Design and Application of a Many-to-One Communication Protocol

Sudipta Saha
Indian Institute of Technology Bhubaneswar
saha_sudipta@ymail.com

Mun Choon Chan
National University of Singapore
chanmc@comp.nus.edu.sg

***Abstract*—In this paper, we address the fundamental problem of improving the performance of many-to-one and many-to-many communications. Our approach is Time Division Multiple Access (TDMA) based but addresses the limitations of existing TDMA implementations in a novel way. In a nutshell, we combine packets from many senders into a single large packet transmission by exploiting capture effect achieved through fine grained power control at the level of segments within a single packet. We applied our technique to the design of a one-hop, many-to-one communication protocol, *SyncMerge*, and a multi-hop, many-to-many communication protocol, *ByteCast*. Our evaluation shows that SyncMerge is able to achieve 2 to 15 times improvement over traditional many-to-one communication schemes. In addition, ByteCast is able to disseminate 1 byte of data from every node to all other nodes in about 600 ms with 99.5% reliability on a 90 node testbed. Compared to the state-of-the-art protocols such as LWB and Chaos, ByteCast is able to reduce the radio-on time by up to 90% while achieving similar reliability.**

I. Introduction

In many applications, many-to-one communication, whereby multiple nodes simultaneously try to communicate to a single node, is needed. For example, to do a quick neighbor discovery, a node broadcasts a probe message asking all its neighbors to respond. Similarly, in *receiver initiated MAC protocols* [18], [6], a receiver node broadcasts a probe message to inform potential senders about its willingness to accept data. In other applications such as data collection [11], convergecast [19], [12], data aggregation/sharing [14], such simultaneous many-to-one communication is a common requirement.

The challenge in supporting efficient many-to-one communication is in the need to coordinate transmissions from many nearby nodes with the minimum overhead. Even with carrier sensing (CSMA), these simultaneous transmissions collide at the initiator node. Subsequently, because of multiple back-offs and re-transmissions of packets, either it takes a very long time for all the source nodes to succeed in communicating their data to the initiator, or they may ultimately decide to give up and try later, if the maximum number of re-transmissions allowed is already reached. This naturally wastes a lot of time and energy in both initiator as well as source. The situation becomes worse as the number of simultaneous transmissions increases. An alternative to CSMA is to have the initiator node polling the source nodes one at a time using unicast communication to avoid collision. However, even such polling can take substantial time to complete as the initiator has to send probe packet one at a time to all the source nodes individually. Both approaches are thus inefficient in their own ways.

In this work, we introduce a novel solution for simultaneous many-to-one communication where multiple source nodes, *instead of contending with each other for the channel, cooperatively transmit their data only in their respective time-slots within the transmission of a single packet*. We achieve this by using a fine grained byte level power control technique. The proposed approach, which we term *SyncMerge*, thus, naturally avoids all the channel contentions among the source nodes. The overheads of transmitting a single packet, e.g., the preamble, header, trailer etc. are also minimized as they are shared by transmissions from multiple sources.

Based on an efficiently designed one-hop many-to-one communication protocol, more complex communication protocols can be designed. In particular, we present the design of a many-to-many communication protocol, *ByteCast*, for distribution of small amount of data in wireless sensor networks. We achieve this by appropriately arranging units of SyncMerge operations in a cascading fashion starting from a single initiator node in a network.

Although the basic concepts behind SyncMerge and ByteCast are general, in this work we implement them for low power wireless sensor network.

In summary, the contributions of this work are as follow -

1) We present the design of SyncMerge, a highly efficient scheme to support one-hop, many-to-one communication. The key innovation behind SyncMerge is the use of byte-level power control so that it is possible for multiple source nodes to transmit different data in a single packet transmission using fine-grained intra-packet TDMA.
2) We fully implemented SyncMerge in TelosB motes using *Contiki* operating system. TesloB and Contiki are the popular device and operating system, respectively, for low power wireless sensor networks. Our evaluation shows that SyncMerge achieves an improvement of at least 2 to 15 times in terms of radio-on time over traditional many-to-one communication schemes based on polling or channel sensing.
3) We present the design of ByteCast, a multi-hop, many-to-many communication protocol. ByteCast theoretically reduces the number of steps required to do all-to-all data sharing to the order of the diameter of the network.

4) We fully implement ByteCast on TelosB motes using Contiki. We tested ByteCast on Indriya [2] (using 90 nodes), and show that ByteCast can disseminate more than 99% of the data within 600 millisecond. In comparison to the state-of-the-art protocols such as LWB and Chaos, ByteCast can reduce the radio-on time up to 50% depending on the data size and number of source nodes.

The paper is organized as follows. We first present related work in section II. In sections III and IV, we present the design and implementation of SyncMerge and ByteCast, respectively. The evaluations are presented in section V. In section VI we discuss implementation limitations and potential improvement of the protocols. Finally, we conclude in section VII.

II. Related Work and Background

Many-to-one communication is a core communication pattern in various wireless communication protocols and applications. In data collection protocols [11], [12], many nodes transmit information to a single node. Similarly in operations such as gathering topological information from a single node [20], collecting the information of the surroundings by a mobile node in a mobile network, fast collection of data through mobile sink [15] bulk collection of data and many others, many-to-one communication forms a key pattern.

Existing approaches for efficient many-to-one communication are either based on collision avoidance [10], [16] or collision resolution [13], [17]. Our approach employs a special multi-packet reception [13] based strategy. We also combine it with intra-packet TDMA to avoid collision. TDMA has been already used in many works but the smallest granularity is at the level of a single packet per single node. In this paper for the first time we show that TDMA can be used even at the level of bytes inside a single packet. Thus, the proposed strategy in this work is a hybrid one based on both collision resolution as well as collision avoidance.

