Virtual Synchrony Guarantees for Cyber-Physical Systems

Federico Ferrari* Marco Zimmerling* Luca Mottola† Lothar Thiele*
*Computer Engineering and Networks Laboratory, ETH Zurich, Switzerland
†Politecnico di Milano, Italy and Swedish Institute of Computer Science (SICS)
{ferrari, zimmerling, thiele}@tik.ee.ethz.ch luca.mottola@polimi.it

***Abstract*—By integrating computational and physical elements through feedback loops, CPSs implement a wide range of safety-critical applications, from high-confidence medical systems to critical infrastructure control. Deployed systems must therefore provide highly dependable operation against unpredictable real-world dynamics. However, common CPS hardware—comprising battery-powered and severely resource-constrained devices interconnected via low-power wireless—greatly complicates attaining the required communication guarantees. VIRTUS fills this gap by providing *atomic multicast* and *view management* atop resource-constrained devices, which together provide *virtually synchronous* executions that developers can leverage to apply established concepts from the dependable distributed systems literature. We build VIRTUS upon an existing best-effort communication layer, and formally prove the functional correctness of our mechanisms. We further show, through extensive real-world experiments, that VIRTUS incurs a limited performance penalty compared with best-effort communication. To the best of our knowledge, VIRTUS is the first system to provide virtual synchrony guarantees atop resource-constrained CPS hardware.**

I. INTRODUCTION

Cyber-physical systems (CPSs) are engineered systems deployed in the physical world whose operation is controlled by a computing and communication core. By gathering data from the environment through sensors and by taking actions on it through actuators, CPSs can realize safety-critical control loops in scenarios where traditional systems are hardly applicable.

Motivation. As a concrete example, consider the TRITon project [1], which deals with the implementation of closed-loop control for adaptive lighting in road tunnels. TRITon uses battery-powered sensor nodes to report, via low-power wireless, periodic light readings to a central controller running on embedded hardware [1]. This closes the loop by setting the lamp intensity to match a legislated curve. The ability to dynamically match the lighting levels to the environmental conditions improves the tunnel safety.

Nevertheless, the safety-critical nature of many CPSs raises concerns [2]. Using current communication protocols, for example, TRITon designers cannot provide dependability assurances [1]. Moreover, the centralized controller represents a single-point of failure. Designers wish to address these concerns, for example, by replicating the control logic across devices, as the mainstream practice would recommend [3]. Similar issues are also found in diverse CPS applications, such as healthcare [4], logistics [5], and automation [6].

Unfortunately, applying established designs of dependable distributed systems to CPSs is often not possible, as these require communication guarantees that existing CPS network protocols do not provide. Such guarantees notably include, for example, well-defined message delivery orderings that facilitate the implementation of replicated functionality, as well as failure handling mechanisms operating w.r.t. both node crashes and message omissions [3], [7].

In fact, existing low-power wireless protocols typically operate in a best-effort manner, their design being optimized towards non-functional properties, such as energy consumption [8]. Nevertheless, low-power wireless networks are most often characterized by *dynamic multi-hop topologies*, created out of *unreliable channels* with *limited bandwidth*. Typical CPS devices also feature *poor processing capabilities* and *limited storage facilities*. Such characteristics make providing even simple communication guarantees impractical. For example, it may be extremely difficult to enforce message orderings whenever nodes: *i)* are unable to buffer several messages due to the memory shortage, and *ii)* need to rely on a time-varying set of intermediate devices to achieve global coordination.

Virtual synchrony. The virtual synchrony [9] model for distributed computation may be one of the designated technologies to underpin dependable CPSs. Birman et al. describe virtual synchrony as [10]:

> It will appear to any observer that all processes observed the *same events* in the *same order*. This applies not just to message delivery events, but also to failures, recoveries, and *group membership changes*.

Two key concepts thus concur to create virtually-synchronous executions. First, virtual synchrony entails a notion of *group*: a set of processes exchanging messages originated at one node in the group and addressed to all other group members. The group membership is reflected in a data structure called *view*, which reports information on the nodes in a group at a given point in time. As group members fail and new nodes possibly join, a virtually-synchronous system must accordingly reflect such changes in the view.

Second, message exchanges must occur according to a notion of *atomic multicast*. This grants applications the guarantee that messages are delivered to either *all* or *no* group members. Moreover, message deliveries must happen in the *same order* at all group members: a feature called *total order*. As a result, every process in a group receives the same messages in the same order. Applications thus run with the illusion that the underlying distributed executions are synchronous and fault-free, although the underlying interactions are way more complex. This greatly eases the design of dependable distributed applications, *e.g.*, based on replication techniques [3], as every replica sees the same events in the same order.

Contribution and road-map. This paper presents VIRTUS, a virtually-synchronous inter-process messaging layer we conceive for typical resource-constrained CPSs platforms, *e.g.*, limited to a few kB of RAM and with short-range low-power wireless radios. To provide virtual synchrony, VIRTUS combines a dedicated *atomic multicast service*—delivering messages reliably and with total order—with a custom *view management service*—managing group changes as nodes fail or join. This renders applicable a vast portion of the existing literature on dependable distributed systems [3], enabling formally-proven dependable operation of CPSs.

After illustrating in Sec. II the system model we base this work upon, in Sec. III we briefly describe Low-Power Wireless Bus (LWB), an existing best-effort communication protocol we use as a foundation for VIRTUS. LWB misses, however, a number of features required for virtual synchrony. Sec. IV describes the functionality VIRTUS adds to provide atomic multicast and view management, along with formal proofs that our design does provide virtually-synchronous executions. We then describe in Sec. V how we complement virtual synchrony in VIRTUS with further delivery policies, and report implementation details for our target platform in Sec. VI.

Virtual synchrony comes at a cost. Based on extensive real-world experiments, we show in Sec. VII that our VIRTUS implementation provides virtual synchrony at a marginal cost compared with LWB's best-effort operation. For example, message latency and energy consumption increase only by 1 % and 11 %, respectively. We also report on the impact of different settings of the VIRTUS parameters, demonstrating the ease to fine-tune the system.

To the best of our knowledge, we are the first to offer formally-proven virtual synchrony atop similarly resource-constrained hardware. Nevertheless, our work "stands on the shoulders of giants", leveraging decades of work on dependable distributed systems that we revisit in a new context. We provide due account of such literature in Sec. VIII, together with a brief description of CPS protocols that provide communication guarantees in specific applications.

II. SYSTEM MODEL

We consider wireless *multi-hop* networks of resource-constrained embedded devices. This generally entails a device can directly exchange data only with a subset of other nodes: those that lie within its radio range. However, nodes cooperate to relay packets on each other's behalf to enable communication between nodes outside of each other's radio range.

We target typical CPS applications where processing occurs in distinct and periodic sense-process-actuate cycles [2]. Sensing occurs at nodes equipped with application-specific sensing devices, which periodically report sensed data to nodes with attached actuators. These nodes process the data and drive the actuators accordingly. Unlike mainstream systems, a distinction therefore exists between sensor nodes—which generate data and act as *senders*—and actuator nodes—which consume data and act as *receivers*. Our work is based on such a distinction, although a node may simultaneously act as both.

We accept that both nodes and links between nodes may fail, although such failures do not occur infinitely often or liveness may be compromised. Nodes fail according to a *crash-stop* model [11], *i.e.*, nodes execute correctly until they *silently* halt and execute no further action. In principle, nothing prevents us from considering a crash-recovery failure model [11], where a process silently halts but then recovers from where it left. CPS devices, however, often lack the stable storage required to log information for recovery [12]. Nevertheless, a crash-stop model fits the reality of deployed systems, where nodes may fail because of battery depletion and lose the previous state upon rebooting when power is again available.

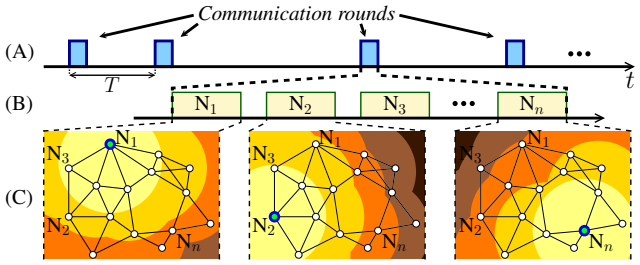


Figure 1. Time-triggered operation in LWB.

We consider a synchronous and unreliable communication model. This entails that: *i)* there is a known upper bound on message transmission delays, and *ii)* the communication channel may *silently* lose individual messages. The latter, in particular, matches experimental evidence about the *time-varying* nature of network topologies in low-power wireless, for example, due to interference and obstacles [13].

