Abstract—Due to limited energy supply on many Internet of Things (IoT) devices, asynchronous duty cycle radio management is widely adopted to save energy. Flooding is a critical way to disseminate messages through the whole network. Capture effect enabled concurrent broadcast is appealing to accelerate network flooding in asynchronous duty cycle networks. However, when the flooding payload’s size is large, the concurrent broadcast performance is far from efficient due to the frequently unsatisfied capture effect. Intuitively, senders can send a short packet containing partial flooding payload to keep concurrent broadcast efficiency. In practice, we still face two challenges. Considering packet loss, a receiver needs an effective way to recover the entire flooding payload from several received packets as soon as possible. Moreover, considering different channel states of different senders, how a sender chooses the optimal packet length to guarantee high channel utilization is not easy.

In this paper, we propose Chase++ a Fountain-code based concurrent broadcast control layer to enable fast flooding in asynchronous duty cycle networks. Chase++ uses Fountain code to alleviate the negative influence of a certain part of the flooding payload’s continuous loss. Moreover, Chase++ adaptively selects packet length with the local estimation of channel utilization. Specifically, Chase++ partitions long payload into several short payload blocks, further encoded into many encoded payload blocks by Fountain-code. Then, with temporal and spatial features of the sampled RSS (received signal strength) sequence, a sender estimates the number of concurrent senders. Finally, according to the estimated number of concurrent senders, the sender determines the optimal number of encoded payload blocks in a packet and assembles the encoded payload blocks as lots of packets. Then, the concurrent broadcast layer continuously transmits these packets. Receivers can recover the original flooding payload after several independent encoded payload blocks are collected. We implement Chase++ in TinyOS with TelosB nodes. We further evaluate Chase++ on Local testbed with 50 nodes and Indriya testbed with 95 nodes. The improvement of network flooding speed can reach 23.6% and 13.4%, respectively.

I. INTRODUCTION

More and more IoT applications have appeared in many scenarios, such as Industry 4.0 [1], smart city [2], smart home, and so on. In an IoT application, tens to thousands of devices are used to collect sensory data. For the feasibility of IoT deployment, some kinds of wireless techniques (e.g., Lora, ZigBee, Bluetooth, WiFi) are usually utilized to forward packets. In sensory data collection, some system settings (e.g., radio management [3], time synchronization [4], binary image [5]) of every node usually need a timely update to adapt to diverse system requirements. Network flooding serves as the basic function which disseminates control packets to every node in the whole network-wide. During the update of system settings, considering system reliability and consistency, the flooding packet must be quickly forwarded to every node.

When nodes keep their radio always on, flooding becomes a connected dominating set problem with link quality and conflict constraints. Under this scenario, many structured [6] [5] and structureless [7] [8] [4] network flooding protocols had been widely adopted. However, due to the limited onboard energy resource [9], duty cycle radio management is widely used to save energy. Low Power Listening (LPL, e.g., Box-MAC [10] ALIGNER [11], Zisense [12]) is a widely used asynchronous duty cycle control mechanism. In LPL as shown in Figure 1, each node periodically turns on its radio to sample signals (i.e., Clear Channel Access (CCA)) with a preconfigured interval (called sleep period). If a signal is detected, the node further keeps its radio on for a while (call listen tail). Otherwise, the radio is directly turned off. The schedule to turn the radio on (called sleep schedule) is not synchronized among different nodes. To meet every neighbor’s rendezvous, a broadcaster needs continuously send the same packet (called preamble packet) for the whole sleep period. The significant difference (i.e., asynchronous sleep schedule, long time channel occupation) between LPL broadcast and always radio-on broadcast keeps the directly inherited LPL network flooding schemes far from efficient.

To solve this urgent problem, Chase [13] had proposed a capture effect based LPL concurrent broadcast to accelerate network flooding. With Chase, each node could immediately forwards the received flooding packet without any channel access backoff. In this way, each node cannot miss the earliest chance to receive a flooding packet. Unfortunately, we observe that the performance of Chase is quite sensitive to preamble packet size. The flooding speed is slowed down with the increasing of preamble packet size (Section II-B). However, to shorten flooding rounds, the preamble packet size is usually set long during bulk data (e.g., binary image, batched system settings) dissemination. Consequently, the efficiency of Chase is dramatically degraded under these scenarios.
The root cause is the inefficient capture effect when the size of preamble packets is increasing. To control the channel utilization, an intuitive idea is that instead of setting the whole flooding payload as a long preamble packet, the flooding payload can be split into several different short payload blocks. Each preamble packet only contains several short payload blocks to shorten its length to keep the efficiency of capture effect. Hopefully, a receiver can collect all short payload blocks to recover the flooding payload. However, it faces two challenges. One is it may need a long time to collect all short payload blocks due to packet loss. The other is that a small preamble packet may underutilize the channel capacity of concurrent broadcast due to accumulated idle time during preamble packet broadcast. Thus, the optimal preamble packet length varies for different senders due to diverse channel state. For each sender, how to efficiently measure channel state to select the optimal preamble packet length is not easy.

In this paper, we propose Chase++, a Fountain code based concurrent broadcast control layer to enable fast flooding in LPL networks. First, Chase++ splits long flooding payload into some short payload blocks. It further uses Fountain code [14] to encode these short payload blocks as a set of rateless encoded payload blocks (called encoded payload block). Second, Chase++ develops a new method that uses the temporal and spatial features extracted from locally sampled RSS sequence to infer the number of concurrent senders. A sender then combines a different number of encoded payload blocks to adjust the preamble packet length to adapt the number of concurrent senders. Finally, senders send these preamble packets that consist of different encoded payload blocks for a whole sleep interval. A receiver can recover the flooding payload after enough encoded payload blocks are received. Fountain code’s rateless property can avoid long collection time incurred by the continuous loss of certain payload blocks.

We implement Chase++ on TelosB nodes in TinyOS 2.1.2. We further conduct extensive experiments to evaluate its performance on two real testbeds. The results show the flooding performance of asynchronous duty cycle networks can be greatly improved. Our contributions are summarized as follows:

- We propose Chase++, which first introduces a Fountain code based LPL concurrent broadcast control to improve the flooding performance in asynchronous duty cycle networks.
- We propose a novel method to enable a sender to count the number of concurrent senders. This method can be easily extended to infer other information of channel occupation.
- We implement Chase++ on TelosB nodes in TinyOS 2.1.2. We further evaluate its performance on two real testbeds with different network density and scale. The results show the efficiency of Chase++.

The rest of the paper is organized as follows. Section II shows the preliminary and limitation of state-of-the-art LPL network flooding. Section III show our system overview. Next, Section IV gives the detailed system design. Section V shows the implementation issues and the evaluation results.

Fig. 2. Illustration of random IPPI mechanism in Chase.

Section VII introduces the related work. We conclude our work in Section VIII.

II. EMPIRICAL STUDY AND MOTIVATION

Chase [13] proposed capture effect based LPL concurrent broadcast to allow all nodes can immediately forward the received flooding packet without any backoff. With high spatial reuse of concurrent broadcast, LPL flooding delay is significantly reduced. Next, we briefly illustrate the mechanisms used by Chase. Then, we conduct experiments to show the preamble packet size problem of Chase.

