(article starts on next page)

Continuous Monitoring meets Synchronous Transmissions and In-Network Aggregation

Charalampos Stylianopoulos[†], Magnus Almgren[†], Olaf Landsiedel[‡†], Marina Papatriantafilou[†]
[†]*Chalmers University of Technology, Sweden* [‡] *Kiel University, Germany*
Email: {chasty, magnus.almgren, ptrianta}@chalmers.se, ol@informatik.uni-kiel.de

Abstract—**Continuously monitoring sensor readings is an important building block for many IoT applications. The literature offers resourceful methods that minimize the amount of communication required for continuous monitoring, where Geometric Monitoring (GM) is one of the most generally applicable ones. However, GM has unique communication requirements that require specialized network protocols to unlock the full potential of the algorithm.**

In this work, we show how application and protocol co-design can improve the real-life performance of GM, making it an application of practical value for real IoT deployments. We orchestrate the communication of GM to utilize the properties of a state-of-the-art wireless protocol (Crystal) that relies on synchronous transmissions and is designed for aperiodic traffic, as needed by GM. We bridge the existing gap between the capabilities of the protocol and the requirements of GM, especially in the case of periods of heavy communication. We do so by introducing an in-network aggregation technique relying on latent opportunities for aggregation that we exploit in Crystal's design, allowing us to reliably monitor duplicate-sensitive aggregate functions, such as sum, average or variance. Our results from testbed experiments with a publicly available dataset show that the combination of GM and Crystal results in a very small duty-cycle, a 2.2x - 3.2x improvement compared to the baseline and up to 10x compared to previous work. We also show that our in-network aggregation technique reduces the duty-cycle by up to 1.38x.

Keywords-**continuous monitoring; synchronous transmissions; aggregation; co-design; sensor networks**

I. INTRODUCTION

The problem of monitoring a network-wide system state has fundamental uses in Wireless Sensor Networks (WSN). Whether we are monitoring the temperature in a room to detect outliers [3] and hot-spots [27] or machinery operation statistics in a factory to detect malfunctions [20], we are often interested in keeping track of a function calculated over all the network's readings. More specifically, given a set of sensor nodes $n_1,...,n_N$ with readings $\vec{v_1},...,\vec{v_N}$ that vary over time, we want to continuously track whether the value of a function f, defined over the network-wide weighted average of the readings, is higher (or lower) than a predefined threshold T.

One of the most generally applicable solutions that addresses the communication complexity of the problem is Geometric Monitoring (GM) by Sharfman et al. [26]. GM is a method that can monitor any function, linear or not, with respect to a threshold and can suppress most of the readings without having to communicate. Since its introduction, GM has been extended with sketches [10] and prediction models [11], showing that it is extensible and applicable in many scenarios.

When considering the applicability of GM for IoT deployments, there are aspects to consider that go beyond the algorithm itself. The communication pattern of GM is highly data-dependent and varies significantly during run-time. This implies a challenge for most network protocols as they are usually optimized for periodic communication. Moreover, the actual energy savings on the nodes depend on the underlying communication stack and the way it interacts with GM. Previous work [28] studied the performance of GM in an applied IoT context and showed that the energy overhead of mainstream network stacks limits the effectiveness of the algorithm in practice. Thus, it is interesting to study how to close the gap between the requirements of GM and appropriate protocols that can support them.

Crystal [12], [13] is a recently introduced wireless protocol based on synchronous transmissions that is specifically designed to favor aperiodic communication patterns: it efficiently handles communication when there is little to send within an epoch, but can also accommodate epochs with heavy communication. However, Crystal was originally designed with applications featuring aperiodic data collection from multiple senders to a single sink node. Contrary, GM requires that any node should be able to broadcast updates to every other node in the network. Moreover, even though Crystal gracefully handles the case of low communication load, it misses opportunities to save energy in cases of high load that can benefit, e.g., from in-network aggregation.

In this work, we bridge this gap between the requirements of GM as an application and Crystal as a state-of-the-art wireless protocol that relies on synchronous transmissions. By doing so, we provide a practical realization of a system that continuously monitors sensor values with a high degree of communication suppression (due to properties of GM) and operates at low duty-cycle with high reliability (due to properties of Crystal). We also propose *Arctium*, adding in-network aggregation to synchronous transmission protocols for aperiodic communication; this orthogonal design allows any application that monitors aggregated values from sensors

to get additional reduction in communication on top of the existing design of Crystal. Hence, we show how to: (i) make algorithms such as GM practical for real IoT deployments, in combination with modern protocols such as Crystal and (ii) reduce the energy consumption of such algorithms even further by adding in-network aggregation.

In particular, we make the following contributions:

- We orchestrate and deploy Geometric Monitoring on top of a state-of-the-art wireless network protocol that relies on synchronous transmissions.
- We introduce *Arctium*,[1] a novel method for in-network aggregation that relies on latent opportunities for aggregation in Crystal's design and reliably tracks aggregates, including duplicate-sensitives ones, using fewer transmissions than Crystal.
- We evaluate the effects of application and protocol co-design on real IoT nodes, using publicly available datasets and show a reduction in energy consumption of up to 10x compared to previous work, and how in-network aggregation further reduces it by 1.13-1.38x.

The paper is organized as follows. Section II summarizes GM and Crystal. In Section III, we present our GM and Crystal co-design and describe *Arctium*, our aggregation scheme. In Section IV, we show the results from our evaluation. We present related work in Section V and conclude in Section VI.

II. PRELIMINARIES

A. The Geometric Monitoring Method (GM)

In their seminal work [26], Sharfman et al. present a general method able to monitor (with respect to a threshold) arbitrary functions defined over network-wide aggregates. Instead of having nodes communicate for every new reading, each node uses local information to decide whether to send an update. Each node keeps track of: (i) how far its local readings have drifted from the last reading it broadcasted to everybody in the network and (ii) the *global estimate* vector, which is the weighted sum of all the readings each node in the network broadcasted last. Based on these, each node locally, i.e. without communication, creates a region (depicted as a sphere in the domain of the values being monitored) and checks whether that region crosses the surface that indicates the threshold. As long as this region remains fully on one side of the threshold, no communication is required. Otherwise, broadcasts are needed to see whether the local change is offset by changes at other nodes (false alarm) or if the function has actually crossed the threshold. A more detailed presentation can be found in Section A in the Appendix, as well as in the original GM publication by Sharfman et al. [26].

