LASSO: A Device-to-Device Group Monitoring Service for Smart Cities

Matteo Saloni
University of Trento, Italy
matteo.saloni@studenti.unitn.it

Christine Julien
The University of Texas, USA
c.julien@utexas.edu

Amy L. Murphy
Bruno Kessler Foundation, Italy
murphy@fbk.eu

Gian Pietro Picco
University of Trento, Italy
gianpietro.picco@unitn.it

***Abstract*—Many smart city applications involve groups of individuals that wish to remain together as they move throughout the city. For example, a group of tourists may be monitored by a tour operator to keep the group together and on schedule. Alternatively, a group of elementary school children in transit to school should be closely supervised by an adult to ensure the children stay safe. This paper presents LASSO, a smartphone-based service that exploits wireless devices carried by each group member to provide infrastructure-free group formation and monitoring. We show how smartphones equipped with Bluetooth Low Energy (BLE) can be used as personal beacons in a device-to-device group monitoring protocol to allow each user to join a group and see a distributed view of group membership in real time. While LASSO is general purpose in nature, we demonstrate it and evaluate its performance through a prototype application used by a tourist guide to monitor tour participants.**

***Index Terms*—IoT, WSN, neighbor discovery**

I. INTRODUCTION

Applications in which cohesive groups of individuals move about abound in smart cities. In some cases a leader carefully monitors those present, while in other cases the groups operate with a kind of group responsibility, without a leader, but still in a cohesive manner.

To exemplify the generic needs of groups in smart cities, this paper focuses on city tours in which a group of tourists is often coherent as it moves from site to site. In this case, a tour guide desires to ensure no one wanders away from the group. Alternatively, the large group can disperse throughout the city, with smaller groups moving together without a specifically designated leader, but with the objective to stay together.

Similar scenarios arise in many other domains, including groups of children in transit to or from school or under a teacher's supervision on a field trip. Other groups may include maintenance or construction workers or groups of friends on a social outing. Here we focus on the tour group as the dynamics of the scenario are sufficiently diverse to capture many cases, including the optional presence of a designated leader.

Our objective is to support *group monitoring* through a smartphone service that leverages device-to-device interactions among individuals' devices. Developers can then directly incorporate this service into application implementations. We assume that each group member carries a wireless-capable device, but we build our solution using only local, device-to-device interactions. In other words, we do not use any cellular or WiFi infrastructure for group monitoring. In the end, our service, LASSO, presents each group member with a real-time representation of the current group membership. To the best of our knowledge, LASSO is the only group membership solution that operates in an entirely distributed manner and supports the discovery and maintenance of groups in which devices may be connected only across multiple network hops.

Under the hood we use a group membership algorithm [1] that relies on local, one-hop messages exchanged between neighboring nodes. These messages facilitate device discovery and information exchange among pairs of nodes to allow the distributed discovery of a group whose connectivity may span multiple network hops. LASSO instantiates this algorithm using Bluetooth Low Energy (BLE); we provide applications access to the group membership capabilities through an Android service that continuously reports to applications changes in group membership (e.g., an individual joining or leaving).

This work has been partially supported by the CLIMB project of the FBK Smart Community Lab and the National Science Foundation, CNS-1239498. Matteo Saloni was the recipient of a student grant by the IEEE Smart Cities initiative in Trento.

II. BUILDING BLOCKS

We introduce the two key building blocks of our service: (1) the use of BLE advertisements to support one-hop neighbor discovery and (2) the CLOCKS group monitoring protocol.

A. Neighbor Discovery with BLE Advertisements

While many approaches provide neighbor discovery using infrastructure assistance, we focus on an entirely infrastructureless solution in which each device plays both sides of the discovery role: periodically announcing its presence to other devices and scanning on some schedule to detect the presence of nearby devices. Many general neighbor discovery schemes exist [2]; here we briefly describe a simple scheme and how we adapt it to function on Bluetooth Low Energy (BLE), a technology that is readily available on many smartphones.

Generically, an infrastructureless neighbor discovery service delivers to the application a list of the identifiers of discovered neighboring devices. As the neighbors change due to device mobility, the service must regularly update the application. Many variations of such *continuous* neighbor discovery protocols exist, employing differing strategies for scheduling the beaconing and scanning schedules, both based on fixed-length slots [3]–[7] and without such slots [8], [9]. In this paper, we

978-1-5386-2524-8/17/$31.00 © 2017 IEEE

use a simple, slotless scheme that alternates between scanning and beaconing based on a fixed schedule. In the future, we intend to connect our group discovery service with our own slotless neighbor discovery protocol for BLE, BLEnd [8]. In the remainder of this section, we highlight some idiosyncrasies of BLE that must be handled in order to use it as the foundation of a neighbor discovery service.

