the `NeighborDown` notification (Alg. 2 lines $30 - 35$). When a node a detects the failure of a peer p that was not the parent of a, no special measures are required except forgetting p. This is true since the tree topology, from the perspective of a, did not become compromised. However, if p was the parent of a, the tree topology has become incorrect and measures have to be taken to repair, or recover the tree.

In this case, a will attempt to locate a viable replacement for its parent (Alg. 2 lines 32 and $24 - 29$). This is done by inspecting the state of all neighbors to find a suitable candidate that must be connected to the same tree, whose link is not currently part of the tree (i.e, status = `Passive`), and whose level in the tree is strictly below the level of a, as to avoid the creation of cycles. If a valid candidate is found, a updates its parent information and its current level in the tree (this information will be propagated downwards in the tree on messages transmitted afterwards).

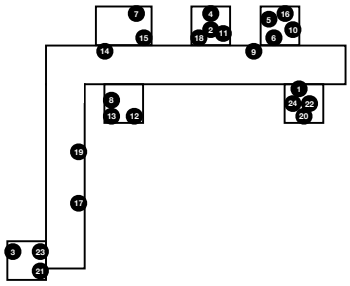
However, if no suitable candidate is found, a switches to the tree rooted on itself, triggering the process for establishing a new dominating tree as discussed above (Alg. 2 line 34). It is important to note that in the case

Algorithm 2: MiRAge: Tree Management

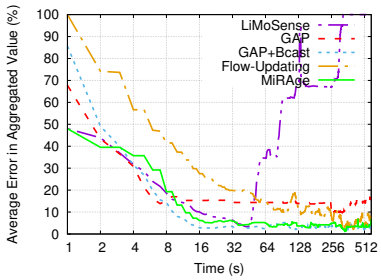
```
1: Procedure updateTree( tid, tts, tlvl, pid, id, val ) do:
2:     if ( ∄ ( id′, _, stat, _, _ ) ∈ Neighs : id′ = id ) then
3:         Neighs ⟵ Neighs ∪ ( id, val, Passive, tid, tlvl )
4:     if ( N_id = pid ) then // I am neighbor's parent
5:         if ( P_id = id ) then
6:             Call commuteToMyTree ( )
7:         else if ( T_id = tid ) then
8:             Call changeNeighborStatus ( id, Active )
9:     else if ( P_id = id ) then // Neighbor is my parent
10:        if ( T_id = tid ∨
                ( tid < N_id∧ ( Trees[tid] = ⊥ ∨ Trees[tid] < tts ) ) then
11:            Call changeNeighborStatus ( id, Active )
12:            Call updateTreeStatus ( tid, tlvl + 1, max(T_ts,tts), id)
13:        else
14:            Call commuteToMyTree ( )
15:    else // None of the above
16:        if ( tid < T_id ∧ ( Trees[tid] = ⊥ ∨ tts < Trees[tid] ) ) then
17:            Call changeNeighborStatus ( id, Active )
18:            Call updateTreeStatus ( tid, tlvl + 1, tts, id)
19:        else
20:            Call changeNeighborStatus ( id, Passive )
21:    if ( tid ≠ N_id ∧ ( Trees[tid] = ⊥ ∨ Trees[tid] < tts ) ) do
22:        Trees[tid] ⟵ tts
23:    Call checkTreeTopology ( T_lvl − 1 )

24: Procedure checkTreeTopology( lvl ) do:
25:    foreach ( id, _, stat, tid, tlvl) ∈ Neighs do
26:        if ( stat = Passive ∧ tid = T_id ∧ tlvl < lvl ) then
27:            Call changeNeighborStatus ( P_id, Passive )
28:            Call changeNeighborStatus ( id, Active )
29:            Call updateTreeStatus ( T_id, tlvl + 1, T_ts, id)

30: Upon NeighborDown (id) do:
31:    if (P_id = id) then // My parent failed
32:        Call checkTreeTopology ( T_lvl )
33:        if (P_id = id) then// No suitable parent found
34:            Call commuteToMyTree ( )
35:    Neighs ⟵ Neighs \(id, _, _, _, _ )

36: Procedure changeNeighborStatus( id, stat ) do:
37:    Neighs ⟵ Neighs \ ( id, val, _, tid, tlvl )
38:    Neighs ⟵ Neighs ∪ ( id, val, stat, tid, tlvl )

39: Procedure updateTreeStatus( tid, tlvl, tts, pid ) do:
40:    T_id ⟵ tid
41:    T_lvl ⟵ tlvl
42:    T_ts ⟵ tts
43:    P_id ⟵ pid

44: Procedure commuteToMyTree( ) do:
45:    Call updateTreeStatus ( N_id, 0, now(), N_id)
```

