PISTIS: From a Word-of-Mouth to a Gentleman's Agreement

David Kozhaya[1], Jérémie Decouchant[2], Vincent Rahli[3], and Paulo Esteves-Verissimo[2]
[1]ABB Corporate Research; [2]SnT, University of Luxembourg; [3]University of Birmingham

***Abstract*—The accelerated digitalisation of society along with technological evolution have extended the geographical span of cyber-physical systems. Two main threats have made the reliable and real-time control of these systems challenging: (i) uncertainty in the communication infrastructure induced by scale, openness and heterogeneity of the environment and devices; and (ii) targeted attacks maliciously worsening the impact of the above-mentioned communication uncertainties, disrupting the correctness of real-time applications.**

This paper addresses those challenges by showing how to build distributed protocols that provide both real-time with practical performance, and scalability in the presence of network faults and attacks. We provide a suite of real-time Byzantine protocols, which we prove correct, starting from a reliable broadcast protocol, called *PISTIS*, up to atomic broadcast and consensus. This suite simplifies the construction of powerful distributed and decentralized monitoring and control applications, including state-machine replication. Extensive empirical evaluations showcase PISTIS's robustness, latency, and scalability. For example, PISTIS can withstand message loss (and delay) rates up to 40% in systems with 49 nodes and provides bounded delivery latencies in the order of a few milliseconds.

***Index Terms*—real-time distributed systems, probabilistic losses, consensus, atomic broadcast, Byzantine resilience, intrusion tolerance.**

I. Introduction

The accelerated digitalisation of society has significantly shifted the way that physical infrastructures—including large continuous process plants, manufacturing shop-floors, power grid installations, and even ecosystems of connected cars—are operated nowadays. Technological evolution has made it possible to orchestrate a higher and finer degree of automation, through the proliferation of multiple sensing, computing, and communication devices that monitor and control such infrastructures. These monitoring and control devices are distributed by nature of the geographical separation of the physical processes they are concerned with. The overall systems, i.e., the physical infrastructures with their monitoring and control apparatus, are generally known as *cyber-physical systems* (CPS) [24]. However, transposing the monitoring and control functionality normally available in classical, small-scale and homogeneous real-time and embedded systems, to the wide-scale distributed CPS scenarios mentioned above, is a very challenging task, due to two main reasons.

First, the scale and frequent openness of the environment, as well as the heterogeneity of devices (sensors, actuators and gateways), induce uncertainty in the communication infrastructure interconnecting them, itself often diverse too, e.g., Bluetooth, Wireless IEEE 802.11, or Fiber [19, 30, 15, 10]. These communication uncertainties become evident [30, 15, 10], namely in the form of link faults and message delays, which hamper the necessary reliability and synchronism needed to realize real-time operations, be it when fetching monitoring data or when pushing decisions to controllers.

Second, security vulnerabilities of many integrated devices, as well as the criticality of the managed physical structures, increase the likelihood of targeted attacks [27, 20]. Such attacks can aim to inflict inconsistencies across system components or to disrupt the timeliness and correctness of real-time applications. The consequences of such attacks can range from loss of availability to severe physical damage [29].

This paper addresses the challenges mentioned above, which render traditional approaches for building real-time communications, ineffective in wide-scale, uncertain, and vulnerable settings. We investigate, in particular, how to build large-scale distributed protocols that can provide real-time communication guarantees and can tolerate network faults and attacks. These protocols simplify the construction of powerful distributed monitoring and control applications, including state-machine replication for fault tolerance. To our knowledge, literature, with the exception of [8, 21], has targeted achieving either real-time guarantees or Byzantine-resilience with network uncertainties, but not both.

To bridge this gap, we present a protocol suite of real-time Byzantine protocols, providing several message delivery semantics, from reliable broadcast (*PISTIS*[1]), through consensus (*PISTIS-CS*), to atomic broadcast (*PISTIS-AT*). PISTIS is capable of: (i) delivering real-time practical performance in the presence of aggressive faults and attacks; and (ii) scaling with increasing system size. The main idea underlying PISTIS is to have every process adopt an event-triggered approach using digital signatures to constantly monitor how well it is connected to the rest of the network. Connectivity among processes is measured thanks to the broadcast messages: processes embed signed monitoring information within the messages of the broadcast protocol and exclude themselves from the protocol when they are a threat to timeliness. Hence, PISTIS does not modularly build on membership/failure detector oracles (like in traditional distributed computing) but rather directly incorporates such functionalities within. In fact, modularity in this sense was proven to be impossible for algorithms implementing PISTIS-like guarantees [21]. In order to mask network uncertainties in a scalable manner, PISTIS uses a temporal and spatial gossip-style message diffusion with fast signature verification schemes.

We empirically show that PISTIS is robust. For example PISTIS can tolerate message loss rates of up to 30%, 40%, and 50 % in systems with 25, 49, and 73 nodes respectively: PISTIS has a negligible probability of being unavailable under such losses. We also show that PISTIS can meet the strict timing constraints of a large class of typical CPS applications, mainly in SCADA and IoT areas, such as: release and status changes (time constants $\leq$ 10ms); fast automatic interac-

[1]PISTIS was a Greek goddess who represented the personified spirit (daimona) of trust, honesty and good faith.

tions ($\leq$ 20ms); power system automation and substation automation applications ($\leq$ 100ms); slow speed auto-control functions ($\leq$ 500ms); continuous control applications ($\leq$ 1s); and operator commands of SCADA applications ($\leq$ 2s). Such SCADA and IoT applications could include up to hundreds of devices where reliable and timely communication is required.

By using PISTIS as the baseline real-time Byzantine reliable broadcast protocol, we prove that (and show how) higher-level real-time Byzantine resilient abstractions can be modularly implemented, namely, consensus and atomic broadcast. Interestingly, we prove that this can be realized with negligible effort: (1) we exhibit classes of algorithms which are amenable to real-time operations by re-using existing synchronous algorithms from the literature; and (2) we rely on PISTIS, which addresses and tolerates the most relevant problems posed by the communication environment, including the impossibility of modularly handling membership/failure detection [21].

In short, this work makes the following contributions:

- The PISTIS protocol suite provides several message delivery guarantees (from reliable to atomic). First, PISTIS itself is an event-triggered real-time Byzantine reliable broadcast algorithm that has higher scalability and faster message delivery than conventional time-triggered real-time algorithms, in the presence of randomized and unbounded network disruptions. Building on top of PISTIS, we present classes of algorithms, PISTIS-CS and PISTIS-AT, that respectively implement real-time Byzantine consensus and atomic broadcast.
- Correctness proofs of the PISTIS protocol suite. For space reasons, we provide the main proof results in this paper, while exhaustive proofs are deferred to Appx. B.
- Extensive empirical evaluations showcasing PISTIS's robustness, latency, and scalability, based on a C++ implementation in the Omnet++ [25] network simulator.

Roadmap. The rest of the paper is organized as follows. Sec. II discusses related work. Sec. III details our system model. Sec. IV recalls the properties of a real-time Byzantine reliable broadcast, and presents our algorithm, PISTIS, in details. Sec. V shows and proves how real-time Byzantine atomic broadcast and consensus can be realized on top of PISTIS's guarantees using classes of existing algorithms. Sec. VI evaluates the performance and reliability of PISTIS. Finally, Sec. VII concludes the paper. For space limitations, proofs and additional material are deferred to Appendices.

II. Related Work

This paper has evolved from, and improved over, a research line paved by [8, 31, 21] on timing aspects of reliable broadcast and Byzantine algorithms. Besides these works, the literature on broadcast primitives, to the best of our knowledge, either does not take into account timeliness and maliciousness or addresses them separately.

Cristian et al. [8] assumed that all correct processes remain synchronously connected, regardless of process and network failures. This strong network assumption is too optimistic, both in terms of scale and timing behaviour, which in practice leads to poor performance (latency of approximately 2.4 seconds with 25 processes—see Table I in Sec. VI-D for more details). Moreover, Cristian et al.'s system model does not allow processes that malfunction (e.g., by violating timing assumptions) to know that they are treated as faulty by the model. Our algorithm, in comparison, provides latencies in the range of few milliseconds and our model makes processes aware of their untimeliness.

Verissimo et al. [31] addressed the timeliness problem by *weak-fail-silence*: despite the capability of the transmission medium to deliver messages reliably and in real-time, the protocol should not be agnostic of potential timing or omission faults (even if sporadic). The bounded omissions assumption (pre-defined maximum number of omissions) of [31] could not be taken as is, if we were to tolerate higher and more uncertain faults (as we consider in this paper): it could easily lead to system unavailability in faulty periods. Hence we operate with much higher uncertainty levels (faults and attacks).

Kozhaya et al. [21] devised a Byzantine-resilient algorithm that provides an upper bound on the delivery latency of messages. This algorithm is time-triggered and relies on an all-to-all communication that limits the algorithm's scalability. Our work improves over [21] on several points: (i) we reduce the delivery latency (few milliseconds as shown in Fig. 7 and Fig. 8 compared to a few hundred as shown in [21, Fig. 8]—see also Table I for a comparison of worst case latencies) by adopting an event-triggered approach instead of a round-based one; (ii) we improve the system's scalability (at least 5 times less bandwidth consumption) by adopting a gossip-based dissemination instead of an all-to-all communication; and (iii) we show how real-time broadcast primitives can be modularly used to build real-time Byzantine-resilient high-level abstractions like consensus and atomic broadcast.

Guerraoui et al. [18] designed a scalable reliable broadcast abstraction that can also be used in a probabilistic setting where each of its properties can be violated with low probability. They achieve a scalable solution by relying on stochastic samples instead of quorums, where samples can be much smaller than quorums. As opposed to this work, our goal is to design a deterministic abstraction where the property are never violated: RTBRB is deterministic because late processes become passive, and therefore count as being faulty.

In [5, 6], the authors present a Byzantine fault-tolerant SCADA system that relies on the Prime [3] protocol to ensure both safety and latency guarantees. As opposed to PISTIS, Prime relies on an asynchronous primary-based BFT-SMR protocol. As opposed to Prime, PISTIS-CS and PISTIS-AT algorithms are designed modularly from a timely reliable broadcast primitive; and PISTIS allows slow connections between any processes in a probabilistic synchronous environment, while Prime relies on the existence of a "stable" timely set of processes. See **??** for further details.

III. System and Threat Model

A. System Model

Processes. We consider a distributed system consisting of a set $\Pi = \{p_0, p_1, ..., p_{N-1}\}$ of $N > 1$ processes. We assume that processes are uniquely identifiable and can use digital

signatures to verify the authenticity of messages and enforce their integrity. We denote by $\sigma_i(v)$ the signature of value v by process p_i. We often write σ_i, when the payload is clear from the context. Processes are synchronous, i.e., the delay for performing a local step has a fixed known bound (note that this does not apply to faulty processes—see below).

Clocks. Processes have access to local clocks with a bounded and negligible rate drift to real time. These clocks do not need to be synchronized.

Communication. Every pair of processes is connected by two logical uni-directional links, e.g., p_i and p_j are connected by links l_{ij} and l_{ji}. Links can abstract a physical bus or a dedicated network link. We assume that links are reliable and timely with high probability. This means that in any transmission attempt, where a message is sent over a link, there is a high probability that the message reaches its destination and within a maximum delay d (known to the processes) after being transmitted. We briefly discuss why communication in our system is not synchronous and how it differs from partial synchrony [13] in Appx. A. Moreover, we also introduce a parameter X, which stands for the maximum number of processes to which a process can send a message in a communication step. This parameter can range between 0 and $N-1$, and is used to avoid network congestions by enforcing that processes selectively send their messages to an arbitrary subset of the system.

B. Threat Model

Processes. We assume that some processes can exhibit arbitrary, a.k.a. *Byzantine*, behavior. Byzantine nodes can abstract processes that have been compromised by attackers, or are executing the algorithm incorrectly, e.g., as a result of some fault (software or hardware). A Byzantine process can behave arbitrarily, e.g., it may crash, fail to send or receive messages, delay messages, send arbitrary messages, etc.

We assume that at most $f = \lfloor \frac{N-1}{3} \rfloor$ processes can be Byzantine. This formula was proved to be an upper bound for solving many forms of agreement in a variety of models such as in non-synchronous models [12, 16].

In Sec. IV, we allow nodes to become *passive* in case they fail to execute in a timely fashion. A process that exhibits a Byzantine behavior or that enters the passive mode (see Section IV-C) is termed *faulty*. Otherwise, the process is said to be *correct*. Note that passive nodes are considered faulty (at least) during the time they are passive, but are not counted against the f Byzantine faults. Therefore, more than f nodes could be faulty in a system over the full lifespan of a system (up to f nodes could be Byzantine, and up to N processes could be momentarily passive).

Clocks. The bounded and negligible rate drift assumption in the system model has to hold only on a per protocol execution basis, easily met by current technology (such as techniques that rely on GPS [33] or trusted components [32]). Hence the clock of a non-faulty process always behaves as described in Section III-A.