---

[1]Arctium is a genus of plants, notable for their velcro-like heads that tend to stick and aggregate on other materials.

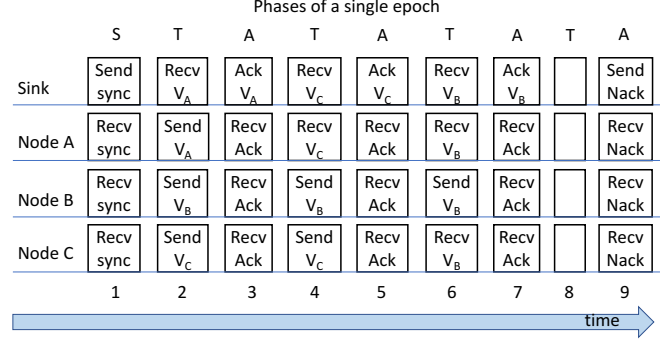|	Phases of a single epoch								
	S	T	A	T	A	T	A	T	A
Sink	Send sync	Recv V_A	Ack V_A	Recv V_C	Ack V_C	Recv V_B	Ack V_B		Send Nack
Node A	Recv sync	Send V_A	Recv Ack	Recv V_C	Recv Ack	Recv V_B	Recv Ack		Recv Nack
Node B	Recv sync	Send V_B	Recv Ack	Send V_B	Recv Ack	Send V_B	Recv Ack		Recv Nack
Node C	Recv sync	Send V_C	Recv Ack	Send V_C	Recv Ack	Recv V_B	Recv Ack		Recv Nack
	1	2	3	4	5	6	7	8	9

time

Figure 1: A direct adaptation from [13], showing the different phases of Crystal, for a network with three nodes and one sink. Each box indicates a (possibly multi-hop) Glossy flood.

What is relevant for this work, from a communication behavior point of view, is that the method results in nodes communicating aperiodically: there are intervals when most nodes suppress their readings with almost no communication, while at other times multiple nodes need to broadcast their values so that all nodes in the network can update their global estimate. Such a dynamic communication pattern makes GM a challenging application for many state-of-the-art low-power wireless protocols.

B. Crystal

Crystal [12], [13] is a protocol for low power and highly reliable aperiodic collection of data from multiple nodes to a single sink. Crystal has *epochs* and is designed to be particularly useful for applications where the number of nodes that have data to be collected at each epoch varies but is typically low. In its core, Crystal relies on Glossy [7], the seminal work of Ferrari et al. By tight (sub-microsecond) scheduling of transmissions, Glossy makes use of the *capture effect* [21] and *constructive interference* [7] to achieve network-wide flooding with extremely small latency.

Crystal, in turn, builds a schedule on top of Glossy that consists of a series of phases that form an *epoch*. Figure 1 shows an example of a single Crystal epoch for a network with three nodes about to send their values to the sink. In this example, the epoch takes nine phases until completion. V_X indicates the value that node X is trying to send to the sink. First, during the S-phase, the sink floods a synchronization message to ensure all nodes have a common reference point. Then, during the T-phases (phases 2, 4, 6 and 8), any node that has data to be collected initiates its own Glossy transmission. With high probability, one of the transmissions reaches the sink, which acknowledges the transmission in the following A-phase. In this example, the value from Node C reached the sink at phase 4 and was acknowledged at phase 5. Nodes that did not get their transmissions acknowledged transmit again during the next T-phase, until one (or more,

for safety) T-phases are empty. In this case, the sink issues a negative acknowledgment (phase 9) and the network can go back to sleep until the beginning of the next epoch.

III. GM, Crystal and *Arctium* co-design

A. Overview

The first idea is to use GM on top of the existing communication schedule followed by Crystal (see Figure 1), with benefits from both worlds: (i) communication reduction from using GM as an application, and (ii) efficient handling of the aperiodic communication pattern using Crystal as the underlying protocol. This can be achieved by using the T-phase to send updates from nodes who detected threshold violations and the A-phases to trickle down the changes to the global estimate. All nodes perform the computation required by GM (threshold checking) between the T-A pairs to decide whether to transmit or just forward packets.

The second idea is to gracefully handle the (so far overlooked) cases of multiple concurrent transmitters. We propose *Arctium* to extend the design of Crystal in a novel way by introducing in-network aggregation inside Crystal's schedule. We do so by extending the role of the T-phases: during these phases, nodes do not only forward messages towards the sink, but they also have the opportunity to overhear neighbouring transmissions and combine these with their own. As a result, *Arctium* requires fewer messages to complete the epoch and is more energy efficient than Crystal. *Arctium* is orthogonal to GM and works for any application that is monitoring aggregates, such as *min, max, sum, average* and *variance*.

In the rest of this section, we first describe how we design GM in coordination with Crystal. Then we introduce *Arctium* and outline how it enables aggregation on top of Crystal.

B. Orchestrating GM communication with synchronous transmissions

Crystal was originally designed for data collection: sensor values from any node must be collected at a single, fixed sink. The original requirements of GM are however different:[2] when a node detects a local threshold violation and transmits its update, all other nodes must receive it to recompute the global estimate. In order to bridge the gap between the two models and fit the requirements of GM on top of the existing Crystal schedule, we make use of the Glossy transmission during the A-phase. That transmission not only acknowledges the reception of a node's value by the sink, but it also lets every node in the network know the new, updated value of the global estimate.

Overall, in our system, nodes first collect new sensor readings and perform threshold checking (see Section II-A)

---

[2]There are models of GM that consider a single, coordinating node that is able to query values from specific nodes (using e.g. a unicast) and resolve threshold violations. However, since unicast is not part of Crystal's schedule, we do not consider such models in this work.

at the beginning of the epoch. If they cross the threshold, they send their update during the T-phase. If an update reaches the sink, the sink updates the global estimate. The new global estimate will then trickle-down to the other nodes during the A-phase, piggybacked on the acknowledgment of the previous T-phase transmission. When nodes receive the new global estimate, they have until the next T-phase to perform the threshold checking again (based on the new global estimate) and decide whether they need to transmit their update. This continues until there are no further transmissions (2-3 empty T-phases) and the epoch ends. Note that, on rare cases, a packet loss during the A-phase might go undetected resulting in some nodes momentarily having an outdated global estimate. This is corrected in the next A-phase when the sink disseminates the new global estimate.

