Concurrent Transmissions for Communication Protocols in the Internet of Things

Martina Brachmann
Embedded Systems Lab
TU Dresden, Germany
martina.brachmann@tu-dresden.de

Olaf Landsiedel
Computer Science and Engineering
Chalmers University of Technology, Sweden
olafl@chalmers.se

Silvia Santini
Embedded Systems Lab
TU Dresden, Germany
silvia.santini@tu-dresden.de

***Abstract*—Standard Internet communication protocols are key enablers for the Internet of Things (IoT). Recent technological advances have made it possible to run such protocols on resource-constrained devices. Yet these devices often use energy-efficient, low-level communication technologies, like IEEE 802.15.4, which suffer from low-reliability and high latency. These drawbacks can be significantly reduced if communication occurs using concurrent transmissions – a novel communication paradigm for resource-constrained devices. In this paper, we show that Internet protocols like TCP/UDP and CoAP can run efficiently on top of a routing substrate based on concurrent transmissions. We call this substrate LaneFlood and demonstrate its effectiveness through extensive experiments on Flocklab, a publicly available testbed. Our results show that LaneFlood improves upon CXFS – a representative competitor – in terms of both duty cycle and reliability. Furthermore, LaneFlood can transport IoT traffic with an end-to-end latency of less than 300 ms over several hops.**

***Index Terms*—Wireless Sensor Networks; Internet of Things; Concurrent Transmissions**

I. Introduction

One-to-one traffic is becoming an essential building block for applications in the Internet of Things (IoT): For example, the Constrained Application Protocol (CoAP) [1], UDP and TCP are widespread one-to-one protocols in the IoT [2], [3], [4]. Traditionally, these are deployed on RPL [5], which provides one-to-many and one-to-one routing.

In this paper, we depart from using a traditional routing protocol, such as RPL [5], and introduce LaneFlood as routing substrate for one-to-one traffic in the IoT. LaneFlood exploits fast network-wide flooding and concurrent transmissions [6] to efficiently establish a path between any source and destination in the network. Once the path is established, LaneFlood involves only the nodes along that path in the forwarding of data while all others nodes enter deep sleep states and periodically wake up to be available for further connections.

LaneFlood is tailored for IoT applications and fully supports their protocols including CoAP, UDP/TCP, IPv6, and 6LoWPAN. The design of LaneFlood reflects their communication patterns: It sets up connections between any two nodes in the network and then facilitates the data exchange. For example, a CoAP request message will trigger a connection setup, and LaneFlood maintains this connection until it is either closed or has timed out.

LaneFlood does not maintain routes by selecting parent nodes, announcing routing metrics, discovering neighbors and maintaining routing tables as traditional routing protocols. Instead, we build upon Glossy [6] and the Concurrent Transmissions Forwarder Selection (CXFS) protocol [7], recent work on concurrent transmissions and capture. Compared to existing approaches, LaneFlood provides three key advantages: (1) Low idle cost, (2) full and transparent integration into the IoT protocol stack, and, (3) native support for one-to-one communication.

We show the feasibility of LaneFlood as routing substrate for one-to-one communication in the IoT. In particular, we make the following contributions:

- We provide a robust and energy efficient unicast and broadcast communication protocol, called LaneFlood, that relies on concurrent transmissions, has no routing overhead, and does not require neighbor discovery;
- We design LaneFlood to be transparent to the upper layers of the protocol stack, including network, transport or application layer and thus enable the use of multiple request-response-based protocols (e.g. CoAP, TCP);
- We show through extensive experiments on the Flocklab testbed [8] that LaneFlood improves CXFS in terms of duty cycle and reliability and that it can transport IoT traffic with a latency of less than 300 ms over several hops.

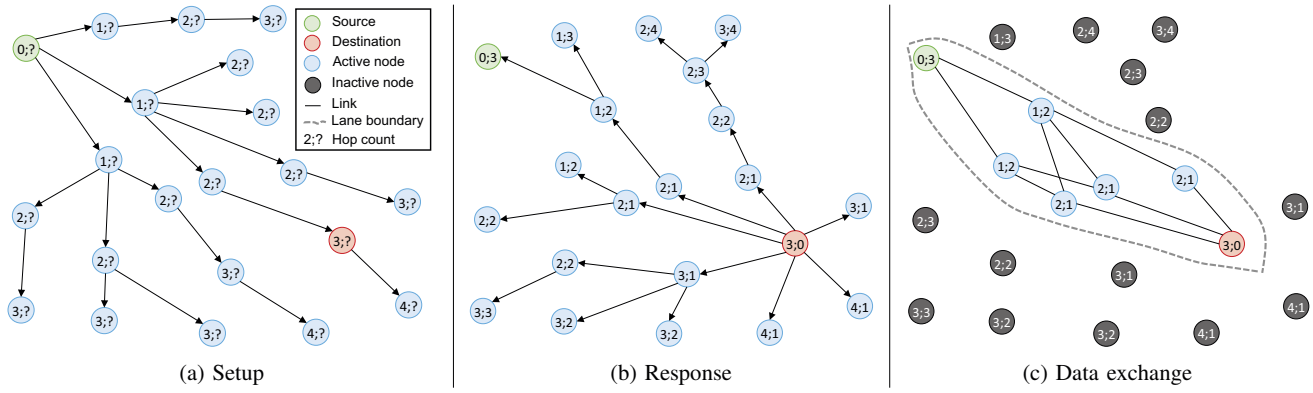
The source code of LaneFlood, along with additional resources, is publicly available at: `https://github.com/martinabr/laneflood`.

The remainder of this paper is structured as follows. In Sec. II, we provide a brief overview of LaneFlood and distinguish it from the state of the art. Next, Sec. III details on the design of LaneFlood. We evaluate LaneFlood on a testbed in Sec. IV and present related work in Sec. V. Sec. VI discusses future work and concludes the paper.