of the failure of the root node this will always happen, as no other node can have a level of zero in that tree. Additionally, if a node detects a direct cycle with a neighbor, both nodes will, similarly, switch to the tree rooted on themselves (Alg. 2 line 6).

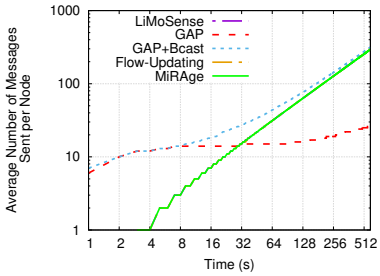
Tree Stability: When the root of the dominating tree fails, all nodes will have to converge to a new dominating tree. However, and since information does not propagate throughout the system instantaneously, it can happen that a node a discovers that the previous dominating tree is no longer correct, switches to a new tree, and afterwards receives a message from a different node that has yet to detect the failure of the previous tree. This can lead a to go back to the previous (incorrect) dominating tree. This happens because, with a high probability, the failed tree



(a) Deployment Configuration



(b) Average Error in the Aggregated Value



(c) Total number Messages sent by all nodes

Figure 1. Disperse Deployment

dominates over all other trees in the system (i.e, its identifier is lower than the identifiers of other correct trees). This can generate instability in the process of establishing the tree, compromising the convergence of nodes to a new dominating tree.

To avoid this situation, we resort to the timestamps that are associated with each tree in the system. The timestamp of a tree is only incremented by the corresponding root node (Alg. 1 lines $22-23$), and serves as a form of (multi-hop) heartbeat for that tree. Messages exchanged among nodes carry the highest timestamp observed for the sender's current tree. Hence, observing increasing timestamps for a tree, indicates that the root node is still active. To avoid the situation described above, nodes only switch to a dominating tree if it is the first time they become aware of that tree (which is relevant for the bootstrap process) or if the message being processed by the node carries a timestamp greater than all previously observed timestamps for that tree.

Optional Optimization: Optionally, and to promote trees with lower heights, nodes can run an optimization procedure to try and find a replacement for their current parent in a tree (Alg. 2 line 23). This is similar to the mechanism used to recover from faults, with the exception that the candidate must have a level lower than the current parent. This mechanism is not strictly required to ensure the correctness of the support tree.

5 Evaluation

In this Section we present an extensive experimental evaluation of MiRAge comparing its performance against state of the art solutions for continuous aggregation.

We start by noting that, contrary to the largest body of the literature regarding wireless ad hoc protocols, our experimental evaluation does not resort to simulation. Instead, we have implemented prototypes of both MiRAge and the relevant competing baselines using the C language. All protocols rely on similar code bases, use the same fault detector mechanism, and execute in similar conditions. We believe this is an essential step to demonstrate the practicality of our solution.

In the following, we discuss in more detail our experimental methodology and present our experimental results composed of two different deployments that exercise these protocols in varied conditions. These include fault-free, input value changes, and multiple node failures scenarios.

5.1 Experimental Methodology

As discussed above, we have conducted our experimental evaluation by implementing prototypes of both our protocol and relevant baselines that represent different alternatives found in the literature. All protocols were implemented in an event driven way, having a dedicated thread that handles events. These events can either be timers (for executing periodic tasks), message reception, or notifications of failures from a failure detector. This implementation strategy minimizes problems that arise from concurrency within the scope of the protocol execution. All protocols used in our evaluation resort to the same unreliable failure detector, which operates as described in Section 4.1 configured with $\Delta D = 1s$ and $K_{fd} = 10$. In practice this means that each node transmits an announcement with its own identifier every second, and that a node a is suspected to have failed by node b, when b is unable to receive an announcement from a for a period longer than 10 seconds.