BLE is a wireless communication technology that emphasizes low-power and short-range operation. BLE operates in the same spectrum as classic Bluetooth (2.4GHz band) but uses a different set of channels, three of which are dedicated to device discovery; the standard natively supports neighbor discovery, as it is fundamental to most BLE use cases. BLE uses a message-based neighbor discovery procedure that relies on the periodic broadcast of undirected beacons over dedicated *advertising* channels. In most BLE use cases, a device takes on one of two roles: the advertising device, which is *discoverable*, or the scanning device, which discovers other devices. To support group discovery, however, every device must be *both* discoverable and able to discover other devices. Therefore we create a BLE profile in which a device periodically switches between the advertising and scanning behaviors, and we employ randomization to prevent long-term synchronization of peers' advertising and scanning schedules.

Continuously broadcasting a device's BLE address can be considered a privacy concern, since it enables tracking of the person associated with the device. For this reason, BLE uses short-lived, fictional addresses to limit exposure. As we must uniquely identify the devices for the lifetime of the group management service, we cannot exploit these changing identifiers and therefore assign each device a fixed (application-specific) identifier. The details of how to handle this for each application are outside the scope of this paper.

Finally, BLE beacons are significantly limited in size. Nevertheless, after the application-defined ID is included, a small amount of payload remains available for customization by higher layers. The next section outlines the data our group monitoring service needs to transmit inside the beacons, while Section III-B provides implementation details about how this data is carried by BLE advertisements. We do not consider privacy, but note that the payload including the identifier can be encrypted by a key shared with group members.

B. Group Monitoring with CLOCKS

For our purposes, a *group* is defined as a set of devices that are *transitively connected to one another via network communication*. The primary requirement of a *group monitoring* service is to enable every member device to know, at any given time, the identity of the other devices in the connected group. We note that this definition is agnostic of whether the group has a designated leader responsible for group cohesion.

Because we use transitive network connectivity as a proxy for group connectivity, a group monitoring protocol can be directly supported by the ability of individual devices to detect the presence of other members in the reachable area. Therefore, we assume that every device participates in a low-level device discovery protocol as described previously, over which the device shares its view of the group status, thereby enabling distributed discovery of the entire group over multiple network hops. Simply discovering neighboring devices and sharing this information with other connected devices is however not sufficient; changes in group membership caused by the departure of devices must also be detected and disseminated to properly ensure a consistent view.

In summary, a group monitoring service satisfying our requirements should, for every device in the network:

- R1) detect nearby devices directly reachable (i.e., within one network hop);
- R2) detect nearby devices reachable over multiple hops;
- R3) detect changes in group composition over time; and
- R4) deliver a complete and (eventually) consistent group view to each group member.

To achieve these goals, we rely on the CLOCKS group monitoring protocol [1], which was designed to provide a *run-time* view of the group membership of a set of potentially mobile wireless sensor nodes, in contrast to pre-existing approaches that focused on *a-posteriori* log and trace analyses.

CLOCKS relies on *vector clocks* [10]. Each device maintains a local *logical* clock, i.e., a counter that represents "ticks", and an array containing the logical clocks of all the nodes in of the group—the vector clock. Each node periodically broadcasts its vector clock, asynchronously with respect to other nodes; in our case, we insert the node identifier and the vector clock directly in the neighbor discovery advertisement. Upon receiving a vector clock, each node merges it with its local one by preserving, for each element, only the larger value, as the latter represents the most up-to-date timestamp associated with a node. Further, the node's local clock is set to the maximum clock value found in the vector clock, to re-establish a common time reference of sorts.

As an example, consider a small network composed of four nodes arranged in a line A-B-C-D, and assume that the vector clock is $\langle 5, 3, 2, 1\rangle$ at A and $\langle 3, 4, 3, 4\rangle$ at B. The i^{th} element is the clock of the i^{th} node. When A receives the vector clock broadcast by B, it learns about the most recent clocks at the other nodes; this knowledge is reflected by updating A's vector clock to $\langle 5, 4, 3, 4\rangle$. On the other hand, when A broadcasts its own vector clock, B learns about A's new local clock; as this is also higher than B's own local clock, the latter is updated accordingly, yielding a new vector clock $\langle 5, 5, 3, 4\rangle$ at B.