II. LaneFlood: An Overview

LaneFlood creates an exclusive communication channel – a *lane* – between a source and a destination with all currently active nodes in the path serving as forwarders. The lane is established through two consecutive network floods initiated by the source and the destination. As illustrated in Fig. 1a the first flood is the *Setup flood*, which is initiated by the source.

© 2016, Martina Brachmann. Under license to IEEE.
DOI 10.1109/LCN.2016.69



(a) Setup (b) Response (c) Data exchange

Fig. 1: Basic idea behind LaneFlood: We utilize two network-wide floods (based on Glossy) to establish a lane within the network. The first flood is initiated by the source (Setup) and the second by the destination (Response). When the lane is established, nodes that are outside the lane enter sleep states, while the other nodes remain active (Data exchange).

The second flood is initiated by the destination and is called *Response flood*, as shown in Fig. 1b.

Using the information collected during the Setup and the Response floods, nodes can autonomously decide whether they belong to the lane or not. In the first case, they remain active to enable the data exchange between the source and the destination. Otherwise, they turn their radios off until their participation in the next lane construction is required. While a lane is active, source and destination exchange packets through *lane floods*, i.e., network floods propagated by active nodes only. Fig. 1c illustrates a lane within a network.

Network and lane floods are executed "à la Glossy". Glossy [6] is a network flooding protocol that exploits concurrent transmissions to achieve reliable and fast dissemination of packets in a multi-hop network. It assumes there is an initiator in the network that first transmits the packet to disseminate – which in our case is either the source or the destination. Glossy assumes that all nodes in the network participate in the flood. In LaneFlood, floods used to establish a lane also involve all nodes. During data exchange, however, only nodes within a lane remain active and participate in the floods, as illustrated in Fig. 1c. Nodes outside the lane switch their radio off to save energy.

To let nodes decide whether they should remain active during a data exchange or not, LaneFlood builds upon the forwarder selection strategy proposed for the CXFS protocol [7]. CXFS determines the length – measured in number of hops – of existing paths between the source and the destination. It then selects the nodes on minimal paths as forwarders. Nodes on non-minimal paths can also be selected, to improve robustness.

LaneFlood improves upon CXFS in three ways. First, LaneFlood's forwarder selection mechanism makes nodes decide whether they belong to a lane or not using a partially randomized approach, which we describe in detail in the next section. Second, LaneFlood is implemented in Contiki [9] and can operate on common resource-constrained platforms – like, e.g., the Tmote Sky [10]. Instead, CXFS is implemented in TinyOS [11] and adapted to run on a dedicated platform – the "Bacon" mote [7] – which features a high-precision timer capture module. Low-power hardware platforms are typically equipped with cheap and imprecise system clocks [12], [13]. Thus, refraining from assuming the presence of a high precision clock guarantees LaneFlood's portability to a large number of hardware platforms. Third, LaneFlood enables the seamless operation of standard protocols like TCP/UDP and CoAP. Instead, CXFS [7] supports only proprietary transport protocols.

A recently presented approach also shows that Glossy-like floods can be used to relay messages from high-level, standard Internet protocols [14]. However, this approach relies on a central entity that schedules communication. Instead, in LaneFlood connections are established spontaneously between arbitrary source and destination nodes. Furthermore, Hewage et al. [14] do not consider any forwarder selection mechanism in their approach. Their main focus is on maximizing throughput. Instead, LaneFlood can trade-off energy consumption with achievable throughput and reliability. Further, while Hewage et al. [14] evaluate their protocol on a small-scale, proprietary testbed, we tested the performance of LaneFlood on both a well-established simulator (Cooja [15]) and a large, publicly available testbed (Flocklab [8]).

III. LaneFlood: A Closer Look

LaneFlood enables fast and reliable communication in IEEE 802.15.4 networks and ensures their interoperability with the Internet. Traffic from and to the IEEE 802.15.4 network flows through the *initiator* node, which acts as a border router.

LaneFlood is a time-slotted protocol and organizes its operation in *sessions* T_s. Each session consists of several *rounds* T_r. Fig. 2 shows an example of a session that contains six rounds. A round is further split into two parts. In the first part of each round the nodes keep their radio transceivers off. In the meanwhile, the application(s) running on the nodes can operate (e.g., perform computations or collect sensor data). The second part of each round is reserved for communication and we accordingly refer to it as the *communication slot* T_d. In each slot, a network or lane flood is executed. A network

flood provides a network-wide broadcast while a lane flood implements unicast communication. A short guard period T_g at the end of each session ensures that all nodes are ready to wake up in the next session.

At startup, nodes running LaneFlood behave similarly as in Glossy [6]. They keep their radio on to synchronize with the initiator. Once synchronized, they start participating in network and lane floods.

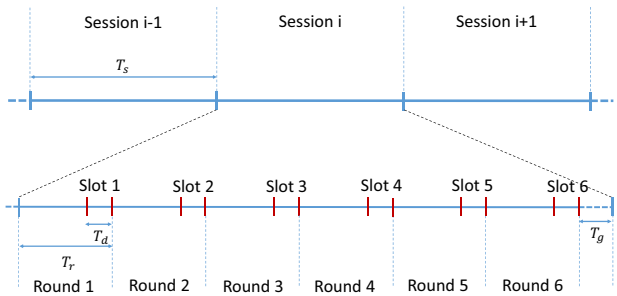


Fig. 2: A LaneFlood session with six rounds.

A. Establishing a connection (Part 1)

To illustrate how LaneFlood operates, we first consider the case in which the initiator must retrieve data from a specific node in the network. This corresponds to the case in which, e.g., a CoAP client running on an Internet host requests sensor data from a node within the IEEE 802.15.4 network. To relay the request from the Internet host, the initiator establishes a connection with the intended recipient. To this end, it transmits a *Setup message* during the first round of a session. The first communication slot of a session is reserved for the initiator, so there is no contention for this slot. A Setup message is disseminated to all nodes through a network flood.