The baselines employed in our experimental work are: **Flow-Updating** [23], a protocol that can compute the average function; a version of **LiMoSense** [22] published by the authors, where counters maintained by nodes are never garbage collected. LiMoSense is a representative of the well known Push-Sum protocol [21] that can compute the sum, count, and average functions, enriched to ensure fault tolerance; **GAP** [25] which is the protocol that mostly resembles our own solution, being able to compute any aggregate function but unfortunately, unable to tolerate the failure of its static tree root. Since GAP does not enable every node in the network to obtain the computed aggregate result, we also developed a simple variant of GAP, that we named **GAP+Bcast** where the root of the tree broadcasts the currently computed result. This is achieved by having the result propagated in piggyback along the tree used by GAP, enabling all nodes to learn the result of the aggregation. All protocols were configured to perform their periodic communication step every two seconds. In both GAP and GAP+Bcast experiments, the root was statically configured in runs with faults, and randomly assigned in experiments with no faults.

All experiments reported here were conducted by executing each protocol in a fleet of 24 Raspberry Pi3 - Model B. All communication among nodes is performed through a wireless ad hoc network using one-hop broadcast.

While MiRAge can easily be employed to compute any arbitrarily aggregate function, in our experiments every protocol was configured to compute the average. The initial input values of nodes were fixed, being the numbers 1 to 24 attributed statically to each of the 24 Raspberry Pis. Hence, the average value, based on the initial input values, is 12.5. Each experiment reported here was executed three times. Protocols were rotated between these executions to amortize the effects of external and uncontrollable factors. Each execution was configured to have a duration of 10 minutes (600 seconds). Results show averages of results obtained across the multiple runs.

We have conducted experiments in two different settings:

Disperse Deployment: In this deployment we have positioned the nodes across multiple rooms in two hallways of our department building. Figure 1a illustrates the distribution of nodes in the space. Each of the hallways has approximately $30m$. This is a particularly challenging environment, as there are multiple factors that affect transmissions among nodes in a very dynamic fashion. This deployment attempts to illustrate a scenario where the radio signal strength is highly variable among devices, which we have observed to trigger our fault detector multiple times.

Dense Deployment with Overlay: In this deployment we have positioned all nodes within a single room however, we have added to all tests prototypes an application-level filter that restricts, for each node, the set of nodes from which it can receive messages. Effectively, this produces a logical network that defines (potential) neighboring relationships among nodes. This overlay is represented graphically in Figure 2. We note that in this setting transmissions by any device can still produce collisions. This setting tries to illustrate a scenario where there are multiple sources of interference that can produce a somewhat higher number of collisions in the wireless medium.

5.2 Experimental Results

We now report our experimental results. In our experimental work we focus on the *Average Error in the Aggregated Value*, abbreviated *AvgErr*, that illustrates how far on average are all nodes from the correct aggregate result, being defined as:

$$AvgErr = \frac{\sum_{i=1}^{n}(|Avg_{real} - Avg_i|)}{n \times Avg_{real}} \times 100$$

where n represents the total number of nodes, Avg_{real} is the current (and correct) aggregate result considering all input values, and Avg_i represents the current value computed by node i. We present this normalized for the real aggregate result. Intuitively, in a scenario where all nodes have computed the correct average, the *AvgErr* will be 0%, which is the ideal scenario. On the other hand, when the real average value is 12.5 and the computed average value by all nodes is 25, the *AvgErr* will be 100%; a computed average value of 50 would yield an *AvgErr* of 300%. Additionally, we also measure the total number of messages transmitted by all nodes in function of time. This is a measure of the overhead produced by each protocol.

