Decentralized Slicing in Mobile Low-Power Wireless Networks

Piotr Jaszkowski, Pawel Sienkowski, and Konrad Iwanicki
Faculty of Mathematics, Informatics, and Mechanics
University of Warsaw
Warsaw, Poland
{pj306249, ps319383}@students.mimuw.edu.pl iwanicki@mimuw.edu.pl

***Abstract*—The slicing problem is to partition a partially-ordered collection of values into a given number of totally-ordered disjoint sets—slices—so that each slice contains a predefined fraction of values that are greater than those in the previous slice and smaller than those in the next slice. In this paper, we investigate a decentralized variant of the problem, which we encountered in our experiments with wearable low-power wireless devices. In this variant of the problem, each mobile device has a local value and, by opportunistically communicating with other devices, has to autonomously assign itself to an appropriate slice, depending on how its value compares to the values of others. We propose an algorithmic framework for this setting, within which we investigate several techniques that can potentially be employed to solve the slicing problem. We then empirically study the advantages and drawbacks of such solutions in low-level simulations and on a testbed of 80 low-power wireless devices.**

***Keywords*-decentralized slicing; decentralized sorting; decentralized quantile search; decentralized counting; decentralized aggregation; mobile network; low-power wireless network; opportunistic communication; quantified crowd; gamification**

I. Introduction

Given a collection of entities, each with a value, the goal of *slicing* is to partition this collection, based on the entities' values, into predefined disjoint sets: slices [1]. A partitioning may involve any number of slices, each with an arbitrary fraction of entities, for instance, three slices containing respectively 13% of entities with the highest values, 58% with medium values, and 29% with the lowest values. In the decentralized variant of the problem, which we consider here, entities correspond to devices that perform the partitioning without any central authority: by communicating wirelessly with nearby devices, each device autonomously assigns itself to an appropriate slice, depending on how its value compares to the values of others.

The decentralized slicing problem appears in several applications of low-power wireless networks. We encountered it when developing gamification mechanisms [2] for our experiments on so-called *quantified crowds* [3], which employ wearable sensor devices to mine real-world interpersonal interactions. In essence, to improve user engagement, each participant gains points for interacting with others, and the participant's device utilizes slicing to display how well its bearer is doing, that is, whether she belongs to, for instance, the "party soul" slice, the "sociable" slice, or the "nerdy" slice. The slicing is decentralized, so that it operates even when groups of participants are disconnected from other groups and infrastructure. Other potential applications of decentralized slicing include in turn self-division of a robo-bee swarm [4] to perform tasks depending on the current state of the individual robots or assigning sensors to levels at which they are eligible to be cluster heads in an area or landmark hierarchy [5]. It is not difficult to extend this list with further application examples.

However, despite multiple potential applications, little work has been done to date on decentralized slicing in highly dynamic networks of short-range wireless resource-constrained devices, as we discuss in the next section. The past work focuses mostly on the case where the devices either are connected in a global network supporting point-to-point communication or do have limited communication range but are themselves static, which allows for employing overlay-based algorithms. In contrast, we are not aware of any prior work that would consider the decentralized slicing problem in scenarios where the devices both have a limited communication range and may move constantly.

Our goal here is thus to bridge this gap with an exploratory study of various techniques that can be used to address the slicing problem under the previous assumptions. More specifically, since the device connectivity in the considered networks can be highly dynamic, our objective is to devise slicing algorithms for such networks that not only are scalable and decentralized but also maintain no overlays. To this end, we take three approaches. First, we adapt the decentralized slicing algorithms proposed for dynamic peer-to-peer networks to the considered mobile wireless networks. Similarly, we adapt overlay-based algorithms for static wireless sensor networks to our assumptions. Finally, based on our observations, we also propose novel slicing schemes. We implement all these algorithms within a common framework, which facilitates performance comparisons. We then conduct extensive simulations of the algorithms in various networks and under different mobility patterns. We also deploy selected algorithms in a network of 80 actual wireless resource-constrained sensor devices to demonstrate that they can be used in the real world.

The rest of this paper is organized as follows. Sect. II surveys related work. Sect. III formalizes the slicing prob-

lem. Sect. IV presents the studied algorithms. Sect. V and VI analyze the algorithms empirically. Sect. VII concludes.

II. Related Work

From a broader perspective, the slicing problem is related to the sorting problem because if devices are ordered by their values, then each of them can be assigned, based on its rank in the ordering, to an appropriate slice. Parallel external sorting has received a considerable research attention, especially in the context of (distributed) databases with large volumes of data [6], [7]. The applicability of those algorithms to our settings is however limited because they typically require a stable device population and a high-throughput reliable network or shared memory. Nevertheless, some of the algorithms we study here may be viewed as decentralized sorting algorithms.

Other related problems are the selection problem and the quantile search problem [8], [9]. Their goal is to find a specific value with as few comparisons as possible: the i-th smallest among N values in selection, where N is given, and the $q \cdot N$-th smallest in quantile search, where $q \in [0,1]$ and N need not be known in advance. Kempe et al. [10] present a distributed gossip-based algorithm for the problems, but it relies on a leader gathering data and driving computations, as such being difficult to deploy in dynamic networks. Kuhn et al. [11] prove in turn that these problems are indeed more complex than classic distributed aggregation, like summing, counting, or averaging. Although the problems have a different goal than slicing, some of the ideas we explore in this paper borrow from those algorithms for selection and quantile search.

To the best of our knowledge, decentralized slicing itself was first considered by Jelasity and Kermarrec [1] in the context of peer-to-peer networks. In their algorithm, devices assign themselves random ranks from a unit interval and, by repeatedly gossiping in pairs, exchange and switch their ranks to reflect the relative order of their associated values. The algorithm may thus be viewed as decentralized bubble sort. An improved version of the algorithm was later proposed by Fernandez et al. [12]. They also devised an alternative gossip-based algorithm in which every device simply counts the number of observed devices with lower and higher values. Gramoli et al. [13] utilized the same idea but with each device maintaining a history rather than a counter of observations. Maia et al. [14], in turn, improved those algorithms with a hysteresis mechanism to limit slice switching and Bloom filters to minimize memory usage.

All those gossip-based algorithms for peer-to-peer networks rely on point-to-point routing. Such functionality, however, is difficult to provide in large multi-hop networks of highly mobile, short-range wireless devices, which we consider here. Therefore, to experiment with the techniques underlying those algorithms, we adapt them here to the considered settings.