This process is performed continuously at all nodes, updating each node's local view. Notably, this happens irrespective of changes in the network topology, as long as the network remains connected. For instance, D could move to the head of the line yielding a topology D-A-B-C, yet the periodic dissemination and merging of vector clocks ensures that the correct view is eventually re-established.

Further, by analyzing the received vector clocks for differences between logical clocks, each node can detect the disconnection of others. For instance, assume that A receives consecutive vector clocks in which the timestamp associated with D is always 4, while everyone else's clocks are increased.

This means that no node in the network has disseminated a new value of D, and therefore D is no longer connected to the others. In our group monitoring service, the disconnection of a node is signaled at another when the difference between their clocks is above a configurable threshold.

The frequency of neighbor discovery advertisements impacts the convergence time of the group monitoring service; the longer the interval between periodic beacons, the longer it takes the group to notice the departure of one or more of its members. In its original formulation, the CLOCKS protocol assumes that every device "ticks" its local logical clock at a fixed rate relative to a real-time clock on the device [1]. However, the clock also ticks when an advertisement is received in which one of the elements in the vector clock has a value higher than the the local logical clock. From a practical perspective, clocks are represented by an integer with a fixed maximum value that depends on the number of bits allocated to store and communicate the value. If the clock ticks more frequently than necessary, the value will reach the maximum and the clock will rollover back to zero. If the clock rolls over too frequently, it can be difficult to set a reasonable threshold value. To reduce the frequency of clock ticks, we make a simple modification to the original CLOCKS, namely we reset the timer associated to the periodic tick each time we update the local clock as a result of a received vector clock.

We note that one benefit of this approach is that it does not require symmetric discovery. That is, it is not necessary for pairs of devices to discover each other for them to agree on a view of the group membership. Instead, CLOCKS inherently enables transitive discovery through its shared vector clocks.

Revisiting the requirements, our solution supports R1 by detecting one-hop neighbors directly via advertisement reception and R2 by identifying nearby devices multiple hops away via the dissemination of vector clocks of transitively connected devices. Changes in group composition over time, R3, are detected by monitoring the contents of the received vector clocks and detecting neighbors with stale clock values. Finally, R4 is ensured by the combination of the first three and the fact that every device behaves the same with no designated leader.

III. LASSO: GROUP MONITORING SERVICE

Given the building blocks of the previous section, we now describe our complete group monitoring service for Android, LASSO, which realizes the CLOCKS protocol, addressing the added constraints of using BLE as the underlying technology for neighbor discovery. By leveraging common, personal devices, our service has a lower adoption barrier than one running on embedded or custom devices.

This section starts with an overview of the architecture then provide the details of the BLE beacons that provide the communication substrate and finishes with the application programming interface (API) of the group monitoring service and a sample application that uses it.

A. Architecture

Fig. 1 shows the overall architecture of the system with the bottom layers reifying the building blocks described in the previous section. Our solution relies on every participating device having BLE to send and scan for beacons; this use of BLE is incorporated into a neighbor discovery abstraction, wherein we define the contents of a beacon including our service-defined unique device IDs and a payload that carries information specific to the group monitoring service. The group monitoring service implements the CLOCKS protocol. Finally, LASSO offers an application programming interface (API) so that applications can incorporate group monitoring behavior, ultimately supporting user interaction, visualizations, notifications, or any other use of group information.

B. Implementation Details

The key abstraction in CLOCKS is the vector clock that is shared and received by each group member. In this section, we walk through the creation, update, dissemination, reception, and use of vector clocks, including their interplay with the BLE beacon technology used as a foundation.

At the core of the neighbor discovery layer, our system relies on device-to-device communication to detect neighbors and to distribute vector clocks. In the development of LASSO, we explored the use of both BLE [11] and Wi-Fi Direct [12]. To select the appropriate technology, we considered the requirements of the group monitoring service. The selected technology should provide a periodic beacon that can carry the CLOCKS vector clocks. It should also be widely available and usable on commodity mobile devices. BLE is widely available, but largely due to the low-energy nature of the protocol, the beacons in BLE are very small. They can accommodate our vector clocks, but only for groups of a limited size. On the other hand, Wi-Fi Direct's beacons are variable in length and can therefore accommodate vector clocks of any size. Wi-Fi Direct is also available on most commodity devices, but the protocol stack for employing it is unwieldy due to human-in-the-loop authentication requirements, and its behavior on commodity devices is still unstable.

