Chaos: Versatile and Efficient All-to-All Data Sharing and In-Network Processing at Scale

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

Federico Ferrari
Computer Engineering and Networks Laboratory
ETH Zurich
ferrari@tik.ee.ethz.ch

Marco Zimmerling
Computer Engineering and Networks Laboratory
ETH Zurich
zimmerling@tik.ee.ethz.ch

ABSTRACT

An important building block for low-power wireless systems is to efficiently share and process data among all devices in a network. However, current approaches typically split such all-to-all interactions into sequential collection, processing, and dissemination phases, thus handling them inefficiently.

We introduce Chaos, the first primitive that natively supports all-to-all data sharing in low-power wireless networks. Different from current approaches, Chaos embeds programmable in-network processing into a communication support based on synchronous transmissions. We show that this design enables a variety of common all-to-all interactions, including network-wide agreement and data aggregation. Results from three testbeds and simulations demonstrate that Chaos scales efficiently to networks consisting of hundreds of nodes, achieving severalfold improvements over LWB and CTP/Drip in radio duty cycle and latency with almost 100 % reliability across all scenarios we tested. For example, Chaos computes simple aggregates, such as the maximum, in a 100-node multi-hop network within less than 90 milliseconds.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*wireless communication*

Keywords

All-to-all communication, in-network processing, capture effect, synchronous transmissions, wireless sensor networks

1. INTRODUCTION

A wide range of low-power wireless applications and protocols needs to share and process data among all devices in a network. For example, emerging control systems compute the control law in a fully distributed manner inside the network based on all sensor readings [30]; in safety-critical applications, nodes periodically agree on a different radio channel to be resilient to jamming attacks and interference [29]; handing over the leader role in tracking applications requires the exchange of each node's current state and network-wide consensus [1]; and the computation of aggregates involves every node in propagating and processing the query [25].

However, a scheme expressly designed for all-to-all interactions has not yet been proposed. As a workaround solution, all-to-all interactions are commonly split into three sequential phases: (*i*) all-to-one data collection, possibly combined with application-specific data aggregation [32], (*ii*) centralized processing, and (*iii*) one-to-all dissemination of the result. For instance, CTP [15] and Drip [38] can be used for (*i*) and (*iii*), while LWB [13] offers both in a single protocol logic. This sequential approach, however, handles all-to-all interactions inefficiently, yet control decisions often must be made in real-time at low energy costs [1, 30].

This paper introduces Chaos, the first primitive that natively supports all-to-all data sharing in low-power wireless networks. Unlike current approaches, Chaos essentially *parallelizes* collection, processing, and dissemination *inside* the network by building on two main mechanisms:

1. *Synchronous transmissions.* In Chaos, nodes synchronously send the data they want to share. Nodes overhearing these transmissions receive packets with high probability due to the capture effect [22]. Upon reception, nodes merge their own with the received data and transmit the resulting packets again synchronously. An appointed node triggers this process, which continues in a fully distributed manner until all nodes in the network share the same data. Synchronous transmissions are key to the efficiency of Chaos.
2. *User-defined merge operators.* Nodes merge their own data with the received data according to a user-defined merge operator. Chaos allows users to freely program various merge operators, from simple aggregates to complex computations taking tens of thousands of clock cycles to execute. Merge operators both enable and define the meaning of an all-to-all interaction; they are key to the functioning and the versatility of Chaos.

We build Chaos upon Glossy [14] to leverage synchronous transmissions, which have been shown to boost the performance of low-power wireless communications [11, 14]. Synchronous transmissions in IEEE 802.15.4 work due to two physical-layer phenomena. *Power capture* enables a receiver to correctly decode a packet when the received signal from one node is about 3 dB stronger than the sum of the received signals from all other nodes [2, 11]. In addition, the strongest signal must arrive no later than 160 μs after the first weaker signal, corresponding to the air time of the IEEE 802.15.4

SenSys'13, November 11–15, 2013, Roma, Italy.

ACM 978-1-4503-2027-6/13/11 ...$15.00
http://dx.doi.org/10.1145/2517351.2517358.

synchronization header, so the radio locks onto the strongest signal and correctly decodes the packet. While capture occurs irrespective of the content of the colliding packets, *constructive baseband interference* occurs only when the packets are identical and overlap within 0.5 μs [11, 14]. Glossy exploits both phenomena for efficient network flooding.

Transforming Glossy from one-to-all flooding of *identical* packets into Chaos, a primitive for all-to-all data sharing of *different* packets, is challenging for at least two reasons:

- While Glossy minimizes processing during a flood to make the transmissions precisely overlap, Chaos relies on processing when applying the merge operator. This can take as long as a few milliseconds, so the effects of clock drift become noticeable. The required processing may also vary among nodes and change over time, for example, due to different payload sizes and conditional statements in the merge operator. We therefore need to ensure that nodes send synchronously despite significantly longer and possibly varying processing times.
- Since nodes in Chaos transmit different packets, they cannot exploit constructive interference for successful reception and rely only on the capture effect. Although this relaxes the timing requirement for synchronous transmissions compared with Glossy, communication becomes more fragile—the probability of receiving a packet decreases as more nodes send together [24]. To illustrate this aspect, we show in Fig. 1 the results of an experiment on the Indriya testbed [7], where we looked at the packet reception rate (PRR) as the number of synchronous transmitters increases, averaged over four receivers and 16126 packets. The PRR decreases gradually from 0.65 with two transmitters to 0.15 with 15 transmitters. We therefore need to balance the number of synchronous transmitters especially in dense areas to achieve high reliability, while making the information spread quickly in the network for high efficiency.

Our design ensures synchronous transmissions by letting every node execute the same number of clock cycles regardless of the implementation and actual processing time of a merge operator. This gives Chaos users substantial freedom in the implementation of merge operators and allows for sophisticated in-network processing. We demonstrate through testbed experiments on the TelosB platform that Chaos supports merge operators taking tens of thousands of microcontroller unit (MCU) clock cycles to execute, without sacrificing on reliability. This leaves enough time to compute, for example, a fixed-point fast Fourier transform (FFT) over thirty-two 16-bit samples, which takes about 9 milliseconds on a TelosB [31]. We illustrate various applications of Chaos in Sec. 2, including data aggregation, three-phase commit, reliable dissemination, and network-wide consensus.

Our solution to balance the number of synchronous transmitters leverages the fact that, in contrast with Glossy, nodes in Chaos know the approximate degree of completion of an all-to-all interaction, based on the packets they receive from their neighbors. As described in Sec. 3, we exploit this feedback to let a node decide whether to transmit or not, and to keep the packet propagation alive against premature termination. Using our Chaos implementation on TelosB devices, described in Sec. 4, we explore in Sec. 5 how our mechanisms blend together and perform in practice.

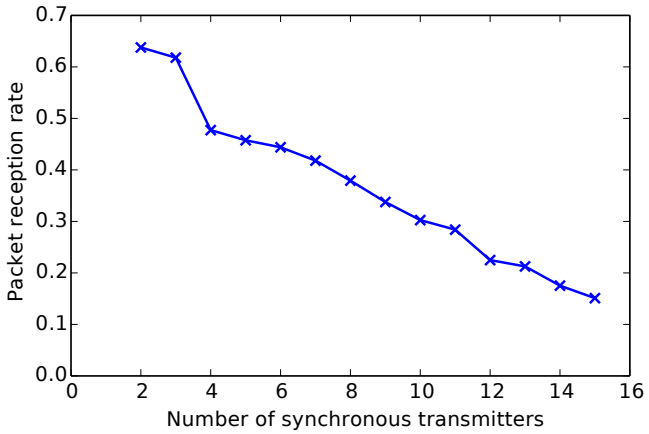


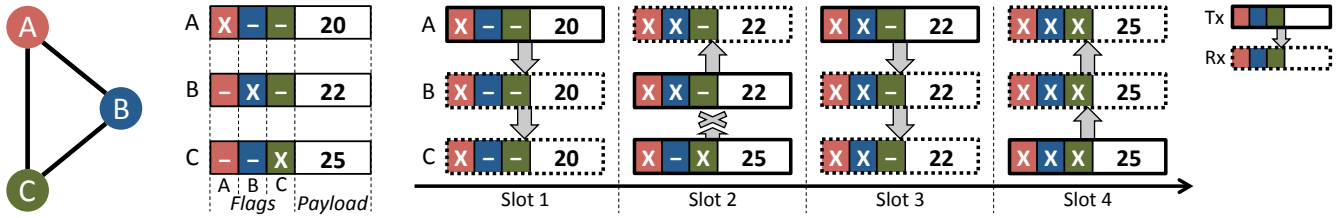
Figure 1: Packet reception rate (PRR) against number of synchronous transmitters, averaged over four receivers and 16126 packets. *The probability of receiving a packet due to capture drops as more nodes send together.*

We evaluate Chaos through both testbed experiments and simulations. We deploy our Chaos implementation on three testbeds to evaluate its performance across a wide range of real-world scenarios in Sec. 7, and to compare Chaos against LWB [13] and CTP/Drip [15, 38] on top of Box-MAC [27] in Sec. 8. We use simulations to study the scalability of Chaos in networks of arbitrary size and node density in Sec. 9. Our results reveal the following main findings:

- Chaos operates efficiently across a variety of network and application scenarios at a reliability that is consistently very close to 100 %. This includes full-size packets, merge operators taking tens of thousands of MCU clock cycles to execute, as well as dense networks where Chaos effectively overcomes the instability of capture.
- Chaos outperforms LWB and CTP/Drip in radio duty cycle and latency. Depending on the network diameter and node density, Chaos improves latency by 20–23× and 3–20× and reduces radio duty cycle by 3–5× and 17–22× over LWB and CTP/Drip, respectively.
- Due to a limitation on the maximum packet size, Chaos needs some form of data compression in order to support more than 1000 nodes in IEEE 802.15.4 networks. Nevertheless, Chaos scales efficiently in simulations to larger networks consisting of several thousands of nodes. This shows the scalability of our mechanisms when applied to other wireless technologies that support larger packets. We particularly believe that Chaos is compatible with Wi-Fi by leveraging its physical-layer capture and Message in Message (MIM) capabilities [21, 26].

Contributions. In summary, we make the following three contributions in this paper:

- We present Chaos, the first primitive providing native support for versatile and efficient all-to-all data sharing in multi-hop low-power wireless networks.
- We demonstrate a working implementation of Chaos that achieves severalfold improvements over the state of the art in the efficiency of all-to-all interactions.
- We show through simulations the scalability of Chaos to multi-hop networks with several thousands of nodes and various node densities.