Researchers have attempted to exploit the capabilities available in the low level radio hardware. For example, the work [9] uses the unmodulated carrier wave and the RSSI sampling mechanism in a radio to alarm the whole network. Variation of transmission power has also been explored in many research at either inter-packet level [12], [3] to increase spatial reuse or on long unmodulated carrier wave to mimic different interference patterns [1]. However, in our work for the first time we use per-byte power control within a IEEE 802.15.4 physical layer protocol data unit which provides a way to allow simultaneous transmissions of different data from different senders through intra-packet TDMA.

Ferrari et al. presented Glossy [8] - a protocol that supports efficient flooding of small packets and time synchronization. A key idea exploited in Glossy is the technique of synchronous transmission whereby with sufficiently well synchronized timing, it is possible to transmit the same data from multiple different nodes at the same time and the data can be correctly decoded in the receiving radio. One of the main contributions is the technique to achieve one-hop time synchronization at the sub-microsecond level. Various protocols have been proposed based on the Glossy-based flooding (one-to-many). Low Power Wireless Bus (LWB) [7] repeatedly applies the basic one-to-many flooding mechanism to perform other communication patterns such as many-to-one and many-to-many. Splash [3] combines synchronous transmission based flooding with pipeline transmissions over multiple channels to perform data dissemination. Finally, Chaos [14] exploits capture effect along with Glossy to do many-to-many communication with data aggregation.

While our proposed protocols exploit synchronous transmission based flooding, we also utilize fine-grained byte-level power control so that TDMA based mechanism can be incorporated into the protocols such that different data can be transmitted by different nodes within the time scale of a single data packet.

III. SyncMerge (Many-to-One)

In this section, we first present the design of the one-hop many-to-one protocol called SyncMerge.

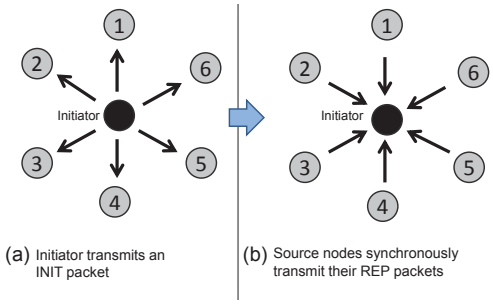

Fig. 1. Many-to-one communication.

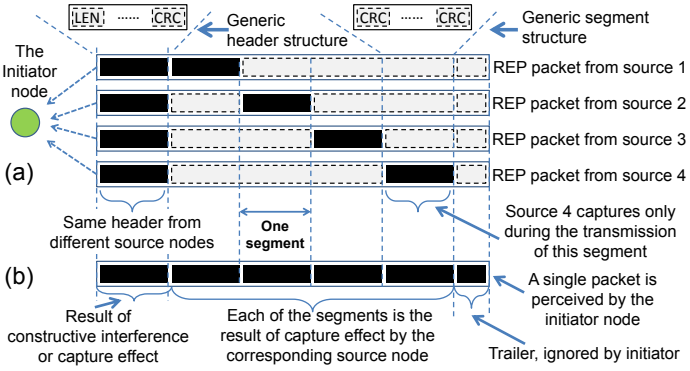

Fig. 2. Formation of the REP packet in SyncMerge.

A. Design

In SyncMerge, there is one initiator node who initiates the process and multiple source nodes, who want to send their data to the initiator node. Conceptually, the steps of a single unit of many-to-one communication are as follows.

1) The initiator broadcasts a packet expressing its intention to receive data. For clarity in the description, we call this the *INIT* packet.
2) On reception of the INIT packet, the source nodes reply to the initiator node with their respective packets. We call these collectively as the *REP* packets.
3) Depending on the status of the reception of the REP packets, the initiator sends acknowledgement (ACK) to the source nodes.

The primary challenge in supporting this interaction efficiently is in the second step, whereby there are simultaneous transmissions from multiple source nodes. We rethink this basic problem and design a synchronous transmission based solution for efficient (one-hop) many-to-one communication, which we term as SyncMerge. This basic many-to-one interaction (INIT and REP) is shown in figure 1. The first step (sending INIT packet) and the third step (sending ACK packet) are the same as described previously. The key difference is in step 2. On completion of the reception of the INIT packet, the transmissions from the various source nodes are coordinated by a combination of TDMA and fine-grained power control.

Our technique works as follows. The transmission period of a single data packet is divided into fixed length slots comprising of transmission time of one or more bytes. We refer to the bytes transmitted in a single slot as a *segment* in the REP packet. We assume that transmission power can be varied to transmit either at the highest transmission power or the lowest (zero if possible) transmission power. Each source node makes the decision to transmit either at the highest or lowest transmission power at each of these fine-grained intervals.

Regarding the transmission of a particular segment of a packet, the design of the protocol ensures that there are only two possible outcomes:

1) *The content transmitted by all the sources are the same.* By ensuring that the transmissions are time synchronized based on a Glossy-like mechanism [8], the initiator node will be able to receive the data with high reliability.
2) *The contents transmitted by the sources are different.* Since the contents are different, synchronous transmissions will not provide reliable reception as in the previous case. Instead, in each segment's transmission, only one source node will transmit at the highest transmission power, while all other source nodes transmit at the lowest transmission power. As a result, due to capture effect, transmission from only one source node will be received by the initiator during the respective time slots.

Figure 2 shows a graphical illustration of the formation of the REP packet. The reception of the INIT packet is used as a synchronization time point among the contending source nodes. The INIT packet from the initiator also contains a scheduling information for the source nodes. On completion of the reception of the INIT packet all the source nodes start to synchronously transmit the REP packet. In the REP packet, the header is common for all source nodes. Because of synchronous transmission and capture effect, all source nodes' transmissions can be perceived as a "single packet" transmission by the initiator. Thus, the initial processing such as the calibration time of the radio, transmission of preamble for time synchronization, transmission of Start Field Delimiter (SFD) to indicate the start of the packet, transmission of the length field as well as even the two bytes of trailer information etc., all can be done in parallel in all the source nodes. The initiator node also needs to receive and process these common parts only once. This reduces significantly the required overhead. The rest of the packet is divided in equal sized data segments. A source node uses the highest transmission power possible only while transmitting its own data-segment, i.e., those bytes that fall in its respective transmission time slot as it understood from the schedule encoded in the INIT packet. For all other slots, it transmits at the lowest power.