We do not consider Byzantine failures, which may affect communication or a node's state in ways different than those stipulated by a protocol's actions. For example, messages are either correctly delivered or not delivered at all—a node never processes corrupted messages. Similarly, a node's state always evolves in ways that map to a feasible execution of a protocol's actions. In general, such Byzantine failures require dedicated solutions that we plan to investigate in the near future.

III. COMMUNICATION SUPPORT

We build VIRTUS on top of Low-Power Wireless Bus (LWB), a best-effort low-power wireless protocol [14]. The key idea in LWB is to abstract away the multi-hop nature of a low-power wireless network, turning it into a communication infrastructure similar to a shared bus, where all nodes are potential receivers of all messages. To achieve this, LWB maps *all* communication demands onto Glossy network floods [15]. A Glossy flood blindly propagates every message to all nodes. Multicast communication is implemented by filtering messages at the receiver side. Although this may appear wasteful, the techniques employed by LWB, illustrated next, make it outperform state-of-the-art low-power wireless protocols [14].

LWB employs a *time-triggered* scheme to arbitrate access to the (wireless) bus: nodes are time-synchronized and communicate according to a global *communication schedule* computed and distributed by a dedicated *host* node. This adheres to the bus analogy, as traditional bus communication systems employ similar techniques [16]. The host computes the schedule based on traffic demands from the senders, but independently of the receivers' identities, as all nodes in fact receive all messages. A message is delivered to the upper-level application only if it lists the node as an intended receiver. Should the current host crash, LWB includes failover mechanisms to automatically elect a new host [14].

LWB employs a round-based operation [17]. As shown in Fig. 1 (A), activity on the bus is confined within *communication rounds*—executed simultaneously at all nodes—that repeat with a possibly varying *round period* T. Nodes keep their radios off between two rounds to save energy. Each round consists of non-overlapping *communication slots*, as shown in Fig. 1 (B). All nodes participate in the communication during a slot: one node puts a message on the bus (initiates a flood) and all other nodes read the message from the bus (receive and relay the flood), as shown in Fig. 1 (C). At the end of a slot, the intended receivers deliver the received message to the application, while all other nodes discard it.

Every round starts with a slot used by the host to distribute the communication schedule. This sched message specifies the round period T and which nodes can transmit data messages in the subsequent data slots. If a node receives the schedule, it time-synchronizes with the host and participates in the round; otherwise, it does not take any action until the next round. To inform the host about their traffic demands, nodes compete in a final *contention slot*. Because of a wireless phenomenon called "capture effect" [18], with high probability one node succeeds and reaches the host. Traffic demands take the form of *periodic streams* of data messages, as LWB targets the periodic traffic pattern typical of CPS applications [5], [1]. Based on the received traffic demands, the host computes the schedule for the next round.

We choose LWB as the foundation for VIRTUS mainly because of its bus-like operation, which eases the design of the interactions required to implement virtual synchrony. Moreover, LWB already provides some of the mechanisms required for virtual synchrony, such as: *i)* implicit total ordering *if data messages are received*, due to the exclusive access to the bus during data slots; and *ii)* an explicit join operation *for senders*, along with mechanisms to detect possible failures afterwards.

Nevertheless, using LWB, the gap to provide virtual synchrony includes functionality such as: *i) guaranteed delivery*, because LWB does not ensure by itself that messages eventually reach the intended receivers; *ii) total ordering in the presence of communication failures*, as the ordering feature in LWB, which is a side-effect of the time-triggered operation, breaks if messages are not delivered; *iii) explicit join operations for receivers*, together with the required mechanisms to detect their failures, necessary to create the group; and *iv) view management*, as a notion of view, along with its management as senders and receivers join or fail, is absent in LWB.

IV. BUILDING UP TO VIRTUAL SYNCHRONY

Many variants of virtual synchrony exist [19]. We consider the most traditional incarnation, corresponding to the intuitive definition in the Introduction. Formally, given any two nodes P and Q, any two messages ① and ② generated in any arbitrary relative order, and any two *consecutive* views V and V' that include P and Q, we wish to ensure that if P delivers message ① before message ② between view V and view V', then Q also delivers ① before ② between V and V' [9].

VIRTUS achieves the above by implementing two core functionality: *i)* an *atomic multicast service*, providing reliable and totally-ordered multicast delivery at member nodes, illustrated in Sec. IV-B; and *ii)* a *view management service*, used at any non-faulty member to maintain the list of view members,

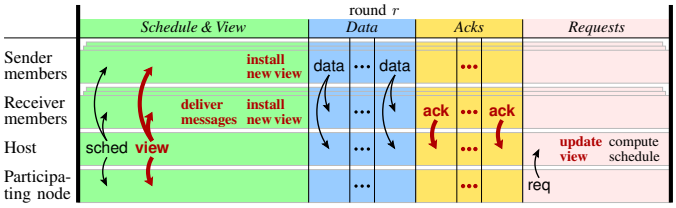


Figure 2. Operation and exchange of messages during a VIRTUS round. Highlighted in red is the functionality added to LWB.

described in Sec. IV-C. We conclude in Sec. IV-D by proving that our design provides virtual synchrony guarantees.

Compared to virtual synchrony systems for mainstream platforms [10], we reckon that some of our design choices may appear pessimistic. This is intentional, as it simplifies processing: to provide virtual synchrony in the challenging CPS scenarios, we favor easier reasoning and provably sound mechanisms over uncertain performance improvements. Arguably, the difficulty in understanding the system operation has been a hampering factor in low-power wireless [20].

A. Overview

VIRTUS provides applications with traditional virtual synchrony operations such as sending and receiving messages, and notification of view changes. Moreover, the application may use a `join()` operation to notify VIRTUS that it intends to join a view as a sender or a receiver. We distinguish between *i) view members*: non-faulty nodes that are members of the current view V; *ii) participating nodes*: nodes not yet in view V that use (or used) `join()` to notify their intent to join the view; and *iii) non-participating nodes*: nodes that only help propagate packets across multiple hops, and thus operate transparently w.r.t. virtual synchrony. A view member may also request at any time to be removed from the view.

VIRTUS round. Fig. 2 depicts an example VIRTUS *round*, highlighting the functionality added to LWB. The VIRTUS-specific processing occurs mainly at four distinct stages:

- *Schedule & View.* After a sched message, the host distributes a view message with the current view $V = \{V.id, V.\mathcal{S}, V.\mathcal{R}\}$. This consists of an identifier $V.id$ and a list of member nodes, split between senders $V.\mathcal{S}$ and receivers $V.\mathcal{R}$. Based on sched and view messages, receivers possibly deliver previously buffered messages to the application right after processing the view message. Should the received view V differ from the currently installed one, members of the new view perform a *view installation* and deliver a `view_change()` notification to the application.
- *Data.* As in the original LWB, senders in $V.\mathcal{S}$ transmit data messages during data slots according to the content of the sched message. Unlike LWB, receivers in $V.\mathcal{R}$ locally buffer received messages and wait until the next view message before possibly delivering them to the application.
- *Acks.* After the exchange of data messages, each receiver in $V.\mathcal{R}$ sends an ack message to inform the host of the set of messages currently in its buffer. This information is mainly used for reliable delivery in atomic multicast, but also to ensure correct view changes, as we describe next.
- *Requests.* As in LWB, the round ends with a non-allocated contention slot, where participating nodes compete to in-

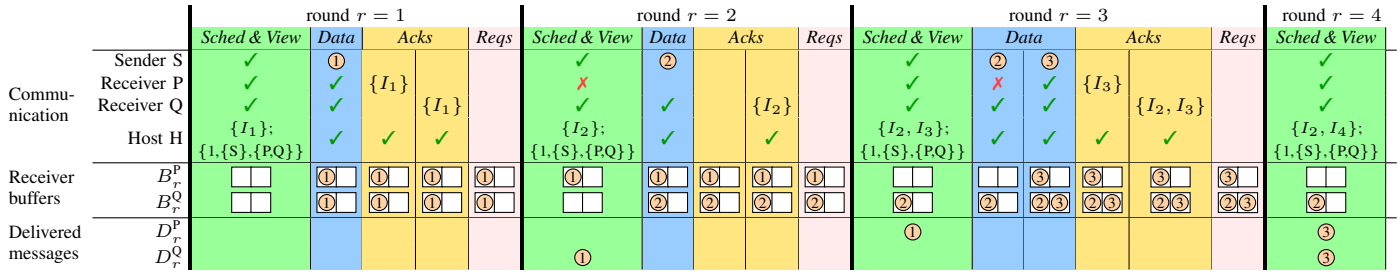


Figure 3. VIRTUS: example execution of atomic multicast. Symbol ✓ denotes a successful reception; symbol ✗ denotes a communication failure.

