Quarts: Quick Agreement for Real-Time Control Systems

Wajeb Saab, Maaz Mohiuddin, Simon Bliudze, Jean-Yves Le Boudec
École Polytechnique Fédérale de Lausanne, Switzerland
{firstname.lastname}@epfl.ch

***Abstract*—Real-time control systems (RTCSs) tolerate delay and crash faults by replicating the controller. Each replica computes and issues setpoints to actuators over a network that might drop or delay messages. Hence, the actuators might receive an inconsistent set of setpoints. Such inconsistency is avoided either by having a single primary replica compute and issue setpoints (in passive replication) or a consensus algorithm select one sending-replica (in active replication). However, due to the impossibility of a perfect failure-detector, passive-replication schemes can have multiple primaries, causing inconsistency, especially in the presence of intermittent delay faults. Furthermore, the impossibility of bounded-latency consensus causes both schemes to have poor real-time performance. We identified three properties of RTCSs that enable active-replication schemes to agree on the measurements before computing, instead of using traditional consensus. As all computing replicas compute with the same state, the resulting setpoints are guaranteed to be consistent. We present the design of Quarts, an agreement solution for active replication that guarantees consistency and bounded latency-overhead. We prove the guarantees and compare the performance of Quarts with existing solutions through simulation. We show that Quarts provides an availability higher than existing solutions, and that the availability improvement is up to 10x with two replicas.**

I. Introduction

We consider real-time control systems (RTCSs) that consist of controllers built from commercial-off-the-shelf (COTS) software and/or hardware components. Such systems include real-time control of electric grids [1], [2] and manufacturing processes [3]. The controller receives labelled (time-stamped or logically sequenced) measurements from sensors. It uses the measurements to compute and issue setpoints, within bounded latency of a few milliseconds, to actuators to complete the control. The measurements and setpoints are sent over a non-ideal network, and hence might experience delays and losses.

COTS-based RTCSs are susceptible to faults [4] such as delay- and crash-faults [5]. Being mission-critical, such systems achieve the desired high-reliability by replicating the controller. Replication increases the availability by enabling the system to continue issuing setpoints when some replicas are faulty. However, replicas might receive different subsets of the measurements as input and, consequently, produce different — possibly inconsistent — setpoints. Such behavior diverges from that of a non-replicated controller and could result in incorrect control of the controlled process. For instance, in an RTCS that balances power in an electric grid by using two actuators, if the controller replicas issue setpoints ($+5$ kW, -5 kW) and ($+4.8$ kW, -4.8 kW), the actuators might implement ($+5$ kW, -4.8 kW) or ($+4.8$ kW, -5 kW).

Achieving consistency among the replicas within bounded latency is impossible, as shown in the CAP theorem [6]. RTCSs circumvent this by using passive replication [7], where only one replica (the primary) issues setpoints. In the event of the failure of the primary, the standbys perform consensus [8] to elect a new primary among themselves. As perfect failure-detection is impossible in non-ideal networks [9], the standbys might elect a new primary when the existing primary is non-faulty, resulting in multiple replicas issuing setpoints and, consequently, possible inconsistency. This is particularly relevant for RTCSs, due to the intermittence of delay faults.

Alternatively, RTCSs that use active replication perform consensus before issuing each setpoint to provide consistency. However, termination of consensus takes unbounded time [10], resulting in low availability of the RTCS. The shortcomings of existing solutions are discussed in Section II.

We identified a class of RTCSs that enables providing consistency and remaining highly available. The controllers of these RTCSs can compute setpoints, despite missing previous setpoint computations, such as the Kalman filter in [11]. Also, the state of the controller used for computation can be known exactly, e.g., the Kalman gain matrix. Lastly, the actuators should be able to handle identical duplicate setpoints.

For these RTCSs, we developed Quarts, an active-replication agreement protocol that guarantees consistency and provides high availability. In Section III, we describe the model of the RTCSs to which Quarts applies. The model encompasses a wide-range of RTCSs [1], [2], [11], the most famous of which use Kalman-filter-based controllers [12], in which the value of computed setpoints depends on the state of the controller in addition to the measurements. Moreover, in contrast to the crash-only fault model considered by classic solutions, we consider a stronger model that consists of delays and crashes, which better suits COTS-based RTCSs [5].

Our key contribution in the paper is the design of Quarts, described in Section IV. To guarantee consistency, Quarts votes on the measurements to be used in computation. To enhance the chances of a successful vote, voting is preceded by a phase of measurement exchange that enables the replicas to have the same measurements. Quarts is also designed to have bounded latency-overhead. The formal guarantees of consistency and latency overhead are presented in Section V.

Another contribution is an extensive performance evaluation of existing agreement protocols, presented in Section VI. We study the effect of several factors, including the number of replicas, network loss probability, and fault probability, on availability, consistency, latency, and messaging cost. By comparing Quarts to three state-of-the-art protocols [7], [13], we find that Quarts provides higher availability and maintains 100% consistency. This comes at a small expense in messaging cost. Also, addition of replicas improves availability with Quarts significantly, but only marginally for other protocols.