B. Implementation

We implement SyncMerge on Contiki [5] and TelosB motes which have CC2420 chip as the 802.15.4 radio transceiver and MSP430 microcontroller. However, to use synchronous transmission, we write our own miniature radio driver similar to the one used in Glossy [8].

Feasibility and portability: Programmable radio transceivers offer mechanisms for controllable variation of transmission power by writing to some internal register. In CC2420 radio, the available maximum and the minimum power level options are 0 dBm and -55 dBm, respectively. At the minimum power level, the signal is identifiable only if two motes are placed side by side with a distance less than around 6 cm [1]. To change the transmission power dynamically, we change the content of the power control register in the radio on-the-fly during the transmission of the packet. In principle intra-packet power control can be supported by any hardware setup if the microcontroller can write the radio power control register multiple times within the transmission time of a packet. In our case with 250 Kbps over-the-air data rate, time to transmit a byte takes around 32 microseconds. On the other hand, MSP430 microcontroller of TelosB set at 3.9 GHz, takes around 22 microsecond to send the command to CC2420 radio over SPI to write the radio power control register. Since byte transmission time is higher than transmission power setting time, in principle it is possible to change transmission power level on a per-byte granularity in this setup.

Packet structure: The first segment of every packet is the header. The first field of header is the length field. The header is considered to be an independent component and needs its own CRC to indicate its validity. Figure 2 shows these two essential parts of the header. The rest of the fields in the header depend on the specific requirements. In our implementation, the packets contain two more fields - (i) one-byte packet type field, since there are three types of packets involved, e.g., INIT, REP and ACK and (ii) protocol specific two-byte common data field which we reserve for any future use.

Structure of INIT Packet: INIT packet also contains a 5 byte header. The purpose of the INIT packet is to provide a

schedule to the source nodes. This schedule can be an explicit or implicit schedule. An explicit schedule contains a list of node identifiers. If a default/implicit schedule is to be used, the nodes need to have prior knowledge of the schedule. Given a short index, they can derive the set of the node identity numbers as well as the specific order of transmissions.

Structure of a segment in a REP packet: A segment, transmitted by a single source node, can be as small as 1 byte. But since we need to make the segments fully independent of each other, we need to add at least one byte of CRC with each segment. Further, to perform byte level power control, we need to set the radio power control register at the exact time. But, since byte transmission time and register writing time do not match exactly, we need to pad the extra intervals with NOP operations. In our case, at 3.9 GHz clock frequency of the microcontroller one byte time is 124.8 clock cycles. So, to pad a full byte we need to consume either 125 or 124 cycles. This mismatch creates a timing problem and hence to be safe we use one guard byte at each boundary of a segment. We use the simplest possible segment structure consisting of three parts - (a) one (leading) guard byte , (b) followed by data segment and (c) one (trailing) guard byte. Since, both the guard bytes are vulnerable to error, we store the calculated CRC in both of the guard bytes and rely on the correct reception of at least one of the guard bytes for validation. The number of data byte in a segment can be one or more depending on how many nodes we want to put in a single largest size packet. Figure 2 shows this generic segment structure. With these overhead, a 128 byte data packet can accommodate up to 40 segments from 40 different source nodes.

Structure of the ACK packet: The purpose of the ACK packet is to convey the status of the reception of the segments from each of the source nodes. We use bit values in few bytes to denote whether the corresponding segment was received properly.

We will present the evaluation of SyncMerge in section V-A and discuss issues with the limitation of power control on existing devices in section VI

IV. ByteCast: many-to-many communications

A. Design of ByteCast

In this section, we present the design of a multi-hop protocol to support many-to-many data communication, called *ByteCast*. Many-to-many communication can be considered as a combination of many-to-one communications. Hence, we design ByteCast on the basis of SyncMerge but with a couple of changes to improve efficiency. First, instead of the 3 stages (INIT/REP/ACK) used in SyncMerge, in ByteCast we use only two stages - SEND and RECV. As indicated by the names, in the SEND stage a node transmits while in the RECV stage a node receives. On completion of a SEND stage, RECV stage starts and similarly on completion of a RECV stage, SEND stage starts. Second, we use only one packet type, which can be considered as a combination of the three packets/stages of SyncMerge "merged" into a single packet.

Packet Structure: The layout of a packet in ByteCast is similar to the REP packet in SyncMerge. However, since there is no separate INIT packet, the transmission schedule (or the mapping of node ids to time slots/segments) needs to be conveyed in a very concise way. As a result, only default/implicit schedule can be used. This is achieved by putting a schedule index in the header of every packet. For example, if the index is 1, transmissions are for nodes with id 1 to 30, and if index is 2, transmissions are for nodes with id 31 to 60 and so on.

Initial Step: In the initial phase, each node only knows its own data. ByteCast begins when the initiator transmits the first packet with the chosen schedule index in the header.

Subsequent Steps (RECV/Receive): When a node receives a packet, it first checks the schedule index. If its node id is included in the implicit schedule it puts its own content in the next SEND stage. However, in all cases it records the data in all the valid segments present in the received packet.

Subsequent Steps (SEND/Transmit): In a SEND stage, each node transmits only those segments that it has received and accumulated so far from all earlier RECV stages under the current schedule. It also transmits its own segment if its id falls in the current schedule. For all other data segments it switches its radio transmission power to the lowest possible value.