Communication. Links are assumed to be faithful, reliable and timely, with high probability. That is, (i) Byzantine processes or network adversaries cannot modify the content of messages sent on a link connecting correct processes (implemented by authentication through unforgeable signatures [4]). However, (ii) there is a small probability that reliability and timeliness are violated. Precisely, we define $P_{ij}(t)$ as the probability that a message transmitted on link l_{ij} (with $i \neq j$) at time t gets lost or is delayed, such that $\epsilon_1 < P_{ij}(t) < \epsilon_2 \ll 1$ are small strictly positive values. Such violations exist in networks, as arguably all communication is prone to unpredictable disturbances, e.g., bandwidth limitation, bad channel quality, interference, collisions, and stack overflows [15]. We consider message losses as pure omissions[2], and we leave it up to the system/algorithm to define how to deal with late messages (i.e., violating the d delay assumption), that is, how to tolerate *timing faults* [2], according to their anatomy in terms of negative impact on system properties [32].

IV. Real-Time Byzantine Reliable Broadcast

Let us now present our solution to guarantee that correct nodes reliably deliver broadcast messages in a timely fashion, despite Byzantine nodes, and communication disruptions. Sec. IV-A recalls the properties of the real-time Byzantine-resilient reliable broadcast (RTBRB) primitive [21]. Then, Sec. IV-B presents a high-level overview of the PISTIS event-triggered algorithm, which implements the RTBRB primitive, while Sec. IV-C provides a detailed presentation of PISTIS. Finally, Sec. IV-D explains how passive nodes can recover and become active again to ensure the liveness of the system.

A. Real-time Byzantine Reliable Broadcast Abstraction

Definition 1 (RTBRB)**.** *The real-time Byzantine reliable broadcast (RTBRB) primitive guarantees the following properties [21], assuming every message is uniquely identified (e.g., using the pair of a sequence number and a process id—the broadcaster's id).*[3] *In this abstraction, a process broadcasts a message by invoking* `RTBRB-broadcast()`*. Similarly, a process delivers a message by invoking* `RTBRB-deliver()`*.*

- ***RTBRB-Validity:*** *If a correct process p broadcasts m, then some correct process eventually delivers m.*
- ***RTBRB-No duplication:*** *No correct process delivers message m more than once.*
- ***RTBRB-Integrity:*** *If some correct process delivers a message m with sender p_i and process p_i is correct, then m was previously broadcast by p_i.*
- ***RTBRB-Agreement:*** *If some correct process delivers m, then every correct process eventually delivers m.*
- ***RTBRB-Timeliness:*** *There exists a known $\Delta_{\mathtt{R}}$ such that if a correct process broadcasts m at real-time t, no correct process delivers m after real time $t + \Delta_{\mathtt{R}}$.*

[2]We do not model correlated losses explicitly, as previous works like [21] have shown that such bursts can be mitigated.

[3]RTBRB's properties are equivalent to the ones of the Byzantine reliable broadcast abstraction defined in [7, Module 3.12,p.117], excluding *Timeliness*.

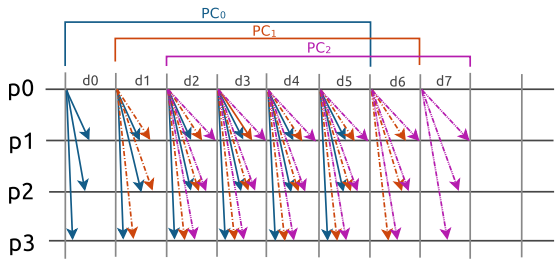

Figure 1. Example of a proof-of-connectivity run

It is important to note that the above abstraction does not enforce ordering on the delivery of messages sent. We elaborate more on that and how to achieve order in Sec. V. Note also that in a system consisting of correct and faulty nodes, these properties ensure that correct nodes deliver broadcast messages within a bounded delay, while no such guarantee is (and can be) provided about faulty nodes.

B. Overview of PISTIS

This section presents a high-level description of *PISTIS*, our Byzantine *event-triggered* RTBRB algorithm. For simplicity, we assume the total number of processes to be $N = 3f + 1$, in which case a Byzantine quorum has a size of $2f + 1$.

System Awareness. Given that broadcasts can be invoked at unknown times, there might exist a correct process in $\Pi \setminus \{p_i\}$ that is unaware of p_i's broadcast for an unbounded amount of time after it was issued, since all links can lose an unbounded number of messages. The occurrence of such scenarios may hinder the system's ability of delivering real-time guarantees. To this end, we require that every process p_j constantly exchanges messages with the rest of the system. This regular message exchange aims at capturing how well p_j is connected to other processes, and hence to what extent p_j is up-to-date with what is going on in the system (and to what extent the system knows about p_j's state). We achieve this constant periodic message exchange via a function, which we call *proof-of-connectivity*.[4] It requires each process to diffuse heartbeats to the rest of the system in overlapping rounds: a new round is started every d time units, and each round is of a fixed duration $\mathbb{T}$ (where $d < \mathbb{T}$). A round consists in repeatedly (every d units of time) diffusing a signed heartbeat message to X other processes. The set X of processes in every repetition can change such that the union of processes in all repetitions covers all processes in the system. Heartbeat messages are uniquely identified by sequence numbers, which are incremented prior to each round. On receipt of a heartbeat message, a process appends its own signature to it as well as all other seen signatures relative to that heartbeat; and sends it to X other processes. At the end of each round, if a process does not receive at least $2f + 1$ signatures (including its own) on its own heartbeat, it enters the passive mode.

Fig. 1 provides an example of a run of the proof-of-connectivity protocol, depicted as a message sequence diagram, in a system composed of 4 processes. This figure depicts part of the three first rounds of proof-of-connectivity initiated by p_0 (we only show the messages sent by p_0 to avoid cluttering the picture), namely PC_0 in blue, PC_1 in orange, and PC_2 in purple. In addition, in that case, each proof of connectivity round is of length $\mathbb{T} = 6d$. Therefore, the blue PC_0 heartbeats are sent 6 times between d_0 and d_5, the orange PC_1 heartbeats are sent 6 times between d_1 and d_6, and the purple PC_2 heartbeats are sent 6 times between d_2 and d_7. If by the end of PC_0, p_0 has not received $2f$ replies to its heartbeats, it will become passive.

Diffusing Broadcasts. PISTIS relies on two types of messages (`Echo` and `Deliver` messages) to ensure that broadcast values are delivered in a timely fashion. Processes exchange `Echo` messages either to start broadcasting new values, or in response to received `Echo` messages. `Echo` messages help processes gather a valid quorum (a Byzantine write quorum [23] of size $2f + 1$) of signatures on a single value v relative to a broadcast instance. A broadcast instance is identified by the id of the process broadcasting v and a sequence number. `Echo` messages help prevent system inconsistencies when malicious nodes send different values with the same sequence number (same broadcast instance) to different recipients. However, additional messages, namely `Deliver` messages, are needed to help achieve delivery within a bounded time after the broadcast.

When a process p_i receives a value v through an `Echo` message, it appends its signature to the message as well as all other signatures it has received relative to v; and sends it to X other processes. In addition, when p_i receives a value for the first time, it triggers a local timer of duration $\mathbb{T}$. Upon receiving a value signed by more than $2f$ processes, a process delivers that value. However, a process that does not receive more than $2f$ signatures on time (i.e., before the timer expires) enters the passive mode. In case multiple values are heard relative to a single process and sequence number (equivocation), then the first heard value is the one to be echoed. Note that processes continue executing the proof-of-connectivity function during the *echo* and *deliver* phases however by piggybacking heartbeats to echo/deliver messages.

As opposed to `Echo` messages that are diffused (i.e., re-transmitted temporally and sporadically) for a duration $\mathbb{T}$, `Deliver` messages are diffused for $2\mathbb{T}$. This is needed to ensure that if some correct processes start diffusing a message between some time t and $t + \mathbb{T}$, possibly at different times, then there must be a $\mathbb{T}$-long period of time where all of them are diffusing the message (see Lemma 4 in Appx. B for more details). Given a large enough collection of such processes ($f + 1$ correct processes), this allows other processes to learn about delivered values in a timely fashion.

Fig. 2 provides an example of a run of PISTIS, depicted as a message sequence diagram. The system is composed of 4 processes. This figure depicts part of the echo (in blue) and deliver (in orange) phases of one broadcast initiated by p_0 (for the purpose of this illustration, only the messages sent by p_0 are shown). The purple "broadcast" and "deliver" tags indicate the times at which p_0 initiated its broadcast, and delivered

[4] Periodic message exchange (heartbeats) has been used to discover the network state in many monitoring algorithms [1, 17]

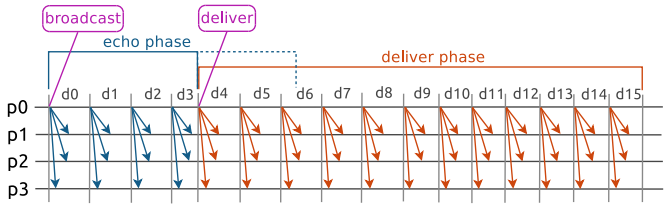

Figure 2. Example of a PISTIS run

Algorithm 1 *proof-of-connectivity*($\mathbb{T}$) @ process p_i

1: $seq = [0]^n$; *// stores smallest valid sequence number per process.*
2: $sq = 0$; *// local sequence number.*
3: $\mathcal{R}_{HB} = [\emptyset]^n$; *// stores signatures on last* $\lceil \frac{\mathbb{T}}{d} \rceil$ *heartbeats of processes.*
4:
5: **upon event** `initialization()` $\vee$ `check-connectivity()` **do**
6: **trigger** `Timeout`$(msg, \mathbb{T})$;
7: **Execute** `h-diffuse`$(\langle p_i, sq\rangle, \{\sigma_i\})$;
8: $\mathcal{R}_{HB}[p_i].add(\langle p_i, sq\rangle; \{\sigma_i\})$; $sleep(d)$; sq++;
9: **if** $sq - seq[p_i] > \lceil \frac{\mathbb{T}}{d} \rceil$ **then** $seq[p_i]$++;
10: **end if**
11: **trigger** `check-connectivity`();
12:
13: **upon event** `Expired-Timer`$(\langle p_i, sq'\rangle, timeout)$ **do**
14: **if** $|\mathcal{R}_{HB}[p_i].getsig(sq')| \leq 2f$ **then**
15: *// gets signatures on message with sequence number* sq'
16: **Initiate** *passive mode*;
17: **else** $\mathcal{R}_{HB}[p_i].remove(sq')$; *// remove entry with seq. num.* sq'
18: **end if**
19:
20: **upon event** receive `HB` $(\langle p_j, sq'\rangle, \Sigma)$ **do**
21: **if** $(sq' \geq seq[p_j])$ **then**
22: $\mathcal{R}_{HB}[p_j].setsig(sq', \mathcal{R}_{HB}[p_j].getsig(sq') \cup \Sigma \cup \{\sigma_i\})$;
23: **if** $j \neq i \wedge sq' \neq seq[p_j]$ **then**
24: **Execute** `h-diffuse`$(\langle p_j, sq'\rangle, \mathcal{R}_{HB}[p_j].getsig(sq'))$;
25: **end if**
26: **end if**
27: **if** $(sq' - seq[p_j]) > \lceil \frac{\mathbb{T}}{d} \rceil \wedge j \neq i$ **then**
28: $seq[p_j] = sq' - \lceil \frac{\mathbb{T}}{d} \rceil$;
29: $\mathcal{R}_{HB}[p_j].remove(sq'')$, $\forall sq'' < seq[p_j]$;
30: **end if**
31:
32: **Function** `h-diffuse`(msg, Σ)
33: **for** (`int` $i = 0$; $i \leq \lceil \frac{\mathbb{T}}{d} \rceil$; i++) **do**
34: **send** `HB` (msg, Σ) to X other processes;
35: $sleep(d)$;
36: **end for**
37:

it. In this example, the echo phase is initially meant to last for a duration of $\mathbb{T} = 6d$. However, it happens here that p_0 received $2f$ echo messages for its broadcast by $3d + k$, where $0 < k < d$, which is why d_3 is shorter than the other intervals. Therefore, p_0 stops its echo phase and starts its deliver phase at $3d + k$. As mentioned above, the deliver phase lasts for $2\mathbb{T}$. If p_0 has not received $2f$ deliver messages in return by the end of that deliver phase, then it becomes passive.

C. Detailed Presentation of PISTIS

We now discuss PISTIS (Algorithm 2) in more details.[5] PISTIS's proof of correctness can be found in Appx. B.

Process states. Processes can become passive under certain scenarios by calling "**Initiate** *passive mode*". Processes that were behaving correctly thus far, are considered faulty when

[5]Note that all functions presented in Algorithms 1 and 2 are non-blocking.

