Chase++: Fountain-Enabled Fast Flooding in Asynchronous Duty Cycle Networks

Zhichao Cao*, Jiliang Wang*, Daibo Liu†, Xin Miao*, Qiang Ma*, Xufei Mao‡
*School of Software and TNLIST, Tsinghua University, China
†School of CSE, University of Electronic Science and Technology of China, China
‡WIN Lab, Dongguan University of Technology, China
{caozc, jiliang, miao, maq}@greenorbs.com, {dbliu.sky, xufei.mao}@gmail.com

***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 quickly disseminate system parameters to adapt diverse network requirements. Capture effect enabled concurrent broadcast is appealing to accelerate network flooding in asynchronous duty cycle networks. However, when the length of flooding payload is long, due to frequently unsatisfied capture effect construction, the performance of concurrent broadcast is far from efficient. Intuitively, senders can send short packet that contains partial flooding payload to keep the efficiency of concurrent broadcast. In practice, we still face two challenges. Considering packet loss, a receiver needs an effective way to recover entire flooding payload from several received packets as soon as possible. Moreover, considering diverse channel state of different senders, how a sender chooses the optimal packet length to guarantee high channel utilization in a light-weight way 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 the continuous loss of a certain part of flooding payload. Moreover, *Chase*++ adaptively selects packet length with the local estimation of channel utilization. Specifically, *Chase*++ partitions long payload into several short payload blocks, which are 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, concurrent broadcast layer continuously transmits these packets. Receivers can recover 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 timely update to adapt diverse system requirements. Network flooding serves as the basic function which disseminates control packets to every node in whole network wide. During the update of system settings, considering system reliability and consistency, the flooding packet must be quickly forwarded to every node.

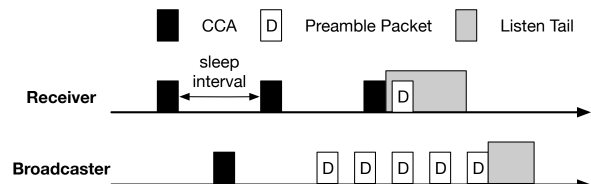


Fig. 1. Illustration of LPL radio management.

When nodes keep their radio always on, flooding becomes a connected dominating set problem with the constraints of link quality and conflict. Under this scenario, many structured [6] [5] and structureless [7] [8] [4] network flooding protocols have been widely adopted. However, due to the limited on-board energy resource [9], duty cycle radio management is widely used to save energy. Low Power Listening (LPL, e.g., Box-MAC [10], Zisense [11]) 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 radio on (called *sleep schedule*) is not synchronized among different nodes. To meet the rendezvous with every neighbor, a broadcaster needs continuously send the same packet (called *preamble packet*) for 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* [12] has proposed a capture effect based LPL concurrent broadcast to accelerate network flooding. With *Chase*, each node can immediately forwards the received flooding packet without any channel access backoff. In this way, each node cannot miss the earliest chance to receive flooding packet. Unfortunately, we observe

that the listen tail quickly increases with the increasing of preamble packet length (Section II-B). However, to shorten flooding rounds, the preamble packet length 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 construction when preamble packet length becomes large. An intuitive idea is that instead of setting whole flooding payload as a long preamble packet, the flooding payload can be split as several different short payload blocks. Each preamble packet only contains several short payload blocks to shorten its length so that the efficiency of capture effect construction is kept. Hopefully, a receiver can collect all short payload blocks to recover the flooding payload. However, it faces two challenges. One is it may need long time to collect all short payload blocks due to packet loss. The other is that too 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 in a light-weight way 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 [13] to encode these short payload blocks as a set of rateless short payload blocks (called *encoded payload block*). Second, *Chase*++ develops a new method that uses the temporal and spatial features of locally sampled RSS sequence to infer the number of concurrent senders. Then, a sender combines 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 recovery the flooding payload after enough encoded payload blocks are received. The rateless property of Fountain code can avoid long collection time incurred by continuously loss of certain payload blocks.

We implement *Chase*++ on TelesB 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 and light-weight 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*++.