In contrast, in wireless sensor networks, rather than gossiping between random devices, a common distributed computation scheme relies on organizing the devices into a tree-based overlay. Given the overlay, simple aggregation, such as counting or summing, can be done in the network [15]. Yet, slicing still requires gathering values from all devices at the tree root, having the root perform slice assignments, and disseminating the assignments back to the individual devices. To alleviate the need for collecting all values at the root device, Shrivastava et al. [16] introduce q-digest, a tree-based data structure for approximate quantile search and rank queries, which is based on a fixed number of buckets, thereby resembling dynamic histograms [17]. Nevertheless, also in their approach, slicing must be done at the root. Likewise, while in synopsis diffusion by Nath et al. [18] the overlay shape is relaxed, slice assignment must still be done centrally and later disseminated to the devices.

The existing algorithms for wireless sensor networks are thus poorly suited to our settings. Maintaining a tree-like overlay in highly dynamic networks is extremely difficult. Similarly, a central root device is problematic under network partitions, which are not uncommon under mobility. Consequently, to explore the ideas behind those algorithms, we have to follow approaches involving computation schemes that do not require overlays and leader election. These schemes also give rise to novel ideas, which we explore.

III. Problem Formalization

Before delving into the particular decentralized solutions to the slicing problem, let us formalize the problem itself. To this end, consider a large population of N mobile, resource-constrained, low-power wireless devices. Each device can communicate over a short-range radio with other nearby devices. The communication opportunities thus depend on the proximity of devices, and hence their mobility patterns.

Each device A has a unique *identifier*, id_A, and a local *value*, v_A. Initially, this is the only knowledge the device has. In particular, no device has any information on the size of the population, N, and the local values and identifiers of other devices. What is more, the distribution of the values over the device population may be arbitrarily skewed. Likewise, the identifiers of the devices can be distributed arbitrarily.

Assume that the device local values are (partially) ordered with relation $<$. The *rank* of device A, $r(A)$, is defined as the number of devices with smaller values, that is, the number of X for which $v_X < v_A$. For example, if we have four devices with values $v_A = 20$, $v_B = 31$, $v_C = 7$, and $v_D = 20$, then the rank of device A is $r(A) = 1$, as there is only one device, C, with the local value, $v_C = 7$, smaller than $v_A = 20$. For the same reason, $r(D) = 1$. In contrast, the rank of device C is $r(C) = 0$, as there are no devices with values smaller than $v_C = 7$. Finally, the rank of device B, $r(B) = 3$. We use $\hat{r}(A)$ to denote the *normalized rank* of device A, that is, $\hat{r}(A) = \frac{r(A)}{N} \in [0,1)$.

We define a *slice* S_l^u as the set of those devices X whose normalized ranks fall between l and u, that is, $l \leq \hat{r}(X) < u$, where $l \in [0,1)$ and $u \in (0,1]$ are respectively the slice's lower and upper boundaries. We assume that all devices know the division of $[0,1)$ into a fixed number of nonoverlapping slices.

The slicing problem requires assigning each device to an appropriate slice based on the device's normalized rank. In our decentralized version of the problem, each device performs the assignment autonomously, by opportunistically exchanging data with other devices. Importantly, due to its resource constraints, a device is allowed to use memory that is sublinear with N, preferably constant or poly-logarithmic.

IV. Algorithms

As mentioned previously, given the opportunistic and highly dynamic nature of the low-power wireless communication between mobile devices, we assume that a device is allowed to communicate only with the devices currently within its radio range, so-called *neighbors*. To send a message, it broadcasts the message to all its neighbors, with only best-effort reception guarantees. It can also unicast the message to a neighbor with a specific identifier, provided that it has learned this identifier, for instance, by having received a message broadcast by the neighbor. In other words, the techniques we consider employ neither multi-hop routing nor overlay networks, relying instead on simple one-hop communication between devices.

To enable comparisons of the explored slicing techniques, we implement them within the following algorithmic framework. The devices operate in rounds of a fixed duration, but the rounds of different devices need not be synchronized. Each device has a small local state, based on which it autonomously computes the slice it believes it belongs to. There is thus no single device that performs slice assignment for others. To gather information necessary to compute its slice, in every round a device broadcasts (and unicasts) a constant number of messages containing (parts of) its local state. By symmetry, it receives such messages from its neighbors and merges the data they carry with its local state. Such repeated state merges allow information to spread throughout the system.

We are interested in a single instance of slicing. If many slicing instances are to be done simultaneously or repeatedly, they can be managed via instance numbers and epochs [19].

We divide the presented techniques into three classes: sorting, generic counting, and domain-dependent aggregation. We present representative algorithms from each class.

A. BSort

The first considered technique is a version of decentralized bubble sort by Jelasity and Kermarrec [1] with improvements by Fernandez et al. [12], adapted to our framework. We thus dubbed it *BSort*. Like in the original algorithm, each device, A, initially draws a uniformly random number $s_A \in [0,1)$, which denotes the device's position in a sequence totally ordered with the lexicographic relation $\prec$ on $\langle v_X, id_X \rangle$ pairs. In every round, it selects another random device, B, and the two devices exchange their numbers, s_A and s_B, values, v_A and v_B, and identifiers, id_A and id_B. If $\langle v_A, id_A \rangle \prec \langle v_B, id_B \rangle$ but $s_A > s_B$, then the devices are misplaced in the total-order sequence, so they swap their sequence numbers: s_A becomes s_B and vice versa. As a result of such repeated swaps, eventually the sequence numbers of all devices reflect the total order of their $\langle v_X, id_X \rangle$ pairs. A device thus estimates its normalized rank with its sequence number, $\hat{r}(X) = s_X$, thereby assigning itself to a slice.

In the original algorithm, the peer device to exchange state with is selected conceptually from the entire device population. To this end, each device maintains a constantly-updated random partial view of the population with a special service [20]. For communication in our settings, however, the service as well as the algorithm itself would require multi-hop routing, which we have deliberately abandoned.

Therefore, our adapted version of the algorithm uses local peer sampling instead. In every round, each device, A, broadcasts a message with s_A, v_A, and id_A. Upon reception of such a message, each neighboring device, B, can add A to its local view. Since the view size is limited, B employs reservoir sampling [21] for deciding whether to add A to the view. After broadcasting its local state, B selects a device, C, from its view with which it exchanges its local state using unicast messages, as in the original algorithm. Finally, it empties its view. The algorithm also employs the improvements by Fernandez et al. [12]. In particular, for swapping sequence numbers, B selects not a random device from its view, but one for which the swap would maximize the gain, that is, the decrease in the local misplacement of the devices in the total-order sequence. Algorithm 1 presents skeleton pseudocode of *BSort*.