Algorithm 2 PISTIS @ process p_i

1: **Execute** *proof-of-connectivity*($\mathbb{T}$);
2:
3: **upon event** `RTBRB-broadcast`(p_i, sq, v) **do**
4: **Execute** *proof-of-connectivity* in piggyback mode;
5: **Initialize** $\mathcal{R}_{echo}(p_i, sq, v) = \{\sigma_i\}$;
6: **Execute** `t-diffuse`$(\langle p_i, sq, v\rangle, \mathbb{T}, \texttt{echo})$;
7:
8: **upon event** receive `Echo` $(\langle p_j, sq, v\rangle, \Sigma)$ **do**
9: **if** $\nexists \mathcal{R}_{echo}(p_j, sq, ...)$ **then**
10: **Initialize** $\mathcal{R}_{echo}(p_j, sq, v) = \{\sigma_i\} \cup \Sigma$;
11: **Execute** *proof-of-connectivity* in piggyback mode;
12: **if** $\mathcal{R}_{echo}(p_j, sq, v) \leq 2f$ **then**
13: **Execute** `t-diffuse`$(\langle p_j, sq, v\rangle, \mathbb{T}, \texttt{echo})$;
14: **else Execute** `deliver-msg`$(p_j, sq, v, \mathcal{R}_{echo}(p_j, sq, v))$;
15: **end if**
16: **else if** $\exists \mathcal{R}_{echo}(p_j, sq, v)$ **then**
17: $\mathcal{R}_{echo}(p_j, sq, v) = \mathcal{R}_{echo}(p_j, sq, v) \cup \Sigma$;
18: **if** $\mathcal{R}_{echo}(p_j, sq, v) > 2f$ (for the first time) **then**
19: **Execute** `deliver-msg`$(p_j, sq, v, \mathcal{R}_{echo}(p_j, sq, v))$;
20: **end if**
21: **else if** $\exists \mathcal{R}_{echo}(p_j, sq, v' \neq v)$ **then**
22: *// p_j has lied about message with sq*
23: **if** $\Sigma > 2f$ **then**
24: **garbage-collect** $\mathcal{R}_{echo}(p_j, sq, v')$;
25: $\mathcal{R}_{echo}(p_j, sq, v) = \Sigma$;
26: **Execute** `deliver-msg`(p_j, sq, v, Σ);
27: **end if**
28: **end if**
29:
30: **upon event** receive `Deliver` $(\langle p_j, sq, v, \Sigma\rangle, \Sigma')$ **do**
31: **if** $\nexists \mathcal{R}_{deliver}(p_j, sq, v)$ **then**
32: $\mathcal{R}_{echo}(p_j, sq, v) = \mathcal{R}_{echo}(p_j, sq, v) \cup \Sigma$;
33: **Execute** `deliver-msg`(p_j, sq, v, Σ);
34: **end if**
35: $\mathcal{R}_{deliver}(p_j, sq, v) = \mathcal{R}_{deliver}(p_j, sq, v) \cup \Sigma'$;
36:
37: **upon event** `Expired-Timer`$(msg, timeout, mode)$ **do**
38: **if** $\exists \mathcal{R}_{mode}(msg) \wedge |\mathcal{R}_{mode}(msg)| \leq 2f$ **then**
39: **switch** *mode* **do**
40: **case** `echo`
41: **if** no lie is discovered on msg **then**
42: **Initiate** *passive mode*;
43: **end if**
44: **case** `deliver`
45: **Initiate** *passive mode*;
46: **end if**
47:
48: **Function** `t-diffuse`$(msg, timeout, mode)$
49: **trigger** `Timeout`$(msg, timeout, mode)$;
50: **for** (`int` $i = 0$; $i \leq \lceil \frac{timeout}{d} \rceil$; i++) **do**
51: $\Sigma = \mathcal{R}_{mode}(msg)$;
52: **switch** *mode* **do**
53: **case** `echo`
54: **send** `Echo` (msg, Σ) to X random processes;
55: **case** `deliver`
56: **send** `Deliver` (msg, Σ) to X random processes;
57: `sleep(`d`)`;
58: **end for**
59:
60: **Function** `deliver-msg`$_{p_i}(p_j, sq, v, \Sigma)$
61: **if** $\nexists \mathcal{R}_{deliver}(p_j, sq, v)$ **then**
62: **Execute** *proof-of-connectivity* in piggyback mode;
63: **trigger** `RTBRB-deliver`(p_j, sq, v);
64: **Initialize** $\mathcal{R}_{deliver}(p_j, sq, v) = \{\sigma_i\}$;
65: **Stop** sending any `Echo` ()
66: **end if**
67: **Execute** `t-diffuse`$(\langle p_j, sq, v, \Sigma\rangle, 2\mathbb{T}, \texttt{deliver})$;
68:

they initiate a passive mode and can notify the application above of this fact. Later in this section, we show how processes in the passive mode can come back to normal operation by calling "**Initiate** *active mode*".

Ensuring sufficient connectivity. In PISTIS every process executes the *proof-of-connectivity* Algorithm 1. Namely, a process p_i forms a heartbeat $\texttt{HB}(\langle p_i, sq\rangle, \{\sigma_i\})$, where sq is p_i's current heartbeat sequence number and σ_i is p_i's signature on $\langle p_i, sq\rangle$. Process p_i also stores (in array $\mathcal{R}_{HB}$) for every process (including itself) all signatures it receives on heartbeats with a *valid sequence number*. A valid heartbeat sequence number for some process p_j is a sequence number $\in [seq[p_j], seq[p_j] + \lceil \frac{\mathbb{T}}{d} \rceil]$ where $seq[p_j] + \lceil \frac{\mathbb{T}}{d} \rceil$ is the largest heartbeat sequence number known for p_j. Heartbeats with invalid sequence numbers are simply ignored. After forming its heartbeat, p_i sets a timeout of duration $\mathbb{T}$, and sends this heartbeat to $X > f$ random processes $\lceil \frac{\mathbb{T}}{d} \rceil$ times (lines 32–36). Process p_i increments its heartbeat sequence number and repeats this whole procedure every $d < \mathbb{T}$. Upon incrementing its heartbeat sequence number, p_i updates its own valid heartbeat sequence numbers (lines 9–10).

A process p_i receiving $\texttt{HB}(\langle p_j, sq'\rangle, \Sigma)$ ignores this heartbeat if sq' is smaller than the smallest valid heartbeat sequence number known for p_j. Otherwise, p_i updates p_j's valid heartbeat sequence numbers (lines 27–30) and the list of all seen signatures on these valid heartbeats (line 22). Then, p_i diffuses the heartbeat with the updated list of seen signatures to X random processes (line 24).

When a timer expires, p_i checks $\mathcal{R}_{HB}[p_i]$ for the number of accumulated signatures on its corresponding heartbeat. If that number is $\leq 2f$, p_i enters the passive mode; otherwise it removes the corresponding entry from $\mathcal{R}_{HB}[p_i]$ (lines 13–19).

Broadcasting a message. A process p_i that wishes to broadcast a value v, calls $\texttt{RTBRB-broadcast}(p_i, sq, v)$ from Algorithm 2 (lines 3–7), where sq is a sequence number that uniquely identifies this broadcast instance. Given such an event, p_i produces a signature σ_i for the payload $\langle p_i, sq, v\rangle$. It then triggers a timeout of duration $\mathbb{T}$ and sends an $\texttt{Echo}(\langle p_i, sq, v\rangle, \{\sigma_i\})$ message $\lceil \frac{\mathbb{T}}{d} \rceil$ times to X other random processes. *Proof-of-connectivity* information from p_i is now piggybacked on these messages, as on all other $\texttt{Echo}()$ and $\texttt{Deliver}()$ messages.

Sending and Receiving Echoes. When p_i receives an $\texttt{Echo}(\langle p_j, sq, v\rangle, \Sigma)$, p_i reacts differently depending on whether it is not already echoing for this instance (lines 8–15), already echoing v (lines 16–20), or already echoing a different value (lines 21–27). In all three cases, p_i starts delivering a message (and stops sending echoes) as soon as at least $2f+1$ distinct signatures have been collected for that message.

Sending and Receiving Deliver Messages. When p_i receives $\texttt{Deliver}(\langle p_j, sq, v, \Sigma\rangle, \Sigma')$ for the first time (lines 60–67), it delivers $\langle p_j, sq, v, \Sigma\rangle$, and sends $\texttt{Deliver}(\langle p_j, sq, v, \Sigma\rangle, \mathcal{R}_{deliver}(p_j, sq, v))$ using $\texttt{t-diffuse}()$. In case that *deliver* message is not the first one received (lines 30–35), p_i aggregates all seen signatures for $\langle p_j, sq, v\rangle$ in $\mathcal{R}_{deliver}(p_j, sq, v)$ (all functions that use $\mathcal{R}_{deliver}(p_j, sq, v)$ now use the new updated value).

Process Passive Mode. When a timeout set by process p_i with parameters $(msg, timeout, mode)$ expires, p_i enters the passive mode if the set $\mathcal{R}_{mode}$ has less than $2f+1$ distinct signatures, for $mode = \texttt{deliver}$. For $mode = \texttt{echo}$, p_i enters passive mode if in addition to $\mathcal{R}_{mode}$ not having $2f+1$ signatures, p_i did not discover a lie for that broadcast instance.

Remark 1. *Any message of the form* $\texttt{Echo}(\langle p_j, sq, v\rangle, \Sigma_1)$ *or* $\texttt{Deliver}(\langle p_j, sq, v, \Sigma_2\rangle, \Sigma_3)$ *is termed invalid if: (1)* Σ_1 *contains an incorrect signature (and similarly for* Σ_2 *and* Σ_3*); or (2)* Σ_1 *does not contain a signature from* p_j *(and similarly for* Σ_2*); or (3)* Σ_2 *has less than* $2f+1$ *signatures. Invalid messages are simply discarded.*

Remark 2. *We assume that processes sign payloads of the form* (p_i, sq, v, E) *for echo messages and of the form* (p_i, sq, v, D) *for deliver messages. We use the* E *and* D *tags to distinguish echo and deliver payloads, thereby ensuring that an attacker cannot use echo signatures as deliver signatures. Note that echo signatures are sent as part of deliver messages as a proof that a quorum of processes echoed a certain value.*

As mentioned at the beginning of this section, PISTIS is correct is the sense that it satisfies all five properties of the RTBRB primitive presented in Sec. IV-A:

Theorem 1 (Correctness of PISTIS)**.** *Under the model presented in Sec. III, the PISTIS algorithm presented in Fig. 2 implements the RTBRB primitive.*

A proof of this theorem can be found in Appx. B. Let us point out here that the $\Delta_{\mathtt{R}}$ bound of the RTBRB-Timeliness property turns out to be $3\mathbb{T}$.

D. Byzantine-Resilient Recovery

If process p_i detects that it is executing under bad network conditions, it enters the passive mode and signals the upper application. As a result, p_i stops broadcasting and delivering broadcast messages (by not executing line 3 and line 63) to avoid violating RTBRB-Timelines. However, p_i continues participating in the dissemination of the broadcast and proof-of-connectivity messages to avoid having too many nodes not collecting enough messages and hence becoming passive.

Once the network conditions are acceptable again, p_i can recover and resume delivering broadcast messages. More precisely, a process p_i that enters passive mode at time t can operate normally again if the interval $[t, t+\Delta_{\mathtt{R}}]$ is free of any passive mode initiations. This $\Delta_{\mathtt{R}}$ duration ensures that the messages delivered by a recovered process p_i do not violate any RTBRB properties. After a delay $\Delta_{\mathtt{R}}$, nodes will resume their full participation in the protocol, and either deliver messages or stay on hold.

Note that in case of multiple broadcast instances, passive nodes that become active again should learn the latest sequence number of broadcasts for other nodes. Otherwise Byzantine nodes can exploit this to hinder the liveness of the system.

Remark 3. *Given that processes can now shift between passive and active modes, we specify our notion of correct processes as follows. A system run is modeled by a trace of events happening during that run. An event has a timestamp and a node associated with it. Moreover, an event can either be a correct event or a Byzantine event. Given an algorithm A, a process p is deemed correct w.r.t. A and a trace τ, if: (1) it follows its specification from e_1, the first correct A-related event happening in τ, to e_2, the last correct A-related event happening in τ; (2) p's events between e_1 and e_2 must all be correct; (3) p must also have followed its specification since it last started; and (4) p must never have lost its keys (so that no other node can impersonate p when p follows its specification). The results presented below also hold for this definition of correctness, because correct processes are required to be active through the entire broadcast instance.*

This recovery mechanism improves the overall resilience of the system. Indeed, having all processes in passive mode can occur if $2f + 1$ nodes are passive, which is now harder to achieve if nodes can recover sufficiently fast enough.

V. Beyond a Reliable Broadcast

Unlike liveness in asynchronous reliable broadcast, the RTBRB-Timeliness property (a safety property) introduces a scent of physical ordering. This ordering is due to the fact that timeliness stipulates, for each execution, a termination event to occur "at or before" some Δ_R on the time-line. This said, the reader may wonder to what extent does the real-time Byzantine-resilient reliable broadcast (of Sec. IV-A) help in establishing total order?

The answer to this question lies in examining what happens to multiple broadcasts issued by the same or by different nodes. When multiple broadcasts interleave, e.g., when they are issued within a period shorter than Δ_R (the upper time bound on delivering a message), messages might be delivered to different processes in different orders. The timeliness property of the real-time Byzantine-resilient reliable broadcast only ensures that a message m that is broadcast at time t is delivered at any time in $[t, t + \Delta_R]$. Thus, to ensure total order on all system events, e.g., for implementing *State Machine Replication*, additional abstractions need to be built on top of the real-time Byzantine-resilient reliable broadcast primitive that we have developed so far.