For example, let us consider that the schedule index is 1 (i.e., nodes with id 1 to 30) and a node has received data segments from nodes 5 and 15 so far. Let us further assume that in the current RECV stage it gets data segments from nodes 1 and 10. Thus, in the next SEND stage, it will transmit data segments of nodes 1, 5, 10 and 15. Note that a node includes its own data segment in this SEND stage only if the id of the node falls in the current schedule.

One can observe that if at least one node received the data segments from a node X, X will find its data segments in the packet to be received in the next slot. Thus, a received packet also serves the purpose of an ACK for the nodes. Furthermore, data from multiple nodes are distributed within a single packet transmission. This process continues upto a predetermined number of transmissions. This duration depends on the size and diameter of the network.

Behavior of ByteCast: We analyze the time complexity of ByteCast through a simple model. Assume that the nodes are organized into logical layers through their SEND and RECV stages. Since we use a single packet size, the time to complete a SEND stage is equal to that of a RECV stage. This is in turn equal to the over-the-air transmission time of the packet through the radio. For the transmission started at the initiator, it takes (L-1) such time slots to reach a node L hops away from the initiator by propagating through the network. Thus, layer L nodes start their SEND stage at time slot L. On their first RECV stage they receive data segments from many nodes. But they cannot receive data from the nodes in the same layer. They need to wait their data to propagate to the initiator and get "reflect" back down to layer L. This takes another 2(L - 1) time slots. Thus, for the nodes at layer L to get data from

all the nodes at the same layer it takes 3(L - 1) time slots. Figure 3 illustrates this process for a node at layer 5.

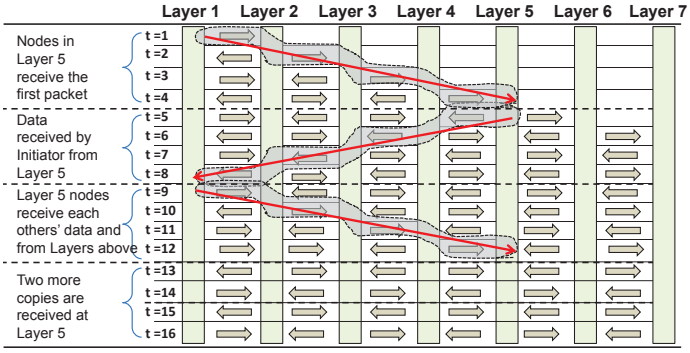

Fig. 3. Analysis of time requirement of ByteCast to complete upto layer 5.

B. Implementation Details

We use the largest possible packet size (128 bytes) available in the CC2420 radio. Implicit global schedules are used where every node reserves a segment. Segment size is globally fixed to the minimum required value of 3 bytes (see section III). Packet header size is fixed to the minimum possible size, i.e., 3 bytes - one byte length, one byte schedule index, and a CRC byte. We do not use the 33 bytes at the end of a packet so that there is time left for processing and other calculations. So, 128 - 3 - 33 - 2 (for trailer) = 90 bytes are available to be used for the data segments. Thus, we can fit $\frac{90}{3}$ = 30 different segments in a single packet.

A single schedule will not be sufficient when the network is large. For larger network, we use multiple schedules and identify a schedule by an id. Nodes put their data in the packet in the right segment if and only if its id belongs to the current implicit schedule. To cover all the nodes in a system with N nodes, the number of schedules (i.e., independent runs) required in ByteCast is $\lceil \frac{N}{M} \rceil$, where M is the maximum number of nodes that can be accommodated in a single data packet.

Note that in an iteration of ByteCast with a specific schedule index, even those nodes which do not fall in the schedule participate in the process. If a node is not part of the schedule, it does not contribute data but needs to transmit data it has received from other nodes. This is required to make sure that the process does not stop due to insufficient connectivity.

V. Evaluation

In the evaluation, we first present the results for SyncMerge in section V-A followed by ByteCast in section V-B.

Metrics: We measure the total radio-on time and reliability. Reliability is computed as the ratio of data received by nodes to the total amount of data to be disseminated to all nodes.

A. SyncMerge

We compare the performance of SyncMerge to existing asynchronous communication based techniques. We performed a controlled experiment using TelosB motes and Contiki operating system. A total of 11 TelosB motes were arranged on a table - 10 acting as the source nodes and 1 acting as the initiator. Note that saving energy is an inevitable issue in the operating systems for the motes used in low power wireless sensor networks. Radio transceiver is typically the highest power consuming module in a mote. In synchronous transmission based technique, although the radio is kept always on, the whole protocol is arranged in such a way that it is completed in a very compact fashion with a fixed and small time requirement for all the nodes in the system. On the other hand, in asynchronous communication based technique, the energy saving is achieved through low duty cycling of the radio module. Contiki implements duty cycling through its RDC (radio-duty-cycling) layer in the network stack by periodically checking the channel at a certain rate for any possible transmission. RDC layer also efficiently handles possible communication problems due to the duty cycling of the radio.

For our experiments, we use the default Contiki setting where the ContikiMAC [4] and CSMA are used as the RDC and MAC layer protocols respectively. Note that although in asynchronous communication based protocols use of radio duty cycling saves energy, it reduces the chance of successful communications between motes. In order to minimize the effect of duty cycling as much as possible, we set a high (128 times per second) channel checking rate in Contiki.

We compare SyncMerge to two baseline approaches. The first approach is based on polling where the initiator polls every source and collects data one-by-one. In the second approach, the initiator sends a single broadcast request and the source nodes start sending their data at the same time. Coordination among different source nodes were performed through CSMA/CA. In case of transmission error, up to three retransmission attempts were performed. The payload size was set to one byte in the baseline schemes while SyncMerge used a three-byte segment for each source node with a common five-byte header.

