Singlehop Collaborative Feedback Primitives for Wireless Sensor Networks

Murat Demirbas Onur Soysal Muzammil Hussain
Department of Computer Science and Engineering
University at Buffalo, SUNY, Buffalo, NY, 14260
{ demirbas | osoysal | mh69}@buffalo.edu

Abstract

To achieve scalability, energy-efficiency, and timeliness, wireless sensor network deployments increasingly employ in-network processing. In this paper, we identify *singlehop feedback collection* as a key building block for in-network processing applications, and introduce two basic singlehop primitives, *pollcast* and *countcast*. The key idea behind our primitives is to exploit the *receiver-side collision detection* information at the MAC-layer to speed-up collaborative feedback collection. Using pollcast, a node can get an affirmation about the existence of a node-level predicate P in its neighborhood in constant time by asking all nodes where P hold to reply simultaneously. Using countcast, a node can get an approximate count of the number C_P of such nodes in $log(C_P)$ time. We have implemented pollcast and countcast on Tmotes using Chipcon 2420 radio. Our results show that these primitives are indeed lightweight, resilient, and effective. Our paper is also the first time receiver-side collision detection is achieved in a practical manner for Chipcon 2420 radio.

1 Introduction

Early deployments of wireless sensor networks (WSNs) have been mostly limited to passive data collection, where sensor readings from the network are relayed towards a basestation for storage and processing [2, 22]. In order to cope with the bandwidth/energy consumption and latency concerns associated with this centralized approach, *in-network information processing* has been advocated and widely adopted [1, 12, 14]. In-network processing exploits the computation capability of sensor nodes to process data locally in the network close to where it originates. Two main use-cases of this approach are 1) to summarize and reduce the data transmitted to the basestation, and 2) to perform decisions locally to avoid contacting the basestation for each decision. An example of the first is the aggregation of data and filtering of false-positives and duplicates, reducing the energy wasted in routing these packets all the way to the basestation only to be disposed there. An example of the second appears in intruder-interceptor applications, where having a leader responsible for the current intruder detection is critical for maintaining a tracking structure [7, 9, 15]. [1]

Although there have been many efficient "point solutions" [1, 8, 14] to the problems that appear in the context of these two use-cases, there has been no effort to address the issue by developing efficient and general collaboration primitives. In contrast to devising point solutions, designing such primitives would improve reusability and integration, and provide a unified framework for standardization of in-network processing protocols.

Contributions and overview. We identify collaborative feedback collection from a singlehop neighborhood as a key building block for both use-cases of in-network processing. We observe that wireless broadcast has many useful features for facilitating collaboration. Firstly, broadcasting is atomic: that is, all recipients receive a broadcast at the same time, which is useful for synchronizing the nodes in singlehop for building a structured feedback collection. Secondly, broadcast allows receiver-side collision detection (RCD): that is, a snooping node can detect collisions of messages, which is useful for extracting feedback from multiple nodes in a quick manner. Exploiting these features of wireless broadcast, we propose two efficient and lightweight singlehop collaborative feedback primitives, *pollcast* and *countcast*.

Using pollcast, a node can take a quick poll from its neighborhood by asking all nodes with a certain property to reply simultaneously. An operation starts with a "poll" phase, where the initiator broadcasts a poll message of the form "does there exist any node with property P?". The initiator then moves on to the second phase, the "vote" phase, to listen for the responses to its poll. If the initiator hears silence in the vote phase, it concludes that the polled predicate P does not hold for any node. Otherwise, if there is a response or there are multiple

[1]In this scenario, in-network processing is required even for correctness, as it has been shown that for satisfying optimality constraints, the latency with which an interceptor requires information about the intruder it is tracking depends on the relative locations of the two: the closer the distance, the smaller the latency [4].

responses (in which case the initiator detects a collision via RCD), the initiator concludes that P holds for some nodes. Thus, regardless of the number of nodes that need to reply, pollcast completes in $O(1)$ time.

Some applications of pollcast operation are false-positive suppression, clustering, and the querying of the neighborhood for debugging purposes. A desirable and obvious enhancement to the pollcast operation is to return an approximate count of the number C_P of nodes a property P holds. This enhancement would be useful for querying of the neighborhood for classification of an intruder (say as a soldier, car, or tank as in Exscal [3]) by counting the detections in the neighborhood. Our countcast primitive is in this sense an enhancement over the pollcast operation. Countcast estimates C_P by what amounts to invoking pollcast for P successively, such that at each invocation voters may decide to stop participating further based on the outcome of their coin tosses. This effectively halves the number of responders at each invocation until a round is reached where no votes are cast. Countcast approximates C_P to be 2^k where k is the number of rounds it took the votes to decay completely. After estimating C_P the initiator may use the likelihood probabilities of predefined events and Bayesian inference to classify the result more accurately. Countcast completes in $O(\log C_P)$ expected time.