Efficient threshold checking: As mentioned in Section II-A, nodes need to perform the threshold checking required by GM between the T-A pairs in the protocol, which can be computationally challenging for IoT devices with very limited computing and energy resources as the threshold surface might have an arbitrary shape.

We address this by approximating the GM-spheres with a simpler shape that makes the computations significantly faster while ensuring that we do not introduce false negatives. As an example in 2D, the original spheres (now circles, similar to the example Figure 1 in Appendix) can be replaced with squares containing the former, which results in simpler boundary conditions (as it is simpler to check whether the sides of a square, rather than points on a circle, cross a surface). Naturally, there will be cases where the square check will report a violation even though the circle inside it does not actually cross the threshold, hence sacrificing communication reduction for ease of computation. This relaxation might not cope with high dimensions, but in many cases of monitoring statistics such as variance or correlations between nodes, it provides a simple and efficient solution which we have also experimentally verified (e.g. when monitoring the variance, the computation time decreases from 20ms to just 7ms on the TelosB platform, see Section IV). In Section V we discuss other alternatives in existing literature [19].

C. Arctium: Enhancing Crystal with in-network aggregation

Motivation: Crystal is designed to efficiently handle the case where only a few nodes have data to transmit within the epoch. However, for many applications including GM, there are epochs where many or all the nodes in the network have data to transmit. For example, for GM this happens when the monitored value is close to the threshold. Hence, there is potential for improvement on a protocol level, which we exploit by proposing *Arctium*.

The following two observation has guided our work of *Arctium*. (i) In applications such as GM, the individual values by themselves are not interesting but rather only the

	Phases of a single epoch						
	S	T	A	T	A	T	A
						1	
Sink	Send sync	Recv V_A	Ack V_A	Recv V_B,V_C	Ack V_B,V_C		Send Nack
Node A	Recv sync	Send V_A	Recv Ack	Recv V_C	Recv Ack		Recv Nack
Node B	Recv sync	Recv V_C	Recv Ack	Send V_B,V_C	Recv Ack		Recv Nack
Node C	Recv sync	Send V_C	Recv Ack	Send V_C	Recv Ack		Recv Nack
	1	2	3	4	5	6	7

time →

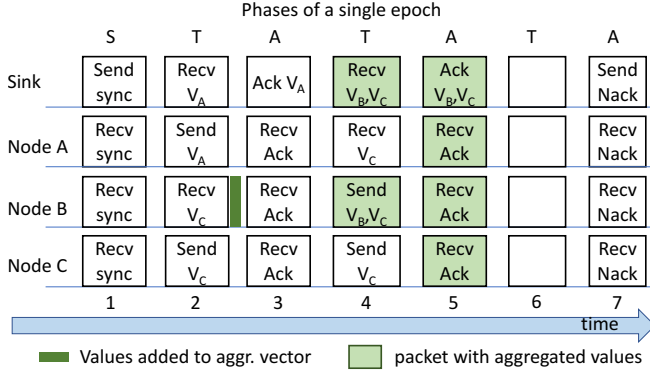■ Values added to aggr. vector   ▢ packet with aggregated values

Figure 2: Example of an epoch with three nodes with values to send and one sink, using *Arctium*. During phase 2, node B overhears a transmission from node C and combines C's value with is own. Compared to Crystal in Figure 1, *Arctium* requires one less T-A pair to complete the epoch.

aggregate (e.g. GM needs to keep track of the average of the values to compute the global estimate). (ii) During each T-phase, only one value from one node reaches the sink, while the other concurrent senders fail.

The goal is to allow nodes that did not manage to reach the sink to aggregate their own values with the values of neighbouring nodes. Subsequently, these nodes try to send their aggregated values. If they reach the sink, the sink acknowledges multiple values in a single phase and the total number of phases in the epoch is reduced.

Fitting aggregation into Crystal: Figure 2 shows an example epoch using *Arctium*. Our aggregation scheme leverages the T and A communication phases of Crystal. If, during a T-phase, a node does not try to transmit a value of its own (Node B, phase 2), it can overhear a neighbouring transmission and propagate it. At the end of the T-phase, the value that was overheard is stored in a vector. The node then aggregates that value with its own or other, previously aggregated values and tries to send the aggregate to the sink. The sink in our system acknowledges the aggregated information during a single A-phase, in response to receiving an aggregated message (phase 5). As a result, the total number of phases in an epoch is reduced. In this case, two phases less than the Crystal example in Figure 1.

Keeping track of values from multiple nodes: To track which nodes' values we have aggregated, packets in *Arctium* hold, apart from the value, a small vector with the IDs of the nodes whose values are taken into account in the aggregate. The size of this vector determines the maximum number of nodes whose values we can aggregate in a single message. Even though this vector increases the size of the packet and adds overhead, we show in Section IV that even with a small vector (two or three IDs) we have good opportunities for aggregation without adding excessive overhead. A node also keeps a local vector with the IDs and values that are part

of the aggregate. Keeping track of that vector allows nodes to add or remove values from the aggregate, based on the design choices described next.

Increasing the chances to aggregate: In order to give the opportunity for nodes that have values to transmit to overhear neighbouring transmissions and to combine them with their own, we introduce a *random back-off* during the T-phases. If a node has values to transmit, it will do so with a given probability p. Otherwise, the node will still be part of the T-phase but will only overhear and forward packets. Note that introducing such a back-off is not wasting opportunities for communication: only one node's value will reach the sink anyhow, so unless all nodes with data to send back-off simultaneously, there will be a successful reception at the sink.

Aggregation design choices: We now outline and motivate important design choices for the logic of when and how to aggregate. In the following, we denote with V_X the value that node X is trying to send to the sink.