(a) Nodes A, B, and C are in communication range of each other.

(b) All nodes prepare a packet with the proposed number and their own bit set in the flags.

(c) Node A initiates Chaos by sending its prepared packet in slot 1. B and C merge their own with A's data by taking the larger number and setting the bit of A. Both transmit synchronously in slot 2. A receives from B due to capture, merges data and manipulates the flags, and sends in slot 3. B does not send in slot 4 since it has nothing new to tell. Chaos completes in slot 4 when all nodes are aware of the maximum.

Figure 2: Basic operation of Chaos as three nodes try to find a consensus on the maximum number proposed. *A packet in Chaos consists of the flags, one bit for every node bound to participate in an all-to-all interaction, and the payload. During operation, the flags indicate which nodes already participated and the payload holds an intermediate (or the final) result.*

2. OVERVIEW

We introduce Chaos, the first primitive that natively supports all-to-all data sharing in low-power wireless networks. Chaos integrates in-network processing into a communication support based on synchronous transmissions to incrementally merge and share data among all nodes.

2.1 Basic Operation and Terminology

We begin with a simple example that illustrates the basic operation and benefits of Chaos, and provides some intuition on how applications and protocols may harvest these opportunities. To this end, we consider three nodes that are in communication range of each other, as shown in Fig. 2(a). Every node proposes a number and the goal is to find a consensus among the nodes on the maximum number proposed.

Using Chaos, each node prepares a packet consisting of two parts: *flags* and *payload*. As shown in Fig. 2(b), the flags are three bits, one for every node in the sample network. Initially, a node sets only the bit corresponding to itself and inserts its proposed number as the payload into the packet.

The operation of Chaos commences by letting an appointed node, called *initiator*, send its prepared packet. In our example, node A is the initiator and transmits its packet in *slot* 1 of Fig. 2(c). Because nodes B and C are in A's communication range they both receive the packet. Then, they combine their own data (their proposed numbers) with A's data by applying a *merge operator*. In this case, the merge operator is the `max(x,y)` function, which returns the larger of the two numbers `x` and `y`. Nodes B and C insert the return value of `max(x,y)` as the new payload into their packets and process the flags by setting the bit corresponding to node A.

In slot 2 of Fig. 2(c), B and C transmit *synchronously*. The two packets are different since they carry different flags and payloads. Nevertheless, node A correctly receives B's packet due to capture effects. Using the merge operator, A learns that 22 is the new (intermediate) maximum and sets the bit corresponding to node B. A is the only node transmitting in slot 3, because B and C transmitted in the previous slot.

In slot 4, B *suppresses* its transmission, since its flags are identical to those in the packet it received from A in slot 3. Thus, B has nothing new to tell in slot 4 and stays quiet. As detailed in Sec. 3, Chaos uses such mechanisms to keep the number of synchronous transmitters low, thereby increasing chances at other nodes to receive packets due to capture.

Finally, the *round* completes at the end of slot 4. At this point, all three nodes are aware of the maximum number proposed, because they know from the flags that every node contributed to the consensus. At the end of a round, Chaos notifies the application about the result of the all-to-all interaction, 25 in our example.

The above illustrates that the merge operator both enables and defines the meaning of an all-to-all interaction. Throughout such interaction, the flags indicate which nodes already participated and the payload contains an intermediate result, both of which may differ among nodes before completion. Over time more and more nodes participate, ultimately covering the whole network. Thus, by utilizing flags and payload in the specification of a merge operator, Chaos creates various opportunities for all-to-all data sharing and in-network processing across several hundreds of nodes.[1]

Chaos users are responsible for providing the merge operator, including the mapping of individual nodes to the flags. This is essential because Chaos can only complete if all nodes expected to participate in an all-to-all interaction are indeed part of the network. Chaos takes care of executing the interaction and delivering the result to the user. As described in Sec. 3, Chaos also provides accurate time synchronization, allowing users to schedule Chaos rounds according to their needs. Time-triggered protocols like LWB [13] readily support the scheduling of Chaos, and integrating a synchronous protocol like Glossy or Chaos into an asynchronous stack has been shown to be feasible and highly effective [41].

2.2 Example Applications

The following are examples of real-world applications for which Chaos would provide a solid framework.

Aggregate functions. Aggregation is considered the most common operation in sensor networks [25]. Computing the maximum (or minimum) using Chaos is trivial: nodes merge the flags by taking the bitwise OR and the payload by taking the larger (or smaller) number. Developing merge operators for duplicate-sensitive aggregates (*e.g.*, sum, average, median) is more challenging. This is because a node is likely

[1]IEEE 802.15.4 defines a maximum packet size of 128 bytes, of which 125 bytes can be used for actual data. Our Chaos implementation has an 8-byte header. That is, if the payload is, say, 10 bytes, it supports up to 856 nodes since at most $\lceil 856/8 \rceil = 107$ bytes are available for flags, one bit per node. The use of data compression may overcome this limitation.

to receive a partial aggregate multiple times during a Chaos round, yet every number should contribute only once to the final aggregate. This well-known overcounting problem has been extensively studied in the database and sensor network literature. Particularly, the work of Nath et al. [28] on order- and duplicate-insensitive synopsis diffusion can be applied to compute various aggregates using Chaos.

Unlike most prior aggregation schemes that fuse data toward a collection sink [32], Chaos delivers the final aggregate to *all* participating nodes. Moreover, experiments in Sec. 7 demonstrate that Chaos can aggregate significantly faster and more energy-efficiently than previous approaches.

Network-wide agreement. Low-power wireless nodes often need to agree on important pieces of information, for instance, when electing a new leader [1], switching to a different radio channel [29], or establishing common keys for encryption [9]. To accomplish the latter with Chaos, the initiator could send the IDs of its pre-distributed keys [9] as the payload, and nodes incrementally remove IDs they do not have. In the end, every node knows which keys it shares with all others. Recent work [3] shows how to achieve agreement among one-hop neighbors, whereas Chaos provides the infrastructure to achieve agreement at the network scale.

Three-phase commit and atomic broadcast. As a concrete incarnation of network-wide agreement, Chaos can provide 3-phase commit (3PC) [35]. A round would consist of three sub-rounds, and the flags would have two bits for every node. In the first sub-round, nodes confirm their availability by setting their flag to 1. Nodes authorize the commit in the second sub-round by setting the flag to 2. Finally, they confirm the commit in the third sub-round. If a node fails, its flag does not advance. Using 3PC, Chaos can provide distributed atomic operations, such as atomic broadcast [16].

Reliable data dissemination. In long-term deployments it is often necessary to update code and system parameters as requirements and network conditions change. A reliable dissemination service is needed to propagate those updates to every node, because inconsistencies in code or parameter settings can be detrimental to the system operation [8, 41]. Reliable dissemination with Chaos works by using the data to be distributed as the payload, and the flags as an implicit confirmation whether all nodes received it. Based on the final flags nodes also understand which nodes missed an update and can thus initiate loss recovery without awaiting an explicit request, as required in Deluge [18] or Splash [8].

Multiple communication patterns. Chaos supports multiple communication patterns including all-to-one and all-to-all traffic, where nodes would merge by inserting their data into reserved parts of the payload field. Due to the limited packet size in IEEE 802.15.4, multiple Chaos rounds might be necessary to share data among all nodes, depending on the amount of data and number of nodes in the network. As a concrete example, if nodes want to share one byte of data using our implementation of Chaos, two rounds would be needed on the 139-node Indriya testbed [7], whereas one round would be sufficient on the 90-node Twist testbed [17]. Nevertheless, our experiments in Sec. 8 show that splitting across multiple Chaos rounds in an N-node network is more efficient than N sequential Glossy floods needed in LWB [13].

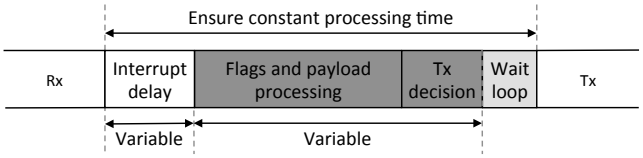


Figure 3: Achieving synchronous transmissions in the face of substantial in-network processing. *Using a wait loop, Chaos accounts for variable interrupt delays and processing times by deferring the transmission for a constant number of MCU clock cycles after the interrupt has occurred.*

3. DESIGN

This section presents the design of Chaos. We begin with the basic communication and in-network processing support, followed by mechanisms required to achieve an efficient and reliable operation in real-world settings.

3.1 Communication and Processing Support

The design of Chaos relies on synchronous transmissions for efficient communication and on merge operators for enabling various all-to-all interactions. As a foundation for the former, we build Chaos upon Glossy [14]. Glossy exploits synchronous transmissions for efficient network flooding; it uses a careful software design to *minimize processing* during a flood, in order to make transmissions overlap within 0.5 μs of each other and thus benefit from constructive interference.

Problem. By contrast, Chaos *relies on processing*. When a node receives a packet, it needs to process its flags and payload according to the merge operator. The required processing time is thus significantly longer than in Glossy and may also vary among nodes and change over time, for example, when the payload size increases throughout a Chaos round or when the branches of a conditional statement (*e.g.*, `if-then-else`) result in different execution times. This poses a significant challenge: Chaos must ensure that nodes transmit synchronously (within 160 μs) despite longer and varying processing times, so receivers can benefit from capture to correctly decode packets with high probability.

Solution. We solve this challenge in Chaos by letting nodes always send a fixed interval after the reception of a packet. As shown in Fig. 3, we need to account for (i) variable delays from when a start of frame delimiter (SFD) interrupt occurs at the end of a reception until the MCU begins to serve the interrupt, and (ii) variable processing times when applying the merge operator.

To account for both (i) and (ii), we use the timer capture functionality of the MCU to measure the time from when the SFD interrupt occurs until when the MCU finishes executing the merge operator. Using a *wait loop* (see Fig. 3), we compensate for this variable time by deferring the upcoming transmission for a constant number of MCU clock cycles after the SFD interrupt. Chaos users have to set the number of clock cycles based on the expected execution time of the merge operator. If the execution takes longer, Chaos suppresses the transmission to avoid spreading erroneous data. Setting the number of clock cycles conservatively can remedy this issue, but may cause a loss in efficiency.[2]

[2]An optimization we do not explore in this paper is to turn off the radio while processing. This would allow users to set the number of clock cycles conservatively without sacrificing energy, but latency would be affected nevertheless.