We have implemented pollcast and countcast primitives on the Tmotes [17] using the popular Chipcon 2420 radios. In addition, we have implemented pollcast and countcast under a WSN simulator [19] to be able to perform more controlled experiments for a larger scale networks and compare our primitives with other protocols in detail.

Finally, our paper is the first time RCD feedback is achieved for the popular CC2420 radios in a practical manner. Our experiments indicate that our RCD implementation has around 100% completeness and 0% false-positive detections. Our RCD technique is easily achievable at the MAC layer in software and does not require any modification to the physical layer or the wireless radio hardware.

Singlehop wireless broadcast has recently been identified as a narrow-waist suitable for standardization efforts in the WSNs [6]. Our efficient and lightweight singlehop collaborative feedback primitives for supporting in-network processing will help boost these standardization efforts. Researchers working on cooperative control may become end-users of our primitives, since pollcast and countcast are suitable for a control-theoretic framework of periodically collecting information about the state of the system and imposing a compensated control over the system.

Outline. We discuss related work in Section 2. In Section 3, we present our RCD implementation at the MAC layer and provide performance results. We present our pollcast operation in Section 4 and our countcast operation in Section 5.

2 Related Work

Work on collision detection. The feasibility of collision detection for CC1000 (mica2) radios has been demonstrated in [25] for a limited context (for certain capture/shadowing effect scenarios). The success rate of the preamble-based collision detection used in [25] drops quickly for more than two simultaneous senders. Our RCD implementation is based on receiver-side carrier sensing and is more general and inclusive than preamble-based RCD. Several existing MAC layers, such as B-MAC [18], already support the carrier-sensing capability required for our collision detector.

A recent empirical study on CC1000 radios [20] linked the successful message reception in the presence of concurrent transmissions to the signal-to-interference-plus-noise-ratio (SINR) exceeding a critical threshold. The results also showed that it becomes harder to estimate the level of interference in the presence of multiple interferers, and that the measured SINR threshold generally increases with the number of interferers. As this imply, CC1000 performs poorly in the presence of more than a couple of concurrent transmitters. Our preliminary experiments find that the radio behavior for CC2420 under concurrent transmissions is more resilient. CC2420 radio is able to receive a message successfully among a set of several concurrently transmitted messages, due to the direct-sequence spread spectrum (DSSS) which renders the CC2420 radio more resistant to interference. Thus, in CC2420 radio, SINR threshold for successful message reception becomes harder to define and it becomes harder to correlate RSSI with the successful message reception.

Singlehop programming abstractions. Several programming abstractions have been proposed for WSNs [5, 13, 23, 24].

The Tenet project [13] proposes a tiered WSN architecture with small-form-factor motes and more powerful master nodes. Tenet asserts that complex application logic should be implemented only on the masters. Applications running on masters task motes, and motes just communicate back to the masters the results from these tasks. Our singlehop polling and collaboration primitives would be instrumental for the masters in Tenet for quick and ad hoc feedback collection from the motes on-demand.

In an effort to simplify the adoption of distributed algorithms for WSNs in terms of a neighborhood abstraction, Hood [24] provides an API that facilitates exchanging information among a node and its neighbors. For example, Hood can define a one-hop neighborhood over which light readings are shared. Beneath the API, Hood automatically discovers neighbors and caches the values of their attributes periodically, while simultaneously sharing the values of the node's own attributes. Similar to Hood, abstract regions [23] and TeenyLime [5] propose mechanism for discovery and sharing of data (structured in terms of tuples) among sensor nodes.

Using the information exchange mechanisms proposed in these abstractions [5, 23, 24], it may be possible to achieve a constant response time to a query by performing periodic state exchange among neighbors behind the curtains. However, a big problem facing these approaches is to decide on the frequency of this exchange. If the exchange is done infrequently, the query will be answered using stale data. (This is especially problematic for real-time applications such as intruder-interceptor applications.) If the exchange is done frequently, a lot of traffic is generated wasting precious energy and bandwidth. In contrast to these work that deal with state exchange among nodes, our focus in pollcast and countcast primitives is to provide a lightweight and efficient framework for on-demand binary feedback collection from neighbors.

3 Receiver-side Collision Detection

We present how receiver-side collision detection (RCD) can be implemented for CC2420 radios in Section 3.1, and provide an evaluation of our RCD implementation in Section 3.2.

3.1 RCD Implementation

Below we discuss pros and cons of three possible approaches to RCD implementation.

Received-signal-strength-indicator (RSSI) based collision detection depends on monitoring the RSSI information from the radio frequently (i.e., for every byte) and looking for patterns that imply the existence of a collision. RSSI-based RCD is a low-level and general technique for collision detection, but it brings additional processing burden (due to the frequent interrupts it generates for RSSI processing) on the limited CPU of the sensor nodes. Moreover, compared to the CC1000 radios where the correlation with interference and SINR is observable [20], achieving RSSI based collision detection is much harder for the CC2420 radios.