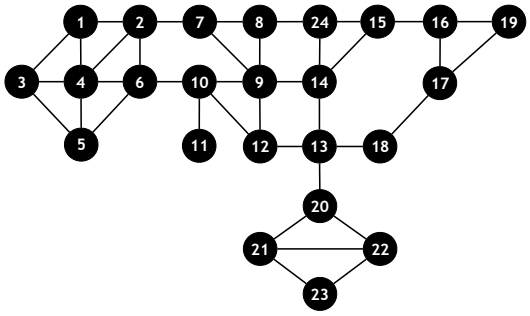


Figure 2. Deployment Configuration and Overlay Topology

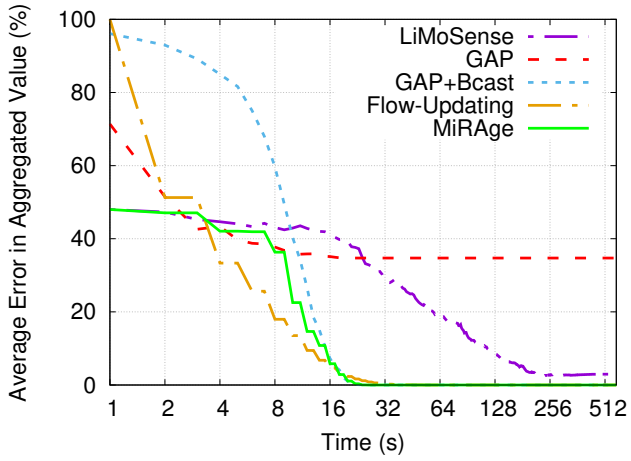
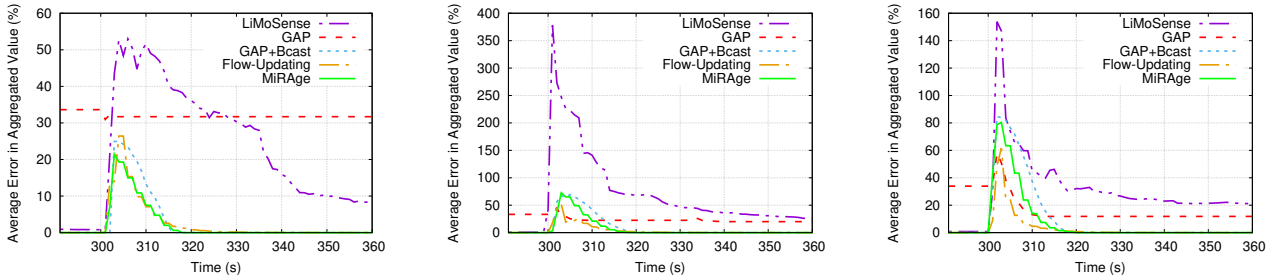


Figure 3. Average Error in the Aggregated Value in Fault-free Scenario



(a) 1 node (b) 12 nodes (c) 24 nodes

Figure 4. Average Error in Aggregated Value with Dynamic Input Values at Different Number of Nodes

5.2.1 Disperse Deployment

Figure 1 presents the schematics and results for our experiments in the disperse deployment. In this experiment we have deployed the nodes as represented in Figure 1a.

Figure 1b reports the measured *AvgErr* across all nodes as the experience progresses. Note that the x-axis is in logarithmic scale, as the main point of this plot is to show the convergence of nodes towards the correct result, a process that is more noticeable at the start of the experiment. Results show that Flow-Updating is the protocol

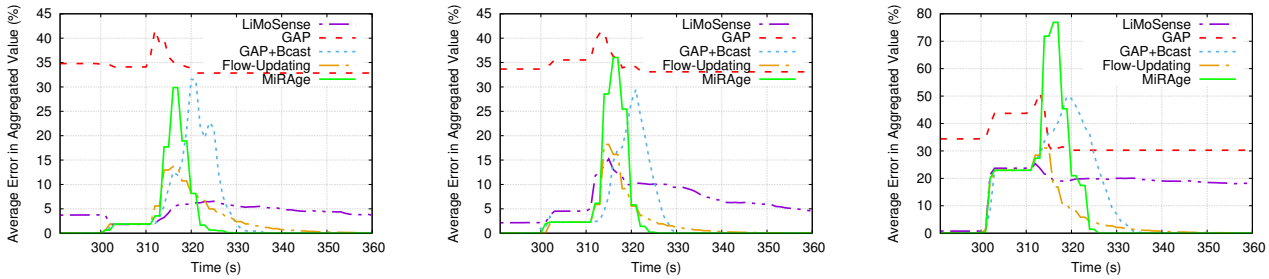


Figure 5. Average Error in Aggregated Value with Variable Number of Node Failures