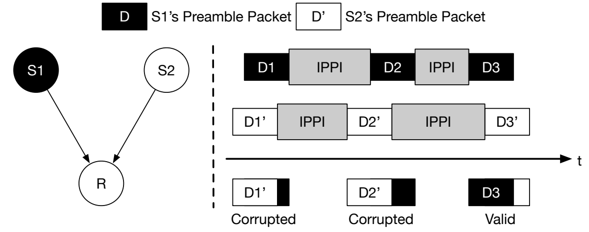


Fig. 2. Illustration of random IPPI mechanism in *Chase*.

The rest of paper is organized as follow. Section II shows the preliminary and limitation of state-of-the-art LPL network flooding. Next, Section III gives the detailed system design. Section IV shows the implementation issues and the evaluation results. Section V introduces the related work. We conclude our work in Section VI.

II. Empirical Study

Chase [12] proposes 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. All senders distributedly 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 a random IPPI to send their next preamble packet D2 and D2'. The maximum IPPI is bounded by the time of LPL signal sampling [11]. 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.

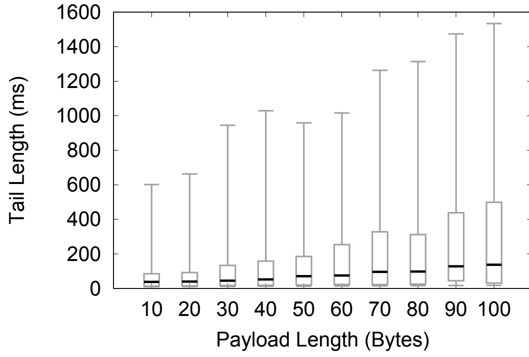


Fig. 3. The distribution of tail length under different packet length with *Chase*.

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 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* guarantees the reliability of LPL concurrent broadcast.

B. Preamble Packet Size Problem

When flooding payload becomes large, *Chase* tends to use long preamble packet that contains flooding payload as much as possible. However, this strategy may incur two negative effects. First, because the maximum IPPI is bounded, the impact of random IPPI on temporal order adjustment among the preamble packets of different senders becomes limited when preamble packet is long. A valid preamble packet is difficult to construct in a short listen tail. Second, long preamble packet increases the overlapping probability so that enlarges the number of listen tail extension with ATE. In real IoT deployments, there are usually more than two concurrent senders during network flooding. More frequent channel access of multiple senders further aggravate these two negative influences.

To verify our inference, we conduct experiments on a *local testbed* with 50 TelosB nodes. We set the transmission power of TelosB CC2420 radio to 2. The number of neighbors of different nodes are in the range of [12, 27]. We use default parameters of LPL and *Chase* as shown in [12]. We evaluate the distribution of listen tail length under different payload length. Given a payload length, we collect listen tail length of all 50 nodes in 100 times flooding as the dataset.

The experimental results are shown in Figure 3. We can see that listen tail length indeed monotonically increases with the increase of payload length. More specifically, the median and 75% value of listen tail is increased by 3.6 and 5.9 times when the payload length increases from 10 bytes to 100 bytes, respectively. The different increasing rate is caused by different number of concurrent senders for different nodes on local testbed. When the number of concurrent senders is low, listen tail is short and the increasing rate is relatively low. In contrast, listen tail is long and the increasing rate is relatively high when the number of concurrent senders is high. Thus, when preamble packet length becomes large, the time of capture effect construction becomes high. Considering the multi-hop relay of network flooding, long per-hop listen tail significantly increases the end-to-end delay.

The question is that *can we develop an optimal scheme of preamble packet generation to enhance the efficiency of concurrent broadcast?*. 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 successfully recover original flooding payload. 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 the 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 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 together 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 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 packet 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 4 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 4, L_P and l_p is 70

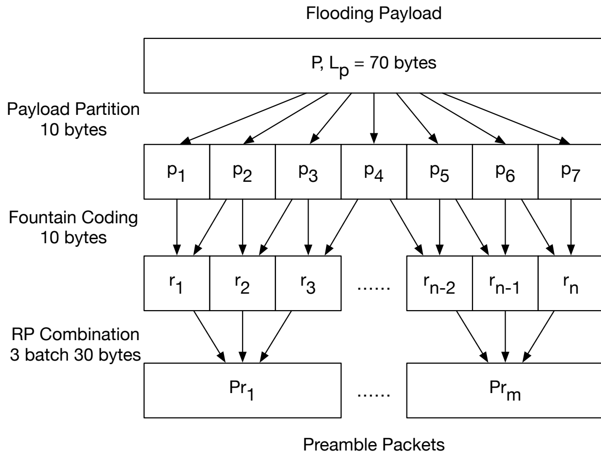


Fig. 4. An Example of *Chase*++ design.