In this section, we investigate how to modularly obtain such an order on system events while still preserving real-time and Byzantine-resilience. We define two build blocks that build on top of RTBRB, namely the RTBC real-time Byzantine consensus abstraction (Def. 2)—a fundamental building block for state machine replication, atomic broadcast and leader election [9, 7]; and the RTBAB real-time atomic broadcast abstraction (Def. 4)—to establish total order on system events. We then provide characterizations of classes of algorithms that implement these abstractions: Thm. 2 provides a characterization of the PISTIS-CS class of algorithms that implement RTBC, while Thm. 3 provides a characterization of the PISTIS-AT class of algorithms that implement RTBAB. Finally, we provided examples of algorithms that belong to these classes (see Examples 1 and 2).

We start with the following assumption that constrains the ways processes can communicate.

Assumption 1. *Correct processes access the network only via the RTBRB primitive, namely using the two operations:* `RTBRB-broadcast`() *and* `RTBRB-deliver`().

From Assumption 1, a correct process p_i that receives a message from an operation other than `RTBRB-deliver`() simply ignores that message by dropping it.

A. Real-Time Byzantine Consensus

Roughly speaking, solving the *Byzantine consensus* problem consists in having distributed processes agree on a given value, even though some of the processes may fail arbitrarily. Byzantine consensus was first identified by Pease et al. [28], and formalized as the *interactive consistency* problem. An algorithm achieves interactive consistency if it allows the non-faulty processes to come to a consistent view of the initial values of all the processes, including the faulty ones. Once interactive consistency has been reached, the non-faulty processes can reach consensus by applying a deterministic averaging or filtering function on the values of their view. We apply the following assumption to reach consensus.

Assumption 2. *Once interactive consistency terminates, every correct process scans the obtained vector and decides on the value that appears at least $2f + 1$ times. If no such value exists, then the process decides $\perp$, a distinguished element that indicates that no value has been decided.*

Definition 2 (RTBC)**.** *The real-time Byzantine consensus (RTBC) abstraction is expressed by the following properties:*[6]

- ***RTBC-Validity:*** *If all correct processes propose the same value v, then any correct process that decides, decides v. Otherwise, a correct process may only decide a value that was proposed by some correct process or $\perp$.*
- ***RTBC-Agreement:*** *No two correct processes decide differently.*
- ***RTBC-Termination:*** *Correct processes eventually decide.*
- ***RTBC-Timeliness:*** *If a correct process p_i proposes a value to consensus at time t, then no correct process decides after $t + \Delta_C$.*

In RTBC a process p_i can propose a value v to consensus by invoking `RTBC-propose`$(p_i, inst, v)$, where $inst$ is a sequence number that uniquely identifies a RTBC instance. Similarly, a process p_i decides on a value v by invoking `RTBC-decide`$(p_i, inst, v)$. In addition `RTBC-init`$(inst)$ instantiate a new instance of RTBC with id $inst$, i.e., for sequence number $inst$.

Definition 3. *An algorithm is said to be* bounded *if it only uses a known bounded number of communication rounds.*

[6]The properties of RTBC are the same as the ones of the traditional (strong) Byzantine consensus defined in [13] (see also [7, Module 5.11,p.246]), excluding the *Timeliness* property.

Theorem 2 (Characterization of the PISTIS-CS class)**.** *Let PISTIS-CS be the class of* bounded *(Def. 3) algorithms that implements interactive consistency under Assumptions 1 and 2. Then, PISTIS-CS algorithms also implement RTBC in our model (described in Sec. III).*

See Appx. C for a proof of this result.

Example 1 (Examples of PISTIS-CS algorithms)**.** *Because the interactive consistency problem has been solved using different algorithms that satisfy Def. 3, our result applies to various existing algorithms, such as [28, 11, 14, 22].*

B. Real-Time Byzantine-Resilient Atomic Broadcast

Definition 4 (RTBAB)**.** *A real-time Byzantine-resilient atomic broadcast (RTBAB) has the same properties as RTBRB (with a different timeliness bound) plus an additional ordering property (therefore, we only present the properties that differ from RTBRB's):*

- ***RTBAB-Timeliness:*** *There exists a known $\Delta_{\mathtt{A}}$ such that if a correct process broadcasts m at time t, no correct process delivers m after real time $t + \Delta_{\mathtt{A}}$.*
- ***RTBAB-Total order:*** *Let m_1 and m_2 be any two messages and suppose that p_i and p_j are any two correct processes that deliver m_1 and m_2. If p_i delivers m_1 before m_2, then p_j delivers m_1 before m_2.*

We now define the class of algorithms (called $\mathcal{R}ound\mathcal{B}ased$), through the properties listed below, that modularly implement RTBAB properties. $\mathcal{R}ound\mathcal{B}ased$ algorithms make use of a single RTBRB instance and multiple instances of RTBC. We first constrain a $\mathcal{R}ound\mathcal{B}ased$ algorithm to start an RTBRB instance within a bounded amount of time for any broadcast call.

Property 1. *If a correct process p_i RTBAB-broadcasts a message m at time t, then it also RTBRB-broadcasts m by time $t + \Delta_B$, for some bounded Δ_B.*

We then require a $\mathcal{R}ound\mathcal{B}ased$ algorithm to start (or end in case this has already been done before) an RTBC instance, within a bounded amount of time, every time the RTBRB instance delivers.

Property 2. *If a correct process RTBRB-delivers a message m at time t, such that m's broadcaster is also correct, then it either RTBC-proposes or RTBC-decides m by $t + \Delta_P$, for some bounded Δ_P.*

In addition, the next property constrains the values that can be proposed at each RTBC instance, namely that at most one non-$\perp$ value can be proposed at each instance.

Property 3. *Given an RTBC instance $inst$, there exists a value v, such that each correct process either RTBC-propose v or $\perp$ at $inst$.*

Next, we require a $\mathcal{R}ound\mathcal{B}ased$ algorithm to deliver a RTBC-decided value within a bounded amount of time (Property 4) and to ensure that non-RTBC-decided values are re-proposed in later RTBC rounds (Property 5).

Property 4. *If a correct process RTBC-decides a message m at time t, then it also RTBAB-delivers m by time $t + \Delta_D$, for some bounded Δ_D.*

Property 5. *A correct process p_i that proposes a value v at a given time t, using a given RTBC instance $inst$, and such that this instance does not decide v, also RTBC-propose v at some instance $inst + k$, where $0 < k$. Moreover, p_i RTBC-proposes v at the smallest instance between $inst+1$ and $inst+k$ where m is proposed by some process.*

Finally, we require that nodes participate in all successive RTBC instances in a monotonic fashion.

Property 6. *Correct processes RTBC-propose exactly one value per RTBC instance; propose values in all RTBC instances (i.e., for all instances $inst \in \mathbb{N}$); in increasing order w.r.t. the instance numbers of the RTBC instances (i.e., if p_i proposes values at times t_1 and t_2 using the RTBC instances $inst_1$ and $inst_2$, respectively, and $t_1 < t_2$, then $inst_1 < inst_2$); and not in parallel (i.e., if p_i proposes a value at time t using an RTBC instance $inst$, and that this RTBC instance has not decided by time $t' > t$, then p_i does not propose any other value between t and t').*

Definition 5. *Let $\mathcal{R}ound\mathcal{B}ased$ be the class of* round-based *algorithms that satisfy the properties 1, 2, 3, 4, 5, and 6.*

Theorem 3 (Characterization of the PISTIS-AT class)**.** *Let PISTIS-AT be the class of $\mathcal{R}ound\mathcal{B}ased$ algorithms that implement the traditional Byzantine total-order broadcast under Assumption 1. Then, PISTIS-AT algorithms also implement RTBAB in our system (described in Sec. III).*

To prove Theorem 3, it is sufficient to prove that a RTBAB-broadcasted value m is always RTBAB-delivered within a bounded amount of time. Because $\mathcal{A}$ is round-based, m must be RTBRB-proposed and RTBRB-decided within a bounded amount of time. Consequently there is (within a bounded amount of time) an RTBC instance where "enough" correct nodes RTBC-propose m, so that m gets RTBC-decided upon and RTBAB-delivered within a bounded amount of time. The proof of Theorem 3 is detailed in Appx. D.

Example 2 (Example of a PISTIS-AT algorithm)**.** *Finally, algorithm 3 provides an example of a PISTIS-AT algorithm that implements RTBAB modularly, which we adapted from [7, Alg.6.2,p.290] to guarantee timeliness.*

VI. Evaluation

In this section, we evaluate PISTIS's reliability, latency, and incurred overhead on network bandwidth.

A. Optimizations

We implemented three optimizations to improve the performance of PISTIS (as described in Section IV-C). (1) If a process p_i knows that some process p_j has already received $2f + 1$ *echo* signatures for some message m, p_i stops sending echoes related to m to p_j. Every process implements this optimization by maintaining a list, say $\mathcal{L}$, that contains all the

Algorithm 3 Example of a *PISTIS-AT* algorithm @process p_i

```
1: upon event RTBAB-init(rtbab) do
2:     unordered = []^n; next = [0]^n; seq = 0;
3:     delivered = ∅; busy = False; inst = 0;
4:
5: upon event RTBAB-broadcast(p_i, m) do
6:     trigger RTBRB-broadcast(p_i, seq, m);
7:     seq++;
8:
9: upon event RTBRB-deliver(p_j, num, m) do
10:    if num = next[p_j] then
11:        next[p_j] = next[p_j] + 1;
12:        if m ∉ delivered then
13:            unordered[p_j] = unordered[p_j].append(⟨p_j, m⟩);
14:        end if
15:    else {wait(Δ_W); triggerRTBRB-deliver(p_j, num, m);}
16:    end if
17:
18: upon event ∃p_j : unordered[p_j] ≠ [] ∧ busy = False do
19:    busy = True;
20:    trigger RTBC-init(inst);
21:    // initiate a new real-time Byzantine consensus instance
22:    if unordered[leader(inst)] ≠ [] then
23:        m = unordered[leader(inst)].head();
24:    else {m = ⊥;}
25:    end if
26:    trigger RTBC-propose(p_i, inst, m);
27:
28: upon event RTBC-decide(p_i, inst', decided) do
29:    if inst' = inst then
30:        if decided ∉ delivered ∧ decided ≠ ⊥ then
31:            delivered = delivered ∪ {decided};
32:            trigger RTBAB-deliver(leader(inst), decided);
33:        end if
34:        unordered[leader(inst)].remove(decided);
35:        inst++; busy = False;
36:    else {wait(Δ_W); trigger RTBC-decide(p_i, inst', decided);}
37:    end if
38:
39: Function leader(instance) {return(instance mod n);}
40:
```

processes from which it has heard $2f+1$ signatures for a given message. During a broadcast, a process diffuses a message to X processes at random among $\Pi \setminus \mathcal{L}$. Processes do the same for *deliver* messages. (2) Processes do not verify signatures that they have already received. (3) Processes skip messages that only contain signatures that were already received.

B. Methodology and Parameter Settings

We implemented PISTIS in C++ on the Omnet++ 5.4.1 network simulator [25]. In order to accurately measure PISTIS's communication overhead, we configure network links to have a non-limiting 1Gbps throughput, and a communication latency of either 1ms or 5ms. We evaluated PISTIS's performance using two signature schemes of similar security guarantees, and available in the OpenSSL library [26]: RSA-2048 (i.e., 256 bytes long signatures) and elliptic curves (EC) with prime256v1 curves (i.e., 71 bytes long signatures). We use broadcast messages of sizes equal to 1 byte and 1Kbits.

We consider the probability of losing/omitting a message sent at any point in time to be $p \in \{0.i \mid 0 \leq i \leq 9\}$. We run our simulations for systems with $N \in \{25, 49, 73\}$ processes, and for several values of X, which is the number of processes each process forwards a message m to during diffusion. For each value of N, we consider various probability values with which a sent message can be lost/omitted.

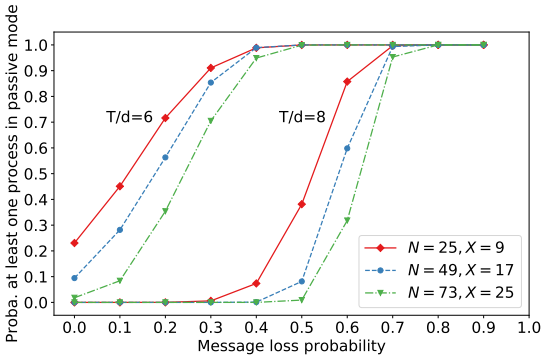

Figure 3. Probability of a correct process entering passive mode when $\mathbb{T} = 6d$ or $\mathbb{T} = 8d$

C. PISTIS's Reliability

To assess PISTIS's reliability, we evaluate the probability that a correct process enters the passive mode. Such a predicate is crucial, as it may render PISTIS unresponsive. For example, when $N = 3f+1$, a single correct process staying passive for long-enough can, in the worst case (when f Byzantine processes are not sending messages), leave $2f$ correct processes, which would not be enough to gather quorums of size $2f+1$, leading those $2f$ processes to become passive.