A. Chase Flooding

With capture effect, a receiver can successfully decode the strongest signal, which fulfills both time requirement (i.e., the strongest signal comes no 160μs later than other signals) and spatial constraint (i.e., the strongest signal must be 3dB higher than the sum of other signals). In Chase, random IPPI (Inter Preamble Packet Interval) and adaptive tail extension (ATE) are two main mechanisms to enable concurrent broadcast. Senders use random IPPI to construct valid preamble packets that fulfill the time requirement of capture effect at the receiver.

Figure 2 illustrates the random IPPI mechanism. S1 and S2 are two concurrent senders. R is a receiver in the communication range of both S1 and S2. For easily understanding the function of random IPPI, we currently assume the RSS (received signal strength) of S1 is 7dB higher than S2 at R. Hence, the spatial constraint is satisfied, and R can receive preamble packet of S1 when it fulfills the time requirement of capture effect. When R turns its radio on, D1 is the first heard preamble packet from S1. However, D1’ comes much earlier than D1. R fails to receive D1. Then, S1 and S2 wait for a random IPPI to send their next preamble packet D2 and D2’. The maximum IPPI is bounded by the time of LPL signal sampling [12]. With different IPPI waiting, the temporal order varies between different pairs of heard S1’s and S2’s preamble packets. As shown in Figure 2, D2 is also corrupted, but D3 (called valid preamble packet) is valid.

Moreover, if the spatial constraint is unsatisfied (i.e., the RSS difference between S1 and S2 is less than 3dB at R), R cannot successfully receive any overlapped preamble packet. When R does not receive any preamble packet in the passed listen tail, it further uses ATE to detect whether concurrent broadcast exists according to the RSS sequence sampled in the listen tail. If a concurrent broadcast is detected, R keeps
its radio on for another listen tail. Under this scenario, listen tail is not further extended until a non-overlapped preamble packet is received. In this way, Chase guaranteed the reliability of LPL concurrent broadcast.

B. Preamble Packet Size Problem

In practice, payload size is usually small for time synchronization or system parameter configuration, but getting large for binary image dissemination. When flooding payload becomes large, Chase tends to use long preamble packet that contains flooding payload as much as possible. As a result, packet size can vary from the minimum 18 bytes to the maximum 133 bytes [12]. To measure Chase under different payload size, we conduct experiments on Local testbed with 50 TelosB nodes. We set the transmission power of TelosB CC2420 radio to 2. For different nodes, the number of neighbors is in the range of [12, 27], while some links are lossy but the packet reception ratio is not smaller than 0.2. We use default parameters of LPL and Chase as shown in [13]. Given a payload size, we collect all 50 nodes’ logs in 100 times flooding as the dataset.

First of all, we evaluate the influence of payload size on a node’s reception delay, which indicates the node’s time to receive the flooding packet. Figure 3(a) shows the distribution of reception delay when the payload size is set to 10, 30, 50, 70 and 90, respectively. We can see 90% nodes can receive the flooding packet in 590 ms when payload size is 10 bytes. In contrast, given the same bound of reception delay (i.e., 590 ms), if payload size increases to 70 and 90 bytes, only 63.9% and 57% nodes can receive the flooding packet. Moreover, in the worst case, with 70 and 90 bytes payload, all nodes can be reached in 1440 ms and 1940 ms. In comparison with 10 bytes payload, the time is 35.8% and 83% larger. The results verify that the network flooding speed is significantly slowed down with the increase of payload size.

The root cause is the efficiency degradation of capture effect. With the increase of payload size, a node needs more time to receive a valid preamble packet that fulfills the temporal and spatial requirements of capture effect.

Specifically, because the maximum IPPI is bounded, the impact of random IPPI on temporal order adjustment among the preamble packets of different concurrent senders becomes limited when preamble packet is long. A valid preamble packet is difficult to construct in a short listen tail. According to our experiments, the distribution of listen tail length under different payload size is shown in Figure 3(b). We can see that the median and expected listen tail indeed monotonically increase with the increase of payload size. The median and expected listen tail values are increased by 3.6 and 4.9 times when the payload length increases from 10 bytes to 100 bytes, respectively. Consequently, the risk of long listen tail incurs large reception delay.

In Figure 3(b), we also notice that the range of the listen tail length is wide for all settings of payload size. Meanwhile, the listen tail length exhibits long tail distribution. For example, when payload size is 10 bytes, the range is from 10 ms to 602 ms. When payload size is 90 bytes, the range becomes wider from 17 ms to 1474 ms. The corresponding 75% values are 85 ms and 278 ms which are far less than the maximum. The huge gap is mainly caused by diverse number of concurrent senders for different nodes. If the number of a node’s concurrent senders is low, the node’s listen tail is short. In contrast, its listen tail becomes long when the number of its concurrent senders becomes high because more concurrent senders make the capture effect less efficient, equivalent to increasing the packet size.

Next, by utilizing the distribution of the nodes’ reception delay, we further show the number of concurrent senders in the whole network scale during network flooding. The results are shown in Figure 3(c). In Chase, each node continuously broadcast its received flooding packet for a whole sleep interval, namely 512 ms in our experiments. We can see the total amount of concurrent senders is monotonously accumulated at the beginning 512 ms. The maximum can reach 41 when the payload size is 10 bytes. Considering the topology of Local testbed is relative dense as mentioned in our experimental settings, its concurrent senders are quite dynamic and can be up to more than 20. Then, with the cease of some nodes’ broadcasts, the number of concurrent senders falls. It takes an extra long time 1050 ms until the last sender finishes its broadcast. Hence, even payload size is as short as 10 bytes, the node’s listen tail could still be long. Moreover, in Figure 3(c), with long reception delay, when payload size is getting large to 90 bytes, the maximum concurrent senders are smaller as 27. It takes more time 1920 ms until the last sender finishes its broadcast. We can see long payload size can incur the crowded
channel with less number of concurrent senders. The crowded channel has to last for a long time to cover the whole network leading to larger listen tail and reception delay.

Overall, we observe that when preamble packet size is large, the cost of capture effect becomes higher due to the long listen tail. Moreover, the network topology can also degrade the efficiency of capture effect. Without carefully setting a node’s concurrent senders, its listen tail can still be long under short payload size. Considering the multi-hop relay of network flooding, long per-hop listen tail significantly increases the end-to-end reception delay.

C. Motivation

We motivate to improve the channel utilization of Chase regardless of any network topology. The key question is that can we develop an optimal preamble packet generation scheme to enhance the efficiency of concurrent broadcast construction? Intuitively, we can partition flooding payload as several short payload blocks. A preamble packet only contains some of them to achieve efficient concurrent broadcast. However, we still face two challenges in practice. First, a receiver must receive multiple preamble packets to recover original flooding payload successfully. This may further increase delivery delay due to the frequent loss of a particular short payload block. Hence, a coding/decoding method should be designed to mitigate the long-tail problem [5]. Second, because different senders face a different number of concurrent senders, to optimize local channel utilization of capture effect based concurrent broadcast, the optimal preamble packet length varies for different senders. Moreover, to avoid the extra overhead and delay of network coordination, determining the optimal preamble packet length with only local knowledge is not a trivial problem.