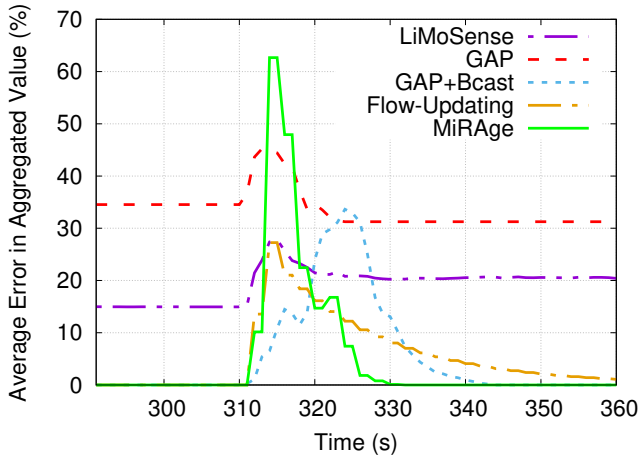


Figure 6. Average Error in Aggregated Value with 12 Link Faults

that converges more slowly towards the correct average, having an *AvgErr* in the order of 10% even after more than one minute of execution.

GAP converges relatively fast towards an *AvgErr* close to 15% and stabilizes in this value. This is expected, as GAP was not designed to provide all nodes in the system with the aggregate result. GAP+Bcast and MiRAge show the best performance, as they converge quite fast to an *AvgErr* of approximately 4%. Both algorithms are unable to compute the correct value due to the fact that the communication among many pairs of nodes is highly unstable, leading to a high level of message loss (we have confirmed this by inspecting logs). However, we note that, contrary to MiRAge, GAP+Bcast is not fault-tolerant, as the failure of the root would make it impossible for any node to compute the correct value.

Finally, and interestingly, LiMoSense appears to converge to a similar value to those computed by MiRAge and GAP+Bcast, but at some point the *AvgErr* starts to increase without the protocol being able to regain an adequate *AvgErr*. We inspected this case carefully and discovered that this happens due to an aspect in the design of LiMoSense, that tries to compensate transferred values when a node failure is detected. Unfortunately, in this environment, and due to the instability of the wireless medium, nodes detect each other as failed in an asymmetric way. This leads one of the nodes to apply the compensation mechanism while the other does not. This produces an error that propagates through the entire network, leading all nodes to compute an incorrect average of zero. While LiMoSense is indeed fault-tolerant, it was not designed to tolerate asymmetric communication links.

Figure 1c shows the total number of messages sent over time for each protocol. The results show that MiRAge, Flow-Updating, and LiMoSense have exactly the same cost. This is expected as all protocols were configured to exchange information at the same rate. GAP and GAP+Bcast issue more messages at the start of the experiment, as they send bootstrap messages to newly found nodes. GAP stops transmitting messages when values become stable, reducing GAP's communication overhead. However, this could lead to missed updates due to message loss,

and as such, GAP+Bcast was modified to cope with this issue by always transmitting messages, converging to a slightly higher cost than the remaining protocols.

In all experiments that we conducted, communication overhead always followed this pattern and hence, we omit those results from the following sections.

5.2.2 Dense Deployment with Overlay

Figure 2 reports the overlay configuration employed in this scenario. In this setting we have conducted multiple experiences. We note that while collisions in the wireless medium are highly probable, asymmetric communication and fault detection is less likely. We start by examining the behavior of protocols in a fault free environment. Then we explore the effect of three dynamic aspects: *i*) input value change; *ii*) node failure; and *iii*) link failure. In experiments where we introduce dynamic aspects, these happen in the middle of the experiment (around the 300 seconds mark).

Fault-Free Scenario Figure 3 presents the *AvgErr* in a fault free execution for all protocols.

The results show that in this setting Flow-Updating quickly starts to converge towards a perfect aggregate value. MiRAge converges somewhat slower but reaches an *AvgErr* of 0% slightly before. This happens due to the fact that MiRAge uses a deterministic tree topology to achieve convergence, whereas Flow-Updating relies on an iterative (averaging) technique that iteratively converges towards the correct value. The results of GAP are consistent with those presented above, while GAP+Bcast converges towards the correct value across all nodes, albeit, slightly slower than MiRAge. LiMoSense in this setting, where asymmetric communication is less likely, is able to converge to a good approximation of the value although, taking much more time.