Algorithm 1: BSort

```
1  function init()
2  |   myRandNo ← init with a uniformly random value from [0,1)
3  |   localView ← init with an empty list
4  function onTimer()
5  |   broadcast (myValue, myId)
6  |   swapNeighbor ← pick a neighbor from localView
7  |   perform a swap of myRandNo with swapNeighbor
8  |   sliceEstimate ← floor(myRandNo · totalSlices)
9  |   localView ← reset to an empty list
10 function onReceive(msg)
11 |   combine localView with msg using reservoir sampling
```

B. ICount

The second algorithm in turn employs the counting technique by Fernandez et al. [12], again adapted to our framework through the use of local-only state exchanges. In the algorithm, each device locally stores two integer counters

(hence we call the algorithm *ICount*), initially set to zeroes: the number of observed devices with their values lesser than the device's own value and the total number of observed devices. The quotient of these two counters estimates the device's normalized rank.

In the original algorithm, each device maintains its partial view as in the previous algorithm. Periodically, it selects two peer devices from this view, to which it pushes its identifier and local value: a random peer and a peer whose normalized rank is the closest to a slice boundary. Upon reception of a similar pair from another device, it updates its counters, depending on how the pair compares to its local state.

In our settings, having a device push its pair to individual random devices would again require the abandoned multi-hop routing. Unicasting the pair to the device's selected neighbors is in turn inefficient considering that a local broadcast reaches all neighbors. Therefore, in the adapted version of the algorithm, each device periodically broadcasts its identifier and value. Based on similar broadcasts received from its neighbors, it updates its local counters, as in the original algorithm. Algorithm 2 shows the pseudocode.

Algorithm 2: ICount

```
1  function init()
2  |  totalCounter ← 1
3  |  lesserCounter ← 0
4  function onTimer()
5  |  broadcast (myId, myValue)
6  function onReceive(msg)
7  |  (id, value) ← msg
8  |  if value < myValue then
9  |  |  lesserCounter ← lesserCounter + 1
10 |  end
11 |  totalCounter ← totalCounter + 1
12 |  sliceEstimate ← lesserCounter · totalSlices/totalCounter
```

C. LCount

The next technique is a simple extension of the previous one: instead of merely counting the observed values, a device can store a list of the observed $\langle v_X, id_X \rangle$ pairs, so that the entire history can be used to compute the device's rank [13]. In effect, a single value is guaranteed to be counted at most once. Such an approach, however, cannot be directly used in our settings because the memory required for the list would scale linearly with the total device population size, N, as such being at odds with the devices' resource constraints.

Consequently, in our algorithm, dubbed *LCount*, a device does store a list of observed pairs, but only of a small size, and in addition to the two counters as in *ICount*. Given such a local state, in every round, each device broadcasts a message containing its list of observed pairs, including a pair comprising its own identifier and value. Upon reception of a similar message from a neighbor, it updates its two counters based on the pairs in the message. Moreover, it merges the received pairs with its local ones, rejecting duplicated entries, followed by random entries from either of the lists, until the size of its local list no longer exceeds the limit. Broadcasting entire lists aims to increase the pace of information discovery. Algorithm 3 presents the pseudocode.

Algorithm 3: LCount

```
1  function init()
2  |  totalCounter ← 1
3  |  lesserCounter ← 0
4  |  records ← { (myId, myValue) }
5  function onTimer()
6  |  broadcast records
7  function onReceive(msg)
8  |  foreach (id, value) ∈ msg do
9  |  |  if value < myValue then
10 |  |  |  lesserCounter ← lesserCounter + 1
11 |  |  totalCounter ← totalCounter + 1
12 |  end
13 |  records ← records ∪ msg
14 |  remove random entries from records to fit the size limit
15 |  sliceEstimate ← compute as in ICount
```

D. SCount

In the previous counting algorithms, a single value may be counted multiple times, which may lead to an invalid slice assignment. To overcome this problem, we introduce an algorithm inspired by our earlier research on crowd counting [19]. More specifically, instead of two integer counters, each device maintains two so-called *sketches* or *synopses* [18] for counting devices with values smaller and nonsmaller than the device's own value, hence the name of the algorithm: *SCount*. A sketch is a small probabilistic data structure for estimating the number of unique items in a multi-set. It supports four basic operations: `init`, `add`, `merge`, and `estimate`. `Init` is used to reset the structure so that it represents an empty multi-set. `Add` corresponds to inserting to the multi-set another element. `Merge` combines two structures as if the two multi-sets were united. `Estimate` computes an approximation of the cardinality of the multi-set corresponding to the structure.

The algorithm thus works as follows. Initially, each device resets its sketches and adds its identifier to the sketch representing devices with values greater than or equal to the device's own value. It then exchanges its sketches with other devices, merging sketches received from these devices with its own ones. Which of the two local sketches of device A are merged with the corresponding sketches received from device B depends on how A's value compares to B's value. The order of merging as well as whether two sketches are merged more than once is irrelevant because `merge` is idempotent, commutative, and associative. As a result of such repeated merges, eventually the sketches of each device contain information on all values relevant to slice assignment. A device thus assigns itself to a slice, by invoking `estimate` on the sketches and computing its normalized rank based on the results.

State exchanges between devices are done with a scheme by Gavidia and van Steen for decentralized sharing of data records in networks of short-range wireless devices [22]. In that scheme, each device maintains a limited collection of data records. In every round, it broadcasts to its neighbors a random subset of such records, including a record for itself. Likewise, it receives similar records from its neighbors and adds them to its local record collection. It then randomly selects which of the records in the collection to keep and broadcast in the next round, so that the collection size limit is preserved. The scheme thus uses record broadcasting similarly to the way *LCount* broadcasts pairs, albeit with minor differences [22].

In our algorithm, a data record corresponds to a device and comprises the device's value, identifier, and its two sketches. Moreover, apart from merely exchanging the records, as in the sharing scheme, a device merges the sketches in the received records with its own two sketches. Actually, as an optimization, this is more involved and makes use of the order- and duplicate-independence of `merge`. More specifically, for each received record, a device tries to merge the sketches from the record not only with its own sketches but also with the sketches in all records it has at that time in its collection. This aims to improve the pace of information dissemination. Algorithm 4 shows the pseudocode.