For a given value of N and p, we invoke a broadcast at one of the processes and record any non-Byzantine process that crashed itself during broadcast. We obtain our results by repeating each experiment 10^5 times, and we report the probability that a process crashes itself as:

$$(\text{num. of experiments with self-crashed processes})/10^5$$

We study the impact of several parameters, including $\mathbb{T}$, N, X, f, and p, on PISTIS's reliability, and determine which values should be used to enforce an intended system reliability.

Figure 3 shows that the system's reliability increases with its size and $\mathbb{T}$'s value. For example, when $\mathbb{T} = 8d$, a system with 25 (resp. 49) processes operates with high reliability (i.e., there is a negligible probability that a process becomes passive) under message loss rates reaching up to 30% (resp. 40%).

Figure 4 shows that the actual number of Byzantine processes, which varies between 0 and f (the maximum number of tolerable Byzantine nodes), influences the system's resiliency. As one could expect, with fewer processes being Byzantine, higher message loss rates are tolerated without any process shutdown.

Impact of the diffusion fanout. In the results presented so far, processes forward each message to $X = f+1$ other random processes. We now study the effect of X by measuring PISTIS's reliability when it varies. Figure 5 shows that increasing X helps increase the overall system reliability. As expected increasing the fanout (value of X) reduces the probability of having a non-Byzantine node becoming passive.

Recovery. Figure 6 details the probability that no Byzantine quorum remains active after a broadcast instance when the message loss probability increases. First, one can observe that the recovery mechanisms improve the resiliency of the system. For example, with $N = 49$, PISTIS can tolerate a

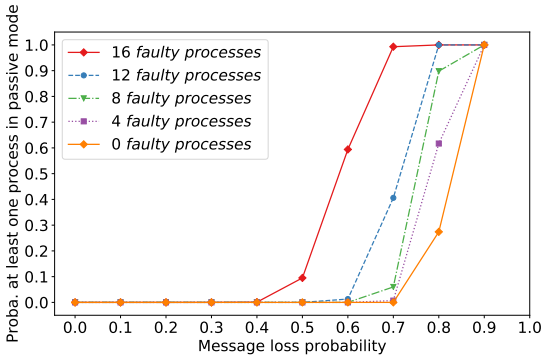

Figure 4. Probability of a correct process entering passive mode in a system of 49 processes (i.e., $f = 16$) using $\mathbb{T} = 8d$ and $X = 17$, when 0, 4, 8, 12 or 16 processes are faulty

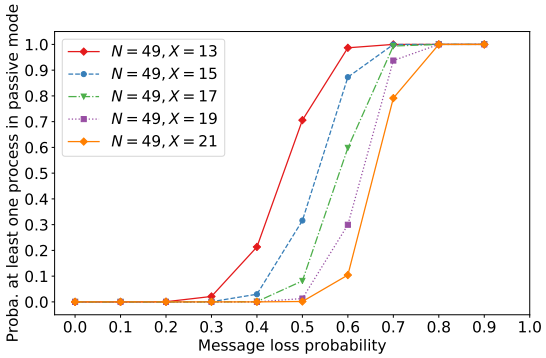

Figure 5. Probability of a correct process entering passive mode in a system of 49 processes using $\mathbb{T} = 8d$, and where X varies

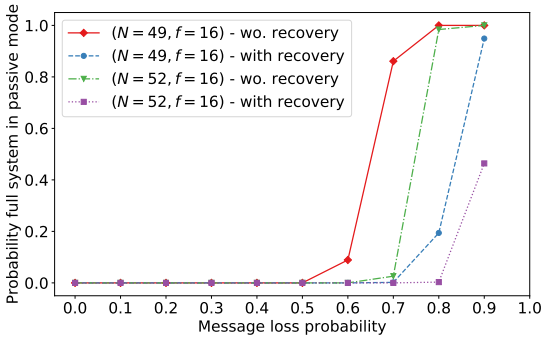

Figure 6. Probability that no Byzantine quorum remains active in systems of 49 or 52 processes, when $\mathbb{T} = 8d$, $X = 17$ or 18 respectively, and $f = 16$ processes are Byzantine.

70% message loss rate without system-wide crashes thanks to the recovery mechanisms, improving over the value of 50% obtained without recovery. Second, we show that one can further improve the system's tolerance to message losses by overprovisioning the system. By using three more nodes, i.e., 52 in total, the system can tolerate $f = 16$ Byzantine nodes and now tolerate up to 80% of message losses.

D. PISTIS latency and bandwidth consumption

Next, we evaluate PISTIS's overhead, precisely quantifying the corresponding incurred bandwidth and latency. For these experiments, we average results over 100 runs. We consider two possible network delays between pairs of nodes, respectively 1ms and 5ms, and two broadcast message sizes, 1 byte and 1 Kbits. We use $\mathbb{T} = 8d$, since our reliability results show it allows a very large number of message losses to be tolerated. However, we now run our experiments without

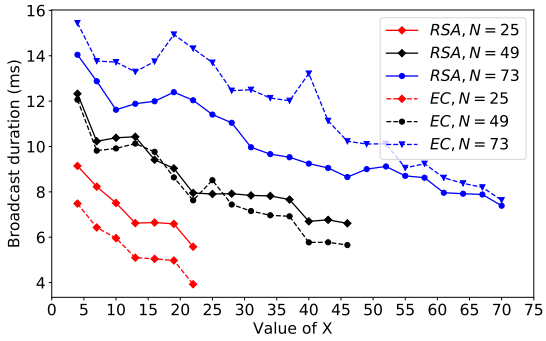

Figure 7. Average latency with a 1ms link latency with $\mathbb{T} = 8d$ and without message losses

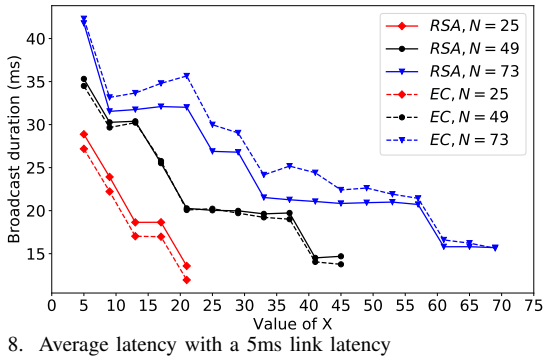

Figure 8. Average latency with a 5ms link latency

any message losses to measure the worst case bandwidth consumption. We measure both the protocol latency and bandwidth consumption depending on the value of X that the processes use.

Latency. Figures 7 and 8 detail the latency for a broadcast message to be delivered by all correct processes in systems of size 25, 49, and 73, using either RSA or EC: PISTIS delivers latencies within $[3\text{ms}, 45\text{ms}]$ depending on the network delay d. The latency increases when N increases, and decreases when X increases. Note that due to their signature and verification costs, EC is better suited for small systems (≤ 49 nodes), while RSA is better suited for larger systems.

PISTIS's latency vs. related systems' latency. Moreover, we provide readers with a latency comparison between PISTIS and related works on real-time Byzantine broadcasts demonstrating that PISTIS has a superior performance. The comparison is done based on the worst case delay (a direct experimental evaluation would not be fair, since not all previous work [8] consider probabilistic synchronous networks). Let us first refine the definition of d introduced in Sec. III-A. Let d_n be the maximum network delay, and d_p be the maximum local processing time, which includes the cryptographic operations overhead, such that d can be decomposed as $d_p + d_n$. Christian et al. [8] compute the worst case delay as $10 * (f + 2) * (n - 1) * d_n$ where f is the maximum number of faulty processes, n the total number of processes, and d_n the network delay. In this work, d_p is equal to 10. Kozhaya et. al [21] compute the worst-case delay as $3 * R * d$, where R is the number of consecutive synchronous communication rounds the same message gets disseminated (time-triggered retransmissions). PISTIS's worst case delay is proved to be $3 * \mathbb{T}$.

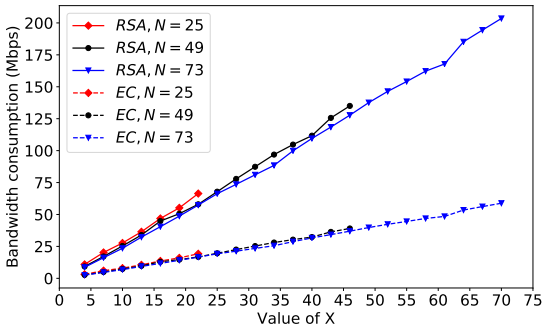

Figure 9. Average bandwidth consumption per node and per communication link with a 1ms link latency without message losses

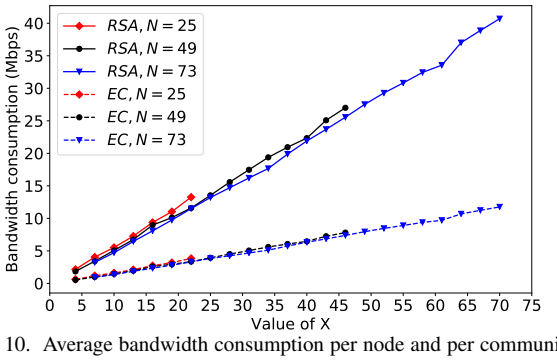

Figure 10. Average bandwidth consumption per node and per communication link with a 5ms link latency without message losses

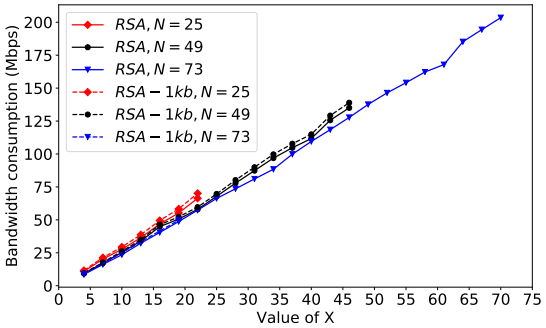

Figure 11. Average bandwidth consumption per node and per communication link with a 1ms link latency using either 1 B or 1 KB messages, without message losses

To ensure fairness and consistency with latency experiments earlier in this section, we set $R = 8$ and $\mathbb{T} = 8d$. However, due to PISTIS's signature management (see, for example, the optimizations described in Sec. VI-A), PISTIS's worst case delay can be alternatively computed as $(3{*}8{*}d_n){+}(2{*}N{*}d_p)$. This is in part due to the fact that in PISTIS nodes avoid re-verifying signatures that they have already verified.

Table I shows the worst case latencies of all algorithms for $d_n = 1ms$ (note that, as mentioned above, in the first column $d_p = 10$, while in the last two columns d_p is such that $1 < d_p < 10$, and can be derived from the numbers provided in the table).

	[8]	[21]	PISTIS
$N = 25,\ f = 8$	2400 ms	26 ms	25.6 ms
$N = 50,\ f = 16$	8640 ms	70 ms	27 ms
$N = 100,\ f = 33$	34650 ms	150 ms	30 ms

Table I
WORST CASE LATENCIES

Let us make two main observations. First, compared to the other protocols, PISTIS has superior performance, which is attributed to the fact that PISTIS is event triggered, utilizes fast signature schemes and implements some optimizations to reduce the number of signatures created and verified, induces less network congestion (which increase individual message failures) and allows processes suffering timing failures for fast detection of their tardiness. Second, PISTIS's expected performance in practice (see Figure 7) is significantly better than the worst case delay bound reported in the table.

Network bandwidth consumption. We now measure PISTIS's bandwidth overhead per broadcast invocation, using RSA and EC signatures. Figures 9 and 10 present the bandwidth consumption for 1 byte payloads with 1ms and 5ms link delay, respectively. One can observe that using the gossip-like approach with $X = f + 1$, the bandwidth is 8 (resp. 5) times lower than when using RSA (resp. EC) as opposed to using all-to-all communication, i.e., $X = N$. We also showcase in Figure 11 that the bandwidth consumption very reasonably increase when the message payload is increased to 1Kbits.

VII. CONCLUSION

In this paper, we studied how to build large-scale distributed protocols that tolerate network faults and attacks while providing real-time communication. We introduced a suite of proven correct algorithms, starting from a baseline real-time Byzantine reliable broadcast algorithm, called PISTIS, all the way up to real-time Byzantine atomic broadcast and consensus algorithms. PISTIS is empirically shown to be robust, scalable, and capable of meeting timing deadlines of real CPS applications. PISTIS withstands message loss (and delay) rates up to 40% in systems with 49 nodes and provides bounded delivery latencies in the order of a few milliseconds. PISTIS improves over the state-of-the-art in scalability and latency through its event-triggered nature, gossip-based communications, and fast signature verifications. Our work simplifies the construction of powerful distributed and decentralized monitoring and control applications of various CPS domains, including state-machine replication for fault and intrusion tolerance.

REFERENCES

[1] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. "On implementing omega in systems with weak reliability and synchrony assumptions". In: *Distributed Computing* 21.4 (2008), pp. 285–314.

[2] Carlos Almeida and Paulo Verissimo. "Using Light-Weight Groups to Handle Timing Failures in Quasi-Synchronous Systems". In: *RTSS*. 1998, pp. 430–439.

[3] Yair Amir, Brian A. Coan, Jonathan Kirsch, and John Lane. "Prime: Byzantine Replication under Attack". In: *IEEE Trans. Dependable Sec. Comput.* 8.4 (2011), pp. 564–577.