Based on these considerations, our initial implementation of the neighbor discovery layer from Fig. 1 relies on BLE and uses a single BLE beacon to disseminate the vector clock.

A BLE beacon is a 47-byte packet, of which 8B are reserved by header and CRC. Android reserves 8B for its

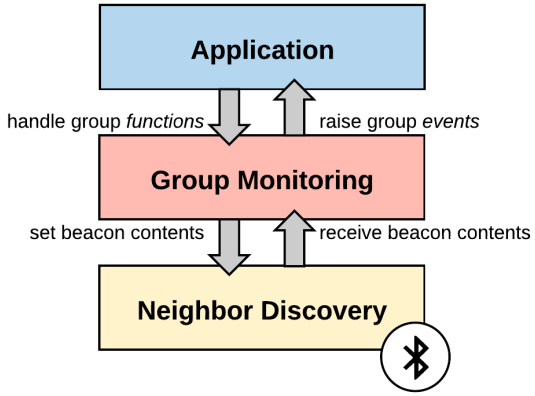


Fig. 1. High-level group monitoring service architecture.

own header and sending device's address, leaving only 31B to the application. Further, Android offers two constrained ways to write the payload of a BLE beacon: using *service data* or *manufacturer data*. Using service data, each payload must carry a 128-bit UUID, used to streamline beacon filtering associated with a given service. This, however, leaves only 15B available to the application. Using manufacturer data, on the other hand, Android reserves 10B to identify the manufacturer, leaving the remaining 21B application-writeable.

To implement CLOCKS using these 21B, each beacon contains the following:

- a custom identifier for our group monitoring service (2B)
- the group membership protocol's application identifier for the sender device (1B)
- the vector clock (up to 18 entries, each 1B)

The service identifier allows us to easily determine beacons that belong to our group monitoring service. The application-level device identifier addresses one of the Bluetooth limitations described in Section II: for privacy reasons, the device addresses in Android BLE change continuously. Using one byte means that we can address up to 255 different devices (i.e., potential group members). However, the 18B reserved for the vector clock limits us to 18 group members. The vector clock is then an ordered list of each of the 18 devices' known clock values. Because each clock is 1 byte, the clocks can tick 255 times before they "roll over" to 0. For a group of size 18, this granularity is more than sufficient to avoid confusion.

The limitations above are induced primarily by the Android BLE implementation. Other implementations of the neighbor discovery service could fragment larger vector clocks across multiple successive beacons or simply utilize alternate beacon technologies. On the other hand, such implementations come with the need to mitigate delays in the propagation of vector clock information in the case of fragmented vector clocks or increased energy usage in the case of larger beacons. With the emergence of Bluetooth 5, it is possible to consider spreading the beacon contents across multiple beacons or using variably sized beacons. However, as Bluetooth 5 is still largely unavailable, we consider only Bluetooth 4.

Starting with Android 5, the operating system supports devices operating in *peripheral mode*, the mode necessary to allow the device to send beacons. Our implementation of the CLOCKS protocol, is therefore compatible with any device running Android 5 or later. However, support for peripheral mode goes beyond just the operating system support and requires hardware support as well. Therefore, our group monitoring service will only function completely on devices that physically support peripheral mode[1].

As not all Android devices support this version of the BLE stack, we are also developing a proxy interface that allows us to extend a standard Android smartphone with a TI SensorTag[2], where we have full control over the BLE protocol.

[1]https://altbeacon.github.io/android-beacon-library/beacon-transmitter-devices.html

[2]http://www.ti.com/ww/en/wireless_connectivity/sensortag/

TABLE I
LASSO: GROUP MEMBERSHIP SERVICE API

Functions
`function nameID()` allows application to assign a (local) intuitive name to a specific ID
`function setClockResolution(int milliseconds)` set the frequency of the logical clock tick (default = 30 seconds)
`function setTimeout(int ticks)` set number of missed ticks that indicate a member left (default = 2)
`function getGroup()` return the list of IDs of current group members
Events
`event newMember(ID)` raised when a new member becomes part of the group
`event departedMember(ID)` raised when a member departs the group

In this case, the neighbor discovery layer in Fig. 1 is replaced by a proxy implementation that connects the group monitoring service running on the Android device to a neighbor discovery service running on the SensorTag. The connection from the Android device to the SensorTag uses BLE's connected mode, and this connection carries information transferred across the interface between group monitoring and neighbor discovery in both directions. This simple mechanism allows us to test our system with a wider range of smart phones, albeit adding the requirement to pair with an external SensorTag.