III. System Overview

In this section, we introduce the overview of Chase++. As shown in Figure 4, the whole flooding system contains three layers, Flooding service layer, Chase++, and MAC layer. Flooding service layer sits upon Chase++. It abstracts two interfaces (i.e., initiate and receive) of Chase++ to allow upper layer applications to initiate network flooding and fetch the new arrival flooding payload. MAC layer is under Chase++.

It provides LPL communication primitives (i.e., Chase concurrent broadcast and receive) and wireless channel state (i.e., sampled RSS sequence) for Chase++.

The core layer is Chase++ that contains four modules to avoid over-utilized a channel, then achieve efficient concurrent broadcast. Specifically, after a flooding command is issued, payload partition module and fountain coding module convert long flooding payload to many short rateless payload blocks, which provide the feasibility to control the size of a node’s preamble packet. The rateless payload blocks do not rely on the reception of any individual block to reduce the block assembling delay. Moreover, according to the sampled RSS sequence, concurrent sender estimation module uses a clustering method to estimate the number of concurrent senders as current channel state. According to estimated channel state, adaptive preamble packets generator module defines a novel metric of channel utilization, then generates the optimal size of preamble packets for LPL concurrent broadcast. In this way, the settings of preamble packets are locally determined without any extra coordination delay.

After a preamble packet is received, MAC layer delivers the received payload to Chase++. The adaptive preamble packets generator module extracts several new rateless payload blocks. According to these rateless payload blocks, fountain coding module decodes several new original payload blocks. When payload partition module collects all original payload blocks, it assembles them as the complete flooding payload and triggers an event in flooding service layer. If a new flooding payload is received, flooding service layer immediately initiates network flooding to propagate the payload to the entire network further.

IV. System Design

In this section, we illustrate the detailed design of Chase++. Specifically, Chase++ contains four modules to achieve fast LPL concurrent broadcast based network flooding. After a flooding command is issued, payload partition module and fountain coding module convert long flooding payload to some short encoded payload blocks. The encoded payload blocks do not rely on the reception of any individual block so that the block assembling delay is optimized. Moreover, according to the sampled RSS sequence, concurrent sender estimation module estimates the number of concurrent senders as local channel state. According to estimated channel state and a novel metric of channel utilization, adaptive preamble packets generator module generates the optimal preamble packets for LPL concurrent broadcast. In this way, the length of preamble packet is locally determined without any extra coordination delay. Figure 5 illustrates an example of Chase++ design. The input of Chase++ is a flooding payload indicated as $P$. $L_p$ indicates the length of $P$. The output of Chase++ is a set of preamble packets $\{Pr_1, Pr_2, ..., Pr_m\}$. These preamble packets are further sent in turns with random IPPI.

A. Payload Partition

In this module, Chase++ sequentially partitions whole flooding payload to $k$ non-overlapped short payload blocks $\{p_1, p_2, ..., p_k\}$. The size of all short payload blocks is the
same and indicated as $l_p$. In Figure 5, $L_P$ and $l_p$ is 70 and 10, respectively. Thus, 7 payload blocks \{p_1, p_2, ..., p_7\} form the original flooding payload. It is possible $L_P$ cannot be divided by $l_p$. Then, Chase++ sets $k$ as \(\lceil \frac{L_P}{l_p} \rceil\) and fills the redundant bytes of the last short payload block $p_k$ with 0. Any node can recover the original flooding payload by collecting all $k$ short payload blocks. It is a tradeoff to choose $l_p$. A preamble packet contains multiple encoded short payload blocks (Section IV-D). Small $l_p$ provides fine-grained control of preamble packet length but needs to receive more encoded payload blocks to decode all $k$ short payload blocks in Fountain code [14] (Section IV-B). We discuss the selection of $l_p$ in Section V.

B. Fountain Coding

In this module, Chase++ converts $k$ short payload blocks to $n$ encoded payload blocks with Fountain code [14]. Given the set of short payload blocks \{p_1, p_2, ..., p_k\}, Fountain code randomly selects several short payload blocks and encodes them as an encoded payload block. The length of an encoded payload block is $l_p$ the same with a short payload block. For the efficient computation of short payload block encoding, bitwise XOR operation (i.e., Galois Field GF(2)) is usually adopted to encode the randomly chosen short payload blocks. As shown in Figure 5, $r_1$ is the bitwise XOR of $p_1$ and $p_2$. With different combinations of short payload blocks, a set of encoded payload blocks \{r_1, r_2, ..., r_n\} ($n > k$) are generated. Every short payload block is guaranteed to be contained in some encoded payload blocks. Instead of sending the short payload blocks, senders continuously send the preamble packets that contain these encoded payload blocks. After $w$ ($w \geq k$) encoded payload blocks are successfully received, a receiver can recover all short payload blocks with Gaussian Elimination (GE) or sum-product algorithm. The recovery does not rely on the reception of any individual encoded payload block. Thus, the long-tail problem is avoided.

Fountain code’s key problem is how to randomly select short payload blocks for encoding an encoded payload block. Random Linear (RL) [14] and LT [15] codes are two common schemes. The difference between RL code and LT code is the tradeoff between the encoding/decoding cost and the number of encoded payload blocks needed for successful decoding. In RL code, every encoded payload block has 50% probability to contain any short payload block. The expected encoding cost per encoded payload block is \(\frac{1}{2}\) bitwise XOR operations of two $l_p$ length bytes. Thus, the encoding cost of RL code is $O(k^2)$. With GE, the decoding cost is $O(k^3)$. According to [14], RL code has a high probability of recovering all $k$ short payload blocks when $k$ encoded payload blocks when $k$ is larger than 10.

In comparison, LT code uses a sparse random scheme to select short payload blocks. According to a distribution $\rho(d) = p_d$, an encoded payload block contains $d$ short payload blocks with probability $p_d$. Due to the non-uniform and sparse encoding, besides GE, the light-weight sum-product algorithm (i.e., Belief Propagation) [14] can be used for decoding. However, more encoded payload blocks are needed for decoding. In LT code, the distribution $\rho(d)$ is a critical issue to enable using less encoded payload blocks for decoding. The robust Soliton distribution [14] is designed for a large number of short payload blocks. SYNAPSE++ [16] and Pando [5] use another efficient distribution for a small number of short payload blocks.

Both RL and LT codes can be utilized for encoding in Chase++. To reduce the extra delay incurred by encoding, Chase++ uses a fixed part of IPPI to compute several new encoded payload blocks of the next preamble packet. In comparison with LT code, RL code needs longer fixed part of IPPI to encode so that loses certain temporal diversity among the preamble packets of different senders. This may degrade the efficiency of capture effect. On the other hand, considering the limited number of total short payload blocks, the computational cost of GE is tolerable. Moreover, rather than sum-product algorithm, GE can recover all short payload blocks when the coefficient matrix of received encoded payload blocks have full rank. Hence, Chase++ uses GE to decode. In comparison with LT code, RL code needs less received encoded payload blocks to construct the full rank coefficient matrix. We further compare the performance of RL and LT codes in Section V.