Cyclic-redundancy-check (CRC) based collision detection depends on checking the CRC bits of received messages, and raising a collision detection upon encountering a bad CRC bit. CRC is well-engineered and reliable, however, CRC-based RCD is applicable only for the cases where radio is locked to a certain message and preamble and packet frame are received. Thus, CRC based RCD is not general enough for detecting all type of collisions.

Carrier sensing based collision detection depends on sensing the medium for ongoing transmissions. While carrier sensing is widely used by transmitters in wireless networks with CSMA MAC layers (including IEEE 802.11, IEEE 802.15.4, and all of the WSN MAC protocols), we adopt this technique for use by the receiver to detect collisions. The CC2420 radio exports a Clear-Channel-Assessment (CCA) signal for the purposes of carrier-sensing. CCA is well-engineered and robust, and is calculated by the radio chip based on a window of RSSI readings and thresholding. Since CCA has the additional benefit of radio-level support, CCA obviates the need to involve the CPU in collision detection, and is simpler and much more reliable than RSSI. The challenge with CCA signal is that, it is generated only after a transmission is scheduled to the radio, so for implementing RCD using CCA we manipulate the CC2420 radio to perform CCA also in the idle state. Using CCA-based collision detection we are able to detect collisions even when no intelligible packet information (such as the preambles) is received due to interference.

We implement both CRC- and CCA-based RCD for CC2420 radios and compare their performances in the next section.

3.2 RCD Experiments

To evaluate our RCD implementation, we use a setup consisting of upto 6 Tmote-Invent nodes [17] with CC2420 radios. One of these motes is designated as the poller, and the remaining motes are programmed as voters. In each experiment, the poller broadcasts a start message, upon which the voters transmit a reply immediately (without performing CSMA as per our modification to the TinyOS MAC layer). We initially start with two voters and repeat each experiment at least 1000 times, before increasing the number of voters by one to investigate the RCD performance under a more contended scenario. Figure 1 presents our findings.

CRC-based RCD combined with successful reception of one of the votes achieves upto 100% detections in the 2 and 3 voters case. In the 4 and 5 voter cases, the shadowing effect decreases as the power-sum of other transmissions create excessive interference for successful reception of any single message. So CRC-based detections and successful receptions of one vote decrease for these cases, since the probability of successfully locking to a preamble decreases with the degradation in shadowing effects. These results show that CRC-based RCD is insufficient for detecting most collisions.

CCA-based RCD achieves a thorough detection of collisions and is not noticeably affected by multiple voters as

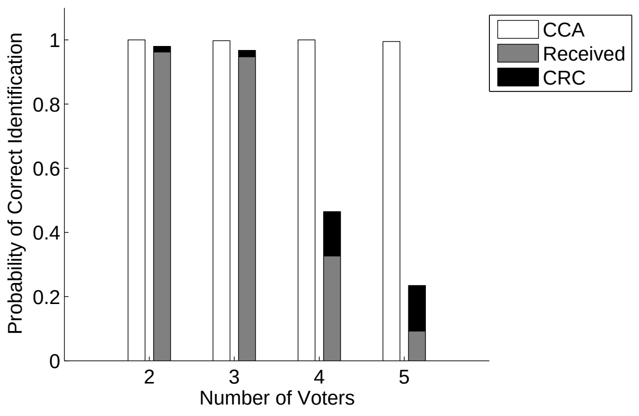

Figure 1: Collision detection performance

Figure 1 shows. We repeat CCA-based RCD experiments with varying the distance between the motes and achieve similar very good completeness results for collision detection.

Finally, we test the false-positive ratio of our RCD implementation by repeating the experiments with 0 voters. In our experiments, CCA-based detection has reported less than 1% false positive in all the setups we experimented with, so we conclude that the false-positives do not constitute a problem for our RCD implementation.

4 Pollcast Operation

We present the pollcast implementation in Section 4.1 and applications of pollcast in Section 4.2. We present experiment results on the Tmote platform and Prowler simulator in Section 4.3.

4.1 Pollcast Implementation

As we outlined in the Introduction, our pollcast operation consists of two consecutive phases: poll-vote. In the poll phase, an initiator may start a pollcast operation for predicate P by broadcasting a $\textsc{Poll}_P$ message. In the vote phase, the initiator switches to listen for the votes: receiving a VOTE or a collision implies that there exist some nodes with the polled property P, whereas silence means that no node satisfies P. Any node receiving a $\textsc{Poll}_P$ message in the poll phase should vote accordingly in the vote phase. A VOTE broadcast is performed only if the answer is "yes" for the node-level predicate P. The nodes communicate a "no" answer by keeping quiet in the vote phase.