Point 1: If node A overhears during a T-phase that its value V_A has been aggregated by node B, node A stops transmitting V_A and allows node B to send it instead. This helps to reduce the number of concurrent transmitters within each T-phase and reduces the size of the epoch.

Point 2: If node A, which has aggregated (V_A, V_B, V_C), hears an acknowledgment from the sink that contains (either standalone or part of an aggregated packet) the ID of B, A removes V_B from its vector and continues with the aggregated values of A and C. This allows us to keep the rest of the values that have been aggregated so far and only remove the ones that were acknowledged by the sink.

Point 3: Nodes do not combine one set of aggregated values with another. E.g., if node A with aggregated values (V_A, V_B, V_C), overhears a transmission from node D which has aggregated (V_D, V_B, V_E), node A will not attempt to combine the aggregates. This serves two purposes: (i) it avoids duplicate values (in the example above V_B would be counted twice) and (ii) it makes enforcing Point 2 above possible, because a node knows the individual contributions of the values that it is aggregating.

Point 4: If an aggregating node A overhears that node B is aggregating some similar IDs and B is a more successful aggregator (has a larger vector of aggregated values), node A will remove those common IDs from its own vector and allow B to take care of them. This helps to decrease the probability of cases where two nodes each have aggregated values which they cannot combine, due to Point 3.

Algorithm 1 summarizes the pseudo-code of the aggregation logic, in correlation with the points described above.

Properties of *Arctium*: We now argue about the following three claims regarding the behavior of *Arctium*. For readability, we first present them considering absence of packet losses and then we discuss the implications incurred by packet losses.

Algorithm 1: Packet structure and packet handlers at node U.

```
// The application payload
struct {
    Data: value
    Vector: idBuffer
} Packet;

// Local variables
```
U.idBuffer : a local vector of node IDs who's values have been aggregated
U.valueBuffer : a local vector of values that have been aggregated
U.aggregate : the current sum of all aggregated values
U.nodeId: current node's ID

Function *upon reception of packet P during the A-Phase*
```
    // Implements Point 2 (§ III-C)
```
    foreach *id* $\in$ *P.idBuffer* **do**
        if *id* $\in$ *U.idBuffer* **then**
            `remove` *id* and its value from *U.idBuffer* and *U.valueBuffer*
            U.aggregate = `Sum` (*U.valueBuffer*)
        end
    end
    With *probability p, back-off in the next T-Phase*

Function *upon reception of packet P during the T-Phase*
```
    // Implements Point 3 (§ III-C)
```
    if *P.idBuffer.size ==1 and idBuffer.notFull* **then**
        `add` *P.idBuffer* in *U.idBuffer*
        `add` *P.value* in *U.valueBuffer*
        U.aggregate = `Sum` of values in *U.valueBuffer*
    end

```
    // Implements Point 4 (§ III-C)
```
    if *(U has aggregated values) and (P.idBuffer.size* $\geq$ *U.idBuffer.size)* **then**
        foreach *id* $\in$ *P.idBuffer* **do**
            if *id* $\in$ *U.idBbuffer* **then**
                `remove` *id* and its value from *U.idBuffer* and *U.valueBuffer*
                U.aggregate = `Sum` (*U.valueBuffer*)
            end
        end
    end

```
    // Implements Point 1 (§ III-C)
```
    if *(U has not aggregated any value) and (U.nodeId* $\in$ *P.idBuffer)* **then**
        stop trying to transmit
    end

Claim 1: If Crystal delivers a value V_X to the sink, *Arctium* delivers it as well in the same epoch. In other words, *Arctium* guarantees delivery of all values.

Arctium makes sure no values are dropped in the following way. Consider a node A trying to send a value V_A to the sink. First, node A will not stop trying to send V_A until it is acknowledged by the sink or picked up for aggregation by another node B (see point 1). Second, if a node B has aggregated the value V_A of node A, node B will not remove it from its aggregation vector until it is acknowledged by the sink (see point 2) or it is picked up by a more successful aggregator (see point 4). In any case, until the value V_A is acknowledged by the sink, there is at least one node (either the originator of that value or the node(s) that aggregated it) that is responsible for that value and keeps trying to send it to the sink.

Claim 2: Arctium guarantees no duplicates to the sink.

This is because: (i) nodes remove the values that are acknowledged by the sink from their aggregation vector (see point 2) so that they are not sent to the sink again and (ii) we disallow combining one set of aggregated values with another (see point 3), as this could cause the values in the union of those sets to be aggregated twice.

Corollary 1: Arctium correctly monitors aggregate functions on the network.

From Claims 1 and 2, since every value will be delivered exactly once, *Arctium* can track distributive aggregates (such as *sum* and *count*) as well as algebraic ones (such as *average* and *variance*). Holistic aggregates [17] (such as *median* or *top-k*) are not covered by the method. *Arctium* can also track any function that fits into the GM framework (since GM aggregates the values from different nodes using the *average*).

Claim 3: With high probability, epochs in *Arctium* are not longer than the ones in Crystal.

In the absence of packet losses, at the end of every T-A pair, exactly one node's packet will reach the sink. In *Arctium* packets contain the aggregated values from at least one node, hence fewer or equal packets are required (less T-A pairs) for all nodes' values to reach the sink. Also, only the nodes that have values to transmit in this epoch participate in aggregation, i.e. we do not introduce new messages on the nodes that would otherwise not have values to send in this epoch. The only case where *Arctium* might introduce extra T-A pairs is when, during a T-phase, all nodes with values to send decide to back-off simultaneously. If N is the number of nodes with values to send in an epoch and p is the back-off probability, the total number of extra T-A pairs introduced this way in an epoch follows a Poisson binomial distribution with success probabilities $p^N, p^{N-1}, ..., p^2, p$. For example, for $p = 0.5$ and $N = 26$ the expected number of extra T-A pairs is close to 1, and the probability of having more than e.g. 4 extra pairs is less than 1.7% (by using Chernoff's bound). Moreover, even if extra T-A pairs exist in an epoch,

the reduction in T-A pairs by the use of aggregation is likely to counter their effect as can be seen experimentally in Section IV-C.