As shown in Figure 5, given the $n$ encoded payload blocks, Chase++ sequentially batches $\lambda$ (i.e., 3) encoded payload blocks as $m$ preamble packets. With a large $\lambda$, the preamble packet size is long so that a receiver may need more time to receive a preamble packet. On the other hand, according to the property of Fountain code, the receiver needs to receive less preamble packets to recover the original flooding payload when $\lambda$ is large. Based on the situation of local channel utilization, which depends on the number of concurrent senders, senders balance this tradeoff to determine the optimal batch size $\lambda$. The next module estimates the number of concurrent senders with only local information.

C. Concurrent Sender Estimation

In this module, a sender utilizes temporal and spatial features of the locally sampled RSS sequence to estimate the number of concurrent senders, which indicates the local channel utilization. Figure 6 shows two examples of the RSS spatial feature when 2 and 5 concurrent senders exist. The floor RSS (called noise floor, about -99dB) indicates the background
noise. The RSS samples, whose values are higher than noise floor, are the concurrent preamble packets of different senders.

The spatial feature indicates that due to the spatial diversity of different senders, the corresponding RSS values are usually different. With the concurrent preamble packets of different senders, a receiver can observe different RSS values. In Figure 6(a), two RSS values (i.e., -74dB, -61dB) are observed from two RSS segments. In Figure 6(b), the RSS values of different senders can be classified into 5 categories (i.e., -43dB, -54dB, -58dB, -69dB, -83dB). Based on this observation, Chase++ splits the RSS sequence as several RSS segments. In each RSS segment, the difference of RSS samples is less than a threshold $\kappa$. Then, Chase++ clusters all RSS segments to several categories as the rough estimation of the number of concurrent senders.

The RSS values of several senders may be close. These senders cannot be recognized through their spatial feature. In each RSS category, the temporal features (i.e., on-air time, the interval between adjacent RSS segments) of RSS segments are different when multiple senders exist. Specifically, given the maximum preamble packet on-air time $T_{on-air}$ and the minimum interval between adjacent preamble packets $IPPI_{min}$, if the on-air time of a RSS segment is larger than $T_{on-air}$ or the interval of adjacent RSS segments is less than $IPPI_{min}$, multiple concurrent senders exist. Chase++ further uses the temporal features to refine the estimation of the number of concurrent senders. The detailed estimation algorithms are illustrated as follow.

1) RSS Sequence Segmentation: $\{s_1, s_2, ..., s_n\}$ indicates the sampled RSS sequence. Chase++ splits it as $w$ segments $\{Seg_1, Seg_2, ..., Seg_w\}$ and uses start point $G_i$ and end point $E_i$ (i.e., sample index in RSS sequence) to depict the $i^{th}$ RSS segment $Seg_i$. Chase++ sequentially checks each RSS sample to create the start point set and end point set. Chase++ adds the first RSS sample as the 1st start point $G_1$. For any $s_j(j \in [1, n - 1])$, if the absolute difference between $s_j$ and $s_{j+1}$ is larger than a threshold $\kappa$, $j$ is the end point of the current RSS segment (Equation 2) and $j + 1$ is the start point of the next RSS segment (Equation 1). Chase++ adds the last RSS sample $s_n$ as the $w^{th}$ end point $E_w$. Chase++ sorts the elements of $G$ and $E$ in ascending order.

$$G = \{G_j | j \in [1, w], |s_{G_j-1} - s_{G_j}| > \kappa\}$$

$$E = \{E_j | j \in [1, w], |s_{E_j} - s_{E_{j+1}}| > \kappa\}$$

Then, the RSS samples, whose values are higher than noise floor, are the concurrent preamble packets of different senders.

2) RSS Segment Clustering: The RSS segment clustering algorithm is shown in Algorithm 1. From line 1 to line 3, Chase++ uses median RSS value as the spatial feature of a RSS segment. For the $i^{th}$ RSS segment, its median RSS value $s_{med}^{i}$ is the median value of $\{s_{G_i}, s_{G_i+1}, ..., s_{E_i}\}$. At line 4, Chase++ divides $\{s_{med}^{1}, ..., s_{med}^{w}\}$ to $N_c$ categories with Algorithm 2. The $i^{th}$ category of RSS segments is indicated as $C_{rss}^{i}$. For each RSS segment that belongs to $C_{rss}^{i}$, Chase++ uses the corresponding segment index in set $\{Seg_1, Seg_2, ..., Seg_w\}$ to represent it. If all segments of a category have few RSS samples, Chase++ treats this category as an outlier. Further, Chase++ removes the category of noise floor segments. As shown from line 5 to line 9, if the total RSS samples of all segments of a category is less than a threshold $\Delta$ or the median RSS value of a category is less than noise floor, Chase++ removes it from clustering set $C_{rss}$. Then, the remained $N_c$ categories is obtained.

According to the spatial features of RSS segments, Algorithm 2 uses a threshold (i.e., the median RSS difference between two categories is larger than this threshold) to automatically cluster them to $N_c$ categories. Chase++ sets the clustering RSS threshold as $\kappa$ the same with RSS sequence segmentation. In line 1, Algorithm 2 initiates the category set $C_{rss}$ as empty. From line 2 to line 16, it traverses all RSS segments. For each RSS segment, if $C_{rss}$ is empty, Chase++ creates a new category and adds it to $C_{rss}$. Otherwise, if the RSS difference between its median RSS value and the median RSS value $s_{med}(C_{rss})$ of the $i^{th}$ category $C_{rss}^{i}$ is within the range $[-\kappa, \kappa]$, Chase++ adds this RSS segment to $C_{rss}^{i}$. If the RSS segments does not belong to any category, Chase++ creates a new cluster and adds it to $C_{rss}^{i}$.

According to the spatial feature, Algorithm 1 and 2 together output category set $C_{rss}$ with $N_c$ categories, $N_c$ is the rough estimation of the number of current senders. In the $i^{th}$ category $C_{rss}^{i}$, it contains $N_c^{i}$ RSS segments. Their corresponding index in set $\{Seg_1, ..., Seg_w\}$ is $\{idx_{C_1}^{i}, ..., idx_{N_c}^{i}\}$. Hence, the corresponding start and end points are $\{G_{idx_{C_1}^{i}}, ..., G_{idx_{N_c}^{i}}\}$ and $\{E_{idx_{C_1}^{i}}, ..., E_{idx_{N_c}^{i}}\}$, which are sorted in ascending order.

3) Temporal Feature Checking: For each category, Chase++ uses the temporal features of the clustered RSS segments to check the potential multiple senders that provide...
Algorithm 2 Threshold Constraint Clustering Algorithm

Input: spatial feature of RSS segments \{s_1^{med}, ..., s_u^{med}\}, RSS difference threshold \(\kappa\).
Output: Category set \(C^\text{rss}\) with \(N_c\) categories.