Algorithm 4: SCount

```
1  function init()
2      myLtSketch ← init(myLtSketch)
3      myGeSketch ← init(myGeSketch)
4      myGeSketch ← add(myGeSketch, myId)
5      sharedRecords ← init with a record containing
         (myId, myValue, myLtSketch, myGeSketch)
6  function onTimer()
7      records ← pick a subset of sharedRecords
8      broadcast records
9  function onReceive(msg)
10     foreach record ∈ msg do
11         (rId, rValue, rLtSketch, rGeSketch) ← record
12         if myValue > rValue then
13             myLtSketch ← merge(myLtSketch, rLtSketch)
14             myLtSketch ← add(myLtSketch, rId)
15         else if myValue < rValue then
16             myGeSketch ← merge(myGeSketch, rGeSketch)
17         else if myValue = rValue then
18             myLtSketch ← merge(myLtSketch, rLtSketch)
19             myGeSketch ← merge(myGeSketch, rGeSketch)
20         foreach record' ∈ sharedRecords do
21             update record' with record in the same way as the
                 local sketches in lines 12–19 and vice versa
22         end
23     end
24     combine sharedRecords with msg
25     sliceEstimate ← estimate(myLtSketch) · totalSlices /
         (estimate(myLtSketch)+estimate(myGeSketch))
```

E. DTree

The next algorithm is a representative of the domain-dependent aggregation techniques. It is an adaptation of the q-digest tree technique by Shrivastava et al. [16], and hence we dubbed it *DTree*. To apply this technique, we have to assume that the values stored by devices are nonnegative integers from range $[0 \ldots 2^K)$ for some fixed K. This range can be decomposed into a so-called *range tree*: a full binary tree of height K, where subsequent levels of the tree, counting from the root, correspond to subsequent bits in binary notation of the values, counting from the most significant bit (for an example, see Fig. 1). Any value, $v_X \in [0 \ldots 2^K)$ corresponds to a leaf vertex in this tree and its binary notation can be obtained by following the path to the leaf from the root of the tree. For example, in binary notation, value $v_X = 5$ is written as $101b$, and thus, in the tree for $K = 3$ from Fig. 1, it corresponds to the leaf that can be reached by following the path right-left-right. Actually, a leaf vertex for value v_X represents a singleton value range: $[v_X \ldots v_X + 1)$. An internal vertex with two children, in turn, corresponds to the range that is the union of the ranges corresponding to the child vertices. In particular, the root vertex represents the entire range $[0 \ldots 2^K)$.

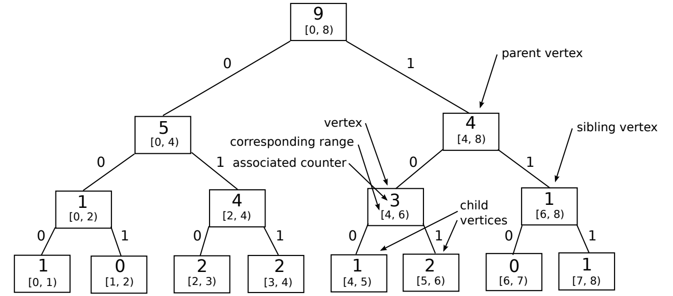


Figure 1. An example of a range tree used in the *DTree* algorithm.

Looking at the total device population, we can associate a counter with each vertex of a range tree for the domain of the devices' values. The counter for a leaf vertex representing range $[v \ldots v+1)$ denotes the total number of devices whose local values fall into range $[v \ldots v+1)$, that is, the devices, X, for which $v_X = v$. The counter for an internal vertex representing range $[v_l \ldots v_u)$ denotes in turn the total number of devices X for which $v_l \leq v_X < v_u$. The counter for the root thus equals to the total device population size, N. Given such a tree, the normalized rank of a device with value v_X can be computed by summing the counters for any disjoint ranges whose union is range $[0 \ldots v_X)$, and dividing the resulting sum by the value of the counter for the root vertex. For instance, the rank of a device with value 5 = $101b$ in the tree from Fig. 1 can be computed by dividing by the counter value of the root range $[0 \ldots 8)$ the sum of the counters for ranges $[0 \ldots 1)$, $[1 \ldots 2)$, $[2 \ldots 4)$, and $[4 \ldots 5)$, or the sum for ranges $[0 \ldots 4)$ and $[4 \ldots 5)$, to name just two examples.

Due to resource constraints, a device must not store the entire tree, as the tree scales linearly with N. Instead, the device keeps only a few vertices of the tree whose counter values are positive and satisfy the following inequalities:

$count(vertex) \leq \lfloor \frac{N}{f} \rfloor$ and $count(vertex) + count(parent) + count(sibling) > \lfloor \frac{N}{f} \rfloor$, where f is a compression factor. There are two exceptions to this rule: a leaf vertex can violate the first inequality while the root can violate the second inequality. Given such constraints, a device does not store more than $3 \cdot f$ vertices and can determine its normalized rank with an error inversely proportional to f.

In the original algorithm [16], the devices build the vertices they store by organizing themselves into a tree-based overlay. Child devices in the overlay forward their tree vertices to their overlay parent. The parent merges these vertices with its own ones by adding the counters of corresponding vertices, compresses the resulting tree by removing vertices that do not satisfy the previous constraints, and forwards the remaining vertices to its parent, and so on. Eventually, the same is done by the overlay root, which then has the necessary information to determine the normalized rank of any device. It can thus disseminate its vertices back to the devices, so that each device can compute its normalized rank and slice assignment.

In contrast, our framework abandons overlays in favor of opportunistic broadcast-based state exchanges between devices. This forced us to revisit the q-digest tree technique in our *DTree* algorithm. To start with, in *DTree* the counters of the tree vertices are implemented as sketches because, unlike in a tree-based overlay, a device can merge its vertices with the vertices of other devices multiple times and in a different order. Moreover, preliminary experiments showed that estimating a device's rank based on the range tree leads to huge errors. This is because the more merges of its tree vertices a device performs with other devices' vertices, the more selective the second inequality becomes, which in turn increases the error in the estimated normalized rank of the device. For this reason, apart from the tree, each device maintains two sketches as in *SCount*: the sketches are used to compute the device's rank, whereas the tree is used only for propagating aggregated information about the value distribution throughout the network. Finally, we also employ the aforementioned decentralized state sharing scheme for exchanging raw $\langle v_X, id_X \rangle$ pairs between devices, which aims to further reduce the estimation errors.