We measure the total radio-on time in all the experiments. For the initiator this is the total radio-on time until it receives data from all the source nodes and completes the transmission of the ACK. For a source node, this is the time between the source node hearing the polling/unicast message from initiator to the end of successfully communicating its data to the initiator.

Figure 4 shows the comparison of radio-on time. Each data point is the average over 2000 runs. From the results, it can be seen that SyncMerge always takes a fixed amount of radio-on time in the nodes. This is around 6.2 millisecond. The reliability was about 98.4%. On the other hand, the radio-on time of the other protocols are higher. The reduction in radio-on time in SyncMerge is at least 2 to 15 times in the source and the initiator, respectively. It can be also noticed from the results that in the initiator the polling based method takes larger time in comparison to the combined request based method. This is because in polling the initiator has to poll

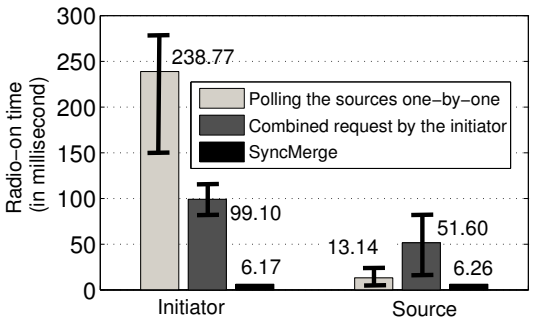

Fig. 4. Comparison of radio-on time.

each of the 10 motes individually using unicast facility in Contiki, while in combined request one broadcast message is sent and all the source motes try to respond together. Note that, since all the motes do duty cycling of their radio, there is a chance that a source mote misses the initial broadcast from the initiator. To make a fair comparison we repeated the experiment many times and considered only those cases where the initial broadcast from the initiator was received by all the sources. 2000 such cases were extracted for preparing the result. The reliability was about 98.6%.

It is to be noted that the current implementation supports up to 40 simultaneous source nodes. Thus, the reduction in radio-on time, relative to the traditional techniques, would be even larger if the number of source nodes increases beyond 10.

B. ByteCast

We implemented ByteCast in Contiki operating system for TelosB motes. We tested it in the Indriya testbed which at present has 90 active TelosB motes spanned over three floors. In this section we first report detailed performance evaluation of ByteCast in Indriya. Then we report the comparison of ByteCast with two state-of-the-art protocols LWB and Chaos in the same settings. In all evaluations, ByteCast attempts to distribute n byte(s) from each of the 90 nodes to the other 89 nodes. The default value of n is 1, unless stated otherwise.

In this evaluation, we first used ByteCast to count the total number of nodes in the testbed. Indriya initially supported up to 140 nodes and the node ids range from 1 to 140. However, the testbed currently has only 90 active nodes. We assume that the identifiers of the active nodes are not known in advance but we know the range of node ids (1 to 140). As up to 30 nodes can be the sources in a single iteration of ByteCast, we need 5 iterations to cover all 140 identifiers. During the execution of ByteCast, each node inserts its data (identifier value) into the segment allocated by the transmission schedule. At the end of the experiment (after 5 iterations), each active node knows the identifiers of all the other active nodes.

Figure 5 shows the average along with the 90^{th} and 10^{th} percentile results over 1000 runs of the execution of ByteCast on Indriya. Since ByteCast does not need any specialized bootstrapping phase, for each execution we picked up a new initiator from some random sequence. The result shows that in less than 500 ms the protocol achieves around 97.5% reliability. However to achieve 99.5% reliability, it needs to spend another 500 ms. Overall, we observe variations in reliability in the result. We will provide more explanation on this variability in Section VI.

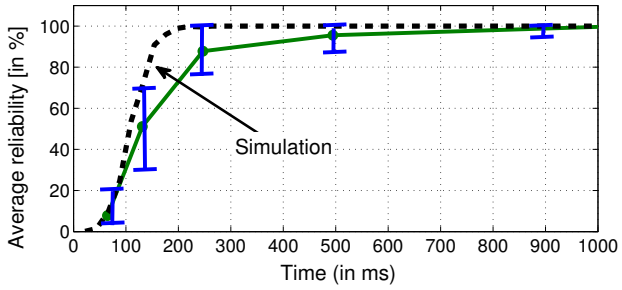

Fig. 5. Percentage of nodes receiving 1 byte data from all the other nodes in Indriya through ByteCast, averaged over 1000 runs and all the nodes.

Finally, note that the transmission schedule used in this experiment is inefficient since not all segments are used. If the transmission schedule is compact, only three iterations will be needed and there will be a 40% reduction in time (5 iterations to 3 iterations). Hence, the time needed for 99.5% reliability is 600 ms instead.

1) Comparison with LWB: Low Power Wireless Bus (LWB) is a Glossy based protocol designed to emulate operations of a "bus" protocol. In LWB, a node uses Glossy based flooding mechanism to transmit its data to all the nodes in the network. In each "flood", only one piece of data from a single source can be disseminated to all the network. Furthermore, in order to improve reliability and for allowing retransmissions, each time slot in LWB needs to be sufficiently long. The factors affecting the slot length are the maximum hop in the network, number of retransmissions and the length of the packet. For example, it has been found that on Indriya, considering a small 4 byte packet, slot length should be at least 15 ms. Note that for higher data size, the slot length needs to increase. For example, for a 32 byte data, the slot length should be at least 40 ms. In our evaluation, we used the default slot time in the Glossy code distribution, which is 20 ms.

For flooding by a single source, ByteCast is similar to Glossy. The difference comes when there are multiple sources willing to disseminate different data at the same time. In our approach, with the use of byte-based dynamic power control, one can think of ByteCast as execution of multiple versions of Glossy flood within a single time slot. This merging brings a lot of savings by removing the requirement of defining boundaries between two consecutive floods. On the contrary, in LWB, each independent Glossy flood has to be separated safely so that they do not overlap. The packet size used by ByteCast depends on the number of source nodes that are

distributing data to all other nodes. For example, if there are 10 nodes, each distributing 1 byte to all other nodes, the packet size used is 35 bytes. In this 35 byte packet, there is a 3 byte header, $10 \times 3 = 30$ bytes of data segments from 10 nodes, and 2 bytes of trailer. Recall that there is a 2 byte overhead per node in the segment data portion, with one byte needed for CRC and one byte needed to handle clock drift.