Table I. SETS OF MESSAGE IDENTIFIERS.

Symbol	Meaning
K_r	Messages scheduled in round r
F_r	Messages scheduled in round r for the first time
C_r	Messages previously generated by senders expelled from a view in round r
A_r	Messages acknowledged by all non-faulty receivers in round r
S_r	Messages generated by senders in the view installed at round r
B_r^{R}	Messages in the buffer of receiver R after the data slots of round r
D_r^{R}	Messages delivered by receiver R during round r
E_r^{R}	Messages discarded by receiver R during round r

form the host about their intention to join the view. If any of these nodes succeeds, the host updates the current view. The host then computes the next round schedule, based also on received ack messages and possible view updates.

Concepts and notations. We say that a view member *executes* during a round only if it receives both sched and view messages. Should instead a member fail to receive either of them, it refrains from any processing during the round. The sched message is needed for the original LWB operation, the view message is required to check the current group membership. We also call a round r *stable* if the host receives ack messages in round r from all non-faulty members in $V.\mathcal{R}$.

For simplicity, the following description considers the host as a non-participating node, although nothing prevents it from being a view member. The discussion does not consider host failures, as they are dealt with by the original LWB failover mechanisms. We show in Sec. IV-D that these mechanisms do not break the virtual synchrony guarantees VIRTUS provides.

We express the VIRTUS processing as operations on sets of message identifiers. A message identifier I_i is a triple $I_i = \{\texttt{sender\_id}, \texttt{stream\_id}, \texttt{generation\_time}\}$ that *uniquely* specifies a data message ⓘ generated by a sender. Table I summarizes the sets we introduce throughout the paper.

B. Atomic Multicast

The atomic multicast service in VIRTUS provides reliable and totally-ordered multicast to view members. As discussed in Sec. III, LWB provides neither guaranteed delivery nor total ordering in the presence of communication failures. We add the following mechanisms to fill the gap:

- Receivers buffer received data messages and use ack messages to inform the host of the messages in their buffers.
- Based on the received ack messages, the host reallocates slots for messages missing from at least one receiver buffer.
- After receiving a new sched message, receivers deliver buffered messages for which no slot is reallocated.

To ease understanding, we explain these mechanisms based on an example where nodes do not fail. We discuss in Sec. IV-C how to account for the output of the view management service.

Example. Fig. 3 shows an example execution in a network with four nodes: one sender S, two receivers P and Q, and a host H. Nodes S, P, and Q are view members and have view $V = \{1, \{\mathrm{S}\}, \{\mathrm{P}, \mathrm{Q}\}\}$ installed. Sender S has a new message ⓡ to transmit at every round r. For each slot, the figure shows messages exchanged, the content of the receiver buffers, and the messages that the receivers P and Q deliver to the application. As no other node intends to join, no req messages are transmitted. Because there are also no node failures, view V never changes. At the beginning of round $r = 1$, the receiver buffers are empty.

Round 1. The host transmits schedule $K_1 = \{I_1\}$, instructing sender S that it can transmit data message ① in the assigned data slot. In general, the schedule K_r for round r is an ordered set of message identifiers $\{I_i, I_j, \dots\}$ that senders can transmit during r; we omit the additional information in sched messages related to the LWB operation, described in Sec. III.

All nodes receive the schedule and communicate during the data slot: S transmits message ①; both P and Q receive ① and insert it into their buffers. To ensure total order, receivers buffer multiple messages in the relative order they appear in the schedule. In the two ack slots, receivers P and Q inform the host about the content of their buffers. As $B_1^{\mathrm{P}} = B_1^{\mathrm{Q}} = \{I_1\}$, both ack messages include the identifier I_1 of message ①.

Round 1 is a stable round, as the host receives ack messages from all receivers in $V.\mathcal{R}$. The host computes the set of data messages it can stop scheduling as $A_1 = \{I_1\}$. In general, for a stable round r, the set of message identifiers A_r not to reschedule in subsequent rounds is the intersection of the messages in the receiver buffers B_r^{R} of any receiver R in $V.\mathcal{R}$:

$$A_r = \bigcap\nolimits_{\mathrm{R}} B_r^{\mathrm{R}}, \quad \forall \mathrm{R} \in V.\mathcal{R}, \quad r \text{ stable} \tag{1}$$

The messages in A_r are indeed already in the receivers' buffers and do not need to be retransmitted.

Round 2. The new schedule specifies that there is only one data slot, for message ②: because all receivers in $V.\mathcal{R}$ received message ①, the host allocates no more slots for it. The schedule K_r of a generic round r is indeed obtained from the schedule of the previous round K_{r-1} by: *i)* removing the identifiers of messages acknowledged by all receivers in the previous round, included in A_{r-1}; and *ii)* possibly adding identifiers of newly-generated messages never scheduled before, included in F_r:

$$K_r = (K_{r-1} \setminus A_{r-1}) \cup F_r \tag{2}$$

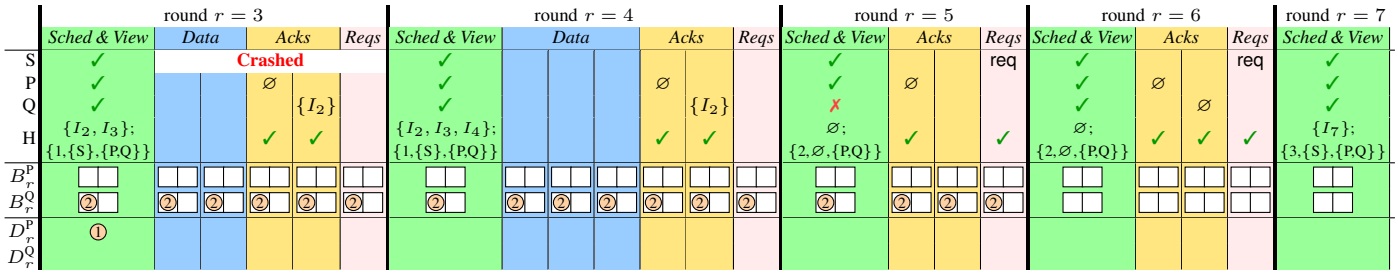


Figure 4. VIRTUS: example execution including view changes. Symbol ✓ denotes a successful reception; symbol ✗ denotes a communication failure.

Fig. 3 shows that P fails to receive the schedule, thus it does not execute in round 2 and the content of its buffer does not change: $B_2^P = \{I_1\}$. Differently, Q receives the schedule and from $K_2 = \{I_2\}$ it infers that message ① reached all receivers in $V.\mathcal{R}$ and can be delivered to the application. We indicate the delivery with $D_2^Q = \{I_1\}$. In general, during a round r, a receiver R delivers messages that are in its buffer from the previous round B_{r-1}^R and whose identifiers are not included in the current schedule K_r, meaning they reached all receivers:

$$D_r^R = B_{r-1}^R \setminus K_r \tag{3}$$

To provide totally-ordered delivery, this operation occurs in the order the messages are found in the buffer.

During the data slot, S transmits message ②, which is buffered at Q only, as P is not participating in round 2. The host receives an ack message from Q but not from P, thus round 2 is non-stable. The host computes the set of data messages not to reschedule as $A_2 = \varnothing$. This applies for any non-stable round r, as at least one receiver may be missing at least one message:

$$A_r = \varnothing, \quad r \text{ non-stable} \tag{4}$$

Round 3. For $A_2 = \varnothing$, the schedule for round 3 reassigns a slot for message ② in addition to a slot for the new message ③. From (2) indeed follows $K_3 = (\{I_2\} \setminus \varnothing) \cup \{I_3\} = \{I_2, I_3\}$.

This time, both receivers obtain the schedule. Finally, P realizes that ① reached all receivers because no slots are allocated to it in K_3, and accordingly delivers it: with $B_2^P = \{I_1\}$, from (3) follows $D_3^P = \{I_1\} \setminus \{I_2, I_3\} = \{I_1\}$. Differently, Q delivers no messages at this round, because although it already received message ②, a slot is still scheduled for it; with $B_2^Q = \{I_2\}$, from (3) follows $D_3^Q = \{I_2\} \setminus \{I_2, I_3\} = \varnothing$.

Based on schedule K_3, sender S retransmits message ②: P does not receive it, while Q does but immediately drops it as ② is already buffered from round 2. Both receivers receive and buffer ③. The host receives both ack messages $B_3^P = \{I_3\}$ and $B_3^Q = \{I_2, I_3\}$, thus the round is stable. From (1) it computes $A_3 = \{I_3\}$: only message ③ is indeed in both buffers.