Dynamic Input Values In these experiments we have introduced variations on the input value of different amounts of nodes in the system. We have conducted experiments where we modify the input value of 1, 12, and 24 nodes concurrently. Results are summarized in Figure 4 and show consistent results for all experiments. LiMoSense is the protocol that is more susceptible to input value variations, whereas MiRAge, Flow-Updating, and GAP+Bcast all present somewhat similar results, being able to converge to the new aggregate result in less than 20 seconds. The reason why only these three protocols are able to cope in a timely fashion with the change of input values is nuanced. These protocols are the only whose computation of the aggregate result directly depends on the (original) input value. This implies that as soon as the input value changes, nodes start propagating aggregation information that completely reflects the input value variation. Therefore, it suffices that messages propagate through the system to ensure that all nodes' result reflect the change.

Node and Link Failures In these experiments we introduced a variable number of faults and measured the impact on the *AvgErr*. In the first experiment we introduced a number of concurrent node crashes that vary from 1, 6, and 12 nodes around 300 seconds in the experience. In these, we have fixed the root of the trees used by GAP and GAP+Bcast to be node 1, and made sure that this node was not selected to become faulty (as these protocols would not tolerate that node's failure). Figure 5 depicts the results for these experiments. In the second experiment we introduce 12 concurrent link failures, where 12 pairs of nodes become permanently unable to communicate. This simulates the existence of obstacles or radio pollution in the environment. Figure 6 reports the obtained *AvgErr* in this scenario.

In both faulty scenarios all protocols suffer a significant increase in the *AvgErr* after the introduction of faults. With the exception of LiMoSense, all protocols can converge to the new aggregate result. LiMoSense takes significantly more time to converge because, following the Push-Sum strategy, at each communication step it only exchanges information with a single neighbor. While MiRAge is the protocol that consistently shows a higher *AvgErr* immediately after faults, it is also the protocol that converges faster. This happens due to our mechanism to manage and repair the support tree in the presence of faults, which reconfigured the tree in an expedite way. During this period however, computed results are affected by (transient) inconsistencies on the tree topology.

We argue that this sudden variance in the value is acceptable, due to the fast recovery of MiRAge. Applications can easily detect these sudden variations (i.e, spikes), and wait for values to stabilize before exposing or acting upon them.

6 Conclusion

In this paper we have studied different protocols for performing continuous aggregation in wireless ad hoc settings. Our work is motivated by the need to support emergent edge applications that delegate some computational components to commodity edge devices, that communicate through the wireless medium. In this context, it is relevant to have robust and efficient distributed aggregation protocols, both for supporting applications, and to build adequate monitoring schemes. Considering existing solutions, we propose a novel distributed continuous aggregation protocol, that we named MiRAge. Contrary to the existing state of the art, MiRAge is adequate to compute any arbitrary aggregate function, while achieving high precision in a manner that is fault tolerant to both node and link failures. Central to the design of MiRAge is a mechanism to maintain a support spanning tree without the need to have a pre-defined root or sink node, additional message exchange, or explicit coordination among nodes.

Experimental results, that were obtained by running real implementations of our protocol and competing state of the art solutions, show that our protocol can achieve faster and precise convergence, while being more robust to faults without additional communication overhead.

As future work, we plan to further improve continuous aggregation protocols by minimizing the instantaneous increase in the aggregation error in the presence of dynamic conditions (i.e, input value changes and node/link faults), and lower the communication overhead under stable conditions. We believe that this can be achieved by allowing protocols to infer additional information regarding the stability of their local neighborhood and adapt their behavior accordingly. Additionally, we also plan to tackle challenges associated with security vulnerabilities in wireless ad hoc systems.

References

[1] T. Dillon, C. Wu, and E. Chang, "Cloud computing: Issues and challenges," in *2010 24th IEEE AINA*, 4 2010, pp. 27–33.

[2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2347–2376, 10 2015.

[3] Cisco, "Cisco visual networking index: Global mobile data traffic forecast update," https://tinyurl.com/zzo6766, 2016.

[4] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE IoT Journal*, vol. 3, no. 5, pp. 637–646, 10 2016.

[5] R. Mehmood, J. Schlingensiepen, L. Akkermans, M. C. Gomes, J. Malasek, L. McCluskey, R. Meier, F. Nemtanu, A. Olaya, and M.-C. Niculescu, "Autonomic systems for personalised mobility services in smart cities," King Khalid University, et. al., Tech. Rep., 2014.