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 III-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 [13] (Section III-B). We discuss the selection of l_p in Section IV.

B. Fountain Coding

In this module, *Chase*++ converts k short payload blocks to n encoded payload blocks with Fountain code [13]. 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 4, r_1 is the bitwise XOR of p_1 and p_2. With different combination 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 contains 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.

The key problem of Fountain code is how to randomly select short payload blocks for encoding an encoded payload block. Random Linear (RL) [13] and LT [14] 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 $\frac{k}{2}$ 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 [13], RL code has high probability to recovery all k short payload blocks with k encoded payload blocks when k is larger than 10.

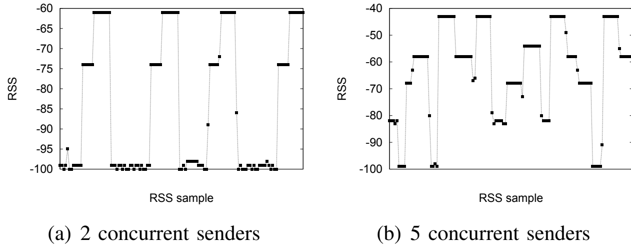
In comparison, LT code uses a sparse random scheme to select short payload blocks. According to a distribution $\rho(d) = p_d$, an encoded payload block contains d short payload blocks with probability p_d. Due to the non-uniform and sparse encoding, besides GE, the light-weight sum-product algorithm (i.e., Belief Propagation) [13] 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 [13] is designed for large number of short payload blocks. SYNAPSE++ [15] and Pando [5] use another efficient distribution for 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 construction 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 IV.

As shown in Figure 4, given the n encoded payload blocks, *Chase*++ sequentially batches λ (i.e., 3) encoded payload blocks as m preamble packets. With a large λ, the preamble packet length 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 λ is large. Base on the situation of local channel utilization, senders balance this tradeoff to determine the optimal batch size λ. The next two modules determine the detailed techniques.

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



(a) 2 concurrent senders (b) 5 concurrent senders

Fig. 5. Examples of the RSS spatial feature under different number of concurrent senders.

channel utilization. Figure 5 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 5(a), two RSS values (i.e., -74dB, -61dB) are observed to distinguish the two concurrent senders. In Figure 5(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 κ. Then, *Chase*++ clusters all RSS segments to several categories as the rough estimation of the number of concurrent senders.

It is possible that 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}^{max} 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}^{max} 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 ith 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 κ, 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 wth end point E_w. *Chase*++ sorts the elements of G and E in ascending order.

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

$$E = \{E_j | j \in [1, w], |s_{E_j} - s_{E_j + 1}| > \kappa\} \quad (2)$$

Give the start point G_i and end point E_i of the ith RSS segment Seg_i, its RSS sequence is depicted as $\{s_{G_i}, s_{G_i+1}, ..., s_{E_i}\}$.

Algorithm 1 RSS Segment Clustering Algorithm

Input: RSS sequence $\{s_1, s_2, ..., s_n\}$, start point set G and end point set E of w RSS segments.
Output: N_c categories of RSS segments.
1: **for** Each RSS segment Seg_i, $i \in [1, w]$ **do**
2: s_i^{med} = Median($\{s_{G_i}, s_{G_i+1}, ..., s_{E_i}\}$)
3: **end for**
4: Apply Algorithm 2 (threshold constraint clustering) to divide $\{s_1^{med}$, ... , $s_w^{med}\}$ to N_c' categories $C^{rss} = \{C_1^{rss}, C_2^{rss}, ..., C_{N_c'}^{rss}\}$.
5: **for** Each category i, $i \in [1, N_c']$ **do**
6: **if** Give a threshold Δ, $\sum_{j \in C_i^{rss}} |Seg_j| < \Delta$ or given noise floor s_{noise}, $s^{med}(C_i^{rss}) < s_{noise}$. **then**
7: Remove C_i^{rss} from C^{rss}.
8: **end if**
9: **end for**
10: **return** The remained N_c categories in C^{rss}.