The header of a Setup message carries several fields. The field *source* indicates the identifier of the sender of the message, in this case the initiator. The field *destination* indicates the identifier of the intended recipient of the message – which in our example is the node from which the initiator must retrieve data. Both the *source* and *destination* fields carry 1-byte addresses that are assigned locally to the nodes.

The field *relay_counter* is set to 0 by the initiator and increased each time the message is retransmitted during a flood, as in [6]. Every node that receives the Setup message uses this counter to determine its distance d_{sf} ("f" stands for "forwarder") – measured in hops – to the source, as done in [7]. Fig. 1a depicts a network flood initiated by the source node. The figure also shows how nodes learn their distance in hops to the source. When the Setup message arrives at the destination, the field *relay_counter* indicates the distance in hops between source and destination, d_{sd}. In the example of Fig. 1a $d_{sd} = 3$.

In the second round, the destination of the Setup message sends a *Response message*. The format of a Response message is the same as that of a Setup message. In our example, the destination sets the *source* field of the Response message with its identifier and the *destination* field with the address of the initiator. The *distance* field contains the value of d_{sd} – which the destination retrieves from the *relay_counter* of the Setup message. When a node receives the Response message it can determine its distance in hop to the destination, d_{df}, by reading the value of the *relay_counter* [7], as depicted in Fig. 1b.

At the end of the second round all nodes that have received both the Setup and Response messages know the values of d_{sd} and their own values of d_{sf} and d_{df} [7]. Setup and Response messages are disseminated using Glossy network floods and all nodes in the network are thus expected to receive them with high probability and very low latency [6]. Every node then uses these values to determine whether it belongs or not to the current lane, as depicted in Fig. 1c and detailed below.

B. Creating a lane

A node always belongs to the current lane, if:

$$d_{sf} + d_{df} \leqslant d_{sd} + s_i, \tag{1}$$

whereas s_i is the integer part of a tunable protocol parameter: the *slack* s. Condition 1 makes nodes belong to the lane that are (a) on minimal paths between the source and the destination, and (b) on non-minimal paths that are at most s_i hops longer than the minimal path. This corresponds to the forwarder selection strategy used in CXFS [7].

Making the lane include only nodes on minimal paths might indeed cause Glossy floods to become unreliable. Adding nodes to the minimal path by following Condition 1 helps to improve the performance, but might in turn cause an unnecessarily high number of forwarders being active. While the optimal value of s_i depends on the specific network topology, we have observed that in practical settings small values of s_i (e.g., 1 or 2) can significantly improve the reliability of network floods with respect to the case in which $s_i = 0$. This hints to the fact that selecting nodes that are "somewhere in-between" s_i and $s_i + 1$ might allow to save energy without sacrificing performance. We thus consider the values of s_i to be small and add additional nodes to the lane that fulfill the following composite condition:

$$\begin{aligned} &[d_{sd} + s_i < d_{sf} + d_{df} \leqslant d_{sd} + s_i + 1] \\ &\quad AND\ [rand(0,1) \leq s_d], \end{aligned} \tag{2}$$

whereas s_d represents the fractional part of the slack s. If, e.g., $s = 2,3$, then $s_i = 2$ and $s_d = 0,3$. Condition 2 implies that nodes that are on paths $s_i + 1$ hops longer than the minimal path are part of the lane with probability s_d. This allows us to improve performance by increasing the number of nodes that participate in a lane in a far more fine-grained way compared to just considering integer values of the slack. This, in turn, allows us to save energy without sacrificing performance. The value of the slack is included in the homonymous field of Setup and Response messages and can thus be disseminated to all nodes in each session.

C. Exchanging data

In the round that follows the dissemination of a Response message, nodes within the lane wake up. All other nodes keep their radios switched off until the beginning of the next session.

The source and destination nodes continue their communication by exchanging *Data messages*. Data messages have the same format as Setup and Response messages and can carry up to 119 bytes of data payload.[1] Packet fragmentation is dealt with by upper-layer Internet protocols (e.g., 6LoWPAN), as discussed below. In each round, a single Data message is disseminated using lane floods.

When a data exchange ends before the end of a session, forwarder nodes keep on listening to potentially incoming messages for two rounds. They then switch themselves off and wake up again when the next session starts. In other words, a connection is released if no communication takes place for two consecutive rounds.

If the last Data message is transmitted successfully during the last round of a session, there are no idle rounds and all nodes wake up again in the next round, which is the first of the new session. If a data exchange is not completed when the session ends, initiator and destination continue the data exchange when the new session starts.

Irrespectively of whether and when a data exchange ends within a session, when the new session starts nodes compete to establish a new connection, as described below.

D. Establishing a connection (Part 2)

When a new session starts, all nodes wake up and listen to incoming messages from the initiator. If the initiator does not have any pending requests – i.e., if it does not need to send any Setup message – it disseminates a *Sync message*.

This message guarantees that all the nodes in the network re-synchronize their clocks to that of the initiator. It also signals to all nodes that connections to the initiator or to any other node in the network can be established in the next round. In the subsequent round, a node that has data to send or retrieve can thus transmit a Setup message to its intended destination. For instance, if the destination in our previous example could not relay all its Data messages, it would try to setup a new connection with the initiator in the new session.[2]

However, several nodes may want to setup a connection and they thus all transmit a Setup message in the same round. These messages compete against each other with three possible outcomes: (1) All Setup messages are lost; (2) Only one of the Setup messages reaches its intended destination; (3) Two or more Setup messages reach their intended destination.

In the first case, no Response message is disseminated in the next round. Nodes that had sent a Setup message recognize this situation and send their Setup message again in the subsequent round. In the second case, only one Response message is disseminated in the next round. A connection between a source and destination-pair is thus successfully established and a data exchange can be performed as described above. The senders of failed Setup messages recognize that another node has won the current session and either switch themselves off or act as forwarders. In the third case, two or more competing Response messages are sent concurrently. Again, there are three possible outcomes: (1) All Response messages are lost; (2) Only one of the Response messages reaches its intended destination; (3) Two or more Response messages reach their intended destination.