Round 4. Schedule K_4 specifies that a slot is rescheduled for message ②, plus another slot is scheduled for message ④: according to (2), $K_4 = (\{I_2, I_3\} \setminus \{I_3\}) \cup \{I_4\} = \{I_2, I_4\}$. This makes both receivers deliver message ③: from (3), $D_4^P = \{I_3\} \setminus \{I_2, I_4\} = \{I_3\}$ and $D_4^Q = \{I_2, I_3\} \setminus \{I_2, I_4\} = \{I_3\}$.

Summary. Throughout the four rounds in the example, and in the presence of arbitrary communication failures, receivers P and Q deliver the same messages ① and ③ in the same order. The key to this functionality is in equations (1)–(4). These equations, however, require modifications to account for node crashes and corresponding view changes, as we illustrate next.

C. View Changes

The view management service informs the application at member nodes about the current view V and updates it in response to node crashes or recoveries. As discussed in Sec. III, LWB has no notion of view and provides no explicit support for receivers. The following mechanisms fill this gap:

- Both senders and receivers compete during contention slots and transmit req messages to the host to join the view. At each round, the host distributes the current view V, possibly updated based on node crashes and received req messages.
- The host overhears messages exchanged among view members to monitor their continuing operation. To detect member crashes, the host uses a simple counter-based scheme that marks a view member as crashed when not heard for $\overline{a}$ consecutive rounds, $\overline{a}$ being a protocol parameter whose tuning we investigate in Sec. VII.
- To provide atomic multicast also in the presence of sender and receiver crashes, delivery occurs only at receivers in $V.\mathcal{R}$ and only for messages from senders in $V.\mathcal{S}$.

We also observe that the failure detector we use may be inaccurate [11] and mistake message loss for node crashes. However, we show that even in case of false positives VIRTUS does not break virtual synchrony guarantees. We again use a concrete example to explain these mechanisms.

Example. Consider the example execution in Fig. 4. The overall setting and the first two rounds are as in Fig. 3. For simplicity, we set the number of rounds for detecting node crashes as $\overline{a} = 1$. This time, the execution unfolds as follows.

Round 3. As seen before, the schedule for this round is $K_3 = \{I_2, I_3\}$, and receiver P delivers message ①. This time, however, sender S crashes immediately after receiving the view message. As a result, S cannot (re)transmit data messages as instructed by the schedule. This potentially creates a situation violating atomic multicast: message ②, already in the buffer of receiver Q, needs to be delivered by both P and Q or neither. For simplicity we make the latter happen, based on the crash-stop model we consider for nodes, as shown in the next rounds.

In the remainder of the round, the data slots remain unused, thus the buffers at both receivers remain unchanged. Receivers send ack messages $B_3^P = \varnothing$ and $B_3^Q = \{I_2\}$, thus $A_3 = \varnothing$.

Round 4. With $K_3 = \{I_2, I_3\}$, $A_3 = \varnothing$, and $F_4 = \{I_4\}$, from (2) follows schedule $K_4 = \{I_2, I_3, I_4\}$. Sender S recovers before the beginning of this round and executes `join()`. As we consider a crash-stop model, S cannot replay the execution before the crash and transmit messages ②, ③, ④ according to the schedule. Therefore, we must treat these situations as if the recovered node were a new device, and force the view

change corresponding to the crash of the now-recovered node. To accomplish this, the recovered node keeps silent while it sees itself listed in the current view, meaning that the crash was not yet detected and no corresponding view change occurred.

As a result, although sender S in Fig. 4 receives both sched and view messages, it transmits no data message in round 4, finding itself listed in the view after booting. Being $\overline{a} = 1$ in this example, at the end of this round the host detects the crash of sender S and expels it from the updated view $\{2, \varnothing, \{\mathrm{P}, \mathrm{Q}\}\}$.

Round 5. As there is no sender in the view, $F_5 = \varnothing$. However, with $K_4 = \{I_2, I_3, I_4\}$ and $A_4 = \varnothing$, computing the schedule based on (2) would incorrectly lead to $K_5 = \{I_2, I_3, I_4\}$. These messages indeed belong to an execution of S that will never be replayed, and S will never retransmit them. Therefore, the host must stop scheduling messages generated by the crashed S. We achieve this by modifying (2) as:

$$K_r = [K_{r-1} \setminus (A_{r-1} \cup C_{r-1})] \cup F_r \tag{5}$$

where $C_{r-1} (\subseteq K_{r-1})$ includes the identifiers of messages from crashed senders removed from the view at the end of round $r-1$. In this example, $C_4 = \{I_2, I_3, I_4\}$ and, as shown in Fig. 4, $K_5 = [\{I_2, I_3, I_4\} \setminus (\varnothing \cup \{I_2, I_3, I_4\})] \cup \varnothing = \varnothing$.

During the remainder of round 5, receiver P executes and installs the new view. Receiver Q, still having message ② in its buffer, misses either the sched or view message, so it does not execute, and is stuck at the previous view that still includes sender S. The now-recovered S sees itself not listed in the new view, so it sends a req to join during the contention slot.

Round 6. Two issues may arise. First, if sender S is immediately readmitted and the view updated again, the new view $\{3, \{\mathrm{S}\}, \{\mathrm{P}, \mathrm{Q}\}\}$ would trick receiver Q to think that S never crashed. Besides the identifier, this view is indeed identical to $\{1, \{\mathrm{S}\}, \{\mathrm{P}, \mathrm{Q}\}\}$, which Q still has installed as it did not execute in round 5. Second, we need receiver Q to discard message ② when it installs view $\{2, \varnothing, \{\mathrm{P}, \mathrm{Q}\}\}$ and realizes that S crashed. As $K_6 = \varnothing$, based on (3) Q would deliver $D_6^{\mathrm{Q}} = \{I_2\} \setminus \varnothing = \{I_2\}$. This violates virtual synchrony, as the other non-faulty receiver P will never deliver ②.

To address these issues, we both postpone adding new senders until after a stable round—ensuring that all non-faulty receivers install the latest view—and modify (3) as:

$$D_r^{\mathrm{R}} = [B_{r-1}^{\mathrm{R}} \setminus K_r] \cap S_r \tag{6}$$

where S_r includes *any* message generated by senders that are members of the current view. Note that S_r is merely a formal artifact: receivers do *not* need to know the list of messages ever generated by senders. It suffices to check whether a message that a receiver is about to deliver is generated by a sender in the current view. If so, the message is delivered. If not, this message is surely not in S_r, and is discarded as it is not guaranteed to be delivered by all non-faulty receivers.

The set E_r^{R} of data messages discarded by a receiver R is:

$$E_r^{\mathrm{R}} = [B_{r-1}^{\mathrm{R}} \setminus K_r] \setminus S_r \tag{7}$$

In the example of Fig. 4, $E_6^{\mathrm{Q}} = I_2$ and message ② is discarded at Q because sender S is not in the current view. Sender S, on the other hand, not seeing itself admitted to the view, retransmits the req message during the contention slot.

Round 7. Round 6 was stable, so sender S is finally admitted to the view, a view change occurs, and the new view is disseminated to the nodes. The processing resumes normally.

Summary. The example shows how VIRTUS retains atomic multicast also against sender crashes. The processing for receiver crashes is simpler: they can be removed from a view as soon as the crash is detected, and admitted to a view as soon as they send a req. If the host expels a non-faulty receiver from a view due to the repeated loss of ack messages, such receiver empties its buffer before sending a req, as no virtual synchrony guarantees can be provided for messages already in its buffer.

As described, we integrate view management with atomic multicast by taking additional care in scheduling and delivering messages, as reflected in (5) and (6), and by possibly discarding them, as specified by (7). From these equations, and (1) and (4) from Sec. IV-B, one also observes that VIRTUS satisfies basic properties of group communication systems [19]: *i) self inclusion*: a node is a member of a view it installs; *ii) local monotonicity:* the identifiers of the views installed by a node are monotonically increasing; *iii) initial view event:* message delivery occurs within a view; and *iv) primary component membership:* views installed by nodes are totally ordered.

D. Virtual Synchrony

We prove that VIRTUS does guarantee virtual synchrony.

Bounded buffer. First, we show that VIRTUS determines an upper bound on the number of messages buffered at a receiver.

Lemma 1: At the end of a round r, the set B_r^{R} of message identifiers buffered at a receiver R is a subset of the schedule K_q received by R in the last round $q \leq r$ where R executes.