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 ith RSS segment, its median RSS value s_i^{med} is the median value of $\{s_{G_i}, s_{G_i+1}, ..., s_{E_i}\}$. At line 4, *Chase*++ divides $\{s_1^{med}$, ... , $s_w^{med}\}$ to N_c' categories with Algorithm 2. The ith category of RSS segments is indicated as C_i^{rss}. For each RSS segment that belongs to C_i^{rss}, C_i^{rss} 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 Δ 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 κ 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 $s^{med}(C_i^{rss})$ of the ith category C_i^{rss} is within the range $[-\kappa, \kappa]$, *Chase*++ adds this RSS segment to C_i^{rss}. If the RSS segment does not belong to any category, *Chase*++ creates a new cluster and adds it to C^{rss}.

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 ith category

Algorithm 2 Threshold Constraint Clustering Algorithm

Input: spatial feature of RSS segments $\{s_1^{med}, ..., s_w^{med}\}$, RSS difference threshold κ.
Output: Category set C^{rss} with N_c' categories.

1: Set C^{rss} as empty.
2: **for** Each segment Seg_i, $i \in [1, w]$ **do**
3:   **if** C^{rss} is empty. **then**
4:     Create segment set $\{i\}$ as C_1^{rss} and add it into C^{rss}.
5:   **else**
6:     Currently, there are x categories in C^{rss}.
7:     **for** Each category C_j^{rss}, $j \in [1, x]$ **do**
8:       **if** $|s_i^{med} - s^{med}(C_j^{rss})| < \kappa$ **then**
9:         Add i to C_j^{rss}, 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_{x+1}^{rss}, add it to C^{rss}.
14:     **end if**
15:   **end if**
16: **end for**
17: **return** Category set C^{rss}.

Algorithm 3 Temporal Feature Checking

Input: Maximum on-air time T_{on-air}^{max}, minimum preamble packet interval $IPPI_{min}$, N_c categories of RSS segments $\{C_1^{rss}, C_2^{rss}, ..., C_{N_c}^{rss}\}$.
Output: The number N_t of concurrent senders detected by temporal features.

1: Initiate N_t as 0.
2: **for** Each category C_i^{rss}, $i \in [1, N_c]$ **do**
3:   **for** Each RSS segment $Seg_{idx_j^{C_i}}$, $j \in [1, N_c^i]$ **do**
4:     $l_{idx_j^{C_i}} = E_{idx_j^{C_i}} - G_{idx_j^{C_i}} + 1$
5:     **if** $j + 1 \leq N_c^i$ **then**
6:       $\pi_{idx_j^{C_i}} = G_{idx_{j+1}^{C_i}} - E_{idx_j^{C_i}}$
7:     **end if**
8:   **end for**
9:   β =MAX$\{l^{C_i}_{idx_j^{C_i}}, \forall j \in [1, N_c^i]\}/T_{on-air}^{max}$
10:   **if** $\beta > 1$ **then**
11:     N_t = N_t + β.
12:   **else if** $\exists j \in [1, N_c^i]$, $\pi^{C_i}_{idx_j^{C_i}} < IPPI_{min}$ **then**
13:     N_t = N_t + 1.
14:   **end if**
15: **end for**
16: **return** N_t