The poller can limit its poll to a subset of the neighborhood by including a list of intended participants in the $\textsc{Poll}_P$ message (otherwise, all neighbors are included in the polling by default). This participant list feature is useful because, using the result of a previous poll, the poller may adaptively select a subset of participants for its next poll to narrow in on a clue. This way incremental searching for a condition as in the 20 questions game may be possible, but we do not explore this path any further in this paper.

Pollcast operations in multihop networks. Even though pollcasts are singlehop operations, when they are executed in a multihop network, interference from neighboring regions in the form of hidden terminal problems are unavoidable. For the sake of simplicity, we assume "atomicity" of pollcasts in a 2-hop region. More specifically, we assume that when a pollcast is in progress, there cannot be another simultaneous pollcast within 2-hops of this pollcast. This assumption is motivated by practical reasons. Using this assumption we are able to keep our pollcast implementation extremely lightweight and short in duration. In contrast, trying to cope with collision of poll messages would require either a globally synchronized rounds approach (as in [10, 11]) or introduction of several new control messages (such as an RTS/CTS handshake with every neighbor as in [21]) and the overhead may defeat the purpose: lightweight singlehop collaborative feedback!

Our atomicity assumption is reasonable for low traffic WSN deployments. In most WSN deployments the network is idle for most of the time. For the case of bursty triggering of the pollcasts (say, for example, due to detection of an intruder in the area), we rely on CSMA to arbitrate our lightweight pollcast operations in singlehop. Although we assume atomicity in the design and presentation of pollcast, our experiment and simulation results investigate the effects of collisions and hidden terminal problems on the consistency of pollcasts under low, medium, and heavy traffic loads.

4.2 Pollcast Applications

Several applications benefit from pollcast. Pollcast enables a node to corroborate its estimates about the environment with the neighboring nodes. One practical example is in suppressing false-positive detections. Due to inexpensive sensors, false-positives are frequent occurrences in WSN deployments. For example, in the "Line in the sand" [2] and "ExScal" [3] applications, sensors would often have false-positive detections (due to heat drifts and sunlight in out-door environments for PIR and due to noise for magnetometer). False-positives cause problems as they use up the precious resources on the network. A false positive detection is forwarded over several hops until it reaches a basestation node or a clusterhead which can decide that this detection is an outlier and drop it.

Pollcast operation is useful for filtering out false-positive detections effectively. A node that has a detection (and that has not yet heard any other detection from neighboring nodes) may incite a pollcast operation to ask neighbors if they also detected the alleged phenomena P. This is a yes/no question and can be executed very quickly using pollcast. Only if the answer is affirmative (i.e., there is activity in the vote phase) the initiator of the poll notifies the basestation about its detection.

Other applications of pollcast follow from the second use-case scenario of in-network processing, namely in-network control. A good example for this case is barrier synchronization. In WSNs barrier synchronization is useful for synchronizing operations in each cluster. For example, the clusterhead can query the member nodes as to whether they are finished with the current phase of the operation before the cluster as a whole can move to the next phase in the operation. Another example of in-network control is leader election. Even though an object is detected by many nodes at the same time, it is important to elect a leader primarily responsible for the object. The leader can then process and send messages, keep track of the trajectory of the object, and hand it off to the next leader. It may be advantageous to select the leader based on the the strength of its detection. This can be checked and established simultaneously, by making the initiator do a pollcast by announcing its detection and challenging anyone with a greater detection to vote. If no node votes, the initiator announces its leadership at the end. Electing leaders is also important in clustering applications. Several metrics may be used in the election of the clusterhead. For example, an initiator may check to ensure that there are no other clusters (clustermembers) in its singlehop [8].

4.3 Pollcast Experiments

For performing large-scale and controlled experiments, we have implemented pollcast in Prowler [19], a MATLAB based event driven simulator for distributed WSNs. We use Prowler to simulate the radio transmission/propagation/reception delays of Tmotes, including collisions in WSNs.

We compare pollcast with a strawman polling protocol *naivepoll*. In naivepoll, after the initiator broadcasts a poll message for a predicate P, the neighbors where P hold send their votes via CSMA. This, of course, means that if all the voters are not within singlehop of each other, there may be collisions of votes at the poller. Unlike pollcast, naivepoll does not employ RCD to learn from collision of votes.

We simulate pollcast over a 10x10 grid of 100 nodes. Each node has 8 neighbors (except the boundary nodes which may have 3-5 neighbors). We perform the simulations varying the number of concurrent pollers from 1 to 10 in unit steps and then incrementing from 20 to 60 in steps of 10. All elected pollers for a given simulation run try to execute their polls in the beginning of the run, with only CSMA arbitrating between these pollers. For each simulation run, we also randomly initialize the P predicates at the nodes, so that the number of voters for a poll may vary randomly.