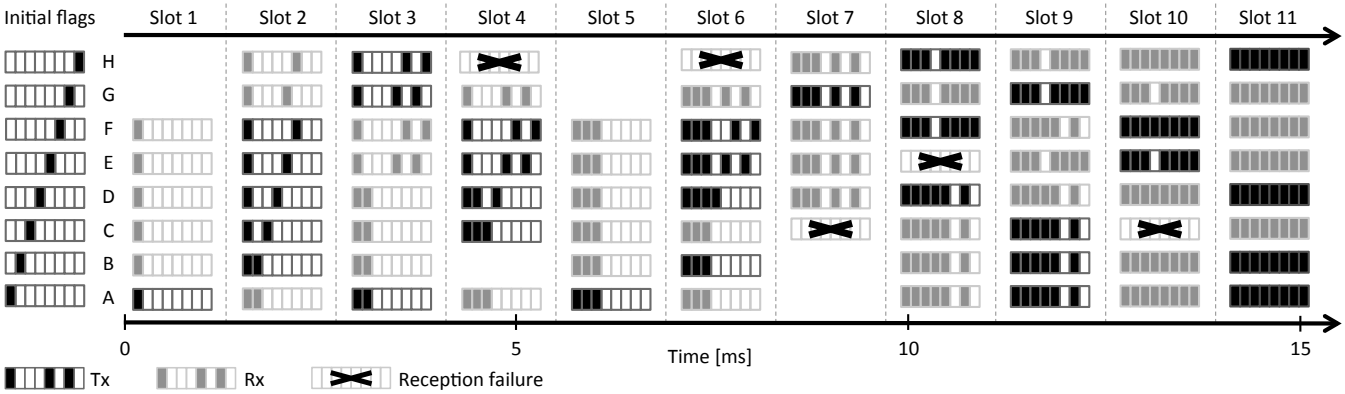


Figure 4: A real trace of a Chaos round, collected from eight nodes forming a multi-hop network on FlockLab. Nodes compute the maximum as described in Sec. 2.1; node A is the initiator. For each slot, the flags that are currently set by a node are shown, corresponding to its degree of completion. Dark flags denote transmitters, grey flags receivers; no flags indicate nodes that neither receive nor transmit. In this run, Chaos completes within 11 slots, which corresponds to about 15 ms. *According to the propagation policy, a node transmits only if the information it received differs from its own. For example, nodes B and G suppress their transmissions in slots 4 and 5, because they received nothing new or outdated in slots 3 and 4, respectively. Slots with many transmitters are followed by slots with few transmitters, since a node never transmits in two consecutive slots and may also fail to receive. For instance, H receives from F in slot 2, but fails to do so in slot 4. This may leave only one transmitter per slot (e.g., A in slot 5 and G in slot 7), justifying the timeout mechanism by which a node re-initiates the communication as it likely came to a premature halt.*

Our solution ensures synchronous transmissions except for variations due to drift of the digitally controlled oscillators (DCOs) that source the nodes' MCUs. Nevertheless, our experiments in Sec. 7 show that drift does not affect the performance of Chaos, even if the execution of the merge operator takes tens of thousands of MCU clock cycles.

Time synchronization. As a result of the above mechanisms, each slot in Chaos has the same duration. Similar to Glossy [14], we leverage this property for time synchronization. Each packet header contains a one-byte *slot counter*, which is set to 1 by the initiator and incremented whenever a node relays a packet. Based on the received slot counter and an estimation of the slot duration, a node can precisely compute the beginning of a round, which serves as the reference time for synchronization. Network-wide time synchronization is important, allowing users to schedule Chaos rounds and other application tasks in between, and to save energy by keeping the radio off for as long as possible between consecutive Chaos rounds.

Summary. With synchronous transmissions, merging, and time synchronization in place, Chaos has all absolutely necessary ingredients. In principle, we could thus stop at this point—upon reception, nodes would apply the merge operator and *always* transmit together. This transmission policy, however, is too aggressive and performs poorly in practice, because the probability of receiving a packet due to capture ranges below 30 % when there are ten or more nodes trying to send at the same time to a common receiver (see Fig. 1).

3.2 Making it Work

The key idea to address the instability of capture in dense networks is to make each participating node in the network aware of the degree of completion of a round. By examining the flags of received packets, a node indeed knows how many nodes still need to contribute and can thus infer the current degree of completion at its neighbors. We utilize this feedback to let a node decide (*i*) whether to transmit or not, (*ii*) when to re-initiate communication as it likely came to a premature halt, and (*iii*) when it is time to aggressively share the final result with all others to reduce energy costs. Below we describe each of these aspects in turn.

3.2.1 To Transmit or Not to Transmit

Our *propagation policy* has two interdependent objectives: (*i*) make data spread quickly in the network to achieve fast completion and (*ii*) keep the number of synchronous transmitters low to increase the reliability of capture in areas where nodes are densely deployed. The intuition underlying our propagation policy, which accomplishes both goals, is to let *the unknown* spread further while suppressing *the known*.

Inspired by work on epidemic communications [6, 20], this materializes in the following: a node sends only if the information contained in a received packet *differs* from its own information. As a result, a node transmits if it learned something new, sharing the previously unknown with its neighbors, or if it knows already more, allowing its neighbors to catch up. Otherwise, a node stays quiet because chances are that it cannot contribute to the knowledge of its neighbors. By suppressing seemingly redundant transmissions, Chaos effectively reduces the number of synchronous transmitters, which helps receiving packets due to capture in dense areas.

Fig. 4 shows our propagation policy in action, using a real trace of a complete Chaos round involving eight nodes in the FlockLab testbed [23]. For each slot and node, the figure shows the flags that are already set, corresponding to each node's current information, and whether a node sends, receives, or stays quiet. We see that, for example, in slot 3, node B receives the same information it already has, so it does not transmit in the following slot. The same happens to node G in slot 4, which leaves only one (instead of two) transmitters in slot 5.

In addition to the described policy, there is another mechanism at play that affects the number of transmitters and receivers in successive slots but is not in direct control of Chaos. Since the probability of capture is low when many

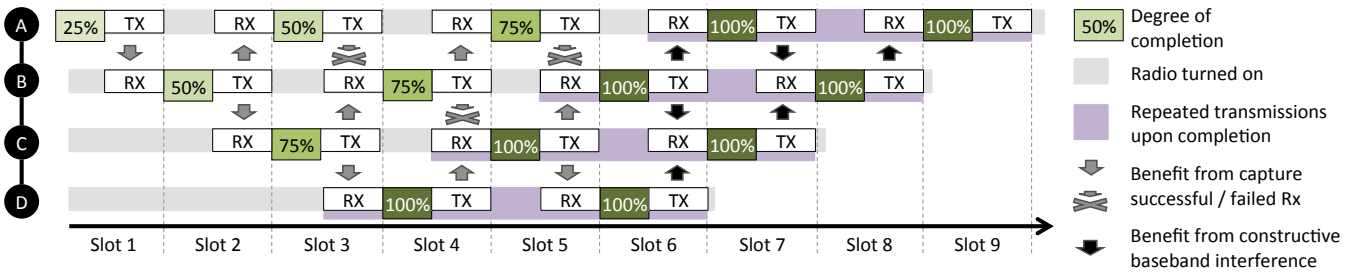


Figure 5: Chaos running on a four-node line topology as nodes gradually switch from the propagation policy to the completion policy. *Upon completion, nodes aggressively share the result by transmitting it up to a certain number of times regardless of what they receive from other nodes. In the example, nodes transmit exactly twice. D is the first node to complete in slot 4; it transmits immediately afterward and in slot 6 as a result of receiving from C in the previous slot. Before slot 4, nodes use the propagation policy and rely only on capture effects for successful reception. From slot 4 onward, nodes start to exploit constructive baseband interference as the transmitted packets are now identical. In this way, nodes complete quickly and reliably, and save energy by turning off the radio after aggressively sharing the result.*

nodes send together, a slot with many transmitters is likely to have only a few receivers. This, in turn, results in only a few transmitters but many receivers in the following slot. While the oscillating behavior in Fig. 4 with eight nodes is mainly due to the fact that a node never transmits in two consecutive slots, capture effects contributes much more to the oscillation in our testbed experiments on large and dense networks, as discussed in Sec. 5.

3.2.2 Fighting Premature Termination

Looking at Fig. 4, we noticed already that there is only node A transmitting in slot 5. Similarly, in slot 7 only node G is transmitting. If these nodes failed to receive in slots 4 and 6, respectively, communication would come to a complete halt—Chaos would terminate prematurely.

To prevent this, we use a timeout mechanism that allows a node to re-initiate the communication if it received nothing for a given number of consecutive slots. In this case, a node transmits a packet on its own containing its current information. We investigate the impact of timeouts and choice of the threshold in Sec. 7.1.1. Our results reveal that while timeouts occur rarely, they are key to achieving a high reliability, particularly in sparse networks.

3.2.3 Aggressively Sharing upon Completion

Early experiments with the described propagation policy and timeout mechanism showed that nodes complete quickly with very high probability. The open problem, however, is to decide when a node should turn off its radio to save energy without sacrificing reliability. Using the propagation policy this decision is difficult to make, because a node receives only little feedback from its neighbors once the bulk of the nodes has reached completion.[3] On the other hand, blindly turning off the radio a certain number of slots after completion may leave some nodes behind, and the threshold for turning off the radio is hard to predict as it strongly depends on network topology and time-varying channel conditions.

Our solution circumvents these issues, while achieving fast completion at all nodes with very high probability. Specifically, using our *completion policy*, a node aggressively shares the final result upon completion by transmitting it up to a certain number of times regardless of what it receives from other nodes. This triggers nearby nodes to also switch to the completion policy, eventually covering the entire network.

This is illustrated in Fig. 5, where D is the first node to complete in slot 4. At this point, D switches from the propagation policy to the completion policy and transmits the final result, in this example exactly twice: the first time in slot 4 when it reaches completion and the second time in slot 6 as a result of receiving from C in the previous slot. Over time more and more nodes switch to the completion policy and transmit the final result. Since the transmitted packets are now identical, receivers benefit from constructive baseband interference, which helps achieve completion at all nodes with very high probability.