Proof: After receiving schedule K_q and view V_q during round q, every non-faulty receiver R in $V_q.\mathcal{R}$ delivers and discards buffered messages according to (6) and (7). From that moment and until the next round where R executes, R buffers only messages with identifiers in K_q and that are not already buffered. Indeed, R can add messages to the buffer only in round q, as it does not execute in any following round. ■

In general, the number of message identifiers that can fit a sched message is bounded by $\overline{m}$, for example, due to platform-dependent restrictions on packet sizes. The cardinality of any schedule K_r is thus bound to $|K_r| \leq \overline{m}$. From Lemma 1, it immediately follows that a receiver has at most $\overline{m}$ messages buffered at any point in time, and thus:

Theorem 1: A buffer size of at least $\overline{m}$ ensures that no buffer overflows occur at a receiver.

Virtual synchrony. We first prove the following lemma, which we use later to study the virtual synchrony properties.

Lemma 2: Every receiver that executes in a stable round r' and is a view member until the next stable round r'' delivers the same set of messages from the end of r' to the end of r''.

Proof: According to (5), the schedule for round $r'+1$ is $K_{r'+1} = [K_{r'} \setminus (A_{r'} \cup C_{r'})] \cup F_{r'+1}$. The schedule for the remaining rounds $r = \{r'+2, \ldots, r''\}$ is $K_r = K_{r-1} \cup F_r$, as the host removes message identifiers from the schedule only *after* a stable round, and only the last round r'' in this sequence is stable; thus, $A_{r-1} = C_{r-1} = \varnothing$ for these rounds. As a result, the schedule for a round $r = \{r'+1, \ldots, r''\}$ is:

$$K_r = (K_{r'} \setminus (A_{r'} \cup C_{r'})) \cup (F_{r'+1} \cup \cdots \cup F_r) \tag{8}$$

Consider a receiver R that is a member of every view V_r in rounds $r = \{r', \ldots, r''\}$. Receiver R thus executes in r' and does not crash between the end of r' and the end of r''. Being round r' stable, the host receives an ack message also from receiver R during this round, which in turn ensures that R receives schedule $K_{r'}$ and has view $V_{r'}$ installed during r'.

Let us name $\rho \in \{r'+1, \ldots, r''\}$ the first round after r' where R executes. Round ρ is stable if and only if $\rho = r''$. The messages delivered by R before, during, and after ρ are:

- *Rounds* $r = \{r'+1, \ldots, \rho-1\}$. Receiver R misses either the sched or the view message, thus it does not execute and delivers no messages: $D_r^{\mathrm{R}} = \varnothing$. The set of messages in its buffer remains the one of the last stable round: $B_r^{\mathrm{R}} = B_{r'}^{\mathrm{R}}$.
- *Round* ρ. Receiver R receives schedule K_ρ and view V_ρ. It delivers messages as per (6), in the same relative order their identifiers appear in the sched message of the last round where R executed, which is r'. From (6) descends:

$$D_\rho^{\mathrm{R}} = A_{r'} \tag{9}$$

  See the Appendix for the derivation of (9). Similarly, as per (7) receiver R discards messages $E_\rho^{\mathrm{R}} = B_{r'}^{\mathrm{R}} \cap C_{r'}$ generated by crashed senders. If view V_ρ differs from the currently installed view $V_{r'}$, R installs V_ρ after the message delivery.
- *Rounds* $r = \{\rho+1, \ldots, r''\}$. If receiver R misses the sched or the view message, it does not execute and delivers no messages; otherwise, it executes and delivers messages as per (6). However, we show in the Appendix that this set is also empty, because every message acknowledged by all receivers was in $A_{r'}$ and was delivered in ρ, and no stable round occurs after r' and until r''.

Because $A_{r'}$ in (9) depends neither on the specific receiver R nor on the round ρ, every non-faulty receiver delivers the same set of messages between the end of r' and the end of r''. ■

The example in Fig. 3 showed a concrete case between $r' = 1$ and $r'' = 3$. Despite message loss, both receivers P and Q deliver the same message ①. Receiver Q delivers it in round $\rho^{\mathrm{Q}} = 2$, whereas receiver P delivers it in $\rho^{\mathrm{P}} = 3$. The virtual synchrony property is a direct consequence of Lemma 2:

Theorem 2: If two receivers both install the same new view V following the same previous view V', then they deliver the same set of messages in the former.

Proof: Lemma 2 ensures that, while members of the same view $V_{r'} = \cdots = V_{r''}$, all receivers deliver the same set of messages within that view. Moreover, any receiver that installs a new view V_ρ following view $V_{r'}$ delivers the same set of messages $A_{r'}$ right before installing the new view. ■

Same view delivery. From Lemma 2 it also follows that:

Theorem 3: If two receivers deliver the same message, they deliver it in the same view.

Proof: Based on the proof of Lemma 2, a receiver R delivers messages $A_{r'}$ during round ρ and within view $V_{r'}$. As $V_{r'}$ depends neither on R nor on ρ, any receiver that delivers these messages delivers them within the same view $V_{r'}$. ■

Total ordering. The following theorem ensures that receiver members deliver messages in the same order.

Theorem 4: When receivers deliver the messages, they deliver them in the same order.

Proof: Based on the proof of Lemma 2, a receiver R delivers messages $A_{r'}$ during round ρ, and in the same relative order their identifiers had in schedule $K_{r'}$. As this order depends neither on R nor on ρ, any receiver member that delivers these messages delivers them in the same order. ■

Host failures. We observe that the theorems above hold also in the face of host failures. After a crash of the current host, nodes stop receiving sched or view messages and the *entire* VIRTUS processing stops. If the host does not recover within a specified amount of time, the LWB failover policy elects a different node as the new host [14]. In this case, the system restarts from scratch, with the new host distributing empty sched and view messages. Senders and receivers thus realize they are not listed in the view and, after discarding all buffered messages, transmit req messages to join the view.

This simple operation entails a performance overhead due to a new bootstrapping process, but it ensures that none of the virtual synchrony guarantees discussed above are violated. We plan to investigate mechanisms that keep the overhead to a minimum, *e.g.*, by making the new host reuse information included in the last view it received from the crashed host.

V. FIFO DELIVERY

Fault-tolerant distributed systems often require messages to be delivered in the same order they are sent [3], [19]. In addition to total ordering, VIRTUS provides per-node and system-wide FIFO delivery by means of very limited modifications.

Per-node FIFO ordering. The default scheduling policy of LWB, which we inherit also in VIRTUS, ensures that the host schedules in FIFO order slots for data messages from each sender [14]. In VIRTUS, however, non-faulty receivers may violate the per-node FIFO ordering when delivering messages, due to retransmissions. This happens, for example, in Fig. 3: receivers P and Q deliver message ③ before message ②.

We can provide per-node FIFO delivery with a simple modification to the scheduling algorithm at the host. The key idea is to keep scheduling slots for data messages even though these are already acknowledged by all non-faulty receivers, should these messages be generated later than non-acknowledged messages from the same sender. In the example of Fig. 3, this entails rescheduling slots for message ③ in round 4, even if it was already acknowledged by both P and Q. Because the identifier of message ③ keeps appearing in the schedule, both receivers do not deliver it as per (6).

Specifically, when the host computes the schedule K_{r+1} at the end of a stable round r, in (5) it uses $A_r^{\mathrm{nF}} \subseteq A_r$ instead of A_r. The set A_r^{nF} includes messages in A_r whose generation time at the sender is not greater than the generation time of any other message in K_r from *the same sender*.

System-wide FIFO ordering. Similarly, system-wide FIFO delivery entails that receivers deliver no messages generated before already delivered messages, regardless of the sender.

This requires two modifications. First, we change the LWB scheduler such that the system-wide FIFO ordering holds within messages scheduled for the first time, included in F_r. This is possible because the host knows when each sender generates new messages, due to the periodic generation of data by the senders. Second, similarly to the per-node FIFO delivery above, we modify how the host decides which data messages to reschedule. Specifically, when the host computes

the schedule K_{r+1} at the end of a stable round r, in (5) it uses $A_r^{\text{sF}} \subseteq A_r$ instead of A_r. The set $A_r^{\text{sF}} \subseteq A_r$ includes messages in A_r whose generation time is not greater than the generation time of any other message in K_r, *regardless of the sender*.

With either of these modifications, VIRTUS maintains totally-ordered delivery, because the mechanisms at the receivers remain the same. FIFO delivery, however, entails allocating slots not strictly needed, as the corresponding messages are already buffered at all non-faulty receivers, introducing additional overhead. We evaluate this aspect in Sec. VII.