C_i^{rss}, it contains N_c^i RSS segments. Their corresponding index in set $\{Seg_1, ..., Seg_w\}$ is $\{idx_1^{C_i}, ..., idx_{N_c^i}^{C_i}\}$. Hence, the corresponding start and end points are $\{G_{idx_1^{C_i}}, ..., G_{idx_{N_c^i}^{C_i}}\}$ and $\{E_{idx_1^{C_i}}, ..., E_{idx_{N_c^i}^{C_i}}\}$, 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 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 jth RSS segment of the ith category C_i^{rss}, its RSS segment length $l_{idx_j^{C_i}}$ equals $E_{idx_j^{C_i}} - G_{idx_j^{C_i}} + 1$. In line 5, if the $(j+1)^{th}$ RSS segment exists, the RSS segment interval $\pi_{idx_j^{C_i}}$ is $G_{idx_{j+1}^{C_i}} - E_{idx_j^{C_i}}$. For each category, *Chase*++ calculates the temporal features of every RSS segment as shown from line 3 to line line 8. In a category, if the ratio β between the maximum RSS segment length and the maximum on-air time T_{on-air}^{max} is larger than 1 (line 9 and line 10), it is possible the preamble packets of β senders are overlapped. Thus, *Chase*++ adds β to N_t (line 11). Otherwise, if there exists a RSS segment interval is 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_c and refined estimation N_t, the estimated number of concurrent senders is $N_c + N_t$.

D. Adaptive Preamble Packet Generator

In this module, *Chase*++ defines an empirical metric CCR (Channel Capacity Redundancy) to determine the batch size λ. CCR reflects the remained channel resource that allows how long preamble packet can be transmitted. $IPPI_{max}$ 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 δ, which considers the spacial efficiency of capture effect. The large δ indicates good spacial diversity.

$$CCR = \text{MAX}(\delta \times \frac{IPPI_{max}}{N_c + N_t}, T_{on-air}^{max}) \tag{3}$$

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

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

$$\lambda = \text{MAX}(1, \lfloor \frac{CCR - l_{mac} \times B}{l_p \times B} \rfloor) \tag{4}$$

The minimum λ equals to 1. As shown in Figure 4, after obtaining the batch size λ, *Chase*++ further batches λ 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_{cw} of contention window. After receiving the requirement packet, a neighbor node will initiate a *Chase*++ broadcast with a random backoff

TABLE I
THE SUMMARY OF SYSTEM PARAMETER SETTINGS

Parameter	Description	Value
κ	RSS threshold of segmentation and clustering	3 dB
Δ	threshold of RSS samples of a category	3
T_{on-air}^{max}	the maximum on-air time	4096μs
$IPPI_{min}$	the minimum preamble packet interval	4ms
$IPPI_{max}$	the maximum preamble packet interval	12ms
l_{mac}	the length of MAC header and footer	13 bytes
B	CC2420 radio bandwidth	250kbps
l_{cw}	initial length of contention window	20ms

TABLE II
THE PERFORMANCE COMPARISON BETWEEN LR AND LT CODING

	Encoding Time (ms)	Decoding Packet #	Ave. Tail Length (ms)
LR Coding	2	9.1	97
LT Coding	1	9.9	121

in the range of $[0, l_{cw}]$. 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.

IV. IMPLEMENTATION AND EVALUATION

We implement *Chase*++ on TelosB node with TinyOS 2.1.2. We implement the default Box-MAC [10] and *Chase* [12] as the layer of LPL concurrent broadcast. Some system parameter settings are shown in Table I. κ and Δ are empirically set. $IPPI_{min}$ is measured in Section IV-B. T_{on-air}^{max}, l_{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.

We use two testbeds, local testbed (Section II) and Indriya testbed [16], 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 III-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 δ of CCR is set as 1. CCR is 1.2 ms in the situation. The batch size λ is 1, 3, 4, 6 for payload block size 20, 10, 8, and 5,

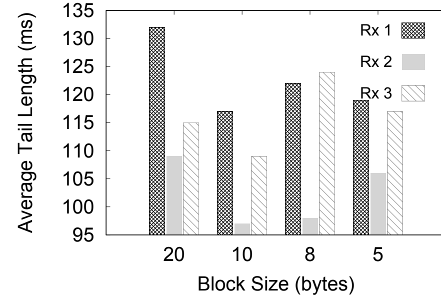


Fig. 6. The performance comparison with different settings of short payload block size.

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 6. 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 need 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 is needed, the more preamble packets is 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. We take the same experiment setting of Section IV-A and set l_p as 10 bytes. For LT code, the encoding coefficient matrix is offline generated according to [15]. Its size is 51×32. Hence, extra 204 bytes storage is needed in LT code. For LR code, the encoding coefficient matrix is generated in real time. We compare the performance between these two schemes in terms of encoding time, the number of received preamble packet 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. 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 encoding efficiency, tail length is much longer since the decoding inefficiency. Thus, we prefer to use LR code scheme in *Chase*++. The $IPPI_{min}$ is set as 4ms to contain the encoding time.