Afterward, a node terminates the round: it delivers the result to the application and turns off the radio. In case a node does not complete, it keeps the radio on until the end of the round and then notifies the application. Experiments in Sec. 7.1.2 show that transmitting the result up to five times achieves very good performance across three testbeds with vastly different numbers of nodes, network diameters, node densities, transmit powers, and packet sizes.

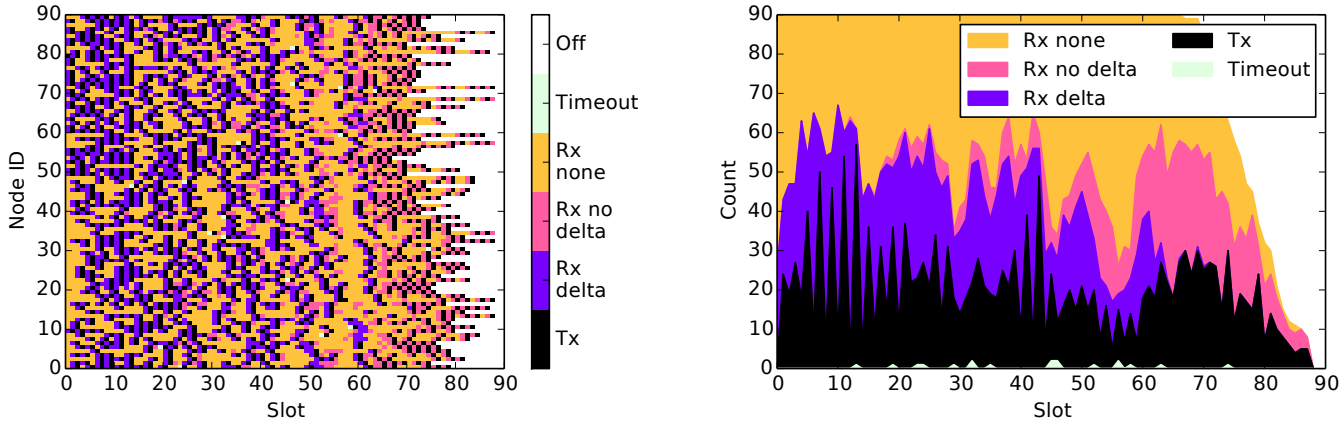
[3]This is because the probability to receive something new or outdated is very small and thus only a few nodes are allowed to transmit according to the propagation policy.

Table 1: Default settings of our Chaos implementation and the test application used in the evaluation

Parameter	*Value*
Timeout window [slots]	3–7
Maximum transmissions after completion	5
Payload size [bytes]	10
Processing time [MCU clock cycles]	2000
Maximum length of a round [seconds]	1.5

4. IMPLEMENTATION DETAILS

We implemented Chaos in Contiki on the TelosB platform. The upper half of Table 1 shows the default configuration. The timeout for re-initiating the propagation expires when a node receives nothing for a number of slots after its last transmission. Nodes randomly draw this number between 3 and 7, independently of each other, after each transmission. We implemented the propagation policy by letting a node transmit if the bitwise XOR of its own flags and the received flags is different from zero. After switching to the completion



(a) Activity of individual nodes.

(b) Activity across all nodes.

Figure 6: Activity over time during a representative round of Chaos. *Off* means a node sent the final result up to five times and turned off the radio. *Timeout* means a node transmitted a packet on its own to re-initiate the propagation. *Rx none* means a node received no packet. *Rx no delta* means a node received but learned nothing new. *Rx delta* means a node received and learned something new. *Tx* means a node sent a packet. *After a short initial phase, nodes typically acquire new information when they receive a packet. This triggers new transmissions and helps spreading each others knowledge throughout the network. Starting from slot 65 they hardly learn anything new.*

policy, a node transmits up to 5 times and then turns off the radio. We motivate these default settings in Sec. 7.

Throughout the experiments we use a test application that executes Chaos periodically. The application requires Chaos to process a 10-byte payload within 2000 MCU clock cycles, as listed in the bottom half of Table 1. This corresponds to a processing time of 0.48 ms since the MCU clock frequency is 4,194,304 Hz, which is sufficient to compute, for example, the maximum across 139 nodes on the Indriya testbed [7]. We note that our implementation currently supports merge operators that take up to 40000 MCU clock cycles; we evaluate the impact of processing time on the performance of Chaos in Sec. 7.3.2. We set the maximum length of a Chaos round to 1.5 s, and determine the mapping of nodes to flags based on information about the networks we use.

5. CHAOS IN ACTION

This section takes a detailed look at a representative round of Chaos to illustrate how the mechanisms described in Sec. 3 blend together in our implementation and perform in practice. To this end, we execute our test application on a 5-hop network of 90 nodes, and record the activity and the degree of completion of every node in each slot throughout a round. In the representative round we discuss, nodes complete after 105 ms, well before the maximum round length of 1.5 s.

5.1 Activity over Time

Fig. 6(a) shows the activity of every node over time. In the beginning, all nodes have the radio turned on. The initiator (ID 80) sends its packet in the first slot and 24 neighboring nodes receive, as shown in Fig. 6(a) by their "Rx delta" in slot 1. Depending on the hop distance from the initiator and how packets propagate in the network, all 90 nodes gradually receive their first packet within the first 9 slots of the round.

Once all nodes are actively involved in the propagation, they acquire new information in most of the slots in which they receive a packet. According to the propagation policy, this in turn triggers new transmissions and helps spreading each others knowledge throughout the network. The likelihood of receiving new information starts to decrease around slot 18 when more "Rx no delta" slots occur. As a result, nodes begin to transmit less frequently. This, however, leads to situations where many nodes receive no packets, as visible in the "Rx none" areas before slots 29, 46, and 58. It is exactly at these moments when the timeout expires at some nodes, causing them to re-initiate the propagation.

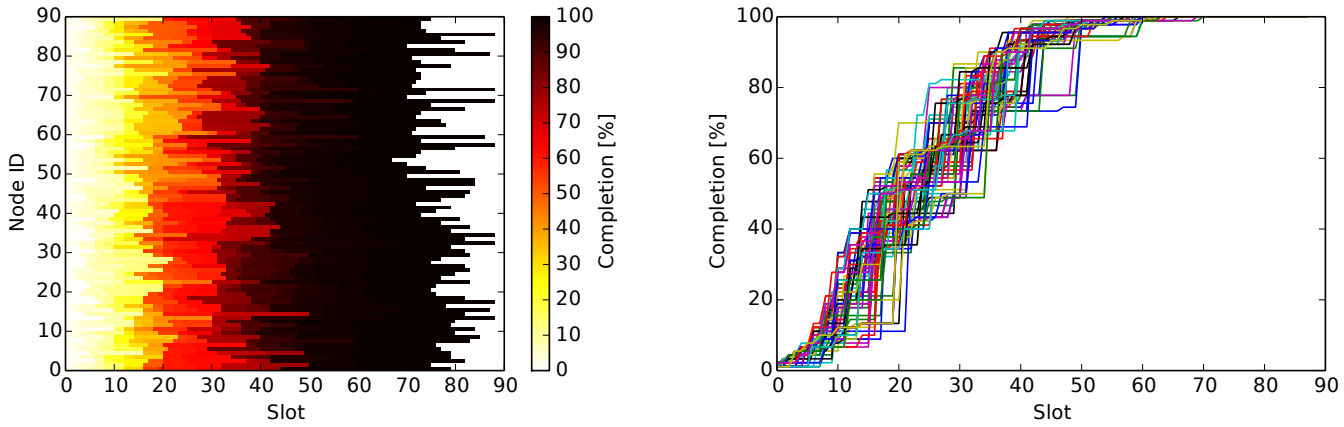
Toward the end of the round, we see an increasing number of nodes that transmit even when they received nothing new. This happens because they reach completion and thus switch to the completion policy, aggressively sharing the final result with all other nodes. After transmitting the result 5 times, nodes turn off the radio and stop participating.

Fig. 6(b) shows the number of nodes engaged in a certain activity during each slot of the same round. We see that in the very beginning the number of participating nodes that either transmit or receive new information increases rapidly. After this initial phase, the number of nodes that receive new information starts to decrease slowly. During this intermediate phase, we can also appreciate the benefits of the timeouts expiring in slots 29, 46, and 58, causing the spikes in the number of nodes that receive new information in the subsequent slots. Finally, the number of participating nodes decreases as more nodes reach completion. In this specific round, nodes turn off the radio between slots 67 and 88.

5.2 Completion over Time

Based on the same representative round of Chaos, we now investigate the amount of information gathered by the nodes as the round evolves. To this end, we depict in Fig. 7 the *completion* of a node in each slot, corresponding to the percentage of bits that are set in the flags of a node. Initially, nodes have only their own flag set. Fig. 7(a) shows that during the first slots completion increases slowly, since only a few nodes are involved in the propagation. By slot 10 nearly all nodes have completion below 30 %, as shown in Fig. 7(b).

Completion grows more rapidly once this initial phase is over and all nodes joined in the propagation. For example,



(a) Completion of individual nodes.

(b) Completion across all nodes.

Figure 7: Completion over time during a representative round of Chaos, which is the percentage of bits that are set in the flags of a node. *Completion grows slowly in the beginning, then rapidly, and finally flattens out. This behavior resembles how the infection rate grows over time in epidemic spreading. All 90 nodes complete by slot 70 (i.e., after 105 ms).*

the average completion increases from 15 % in slot 10 up to 80 % in slot 40. It is exactly during this phase when Chaos fully exploits spatial diversity. Different nodes have different flags set, and the degree of completion differs by up to 20 % among disjoint clusters of nodes. The jumps in completion visible in Fig. 7(b) indicate precisely when nodes from these clusters exchange packets and merge their different flags.

We see in Fig. 7(b) that the completion flattens out beyond slot 40. This behavior, together with the slow start and the rapid increase afterward, resembles how the fraction of infected nodes grows over time in epidemic spreading [6]. Different from epidemics, around slot 60 more and more nodes switch from the propagation to the completion policy, which causes the remaining nodes to suddenly jump to 100 % completion. In this specific round, all 90 nodes share the final result between slots 47 and 70, that is, 71 to 105 ms after the beginning of the round.

6. EVALUATION METHODOLOGY

Using the implementation and test application described in Sec. 4, we evaluate the performance of Chaos through extensive testbed experiments. Sec. 7 investigates the impact of low-level mechanisms, network characteristics, and application parameters, and Sec. 8 compares Chaos with the state of the art. Sec. 9 complements our evaluation with simulations in which we analyze the scalability of Chaos in networks of arbitrary size and node density. Before discussing our results, we describe the metrics and testbeds we use.