To sum up, our *DTree* algorithm works as follows. Each device maintains a subset of the domain range tree according to the previous rules and with sketches as counters, two similar sketch counters denoting the number of devices with smaller and nonsmaller values than the device's own value, and a collection of $\langle v_X, id_X \rangle$ pairs acting as records for the state sharing scheme. In every round, it broadcasts a subset of its tree vertices and $\langle v_X, id_X \rangle$ records. By symmetry, it receives similar information from its neighbors. It thus merges the received tree vertices with its own ones and compresses the resulting vertices, so that they satisfy the previous inequalities. It also merges the received records as in the state sharing scheme. What is more, it uses both, the received tree vertices and records, for updating the sketch counters denoting the number of devices with smaller and nonsmaller values than the device's own value. More specifically, the device invokes `merge` on each received tree vertex and at most one of the two counters, and `add` for each of the received record and exactly one of the two counters. In this way, the sketch counters are updated based on both aggregated and raw device values, which potentially speeds up the slicing process. The properties of sketches guarantee in turn that each device's value contributes at most once to either of the counters. A device thus computes its normalized rank using only the two sketch counters. Algorithm 5 presents the pseudocode for *DTree*.

Algorithm 5: DTree

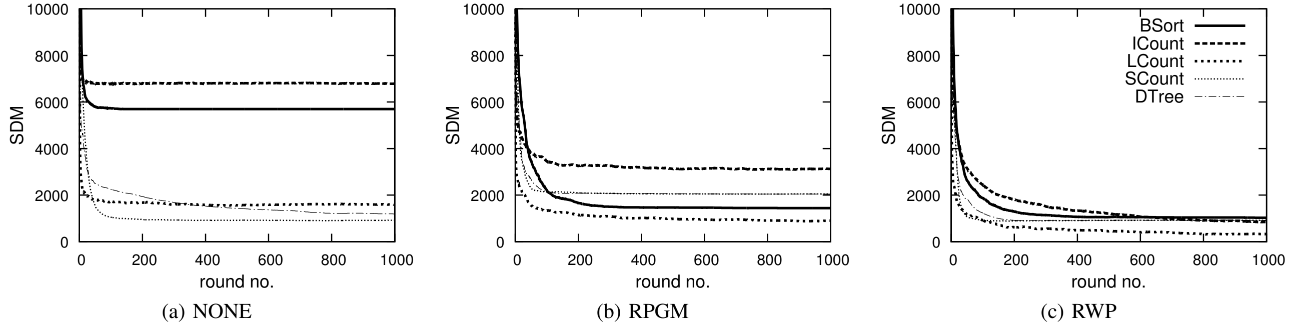
```
1  function init()
2      domainRangeTree ← init with myId and myValue
3      myLtSketch ← init as in SCount
4      myGeSketch ← init as in SCount
5      sharedRecords ← init with a record containing
         (myId, myValue)
6  function onTimer()
7      vertices ← pick a subset of domainRangeTree vertices
8      records ← pick a subset of sharedRecords
9      broadcast (vertices, records)
10 function onReceive(msg)
11     (vertices, records) ← msg
12     foreach (id,value) ∈ records do
13         depending on value and myValue, add id to either
             myLtSketch or myGeSketch
14     end
15     combine sharedRecords with records
16     foreach (sketch,range) ∈ vertices do
17         localVertex ← find a vertex in domainRangeTree that
             matches range
18         merge localVertex's sketch with sketch
19         depending on range and myValue, merge myLtSketch or
             myGeSketch with sketch
20     end
21     compress domainRangeTree so that its vertices satisfy the
         invariants
22     sliceEstimate ← compute as in SCount
```

F. Other Algorithms

We have also explored other decentralized slicing algorithms. However, due to space constraints, here we limit our study to the aforementioned representative ones.

V. Simulation Experiments

We start the study of the algorithms with simulations. To this end, we employ OMNeT++ [23] with the MiXiM [24] and INET [25] extensions. Together, they constitute a signal-level simulation engine with realistic low-power wireless communication models for networks of mobile devices. To further improve the accuracy of the results, we make the simulated devices match the characteristics of our testbed devices: the carrier frequency of 868MHz, the transmission output power of 12mW, and the bitrate of 250kbits/s. As



(a) NONE (b) RPGM (c) RWP

Figure 2. Representative runs involving 1024 devices under different mobility patterns within a $1024m\times1024$m playground.

a result, the effective radio range of a device, with 90% packet reception rate, is 45m, and the maximal range is 100m. Although we have performed numerous simulation runs of the algorithms in various configurations, due to space constraints, in this paper we present only a selected fraction of those simulations.

A. *Experimental Settings*

In the selected simulations, we use three device mobility models: random waypoint mobility (RWP), reference point group mobility (RPGM), and no mobility (NONE). In RWP, the devices move on a playground toward randomly selected points with random velocities between 0.3 and 1.66m/s (i.e., human walking speed), and a random pause of up to 1s after reaching a target. In RPGM, in turn, the devices are divided into groups with sizes from a normal distribution (μ = 10, σ = 1), each group having a leader that is followed by the other members in a distance of up to 30m; the device velocities are configured as in RWP. Finally, in NONE the devices are arranged in an equally-spaced grid and are stationary.

A device's identifier and value have 16 and 10 bits, respectively. The identifiers are from 0 to $N-1$. The values, in turn, are from a normal distribution ($\mu = \frac{2^{10}}{2}$, $\sigma = \frac{2^{10}}{6}$). Unless noted otherwise, a device transmits at most 100 bytes of payload per round to ultimately assign itself to one of up to 100 equally-sized slices. The round duration is in turn 10s. As the sketches in *SCount* and *DTree*, we assume linear counting sketches [26] with 152 bits (19 bytes) stored physically and mapped to a conceptual bit array of $\frac{2N}{\ln 152}$ bits [19], which we selected in preliminary experiments. In *DTree*, the compression factor is $f = 8$. Wherever possible, we aim to utilize the entire bandwidth resulting from the payload limit per round. To this end, in *LCount*, the observed pairs list is limited such that it does not exceed the available payload. In particular, with 100 bytes of payload, the list is limited to 25 pairs. In *SCount*, in turn, the state sharing scheme buffers locally 10 records, of which 2 are broadcast in every message. Finally, in *DTree*, the scheme has the same configuration, albeit with much smaller records, but in addition 4 tree vertices are broadcast in each message.