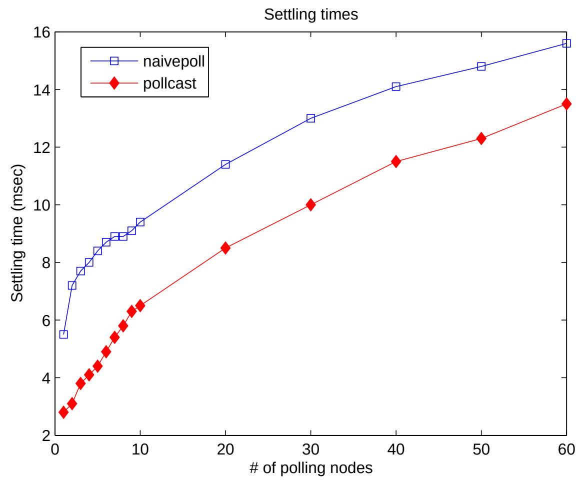

Figure 2: Settling time

Settling time of a simulation run is defined as the duration between the first and last message sent in the simulation run. The settling time graph in Figure 2 show that pollcast takes less time to complete than naivepoll. This is expected as pollcast uses the backoffs in CSMA only for the poll messages, but in naivepoll there are backoffs for both poll and vote messages. A pollcast operation in isolation is about 3msecs, and as the figure shows, for low traffic case the settling time of pollcasts in the WSN is around 3 to 5 msecs.

Figure 3 investigates the loss of poll messages at the receiver nodes and finds that a significant number of poll messages are lost in both protocols due to the hidden terminal problem. Pollcast seems to lose more poll messages because it sends vote messages without any carrier sensing. So in pollcast a vote message may be sent even when another poll message is being transmitted in the neighborhood.

In Figure 4 we compare naivepoll and pollcast for inconsistencies: when no vote is received at the poller whereas in reality the poller has at least one neighbor where P holds. In naivepoll an inconsistency occurs when all votes collide at the poller. Since naivepoll does not employ RCD, it cannot deduce the *existence of at least one vote* in this case. In pollcast an inconsistency occurs only when the poll message is lost on *ALL* the voters. When

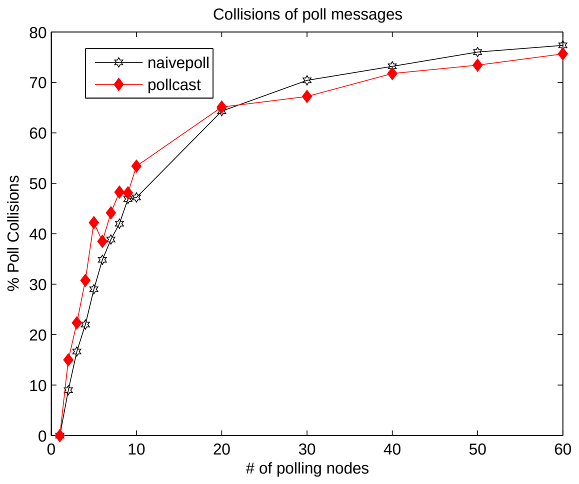

Figure 3: Poll message collisions

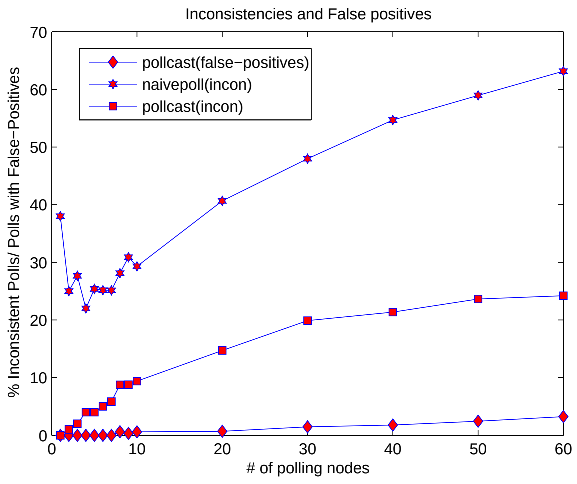

Figure 4: Polling inconsistencies

comparing Figure 4 with Figure 3 for the pollcast protocol, one should keep in mind that the loss of poll messages in Figure 3 is for *any* neighbor, while Figure 4 indicates the loss of poll messages on *all* the voters. Hence we find that the inconsistencies for pollcast is always less than 1/3 of that of naivepoll. Especially for the low traffic case, the inconsistencies in pollcast is almost negligible.

Figure 4 also shows the number of false positives for pollcast. False positives occur when a poller detects at least one vote (collision of votes) when in fact there were no votes for it. This may happen in pollcast because in the high traffic case a poller may misinterpret a collision of two other polls as a vote for its poll. Our experiments show that the number of false-positives for pollcast is very low.

5 Countcast Operation

We discuss the countcast implementation in Section 5.1 and applications of countcast in Section 5.2. We present experiment results in Section 5.3.