The first case results in the same behavior as if all Setup messages were lost. Senders of Setup messages recognize the absence of a Response message and attempt to send their Setup messages again in the next round.[3] In the second case, a connection between a single source and destination-pair is established and all nodes in the network evaluate whether to remain active in a lane or go back to sleep. In the third case, several connections are established concurrently. Since both Setup and Response messages have been successfully delivered, it is likely that several lanes can co-exist at the same time in the network. We reserve it to our future work to investigate experimentally how well LaneFlood can cope with such situations in practical settings.

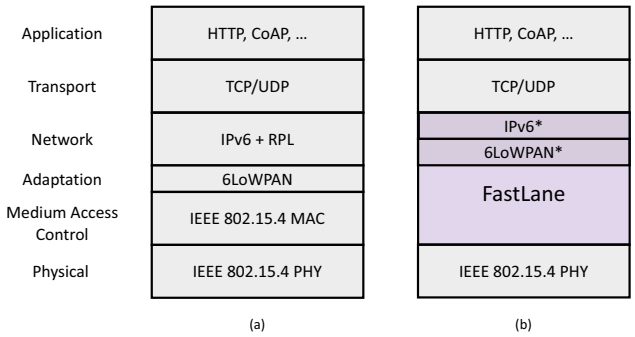


Fig. 3: Network stack of Internet-compliant IEEE 802.15.4 networks (a); Network stack of nodes running LaneFlood (b).

E. LaneFlood's interoperability with Internet protocols

Fig. 3 shows (a) the network stack of an Internet-compliant IEEE 802.15.4 network and (b) the network stack of nodes running LaneFlood. The figure shows that LaneFlood eliminates the need of incorporating routing protocols (like RPL [5]) and any MAC logic on the nodes. Indeed, when Glossy-like floods are used to relay data, no routing or medium access control is needed. Nodes do not need to perform neighbor discovery, maintain routing tables or deal with channel contention. They just "blindly" retransmit packets. Any logic related to IP-routing is not needed either. Thus, nodes running LaneFlood must not load a fully-fledged implementation of IPv6 and 6LoWPAN. "Slimmed" versions of these protocols – indicated as IPv6* and 6LoWPAN* in Fig. 3 – are sufficient. IPv6* and 6LoWPAN* do not contain any logic related to IP routing but keep all

[1] The header of Setup, Response, and Data messages is 8 bytes in total and the maximum physical layer size of a IEEE 802.1.5.4 packet is 127 bytes. Thus, Setup, Response, and Data messages can carry 119 bytes of data payload each.

[2] In the current version of LaneFlood the initiator cannot recognize whether a node has completed relaying its data or not when the session ends. This is because LaneFlood is decoupled from the application layer.

[3] We do not embed a random backoff or similar behavior in the current version of LaneFlood. We reserve it to future work to investigate whether a random backoff or equivalent solution provides better performance than our current "brute force" approach.

components of IPv6 and 6LoWPAN that take care of packet fragmentation, header compression, and IP addresses.

F. LaneFlood's packet queue

Typical IoT protocols like, e.g., CoAP [1], can generate communication requests at any point in time. LaneFlood, on the other hand, operates according to a fixed schedule. To support on-demand traffic from application protocols, LaneFlood uses a dedicated packet queue. This queue receives packets from layers sitting above LaneFlood and we thus refer to it as the *tx queue*. Once a packet is generated by the application and passed down the stack, LaneFlood first enqueues it in the tx queue.

In each round, every node verifies whether it has any packet in its queue. If so, the node further verifies whether it is allowed to transmit in the current round. This is the case when, for instance, a Sync message was transmitted in the previous round. If the tx queue of a node is not empty and the node is allowed to send, it transmits a Setup message with a copy of the first packet from its tx queue as payload. The packet is removed from the queue if and only if – by the end of the current communication slot – the node has received its own packet at least once. This implies that the packet has been forwarded by neighboring nodes and, thus, has most likely also reached the destination. If, instead, the node receives the packet of a contending sender, it passes this packet to the upper layer and keeps its own packet in the tx queue. A new transmission attempt is started in the next allowed round.

Allowing the embedding of application data in Setup and Response messages reduces LaneFlood's control overhead to just a few byte for the header.

The length of the queue is a tunable parameter in LaneFlood. In the current implementation the tx queue can hold 35 packets.

IV. Evaluation

In this section, we evaluate the performance of LaneFlood through extensive simulations and experiments on a publicly available testbed. We show that our forwarder selection strategy improves CXFS in terms of duty cycle and reliability by choosing the number of participating nodes with more granularity. We also find that LaneFlood can transport IoT traffic with an end-to-end latency that can be tuned to less than 300 ms.

A. Methodology

We run LaneFlood on both the Cooja simulator [15] and the Flocklab testbed [8]. We consider a scenario where the initiator, acting as *client*, requests a batch of data (e.g., sensor data, traffic monitoring information) from specific nodes in the network. The client sends requests of 90 byte that are uniformly distributed within a random interval of [1,11] seconds. Thus, a request is sent on average every 5 seconds. The client sends 125 requests to each server, before switching to the next one. We select the node with identifier 1 as our client. The target nodes in the role of *servers* reply immediately with 5 packets and a packet size of 125 byte. Table I shows the server identifiers and their distance to the client. Client and servers are addressed with IPv6 addresses and application messages are transported through UDP. This simulates the scenario in which the requests issued by the initiator are e.g., CoAP requests. LaneFlood is used as underlying communication protocol.

TABLE I: Server settings

Node identifier in Cooja	Node identifier in Flocklab	Distance to client in hops
7	16	2 to 3 (short distance)
2	13	3 to 4 (middle distance)
4	7	4 to 7 (long distance)

We focus on three key performance metrics: Reliability, latency, and duty cycle. The *reliability* is the ratio of received messages and total messages sent by the application without retransmissions. We measure the reliability of packets received at the client. The *latency* is measured end-to-end (at the application level) and describes the time interval between the sending of a packet and its reception. We distinguish between the latency of a packet transmitted from the client to the server, and the latency from server to client. The *duty cycle* indicates the ratio of the total time the radio of a node is on during an experiment and the total duration of the experiment. The duty cycle is averaged over all nodes within the network.