Now let's also consider packet losses for the cases discussed above. Claim 1 only holds with high probability as there is an unlikely scenario with loss of values if all of the following four conditions hold: (i) a node A has aggregated at least one value V_B from another node B, (ii) node B has delegated that value to A and stopped trying to send it to the sink, (iii) no other node has aggregated V_B and (iv) node A repeatedly fails to reach the sink. This scenario is extremely unlikely, given that all these four conditions must hold simultaneously and, as shown in [13] and later in Section IV-B, Crystal achieves high reliability, i.e. packet losses are very rare. Even then, node A can deliver the value at the next epoch, as a last resort.

Claim 2 can still hold in the presence of losses with a simple remedy. Duplicates might arise when a node that holds value V sends it to the sink but fails to receive the acknowledgment for that value, due to packet losses. In this case, value V might reach the sink more than once. This can be remedied if the sink keeps track of the values it has received. If it detects a duplicate value, it can resend the potentially lost acknowledgement. Claim 3 still holds with high probability in the presence of packet losses.

IV. EVALUATION

We implemented Geometric Monitoring and *Arctium* in Contiki [5], a well-known operating system for IoT applications. We targeted the TelosB platform that supports protocols such as Crystal that rely on synchronous transmissions. In this section, we assess the performance of our design based on its duty-cycle as well as with other metrics defined below.

A. Experimental Methodology

Experiment setup: We run our experiments in two settings, similar to the setup used in [28]: (i) A *full-system evaluation* on the Flocklab [22] testbed. Flocklab consists of 26 TelosB nodes deployed with a four hop topology. Using Flocklab, it is possible to test our design on a real deployment with realistic interference due to the presence of people and Wi-Fi signals. Moreover, Flocklab is, at the moment, the only publicly available testbed that still includes the TelosB nodes that support protocols such as Crystal. (ii) A *full-system simulation* on Cooja [24], a cycle-accurate simulator where the *whole network stack* is simulated in software at every node. We use Cooja to reproducibly uncover trends and insights, which we then validate with deployment in Flocklab. The default topology here is similar to the testbed. We also use Cooja to test larger topologies than the one available in Flocklab (see Section IV-C).

Data set and monitoring functions: We use the commonly-used Intel Lab data set [2]. We use the first day

Parameter	Explanation (X is S, T or A)	Value
N_X	number of Glossy transmissions in phase X	4
W_X	the maximum duration of phase X	12 (ms)
R	number of consecutive empty T-A pairs before the epoch is finished	3

Table I: Crystal's parameter values used in experiments.

of temperature readings from 26 nodes as sources of data for the nodes in the testbed. In the original dataset, nodes take a new reading every 31 seconds. In our experiments, we simulate the same period, i.e. *we scale the duty-cycle results to correspond to a period of 31 seconds*.

We experiment with two monitoring functions: the *variance* (also used in [27]) and the *average* of the temperature readings and use different values for the threshold we want to monitor (see Section IV-B). Unless otherwise noted, we will use the *variance* and a threshold $T = 2°C^2$.

Configuring Crystal: Crystal has many knobs that allow the protocol to operate on different topologies and network conditions, e.g. one-hop vs multi-hop networks, noisy vs interference-free networks. Those knobs also offer a configurable trade-off between the performance requirements, i.e. allow the user to favor energy efficiency in place of reliability and vice versa. We refer to the original publications of Crystal [12], [13] where the authors explain the significance and the methodology of choosing correct values for Crystal's parameters. In this work, we simply choose the set of parameter values presented in Table I that allows Crystal to operate in a reliable manner.

Metrics of interest: A key evaluation criterion is *duty-cycle (DC)*, i.e. the fraction of the total time that the radio is turned on. In some experiments we also report the *lifetime improvement*, i.e. the reduction in duty-cycle achieved through GM. Related to GM, we also report the *communication reduction* of GM in terms of the number of updates suppressed by the algorithm; it is a measure of the efficiency of GM, purely from the application point of view. Finally, we also measure the *loss rate*, in terms of the percentage of updates from individual nodes that fail to reach the sink.

Summary of the experiments: The rest of the evaluation section is organized as follows: in Section IV-B we present testbed and simulation experiments that summarize the performance of our Crystal and GM co-design, without using aggregation. In Section IV-C we focus on our aggregation scheme, *Arctium*, and show, through testbed and simulation experiments, how it manages to reduce the average duty-cycle.

B. Combining GM and Crystal: overall performance

We start with a real-life deployment on the Flocklab testbed, where we monitor the variance and the average of the temperature readings using different threshold values. We compare the performance against a baseline (same as

Method	Comm. Reduction	DC (%)	Lifet. Impr.	Loss Rate (%)
Baseline	1X	1.22	1X	0.34
GM (variance / T=0.5)	3.2X	0.54	2.26	0.03
GM (variance / T=1)	4.1X	0.46	2.65	0.00
GM (variance / T=2)	4.3X	0.44	2.77	0.00
GM (variance / T=3)	5.5X	0.38	3.21	0.10
GM (average / T=25)	87x	0.18	6.78	0.00

Table II: **Full system evaluation** on the Flocklab testbed, using GM on top of Crystal, for different monitoring functions and threshold values.
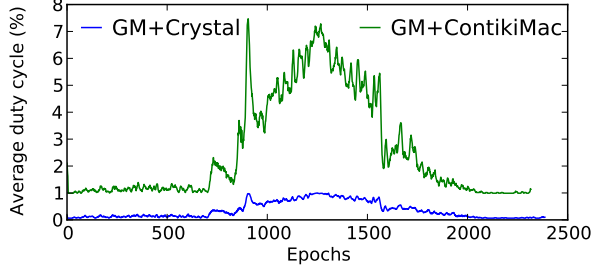
Figure 3: Duty-cycle during runtime when monitoring the variance with $T = 2$. The figure also includes results of using GM with ContikiMAC taken from [28].

in [26], [28]) that does not run GM and instead sends all measurements to the sink. However, in our case, the baseline uses Crystal as the underlying protocol.