C. Concurrent Sender Estimation

We evaluate the accuracy of the concurrent sender estimation on 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

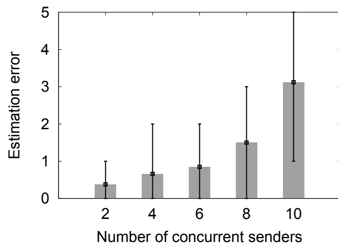
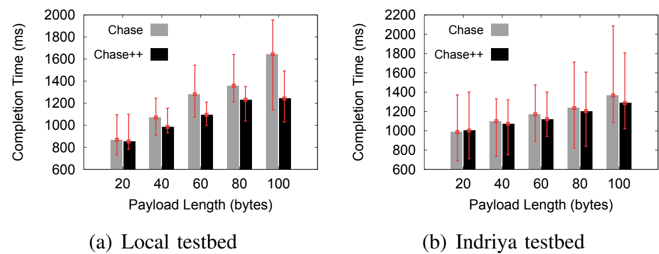


Fig. 7. The estimation error with different number of concurrent senders.



(a) Local testbed

(b) Indriya testbed

Fig. 8. The comparison of completion time between *Chase*++ and *Chase* with different payload length on different testbeds.

TABLE III
THE PERFORMANCE COMPARISON WITH DIFFERENT δ OF CCR METRIC.

Avg. Tail Length(ms)	Coefficient δ Value			
	0.5	1	1.5	2
Local Testbed	148	83.2	99.6	91.1
Indriya Testbed	167.2	121.9	97.2	78.9

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 7 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.

D. CCR Coefficient δ

In the calculation of CCR metric (Equation 3), the coefficient δ indicates the spacial 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 λ 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 spacial diversity, it can further tolerant 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.

The reliability of all experiments can reach 100% coverage on both local testbed and Indriya testbed. Figure 8 shows the distribution of completion time with different payload length. On both local testbed (Figure 8(a)) and Indriya testbed (Figure 8(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.

V. Related Work

Wireless network flooding has been widely studied in the last decade. Many protocols respectively leverage trickle timer [17], duplicate suppression [7], link quality [8], constructive interference [4] [18] 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 [19] [3] [20] (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 [21], link correlation aware flooding[22] and duty cycle aware broadcast [23] [24] 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 [25] develops a sophisticated radio, which can be always on and consume ultra low power to quickly sense the ongoing flooding. Without modified hardware, *Chase* [12] 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 [13] are widely used to improve the efficiency of binary image dissemination. Rateless-Deluge [26] and SYNAPSE++ [15] use Fountain code to improve the performance of Deluge [7]. Pando [5] uses Fountain code to resolve the long-tail problem in constructive interference based code dissemination Splash [18]. All these protocols assume the radios of all nodes are always-on. They are hard to directly be adopted in asynchronous duty cycle networks. *Chase*++ first introduces Fountain code to accelerate network flooding in asynchronous duty cycle networks.

VI. 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 programs under Grant 61472217, 61472219, 61472211, 61722210, 61572277, 61472382, 61379117, 61502271, 61432015, the NSFC key program under Grant 61532012 and the NSFC Joint Research Fund for Overseas Chinese Scholars and Scholars in Hong Kong and Macao under grant 61529202. Jiliang Wang is the corresponding author.

References

[1] C. Zhang, A. Syed, Y. Cho, and J. Heidemann, "Steam-powered sensing," in *Proceedings of Sensys*, 2011.

[2] X. Mao, X. Miao, Y. He, X.-Y. Li, and Y. Liu, "Citysee: Urban co 2 monitoring with sensors," in *Proceedings of INFOCOM*, 2012.

[3] M. Zimmerling, F. Ferrari, L. Mottola, T. Voigt, and L. Thiele, "ptunes: Runtime parameter adaptation for low-power mac protocols," in *Proceedings of IPSN*, 2012.

[4] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh, "Efficient network flooding and time synchronization with glossy," in *Proceedings of IPSN*, 2011.