We compare the radio-on time required by LWB and ByteCast to distribute data sizes of 1 to 8 bytes from many nodes to many nodes in Indriya. For simplicity, we do not include the overhead for channel allocation and setup time of LWB. For an experiment comprising of N nodes as source nodes, we randomly pick N nodes in Indriya and chose a random initiator in each run. Figure 6 shows the result. We plot the radio-on time required by LWB using a slot time of 20 ms.

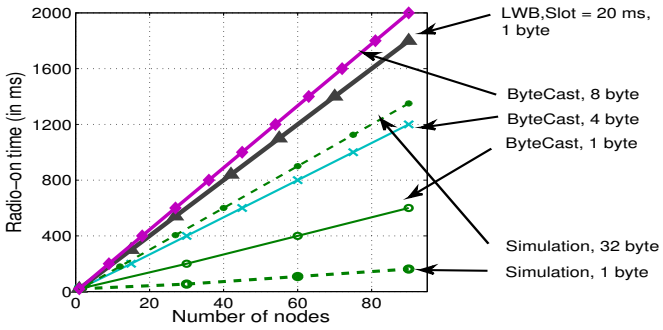

Fig. 6. Comparison of radio-on time in ByteCast and LWB for different data sizes and different number of nodes acting as source nodes.

As expected, the radio-on time for both protocols increases linearly with the number of source nodes. By running multiple distributions within a single packet transmission, ByteCast outperforms LWB for small data sizes from 1 to 7 bytes per node in Indriya for different number of nodes. However, LWB performs better than ByteCast for distribution of 8 bytes or more per node if a 20 ms time slot can be used. This is unexpected since ByteCast performs multiple distributions within a single packet transmission. But this happens due to certain issues regarding the implementation of the proposed schemes. These are discussed in details in the next section.

Finally, one big advantage of ByteCast over LWB is that ByteCast does not need any bootstraping phase. In contrast to that, LWB needs a huge time to prepare and update the schedule. ByteCast does this work implicitly in a transparent fashion within the main protocol itself.

2) Comparison with Chaos: Chaos is a protocol designed mainly for in-network aggregation of data. It is the first work which uses capture effect to perform an efficient many-to-many interaction among the nodes in a network. There are some similarities between Chaos and ByteCast, e.g., both are based on synchronous transmission and both exploit capture effect. However, there are also importance differences. In this section, we will first highlight the differences and then evaluate our approach in comparison to Chaos.

Although Chaos is designed for efficient data-aggregation, it can be modified to perform many-to-many data-sharing. For Chaos to do many-to-many data sharing, the payload has to be divided into N chunks where each chunk is assigned to one node. The size of a chunk corresponds to the size of data to be disseminated by each node. For the purpose of data sharing, the merge operation in Chaos becomes a union operation. Thus, both the packet structure and the merge operation are similar to ByteCast. The fundamental differences between Chaos and ByteCast are as follows. First, Chaos uses packet level capture effect while ByteCast exploits capture effect at the level of segments of a packet. Second, Chaos depends on capture effect in an unplanned and unorganized fashion. The actual behavior thus, depends on both the topology/shape of the network as well as the set of nodes transmitting simultaneously in every step. On the other hand, in ByteCast, nodes transmit or receive in discrete and scheduled time slots. By following a common schedule, ByteCast exploits cooperation among the nodes instead of competing to capture the channel as is done in Chaos. Therefore, in ByteCast, once started from the initiator, the wave reaches very quickly to all the nodes and hence each node gets the chance to put their data in proper segment. Due to this organized behavior one can easily understand the time complexity of ByteCast. In contrast, in practice it is hard to analyze the timing behavior of Chaos.

We illustrate this difference in Figure 7, where it is shown that in case of a small network having 4 nodes, ByteCast takes only 3 slot time to do all-to-all data sharing, whereas Chaos in the same setting will take 6 slot time to complete. The difference will be even higher for higher number of nodes.

In ByteCast, to disseminate 1 byte of data, we need to transmit 3 times more information from each node. This is a large overhead compared to Chaos. In the 802.15.4 radio, using a 128 byte packet, we can only fit information from 30 nodes into a single packet while Chaos can support up to 108 nodes after reserving space for the headers. However, this overhead becomes smaller when we need to transmit more data from each node. In particular, once the total data size (number of sources multiplied by data size per node) along with the required header size exceed the maximum packet length, Chaos will need to perform multiple rounds of dissemination. It may also consider less number of nodes in a single iteration with higher data size per node. In any case, the total number of iterations will increase. On the other hand, increase in the total data size will only increase ByteCast's dissemination time gradually, by using the right combination of packet sizes and more schedules.