Testbed experiments: Table II shows the results from the Flocklab deployment. The choice of monitoring function and threshold value plays an important role on the effectiveness of the GM and the communication reduction it achieves. When monitoring the variance, GM manages to suppress most of the nodes' values and communicates 3.2 to 5.5 times less than the baseline. What is important in this work though, is that, with the combination of GM and Crystal, that communication reduction is translated as lifetime improvement on the nodes. Our experiments report very low duty-cycle, down to 0.38%. This results in an up to 3.2 times lifetime improvement compared to the baseline, which is using Crystal without GM, an already very energy-efficient protocol. We also note that the system remains fairly reliable, with less than 0.34% loss rate. In the case of the average, GM suppresses most of the communication and the system has a duty-cycle of 0.18%.

Runtime behaviour: We now take a closer look at the dynamic behaviour of the above experiments and the evolution of the average duty-cycle during runtime. We have simulated one of the above experiments in Cooja (we chose the one with $T = 2$) and continuously report the average duty-cycle in Figure 3. To highlight the benefits of using GM with Crystal, compared to other network stacks, the figure also includes the equivalent results from [28] where GM is applied on top of ContikiMac [4], a mainstream network stack. The figure shows that, regardless of the protocol stack,
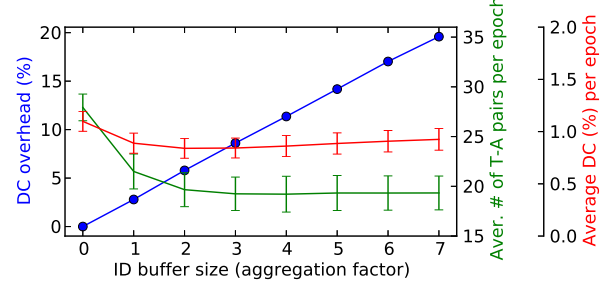

Figure 4: Collection of trends illustrating the effect of the aggregation factor (x-axis): (i) the per-packet DC overhead (blue trend), (ii) the average number of T-A pairs per epoch (green trend) and (iii) the average DC per epoch (red trend).

GM exhibits a highly dynamic behavior, with periods of low activity where most of the communication is suppressed and periods of high activity (usually when the monitored value is close to the threshold) where GM communicates more. However, the choice of communication protocol has a major effect on the average duty-cycle. Our design that combines GM and Crystal has an order of magnitude smaller duty-cycle and consumes a bare minimum amount of power when communication is low. On the contrary, mainstream networks stacks have a significant overhead which they have to pay even under low communication [28].

C. Arctium: In-network aggregation under heavy communication

In this section, we focus on our aggregation scheme *Arctium* and show that aggregation is an effective technique to reduce the already low duty-cycle presented in the previous section, even further. In all the experiments of this section, the back-off probability p was set to 0.5.

Effects of the aggregation factor: We start by presenting simulation results that illustrate the effects of the size of the aggregation buffer. As mentioned earlier in Section III, *Arctium* uses a node-ID buffer that is a placeholder for the IDs of the nodes whose values might be aggregated in a single packet. The size of the buffer is a parameter of our design and it affects the effectiveness of the aggregation. In the following experiment, every node in the network tries to send data to the sink (i.e. we adopt the baseline behavior, without the GM algorithm) to illustrate the effect of aggregation. Figure 4 shows the evolution of different trends as the size of the ID buffer (aggregation factor) increases in the x-axis. We now explain the significance of each trend.

The blue trend shows the added, per-packet overhead that comes from increasing the buffer size, compared to not having a buffer (aggregation factor 0). Since each packet has to statically reserve space for each entry in the buffer, the size of the packet increases linearly with the size of the buffer. This results in a linear increase in duty-cycle

	Aggregation factor (size of the buffer)						
	0	1		2		4	
Method	DC	DC	Impr.	DC	Impr.	DC	Impr.
baseline (+ *Arctium*)	1.24%	0.97%	1.27x	0.90%	1.38x	0.93%	1.33x
GM (+ *Arctium*)	0.54%	0.49%	1.10x	0.48%	1.13x	0.50%	1.08x

Table III: **Full system evaluation** on the Flocklab testbed, for different aggregation factor values.

overhead in order to receive and transmit those packets. Each added slot in the buffer adds approximately 3% extra overhead.

The green trend illustrates the benefits that come from aggregating values in a single packet: it reports the average number of T-A pairs required to complete an epoch. As the aggregation factor increases, the number of T-A pairs quickly drops, since a bigger buffer allows a packet to carry more information and deliver more values to the sink. Most noticeably, even using a single-slot ID buffer, i.e. going from aggregation factor 0 to 1, is enough to reduce the number of T-A pairs by 23%. The benefit saturates after an aggregation factor of 3 which indicates that, for this given topology, there are not many opportunities to aggregate more than 3 values before the sink receives every node's value.

The red trend reports the average duty-cycle per epoch. This metric factors in both the increasing overhead per packet (blue trend) and the decreasing number of T-A pairs per epoch (green trend). As a result, the average duty-cycle initially decreases and has a minimum when the aggregation factor is 2. At this point, using *Arctium* results in 23% smaller duty-cycle. After that point, the increasing packet overhead dominates and the duty-cycle increases slowly.

Testbed experiments: We support the above simulation results regarding *Arctium* with results from real deployments in the Flocklab testbed. Table III summarizes the duty-cycle reported for the baseline and GM, as we change the aggregation factor. In the GM experiments, we monitor the variance with a threshold of $T = 0.5$. We also report the improvement in duty-cyle, due to aggregation. For the baseline, the results validate the previous claims: aggregation further reduces the duty-cycle by a varying amount, up to 1.38x. *Arctium* also has a positive effect for the case of GM, where the already low duty-cycle (0.54%) is further decreased by up to 1.13x. Naturally, since GM communicates far less than the baseline, there are fewer messages to send to the sink and fewer chances to aggregate values. Still, the results show that *Arctium* brings improvement even in applications with aperiodic communication patterns.