VI. IMPLEMENTATION

We implement VIRTUS on top of the Contiki operating system [21]. We target the TelosB platform, which features a 16-bit 8 MHz MSP430 MCU, an IEEE 802.15.4-compliant 250 kbps wireless transceiver, 10 kB of RAM, and 48 kB of program memory [22]. The mechanisms added to LWB occupy only 6 kB of program memory: altogether, VIRTUS occupies 28 kB of program memory, leaving 20 kB for the application.

Compared to the original LWB implementation, we reduce from 60 to 40 the maximum number of data slots $\overline{m}$ allocated per round. This makes our prototype support up to 15 ack slots and thus up to 15 receivers, but it decreases the bandwidth available for data messages. This setting is representative of existing CPS deployments where virtual synchrony may be necessary [2], [1], [4], [5], [6]. Nevertheless, designers can tune this value based on application requirements. All other functional parameters retain the original LWB values [14].

To reduce energy consumption, the host allocates ack slots only when needed, that is, in rounds with at least one data slot or between a view update and the next stable round. The latter is to ensure that all receivers install an updated view, as discussed in Sec. IV-C. Finally, we apply an optimization to overcome the loss of view messages. If a node detects that the current view V—whose identifier $V.id$ is embedded in the sched message—is the same as the one installed, it executes even if it misses the view message, as it already knows V.

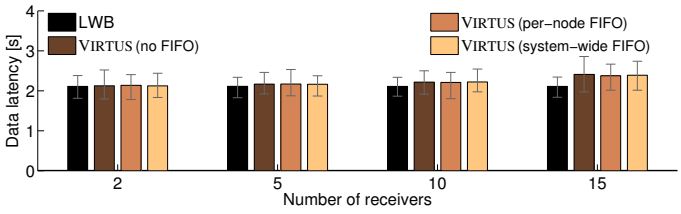
VII. EVALUATION

VIRTUS incurs a run-time overhead compared with LWB's best-effort operation. We use our prototype to quantitatively assess this aspect based on real-world experiments, and to study also the impact of different parameter settings.

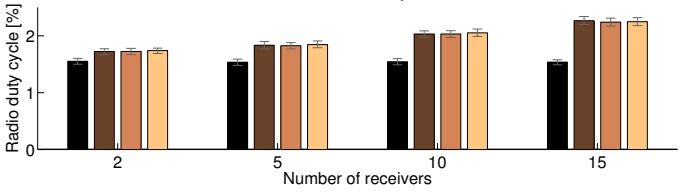
A. Settings and Metrics

We use two real-world testbeds. Twist is an indoor installation of 90 TelosB nodes spanning three floors of a university building [23]. On Twist we use an intermediate transmit power of -15 dBm, yielding a network depth of 4 hops. FlockLab includes 30 TelosB nodes with a network depth of 4 hops at the maximum transmit power of 0 dBm [24]. Both testbeds feature back-channels to every node, allowing fine-grained control of the experiments and inspection of the system state.

To factor out sources of network unreliability we cannot control, we use channel 26 to minimize interference with co-located Wi-Fi networks. In contrast, we artificially emulate message loss during ad-hoc experiments. In all experiments, data messages carry a payload of 15 bytes. Unless otherwise stated, we set $\overline{a} = 10$ as the threshold to detect node crashes.



(a) Data latency.



(b) Radio duty cycle.

Figure 5. Cost of virtual synchrony in a network of 90 nodes.

The specific traffic profile w.r.t. number of senders, receivers, and message rate varies depending on the type of experiment.

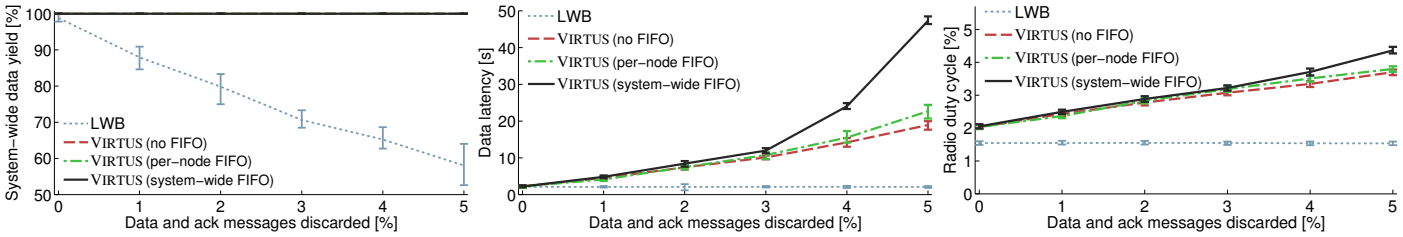
To assess the performance overhead in exchange of virtual synchrony, we consider an unmodified LWB as the baseline and measure for both LWB and VIRTUS: *i)* the *data latency*, defined as the interval from when the application at a sender sends a data message to when a receiver delivers that message; and *ii)* the *radio duty cycle*, defined as the fraction of time a node has the radio turned on, commonly regarded as an indication of energy efficiency in low-power wireless protocols [8]. We expect virtual synchrony to impact both: latency should increase because messages are delivered only after *all* receivers buffered them; radio duty cycle should increase because of additional control traffic and retransmissions absent in LWB.

Complementary to these figures, we assess how effective are the virtual synchrony guarantees VIRTUS provides by measuring: *i)* the *system-wide data yield*, defined as the fraction of generated data messages successfully delivered at *all* receivers; and *ii)* the *view latency*, defined as the interval from when a view member crashes to when all non-faulty nodes install an updated view. In the absence of crashes, the former should measure 100 % due to atomic multicast. Nevertheless, it is interesting to check the LWB performance in the same settings, to relate the gap to virtual synchrony with the performance overhead. View latency is instead useful to understand the tradeoffs for different parameter settings.

B. The Cost of Virtual Synchrony

On Twist, we randomly pick 45 senders and let them generate one data message per minute, addressed to a variable number of 2, 5, 10, and 15 receivers across different experiments. These settings depict scenarios akin to typical CPS deployments [2], [1], [4], [5], [6]. For each scenario, we run 1-hour long experiments with LWB and VIRTUS, the latter with different delivery policies: no FIFO, per-node FIFO, and system-wide FIFO. The round period is set to $T = 10$ s.

Based on the results, we verify that atomic multicast in VIRTUS delivers all data messages, whereas in LWB only 98.04 % of messages are delivered by all 15 receivers. Fig. 5 plots the performance overhead in data latency and radio duty cycle. Bars denote average values, error bars represent 15th and 85th percentiles. Fig. 5(a) shows that with two receivers LWB and VIRTUS deliver messages with similar average



(a) System-wide data yield. (b) Data latency. (c) Radio duty cycle.

Figure 6. Performance of VIRTUS in a network of 90 nodes when communication failures are artificially injected, for different types of ordered delivery.

latency of 2.11 s and 2.13 s, respectively. As LWB's reliability already approaches 100 %, most messages indeed require no retransmissions and the processing in VIRTUS resembles LWB.

With more receivers, it is more likely that *at least* one data or ack message is lost. In VIRTUS, this triggers retransmissions from the senders and buffering at the receivers. This, however, results only in a slight increase in latency, which averages 2.39 s with 15 receivers. The type of ordered delivery has little impact on latency: because of few retransmissions, only in rare cases the host reallocates data slots for already acknowledged messages to enforce FIFO delivery as described in Sec. V.

Fig. 5(b) shows the energy overhead of VIRTUS compared with LWB. With two receivers, the average radio duty cycle in VIRTUS is only 0.18 % higher than in LWB: 1.73 % against 1.55 %. More receivers entail more ack slots and a higher probability that data or ack messages are lost. Different from LWB, the radio duty cycle with VIRTUS thus increases but averages less than 2.25 % even with 15 receivers, again independently of the type of ordered delivery. Notably, this figure is way smaller than in most existing best-effort multicast protocols for low-power wireless. For instance, on the same Twist testbed and in a similar scenario with 8 receivers and 45 senders generating one data message per minute, the Muster multisink routing protocol [25] combined with the Low-Power Listening (LPL) layer [26] achieves an average radio duty cycle of 11.54 % while delivering only 98.67 % of messages [14].

C. Resilience to Network Unreliability

CPSs are often employed in scenarios with significant network unreliability, *e.g.*, due to wireless interference [27]. We evaluate the resilience of VIRTUS to these scenarios by injecting artificial message loss, in a setting with 45 senders and 10 receivers. Specifically, we make *all 90 nodes* on Twist randomly discard between 1 % and 5 % of the messages in data and ack slots, in 1 % steps. Similar scenarios are very challenging; say every node in a 4-hop route drops 5 % of messages: a simple best-effort protocol would yield only about 81 % of the messages *at a single receiver*. Similar settings are extremely unlikely to occur in real deployments. Nevertheless, they are useful to understand the behavior of VIRTUS w.r.t. network reliability. All other settings are as in Sec. VII-B.