Metrics. We consider four key performance metrics: (*i*) *radio on-time* is the time a node has the radio turned on during a Chaos round; (*ii*) *latency* is the time from when a round starts until when a node reaches completion; (*iii*) *reliability* is the percentage of rounds in which *all* nodes reach completion; and (*iv*) *radio duty cycle* is the fraction of time a node has the radio turned on when Chaos executes periodically. We compute latency and reliability based on information output by the nodes, and measure radio on-time and radio duty cycle in software using Contiki's power profiler [10].

Testbeds. We use the following three testbeds: Twist [17], Indriya [7], and FlockLab [23]. All testbeds feature TelosB nodes deployed in university buildings with realistic interference from the presence of people and co-located Wi-Fi. As shown in Table 2, the size of the testbeds ranges from 30 to 139 nodes, and their diameter varies between 3 and 10 hops depending on physical extent and transmit power.

Table 2: Evaluation testbeds and settings

Testbed	*Nodes*	*Initiators [node ID]*	*Tx power [dBm]*	*Diam. [hops]*
Indriya	139	1, 60, 121	0 to -11	5 to 9
Twist	90	12, 192, 230	0 to -25	3 to 7
FlockLab	30	4, 6, 19	0 to -15	4 to 10

7. EVALUATING CHAOS ON TESTBEDS

This section evaluates the impact of low-level mechanisms as well as network and application characteristics on the performance of Chaos. Our test application runs for 30 minutes and executes Chaos periodically every 2 seconds. For each experimental setting and testbed, we perform three independent runs with different initiators (see Table 2), and report per-node averages and standard deviations, plotted as lines and error bars in the figures. Unless otherwise stated, Chaos retains the default values from Table 1.

We note that this section reports latency in terms of number of slots required for a node to reach completion, because we are primarily interested in understanding the connection between the internal workings of Chaos and its performance. We report latency in terms of time to reach completion when comparing Chaos against the state of the art in Sec. 8.

7.1 Benefits of Low-Level Mechanisms

We start by evaluating the benefits provided by the timeout mechanism and the completion policy.

7.1.1 Timeouts: Keep Going!

The timeout mechanism allows a node to transmit a packet on its own if it received nothing for a given number of consecutive slots. Our results confirm that:

Finding 1. *Timeouts are key in fighting premature termination, thus significantly boosting the reliability of Chaos.*

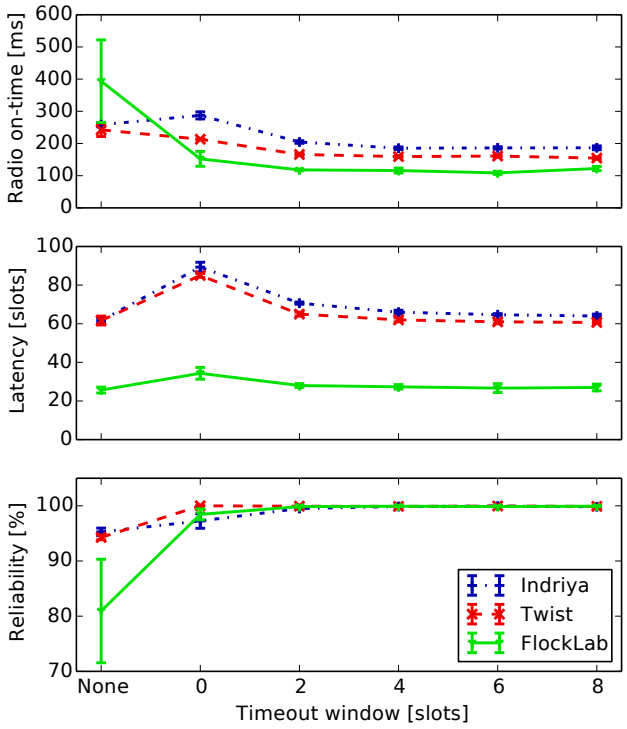


Figure 8: Impact of timeouts. *None* means no timeout. *Numbers* denote the timeout window W, where $W = 0$ means nodes always send after 3 slots without a reception and $W > 0$ means they transmit after a random number of slots between 3 and $3 + W$. *Using randomized timeouts, Chaos completes quickly with an average reliability above 99.85 % across all three testbeds.*

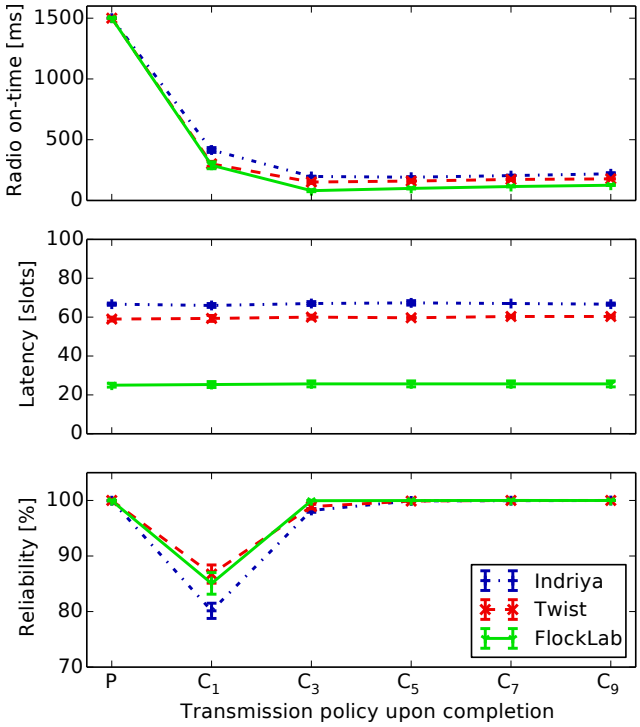


Figure 9: Impact of the completion policy. P means nodes keep transmitting according to the propagation policy upon completion; C_k means nodes switch to the completion policy and transmit the result up to k times before turning off the radio. *Transmitting multiple times upon completion allows Chaos to greatly save on energy without sacrificing reliability.*

Scenario. We first run a few experiments without the timeout. Then we enable the timeout and let it expire between 3 and $3 + W$ slots after the last transmission of a node, where W is the *timeout window*. Nodes, independently of each other, draw a random number from this interval after each transmission. We found in initial experiments that timeouts expiring after less than 3 slots cause high contention and ultimately a loss in performance. We test values of W between 0 and 8 in steps of 2.

Results. Looking at Fig. 8, we see that timeouts are highly beneficial to Chaos. When the timeout is disabled, the propagation often comes to a premature halt and Chaos performs poorly in terms of reliability and radio on-time. This is most evident on FlockLab, where the average reliability ranges below 81 % because nodes have only a few chances to receive packets due to the low node density in the testbed. Moreover, nodes that fail to complete keep their radio on until the end of a round, which explains the high radio on-time.

Fig. 8 shows that enabling the timeout improves the reliability of Chaos. However, we also see that a timeout window of $W = 0$ is not ideal—indeed it is counterproductive and causes an increase in latency. On Indriya, for example, the average latency rises from 62 slots without the timeout to 89 slots with a *fixed* timeout, while the average reliability remains below 99 %. Since there is no randomness in the timeout, nodes transmit exactly after 3 consecutive slots without having received a packet. The resulting contention causes several reception failures, further delaying completion.

A wider timeout window $W > 0$ reduces contention, which improves both latency and reliability. Our Chaos implementation uses $W = 4$ slots as the default timeout window, because with this setting Chaos achieves an average reliability above 99.85 % on all three testbeds. Larger timeout windows do not yield significant performance improvements.

7.1.2 Completion Policy: Get It All Done!

Based on the completion policy, a node aggressively shares the result upon completion by transmitting it up to a certain number of times and then turns off the radio. We find that:

Finding 2. *The completion policy significantly reduces energy costs without sacrificing the reliability of Chaos.*

Scenario. We first perform a few runs in which nodes keep transmitting according to the propagation policy after completion. We then allow the nodes to switch to the completion policy and transmit the result up to k times. We test values of k between 1 and 9 in steps of 2 in different runs.

Results. When keeping to the propagation policy, a node sends after completion only if it receives a packet with at least one flag not set. In this way, Chaos approaches a reliability of 100 %, as shown in Fig. 9, but there is no way for a node to decide when it is safe to turn off the radio. Thus, nodes keep the radio on for the entire duration of a round.

Fig. 9 shows that the completion policy significantly reduces radio on-time. A single transmission after completion, however, is insufficient: many nodes turn off the radio before

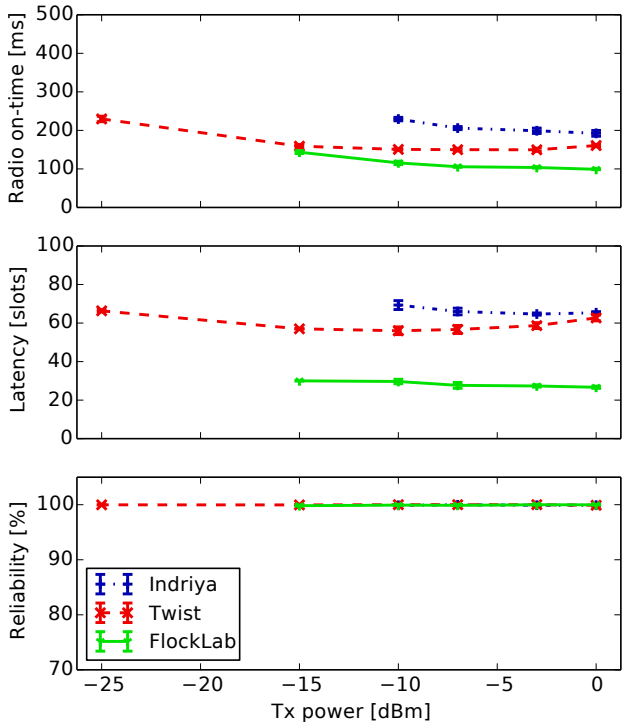


Figure 10: Impact of network properties. *Radio on-time and latency depend on node density and network diameter, whereas reliability is always very close to 100 %.*

others could receive the final result, causing the reliability drop visible in Fig. 9. More transmissions reduce the problem. Our results show that nodes can be aggressive upon completion, because they benefit from constructive interference when transmitting the *same* final result. Our current implementation lets nodes transmit five times after completion. With this setting, Chaos achieves an average reliability above 99.91 % across all testbeds, while reducing the average radio on-time to 191 ms on Indriya—8× smaller compared with the runs in which nodes keep to the completion policy.