5.1 Countcast Implementation

A desirable and obvious enhancement to pollcast is to return an approximate count of the number C_P of nodes the property P holds. Countcast operation achieves this extension. A countcast operation consists of a poll phase followed by max number of vote phases.

An initiator starts a countcast for predicate P by broadcasting a $\textsc{CountReq}_P$ message in the poll phase. Any node where P holds should schedule to transmit a vote in one of the upcoming vote phases upon receiving a $\textsc{CountReq}_P$ message in the poll phase. This scheduling is done with an exponentially decaying probability. Each voter selects a phase k to vote with probability 2^{-k}. This way the expected number of voters is halved at each subsequent vote phase. We remove unrealistically large k selections using the upperbound max (e.g., assuming there can be at most 32 nodes in singlehop, $max = 5$ can be enforced).

The initiator listens for the votes in the vote phases. When the initiator detects a silent vote phase k, it estimates the number of nodes that initially voted as 2^k. A promising approach we will investigate in future work is to estimate C_P by using the distribution of the collisions in the max rounds instead of simply estimating C_P as 2^k. Moreover, it is possible to adjust the base 2 used in the estimation of C_P and improve the sensitivity of counting by tuning the voting probability of the nodes.

5.2 Countcast Applications

The countcast operation can be used for implementing classification of the intruder in the Line in the sand [2] and Exscal [3] applications. The basic idea is similar to that of false-positive elimination using pollcast. Here, the initiator also counts the nodes that made the same detection using countcast, and decides on the classification of the intruder using this result. After estimating C_P the initiator may use the likelihood probabilities of predefined events and Bayesian inference to classify the result into one of these categories more accurately.

Leader election can be improved using countcast. One improvement may be that the number of detections in singlehop should be above a threshold for the node to become a leader. Clustering applications can also be improved using countcast, enabling the initiators to look for at least a minimum number of members before declaring themselves as clusterheads.

5.3 Countcast Experiments

We have also implemented countcast in Prowler [19], and here we present simulation results comparing countcast with naivepoll and another protocol, *tdpoll*. In the tdpoll protocol, the poller includes the IDs of its neighbors in the poll message and the voters use the order of their IDs to time their vote to the poller to avoid the collision of votes at the poller.

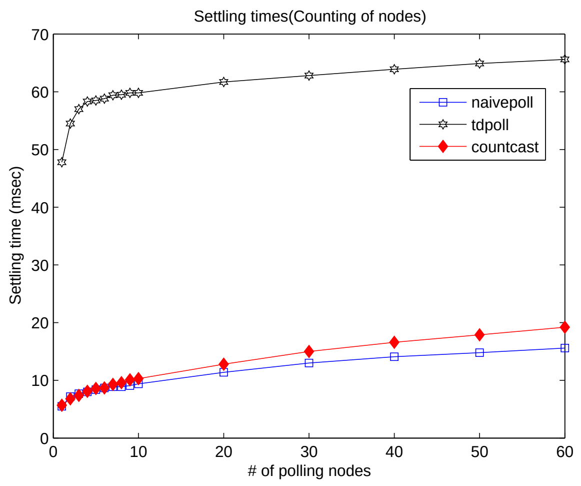

Figure 5: Settling time

In Figure 5 we see that tdpoll takes much more time for completion than naivepoll and countcast. This is because tdpoll has an operation time that is on the order of the number of voters, whereas countcast has an operation time on the order of logarithm of the number of voters, and naivepoll has an operation duration of roughly 2-3 message lengths in addition to the inherent carrier sensing backoff duration.

Figure 6 shows that the loss of the poll message is roughly the same for the three protocols. We can attribute the slightly higher number of poll collisions in countcast to the fact that the vote messages, which are sent without carrier sensing, being capable of disrupting poll messages.

In Figure 7, we compare the three protocols for the counting inconsistencies. For all three protocols the counting inconsistency is defined the same way for fairness of comparisons. An inconsistency is said to occur when the estimate c_P of the protocol is not within $\lfloor \log C_P \rfloor$ and $\lceil \log C_P \rceil$. (For the case when C_P is an exact power of 2, the lowerbound and the upperbound for consistency are