Note that despite unifying the techniques in our algorithmic framework, it is still hard to compare them systematically. One reason is the previous configuration parameters that they expose and that have to be set to avoid bias, which is not easy as the techniques do differ. Another is the probabilistic nature of the techniques, which is at odds with even days necessary to complete a single simulation. Therefore, rather than aiming at a thorough evaluation, here we just highlight our major observations from the experiments with the techniques, by showing only a few selected of those experiments. We hope that these selected experiments will give the reader an overview of the advantages and drawbacks of the techniques.

B. *Impact of Mobility*

To start with, let us present in Fig. 2 the behavior of the five algorithms in representative runs involving 1024 devices under the three mobility patterns in a playground of $1024m\times1024$m. The figure depicts the total error in slice assignment, expressed with a so-called slice disorder measure, *SDM*. *SDM* is defined as $\sum_{X\in Devices}|as(X)-rs(X)|$, where $as(x)$ is the index of a slice assigned to device X by an algorithm and $rs(X)$ is the index of a slice that should have been assigned to X, with an index of a slice denoting the position of the slice in a sequence of all slices ordered by their lower boundaries. In other words, *SDM* describes how much off all devices are from their correct slices. Several phenomena can be observed in the figure.

First, in static networks (Fig. 2(a)), the errors for *BSort* and *ICount* are much higher than for the other algorithms. In *ICount*, this is because a device receives $\langle v_X, id_X\rangle$ pairs belonging to only its neighbors, and hence is unable to compute its normalized rank with respect to the entire population. In *BSort*, rank swaps are also done between neighbors but they can propagate information over multiple hops: device A swaps its rank with B, which then swaps with C, and so on. However, a device may still hit a local extremum where it is unable to swap its rank with a neighbor, even though it could do this with a device two hops away. Additional dissemination mechanisms, like the continuously exchanged, random $\langle v_X, id_X\rangle$ pairs in *LCount* or the state sharing schemes in

DTree and *SCount*, significantly improve not only the error, but also the pace of convergence. Among the algorithms, the best performing ones are *DTree* and *SCount*, in which not only raw $\langle v_X, id_X \rangle$ pairs, but also aggregated information in the form of sketches are disseminated. *SCount* is faster than *DTree* as the latter pushes more data into the network. In any case, their errors are low: on average, a device assigns itself a slice that is off by 1–2 positions from its actual slice in the sorted slices sequence. Likewise, the pace of convergence is fine considering the population size. Within a few rounds the average slice assignment error decreases below 4. The subsequent improvement in the assignment error is much slower, though.

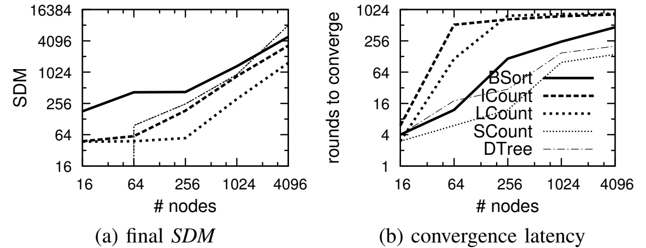
Under RPGM (Fig. 2(b)) the situation changes. For *BSort* and *ICount*, mobility helps spreading information. The improvement in *ICount* is smaller, as the mobility in RPGM is not completely uniform: devices are clustered into groups, and hence contacts between devices from different groups may be less frequent than between devices from the same group. In effect, a device in *ICount* has plenty opportunities to rank itself within its group but fewer in the case of the entire population. Similarly, in *BSort* mobility alleviates the local extremum problem, thereby allowing for rank exchanges virtually between arbitrary devices. In contrast, *SCount* and *DTree* perform worse than in the static networks. This is again likely because the communication opportunities are biased in favor of a device's own group, which negatively affects the state sharing schemes. For *LCount*, this effect does not occur, perhaps due to a slightly different dissemination scheme. In fact, *LCount* performs better under RPGM. Overall, the experiments illustrate that mobility need not improve slicing.

Finally, under RWP (Fig. 2(c)), a device encounters uniformly random other devices, which improves the performance of all algorithms. *LCount* seems the best due to the greatest buffer for random samples, which further speeds information dissemination. Other algorithms, in turn, perform comparably.

What in not visible in the figures, though, is that in *ICount* and *LCount* the slice assignment is not stable: a device may continuously change its slice even when globally *SDM* does not change much. This may be problematic because it makes it difficult for a device to decide whether the assigned slice is final or not. In contrast, in *BSort*, *SCount*, and *DTree*, the assignments of individual devices eventually do stabilize.

C. Impact of Population Size

Fig. 3 shows how the population size affects the algorithms under RWP mobility: the device population grows exponentially from 16 to 4096, and the playgrounds grow correspondingly from 128m×128m to 2048m×2048m to keep the average device density constant. We measure the final *SDM* after 1000 rounds. We also measure convergence latency, defined as the round number in which *SDM* first reaches a value within 10% of the final *SDM*. Each value in the plot is a median over 3 runs in the same configuration.



(a) final *SDM* (b) convergence latency

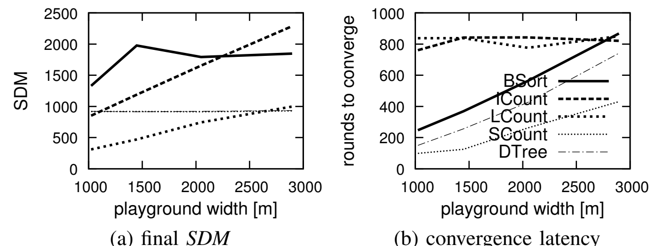
Figure 3. The algorithms' scalability with respect to the device population.

In all algorithms, the final *SDM* (a) seems to scale at least linearly with the device population. In *LCount*, the initial growth in *SDM* is lower because the $\langle v_X, id_X \rangle$ pairs kept by the devices cover a large fraction of the device population; in contrast, as the population grows, the fraction decreases, and thus *SDM* grows as well. In *SCount* and *DTree*, the final *SDM* in the smallest networks is zero because for such networks the sketches can give accurate estimates. Again, however, as the network grows, the estimates deteriorate, and so does *SDM*.

When it comes to convergence latency (b), the algorithms based on exchanging aggregate information, *SCount* and *DTree*, more quickly reach a value of *SDM* close to the final one. In contrast, the other algorithms, which in principle need to sample the entire population, take time to correct their *SDM*, with the lowest correction pace belonging to *ICount* and *LCount*, as can be verified in Fig. 2(c).