The details hitherto discussed concern the neighbor discovery layer in Fig. 1. This layer exposes an interface to the CLOCKS implementation in the group monitoring service that allows the latter to set the beacon contents (via a function `setBeaconContents` that accepts the byte representation of the vector clock as a parameter) and a single event callback that delivers any received vector clock back to the group monitoring service. With respect to the Android ecosystem, we implemented CLOCKS as described in Section II-B as a user-level application. It receives vector clocks from received beacons, maintains a local view of the global group state, and generates new beacon contents whenever the local vector clock representation changes, either because the local clock "ticked" or as a result of received vector clocks.

C. API and Sample Application

Table I shows the API that the LASSO group membership service offers, showing a set of functions applications can use primarily to setup the group membership service. The application can also use the `getGroup` function to retrieve a list of the IDs of current group members. Finally, the API also offers events that the group membership service raises and can trigger callbacks within the application, allowing the latter to be notified of changes in group membership.

Fig. 2 shows two views of the tour guide application. The first view, Fig. 2a, shows a wireframe depicting the user-facing view of the application. From the user's perspective, connected group members are shown with a green checkmark; disconnected members are grayed out and shown with a red cross. Fig. 2b shows a debugging version of the application, which relies on debug-level events to provide visibility into the

logical clock information stored in the group membership service. In this view, one can see the device ID (A) of the display device (in this case, device 2, which corresponds to the group member named Camden) and the logical clock value of the local device (B). The STOP button (C) allows the user to stop and start the group membership service, for debugging purposes. The most interesting part of the debugging view is the list of group member nodes (D). Here we can see a logical name of each device, the assigned ID for the group membership service (in this case a number from 1 to 6), and the device hardware ID (which may change over time due to the BLE implementation-level details discussed previously). The third column for each node entry shows the most recent known logical clock value for that device. Devices with the same clock value as the local clock (7) appear in green. Those within one clock tick are in yellow and are considered part of the group. Those with clock tick values more than two ticks in the past are highlighted in red or gray (depending on their age); these devices are considered disconnected, as indicated for the associated entries in the user view.

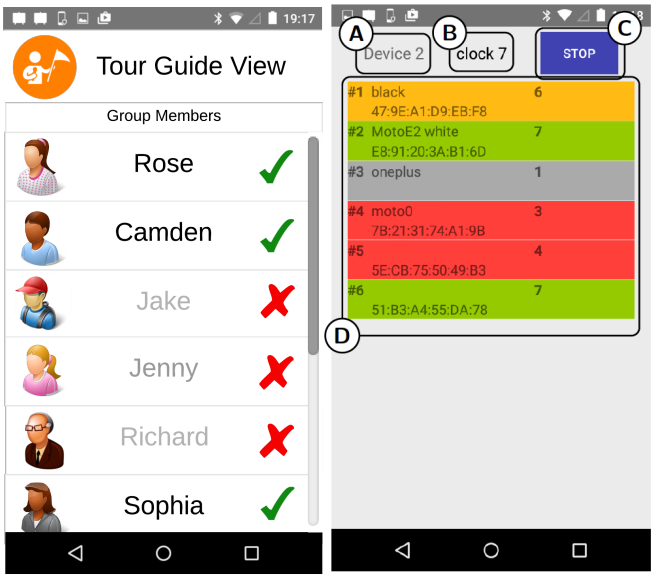
IV. Evaluation

To evaluate our group monitoring service, we performed microbenchmarks to measure reliability at the group monitoring level, performance at the neighbor detection level, and power consumption at the system level.

A. Setup

As previously mentioned, LASSO requires smartphones that support the *peripheral role*, which is not available, even in devices that support Bluetooth 4.1. As such, we limit our testing to a pair of Motorola Moto E 2nd generation devices.