7.2 Impact of Network Properties

Next, we investigate the performance of Chaos depending on network properties, including number of nodes, network diameter, and node density. Our results reveal that:

Finding 3. *Chaos operates efficiently under a wide range of scenarios, including networks with high node density.*

Scenario. To evaluate Chaos on the broadest range of network properties, we vary the transmit power between the maximum output power of a TelosB node (0 dBm) and the minimum that keeps the network on a specific testbed fully connected. Table 2 shows for each testbed the range of transmit powers and the corresponding network diameters. Overall, we evaluate Chaos on networks that range from 30 to 139 nodes in size and span between 3 and 10 hops.

Results. Fig. 10 shows that reliability is almost 100 % across all scenarios we tested. In particular, we measure an average reliability of 99.90 % at 0 dBm on Twist, a setting where 90 nodes are densely deployed within 3 hops. This shows that Chaos successfully copes with the instability of capture in dense networks, thanks to our mechanisms from Sec. 3.

Further, we see that for each testbed there is an optimal transmit power in terms of latency and radio on-time. These optimal settings (-3 dBm on Indriya, -10 dBm on Twist, and 0 dBm on FlockLab) are sufficiently high to keep the network diameter short, yet low enough for Chaos to exploit spatial diversity. Lower transmit powers than the optimal ones entail a larger network diameter and less reliable links. Thus, more slots are needed to reach completion, causing an increase in latency and radio on-time as visible in Fig. 10. Due to high contention, we also see a small increase in both metrics at higher transmit powers on Twist.

By comparing the lowest latency for each testbed, we can also appreciate that Chaos scales gracefully in large networks where it can fully exploit spatial diversity. Chaos completes on average within 30 slots on 30 nodes (FlockLab), but takes only 56 slots on 90 nodes (Twist) and 59 slots on 139 nodes (Indriya). Sec. 9 further evaluates the scalability of Chaos.

The shortest average radio on-times are 199 ms on Indriya, 151 ms on Twist, and 99 ms on FlockLab. To put these numbers into perspective, consider an application that executes Chaos periodically, with periods between 2 s and 5 min. Table 3 shows the corresponding average radio duty cycles. For example, when Chaos runs every 10 s, the radio duty cycle is below 2 % on all testbeds. In fact, results in Sec. 8 demonstrate that Chaos is at least one order of magnitude more energy-efficient than the state of the art.

Table 3: Average radio duty cycles when Chaos executes periodically on the different testbeds

	Period of Chaos				
Testbed	*2 s*	*10 s*	*30 s*	*1 min*	*5 min*
Indriya	9.95 %	1.99 %	0.66 %	0.33 %	0.07 %
Twist	7.55 %	1.58 %	0.50 %	0.25 %	0.05 %
FlockLab	4.95 %	0.99 %	0.33 %	0.17 %	0.03 %

Table 4: Packet size, packet airtime, and slot length for different payload sizes on the Indriya testbed

	Payload size [bytes]				
Parameter	*0*	*25*	*50*	*75*	*100*
Packet size [bytes]	26	51	76	101	126
Packet airtime [ms]	0.83	1.63	2.43	3.23	4.03
Slot length [ms]	1.31	2.11	2.91	3.71	4.51

7.3 Impact of Application Characteristics

We now evaluate how application parameters like payload size and processing time affect the performance of Chaos.

7.3.1 Payload Size

In Chaos, the total size of a packet follows from the fixed size IEEE 802.15.4 and Chaos headers, the size of the flags as determined by the number of nodes bound to participate in an all-to-all interaction, and the variable size payload field as determined by the application. By examining the impact of the latter on the performance of Chaos, we find that:

Finding 4. *Chaos efficiently supports large packets, with no noticeable impact on reliability.*

Scenario. We vary the payload size from 0 to 100 bytes in steps of 25. The IEEE 802.15.4 and Chaos headers together are 8 bytes and the flags occupy $\lceil N/8 \rceil$ bytes in an N-node network. So, for example, on Indriya this corresponds to a total packet size between 26 and 126 bytes (see Table 4).

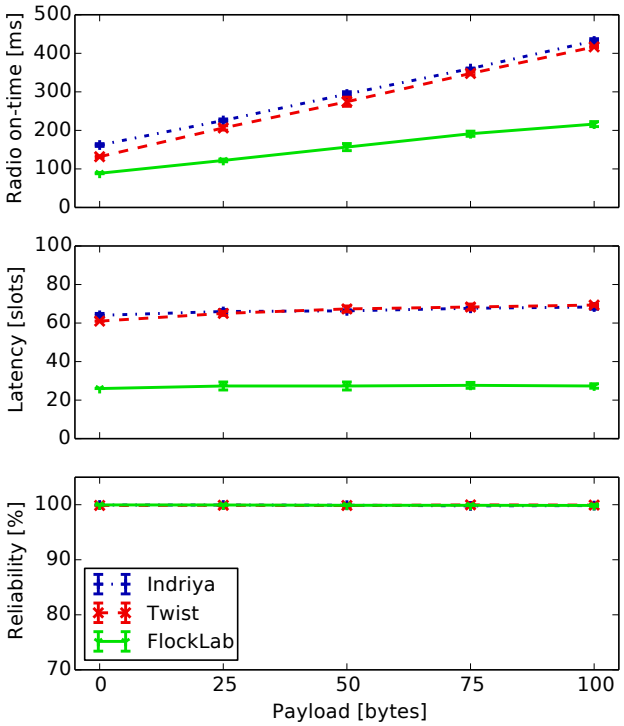


Figure 11: Impact of payload size. *Radio on-time increases linearly with payload size, latency increases at most by a few slots, and reliabilty is consistently above 99.78 %.*

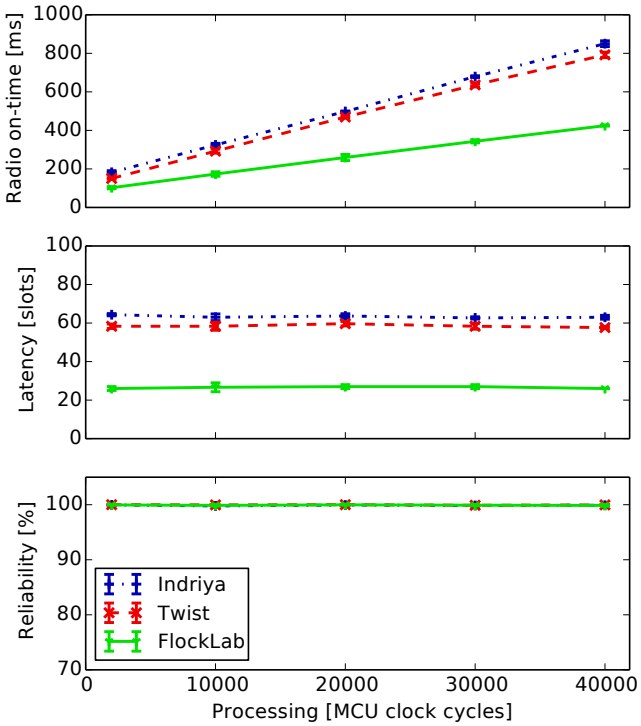


Figure 12: Impact of processing time. *Radio on-time increases linearly with processing time because each slot becomes longer. Latency and reliability are unaffected.*

Results. We learn from Fig. 11 that the payload size has no noticeable impact on reliability, averaging above 99.78 % on all testbeds. Although larger packets are more susceptible to channel dynamics and packet corruption than smaller packets [34], spatial diversity and timeouts mitigate these effects in Chaos. In fact, we observe that with larger packets only a few more slots are required to reach completion. When increasing the payload size from 0 to 100 bytes, the average latency increases from 64 to 68 slots on Indriya, from 61 to 69 slots on Twist, and from 26 to 27 slots on FlockLab.

Because the length of a Chaos slot increases linearly with payload size (see Table 4), radio on-time increases linearly, too. For instance, on Indriya radio on-time averages 226 ms for a 25-byte payload and 432 ms for a 100-byte payload. This 2× increase in radio on-time, however, corresponds to a 4× increase in the amount of shared data, confirming that Chaos is highly efficient also when propagating large packets.

7.3.2 *Processing Time*

Finally, we evaluate how the time required to process the flags and the payload according to the merge operator affects the performance of Chaos. We find that:

Finding 5. *Chaos supports a wide range of processing times, with no noticeable impact on reliability and efficiency.*

Scenario. We evaluate merge operators that take between 2000 and 40000 MCU clock cycles to execute, corresponding to processing times between 0.48 and 9.5 ms. This range covers a broad spectrum of potential merge operators, from simple aggregates like the maximum to more complex computations such as a fixed-point FFT [31].

Results. Looking at Fig. 12, we find that processing time has no noticeable impact on reliability, which averages above 99.78 % across all settings we tested. Further, we see that latency remains constant. This is because the effects of varying MCU clock drifts among nodes associated with longer processing times are canceled out by the chaotic exchange of packets and spatial diversity in Chaos. Radio on-time increases linearly with processing time since each slot becomes longer. It is possible to counteract this trend by turning off the radio while processing. We leave this energy optimization for future work.