D. Impact of Device Density

Fig. 4 shows a similar plot but illustrating the effect of a growing playground (i.e., decreasing device density) with the population size fixed to 1024. A low average density is again problematic for *ICount* and *LCount*, as a device has fewer chances to encounter sufficiently many other devices, and hence to position its value with respect to all other values, which increases *SDM* (a). Exchanging $\langle v_X, id_X \rangle$ pair lists helps *LCount* in terms of absolute values but the trend is still visible. The convergence latency of *ICount* and *LCount*, in turn, seems unaffected by the density (b). In contrast, in *BSort*, *SCount*, and *DTree*, *SDM* does not change much in the plotted density range (a), but the convergence latency grows (b) due to a growing network diameter. In fact, in



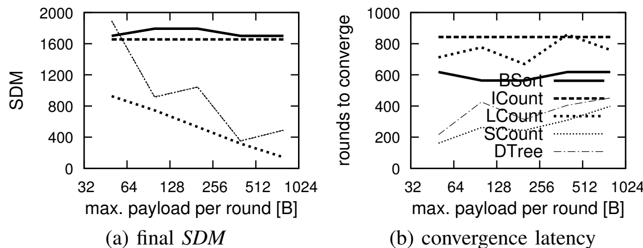
(a) final *SDM* (b) convergence latency

Figure 4. The algorithms' scalability with respect to the device density.

SCount and *DTree*, *SDM* remains constant with a growing playground, which illustrates that their accuracy is limited by the accuracy of the sketches.

E. Impact of Radio Throughput

Finally, Fig. 5 presents the impact of the available radio throughput on the algorithms. We exponentially vary the maximal number of payload bytes a device is allowed to transmit per round from 50B to 800B. Algorithms that exchange a fixed amount of information per round, *BSort* and *ICount*, do not benefit. In contrast, algorithms that can utilize the entire throughput for sharing samples or aggregated information, *LCount*, *SCount*, and *DTree* are able to achieve a lower *SDM* (a). The cost, however, is a slightly slower convergence (b).



(a) final *SDM* (b) convergence latency

Figure 5. The algorithms' scalability with respect to the radio traffic.

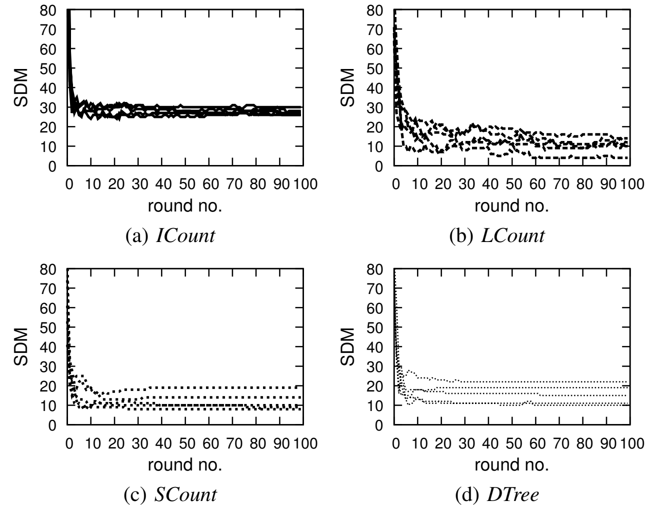
F. Additional Remarks

We have conducted numerous other simulations, in particular, studying the impact of movement velocities, round duration, division of slices, value distribution, and various parameters, to name several examples. Like in the presented ones, in those simulations, *LCount* and *SCount* were the best performing algorithms. Therefore, due to space constraints, we omit those results and proceed to real-world experiments.

VI. Testbed Experiments

To demonstrate that the considered algorithms can operate in the real world, we implemented them in TinyOS and deployed on our wireless sensor network testbed [27]. The testbed consists of 102 low-power wireless devices, of which 80 were operational during the experiments. The devices communicate in the 868MHz band and form a wireless network of 7 hops with nonuniform density and many asymmetric links. We selected to conduct the study on the static testbed rather on our suite of mobile devices because the results from the testbed tend to predict real-world behavior well and are reproducible. In contrast, an experiment involving 50+ human participants wearing, for instance, our smart name badges [3] is hard to reproduce. Again, we present only a subset of the conducted experiments.

In those experiments, the devices had to assign themselves to 10 equally-sized slices with a uniform distribution of values. To this end, they operated with 1s rounds with the transmitted payload limit of 50B/round, as this is the maximal packet payload of our devices. Like previously, wherever possible, we tried to fully utilize the available payload limit. More specifically, *LCount* stored 11 $\langle v_X, id_X \rangle$ pairs and broadcast them in each round; all sketches were 10 bytes long; the state sharing scheme in *SCount* stored 10 two-sketch records at a device and broadcast 2 records in every round; likewise, in *DTree*, 10 $\langle v_X, id_X \rangle$-pair records were stored, 2 of which were broadcast in every round together with 3 vertices of the range tree. The compression factor was again $f = 8$. Each algorithm was run 5 times. Fig. 6 depicts the runs apart from *BSort*, which we omit as its results bring little to the discussion.



(a) *ICount* (b) *LCount* (c) *SCount* (d) *DTree*

Figure 6. The algorithms' performance on the 80-device testbed.

The performance of the algorithms is in line with the simulations. Initially, all algorithms quickly reduce the *SDM*. Again, the reduction in *ICount* is the worst because the testbed devices are static. In contrast, *LCount* performs the best in terms of the final *SDM*. However, it requires many rounds to this end and, moreover, the slice assignments of individual devices continuously change, which may be problematic in some applications. In this view, *SCount* and *DTree* are attractive alternatives. They assigns slices fast, the resulting *SDM* is relatively low, and the assignments eventually stabilize. From the two, *SCount* is arguably better in terms of the final *SDM* but also implementation complexity. From a broader perspective, the results confirm that the considered decentralized slicing techniques can be used in the real world.