In our tests, we statically placed the battery powered smartphones 2m apart in direct communication range. No other Bluetooth devices were present in the area, WiFi was turned on, but there was no 3G/4G connectivity. Each test ran for 7260s (254 clock ticks, approximately 2 hours). The real time clocks of the two smartphones were synchronized with NTP (Network Time Protocol) to ensure that log entries are comparable. The BLE advertisement is scheduled to alternate 5s of sending one advertisement per second, followed by 5s without transmitting. This sequence is repeated indefinitely. Scanning, instead, is repeatedly turned on for 29s, then off for 1s. When an advertisement occurs while the device is scanning, the radio briefly changes mode to transmit the advertisement. Finally, the logical clock advances at most every 30s, following the CLOCKS scheme.



(a) (b)

Fig. 2. (a) Wireframe of tour guide application. (b) Screenshot of debugging version of the tour guide application.

B. LASSO Reliability

Our first goal was to verify that clock values were properly exchanged between peers. For this, we require that each new clock value be received by the peer. We evaluate five executions, for a total of 1270 clock ticks. In all runs, all clock values were received, confirming the reliability of the group monitoring service, at least in this small scale test.

Interestingly, this does not mean that no advertisement packets were lost; this reliability was achieved thanks to the redundancy of the advertisements that, by design, send the same packet multiple times with the same vector clock.

C. BLE-Level Message Performance

Our next objective was to evaluate performance at the BLE level, both in terms of message redundancy and timing.

For the former, we count the number of packets sent and received, identifying the number of missed packets and the number of duplicates. As seen in Table II, for each of the 5 executions analyzed, we see just over 1000 messages sent. This is as expected, as each update to the group membership, whether locally initiated or due to an incoming message, triggers the transmission of a new vector clock. In our test setting with two nodes and 254 clock ticks, we expected, for each clock tick, a local plus a remote-induced update for each node. This leads us to expect around 1024 distinct clock values during a single run. The logged data reports slightly fewer as sometimes nodes increment their clocks due the arrival of an advertisement immediately before the 30 s timer expires.

Given the high data loss of BLE (between 17%-25%), we infer that the repetition of messages plays a fundamental role in ensuring reception of each unique clock value.

To measure the delay to receive a new clock value, we measure the wall-clock time between a *send* event with the new value and the corresponding *receive* event. Table III shows that messages were discovered with an average delay of 2s. Since BLE advertising is performed at a rate of 1 msg/s, the observed data falls within expectations.

D. Power Consumption

Modeling the power consumption of smartphone applications is a complex and challenging task [13]. While gathering knowledge of battery discharge behavior and recording

TABLE II
NEIGHBOR DETECTION MESSAGES

run	sent	received	missed	duplicates
1	1011	761	250	3218
2	1013	761	252	3216
3	1009	766	243	3971
4	1007	829	178	3592
5	1012	768	244	3932

TABLE III
MESSAGE RECEPTION DELAY (S)

run	max	mean	stddev
1	13	1.88	1.25
2	14	1.84	1.21
3	13	2.03	1.32
4	14	2.11	1.66
5	13	1.94	1.19

TABLE IV
POWER CONSUMPTION (MW)

device	profiler	baseline	runtime	Δ%
1	trepn	139.65	177.21	26,9%
1	powerTutor	218.11	262.34	20,3%
2	trepn	142.92	179.08	25%
2	powerTutor	221.36	270.82	22,3%

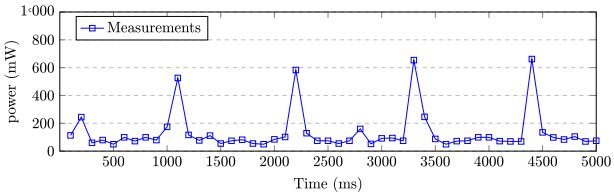


Fig. 3. Power measurements in a 5s window.

instantaneous measurements at different time intervals are effective for estimating the power consumption of the whole hw/sw platform, discriminating individual behaviors usually involves extensive analysis of measured data and domain-specific knowledge. Approaches based on external measurement devices are inherently more accurate and precise, but estimates based on power models and run-time phone measurements are sufficient to understand the basic behavioral trends and to evidence anomalies from the expected results [14].

Our power test represents a high-level overview of the energy consumption of the group monitoring application. With the help of third-party applications (Trepn Profiler[3] and PowerTutor[4]), we measure and log power consumption as reported by the operating system via battery APIs.