[6] W. Tärneberg, V. Chandrasekaran, and M. Humphrey, "Experiences creating a framework for smart traffic control using aws iot," in *Proc. of UCC'16*. NY, USA: ACM, 2016, pp. 63–69.

[7] Y. Yan, N. H. Tran, and F. S. Bao, "Gossiping along the path: A direction-biased routing scheme for wireless ad hoc networks," in *2015 IEEE Global Communications Conference*, 12 2015, pp. 1–6.

[8] I. F. Akyildiz and X. Wang, "A survey on wireless mesh networks," *IEEE Communications Magazine*, vol. 43, no. 9, pp. S23–S30, 9 2005.

[9] P. Jesus, C. Baquero, and P. S. Almeida, "A survey of distributed data aggregation algorithms," *IEEE Communications Surveys Tutorials*, vol. 17, no. 1, pp. 381–404, 1 2015.

[10] S. Abshoff and F. Meyer auf der Heide, "Continuous aggregation in dynamic ad-hoc networks," in *Structural Information and Communication Complexity*, M. M. Halldórsson, Ed. Cham: Springer International Publishing, 2014, pp. 194–209.

[11] R. Van Renesse, "The importance of aggregation," in *Future Directions in Distributed Computing*. Springer, 2003, pp. 87–92.

[12] O. Kennedy, C. Koch, and A. Demers, "Dynamic approaches to in-network aggregation," in *Proc. of IEEE ICDE'09*. IEEE, 2009, pp. 1331–1334.

[13] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tag: A tiny aggregation service for ad-hoc sensor networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 131–146, Dec. 2002.

[14] S. Motegi, K. Yoshihara, and H. Horiuchi, "Dag based in-network aggregation for sensor network monitoring," in *International Symposium on Applications and the Internet (SAINT'06)*, 1 2006, pp. 8 pp.–299.

[15] J.-Y. Chen, G. Pandurangan, and D. Xu, "Robust computation of aggregates in wireless sensor networks: Distributed randomized algorithms and analysis," *IEEE TPDS*, vol. 17, no. 9, pp. 987–1000, 9 2006.

[16] L. Massoulié, E. Le Merrer, A.-M. Kermarrec, and A. Ganesh, "Peer counting and sampling in overlay networks: Random walk methods," in *Proc. of PODC'06*. Denver, Colorado, USA: ACM, 2006, pp. 123–132.

[17] M. Bawa, H. Garcia-Molina, A. Gionis, and R. Motwani, "Estimating aggregates on a peer-to-peer network," Stanford InfoLab, Technical Report 2003-24, 4 2003.

[18] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri, "Medians and beyond: new aggregation techniques for sensor networks," in *Proc. of the 2nd ENSS*. ACM, 2004, pp. 239–249.

[19] M. Haridasan and R. van Renesse, "Gossip-based distribution estimation in peer-to-peer networks," in *Proc. of IPTPS'08*. Berkeley, CA, USA: USENIX Association, 2008, pp. 13–13.

[20] C. Baquero, P. S. Almeida, R. Menezes, and P. Jesus, "Extrema propagation: Fast distributed estimation of sums and network sizes," *IEEE TPDS*, vol. 23, no. 4, pp. 668–675, 4 2012.

[21] D. Kempe, A. Dobra, and J. Gehrke, "Gossip-based computation of aggregate information," in *Proc. of IEEE FCS'03*, 10 2003, pp. 482–491.

[22] I. Eyal, I. Keidar, and R. Rom, "Limosense: live monitoring in dynamic sensor networks," *Dist. Computing*, vol. 27, no. 5, pp. 313–328, 2014.

[23] P. Jesus, C. Baquero, and P. S. Almeida, "Fault-tolerant aggregation by flow updating," in *Distributed Applications and Interoperable Systems*, T. Senivongse and R. Oliveira, Eds. Springer Berlin Heidelberg, 2009, pp. 73–86.

[24] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, *TinyOS: An Operating System for Sensor Networks*. Springer Berlin Heidelberg, 2005, pp. 115–148.

[25] M. Dam and R. Stadler, "A generic protocol for network state aggregation," *self*, vol. 3, p. 411, 2005.