8. COMPARING CHAOS ON TESTBEDS

We compare Chaos with two state-of-the-art solutions for all-to-all data sharing in low-power wireless networks. Our results demonstrate that:

Finding 6. *Chaos is 3–23× more efficient than state-of-the-art approaches at a reliability close to 100 %.*

Protocols. There is no previous scheme that natively supports all-to-all data sharing in low-power wireless networks. In line with the current practice, we therefore compare Chaos with two state-of-the-art solutions for achieving all-to-all interactions. The Low-Power Wireless Bus (LWB) is a Glossy-based protocol that supports multiple communication patterns and is more efficient than most existing solutions [13]. An all-to-all interaction in LWB works by letting each node first flood its data and then locally compute the result. We also consider the classical collect-process-disseminate solution by using the default data collection and dissemination protocols in TinyOS: the Collection Tree Protocol (CTP) [15] and Drip [38]. We run both protocols on top of BoX-MAC, the default low-power listening link layer in TinyOS [27].[4]

[4]We obtained inconsistent results when trying to integrate CTP and Drip over BoX-MAC. The main developers of CTP

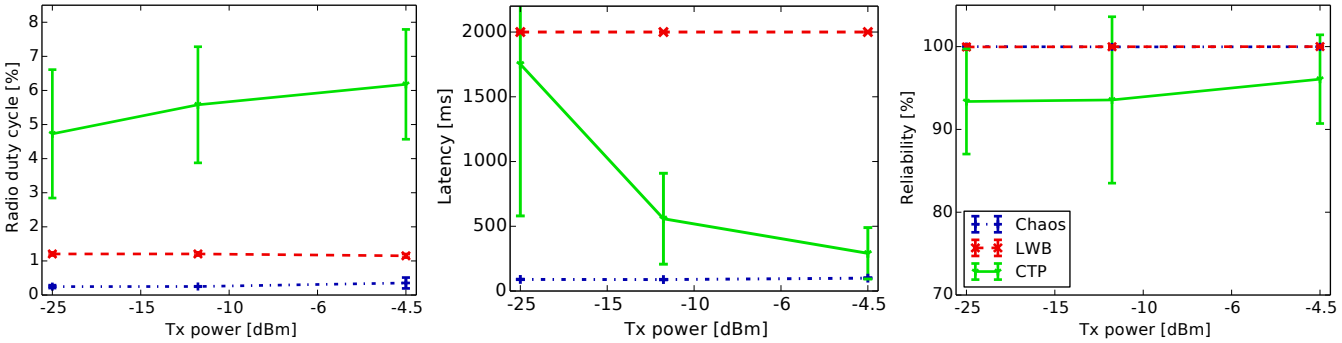


Figure 13: Performance of Chaos, LWB, and CTP on Twist for different transmit powers. *Chaos outperforms LWB and CTP in all metrics, achieving severalfold improvements in radio duty cycle and latency at a reliability near 100 %.*

Table 5: Summary of comparison results

	Radio duty cycle					*Latency*					*Reliability*				
	Average [%]			***Gain vs.***		*Average [ms]*			***Gain vs.***		*Average [%]*			***Gain vs.***	
Tx power	*Chaos*	*LWB*	*CTP*	***LWB***	***CTP***	*Chaos*	*LWB*	*CTP*	***LWB***	***CTP***	*Chaos*	*LWB*	*CTP*	***LWB***	***CTP***
-25 dBm	0.24	1.20	4.73	**5**×	**19**×	89	2000	1752	**22**×	**20**×	100.00	99.96	93.37	**1**×	**7**×
-11 dBm	0.25	1.21	5.58	**5**×	**22**×	88	2000	558	**23**×	**6**×	99.96	99.98	93.56	**1**×	**6**×
-4.5 dBm	0.35	1.15	6.18	**3**×	**17**×	99	2000	291	**20**×	**3**×	99.98	100.00	96.07	**1**×	**4**×

Scenario. We consider again the test application from the previous section with the default settings from Table 1. The application runs for 30 minutes on Twist and executes Chaos periodically every minute. In LWB, nodes generate a packet every minute, and we use the LWB-low-latency scheduling policy to reduce latency [13]. In CTP, nodes generate packets with a random period that averages 1 min. The wake-up interval of BoX-MAC is 256 ms, which showed to be the most energy-efficient setting in this scenario. All other LWB and CTP parameters retain their default values. We use the same node as the initiator in Chaos, the host in LWB, and the sink in CTP. We test three different transmit powers (-25 dBm, -11 dBm, -4.5 dBm), and perform for each transmit power and protocol three independent runs.

Results. Fig. 13 shows the performance of Chaos, LWB, and CTP against transmit power. We see that Chaos is significantly faster and more energy-efficient than LWB, while both achieve a reliability near 100 %. Although LWB utilizes synchronous transmissions, it does not exploit spatial diversity: nodes in an N-node network need to perform N sequential Glossy floods before reaching completion. By contrast, Chaos fully exploits spatial diversity to gain in efficiency.

Fig. 13 shows that Chaos also outperforms CTP in all metrics. Moreover, the latency of CTP significantly increases for lower transmit powers, because the collection tree it uses to route packets toward the sink becomes deeper. By contrast, Chaos and LWB are only marginally affected by different transmit powers. Finally, we see from the standard deviations that, unlike CTP, nodes using Chaos or LWB have similar performance due to the absence of routing.

confirmed that other people experienced the same issue. For this reason, we show results from a configuration that supports only data collection with CTP on top of BoX-MAC. Nevertheless, results in Fig. 13 and Table 5 demonstrate that Chaos greatly outperforms CTP alone—adding dissemination with Drip would further widen this performance gap.

Summary. Table 5 summarizes our comparison results. It shows the average performance of each protocol and the improvements of Chaos over LWB and CTP. We see, for example, that Chaos reduces radio duty cycle by 3–5× compared with LWB and by 17–22× compared with CTP. Moreover, Chaos completes within 89–99 ms, up to 20–23× faster than LWB and CTP. Despite these improvements in efficiency, Chaos achieves a very high reliability above 99.96 %, which is comparable to LWB and significantly better than CTP. Overall, our results demonstrate that Chaos provides extremely efficient and reliable all-to-all data sharing and achieves severalfold improvements over the state of the art.

9. SCALABILITY IN SIMULATIONS

This section uses simulations to analyze how Chaos scales with the number of nodes and the node density. Simulations allow us to consider networks of arbitrary size and density. Furthermore, we can scale beyond the limitation imposed by the maximum IEEE 802.15.4 packet size—without some form of data compression Chaos supports up to 1000 nodes assuming packets contain no payload. Because we maintain that Chaos is applicable to other wireless technologies supporting larger packets, such as Wi-Fi, we want to explore its scalability in even larger networks. We find that:

Finding 7. *Chaos scales efficiently to large networks with several thousands of nodes and various node densities.*

Simulator. We implemented a Chaos simulator in Python. The simulator accounts for the propagation policy, the timeout mechanism, and the completion policy from Sec. 3. It assumes perfect clocks and neglects interrupt delays. Nodes are randomly placed according to a uniform distribution. To simulate the propagation of packets, we use the log-normal path loss model [33], which has been validated through real-world experiments [5]. We augment the model by introducing randomized noise and channel dynamics calibrated from

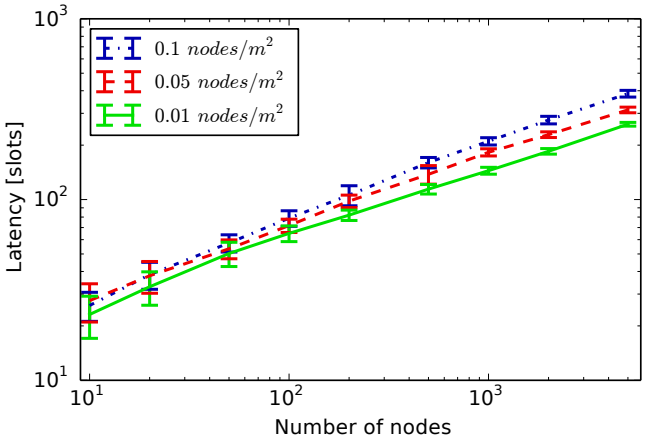


Figure 14: Chaos in simulations. *Latency scales linearly with the square root of the number of nodes.*

real traces [36, 37]. Additionally, the simulator accounts for capture effects and constructive interference based on the received strengths of multiple overlapping signals.

The simulator takes as inputs the total number of nodes, the average node density, and a noise profile. It outputs the number of slots required for a node to reach completion.

Settings. We simulate networks with 10 to 5000 nodes and three average node densities: 0.01, 0.05, and 0.1 nodes per square meter. For each setting, we perform 100 independent runs of a Chaos round. In every run, we use a different random seed to effectively generate a different network topology. This yields a total of 2700 runs with different topologies.

Results. Fig. 14 shows that latency (*i.e.*, the number of slots required for a node to reach completion) scales linearly with the number of nodes, the average slope being smaller than 0.7 across all node densities. For example, doubling the number of nodes yields a 1.4× average increase in latency. Thus, latency practically scales linearly with the square root of the number of nodes. These results confirm that, thanks to spatial diversity, Chaos scales efficiently to large networks with several thousands of nodes and various densities.

10. RELATED WORK

Related work falls in the following two domains:

Synchronous transmissions. Chaos exploits synchronous transmissions for efficient communication. To this end, like several others [4, 8, 13, 39], Chaos builds upon Glossy [14]. Glossy benefits from capture effects and constructive baseband interference to provide efficient flooding in sensor networks. LWB uses Glossy as the foundation for a communication protocol that is similar to a shared bus and supports multiple communication patters as well as mobile nodes immersed in static infrastructures [13]. Splash integrates Glossy with tree-based pipelining for rapid and reliable dissemination of large data objects [8]. In comparison with Glossy and all prior research building upon it, Chaos transforms Glossy from a one-to-all primitive into an all-to-all primitive for versatile data sharing in low-power wireless networks, where nodes send different instead of identical packets and utilize processing opportunities instead of minimizing them.

Backcast demonstrated substantial gains due to constructive interference [12]. Backcast leverages synchronous transmissions of hardware-generated packets for an acknowledged anycast service, serving as the basis for A-MAC, a receiver-initiated link layer for low-power wireless [11]. Glossy demonstrates that, by employing a careful software design, constructive baseband interference can also be achieved through synchronous transmissions triggered in software. Chaos benefits from constructive interference when nodes switch to the completion policy, thereby saving on energy costs.

Flash [24] instead relies only on capture effects for rapid flooding in wireless sensor networks. Like Flash, Chaos relies also only on capture until the first node reaches completion. Unlike Flash, Chaos transmits different packets and merges data on the fly according to a programmable operator.

To enable capture using IEEE 802.15.4 radios, Chaos must ensure that synchronous transmissions overlap within 160 μs, because those radios typically cannot lock onto a stronger signal that arrives later. Instead, in Wi-Fi networks a physical-layer capability called Message in Message (MIM) enables a receiver to decode even if the stronger signal arrives after the receiver has locked onto the interference [21]. Recent work shows how to leverage the opportunities of MIM in IEEE 802.11 a/g/n networks by scheduling links based on their relative signal strengths [26]. We maintain that Chaos is applicable beyond IEEE 802.15.4 and could also harvest capture and MIM opportunities in Wi-Fi networks.