1: Set \(C^\text{rss}\) as empty.
2: for Each segment \(Seg_i, i \in [1, u]\) do
3:     if \(C^\text{rss}\) is empty, then
4:         Create segment set \(\{i\}\) as \(C^\text{rss}_1\) and add it into \(C^\text{rss}\).
5:     else
6:         Currently, there are \(x\) categories in \(C^\text{rss}\).
7:         for Each category \(C^\text{rss}_j, j \in [1, x]\) do
8:             if \(|s_i^{med} - s_j^{med}|(C^\text{rss})| < \kappa\) then
9:                 Add \(i\) to \(C^\text{rss}_j\), break the loop.
10:         end if
11:     end for
12:     if Segment \(Seg_i\) belongs to no category, then
13:         Create \(\{i\}\) as a new cluster \(C^\text{rss}_{x+1}\), add it to \(C^\text{rss}\).
14:     end if
15: end for
16: return Category set \(C^\text{rss}\).

Algorithm 3 Temporal Feature Checking

Input: Maximum on-air time \(T_{\text{on-air}}^\text{max}\), minimum preamble packet interval \(\text{IPPI}_{\text{min}}\), \(N_c\) categories of RSS segments \{\(C^\text{rss}_1, C^\text{rss}_2, ..., C^\text{rss}_{N_c}\)\}.
Output: The number \(N_t\) of concurrent senders detected by temporal features.

1: Initialize \(N_t\) as 0.
2: for Each category \(C^\text{rss}_i, i \in [1, N_c]\) do
3:     for Each RSS segment \(Seg_{idx_j}, j \in [1, N_i^c]\) do
4:         \(t_{idx_j} = E_{idx_j} - G_{idx_j} + 1\)
5:         if \(j + 1 \leq N_i^c\) then
6:             \(\pi_{idx_j} = G_{idx_j} - E_{idx_j}\)
7:         end if
8:     end for
9:     \(\beta = \text{MAX}(|\pi_{idx_j}|, \forall j \in [1, N_i^c]|T_{\text{on-air}}^\text{max}\)
10: if \(\beta > 1\) then
11:     \(N_t = N_t + \beta\)
12: else if \(\exists j \in [1, N_i^c], \pi_{idx_j} < \text{IPPI}_{\text{min}}\) then
13:     \(N_t = N_t + 1\)
14: end if
15: end for
16: return \(N_t\)

similar RSS. Algorithm 3 shows the detailed procedure. In line 1, Chase++ initiates the number \(N_t\) of potential concurrent senders as 0. Chase++ uses RSS segment length and the RSS segment interval as the temporal features. In line 4, for the \(j\)th RSS segment of the \(i\)th category \(C^\text{rss}_i\), its RSS segment length \(l_{idx_j} \cdot c_i\) equals \(E_{idx_j} - G_{idx_j} + 1\). In line 5, if the \((j+1)\)th RSS segment exists, the RSS segment interval \(\pi_{idx_j} \cdot c_i\) is \(G_{idx_j} - E_{idx_j} \cdot c_i\). For each category, Chase++ calculates the temporal features of every RSS segment as shown from line 3 to line 8. In a category, if the ratio \(\beta\) between the maximum RSS segment length and the maximum on-air time \(T_{\text{on-air}}^\text{max}\) is larger than 1 (line 9 and line 10), it is possible the preamble packets of \(\beta\) senders are overlapped. Thus, Chase++ adds \(\beta\) to \(N_t\) (line 11). Otherwise, if there exists a RSS segment interval less than the minimum preamble packet interval (line 12), Chase++ conservatively adds an extra sender to \(N_t\) (line 13). Finally, after checking the temporal features of all categories, Chase++ obtains the extra number \(N_t\) of concurrent senders.

Combining rough estimation \(N_e\) and refined estimation \(N_t\), the estimated number of concurrent senders is \(N_e + N_t\). In practice, if the sampling period is long, we will obtain more RSS samples, which can be used to extract more overlapped patterns of the signals from different concurrent senders. Therefore, the accuracy of the estimation method can be improved. On the other hand, long sampling period will enlarge the processing delay on each node. Hence, there is a tradeoff between the estimation accuracy and the processing delay. We empirically study the parameter setting problem in Section V.

Basically, our clustering algorithm is a heuristic k-means algorithm. Given the DSSS modulation of IEEE 802.15.4, the RSS of a static link is relatively stable. According to our empirical observation (Section V), the maximum difference between two RSS samples of a link is about 3 dB. This difference is much smaller than the RSS difference between two links due to their spatial diversity. Then, our clustering algorithm utilizes this threshold as a heuristic condition to distribute different segments into different clusters. In this way, the deviation of each cluster is bounded and the distance of different cluster centers is much larger than the deviation. Hence, the results of our algorithm equal to that of k-means. The computation complexity of our clustering algorithm is \(O(n)\), where \(n\) is total number of valid RSS segments.

D. Adaptive Preamble Packet Generator

In this module, Chase++ defines an empirical metric CCR (Channel Capacity Redundancy) to determine the batch size \(\lambda\). CCR reflects the remained channel resource that allows how long preamble packet can be transmitted. To better understand the principle behind CCR, based on the results of some Chase experiments [13] and our empirical studies II, we emphasize on three observations as follows.

• Chase++ had evaluated the channel utilization in terms of tail length and packet delivery ratio regarding to different packet size and maximum IPPI. Given a fixed packet size, the results show the channel utilization can be enlarged by increasing the maximum IPPI. The reason is that the larger the maximum IPPI is, the more diverse the timing difference between the strongest signal and left signals is, which further leads to effective capture effect in current broadcast. Hence, CCR is proportion to the maximum IPPI.

• In Section II, our empirical studies (e.g., Figure 3(b) and Figure 3(c)) show that given a fixed maximum IPPI, the channel resource can be shared by less concurrent senders with the increasing of preamble packet length. Hence, CCR has an inverse relationship with the number of existing concurrent senders.

• The channel resource of concurrent broadcast is determined by the potential capture effect at the receiver side. Specifically, for a receiver, the effectiveness of capture effect relates to the strength difference of its received signals, which is determined by the spatial diversity of its potential transmitters, namely the network topology and deployment environment. Chase++ conducted the experiments with the same settings on different testbeds,
where the effectiveness of capture effect is quite different. In comparison with dense deployment (e.g., Local testbed in Section II), with more spatial diversity (e.g., Indriya testbed [17]), more channel resource can be allocated. Hence, an extra coefficient is need to indicate the influence of spatial diversity in CCR.

Overall, CCR is defined as follows. $IPPI_{\max}$ indicates the maximum interval between two preamble packets. $IPPI_{\max}$ is the total channel resource to enable concurrent broadcast. As shown in Equation 3, given the estimated number $N_c + N_t$ of concurrent senders, if $N_c + N_t$ is not zero, CCR is to multiply the ratio between $IPPI_{\max}$ and $N_c + N_t$, which indicates the average channel resource for each sender, by a coefficient $\delta$, which considers the spatial efficiency of capture effect. The large $\delta$ indicates good spatial diversity.

$$CCR = \text{MIN}(\delta \times \frac{IPPI_{\max}}{N_c + N_t} T_{\max}^{\text{on-air}})$$

(3)

We will discuss the $\delta$ selection in Section V. The maximum CCR equals to the maximum preamble packet on-air time $T_{\max}^{\text{on-air}}$. When CCR is large, it allows to transmit long preamble packet. When $N_c + N_t$ is zero, CCR is set as $T_{\max}^{\text{on-air}}$.