Testing larger networks: Finally, we experiment with larger and busier topologies in the Cooja simulator. Figure 5 reports the average duty-cycle for topologies that range between 20 and 50 nodes. In each topology, every node is trying to send values to the sink. Overall, *Arctium* manages to reduce the
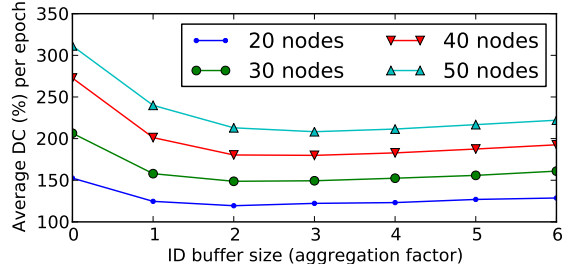

Figure 5: The average duty-cycle per epoch for different topologies, as the aggregation factor changes.

duty-cycle at each topology. The reduction ranges from 21% with 20 nodes where there are few packets to aggregate in the network, up to 33% with 50 nodes where there are more opportunities for aggregation.

V. RELATED WORK

Geometric Monitoring (GM): Since its original publication, GM has been extended in many ways, orthogonal to the design we consider in this work. It has been combined with sketches [10] and prediction models [11] that effectively track aggregates such as join and self-join sizes, while also taking the temporal evolution of the monitored values into account. A summary of the use of GM for query tracking in distributed streaming systems can also be found in [9]. This interest has been motivating also for the work in our paper.

In [15], the authors introduce shape sensitive geometric monitoring, that takes into account properties of the monitored function. Lazerson et al. [19] propose a method to approximate the monitored function to convex/concave components, so that it can be easily checked for violations, providing a good alternative to tackle the complexity of threshold checking we discuss in Section III-B. They also experiment with a high-end embedded platform and show that the method is lightweight. The approximation we propose in our work is orthogonal, in the sense that we leave the function intact and instead bind the local area that nodes have to keep track of.

GM has been studied in the general context of WSN from a high-level perspective. In [27], the authors present an adaption of GM that is designed for clustered topologies. In [3], GM is used for detecting outliers in the readings of wireless sensor nodes. In both of these lines of work, the network is only considered as a communication abstraction and practical system aspects are not studied. Differently, we take a full system perspective and consider a real network stack. The only work that considers deployment of GM on top of real networks stacks is in [28], which we compare against in Section IV-B.

Wireless protocols: Crystal is not the only modern network stack that relies on synchronous transmissions. Ferrari et al. [6] present a protocol that uses synchronous transmissions to support many communication patterns. Landsiedel et

al. [18] also use synchronous transmissions and augment them with in-network aggregation to achieve low power and excellent reliability. However, their approach focuses on duplicate-insensitive aggregates (e.g. *min* or *max*) and does not work out of the box for duplicate-sensitive aggregates (e.g. *sum* or *average*). Al Nahas et al. [1] extend the protocol with duplicate-sensitive aggregates. However, all previously mentioned protocols are not designed for cases where traffic is sparse and aperiodic. That is the main motivation why we chose Crystal to build our design upon.

Aggregation: In-network aggregation has been an active topic of research for many years. A large body of work studies aggregation as an application of gossip-based protocols. Friedman et al. [8] discuss different gossiping protocols. Jelasity et al. [14] present a decentralized aggregation protocol based on gossiping. Koldehofe [16] shows the effect of the buffer size in the performance of gossip-based protocols. Kuhn et at. [17] prove bounds and provide algorithms for holistic aggregates, specifically for distributed selection.

In the context of wireless sensor networks, Rajagopala et al. [25] survey different ways aggregation is used in WSNs, mostly by utilizing the network's architecture. Nath et al. [23] show how to approximately track duplicate-sensitive aggregates in WSNs. In this work, we present an approach that builds on top of a modern network protocol that has not been studied before in the context of in-network aggregation.

VI. CONCLUSION

We show how a general threshold monitoring framework (GM) can be co-designed together with a state-of-the-art wireless protocol (Crystal), for the problem of continuous threshold monitoring. Detailed results from testbed deployments show that the two approaches complement each other and are able to achieve a very low duty-cycle, up to 10x less than a mainstream network stack, which was the limiting factor in previous work. In particular, we present the way we orchestrate the communication of GM using the existing schedule of Crystal. We also introduce an efficient approximation for threshold checking that makes this co-design possible. Moreover, we extend our design to also exploit latent opportunities for aggregation in Crystal and improve the lifetime of applications that are monitoring network aggregates. Our aggregation scheme, called *Arctium*, allows nodes to overhear and correctly aggregate neighbouring node's values and manages to further improve the lifetime of the nodes by up to 1.13-1.38x. Our results show that applications such as GM can have practical value for real IoT deployments, coupled with modern networks stacks that unlock the full potential of the application. Our code is available online: https://github.com/mpastyl/Arctium.

REFERENCES

[1] B. Al Nahas, S. Duquennoy, and O. Landsiedel. Network-wide consensus utilizing the capture effect in low-power wireless networks. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys, 2017.

[2] P. Bodik, W. Hong, C. Guestrin, S. Madden, M. Paskin, and R. Thibaux. Intel Lab Data, 2004.

[3] S. Burdakis and A. Deligiannakis. Detecting Outliers in Sensor Networks Using the Geometric Approach. In *IEEE International Conference on Data Engineering*, 2012.

[4] A. Dunkels. The ContikiMac radio duty cycling protocol. Technical report, Swedish Institute of Computer Science, 2011.

[5] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th IEEE Int'l Conf. on Local Computer Networks*, 2004.

[6] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele. Low-power wireless bus. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, SenSys, 2012.

[7] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh. Efficient network flooding and time synchronization with glossy. In *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*, April 2011.

[8] R. Friedman, D. Gavidia, L. Rodrigues, A. C. Viana, and S. Voulgaris. Gossiping on manets: The beauty and the beast. *SIGOPS Oper. Syst. Rev.*, 41(5), Oct. 2007.

[9] M. Garofalakis. Approximate geometric query tracking over distributed streams. *IEEE Data Eng. Bull.*, 2015.

[10] M. Garofalakis, D. Keren, and V. Samoladas. Sketch-based Geometric Monitoring of Distributed Stream Queries. *Proc. VLDB Endow.*, 2013.

[11] N. Giatrakos, A. Deligiannakis, M. Garofalakis, I. Sharfman, and A. Schuster. Prediction-based geometric monitoring over distributed data streams. In *ACM SIGMOD International Conference on Management of Data*, 2012.