The evaluation for ByteCast and Chaos is shown in figure 8. Note that in the Chaos paper [14], the authors report that in Indriya with 139 nodes, it took around 432 ms for Chaos to converge data aggregation with a 100 byte payload. We find that the Chaos program, with necessary modification for data sharing on Indriya, takes almost twice time to converge for full payload size. This can be explained by the fact that the number of nodes available on Indriya has gone down from 139 to 90, significantly reducing the average node density and shape of

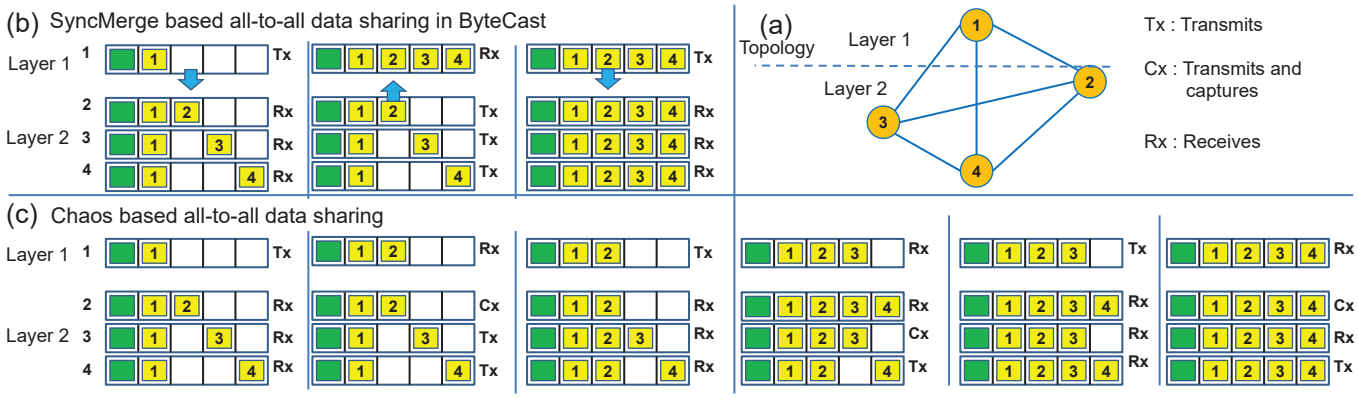

Fig. 7. Comparison of the time requirement between Chaos and SyncMerge based ByteCast to attain full coverage in all-to-all communication in a 4-node network. The topology is provided in part (a). Part (b) shows the step-by-step execution of ByteCast and the data segments acquired in each of the four nodes, whereas part (c) shows the same when Chaos is appropriately modified to do the same task. Each block shows the status of the nodes after the transmission and subsequent reception of a packet.

the network. Second, there may also have been substantial changes in the environment including possible interference in the channel chosen (channel 26). The evaluation results presented in figure 8 is based on running both the protocols on the same testbed one after another to minimize changes in the environment. The average reliability values of Chaos and ByteCast are 99.21% and 99.51% respectively.

The results clearly show that ByteCast has much smaller dissemination time than Chaos. The amount of improvement increases with larger data sizes (per node). For disseminating data of size 1 byte to 12 bytes among 90 nodes, the reduction in the radio-on time varies from about 30% to 70%. When the number of sources is varied from 10 to 90, the reduction in the radio-on time in disseminating 1 byte of data to all of the 90 nodes varies from about 30% to 50%.

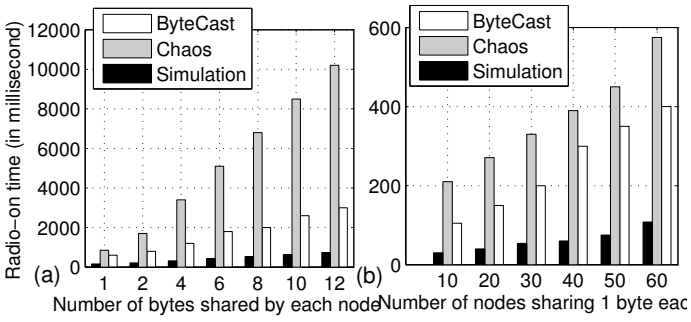

Fig. 8. Comparison of the radio-on time in SyncMerge based ByteCast and Chaos to achieve almost similar (above 99%) coverage for different data sizes and different number of nodes in Indriya.

VI. Discussion and Potential Improvement

The ability to perform fine-grained dynamic power control is a prerequisite for both SyncMerge and ByteCast. When one source node transmits its segment, other source nodes should preferably not interfere at all.

However there are two major issues regarding the implementation of the proposed scheme. The first is the requirement of time synchronization at the level of packet segments among the nodes during the transmission of a packet. This is very difficult to achieve without any hardware support. We take a software based approach for this purpose using the MSP430 internal DCO clock. But it fails several times due to the well-known problems of DCO, and as a result the segments transmitted from different nodes partially overlap with each other.

The other less severe issue is that in the CC2420 radio, the minimum power level option available is not negligible (highest and lowest power are at 0 dBm and -55 dBm, respectively). One possible solution is to turn off the radio immediately after the transmission of a segment. However, in the current radio hardware, turning the unit off involves stopping the crystal oscillator. Once the crystal oscillator is turned off, it takes some amount of time to turn it on and this duration may also vary in an indeterminate fashion. Hence, turning the radio on and off repetitively on short time scale results in loss of synchronization.

The lack of perfect packet-segment level time synchronization as well as the inability to reduce transmission power to a sufficiently low level cause undesirable impact on the protocol efficiency. In particular, we found that the overall connectivity among the nodes reduces. To understand how this happens, we perform two sets of experiments on Indriya. In the first experiment, we measure the PRR of all the links in an usual setting where every node is assigned a slot to transmit a small packet while the others are supposed to listen. At the end of the experiment, every node calculates the PRR of the links to their neighbors based on the logs of successful packet receptions and the number of allocated time slots. In the second experiment, we run the basic SyncMerge protocol from each node sequentially for several rounds. The schedules were prepared in such a way that every node gets a chance to convey one segment of data to the initiator. At the end

of the experiment, every node calculates how many times it could successfully receive the segment from its neighbors. This quantity has a quite similar meaning as PRR. To differentiate the metric, we call it *sPRR*. In both the experiments a packet was transmitted every 20 ms and each experiment lasted for around 5 hours. We collected measurements for both PRR and sPRR in Indriya with 90 active nodes.