Each experiment runs for 1 hour. Results are averaged over at least 3 runs and error bars in the plots indicate the 5th and 95th percentiles.

B. Impact of the Slack

We first evaluate the impact of the slack s in different network topologies. We set the transmission power P_{tx} to 0 dBm and $s = 0,00$ and increase the slack steadily until all nodes participate in the lane flood. Note that in case all nodes participate in a lane flood, this corresponds to a Glossy-flood, indicated with $s =$ Glossy. In case of using integer values for the slack, e.g., $s = 0,00$, $s = 1,00$, $s = 2,00$, this corresponds to CXFS. We repeat the same experiments with $P_{tx} = -10$ dBm. When $P_{tx} = 0$ dBm, paths between client and servers are shorter and more nodes are within a single hop. This results in a broader lane. Vice versa, when $P_{tx} = -10$ dBm paths between client and servers are longer and less nodes are within a one-hop range. Thus, lanes are typically narrower. In the following, we refer to the network resulting from setting $P_{tx} = 0$ dBm and $P_{tx} = -10$ dBm to the *dense* and *sparse* topology, respectively.

Observation 1: *The higher the slack, the higher the duty cycle*. The first plots in Fig. 4 and 5 show that the average duty cycle increases with higher slack. The upper peak of the error bars indicate the duty cycle of the nodes participating in the lane flood while the lower peaks of the the error bars mark the duty cycle of nodes in sleep state. The impact of the slack on the duty cycle is the same in the sparse and dense topology. This result is expected since with a higher slack more nodes participate in the lane flood and leave their radios on to be able to forward packets. To reduce the duty cycle – and thus increase battery lifetime – a small slack is preferable. We

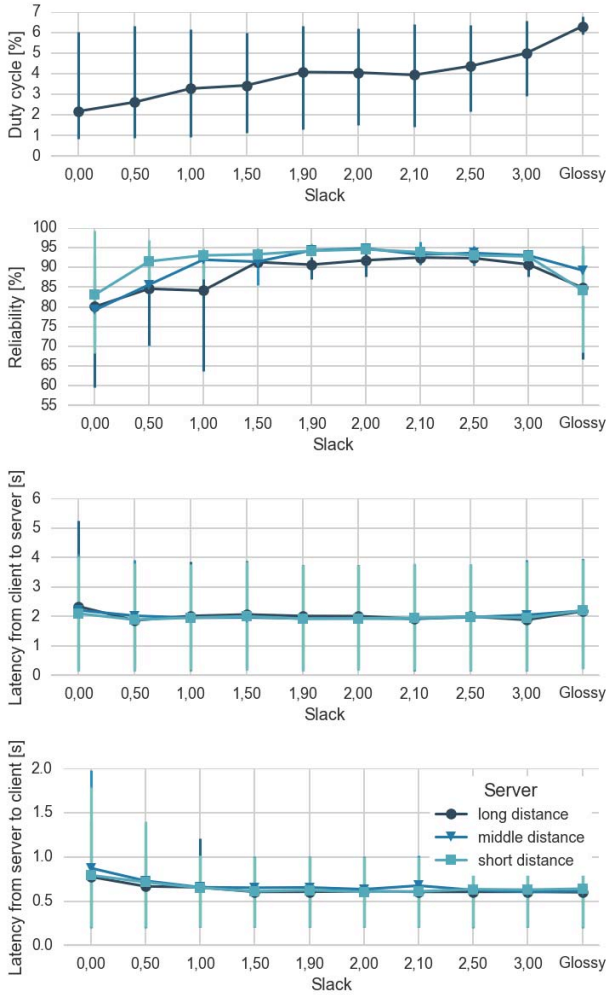


Fig. 4: Impact of the slack in the sparse topology in Flocklab.

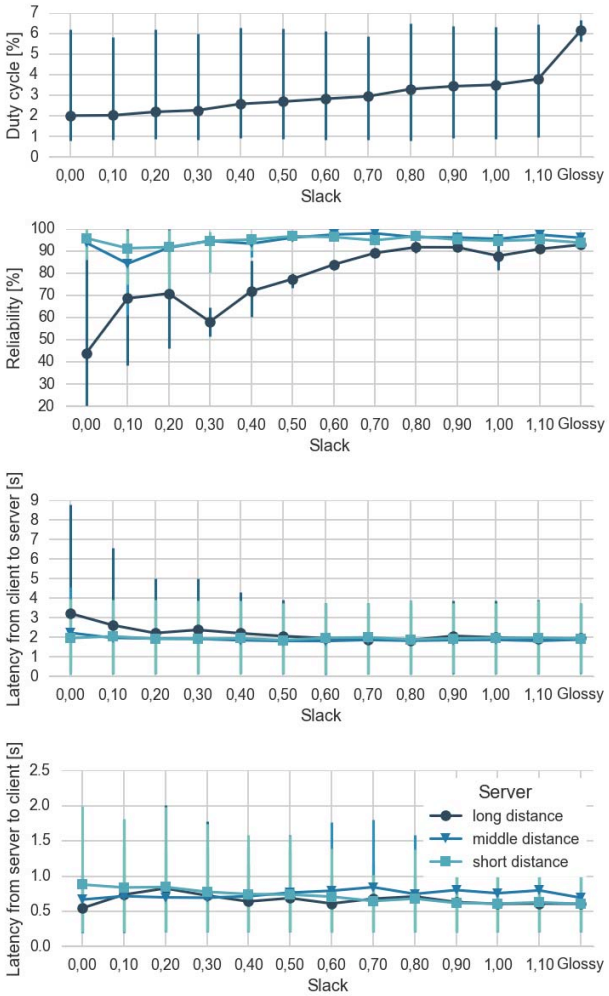


Fig. 5: Impact of the slack in the dense topology in Flocklab.

plan to investigate techniques to adapt the slack at runtime to trade-off duty cycle and reliability.