Since the group monitoring service is designed as a background service, the only relevant power absorption is caused by CPU usage and network communication as during tests the device screens are left off. We used the available tools to collect battery power levels, battery discharge over time, and CPU load and states. To separate the application from other system tasks, we also measure the baseline power absorption of the phone, taken with no active applications.

Table IV reports the consumption values in mW for both phones, baseline and active, along with the calculated increment in consumption. While measurements taken by the two profilers differ, the two phones exhibit similar behavior.

We observe a remarkable increase in global power consumption, mostly due to the Bluetooth radio during the broadcast and scanning. As seen at the 100ms sample rate of Fig. 3, the instantaneous consumption fluctuates substantially. This is due to the internal scaling of the processor, which switches states to perform the computation and then goes to sleep, and due to the behavior of the advertising process, which requires additional energy when active. We observe periodic power spikes at the same rate as the 1 s advertising period: our hypothesis is that BLE activity is the major contributor for power consumption. We plan to significantly reduce this by applying BLEnd [8], our continuous neighbor detection protocol specifically designed for the BLE stack.

[3]https://play.google.com/store/apps/details?id=com.quicinc.trepn

[4]https://play.google.com/store/apps/details?id=edu.umich.PowerTutor

V. CONCLUSIONS

This paper presented LASSO, a service supporting the monitoring of individuals moving as a group in a smart city. LASSO is designed with commodity smartphones in mind, therefore reducing the barrier to adoption. Its reliance on the widespread BLE wireless technology and fully decentralized device-to-device mode of operation enables its use in any mobile scenario, without the need of a pre-existing supporting infrastructure. The simple and versatile API of LASSO makes it easy to use for applications. Our small-scale performance evaluation shows that our approach is feasible. Ongoing work is aimed at integrating our recently-developed efficient neighbor discovery protocol [8] and at performing user studies validating the feasibility of the approach at scale.

REFERENCES

[1] M. Cattani, Ş. Gună, and G. P. Picco, "Group monitoring in mobile wireless sensor networks," in *Proc. of DCOSS*, 2011.
[2] L. Chen, R. Fan, K. Bian, M. Gerla, T. Wang, and X. Li, "On heterogeneous neighbor discovery in wireless sensor networks," in *Proc. of INFOCOM*, 2015.
[3] M. Bakht and R. Kravets, "SearchLight: Won't you be my neighbor?" in *Proc. of Mobicom*, 2012, pp. 185–196.
[4] P. Dutta and D. Culler, "Practical asynchronous neighbor discovery and rendezvous for mobile sensing applications," in *Proc. of SenSys*, 2008.
[5] A. Kandhalu, K. Lakshmanan, and R. Rajkumar, "U-Connect: A low-latency energy-efficient asynchronous neighbor discovery protocol," in *Proc. of IPSN*, 2010.
[6] M. McGlynn and S. Borbash, "Birthday protocols for low energy deployment and flexible neighbor discovery in ad hoc wireless networks," in *Proc. of MobiHoc*, 2001, pp. 137–145.
[7] K. Wang, X. Mao, and Y. Liu, "BlindDate: A neighbor discovery protocol," *IEEE TPDS*, vol. 26, no. 4, 2015.
[8] C. Julien, C. Liu, A. L. Murphy, and G. P. Picco, "BLEnd: Practical continuous neighbor discovery for bluetooth low energy," in *Proc. of IPSN*, 2017.
[9] P. H. Kindt, D. Yunge, G. Reinerth, and S. Chakraborty, "Mutually assisted slotless neighbor discovery protocols," in *Proc. of IPSN*, 2017.
[10] F. Mattern, "Virtual time and global states of distributed systems," in *Proc. of PDAA*, 1989.
[11] Bluetooth SIG, "Bluetooth core specification v4.2, 2014."
[12] D. Camps-Mur, A. Garcia-Saavedra, and P. Serrano, "Device-to-device communications with Wi-Fi direct: overview and experimentation," *IEEE Wireless Communications*, vol. 20, no. 3, pp. 96–104, 2013.
[13] M. A. Hoque, M. Siekkinen, K. N. Khan, Y. Xiao, and S. Tarkoma, "Modeling, profiling, and debugging the energy consumption of mobile devices," *ACM Computing Surveys*, vol. 48, no. 3, 2016.
[14] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proc. of CODES*, 2010.