In-network processing. This domain has been a main research thread since the early days of sensor networks. Consequently, there exists is a large body of previous work that uses in-network processing mainly in the context of data aggregation [25, 32] and query processing [19, 40]. Many of these schemes operate on top of a tree-based routing structure [25], whereas others use multi-path routing to be more resilient to link fluctuations and node failures [28]. Their main goal is to save energy by reducing the amount of data to be transmitted to the collection sink. By contrast, Chaos integrates in-network processing directly into the communication support to enable various all-to-all interactions; maintains no routing structure and instead exploits network-wide synchronous transmissions, causing a chaotic propagation of packets throughout the network; and delivers the final result to all participating nodes rather than to a single sink.

11. CONCLUSIONS

Low-power wireless applications increasingly rely on various all-to-all interactions as they move toward control and safety-critical scenarios. We have presented Chaos, the first primitive with native support for versatile and efficient all-to-all data sharing in low-power wireless networks. By embedding programmable in-network processing in a communication support based on synchronous transmissions, Chaos enables a wide range of all-to-all interactions, including the computation of various aggregates, network-wide consensus, and three-phase commit. Our Chaos implementation on the TelosB platform achieves severalfold improvements over the state of the art in the efficiency of all-to-all interactions with almost 100% reliability across all scenarios we tested. Simulations show that Chaos scales efficiently to large multi-hop networks consisting of several thousands of nodes.

We have made the source code of Chaos publicly available at `https://github.com/olafland/chaos` as a means to foster further research into leveraging all-to-all interactions and applying Chaos to other wireless technologies such as Wi-Fi.

Acknowledgments. We thank Felix Sutton, our shepherd Lin Zhong, and the anonymous reviewers for their valuable comments. This work was supported by Nano-Tera, through projects X-Sense and OpenSense, and partially by the EC, through project FP7-STREP-288195 (KARYON).

12. REFERENCES

[1] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. EnviroTrack: Towards an Environmental Computing Paradigm for Distributed Sensor Networks. In *ICDCS: Proc. of the IEEE Int. Conf. on Distributed Computing Systems*, 2004.

[2] J. Arnbak and W. van Blitterswijk. Capacity of slotted ALOHA in Rayleigh-fading Channels. *IEEE Journal on Sected Areas in Communications*, 5(2), 1987.

[3] C. A. Boano, M. A. Zuniga, K. Römer, and T.Voigt. JAG: Reliable and Predictable Wireless Agreement under External Radio Interference. In *RTSS: Proc. of the IEEE Real-Time Systems Symposium*, 2012.

[4] D. Carlson, M. Chang, A. Terzis, Y. Chen, and O. Gnawali. Forwarder Selection in Multi-Transmitter Networks. In *DCOSS: Proc. of the IEEE Int. Conf. on Distributed Computing in Sensor Systems*, 2013.

[5] Y. Chen and A. Terzis. On the Implications of the Log-Normal Path Loss Model: An Efficient Method to Deploy and Move Sensor Motes. In *SenSys: Proc. of the ACM Conf. on Embedded Networked Sensor Systems*, 2011.

[6] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *PODC: Proc. of the ACM Symposium on Principles of Distributed Computing*, 1987.

[7] M. Doddavenkatappa, M. C. Chan, and A. Ananda. Indriya: A Low-Cost, 3D Wireless Sensor Network Testbed. In *TridentCom: Proc. of the ICST Conf. on Testbeds and Research Infrastructures for the Development of Networks and Communities*, 2011.

[8] M. Doddavenkatappa, M. C. Chan, and B. Leong. Splash: Fast Data Dissemination with Constructive Interference in Wireless Sensor Networks. In *NSDI: Proc. of the USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[9] W. Du, J. Deng, Y. S. Han, S. Chen, and P. K. Varshney. A Key Management Scheme for Wireless Sensor Networks Using Deployment Knowledge. In *INFOCOM: Proc. of the IEEE Int. Conf. on Computer Communications*, 2004.

[10] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He. Software-based On-line Energy Estimation for Sensor Nodes. In *EmNets: Proc. of the Workshop on Embedded Networked Sensors*, 2007.

[11] P. Dutta, S. Dawson-Haggerty, Y. Chen, C.-J. M. Liang, and A. Terzis. Design and Evaluation of a Versatile and Efficient Receiver-Initiated Link Layer for Low-Power Wireless. In *SenSys: Proc. of the ACM Conf. on Embedded Networked Sensor Systems*, 2010.

[12] P. Dutta, R. Musaloiu-E., I. Stoica, and A. Terzis. Wireless ACK Collisions Not Considered Harmful. In *HotNets: Proc. of the ACM Workshop on Hot Topics in Networking*, 2008.

[13] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele. Low-Power Wireless Bus. In *SenSys: Proc. of the ACM Conf. on Embedded Networked Sensor Systems*, 2012.

[14] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh. Efficient Network Flooding and Time Synchronization with Glossy. In *IPSN: Proc. of the ACM/IEEE Int. Conf. on Information Processing in Sensor Networks*, 2011.

[15] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection Tree Protocol. In *SenSys: Proc. of the ACM Conf. on Embedded Networked Sensor Systems*, 2009.

[16] V. Hadzilacos and S. Toueg. *Distributed Systems*. Addison Wesley, 1993.

[17] V. Handziski, A. Köpke, A. Willig, and A. Wolisz. TWIST: a Scalable and Reconfigurable Testbed for Wireless Indoor Experiments with Sensor Networks. In *RealMAN: Proc. of the Int. Workshop on Multi-hop Ad Hoc Networks: from Theory to Reality*, 2006.

[18] J. W. Hui and D. Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *SenSys: Proc. of the ACM Conf. on Embedded Networked Sensor Systems*, 2004.

[19] A. Kamra, V. Misra, and D. Rubenstein. CountTorrent: Ubiquitous Access to Query Aggregates in Dynamic and Mobile Sensor Networks. In *SenSys: Proc. of the ACM Conf. on Embedded Networked Sensor Systems*, 2007.

[20] R. Karp, C. Schindelhauer, S. Shenker, and B. Vöcking. Randomized Rumor Spreading. In *FOCS: Proc. the of IEEE Symposium on Foundations of Computer Science*, 2000.

[21] J. Lee, W. Kim, S. J. Lee, D. Jo, J. Ryu, T. Kwon, and Y. Choi. An Experimental Study on the Capture Effect in 802.11a Networks. In *WinTECH: Proc. of the ACM Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization*, 2007.

[22] K. Leentvaar and J. Flint. The Capture Effect in FM Receivers. *IEEE Trans. Commun.*, 24(5), 1976.

[23] 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 *IPSN: Proc. of the ACM/IEEE Int. Conf. on Information Processing in Sensor Networks*, 2013.

[24] J. Lu and K. Whitehouse. Flash Flooding: Exploiting the Capture Effect for Rapid Flooding in Wireless Sensor Networks. In *INFOCOM: Proc. of the IEEE Int. Conf. on Computer Communications*, 2009.

[25] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Trans. Database Systems*, 30(1), 2005.

[26] J. Manweiler, N. Santhapuri, S. Sen, R. Choudhury, S. Nelakuditi, and K. Munagala. Order Matters: Transmission Reordering in Wireless Networks. *IEEE/ACM Transactions on Networking*, 20(2), 2012.

[27] D. Moss and P. Levis. BoX-MACs: Exploiting Physical and Link Layer Boundaries in Low-Power Networking. Technical Report SING-08-00, Stanford, 2008.

[28] S. Nath, P. B. Gibbons, S. Seshan, and Z. Anderson. Synopsis Diffusion for Robust Aggregation in Sensor Networks. *ACM Transactions on Sensor Networks*, 4(2), 2008.

[29] V. Navda, A. Bohra, S. Ganguly, and D. Rubenstein. Using Channel Hopping to Increase 802.11 Resilience to Jamming Attacks. In *INFOCOM: Proc. of the IEEE Int. Conf. on Computer Communications*, 2007.

[30] M. Pajic, S. Sundaram, G. J. Pappas, and R. Mangharam. The Wireless Control Network: A New Approach for Control Over Networks. *IEEE Trans. Autom. Control*, 56(10), 2011.

[31] A. Pullin. An Energy Audit of Automotive Vibration Sources for Energy Harvesting and Applied Computation in Wireless Sensor Networks. Master's thesis, UC Berkeley, 2010.

[32] R. Rajagopalan and P. K. Varshney. Data-Aggregation Techniques in Sensor Networks: A Survey. *IEEE Communications Surveys & Tutorials*, 8(4), 2006.

[33] T. Rappaport. *Wireless Communications: Principles & Practices*. Prentice Hall, 1996.

[34] Y. Sankarasubramaniam, I. F. Akyildiz, and S. W. McLaughlin. Energy Efficiency based Packet Size Optimization in Wireless Sensor Networks. In *SNPA: Proc. of the IEEE Int. Workshop on Sensor Network Protocols and Applications*, 2003.

[35] D. Skeen and M. Stonebraker. A Formal Model of Crash Recovery in a Distributed System. *IEEE Transactions on Software Engineering*, SE-9(3), 1983.

[36] K. Srinivasan, M. Jain, J. I. Choi, T. Azim, E. S. Kim, P. Levis, and B. Krishnamachari. The κ Factor: Inferring Protocol Performance using Inter-Link Reception Correlation. In *MobiCom: Proc. of the ACM Int. Conf. on Mobile Computing and Networking*, 2010.

[37] K. Srinivasan, M. A. Kazandjieva, S. Agarwal, and P. Levis. The β Factor: Measuring Wireless Link Burstiness. In *SenSys: Proc. of the ACM Conf. on Embedded Networked Sensor Systems*, 2008.

[38] G. Tolle and D. Culler. Design of an Application-Cooperative Management System for Wireless Sensor Networks. In *EWSN: Proc. of the European Workshop on Wireless Sensor Networks*, 2005.

[39] Y. Wang, Y. He, X. Mao, Y. Liu, Z. Huang, and X. Li. Exploiting Constructive Interference for Scalable Flooding in Wireless Networks. In *INFOCOM: Proc. of the IEEE Int. Conf. on Computer Communications*, 2012.

[40] Y. Yao and J. Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. *ACM SIGMOD Record*, 31(3), 2002.

[41] M. Zimmerling, F. Ferrari, L. Mottola, T. Voigt, and L. Thiele. pTunes: Runtime Parameter Adaptation for Low-power MAC Protocols. In *IPSN: Proc. of the ACM/IEEE Int. Conf. on Information Processing in Sensor Networks*, 2012.