Observation 2: *A higher slack increases the reliability until its maximum is reached*. The second plots in Fig. 4 and 5 display the impact of the slack on the reliability. The reliability increases until it reaches its maximum. Since more nodes participate in the lane flood and help in forwarding the packets, the reliability increases. The reliability is lower than 100% because it drops logarithmically with the packet size [6]. The smaller the packet, the higher the reliability. Our rationale for sending 125 byte packets is (a) the simulation of big data streams and fragmented data and (b) the increase of the client-to-server latency with smaller packets. Having a fixed size of bytes to send and decreasing the packet size leads to more packets that need to be sent. This increases the client to server latency since only one packet is transmitted in a single round. The reliability reaches its peak in the sparse topology at $s = 2,00$ and in the dense topology at $s = 0,80$. After reaching the peak the reliability either remains constant or even decreases as shown in Fig. 4. At the peak we have reached a saturation of participating nodes. At this point additional nodes just increase the average duty cycle rather than contribute to the reliability. As reported by Chaos [16] and Sparkle [17], it might be that too many concurrently transmitting nodes result in a low signal-to-noise ratio and thus in a lower reliability. The phenomenon of the reliability drop in sparse networks was also observed in Glossy. The reliability drops logarithmically with the network diameter. In a dense network the packets have to pass more hops to reach either source or destination. Increasing the slack and thus, boosting the amount of participating nodes compensates for this effect until reaching node saturation. After that, adding nodes worsens the effect due to the deteriorating signal-noise-ratio caused by many concurrently transmitting nodes.

Observation 3: *The reliability is highest between a minimum and a maximum of participating nodes*. We already mentioned the saturation of participating nodes during a lane flood. There is a second phenomenon which can be seen in Fig. 5 – the reliability drops at $s = 0,10$. We use Contiki's network simu-

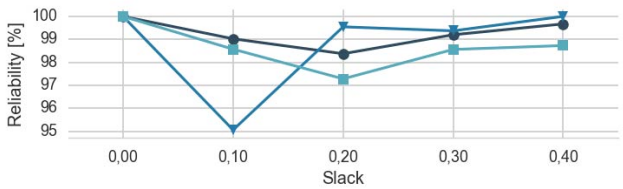


Fig. 6: Reliability in a dense network in Cooja. The reliability drops at s = 0,10 for all servers. In order to concurrently transmit and contribute in the reliability a minimum number of active nodes is required.

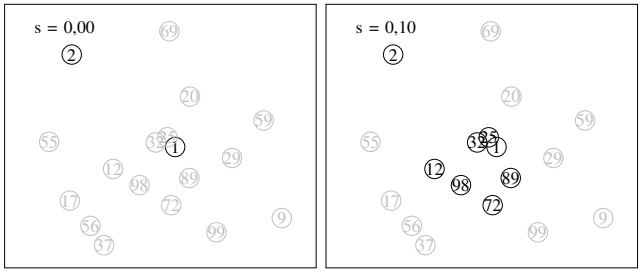


Fig. 7: Exemplary lane between client with identifier 1 and server with identifier 2 in a dense network in Cooja. A slack = 0,00 results in a minimal path, while more nodes participate when slack = 0,10.

lator, Cooja, with the Multi-path Ray-tracer Medium (MRM) communication model to further investigate this phenomenon. We set the network size to 100 nodes, randomly distributed within a 1x1km square. We fix the background noise to -100 dBm with a variance of 60 dB. Table I shows the server identifiers and the distance to the client. The rest of the setting is the same as for Flocklab. Fig. 6 shows that all three servers have a reliability of 100% at $s = 0,00$. However, the reliability drops significantly at $s = 0,10$ to 95% for the middle-distance server. Fig. 7 shows a snapshot of the lane between client and the short distance server with identifier 2. At $s = 0,00$ only source and destination are active, being able to communicate directly. No concurrent transmissions are involved. At $s = 0,10$ additional nodes are added to the lane, which becomes broader, and concurrent transmissions occur between client and server. However, one or more of the additional nodes causes the link quality to decrease, which results in a drop in reliability. A minimum amount of participating nodes and thus, a minimum slack value, are thus necessary to achieve high reliability. The minimum required slack value depends on the topology and the distance between client and server.

Observation 4: *The random slack facilitates a high reliability with a low network duty cycle compared to CXFS*. We have already shown that there is a maximum reliability that we cannot exceed even by adding more nodes to the lane. The goal is thus, to find the maximum reliability with the lowest duty cycle. While the boundary b in CXFS is a fixed integer, our random slack approach allows for a fine-grand selection of nodes participating in a lane. This allows us to achieve the maximum reliability with only the amount of nodes that are necessary to forward the packets between client and server, reducing the energy consumption in the network.

Observation 5: *The best slack value for a client-server-pair depends on the topology*. The best slack value in terms of reliability and duty cycle is determined by the topology and the distance between a client-server-pair. For example, in the dense topology, the short distance server has its highest reliability at $s = 0,50$, the middle distance server at $s = 0,70$ and the long distance server at $s = 0,80$. In the sparse topology short and middle distance servers have their peak at $s = 2,00$ while the long distance server achieves its highest reliability at $s = 2,10$. We, thus, have to assign a slack value for each client-server-pair individually. We leave the implementation of an algorithm that determines the best slack value for each client-server-pair autonomously for future work.

Observation 6: *The latency is not affected by the slack*. As shown by the two lower plots in Fig. 4 and 5, the latency remains almost constant and is not effected by the slack. However, low reliabilities increase the latency, because the nodes need more attempts to transmit their packet. Since only one packet is sent in each round, the latency increase for all enqueued packets in the tx queue. As described below, the error bars mainly indicate, how long a packet was enqueued in the tx queue before successfully transmitting it. The lower peak indicates the latency of the first packet in the tx queue, as it is directly transmitted by LaneFlood.