[4] Jee Hea An, Yevgeniy Dodis, and Tal Rabin. "On the Security of Joint Signature and Encryption". In: *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptology*. EUROCRYPT. Springer-Verlag, 2002, pp. 83–107.

[5] Amy Babay, John L. Schultz, Thomas Tantillo, Samuel Beckley, Eamon Jordan, Kevin Ruddell, Kevin Jordan, and Yair Amir. "Deploying Intrusion-Tolerant SCADA for the Power Grid". In: *DSN*. IEEE, 2019, pp. 328–335.

[6] Amy Babay, Thomas Tantillo, Trevor Aron, Marco Platania, and Yair Amir. "Network-Attack-Resilient Intrusion-Tolerant SCADA for the Power Grid". In: *DSN*. IEEE Computer Society, 2018, pp. 255–266.

[7] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer-Verlag New York, 2011.

[8] Flaviu Cristian, Houtan Aghili, H. Raymond Strong, and Danny Dolev. "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement". In: *Inf. Comput.* 118.1 (1995), pp. 158–179.

[9] GiuseppeAntonio Di Luna, Roberto Baldoni, Silvia Bonomi, and Ioannis Chatzigiannakis. "Conscious and Unconscious Counting on Anonymous Dynamic Networks". In: *ICDCN*. Vol. 8314. 2014.

[10] DLC+VIT4IP. *D1.1 Scenarios and Requirements Specification*. Tech. rep. 2010.

[11] D. Dolev and H. R. Strong. "Authenticated Algorithms for Byzantine Agreement". In: *SIAM J. Comput.* 12.4 (1983), pp. 656–666.

[12] Danny Dolev. *The Byzantine Generals Strike Again*. Tech. rep. Stanford University, CA, USA, 1981.

[13] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. "On the Minimal Synchronism Needed for Distributed Consensus". In: *JACM* 34.1 (1987).

[14] Danny Dolev and Rüdiger Reischuk. "Bounds on Information Exchange for Byzantine Agreement". In: *JACM* 32.1 (1985), pp. 191–204.

[15] Dacfey Dzung, Rachid Guerraoui, David Kozhaya, and Yvonne-Anne Pignolet. "To Transmit Now Or Not To Transmit Now". In: *SRDS*. 2015, pp. 246–255.

[16] Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. "Easy Impossibility Proofs for Distributed Consensus Problems". In: *Distributed Computing* 1.1 (1986), pp. 26–39.

[17] R. Guerraoui, D. Kozhaya, and Y. A. Pignolet. "Right on Time Distributed Shared Memory". In: *RTSS*. 2016, pp. 315–326.

[18] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. "Scalable Byzantine Reliable Broadcast". In: *DISC*. Vol. 146. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 22:1–22:16.

[19] Romain Jacob, Marco Zimmerling, Pengcheng Huang, Jan Beutel, and Lothar Thiele. "End-to-end Real-time Guarantees in Wireless Cyber-physical Systems". In: *RTSS*. 2016, pp. 167–178.

[20] F. Januário, C. Carvalho, A. Cardoso, and P. Gil. "Security challenges in SCADA systems over Wireless Sensor and Actuator Networks". In: *ICUMT*. 2016, pp. 363–368.

[21] D. Kozhaya, J. Decouchant, and P. Esteves-Verissimo. "RT-ByzCast: Byzantine-Resilient Real-Time Reliable Broadcast". In: *IEEE Trans. Comput.* 68.3 (2019), pp. 440–454.

[22] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems* 4/3 (1982), pp. 382–401.

[23] Dahlia Malkhi and Michael K. Reiter. "Byzantine Quorum Systems". In: *STOC*. ACM, 1997, pp. 569–578.

[24] J. R. Moyne and D. M. Tilbury. "The Emergence of Industrial Control Networks for Manufacturing Control, Diagnostics, and Safety Data". In: *Proceedings of the IEEE* 95.1 (2007), pp. 29–47.

[25] *OMNeT++*. Last accessed: Feb 24, 2020. URL: https://omnetpp.org.

[26] *OpenSSL*. Last accessed: Feb 25, 2020. URL: https://www.openssl.org/.

[27] M. M. Patel and A. Aggarwal. "Security attacks in wireless sensor networks: A survey". In: *ISSP*. 2013, pp. 329–333.

[28] M. Pease, R. Shostak, and L. Lamport. "Reaching Agreement in the Presence of Faults". In: *JACM* 27.2 (1980), pp. 228–234.

[29] Pavel Polityuk, Oleg Vukmanovic, and Stephen Jewkes. *Ukraine's power outage was a cyber attack: Ukrenergo*. 2017. URL: https://www.reuters.com/article/us-ukraine-cyber-attack-energy/ukraines-power-outage-was-a-cyber-attack-ukrenergo-idUSKBN1521BA.

[30] L. Schenato, B. Sinopoli, M. Franceschetti, K. Poolla, and S. S. Sastry. "Foundations of Control and Estimation Over Lossy Networks". In: *Proceedings of the IEEE* 95.1 (2007), pp. 163–187.

[31] P. Verissimo, L. Rodrigues, and M. Baptista. "AMp: A Highly Parallel Atomic Multicast Protocol". In: *SIGCOMM*. 1989, pp. 83–93.

[32] Paulo Verissimo and António Casimiro. "The Timely Computing Base Model and Architecture". In: *IEEE Trans. Comput.* 51.8 (2002), pp. 916–930.

[33] S. Viswanathan, R. Tan, and D. K. Y. Yau. "Exploiting Power Grid for Accurate and Secure Clock Synchronization in Industrial IoT". In: *RTSS*. 2016, pp. 146–156.

David Kozhaya is a Scientist at ABB Corporate Research, Switzerland. He received his PhD degree in Computer Science in 2016, from EPFL, Switzerland, where he was granted a fellowship from doctoral school. His primary research interests include reliable distributed computing, real-time distributed systems, and fault- and intrusion-tolerant distributed algorithms. His past work experiences span across interdisciplinary domains ranging from research, teaching programming languages and computer literacy.

Jérémie Decouchant is a Research Scientist at SnT, University of Luxembourg. He received his Ph.D in Computer Science in 2015 from the University of Grenoble-Alpes, France, and an engineering degree (MSc) in 2012 from the Ensimag engineering school, in Grenoble. His research focuses on the design of resilient and privacy-preserving distributed systems.

Vincent Rahli is a Senior Lecturer at the University of Birmingham. He received his Ph.D in Computer Science from Heriot-Watt University, UK. His research focuses on designing, formalizing, and using type theories and on the verification of distributed systems using proof assistants.

Paulo Esteves-Verissimo Paulo Esteves-Verssimo is a professor at the Univ. of Luxembourg FSTM / SnT, and Head of the CritiX lab (https://wwwen.uni.lu/snt/research/critix). He represents UNILU in ECSO. He was Chair of the IFIP WG 10.4 on Dependable Computing and Fault-Tolerance and vice-Chair of the Steering Committee of the IEEE/IFIP DSN conference. He is Fellow of the IEEE and Fellow of the ACM, and associate editor of the IEEE TETC journal, author of over 200 peer-refereed publications and co-author of 5 books. He is interested in architectures, middleware and algorithms for resilient modular and distributed computing, such as: SDN-based infrastructures; autonomous vehicles from earth to space; digital health and genomics; or blockchain and cryptocurrencies.

APPENDIX A
DIFFERENCES BETWEEN PROBABILISTIC SYNCHRONY AND OTHER STANDARD MODELS

a) Comparison with fully asynchronous models: Our model is more informative than traditional fully asynchronous models. More precisely, asynchronous models do not make any assumptions regarding message transmission and processing delays, while we assume that messages are delivered within a maximum transmission delay d with high probability.

b) Comparison with synchronous models: Our communication model is a probabilistic synchronous one. We recall that in every transmission attempt a link may (with some probability) violate reliability and timeliness by dropping the message or delivering it within a delay $> d$. In case of message loss (omission) a sender that needs to re-transmit that message again faces yet another risk of transmission failure. Due to omissions (losses in consecutive transmission attempts) and the required follow-up re-transmissions, the time it takes to send a message reliably from one process to another (measured from the time of the first transmission attempt) may be unbounded. So, despite links being reliable and timely with high probability, our communication system is no longer synchronous.

c) Comparison with partially synchronous models: In comparison with partial synchrony [2], which assumes that communication becomes forever synchronous after some unknown point in time, our probabilistic synchronous model guarantees only finite synchronous periods (with variable durations) that may occur randomly during the lifetime of the system. In fact such probabilistic synchronous communication has been shown to be weaker, in some sense [3], than partial synchrony. For example, while the celebrated failure detectors of [1] can be implemented in partially synchronous systems they are impossible to implement in the systems with probabilistic synchronous communication [3].

d) The need for probabilistic synchrony models: Probabilistic synchronous models (such as the one presented here or [5, 3]) are more "realistic" than synchronous models in the sense that timing assumptions cannot always be ensured in distributed systems because, for example, of the difficulty of guaranteeing reliable communication between the nodes of a system. Making the probability of timing failures (e.g., that messages might be delivered after d) transparent to the model and protocols makes them more robust. For example, it allows designing protocols where messages might not always arrive within a specified maximum transmission delay. Systems that require processes to operate in a timely fashion, such as mission critical systems, can therefore dynamically adapt to such untimely situations to ensure that timing guarantees are fulfilled.

e) Comparison with quasi-synchronous models: Quasi-synchronous models [5] address the timing issues mentioned above. In [5] synchronism is characterized by the following properties: P1—processing speeds are bounded and known; P2—message delivery delays are bounded and known; P3—local clock rate drifts are bounded and known; P4—load patterns are bounded and known; and P5—differences among local clocks are bounded and known. A system is quasi-synchronous if it satisfies properties P1–P5, and at least one of those does not hold with some known non-zero probability. As in a quasi-synchronous model, in our probabilistic model P2 only holds with high probability. Note, however, that in our probabilistic model we do not assume that differences among local clocks are bounded and known.

APPENDIX B
CORRECTNESS OF PISTIS (ALGORITHM 2)

Lemma 1 (Validity)**.** *If a correct process p_i broadcasts m then p_i eventually delivers m.*

Proof outline. Because p_i is correct, it will hear echoes of m from $2f + 1$ processes (including p_i) by $t + \mathbb{T}$, where t is the time p_i broadcasted m. This is true as otherwise, i.e., if less than $2f + 1$ echoes for m are heard, p_i would kill itself (hence is no longer correct). Indeed, p_i triggered a timer (see line 49 of Algorithm 2) when it started broadcasting m (see line 6). Because p_i received $2f+1$ echoes for m, it must have delivered m too (see lines 14, 19, and 26 of Algorithm 2). □

Lemma 2 (No duplication)**.** *No correct process delivers message m more than once.*

Proof outline. According to line 60 of Algorithm 2 a process only delivers a message if the corresponding $\mathcal{R}_{deliver}$ does not exist, and creates one right after delivering, thereby preventing from delivering a message twice. □

Lemma 3 (Integrity)**.** *If some correct process p_j delivers a message m with correct sender p_i, then m was previously broadcasted by p_i.*

Proof outline. Because p_j delivered m, it must have received $2f + 1$ signed echoes for m (see lines 14, 19, 26, and 33 of Algorithm 2). As mentioned in Remark 1, an echo message is not handled unless it is signed by the claimed sender. More precisely, upon receipt of a message of the form $\texttt{Echo}\left(\langle p_i, sq, v\rangle, \Sigma\right)$ or $\texttt{Deliver}\left(\langle p_i, sq, v, \Sigma\rangle, \Sigma'\right)$, p_j only handle the message if Σ contains a signature from p_i. Now, because the sender p_i is correct, it must have indeed sent an echo message for $\langle p_i, v\rangle$. Finally, we prove by induction on the chain of local events happening at p_i (a correct process) that led to this message being sent, that p_i must have broadcasted it. □

Lemma 4 (Intersecting delivery)**.** *Let p be a correct process that starts delivering some message m at some time t_d. Then, there exists a collection B of $2f + 1$ processes such that all correct processes in B only deliver m for a full $\mathbb{T}$ duration starting some time prior to $t_d + \mathbb{T}$.*

Proof outline. Let us first point out that because p starts delivering at t_d, and because it is correct, $2f + 1$ processes must have received this deliver message by $t_d+\mathbb{T}$ (otherwise p would kill itself because it wouldn't be connected—the proof-of-connectivity is executed in piggyback mode). Let A be this collection of $2f + 1$ processes (note that $p \in A$). For each correct process $q \in A$, q must have started delivering some time prior to $t_d + \mathbb{T}$.

Let us now prove this lemma by induction on t_d.