[12] T. Istomin, A. L. Murphy, G. P. Picco, and U. Raza. Data prediction + synchronous transmissions = ultra-low power wireless sensor networks. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems*, SenSys, 2016.

[13] T. Istomin, M. Trobinger, A. L. Murphy, and G. P. Picco. Interference-resilient ultra-low power aperiodic data collection. In *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN, 2018.

[14] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3), Aug. 2005.

[15] D. Keren, I. Sharfman, A. Schuster, and A. Livne. Shape Sensitive Geometric Monitoring. *IEEE Trans. Knowledge and Data Eng.*, 2012.

[16] B. Koldehofe. Buffer management in probabilistic peer-to-peer communication protocols. In *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings.*, Oct 2003.

[17] F. Kuhn, T. Locher, and R. Wattenhofer. Distributed selection: A missing piece of data aggregation. *Commun. ACM*, 51(9), Sept. 2008.

[18] O. Landsiedel, F. Ferrari, and M. Zimmerling. Chaos: Versatile and efficient all-to-all data sharing and in-network processing at scale. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys, 2013.

[19] A. Lazerson, D. Keren, and A. Schuster. Lightweight monitoring of distributed streams. *ACM Trans. Database Syst.*, 43, July 2018.

[20] J. Lee, B. Bagheri, and H.-A. Kao. A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Manufacturing Letters*, 3, 2015.

[21] K. Leentvaar and J. Flint. The capture effect in fm receivers. *IEEE Transactions on Communications*, 24(5), 1976.

[22] 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 *2013 ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2013.

[23] S. Nath, P. B. Gibbons, S. Seshan, and Z. R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, SenSys, 2004.

[24] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with cooja. In *31st IEEE Conf. on Local Computer Networks*, 2006.

[25] R. Rajagopalan and P. K. Varshney. Data-aggregation techniques in sensor networks: A survey. *IEEE Communications Surveys Tutorials*, 8, Fourth 2006.

[26] I. Sharfman, A. Schuster, and D. Keren. A geometric approach to monitoring threshold functions over distributed data streams. In *ACM SIGMOD Int'l Conf. on Management of Data*, 2006.

[27] I. Sharfman, A. Schuster, and D. Keren. Aggregate threshold queries in sensor networks. In *IEEE Int'l Parallel & Distr. Process. Symp.*, 2007.

[28] C. Stylianopoulos, M. Almgren, O. Landsiedel, and M. Papatriantafilou. Geometric monitoring in action: a systems perspective for the internet of things. In *2018 IEEE 43rd Conference on Local Computer Networks (LCN)*, 2018.

APPENDIX

A. Geometric Monitoring

In this section of the Appendix, we give a more detailed summary of the Geometric Monitoring method than the one found in Section II-A. A more detailed analysis, as well as the proofs can be found in the original publication by Sharfman et al. [26].

Assume a network of N nodes, with sensor readings $\vec{v_1},...,\vec{v_N}$, called *local statistics vectors*. Those vectors consist of one or more variables that each node monitors and vary over time. These vectors are only known locally, but sporadically a node n_i will broadcast its $\vec{v_i}$ to every other node. The last broadcasted value from n_i is denoted as $\vec{v_i'}$. The weighted average of the local vectors is called the *global statistics vector*.
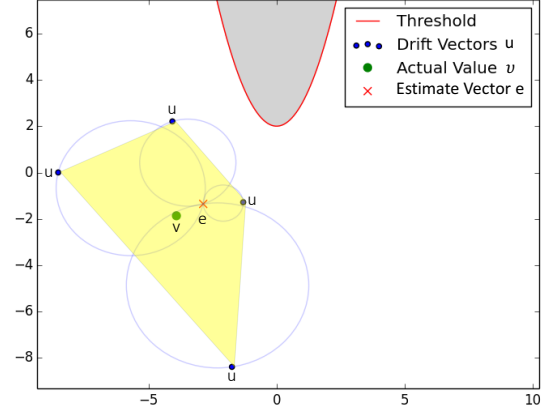

Figure 6: An example illustrating the GM method.

$$\vec{v} = \sum_{i=1}^{N} w_i * \vec{v_i} \qquad (1)$$

Similarly, the weighted average of the last broadcasted values is called the *estimate vector* ($\vec{e}$) and it is known to all nodes. Given a function f and a threshold T, we want to continuously monitor whether or not the value $f(\vec{v})$ is under the threshold. Equivalently, we want to always know whether or not $\vec{v}$ lies in an are where the function takes values below the threshold.

When a node measures a new set of sensor readings, its local statistics vector will drift ($\vec{\Delta v_i} = \vec{v_i} - \vec{v_i'}$). The *drift vector* $\vec{u_i}$, defined as the displacement of the estimate vector because of the new drift, i.e. $\vec{u_i} = \vec{e} + \vec{\Delta v_i}$, can be computed locally without communication. Figure 6 shows an example of the method, also depicting the values defined above.

The convex hull of the drift vectors (yellow area) is defined as the set of all the convex combinations of $\vec{u_i}$ ($\sum \theta_i \vec{u_i}$). As such, it is clear that the weighted average of the drift vectors (defined similarly to Equation 1) would be part of this set. With simple substitution, one can also see that the global statistics vector is equal to the weighted average of the drift vectors and thus it must also lie in the convex hull of the drift vectors. Thus, as long as the convex hull does not cross the threshold (i.e. lies in the white area), $\vec{v}$ is also guaranteed to not have crossed the threshold. However, nodes cannot locally determine the convex hull, as that would require knowing all $\vec{u_1},...,\vec{u_N}$.

This is where the final part of the method comes into play. Let each node create a sphere locally, centered at $\frac{\vec{e} + \vec{u_i}}{2}$ with a radius of $\frac{\vec{e} - \vec{u_i}}{2}$. This is possible since $\vec{u_i}$ is known to n_i and $\vec{e}$ is the same across all nodes at a given time. Sharfman et al. [26] prove that the union of those spheres strictly covers the convex hull. Therefore, a node only needs to track whether its locally computed sphere crosses the threshold. If yes, it will send its local vector to everyone, subsequently updating the estimate vector; else, it can remain quiet.