C. Impact of the session and round length on latency

In this set of experiments we evaluate the impact of the session and round length on the performance of LaneFlood. We use our sparse topology with a transmission power of -10 dBm and set $s = 1,00$. In this configuration we run LaneFlood with round lengths of 100 ms and 200 ms.

Observation 7: *The server to client latency increases with the round length T_r*. Looking at Fig. 8 we find that the server to client latency increases for all servers with the round length while the client to server latency, the reliability and the duty cycle remain constant. This is expected since the latency from server to client is mainly determined by T_r and the position of the packet under consideration. More specific, the server enqueues all data packets in the tx queue before transmission. In our evaluation application, the server creates 5 packets destined to the client. In each round only one packet is transmitted, thus the average server to client latency for the first packet in the tx queue is T_r/2. The average latency from server to client for each following packet increases by T_r assuming the previous packet was transmitted correctly. The 5th packet has, thus, an average server to client latency of $T_r/2 + 4 \cdot T_r$. Minimizing T_r decreases the time until a packet is scheduled and hence, the overall server to client latency.

Observation 8: *Increasing throughput increases duty cycle*. Increasing T_r with a constant interval to send requests has no effect on the duty cycle. Packets in LaneFlood can carry 119 byte of payload. Assuming that data is exchanged in each

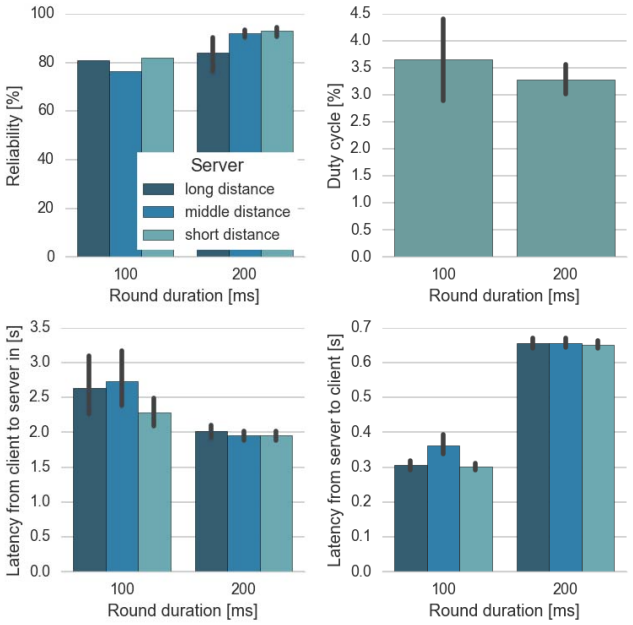


Fig. 8: Increasing the round length T_r increases the latency from client to server.

round with T_r = 200ms, we achieve a best case throughput of 4.76kb/s and with T_r = 100ms the best case throughput is 9.52kb/s. Facilitating more rounds by decreasing T_r forces the active nodes to turn on their radio more often, leading to a higher duty cycle.

Observation 9: *The client to server latency can be reduced by decreasing the session length* T_s. Fig 4 shows that the average client-to-server latency is 2 s. This is expected since the client-to-server latency is mainly influenced by the value of T_s, which we set to 4 s. Since LaneFlood runs completely detached from the application, it issues Setup messages from the client on average every T_s/2 seconds. Reducing T_s thus allows to straightforwardly reduce the client-to-server latency. However, a smaller value of T_s also leads to (a) a higher duty cycle – since Setup and Response messages are exchanged more often – and (b) to less rounds available for data exchange (if T_r is fixed). This, in turn, may results in higher packet drops due to a full tx queue.

V. Related Work

Since its presentation to the research community in 2011, Glossy has been modified and extended in several ways. The Low-power Wireless Bus (LWB) [18] relies on Glossy as an underlying flooding mechanism. LWB schedules communication in a centralized manner to support one-to-many, many-to-one, and many-to-many traffic patterns. Chaos [16] builds upon Glossy to achieve fast and efficient all-to-all data sharing in a completely decentralised manner. Splash [19] adds pipelining to Glossy-like floods to build a full-fledged data dissemination protocol. While Glossy distributes one (typically small) packet in every communication slot, Splash is designed to reliably and quickly deliver large data objects to all nodes in a network. Ripple [20] builds and improves upon Splash in particular in terms of achievable throughput. The approaches listed above differ from LaneFlood because: (1) They do not support one-to-one traffic or do so without forwarder selection; (2) Their interoperability with high-level protocols like TCP/UDP or CoAP has not been discussed nor evaluated.

There exist also protocols like CXFS [7] and Sparkle [17] that build upon Glossy to provide one-to-one communication. The differences between CXFS and LaneFlood have been outlined in detail earlier in this paper. Sparkle [17] also supports one-to-one communication and selects subsets of nodes that participate in Glossy-like network floods. LaneFlood could also operate using Sparkle's forwarder selection mechanism. However, from the results reported in [17] it is unclear whether Sparkle can provide higher end-to-end reliability than CXFS. We plan to evaluate LaneFlood with Sparkle's forwarder selection in our future work. RTF [21] builds upon Sparkle and focuses on improving reliability and energy-efficiency in point-to-point traffic. It uses TDMA for scheduling messages. LaneFlood does not provide any scheduling mechanism as we assume on-demand traffic rather than constant traffic.

A number of communication protocols based on concurrent transmissions, like CXFS [7], rely on proprietary transport protocols to regulate data flows. Instead, LaneFlood supports standard Internet protocols like, TCP/UDP and CoAP. Hewage et al. [14] have recently demonstrated that TCP/IP can run efficiently on top of LWB. In particular, the authors propose two different LWB schedulers to support TCP connections. In LaneFlood, there is no central entity that schedules communication. Furthermore, Hewage et al.'s main focus is on maximizing throughput at the cost of energy and they evaluate their protocol on a small-scale, proprietary testbed. Duquennoy et al. [22] propose a communication primitive called *Burst Forwarding*. They show that they can support standard TCP on top of this primitive. However, Burst Forwarding does not rely on concurrent transmissions and can thus not achieve as high reliability and low latency as Glossy-based protocols.