Either a correct process within A started delivering prior to t_d or not. If one did, in which case $t_d > 0$, then we conclude by our induction hypothesis. Otherwise all correct nodes in A (at least $f + 1$) are only delivering starting from t_d. Because they start delivering prior to $t_d + \mathbb{T}$, and because they deliver for

$2\mathbb{T}$, it must be that all correct processes within that collection only deliver m for a full $\mathbb{T}$ duration starting at most by $t_d + \mathbb{T}$ (until at most $t_d + 2\mathbb{T}$). □

Lemma 5 (Timely agreement)**.** *If a correct process p_i broadcasts m at real time t, then all correct processes deliver m by $t + 3\mathbb{T}$.*

Proof outline. Since p_i is correct during this broadcast, then it must have received $2f + 1$ echoes for m and must then have started delivering m at $t_d \in [t, t + \mathbb{T}]$. By Lemma 4, there exists a collection B of $2f + 1$ processes such that all correct processes in B only deliver m for a full $\mathbb{T}$ duration starting some time prior to $t_d + \mathbb{T}$. Now, every other correct process p_j must be connected to $2f+1$ processes in any proof-of-connectivity period $pc = [t_0, t_0 + \mathbb{T}]$—let $C(pc)$ denote those $2f + 1$ processes. Therefore, because there are $3f + 1$ processes, there must be a correct process, say r, and a proof-of-connectivity period $pc = [t_j, t_j + \mathbb{T}]$ at p_j such that: (1) r is in the intersection of B and $C(pc)$ (there must be at least one correct process in that intersection because it is of size $f + 1$); and such that (2) p_j received m during pc from r, which sent it at most by $t_d + 2\mathbb{T}$. Therefore, p_j must have delivered by $t + 3\mathbb{T}$. □

Lemma 6 (Agreement)**.** *If some correct process p_i delivers m, then all correct processes eventually deliver m.*

Proof outline. This is a straightforward consequence of Lemma 5.

□

Lemma 7 (Timeliness)**.** *If a correct process p_i broadcasts m at real time t, then no correct process delivers m after $t + 3\mathbb{T}$.*

Proof outline. This is a straightforward consequence of Lemma 5. □

Appendix C
Proof of Theorem 2

Recall that since Algorithm $\mathcal{A}$ implements interactive consistency, then when $\mathcal{A}$ eventually terminates all correct processes will have the same vector of proposals where the values relative to correct processes are indeed what these correct processes have proposed. In fact interactive consistency [4] guarantees the two following properties:

- IC.1 The non-faulty processors compute exactly the same vector.
- IC.2 The element of this vector corresponding to a given non-faulty processor is the private value of that processor.

We now prove Thm. 2, i.e., that assuming algorithm $\mathcal{A}$ implements interactive consistency in a known bounded number of communication rounds (this is used to prove Lemma 11), as well as Assumptions 1 and 2, then $\mathcal{A}$ implements RTBC (see Sec. V-A) in our system model (see Sec. III).

Lemma 8 (RTBC-Termination)**.** *Every correct process eventually decides.*

Proof outline. By Assumption 1, a correct process p_i accesses the network only through the RTBRB primitive. Therefore, because p_i is correct and therefore does not enter passive mode while executing RTBRB, it must terminate. By IC.1 p_i must compute a vector. Finally, p_i will apply the deterministic function described in Assumption 2 to that vector to obtain a value v, which is the value p_i decides upon.

□

Lemma 9 (RTBC-Agreement)**.** *No two correct processes decide differently.*

Proof outline. Let p_i be a correct process that decides upon a value v_i, and p_j be a correct process that decides upon a value v_j. Again, by Assumption 1, p_i and p_j must not enter passive mode while using the RTBRB primitive. By IC.1, p_i and p_j must compute the same vector V. Both p_i and p_j apply the deterministic function described in Assumption 2 to this vector V. Therefore, v_i must be equal to v_j. □

Lemma 10 (RTBC-Validity)**.** *If all correct processes propose the same value v, then any correct process that decides, decides v. Otherwise, a correct process may only decide a value that was proposed by some correct process or the special value $\perp$.*

Proof outline. First, note that by Assumption 1, correct processes must not enter passive mode while using the RTBRB primitive. Now, if all correct processes propose the same value v, then by IC.2 the obtained interactive consistency vector computed by a correct process should contain v a number of times equal to the number of correct processes, i.e., at least $2f + 1$ times. Finally, since all correct processes apply the deterministic function described in Assumption 2 to their vectors, they must all decide on v.

Let us now assume that not all correct processes propose the same value v. If a correct process p decides upon a value v' then by Assumption 2, it must be that either (1) its interactive consistency vector contains at least $2f + 1$ times this value v'; or (2) that v' is the special value $\perp$. In case v' appears $2f + 1$ times in p's interactive consistency vector, then by IC.2, it must be that v' was proposed by a correct process. This concludes the proof. □

Lemma 11 (RTBC-Timeliness)**.** *If a correct process p_i proposes a value to consensus at time t, then no correct process decides after $t + \Delta_{\mathtt{C}}$.*

Proof outline. The way we implement consensus is first by reaching interactive consistency and applying a deterministic function after. The deterministic function is a computational load that requires scanning the consistency vector and hence has a known bounded duration since we assume that correct processes are synchronous. Therefore, it is sufficient to prove that the interactive consistency protocol finishes in a bounded duration (in the sense that correct processes compute their interactive consistency vectors in a bounded amount of time).

Recall that we assume that Algorithm $\mathcal{A}$ requires a bounded number of communication rounds to terminate, say k. By Assumption 1 processes send and receive messages over the network only via the RTBRB primitive. Hence any communication round has a bounded duration, that being a multiple, say m, of $\Delta_{\mathtt{R}}$, the duration needed by the RTBRB primitive to complete (which is at most $3\mathbb{T}$). Therefore, because by Assumption 1, correct processes must not enter passive mode while using the RTBRB primitive, it must be that correct processes will decide before $t + (k \times m \times 3\mathbb{T})$, which concludes our proof. □

APPENDIX D
REAL-TIME BYZANTINE-RESILIENT ATOMIC BROADCAST

A. Reduction to RTBAB

To prove Thm. 3, we only have to prove that Algorithm $\mathcal{A}$ satisfies the RTBAB-Timeliness property.

Proof outline. Let us assume that the correct process p_i RTBAB-broadcasts m at time t. We have to prove that no correct process RTBAB-delivers m after real time $t + \Delta_{\mathtt{A}}$, for some $\Delta_{\mathtt{A}}$. We prove this by proving the stronger result that there exists a $\Delta_{\mathtt{A}}$ such that all correct processes RTBAB-deliver m by $t + \Delta_{\mathtt{A}}$.

By Property 1, p_i RTBRB-broadcasts m with some sequence number seq_t by time $t + \Delta_B$. By RTBRB-Validity, RTBRB-Timeliness and RTBRB-Agreement, all correct processes RTBRB-deliver m by some time $t + \Delta_B + \Delta_{\mathtt{R}}$. By Property 2, all correct processes will RTBC-propose or RTBC-decide m by $t + \Delta_B + \Delta_{\mathtt{R}} + \Delta_P$.

If one correct process RTBC-decides m by $t + \Delta_B + \Delta_{\mathtt{R}} + \Delta_P$, then by the RTBC properties, all correct processes will RTBC-decide by $t + \Delta_B + \Delta_{\mathtt{R}} + \Delta_P + \Delta_{\mathtt{C}}$, and by Property 4, they will RTBAB-deliver by $t + \Delta_B + \Delta_{\mathtt{R}} + \Delta_P + \Delta_{\mathtt{C}} + \Delta_D$, which concludes our proof. Therefore, let us now consider the case where they all RTBC-propose m by $t + \Delta_B + \Delta_{\mathtt{R}} + \Delta_P$.

However, it might be that they RTBC-propose m in different RTBC instances. We want to prove that there will be an RTBC instance $inst_m$ where "enough" correct nodes RTBC-propose m at that instance, by time $t + \Delta_m$ (for some fixed Δ_m), so that it results in $inst_m$ deciding m. Then, by RTBC-Termination, RTBC-Agreement, RTBC-Timeliness, and Property 4, we can conclude that all correct processes RTBAB-deliver m by time $t + \Delta_m + \Delta_{\mathtt{C}} + \Delta_D$. Let us now prove that such an instance $inst_m$ indeed exists.

Because all correct processes RTBC-propose m by $t + \Delta_B + \Delta_{\mathtt{R}} + \Delta_P$, there must be a greatest instance $inst_g$ such that a correct process p_g RTBC-proposes m at some time $t_k \leq t + \Delta_B + \Delta_{\mathtt{R}} + \Delta_P$. Now, either (1) m was RTBC-decided at a prior instance $inst_p$ (by all correct processes, by the RTBC properties), or (2) not. In case it was (i.e., case (1)), all correct processes must have RTBC-decided m by time $t + \Delta_B + \Delta_{\mathtt{R}} + \Delta_P + \Delta_{\mathtt{C}}$ by the RTBC properties and because $inst_p$ must have been dealt with by p_g before $inst_g$ by Property 6. Now, by Property 4, it must be that all correct processes must have RTBAB-delivered m by time $t + \Delta_B + \Delta_{\mathtt{R}} + \Delta_P + \Delta_{\mathtt{C}}$.

Let us now focus on case (2), i.e., m was not RTBC-decided at a prior instance. By Property 3, correct processes must be RTBC-proposing either m or $\perp$ at instance $inst_g$. Let us prove that they cannot propose $\perp$, in which case we conclude using RTBC-Validity and Property 4, and $\Delta_{\mathtt{A}}$ is again $t + \Delta_B + \Delta_{\mathtt{R}} + \Delta_P + \Delta_{\mathtt{C}}$. We prove that correct processes cannot propose $\perp$ at instance $inst_g$ by contradiction. Let us assume that some correct process p_j votes for $\perp$ at instance $inst_g$ (therefore, p_j cannot be p_g). By definition of $inst_g$, it must be that p_j votes for m at a prior instance $inst_p$. Because it is an instance prior to $inst_p$, as mentioned above, m was not RTBC-decided at that instance. Therefore, by Property 3, and RTBC-Validity, it must be that this instance ended up in $\perp$ being decided. Finally, we obtain a contradiction from the fact that p_j must also RTBC-propose m at instance $inst_g$, which we prove by induction on the list of instances between $inst_p$ and $inst_g$ and using Property 5. □

B. PISTIS-AT: a Class of Algorithms Implementing RTBAB

Algorithm 3 provides an example of a PISTIS-AT algorithm, which implements the RTBAB primitive presented in Sec. V-B. We assume here that a process broadcasts a message by invoking `RTBAB-broadcast()`, and delivers a message invoking `RTBAB-deliver()`. In addition, `RTBAB-init(rtbab)` instantiates a new instance of RTBAB with id rtbab. To guarantee total order, each process maintains a monotonically increasing sequence number seq, which is incremented every time `RTBAB-broadcast()` is called.

Lemma 12. *Given an RTBAB instance $inst$, such that p_i is the leader of $inst$, all correct processes will either RTBC-propose a value received from p_i or $\perp$ (in case they have not received any new message from p_i since the last one they processed). Moreover, given two correct processes that RTBC-propose such values at instance $inst$, it must be that either those values are equal (to the k^{th} new value broadcasted by p_i, for some k) or one of them is $\perp$ (in case the corresponding process has not received p_i's k^{th} broadcasted new value yet, and has already processed all previous broadcasted value from p_i).*

Proof outline. This can be proved by induction on causal time.

The first time those correct processes RTBC-propose a value at an instance such that p_i is the leader, it must be that either this value is the first value RTBAB-broadcasted by p_i, or $\perp$.

The inductive case goes as follows: we assume that our property is true at a given instance $inst$ such that p_i is the leader, and where correct processes RTBC-propose either v (the $(k-1)^{th}$ new value proposed by p_i) or $\perp$, and we prove that the property is still true at the next such instance $inst'$. By RTBC-Validity, it must be that correct processes either RTBC-decide v or $\perp$, and by RTBC-Agreement, they must not decide differently. Therefore, if they decide v at instance $inst$, then v will be added to the *delivered* set, and therefore never added to *unordered* again; and in addition, it will be removed from *unordered*. At the next instance $inst'$, these processes will vote either for the k^{th} new value proposed by p_i or for $\perp$ if they have not received that k^{th} new value. In particular, if one of those correct processes RTBC-proposed $\perp$ because it had not received v yet, then at instance $inst'$ it will either propose the k^{th} new value proposed by p_i (since v is skipped because already delivered), or $\perp$ in case it has not received this k^{th} new value yet. Otherwise if they decide $\perp$, then the correct processes that voted for v will still vote for v at $inst'$, and those that voted for $\perp$ will either keep on voting for $\perp$ if they still have not received v, or finally receive v and start voting for v. Note that by RTBAB-Agreement, all correct processes must eventually receive v. □

In order to obtain time bounds that do not depend on Algorithm 3's variable, we make the following assumption:

Assumption 3. *Correct processes wait for $\Delta_{\mathtt{R}} + \Delta_{\mathtt{W}} + (n \times (\Delta_{\mathtt{C}} + \Delta_{\mathtt{W}}))$ between two different broadcasts.*

As we will see below, this is the time it takes to guarantee that all correct processes RTBAB-deliver an RTBRB-broadcasted value.

Lemma 13. *Algorithm 3 satisfies Property 1.*

Proof outline. Property 1 holds because Algorithm 3 RTBTB-broadcasts messages on each call to `RTBAB-broadcast` (see lines 5 and 6 of Algorithm 3). □

Lemma 14. *Algorithm 3 satisfies Property 4.*

Proof outline. If a value v (different from $\perp$) is RTBC-decided at time t, and the RTBC instance is the current instance, and v is not in *delivered*, then it is RTBAB-delivered. If v is in *delivered*, then it must be that it was added to that set in the past, in which case it was delivered at that time.