Given the radio bandwidth $B$, total size $l_{\text{mac}}$ of MAC header and tail, and payload block length $l_p$, the batch size $\lambda$ can be calculated as Equation 4.

$$\lambda = \text{MAX}(1, \lceil CCR \times B - l_{\text{mac}} \rceil)$$

(4)

The minimum $\lambda$ equals to 1 which ensure no node miss to receive the flooding packet. As shown in Figure 5, after obtaining the batch size $\lambda$, $Chase++$ further batches $\lambda$ encoded payload blocks for every preamble packet.

E. Reliable Coverage

Considering the possible preamble packet loss, after all neighbors have stopped broadcasting, a node may fail to collect enough encoded payload blocks to recover the original flooding payload, or even worse, not receive any preamble packet at all. To keep a reliable coverage, $Chase++$ allows a node to broadcast requirement packet to pull new information from its neighbor nodes. Specifically, if a node has detected the concurrent broadcast, but not enough encoded payload block or no preamble packet has been received when detecting clear channel in listen tail, it will immediately broadcast a requirement packet which contains the length $l_{\text{cw}}$ of contention window. After receiving the requirement packet, a neighbor node will initiate a $Chase++$ broadcast with a random backoff in the range of $[0, l_{\text{cw}}]$. Furthermore, if the node still fails to recover the original flooding payload, it will exponentially enlarge the length of the contention window and rebroadcast the requirement packet. The process is repeated till the node collects enough encoded payload blocks and successfully recovers the original flooding payload.

V. IMPLEMENTATION AND EVALUATION

We implement $Chase++$ on TelosB node with TinyOS 2.1.2. We implement the default Box-MAC [10] and $Chase$ [13] as the layer of LPL concurrent broadcast. Some system parameter settings are shown in Table I. $IPPI_{\min}$ is measured in Section V-B. $T_{\max}^{\text{on-air}}$, $l_{\text{mac}}$, $IPPI_{\max}$ and $B$ follow the Zigbee standard, $Chase$ default setting and CC2420 datasheet.

In practice, nodes use pseudo-random to generate random encoding coefficient matrix and random $IPPI$ between two adjacent preamble packets. Different nodes select the different seeds (e.g., node ID) to enhance the encoding diversity.

$\kappa$ and $\Delta$ are empirically set. For threshold $\kappa$, we randomly select 10 nodes on Local testbed and set their transmission power as 7. The 10 nodes transmit in turn. Each node continuously transmits 40 bytes preamble packet for 5 minutes. We use a static node to record the RSS values. Then, we measure the maximum difference among the RSS samples from the same transmitter as 3 dB. Therefore, we set $\kappa$ as 3 dB to ensure the RSS samples from the same transmitter can be distributed in the same cluster. We use the dataset collected in Section V-C to determine threshold $\Delta$. We observe that most of valid clusters have more than 10 RSS samples, but some obvious invalid clusters have only 1 or 2 samples since the value transmission RSS samples between two segments is usually random. Hence, we set $\Delta$ as 3 to keep the valid clusters as many as possible.

We use two testbeds, Local testbed (Section II) and Indriya testbed [17], to evaluate the performance of $Chase++$. Indriya testbed has 95 TelosB nodes, which are deployed across three floors. With much larger deployment scale, the spatial diversity of Indriya testbed is much higher than that of Local testbed.

A. Payload Block Length $l_p$

Here, we evaluate the performance with different short payload block length $l_p$. As the discussion in Section IV-A, small $l_p$ can provide fine-grained control of preamble packet length, but needs to receive more encoded payload blocks to construct full rank coefficient matrix in GE decoding. We randomly select 10 nodes as senders on Local testbed. We randomly select 3 nodes as receivers with different spatial diversity. We set the transmission power as 5 to ensure every receiver can hear all 10 senders. We use LR code for encoding. The total payload length is set 70 bytes. The coefficient $\delta$ of CCR is set as 1. CCR is 1.2 ms in the situation. The batch size $\lambda$ is 1, 3, 4, 6 for payload block size 20, 10, 8, and 5, respectively. We repeat 100 times for each setting and measure the average tail length under different short payload block size.

The evaluation results are shown in Figure 7. We can see when $l_p$ is 10 bytes, the average tail length is the smallest for
all receivers. The reason is that when \( l_p \) is 20, each preamble packet contains only 1 payload block. Small preamble packet length underutilizes the channel resource. When \( l_p \) is 10, 8 and 5, each receiver needs to collect more than 7, 9 and 14 encoded payload blocks (3 preamble packets at least) to construct full rank coefficient matrix for decoding. The more encoded payload blocks are needed, the more preamble packets are needed to decode. Thus, we set \( l_p \) as 10 bytes for later experiments.

### B. Coding Scheme

Both LR and LT codes can serve as the Fountain code scheme in Chase++. Theoretically, LR code has higher computational cost, but better coding efficiency than LT code. Moreover, using GF(16) or GF(256) can shorten the tail time since we need a smaller number of preamble packets to construct the full rank coefficient matrix than using GF(2). On the other hand, in comparison with the XOR operation of GF(2), it leads higher computing complexity which significantly increases the decoding time for a node with limited computation resource (e.g., TI MSP430 on Telosb). Hence, the \( q \) selection of GF(\( q \)) is a tradeoff between the tail time and the computation time.

We take the same experiment setting of Section V-A and set \( l_p \) as 10 bytes. For LT code, the encoding coefficient matrix is offline generated according to [16]. Its size is \( 51 \times 32 \). Hence, extra \( 204 \) bytes storage is needed in LT code. For LR code, the encoding coefficient matrix is generated in real time. By default, we use GF(2) to implement LR code and LT code. We also use GF(16) and GF(256) to implement LR code. We compare the performance between these four schemes in terms of encoding time, decoding time, the number of received preamble packets for decoding and tail time.

The experimental results are shown in Table II. The encoding time of LR and LT codes is 2 ms and 1 ms, respectively. The small encoding time is because the number of encoded payload blocks (i.e., 3 in this experiment) is small. The decoding time of LR and LT codes is 20.1 ms and 17.5 ms, respectively. LT code almost needs to receive 1 more preamble packet to successfully decode than LR code. This results that the average tail time of LT code is 24 ms larger than LR code. Comparing with LR code, although LT code has better computing efficiency, tail length is much longer since the inefficiency of channel utilization. Thus, we prefer to use LR code scheme in Chase++. The \( IFFI_{min} \) is set as 4 ms to contain the encoding time.

For LR code using GF(256), the encoding time and decoding time are 10.2 ms and 130.4 ms, respectively. They are at least 5 times of using GF(2). The corresponding number of decoding preamble packets and average tail time are 3.7 and 52.2 ms. In comparison with GF(2), the tail time is reduced 46.2%. We can see the benefits of tail time reduction are not large enough to compensate the increasing computation time. Hence, we prefer to use GF(2) on sensor node with limited computation resource.