In summary, the main novelty of LaneFlood with respect to existing work consists in its ability to integrate the benefits of concurrent transmissions to provide for high reliability and very low latency, an efficient forwarder selection mechanism to save energy, and a protocol design that allows it to support standard, high-level Internet protocols. To the best of our knowledge, LaneFlood is the first approach presented in the literature that integrates these three components.

VI. Conclusions and Future Work

In this paper, we argue that one-to-one communication is the most common traffic paradigm in the Internet of Things. CoAP, TCP/UDP are indeed widespread one-to-one protocols in the IoT. We introduce LaneFlood as a routing substrate to support IoT one-to-one protocols. LaneFlood allows IoT applications to depart from using a traditional routing protocol, such as RPL. It relies on fast network-wide flooding and concurrent transmissions to efficiently establish a path between any source

and destination in the network. Once the path is established, LaneFlood involves only the nodes along that path in the forwarding of data while all others nodes enter deep sleep states. All nodes wake up periodically to build further connections. Our experimental results show that LaneFlood improves upon CXFS, its closest competitor, in terms of duty cycle and reliability. We further show that LaneFlood can transport IoT traffic with a latency of less than 300 ms in the Flocklab testbed.

Our future work includes a deeper integration of LaneFlood in the IoT protocol stack. Our goal is to make LaneFlood become a fully-fledged, transparent replacement to RPL. To this end, we will make LaneFlood provide all features of RPL – such as dissemination and security. Further directions for future research include a qualitative and quantitative comparison of LaneFlood and RPL and the analysis of the behavior of LaneFlood in the presence of node failures and mobility.

Acknowledgments

The authors would like to thank the anonymous reviewers for their very valuable comments and the developers of the Flocklab testbed for enabling us to run our experiments. This work has been partially supported by the Research Training Group GRK 1362 *Cooperative, Adaptive and Responsive Monitoring in Mixed-mode Environments* and the Collaborative Research Center SFB 921 *Highly Adaptive Energy-Efficient Computing* both funded by the German Research Foundation.

References

[1] C. Bormann, K. Hartke, and Z. Shelby, "The Constrained Application Protocol (CoAP)," RFC 7252, 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7252.txt

[2] S. Duquennoy, O. Landsiedel, and T. Voigt, "Let the tree Bloom: Scalable Opportunistic Routing with ORPL," in *The ACM Conference on Embedded Networked Sensor Systems (SenSys)*, September 2013.

[3] O. Gaddour and A. Koubâa, "RPL in a Nutshell: A Survey," *Computer Networks*, vol. 56, no. 14, pp. 3163–3178, 2012.

[4] T. Pötsch, K. Kuladinithi, M. Becker, P. Trenkamp, and C. Goerg, "Performance evaluation of CoAP using RPL and LPL in TinyOS," in *IEEE International Conference on New Technologies, Mobility and Security (NTMS)*, May 2012.

[5] A. Brandt, J. Vasseur, J. Hui, K. Pister, P. Thubert, P. Levis, R. Struik, R. Kelsey, T. H. Clausen, and T. Winter, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks," RFC 6550, 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc6550.txt

[6] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh, "Efficient Network Flooding and Time Synchronization with Glossy," in *ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, April 2011.

[7] D. Carlson, M. Chang, A. Terzis, Y. Chen, and O. Gnawali, "Forwarder Selection in Multi-transmitter Networks," in *IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS)*, May 2013.

[8] R. Lim, F. Ferrari, M. Zimmerling, C. Walser, P. Sommer, and J. Beutel, "Flocklab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems," in *ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, April 2013.

[9] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors," in *IEEE Conference on Local Computer Networks (LCN)*, November 2004.

[10] Moteiv, "Tmote Sky datasheet," 2006.

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

[12] A. Willig, "Recent and Emerging Topics in Wireless Industrial Communications: A Selection," *IEEE Transactions on Industrial Informatics*, vol. 4, no. 2, pp. 102–124, May 2008.

[13] A. Willig, K. Matheus, and A. Wolisz, "Wireless Technology in Industrial Networks," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1130–1151, June 2005.

[14] K. Hewage, S. Duquennoy, V. Iyer, and T. Voigt, "Enabling TCP in Mobile Cyber-Physical Systems," in *IEEE International Conference on Mobile Ad hoc and Sensor Systems (MASS)*, October 2015.

[15] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-Level Sensor Network Simulation with COOJA," in *IEEE Conference on Local Computer Networks (LCN)*, November 2006.

[16] O. Landsiedel, F. Ferrari, and M. Zimmerling, "Chaos: Versatile and Efficient All-to-All Data Sharing and In-Network Processing at Scale," in *The ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2013.

[17] D. Yuan, M. Riecker, and M. Hollick, "Making 'Glossy' Networks Sparkle: Exploiting Concurrent Transmissions for Energy Efficient, Reliable, Ultra-Low Latency Communication in Wireless Control Networks," in *International Conference on Embedded Wireless Systems and Networks (EWSN)*, February 2014.

[18] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele, "Low-Power Wireless Bus," in *The ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2012.

[19] M. Doddavenkatappa, M. C. Chan, and B. Leong, "Splash: Fast data dissemination with constructive interference in wireless sensor networks," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2013.

[20] D. Yuan and M. Hollick, "Ripple: High-throughput, reliable and energy-efficient network flooding in wireless sensor networks," in *IEEE International Symposium on a World of Wireless Mobile and Multimedia Networks (WoWMoM)*, June 2015.

[21] J. Zhang, A. Reinhardt, W. Hu, and S. S. Kanhere, "RFT: Identifying Suitable Neighbors for Concurrent Transmissions in Point-to-Point Communications," in *ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM)*, November 2015.

[22] S. Duquennoy, F. Österlind, and A. Dunkels, "Lossy links, low power, high throughput," in *The ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2011.