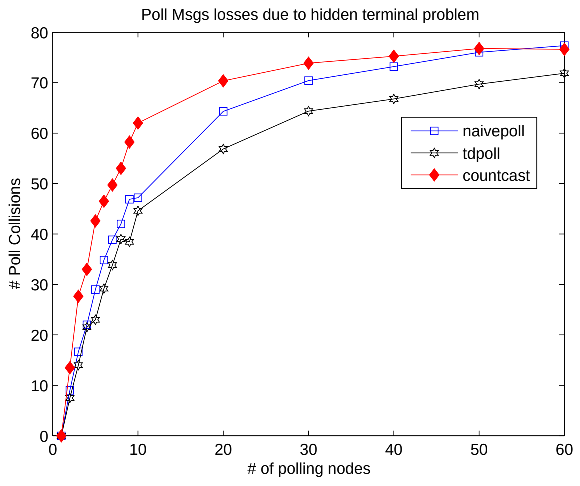

Figure 6: Poll message collisions

defined as $(\log C_P) - 1$ and $(\log C_P) + 1$.) Naivepoll is found be the most inconsistent with the percentage of inconsistent polls starting with 60% with just one poller in the network, worsening as the number of pollers increases. The reason naivepoll has high inconsistency even with a single poller is because the votes may collide at the poller and the poller does not have any mechanism to extract useful information from the collided votes. Countcast is better than naivepoll in estimating the number of voters. As the number of pollers increase the accuracy of countcast still remains quite good. In fact countcast surpasses the accuracy of the tdpoll for the medium and high traffic cases as tdpoll starts choking under the increased number of collisions.

Figure 8 shows the average counting error offset among the three protocols in an effort to quantify on how much the protocols are off-the-mark in case of counting inconsistencies. We find that the result of these experiments mirror that of the counting inconsistency experiments closely.

6 Concluding Remarks

In this paper, we presented two lightweight and efficient singlehop primitives for collaborative feedback collection, pollcast and countcast. Using pollcast a node can query its singlehop neighborhood about a predicate P and in $O(1)$ time learns whether there are neighbors for which P holds. Using countcast a node can estimate the number C_P of neighbors for which P holds in $O(\log C_P)$ rounds. Our proposed RCD technique for the implementation of pollcast and countcast operations is easily achievable at the MAC layer in software and does not require any modification to the physical layer or the wireless radio hard-

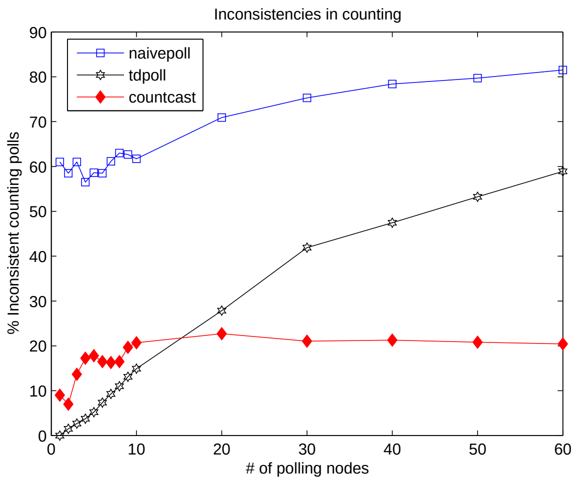

Figure 7: Counting inconsistencies

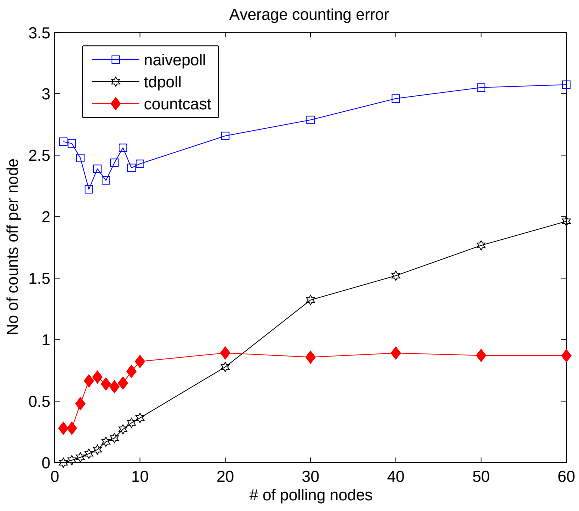

Figure 8: Average estimation error

ware. As such, the pollcast and countcast operations are readily applicable in WSNs and in a more general context for building robust mobile ad hoc network applications over 802.11-enabled networks. Some applications of pollcast and countcast in WSNs are in efficient implementations of false-positive filtering, in-network intruder classification, clustering, leader election, and barrier synchronization algorithms. Applications for ad hoc 802.11 networks are in facilitation group membership, collaboration, and coordination of swarms of robots or vehicles. Notably, a recent study shows "intelligent" cars fitted with sensors to communicate and estimate traffic flows can deliver the same fuel efficiency as hybrid vehicles [16].

Although eliminating false-positive collision detections completely may be challenging due to interference from neighboring regions in the medium and high traffic cases, the application using pollcast operations can designed so that false-positives from the RCD feedback do not affect safety properties, but only progress (e.g., termination) properties. This can be achieved by selecting P such that absence of P would be a precondition to continue with the rest of the action. For example, for the clustering scenario, P can be chosen to be "existence of a clusterhead" so that in the absence of P (i.e., when silence is detected in the vote phase), the initiator goes ahead with its clusterhead announcement. This way, false-positives heard at the initiator during the vote phase will only delay election of a clusterhead. We will investigate such defensive and effective programming practices in our framework. Another interesting topic for future research is to determine which lowerbounds would apply for common tasks (such as majority detection, at-least-k detection, etc.) developed in our framework. Since the poller can use the results from a previous poll to adaptively select a subset of its neighbors for its next poll, it is possible devise several efficient strategies for these tasks.