[5] W. Du, J. C. Liando, H. Zhang, and M. Li, "When pipelines meet fountain: Fast data dissemination in wireless sensor networks," in *Proceedings of Sensys*, 2015.

[6] L. Huang and S. Setia, "Cord: Energy-efficient reliable bulk data dissemination in sensor networks," in *Proceedings of INFOCOM*, 2008.

[7] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proceedings of Sensys*, 2004.

[8] W. Dong, Y. Liu, C. Wang, X. Liu, C. Chen, and J. Bu, "Link quality aware code dissemination in wireless sensor networks," in *Proceedings of ICNP*, 2011.

[9] Z. Li, M. Li, and Y. Liu, "Towards energy-fairness in asynchronous duty-cycling sensor networks," *ACM Transactions on Sensor Networks*, vol. 10, no. 3, p. 38, 2014.

[10] D. Moss and P. Levis, "Box-macs: Exploiting physical and link layer boundaries in low-power networking," *Technical Report SING-08-00, Stanford*, 2008.

[11] X. Zheng, Z. Cao, J. Wang, Y. He, and Y. Liu, "Interference resilient duty cycling for wireless sensor networks under co-existing environments," *IEEE Transactions on Communications*, 2017.

[12] Z. Cao, D. Liu, J. Wang, and X. Zheng, "Chase: Taming concurrent broadcast for flooding in asynchronous duty cycle networks," *IEEE/ACM Transactions on Networking*, 2017.

[13] D. J. MacKay, "Fountain codes," *IEE Proceedings-Communications*, vol. 152, no. 6, pp. 1062–1068, 2005.

[14] M. Luby, "Lt codes," in *Proceedings of FOCS*, 2002.

[15] M. Rossi, N. Bui, G. Zanca, L. Stabellini, R. Crepaldi, and M. Zorzi, "Synapse++: Code dissemination in wireless sensor networks using fountain codes," *IEEE Transactions on Mobile Computing*, vol. 9, no. 12, pp. 1749–1765, 2010.

[16] M. Doddavenkatappa, M. C. Chan, and A. L. Ananda, "Indriya: A low-cost, 3d wireless sensor network testbed," in *TRIDENTCOM*, 2011.

[17] P. A. Levis, N. Patel, D. Culler, and S. Shenker, *Trickle: A self regulating algorithm for code propagation and maintenance in wireless sensor networks*. Computer Science Division, University of California, 2003.

[18] M. Doddavenkatappa, M. C. Chan, and B. Leong, "Splash: fast data dissemination with constructive interference in wireless sensor networks," in *Proceedings of NSDI*, 2013.

[19] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele, "Low-power wireless bus," in *Proceedings of Sensys*, 2012.

[20] L. Zhe-tao, Q. Chen, Z. Geng-ming, C. Young-june, and H. Sekiya, "A low latency, energy efficient mac protocol for wireless sensor networks," *IJDSN*, 2015.

[21] S. Guo, L. He, Y. Gu, B. Jiang, and T. He, "Opportunistic flooding in low-duty-cycle wireless sensor networks with unreliable links," *IEEE Transactions on Computers*, vol. 63, no. 11, pp. 2787–2802, 2014.

[22] S. Guo, S. M. Kim, T. Zhu, Y. Gu, and T. He, "Correlated flooding in low-duty-cycle wireless sensor networks," in *Proceedings of ICNP*, 2011.

[23] Y. Sun, O. Gurewitz, S. Du, L. Tang, and D. B. Johnson, "Adb: an efficient multihop broadcast protocol based on asynchronous duty-cycling in wireless sensor networks," in *Proceedings of Sensys*, 2009.

[24] S. Lai and B. Ravindran, "On multihop broadcast over adaptively duty-cycled wireless sensor networks," in *Proceedings of DCOSS*, 2010.

[25] F. Sutton, B. Buchli, J. Beutel, and L. Thiele, "Zippy: On-demand network flooding," in *Proceedings of Sensys*, 2015.

[26] A. Hagedorn, D. Starobinski, and A. Trachtenberg, "Rateless deluge: Over-the-air programming of wireless sensor networks using random linear codes," in *Proceedings of IPSN*, 2008.