For LR code using GF(16), the encoding time and decoding time are 2.8 ms and 35.4 ms, which are 0.8 ms and 15.3 ms larger than that of GF(2). Additionally, instead of using 9.1 preamble packets to construct the full rank matrix in GF(2), the receiver only needs to successfully decode 5.6 preamble packets in GF(16). As a result, the average tail length is shortened to 77.3 ms, which is 19.7 ms less than using GF(2). We can see the tail benefit of using GF(16) is comparable with its decoding overhead. Considering the implementation complexity, we choose GF(2) in our experiments.

### C. Concurrent Sender Estimation

We evaluate the accuracy of the concurrent sender estimation of Local testbed. We randomly select 2, 4, 6, 8 and 10 senders and set the transmission power as 5. We make each sender continuously broadcast packet with 40 bytes payload length. Then, we randomly choose 10 receivers to estimate the number of concurrent senders. Each receiver runs 100 times estimation for different concurrent sender situation. In each estimation, the receivers continuously sample the channel RSS for 24 ms. Figure 8 shows the distribution of estimation error. We can see that when the number of concurrent senders is not larger than 6, the average estimation error is less than 1 and the maximum estimation error is about 2. However, when the number of concurrent senders is 8 or larger, the estimation error increases quickly. The maximum estimation error reaches 5 when 10 concurrent senders exist. The reason is the increasing of signal overlapping when the number of concurrent senders becomes large. With more signal overlapping, the weak signals may be hidden behind the strong signal. Chase++ cannot count these senders whose signals are hidden. Thus, when the estimated number of concurrent senders is larger than 6, we conservatively set the number of concurrent senders as the average neighbor density of the network to alleviate the estimation error. When the packet reception ratio between two neighbor nodes is higher than 0.8, the average number of 1-hop neighbors is 10 and 7 on Local and Indriya testbeds, respectively. The corresponding transmission power is 2 and 17.
Indriya testbed
Indriya testbed

Completion Time (ms)
Tail Length (ms)
Tail Length (ms)

length is 70 bytes. We measure the tail length under four δ all 10 senders. We use LR code.

In each environment, we randomly select 10 nodes as senders and 1 node as receiver. The transmission power is 5 and 17 on Local testbed and Indriya testbed, respectively. The receiver can hear all 10 senders. We use LR code. \( l_p \) is 10 bytes. The payload length is 70 bytes. We measure the tail length under four δ settings 0.5, 1, 1.5 and 2. The result batch size \( \lambda \) is 2, 3, 5 and 6 for a preamble packet. Under each setting, we repeat the measurement for 100 times.

The experimental results are shown in Table III. We can see that the worst δ is 0.5 on both Local testbed and Indriya Testbed. The small batch size wastes too much available channel resource. The optimal δ is 1 and 2 on Local testbed and Indriya testbed, respectively. The difference is incurred by the different spatial diversity between Local testbed and Indriya testbed. In comparison with Local testbed, Indriya testbed has larger spatial diversity. Consequently, the efficiency of capture effect is better. Thus, with high spatial diversity, it can further tolerate the concurrent broadcast of long preamble packet. Thus, we set δ as 1 and 2 for Local testbed and Indriya testbed for later network flooding experiments.

### D. CCR Coefficient δ

In the calculation of CCR metric (Equation 3), the coefficient δ indicates the spatial efficiency of capture effect. We evaluate the influence of different δ on tail length in different environment (i.e., Local testbed and Indriya testbed). In each environment, we randomly select 10 nodes as senders and 1 node as receiver. The transmission power is 5 and 17 on Local testbed and Indriya testbed, respectively. The receiver can hear all 10 senders. We use LR code. \( l_p \) is 10 bytes. The payload length is 70 bytes. We measure the tail length under four δ settings 0.5, 1, 1.5 and 2. The result batch size \( \lambda \) is 2, 3, 5 and 6 for a preamble packet. Under each setting, we repeat the measurement for 100 times.

The experimental results are shown in Table III. We can see that the worst δ is 0.5 on both Local testbed and Indriya Testbed. The small batch size wastes too much available channel resource. The optimal δ is 1 and 2 on Local testbed and Indriya testbed, respectively. The difference is incurred by the different spatial diversity between Local testbed and Indriya testbed. In comparison with Local testbed, Indriya testbed has larger spatial diversity. Consequently, the efficiency of capture effect is better. Thus, with high spatial diversity, it can further tolerate the concurrent broadcast of long preamble packet. Thus, we set δ as 1 and 2 for Local testbed and Indriya testbed for later network flooding experiments.

### E. Network Flooding

We evaluate network flooding with different payload length on the two testbeds. We compare Chase++ with state-of-the-art concurrent broadcast based asynchronous duty cycle flooding Chase in terms of completion time. The completion time indicates the period from the sink initializes the flooding to the last node successfully receives the flooding payload. The transmission power is set as 2 and 17 on Local testbed and Indriya testbed, respectively. For each testbed, we repeat the flooding 100 times.

Figure 9 shows the distribution of tail length with different payload length. On Local testbed (Figure 9(a)), when the payload length is 20 bytes, the distribution of tail length is similar between Chase++ and Chase. The reason is that with small payload length, a preamble packet can also contain the whole payload with Chase++. We can see that with the increasing of payload length, the maximum tail length of Chase increases much faster than Chase++. When the payload length is 80 bytes and 100 bytes, the tail length of Chase++ is 33.3% and 31.8% less than Chase. Moreover, we can see the average tail length of Chase++ is larger than Chase. The reason is a receive has to collect multiple packets to decode. Since the average tail length is 10 times less than the worst case, the performance degradation of large average tail length is mitigated by the reducing of the worst case tail length. For Chase++ (Figure 9(b)), when payload length is 100 bytes, the worst case tail length of Chase is 802 ms, which is much less than that (1534 ms) on Local testbed. The worst case tail length of Chase++ is only 9.1% less than that of Chase. The performance improvement is also reduced. The reason is that the better spatial diversity of Indriya testbed can guarantee the efficiency of concurrent broadcast with large preamble packet.

The reliability of all experiments can reach 100% coverage on both Local testbed and Indriya testbed. Figure 10 shows the distribution of completion time with different payload length. On both Local testbed (Figure 10(a)) and Indriya testbed (Figure 10(b)), when the payload length is larger than 20 bytes, both the maximum and average completion time of Chase++ is less than Chase. The reduction of maximum and average completion time can reach 23.6% and 24.3% on Local testbed, 13.4% and 6.1% on Indriya testbed. This verifies the overall benefit of Chase++. In comparison with Local testbed, the performance improvement is reduced on Indriya testbed. The reason is that the better spatial diversity of Indriya testbed can guarantee the efficiency of concurrent broadcast even with large preamble packet.
Based on Deluge [7] and Box-MAC [10], we implement SYNAPSE++. Then, we compare its performance with Chase++ on Local testbed. The transmission power is set to 2. The packet length is 60 bytes. In SYNAPSE++, the long packet is divided to 6 short packets for coding. Each short coding packet is 10 bytes. In Table IV, the results show that the average and maximum completion time of SYNAPSE++ is 1874.5 ms and 2544.2 ms which is 1.7 and 2.1 times of Chase++, respectively. The reason is that the MAC layer backoff and contention of SYNAPSE++ can miss some early wake-up forwarding opportunities so that enlarge the completion time. In comparison, Chase++ optimize the efficiency of concurrent broadcasting which utilizes the early wake-up opportunities as much as possible to reduce the completion time.