Now, if the RTBC instance is not the current instance, Algorithm 3 retries handling the messages after a while. The number of times a process will retry handling deliver messages is bounded because instances are handled in a monotonic order and are bounded in time according to RTBC-Timeliness. □

Lemma 15. *Under Assumption 3, Algorithm 3 satisfies Property 2.*

Proof outline. First of all, let us point out that RTBRB-deliver messages are treated in monotonic order. Let us now consider three cases. In the following, we first provide variable-dependent bounds, and we then explain how to get independent bounds using Assumption 3.

Case (1): Whenever a process p_i receives a RTBRB-deliver message m at time t with sequence number num, broadcasted by p_j, which is the next one to receive (i.e., $num = next[p_j]$), and if m is not already in *delivered*, then p_i will append m to its $unordered[p_j]$ list. We now have to prove that m will then be RTBC-proposed or RTBC-decided by some time $t + \Delta_P$, for some bounded Δ_P. Because m is now in p_i's $unordered[p_j]$ list, the event line 18 will be triggered at least until m is removed from the list. Because Algorithm 3 uses the rotating coordinator paradigm, then a value broadcasted by some process p_k is voted upon using an RTBC instance only every n (the total number of processes) instances (i.e., whenever p_k is the leader). However, there might be other values before m in the $unordered[p_j]$ lists maintained by the processes. The processes have to RTBC-decide these previous values to start RTBC-proposing m if m has not been RTBAB-delivered in the meantime (otherwise we can conclude because RTBAB-delivered messages are RTBC-decided upon). Because of the rotating coordinator scheme, and by the RTBC properties and Lemma 12, we get the guarantee that m will be RTBC-proposed by $t + (n \times (\Delta_{\mathtt{C}} + \Delta_{\mathtt{W}}) \times (num + 1))$, where $\Delta_{\mathtt{C}} + \Delta_{\mathtt{W}}$ is the time it takes to complete an RTBC instance, and $n \times (\Delta_{\mathtt{C}} + \Delta_{\mathtt{W}})$ is the time it takes to rotate through the leaders ($\Delta_{\mathtt{W}}$ is the time processes wait for before re-trying to handle a message—see line 15 and line 36). Now, thanks to Assumption 3, we can derive that all previous values stored in $unordered[p_j]$ have already been decided upon when correct processes deliver m. Therefore, we get that m will be RTBC-proposed by $t + (n \times (\Delta_{\mathtt{C}} + \Delta_{\mathtt{W}}))$.

Case (2): If m is already in *delivered*, then p_i must have already RTBC-decided m according to lines 28–31.

Case (3): If m is not the next value that p_i is supposed to receive, it will re-try RTBRB-delivering m after $\Delta_{\mathtt{W}}$ until it has received all the previous values. The RTBRB properties guarantee that if some correct process p_j broadcasts a value v at time t, then correct processes will deliver v by $t + \Delta_{\mathtt{R}} + \Delta_{\mathtt{W}}$. Therefore, it must be that correct processes will have stored m (and all previous values) in their $unordered[p_j]$ list by $t + (\Delta_{\mathtt{R}} + \Delta_{\mathtt{W}}) \times (num + 1)$. Finally, following the same argument as above, we get that m will be RTBC-proposed by $t + ((\Delta_{\mathtt{R}} + \Delta_{\mathtt{W}}) \times (num + 1)) + (n \times (\Delta_{\mathtt{C}} + \Delta_{\mathtt{W}}) \times (num + 1))$. As mentioned above, thanks to Assumption 3, we can derive that all previous values stored in $unordered[p_j]$ have already been RTBRB-delivered and RTBC-decided upon when correct processes deliver m. Therefore, we get that m will be RTBC-proposed by $t + ((\Delta_{\mathtt{R}} + \Delta_{\mathtt{W}})) + (n \times (\Delta_{\mathtt{C}} + \Delta_{\mathtt{W}}))$. □

Lemma 16. *Algorithm 3 satisfies Property 3.*

Proof outline. This is a straightforward consequence of Lemma 12. □

Lemma 17. *Algorithm 3 satisfies Property 5.*

Proof outline. Let p_i be a correct process that proposes a value v, with broadcaster p_j, at a given time t, using a given RTBC instance $inst$, and such that this instance does not decide v. By Lemma 12, all correct processes propose v or $\perp$ at that instance. By the RTBC properties, because $inst$ does not decide v, it must decide $\perp$. Therefore, p_i will increment its RTBC instance number but will keep m at the head of its $unordered[p_j]$ list. After a full rotation through the leaders, it will RTBC-propose v again at the later instance $inst + n$, where $0 < n$.

Moreover, no correct process will propose v between $inst$ and $inst + n$ because p_j (v's broadcaster) is the leader of $inst$ and $inst + n$ but not of the instances in between, and v can only be in the $unordered[p_j]$ lists. □

Lemma 18. *Algorithm 3 satisfies Property 6.*

Proof outline. By design, correct processes RTBC-propose exactly one value per RTBC instance because they only start proposing a value in a new instance if *busy* is `False`; in which case they set *busy* to `True`; wait for this instance to complete; and finally increment the RTBC instance number and set back *busy* to `False`.

Correct processes propose values in all RTBC instances and monotonically because they increment the the RTBC instance number by one every time an RTBC instance complete.

Finally, correct processes do not run RTBC instances in parallel thanks to the *busy* flag. □

C. Direct Proof of Algorithm 3's Correctness

Lemma 19 (RTBAB-Validity)**.** *If a correct p_i process broadcasts m, then p_i eventually delivers m.*

Proof outline. By RTBRB-Validity, p_i eventually delivers m with sequence number num. If p_i has already delivered m, i.e., $m \in delivered$, then we are done. Otherwise, because p_i broadcasts messages monotonically (and without gaps), it will append m to its list of unordered messages (line 13 of Algorithm 3). Therefore, line 18 will be triggered until m is removed from the list, as long as p_i eventually resets *busy* to `False` once it has set it to `True`, which is true by RTBC-termination. When finally p_i is the leader of its current instance, say $inst_1$, and that m is at the head of p_i's unordered list, p_i will RTBC-propose m. By RTBC-Validity, either all the correct processes RTBC-propose m, in which case p_i delivers m; or some correct processes RTBC-propose values different from m. As mentioned above, such proposed values must then be $\perp$, in which case p_i might RTBC-decide m or $\perp$. Again as mentioned above, if p_i does not deliver m, it will again either decide m or $\perp$ at the next instance where it is the leader. Because by RTBAB-Agreement, all correct processes eventually receive m, it must be that eventually, p_i RTBC-decides m for an instance where it is the leader, and in turn RTBAB-deliver m. □

Lemma 20 (RTBAB-No duplication)**.** *No message is delivered more than once.*

Proof outline. This property straightforwardly follows trivially from the fact that delivered values are added to the $delivered$ set line 31, and from the fact that a process always checks whether it has delivered a message m before delivering m (see line 30). □

Lemma 21 (RTBAB-Integrity)**.** *If some correct process delivers a message m with initial sender p_i and process p_i is correct, then m was previously broadcast by p_i.*

Proof outline. First of all, the RTBAB-delivered value m (which must be different from $\perp$) with sender p_i (i.e., such that p_i is the leader of the current instance) must have been RTBC-decided upon. By RTBC-Agreement and RTBC-Termination, it must be that the correct sender p_i has also RTBC-decided upon m. It must be that m was it p_i's own $unordered$ list. Therefore, it must be that p_i RTBRB-delivered m. Finally, by RTBRB-Integrity, it must be that p_i previously broadcasted m. □

Lemma 22 (RTBAB-Agreement)**.** *If some message m is delivered by any correct process, then every correct process eventually delivers m.*

Proof outline. Let p_i be the process that RTBAB-delivered m at instance $inst$, such that p_l is the leader of that instance. This delivered value must be different from $\perp$, and must have been RTBC-decided upon. By RTBC-Agreement and RTBC-Termination, it must be that all correct processes eventually RTBC-decide m as well. Let p_j be one such correct process. We have to prove that p_j RTBAB-delivers m also at instance $inst$. By RTBRB-Agreement, it must be that p_j eventually receives the same broadcasts as p_i, among other things, those for which p_l is the leader. From RTBC-Agreement and RTBC-Termination, it must be that all correct processes eventually decide the same values for each RTBC instance. Therefore,

Proof outline. Timeliness follows from RTBRB-Timeliness and RTBC-Timeliness, as well as of the fact that Algorithm 3 rotates through the processes (processes might have to wait a full rotation before they get a chance to decide on a messages that was RTBAB-broadcasted). Let $\Delta_{\mathtt{C}}$ be the time it takes for all correct processes to decide on a value using RTBC (see RTBC-Timeliness). Let $\Delta_{\mathtt{R}}$ be the time it takes for all correct processes to deliver a message using RTBRB (which exists by RTBRB-Timeliness). Assume that p_i assigns the sequence number seq_t with the message m. As mentioned above, we assume that p_i RTBAB-broadcasts m at time t. Because correct processes might still be RTBRB-delivering messages when they gets the RTBRB-deliver message for m, they might not be able to RTBRB-deliver m right away (it might be that $seq_t > next[p_i]$). However, we are guaranteed that all correct processes will have delivered m by time $T_1 = t + ((\Delta_{\mathtt{R}} + \Delta_{\mathtt{W}}) \times (seq_t + 1))$ (where $\Delta_{\mathtt{W}}$ is the time processes wait for before re-trying to handle a message—see line 15 and line 36). Note that at that time, processes might be RTBAB-delivering other messages broadcasted by other processes than p_i. Also, there might already be some messages from p_i to RTBAB-deliver before m (all those with sequence numbers less than seq_t). In case p_i is currently not the leader, it might have to wait a full rotation through the processes to get a chance to be the leader again. Given the fact that all correct processes have m in their $unordered$ list by time T_1, a full rotation will take at most $n \times (\Delta_{\mathtt{C}} + \Delta_{\mathtt{W}})$. Because processes might have to process seq_t messages from p_i before they get a chance to process m, it follows that m will be RTBAB-delivered by $t + ((\Delta_{\mathtt{R}} + \Delta_{\mathtt{W}}) \times (seq_t + 1)) + (n \times (\Delta_{\mathtt{C}} + \Delta_{\mathtt{W}}) \times (seq_t + 1))$. □

Lemma 23 (variable-dependent RTBAB-Timeliness)**.** *There exists a known $\Delta_{\mathtt{A}}$ such that if a correct process p_i broadcasts m at time t, no correct process delivers m after real time $t + \Delta_{\mathtt{A}}$, where $\Delta_{\mathtt{A}}$ depends on seq_t, the current sequence number at the time m is broadcasted.*

Lemma 24 (RTBAB-Timeliness)**.** *Under Assumption 3, there exists a known $\Delta_{\mathtt{A}}$ such that if a correct process p_i broadcasts m at time t, no correct process delivers m after real time $t + \Delta_{\mathtt{A}}$.*

Proof outline. Using Assumption 3 and a proof similar to the one of Lemma 23, we derive that messages RTBAB-broadcasted at time t are RTBAB-delivered by $t + (\Delta_{\mathtt{R}} + \Delta_{\mathtt{W}} + (n \times (\Delta_{\mathtt{C}} + \Delta_{\mathtt{W}})))$. □

Lemma 25 (RTBAB-Total order)**.** *Let m_1 and m_2 be any two messages and suppose that p_i and p_j are any two correct processes that deliver m_1 and m_2. If p_i delivers m_1 before m_2, then p_j delivers m_1 before m_2.*

Proof outline. Because p_i RTBAB-delivers m_1 before m_2, it must have RTBC-decided m_1 at an instance $inst_1$ and m_2 at an instance $inst_2$ such that $inst_1 < inst_2$. By RTBC-Agreement and RTBC-Termination, p_j must also have RTBC-decided m_1 at $inst_1$ and m_2 at $inst_2$. Using a similar argument as in the proof of RTBAB-Agreement, we derive that p_j must then also have RTBAB-delivered m_1 at instance $inst_1$ and m_2 at $inst_2$. □

References

[1] Tushar Deepak Chandra and Sam Toueg. "Unreliable Failure Detectors for Reliable Distributed Systems". In: *JACM* 43.2 (1996), pp. 225–267.

[2] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. "On the Minimal Synchronism Needed for Distributed Consensus". In: *JACM* 34.1 (1987).

[3] Dacfey Dzung, Rachid Guerraoui, David Kozhaya, and Yvonne-Anne Pignolet. "Never Say Never - Probabilistic and Temporal Failure Detectors". In: *IPDPS*. 2016, pp. 679–688.

[4] M. Pease, R. Shostak, and L. Lamport. "Reaching Agreement in the Presence of Faults". In: *JACM* 27.2 (1980), pp. 228–234.

[5] Paulo Verissimo and Carlos Almeida. "Quasi-Synchronism: a step away from the traditional fault-tolerant real-time system models". In: *Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)* 7.4 (1995), pp. 35–39.