Finally, as part future work, we also plan to focus on the consistency issues of singlehop collaboration and coordination in the presence of collisions and hidden terminal problems. An application of such a consistent coordination primitive would be reliable broadcasting. In contrast to our previous work which achieved reliable broadcasting using synchronized rounds [10, 11], an extension of pollcast that addresses consistency issues would not require synchronized rounds across the network.

References

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 2002.

[2] A. Arora and et. al. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks (Elsevier)*, 46(5):605–634, 2004.

[3] A. Arora and et. al. Exscal: Elements of an extreme scale wireless sensor network. *Int. Conf. on Embedded and Real-Time Computing Systems and Applications*, 2005.

[4] H. Cao, E. Ertin, V. Kulathumani, M. Sridharan, and A. Arora. Differential games in large scale sensor actuator networks. In *IPSN*, pages 77–84, 2006.

[5] P. Costa, L. Mottola, A. Murphy, and G. Picco. Teenylime: transiently shared tuple space middleware for wireless sensor networks. In *MidSens*, pages 43–48, 2006.

[6] D. Culler, P. Dutta, C. T. Ee, R. Fonseca, J. Hui, P. Levis, J. Polastre, S. Shenker, I. Stoica, G. Tolle, and J. Zhao. Towards a sensor network architecture: Lowering the waistline. *HotOS X*, 2005.

[7] M. Demirbas, A. Arora, and M. Gouda. Pursuer-evader tracking in sensor networks. *Sensor Network Operations, IEEE Press*, 2006.

[8] M. Demirbas, A. Arora, V. Mittal, and V. Kulathumani. A fault-local self -stabilizing clustering service for wireless ad hoc networks. *IEEE Trans. on Parallel and Distributed Systems*, 17(9):912–923, 2006.

[9] M. Demirbas, A. Arora, T. Nolte, and N. Lynch. A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. *OPODIS*, pages 299–315, 2004.

[10] M. Demirbas and S. Balachandran. Robcast: A singlehop reliable broadcast protocol for wsns. *ADSN*, pages 1–8, 2007.

[11] M. Demirbas and M. Hussain. A mac layer protocol for priority-based reliable broadcast in wireless ad hoc networks. *BroadNets*, 2006.

[12] D. Estrin, R. Govindan, J. S. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. In *Mobile Computing and Networking*, pages 263–270, 1999.

[13] O. Gnawali, K-Y Jang, J. Paek, M. Vieira, R. Govindan, B. Greenstein, A. Joki, D. Estrin, and E. Kohler. The tenet architecture for tiered sensor networks. In *SenSys*, pages 153–166, 2006.

[14] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Trans. Netw.*, 11(1):2–16, 2003.

[15] V. Kulathumani, M. Demirbas, A. Arora, and M. Sridharan. Trail: A distance sensitive wireless sensor network service for distributed object tracking. *EWSN*, pages 83–100, 2007.

[16] C. Manzie, H. Watsona, and S. Halgamugea. Fuel economy improvements for urban driving: Hybrid vs. intelligent vehicles. *Transportation Research Part C: Emerging Technologies*, 15(1):1–16, 2007.

[17] Moteiv. `http://www.moteiv.com/`.

[18] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *SenSys*, pages 95–107, 2004.

[19] G. Simon, P. Volgyesi, M. Maroti, and A. Ledeczi. Simulation-based optimization of communication protocols for large-scale wireless sensor networks. *IEEE Aerospace Conference*, pages 255–267, March 2003.

[20] D. Son, B. Krishnamachari, and J. Heidemann. Experimental study of concurrent transmission in wireless sensor networks. In *Conference on Embedded networked sensor systems*, pages 237–250, 2006.

[21] M.-T. Sun, L. Huang, A. Arora, and T.-H. Lai. Reliable MAC layer multicast in IEEE 802.11 wireless networks. *Proc. ICPP*, pages 527–537, 2002.

[22] G. Tolle, J. Polastre, R. Szewczyk, N. Turner, K. Tu, P. Buonadonna, S. Burgess, D. Gay, W. Hong, T. Dawson, and D. Culler. A macroscope in the redwoods. In *SenSys*, 2005.

[23] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI*, pages 29–42, 2004.

[24] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSys*, pages 99–110, 2004.

[25] K. Whitehouse, A. Woo, F. Jiang, J. Polastre, and D. Culler. Exploiting the capture effect for collision detection and recovery. *EmNetS*, May 2005.