We compare the performance between fixed batch size and our adaptive scheme on Local testbed. The transmission power is set to 2. The packet length is 60 bytes. For fixed batch method, we use the average number of neighbor nodes 10 to get a 1.2 ms CCR. Hence, the fixed batch size is 3. In Table IV, the results show that the average completion time of fixed batch method is 1187.9 ms which is 8.5% larger than that of Chase++. For the maximum completion time, fixed batch method is 21% larger than Chase++. The main reason is that fixed batch method can underutilize the channel resource when the actual number of concurrent senders is small. For example, at the beginning of a flooding process, the total number of concurrent senders is smaller than 10. It is better to use a larger batch size to utilize the channel resource fully.

VI. DISCUSSION AND FUTURE WORK

Exploring the topology information. When the radios of sensor nodes are always-on and have synchronous duty cycle schedule, the number of concurrent senders is correlated in a flooding process. It is largely determined by the network topology. The correlation is helpful to estimate the local channel utilization. However, if we consider the asynchronous wake-up schedules in Chase++, the number becomes more dynamic since a neighbor node cannot deterministically wake up earlier and serve as a concurrent sender in different flooding processes. Therefore, in Chase++, we do not explore the topology information and make all nodes forward their received flooding packet as Chase does. On the other hand, if we can model the opportunity of early wake-up forwarding based on a deterministic topology structure, we have the chance to optimize the flooding efficiency in a parallel way, which is our future work.

VII. RELATED WORK

Wireless network flooding has been widely studied in the last decade. Many protocols respectively leverage trickle timer [18], duplicate suppression [7], link quality [8], constructive interference [4] [19] and coverage structures [6] to accelerate network flooding. All of them assume the radio is always on for every node. However, in most of unattended IoT deployments, duty cycle radio management is adopted to extend network lifetime. Some works [20] [3] [21] (called synchronous duty cycle flooding protocols) are based on synchronized sleep schedule. All nodes simultaneously and periodically turn their radios on. After a certain period, they simultaneously turn their radios off. Glossy [4] based concurrent broadcast is further used to flood data to whole network. These synchronous duty cycle flooding protocols fit those applications with periodical flooding demands, but do not work for irregular flooding requests. In contrast, asynchronous duty cycle flooding protocols are more agile to free flooding pattern. With the explicit neighbors’ sleep schedule, opportunistic flooding [22], link correlation aware flooding[23] and duty cycle aware broadcast [24] [25] are proposed to improve network flooding efficiency. However, to maintain all neighbors’ sleep schedule needs extra synchronization or wake-up beacons. The synchronization error and wake-up beacon loss reduce the delivery chances. Zippy [26] develops a sophisticated radio, which can be always on and consume ultra low power to sense the ongoing flooding quickly. Without modified hardware, Chase [13] further proposes a completely distributed concurrent broadcast based flooding for asynchronous duty cycle networks. Based on Chase, our work further improves the efficiency of concurrent broadcast under different size of flooding payload.

Fountain code [14] are widely used to improve the efficiency of binary image dissemination. Rateless-Deluge [27] and SYNAPSE++ [16] use Fountain code to improve the performance of Deluge [7]. Pand [5] uses Fountain code to resolve the long-tail problem in constructive interference based code dissemination Splash [19]. All these protocols assume the radios of all nodes are always-on or have synchronous wake-up schedule. They are hard to be adopted in asynchronous duty cycle networks directly. Chase++ shares the idea of using Fountain code to improve the channel utilization. Beyond that, Chase++ first introduces Fountain code to accelerate network flooding in asynchronous duty cycle networks and provides a cross-layer design which combines Fountain code with under-layer Chase concurrent broadcasting.

VIII. CONCLUSION

To conclude, we propose Chase++, a Fountain code based concurrent broadcast control layer to enable fast flooding in LPL networks. First, Chase++ uses Fountain code to convert the long flooding payload to lots of short encoded payload blocks. Then, according to the sampled RSS sequence, Chase++ extracts several features to estimate the number of concurrent senders. Finally, combining concurrent sender information and encoded payload blocks, Chase++ explores an empirical metric to determine the optimal preamble packet length in concurrent broadcast and generates preamble packets. We evaluate Chase++ on two real testbeds. The experimental results show the efficiency in terms of completion time.

ACKNOWLEDGEMENT

This study is supported in part by the NSFC Grant 61972218, 61572277, 61902122, 61872081, NSFC Key Project 61632013, and National Key Research and Development Project 2018YFB2200900.
REFERENCES


Zhichao Cao received the BE degree in the Department of Computer Science and Technology at Tsinghua University, and his Ph.D. degree in the Department of Computer Science and Engineering at Hong Kong University of Science and Technology. He is currently with the Department of Computer Science and Engineering, Michigan State University. His research interests include wireless systems, mobile and edge computing. He is a member of the IEEE and ACM.

Jiliang Wang is currently an associate professor in School of Software and BNRist, Tsinghua University, China. His research interests include wireless and sensor networks, Internet of things, and mobile computing. Jiliang Wang received his BE degree in computer science and technology from University of Science and Technology of China and his Ph.D. degree in computer science and engineering from Hong Kong University of Science and Technology, respectively. He is a member of the IEEE and ACM.

Daibo Liu received the Ph.D. degree in computer science and engineering from the University of Electronic Science and Technology of China, Chengdu, China, in 2018. He was a Visiting Researcher with School of Software, Tsinghua University from 2014 to 2016, and Department of Electrical and Computer Engineering, University of Wisconsin-Madison From 2016 to 2017. He is currently an assistant professor with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, China. His research interests cover the broad areas of low power wireless networks, mobile and pervasive computing, and system security. He is a member of the IEEE and ACM.

Qiang Ma received his BS degree in Department of Computer Science and Technology from Tsinghua University, China, in 2009, and Ph.D. degree in Department of Computer Science and Engineering at the Hong Kong University of Science and Technology in 2013. He is now a assistant researcher in Tsinghua University. His research interests include sensor networks, mobile computing, privacy. He is a member of the IEEE.

Xin Miao received the B.E. degree from the Department of Computer Science and Technology, Tsinghua University, and the Ph.D. degree from the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology. He is currently with the School of Software, Tsinghua University. His research interests include wireless sensor networks, RFID, and mobile computing.

Xufei Mao received the Ph.D. degree in Computer Science from Illinois Institute of Technology, Chicago in 2010. He received the MS degree (2003) in Computer Science and the Bachelor degree (1999) in Computer Science at Northeastern University and Shenyang University of Technology respectively. He is currently with the School of Cyberspace Security, Dongguan University of Technology. His research interests span wireless ad hoc networks, wireless sensor networks and edge computing.