VII. Conclusion

To sum up, decentralized slicing in networks of low-power wireless devices is feasible. The problem itself, however, is difficult, as it potentially requires each device to sample or otherwise learn, directly or indirectly, how its value compares to the value of every other device. Nevertheless, in many cases, it is possible to relatively quickly perform approximate slice assignment. It is also surprising that the best

Table I
A SUMMARY OF THE ALGORITMS' ADVANTAGES AND DRAWBACKS

Algorithm	Advantages	Drawbacks
BSort	Uses a fixed amount of memory and bandwidth. Guarantees that slice estimates finally converge.	Does not perform well in static networks. Converges slowly. May be difficult to implement.
ICount	Uses a fixed amount of memory and bandwidth. Is easy to implement.	Does not perform well in static and mobile clustered networks. Converges slowly. Has its slice estimates fluctuate even if *SDM* is low.
LCount	Converges fast initially. Has a relatively low final *SDM*. Is easy to implement. Benefits from an increased network throughput.	Converges slowly in later stages. Has its slice estimates fluctuate even if *SDM* is low.
SCount	Converges fast. Has its final *SDM* independent of the device density. Benefits from an increased network throughput. Is relatively easy to implement. Guarantees that slice estimates finally converge.	Has its final *SDM* limited by the accuracy of counting sketches.
DTree	Converges relatively fast. Has its final *SDM* independent of the device density. Benefits from an increased network throughput. Guarantees that slice estimates finally converge.	Has its final *SDM* limited by the accuracy of counting sketches. Is difficult to implement.

results are achieved with simple counting techniques, augmented with a form of state sharing: *LCount* and *SCount*. In the considered scenarios, sharing raw values, as in *LCount*, often leads to better, albeit unstable assignments. Sharing approximate aggregated information, as in *SCount*, yields in turn slightly larger assignment errors but the assignments themselves eventually stabilize. In contrast, more complex solutions, like *BSort* and *DTree* typically perform worse. Therefore, in practice, *SCount* and possibly *LCount* are the most attractive choices. Table I summarizes our findings.

ACKNOWLEDGMENTS

This research was supported by the (Polish) National Science Center within the SONATA program under grant no. DEC-2012/05/D/ST6/03582. K. Iwanicki was also supported by the (Polish) Ministry of Science and Higher Education with a scholarship for outstanding young scientists.

REFERENCES

[1] M. Jelasity and A.-M. Kermarrec, "Ordered slicing of very large-scale overlay networks," in *Proc. P2P '06*, 2006.
[2] S. Deterding, D. Dixon, R. Khaled, and L. Nacke, "From game design elements to gamefulness: Defining "gamification"," in *Proc. MindTrek '11*, 2011.
[3] M. Grabowski, M. Marschall, W. Sirko, M. Debski, M. Ziombski, P. Horban, S. Acedanski, M. Peczarski, D. Batorski, and K. Iwanicki, "An experimental platform for quantified crowd," in *Proc. ICCCN '15*, 2015.
[4] K. Dantu, B. Kate, J. Waterman, P. Bailis, and M. Welsh, "Programming micro-aerial vehicle swarms with Karma," in *Proc. SenSys '11*, 2011.
[5] K. Iwanicki and M. van Steen, "Gossip-based self-management of a recursive area hierarchy for large wireless sensornets," *IEEE Tran. Parallel and Distributed Syst.*, vol. 21, no. 4, pp. 562–576, 2010.
[6] B. R. Iyer, G. R. Ricard, and P. J. Varman, "Percentile finding algorithm for multiple sorted runs," in *Proc. VLDB '89*, 1989.
[7] D. J. DeWitt, J. F. Naughton, and D. A. Schneider, "Parallel sorting on a shared-nothing architecture using probabilistic splitting," in *Proc. PDIS '91*, 1991.
[8] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *J. Comput. Syst. Sci.*, vol. 7, no. 4, pp. 448–461, 1973.
[9] R. W. Floyd and R. L. Rivest, "Expected time bounds for selection," *Commun. ACM*, vol. 18, no. 3, pp. 165–172, 1975.
[10] D. Kempe, A. Dobra, and J. Gehrke, "Gossip-based computation of aggregate information," in *Proc. FOCS '03*, 2003.
[11] F. Kuhn, T. Locher, and R. Wattenhofer, "Distributed selection: A missing piece of data aggregation," *Commun. ACM*, vol. 51, no. 9, pp. 93–99, 2008.
[12] A. Fernandez, V. Gramoli, E. Jimenez, A.-M. Kermarrec, and M. Raynal, "Distributed slicing in dynamic systems," in *Proc. ICDCS '07*, 2007.
[13] V. Gramoli, Y. Vigfusson, K. Birman, A.-M. Kermarrec, and R. van Renesse, "Slicing distributed systems," *IEEE Tran. Computers*, vol. 58, no. 11, pp. 1444–1455, 2009.
[14] F. Maia, M. Matos, E. Riviere, and R. Oliveira, "Slead: Low-memory, steady distributed systems slicing," in *Proc. DAIS '12*, 2012.
[15] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: An acquisitional query processing system for sensor networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, 2005.
[16] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri, "Medians and beyond: New aggregation techniques for sensor networks," in *Proc. SenSys '04*, 2004.
[17] G. Piatetsky-Shapiro and C. Connell, "Accurate estimation of the number of tuples satisfying a condition," in *Proc. SIGMOD '84*, 1984.
[18] S. Nath, P. B. Gibbons, S. Seshan, and Z. Anderson, "Synopsis diffusion for robust aggregation in sensor networks," *ACM Trans. Sen. Netw.*, vol. 4, no. 2, pp. 7:1–7:40, 2008.
[19] M. Gregorczyk, T. Pazurkiewicz, and K. Iwanicki, "On decentralized in-network aggregation in real-world scenarios with crowd mobility," in *Proc. DCOSS '14*, 2014.
[20] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "Gossip-based peer sampling," *ACM Trans. Comput. Syst.*, vol. 25, no. 3, pp. 8:1–8:36, 2007.
[21] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, 1985.
[22] D. Gavidia and M. van Steen, "A probabilistic replication and storage scheme for large wireless networks of small devices," in *Proc. MASS '08*, 2008.
[23] OMNeT++ Webpage. http://www.omnetpp.org.
[24] MiXiM Webpage. http://mixim.sourceforge.net.
[25] INET Webpage. https://inet.omnetpp.org/.
[26] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A linear-time probabilistic counting algorithm for database applications," *ACM Trans. Database Syst.*, vol. 15, no. 2, pp. 208–229, 1990.
[27] M. Michalowski, P. Horban, K. Strzelecki, J. Migdal, M. Klimek, P. Glazar, and K. Iwanicki, "A sensornet testbed at the University of Warsaw," University of Warsaw, Warsaw, Poland, Tech. Rep. TR-DS-01/12, 2012.