To quantify the change in connectivity, we define the notion of the degree of a node as the number of links associated with it whose PRR and sPRR is higher than 0.9, for the first and the second experiment, respectively. The average degree based on PRR was found to be 9.1, while the same based on sPRR was found to be 3.2. We believe that this large drop in connectivity explains the lower than expected performance of SyncMerge and ByteCast observed in Section V.

We try to gauge the performance of SyncMerge and ByteCast if perfect segment level time synchronization and true zero power transmission were possible. In order to do so, we try to "virtually" remove the effect of the undesirable low power transmissions and assume perfect segment level time synchronization by simulating these protocols on the network topology consisting of only the links with PRR ≥ 0.9. While these simulations may not be sufficiently accurate, it provides an estimate of the possible performance achievable.

Figure 5 also shows a plot of the simulation result in dashed line. The simulation result shows that ByteCast can achieve 100% reliability in Indriya within 270 ms. The simulation results comparing LWB and ByteCast are shown in Figure 6. It can be seen from the results that the simulated performance of ByteCast is much better than LWB even for data size as large as 32 bytes. This result is much closer to the expected performance achievable given the amount of parallelism that ByteCast allows. Finally, the simulation results comparing Chaos and ByteCast are shown in Figure 8. These results show that improvement of 84% to 90% over Chaos is possible.

VII. Conclusion

In this work, we have designed and implemented SyncMerge, a novel many-to-one communication protocol. SyncMerge can be used as a base component in the design of many other protocols. It opens up a new direction of research where one can think of how complex network operations can be redesigned by dividing the task into multiple units of SyncMerge in parallel and in appropriate sequence. In this paper we showed one such approach through ByteCast. Other applications such as fast all-to-all routing, routing tree construction in dynamic networks, convergecast, quick topology discovery, etc., can get hugely benefited by SyncMerge. On the other hand, the base protocol SyncMerge itself can be improved further. Application of some better segment level time synchronization technique during transmission should be the immediate step. However, to achieve this, special support from hardware might be necessary. One possible way can be to use some suitable external crystal oscillator. Moreover, having options in the radio to transmit in even lower power can also help to improve the performance of SyncMerge. We plan to take all these issues as part of our future work in this direction.

Acknowledgment

This research is partially supported by the Singapore Ministry of Education Tier 1 grant, T1 251RES1609.

References

[1] C. A. Boano, T. Voigt, C. Noda, K. Römer, and M. Zúñiga. Jamlab: Augmenting sensornet testbeds with realistic and controlled interference generation. In *Proceedings of IPSN*, pages 175–186. IEEE, 2011.

[2] M. Doddavenkatappa, M. C. Chan, and A. L. Ananda. Indriya: A low-cost, 3d wireless sensor network testbed. In *Proceedings of TRIDENTCOM*, 2011.

[3] M. Doddavenkatappa, M. C. Chan, and B. Leong. Splash: Fast data dissemination with constructive interference in wireless sensor networks. In *Proceedings of USENIX NSDI*, pages 269–282, 2013.

[4] A. Dunkels. The contikimac radio duty cycling protocol. 2011.

[5] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of Local Computer Networks*, pages 455–462, 2004.

[6] P. Dutta, S. Dawson-Haggerty, Y. Chen, C.-J. M. Liang, and A. Terzis. Design and evaluation of a versatile and efficient receiver-initiated link layer for low-power wireless. In *Proceedings of SenSys*, pages 1–14, New York, NY, USA, 2010. ACM.

[7] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele. Low-power wireless bus. In *Proceedings of SenSys*, pages 1–14, 2012.

[8] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh. Efficient network flooding and time synchronization with glossy. In *Proceedings of IPSN*, pages 73–84, April 2011.

[9] R. Flury and R. Wattenhofer. Slotted programming for sensor networks. In *Proceedings of IPSN*, pages 24–34. ACM, 2010.

[10] M. Garetto, T. Salonidis, and E. W. Knightly. Modeling per-flow throughput and capturing starvation in csma multi-hop wireless networks. *IEEE/ACM ToN*, 16(4):864–877, 2008.

[11] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection tree protocol. In *Proceedings of SenSys*, pages 1–14. ACM, 2009.

[12] O. Incel, A. Ghosh, B. Krishnamachari, and K. Chintalapudi. Fast data collection in tree-based wireless sensor networks. *IEEE Transactions on Mobile Computing*, 11(1):86–99, Jan 2012.

[13] L. Kong and X. Liu. mzig: Enabling multi-packet reception in zigbee. In *Proceedings of MobiCom*, pages 552–565. ACM, 2015.

[14] 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 SenSys*, pages 1:1–1:14. ACM, 2013.

[15] Z. Li, M. Li, J. Wang, and Z. Cao. Ubiquitous data collection for mobile users in wireless sensor networks. In *Proceedings of IEEE INFOCOM*, pages 2246–2254, 2011.

[16] F. Österlind, L. Mottola, T. Voigt, N. Tsiftes, and A. Dunkels. Strawman: resolving collisions in bursty low-power wireless networks. In *Proceedings of the IPSN*, pages 161–172. ACM, 2012.

[17] P. Patel and J. Holtzman. Analysis of a simple successive interference cancellation scheme in a ds/cdma system. *IEEE JSAC*, 12(5):796–807, 1994.

[18] Y. Sun, O. Gurewitz, and D. B. Johnson. Ri-mac: A receiver-initiated asynchronous duty cycle mac protocol for dynamic traffic loads in wireless sensor networks. In *Proceedings of SenSys*, pages 1–14. ACM, 2008.

[19] H. Zhang, A. Arora, Y. ri Choi, and M. G. Gouda. Reliable bursty convergecast in wireless sensor networks. *Computer Communications*, 30(13):2560 – 2576, 2007.

[20] M. Zhang, M. C. Chan, and A. L. Ananda. Connectivity monitoring in wireless sensor networks. *Pervasive and Mobile Computing*, 6(1):112–127, 2010.