Fig. 6 shows the results. Due to best-effort operation, the system-wide data yield in LWB increasingly suffers as the network becomes less reliable, as Fig. 6(a) shows. For example, only around 58 % of data messages are delivered by all receivers when every node discards 5 % of data messages. Atomic multicast in VIRTUS instead ensures a 100 % system-wide data yield across the board. In addition, it ensures that total ordering is adhered to even in this challenging setting.

The cost for the performance in Fig. 6(a) is illustrated in Fig. 6(b) and Fig. 6(c), showing data latency and radio duty cycle as the network becomes less reliable. With LWB, both metrics are largely independent of the network reliability: only one slot is allocated for each data message regardless of how many receivers successfully receive it. In VIRTUS, these figures increase as the network is less reliable, because more slots for data and ack messages are allocated—possibly across multiple rounds—before the receivers finally deliver.

Particularly, Fig. 6(b) shows that data latency grows significantly. The values at hand are, however, within tolerance of most CPS applications, whose dynamics often follow slowly-changing environmental phenomena, *e.g.*, temperature, and control loops run with periods of several minutes [2], [1], [4], [5], [6]. Nevertheless, we may further reduce data latency in VIRTUS by using smaller values for the LWB round period T, at the cost of increased radio duty cycle [14]. As for radio duty cycle in Fig. 6(c), the performance in absolute terms is again better than many multicast protocols for low-power wireless that only provide best-effort operation [8].

By comparing different ordering policies in Fig. 6(b) and Fig. 6(c), one notes that the performance loss is higher when FIFO delivery, and in particular system-wide FIFO, is enforced. Indeed, the latter entails the allocation of data slots for a message until *all* previous messages have been delivered, which in turn causes receivers to delay message delivery.

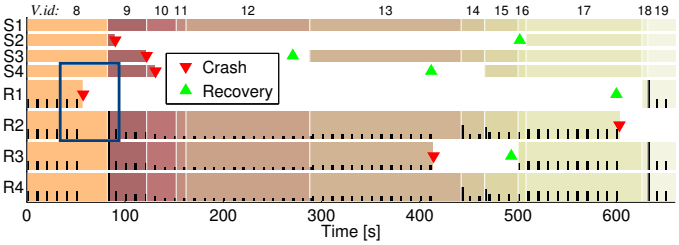
D. Influence of Parameter Setting

A key parameter in VIRTUS is $\overline{a}$: the number of rounds the host must not hear from a view member to detect a crash. This value significantly impacts the resulting view latency.

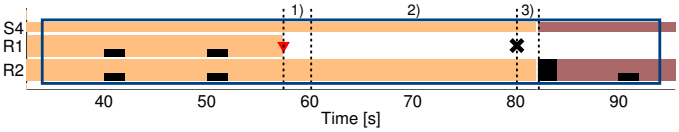
An illustrative example. We analyze a 1-hour experiment on FlockLab with four senders S1, S2, S3, S4 generating one data message every 10 s, and four receivers R1, R2, R3, R4. We emulate node crashes and recoveries by making them not communicate (corresponding to a crash) at a random instant and then reboot (corresponding to a recovery) between 0 s and 600 s after the crash. The round period T is 2 s; $\overline{a}$ is set to 10.

Fig. 7(a) shows a 12-minute excerpt of the VIRTUS operation. Symbol ▼ denotes a node crash; symbol ▲ denotes a recovery. Nodes start with view 8 installed, and the receivers deliver 4 data messages every 10 s. At $t = 57.4$ s receiver R1 crashes; the non-faulty nodes install view 9 at $t = 82$ s, yielding a view latency of 24.6 s. Fig. 7(b) zooms into this time interval and shows the contribution of $\overline{a}$ to this latency.

1) From $t = 57.4$ s to $t = 60$ s: the host allocates no data or ack slots at $t = 58$ s because the senders have no new messages to transmit. Because of this, it can not detect the



(a) Views installed (colored backgrounds) and data messages delivered per round (black bars) when 4 senders and 4 receivers randomly crash and recover.



(b) Breakdown of the view latency after the crash of a receiver.

Figure 7. VIRTUS operation across view changes.

crash of R1. The length of this stage thus depends on the interleaving of the node crash and existing traffic.

2) From $t = 60\,\text{s}$ to $t = 80\,\text{s}$: the host allocates data and ack slots for newly generated messages at $t = 60\,\text{s}, 62\,\text{s}, \ldots$, but it receives no ack messages from R1. The length of this stage is $\overline{a} \cdot T$, and in this case is 20 s.
3) From $t = 80\,\text{s}$ to $t = 82\,\text{s}$: the host detects at $t = 80\,\text{s}$ that it received no ack messages from R1 in the last $\overline{a} = 10$ rounds (✖ in the figure). As a result, it updates the view by removing R1 and starts distributing it in the next round. Non-faulty members receive and install the new view already at $t = 82\,\text{s}$. The length of this stage thus depends on when nodes successfully receive the new view.

After nodes install view 9, senders S2, S3, and S4 crash one after the other, as shown in Fig. 7(a). The following view changes occur with latencies between 30 s and 32 s, due to executions similar to the one above. As expected, within each view all active receivers deliver the same amount of messages.

Setting parameter $\overline{a}$. The example shows that the most critical stage is 2), the one affected by parameter $\overline{a}$. Determining a suitable value for $\overline{a}$ entails exploring a critical tradeoff: the smaller $\overline{a}$, the sooner the host detects node crashes and updates the view. If $\overline{a}$ is too small, however, the host may mistake message loss for crashes and trigger unnecessary view changes.

To understand this tradeoff, we run 1-hour experiments on FlockLab with 24 senders generating one data message every 30 s and 5 receivers. The round period is $T = 2\,\text{s}$. In one series of experiments, we emulate the crash of 25 view members and inject no artificial message loss. In another series of experiments, nodes do not crash but discard 10 % of data and ack messages. In both cases, we vary $\overline{a}$ between 1 and 20.

Table II reports the results. The left columns in the table confirm that a larger value of $\overline{a}$ causes a higher view latency, because the host awaits more rounds before removing a crashed node from the view. The additional data slots unnecessarily allocated to a crashed node also cause the radio duty cycle to increase with $\overline{a}$. However, the right columns show that with severe network unreliability a low value of $\overline{a}$ may lead to false positives, causing unnecessary view changes. The radio duty cycle also increases when $\overline{a}$ is too small, as nodes that are wrongly removed from a view need to re-send requests to join.

The default value $\overline{a} = 10$ in our prototype is sufficiently high to minimize the probability of false positives, while also providing reasonable view latencies. Nevertheless, a user can fine-tune the value of $\overline{a}$ according to the application requirements and the foreseeable amount of message loss.

Table II. IMPACT OF THE THRESHOLD $\overline{a}$ ON THE VIRTUS PERFORMANCE.

	0 % communication failures, 25 node crashes		10 % communication failures, 0 node crashes	
$\overline{a}$	View latency	Duty cycle	False positives	Duty cycle
1	20.64 s	2.18 %	59	3.42 %
2	21.58 s	2.22 %	8	3.22 %
3	21.80 s	2.27 %	1	3.06 %
4	22.49 s	2.27 %	0	3.07 %
10	39.34 s	2.51 %	0	3.04 %
20	56.35 s	2.95 %	0	3.10 %

VIII. RELATED WORK

VIRTUS bridges research efforts in two previously unrelated areas: virtual synchrony and low-power wireless.

Virtual synchrony lies in a vein of research originated from seminal work [28] on distributed agreement. In similar cases, however, the authors often consider a Byzantine environment, a failure model we do not study. Different flavors and implementations of virtual synchrony emerged over the years [10], often to explore the tradeoff between provided guarantees and run-time overhead. Admittedly, our incarnation almost corresponds to the "textbook" definition [9], as we hope it serves as a stepping stone for others. The work by Chockler et al. [19], who systematically survey group communication systems, helped us relate our solutions to the existing literature.

In low-power wireless, solutions exist to provide communication guarantees in specific application scenarios. For example, structural health monitoring applications [29] often require guaranteed message delivery from multiple sensors to a single data sink. Protocols such as RCRT [30] and several ad-hoc solutions [29], for instance, provide such functionality. Different from VIRTUS, however, these protocols only support a many-to-one traffic pattern. This is a mismatch against the sense-process-actuate cycles of CPS applications, which generally require many-to-many coordination. In addition, these protocols seldom provide any guarantee against node crashes. Low-power multicast protocols [8], [25], on the other hand, mostly provide only best-effort operation.

IX. CONCLUSIONS

We presented VIRTUS, a virtually-synchronous messaging layer conceived for extremely resource-constrained devices. VIRTUS provides atomic multicast and view management in CPS applications with a combination of dedicated techniques that build on an existing best-effort communication layer. We formally proved the correctness of such techniques and used extensive real-world experiments to assess their limited performance overhead compared with best-effort operation.

We maintain the value of VIRTUS lies in opening to CPSs a vast and established literature on dependable distributed systems that builds upon virtual synchrony or variations thereof. We are confident that this will increase the dependability of CPS applications to an extent that is not achievable without relying on such sound conceptual basis.

APPENDIX

To compute the set of messages delivered by receiver R after it receives schedule K_ρ and view V_ρ during round ρ, we combine (6) and (8):

$$D_\rho^{\mathrm{R}} = [B_{\rho-1}^{\mathrm{R}} \setminus [(K_{r'} \setminus (A_{r'} \cup C_{r'})) \cup (F_{r'+1} \cup \cdots \cup F_\rho)]] \cap S_\rho$$

R does not execute in rounds $r = \{r'+1, \ldots, \rho-1\}$, thus the set of messages in its buffer does not change since r': $B_{\rho-1}^{\mathrm{R}} = B_{r'}^{\mathrm{R}}$. For the same reason, before the data slots in round ρ, receiver R buffers no messages scheduled for the first time during rounds $r'+1, \ldots, \rho$, thus $B_{r'}^{\mathrm{R}} \setminus (F_{r'+1} \cup \cdots \cup F_\rho) = \varnothing$. Because the set of active senders does not change between the end of two consecutive stable rounds r' and r'', we also have that $S_\rho = S_{r'+1}$. We can thus rewrite D_ρ^{R} as:

$$\begin{aligned} D_\rho^{\mathrm{R}} &= [B_{r'}^{\mathrm{R}} \setminus [K_{r'} \setminus (A_{r'} \cup C_{r'})]] \cap S_{r'+1} \\ &= [((A_{r'} \cup C_{r'}) \cap B_{r'}^{\mathrm{R}}) \cup (B_{r'}^{\mathrm{R}} \setminus K_{r'})] \cap S_{r'+1} \end{aligned}$$

Based on Lemma 1 and knowing that R executes during stable round r', $B_{r'}^{\mathrm{R}} \subseteq K_{r'}$ and thus $B_{r'}^{\mathrm{R}} \setminus K_{r'} = \varnothing$. Moreover, $A_{r'} \cap S_{r'+1} = A_{r'}$ because all messages in $A_{r'}$ are from senders that are members during round $r'+1$, whereas $C_{r'} \cap S_{r'+1} = \varnothing$ because all messages in $C_{r'}$ are from senders that crashed by round r' and that are not members during $r'+1$:

$$D_\rho^{\mathrm{R}} = A_{r'} \cap B_{r'}^{\mathrm{R}}$$

According to (1), for stable round r' we have that $A_{r'} \subseteq B_{r'}^{\mathrm{R}}$, thus we can finally write the result in (9):

$$D_\rho^{\mathrm{R}} = A_{r'}$$

We now show that a receiver R delivers no messages during a round $r = \{\rho+1, \ldots, r''\}$, even if it executes during r. Assuming that q is the last round before r where R executed ($\rho \leq q < r$), we have $B_{r-1}^{\mathrm{R}} \subseteq K_q$ from Lemma 1 and $K_q \subseteq K_r$ from (8). The latter is because the host removes no message identifiers from the schedule between the end of two consecutive stable rounds r' and r''. By combining these two properties, $B_{r-1}^{\mathrm{R}} \subseteq K_r$, and from (6) follows $D_r^{\mathrm{R}} = \varnothing$.

Acknowledgments. We thank Gianpaolo Cugola, Arshad Jhumka, Olga Saukh, and the anonymous reviewers for their helpful comments. This work was supported by Nano-Tera, NCCR-MICS under grant #5005-67322, the Swedish Foundation for Strategic Research, and programme IDEAS-ERC, project EU-227977 SMScom.

REFERENCES

[1] M. Ceriotti *et al.*, "Is there light at the ends of the tunnel? Wireless sensor networks for adaptive lighting in road tunnels," in *Proc. ACM/IEEE Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, 2011.

[2] J. Stankovic, I. Lee, A. Mok, and R. Rajkumar, "Opportunities and obligations for physical computing systems," *IEEE Computer*, vol. 38, no. 11, 2005.

[3] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, 1990.

[4] J. Ko *et al.*, "Wireless sensor networks for healthcare," *Proc. IEEE*, vol. 98, no. 11, 2010.

[5] D. Bijwaard *et al.*, "Industry: Using dynamic WSNs in smart logistics for fruits and pharmacy," in *Proc. ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, 2011.

[6] J. Schlick, "Cyber-physical systems in factory automation – Towards the 4th industrial revolution," in *Proc. IEEE Intl. Workshop on Factory Communication Systems (WFCS)*, 2012.

[7] H. Kopetz *et al.*, "Distributed fault-tolerant real-time systems: The mars approach," *IEEE Micro*, 1989.

[8] K. Akkaya and M. Younis, "A survey on routing protocols for wireless sensor networks," *Elsevier Ad Hoc Networks*, vol. 3, no. 3, 2005.

[9] K. P. Birman, *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer-Verlag New York, Inc., 2005.

[10] K. P. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," in *Proc. ACM Symp. on Operating Systems Principles (SOSP)*, 1987.

[11] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, 1996.

[12] Y. Lee *et al.*, "A modular 1 mm^3 die-stacked sensing platform with low power I^{2}C inter-die communication and multi-modal energy harvesting," *IEEE J. Solid-State Circuits*, vol. 48, no. 1, 2013.

[13] K. Srinivasan, P. Dutta, A. Tavakoli, and P. Levis, "An empirical study of low-power wireless," *ACM Trans. on Sens. Netw.*, vol. 6, no. 2, 2010.

[14] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele, "Low-power wireless bus," in *Proc. ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, 2012.

[15] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh, "Efficient network flooding and time synchronization with Glossy," in *Proc. ACM/IEEE Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, 2011.

[16] H. Kopetz and G. Grünsteidl, "TTP – a time-triggered protocol for fault-tolerant real-time systems," in *Proc. Intl. Symp. on Fault-Tolerant Computing (FTCS-23)*, 1993.

[17] E. Gafni, "Round-by-round fault detectors: unifying synchrony and asynchrony," in *Proc. ACM Symp. on Principles of Distributed Computing (PODC)*, 1998.

[18] K. Leentvaar and J. Flint, "The capture effect in FM receivers," *IEEE Trans. Commun.*, vol. 24, no. 5, 1976.

[19] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: a comprehensive study," *ACM Comput. Surv.*, vol. 33, no. 4, 2001.

[20] J. Choi, M. Kazandjieva, M. Jain, and P. Levis, "The case for a network protocol isolation layer," in *Proc. ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, 2009.

[21] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki – A lightweight and flexible operating system for tiny networked sensors," in *Proc. IEEE Workshop on Embedded Networked Sensors (EmNetS-I)*, 2004.

[22] J. Polastre, R. Szewczyk, and D. Culler, "Telos: Enabling ultra-low power wireless research," in *Proc. ACM/IEEE Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, 2005.

[23] V. Handziski, A. Köpke, A. Willig, and A. Wolisz, "TWIST: A scalable and reconfigurable testbed for wireless indoor experiments with sensor networks," in *Proc. ACM Intl. Workshop on Multi-hop Ad Hoc Networks (REALMAN)*, 2006.

[24] R. Lim *et al.*, "FlockLab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems," in *Proc. ACM/IEEE Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, 2013.

[25] L. Mottola and G. P. Picco, "MUSTER: Adaptive energy-aware multi-sink routing in wireless sensor networks," *IEEE Trans. on Mob. Comp.*, vol. 10, no. 12, 2011.

[26] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in *Proc. ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, 2004.

[27] C.-J. M. Liang, N. B. Priyantha, J. Liu, and A. Terzis, "Surviving Wi-Fi interference in low power ZigBee networks," in *Proc. ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, 2010.

[28] M. Pease, R. Shostak, and L. Lamport, "Reaching agreements in the presence of faults," *Journal of the ACM*, vol. 27, no. 2, 1980.

[29] M. Ceriotti *et al.*, "Monitoring heritage buildings with wireless sensor networks: The Torre Aquila deployment," in *Proc. ACM/IEEE Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, 2009.

[30] J. Paek and R. Govindan, "RCRT: Rate-controlled reliable transport for wireless sensor networks," in *Proc. ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, 2007.