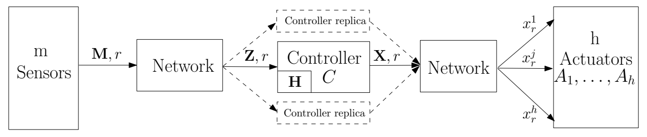

Fig. 1. Architecture of an RTCS

II. Related Work

Passive replication [7] is commonly used in RTCSs as it avoids agreement by having one primary compute and issue setpoints. If the primary is detected as faulty, the standbys elect a new primary by consensus. As failure detection is imperfect in non-synchronous settings [9], the standbys could incorrectly detect the primary as faulty and elect a new primary, and the original primary would not be notified, resulting in two primaries, thus leading to potential inconsistencies. However, these inconsistencies are rare under a crash-only fault model.

Most passive-replication research considers the crash-only fault model, under which passive replication improves availability at a negligible cost of inconsistency. For RTCSs, however, a stronger fault model that includes delay faults is relevant. The intermittent nature of delay faults causes more false positives by failure detectors, hence more inconsistencies.

Alternatively, active-replication schemes such as Fast Paxos [13] and Viewstamped Replication [14] guarantee consistency by using consensus, for deciding the setpoint to be sent or the sender of the setpoint. Consensus consists of 4 properties: agreement, termination, validity, and integrity [8]. Validity and integrity address issues that arise due to Byzantine faults, and are not under consideration in this work. Agreement and termination are impossible to guarantee together, within a bounded delay, in a non-synchronous setting [10]. As RTCSs require low latency, this results in low availability.

Quarts focuses on a class of RTCS for which agreement termination is not always necessary, as they can correctly compute setpoints even if some previous computation was missed (refer to properties in Section III). This enables agreement via voting that, irrespective of the success, ends after a bounded time. Moreover, performing agreement on the input, rather than over the setpoints or the issuing replicas, affords a phase of collection prior to agreement. This increases the chances of a successful vote, thereby improving availability.

We use a composite voter [15] that, in cases of no majority, selects the output of one of the replicas based on a predetermined order. Unlike in [15], where the order is replica-based, we use an ordering based on the number of measurements, a scheme more suited to RTCSs. Also, we use plurality voting [16], instead of majority voting, as it has higher availability.

III. System Model

In this section, we describe the model of an RTCS with a single controller that we will replicate using Quarts. The controller is as shown in Algorithm 1 without the parts in red (lines 11-13, 15); and all replicas are copies of the same controller. The part in red is Quarts, which extends the controller model, and is discussed in detail in Section IV.

Figure 1 shows the architecture of RTCSs that we consider. The m sensors send measurements to the controller, each measurement marked with a logical sequence number r, hereafter referred to as *label*. The labels are global and shared by all the sensors. Such labelling can be obtained either by time-synchronization or through logical clocks [17]. The controller computes and issues h setpoints, one for each actuator. The measurements and setpoints are sent over a non-ideal network that might delay or drop messages. In case of no loss, the propagation delay is bounded by δ_n. The message processing time of non-faulty replicas is negligible w.r.t. δ_n. This network model is called probabilistic synchronous [9].

Algorithm 1: Abstract model of an RTCS controller. Quarts adds the parts in red to guarantee consistency.

```
1  $r \leftarrow 0$;        // highest label of measurements received
2  $r^- \leftarrow 0$;       // highest label of measurements used for computation
3  $\mathbf{Z} \leftarrow \emptyset$;       // vector of measurements with label $r$
4  $\mathbf{H} \leftarrow \emptyset$;      // controller state after computing setpoints with label $r^-$
5
6  repeat        // Thread 1: Receive and aggregate measurements
7      $\mathbf{Z}, r \leftarrow$ aggregate_received_measurements($r$);
8  forever;
9
10 repeat        // Thread 2: Compute and issue setpoints
11     if $r > r^-$ then
12         success, $\mathbf{Z}, \mathbf{H}, r^- \leftarrow$ collect_and_vote($\mathbf{Z}, \mathbf{H}, r, r^-$);
13     end
14     decision $\leftarrow$ ready_to_compute($\mathbf{Z}, \mathbf{H}, r$);
15     if success and decision then
16         $\mathbf{X} \leftarrow$ compute_setpoints($\mathbf{Z}, \mathbf{H}, r - r^-$);
17         $\mathbf{H} \leftarrow$ update_state($\mathbf{Z}, \mathbf{H}, r - r^-$);
18         issue($\mathbf{X}, r$);
19         $r^- \leftarrow r$;
20     end
21 forever;
```

We follow a refined version of the controller model in [5], shown in Algorithm 1. From the received measurements, the controller forms the vector of measurements, $\mathbf{Z}$, corresponding to label r (line 7). This function forms $\mathbf{Z}$ and, when it receives measurements with label greater than r, overwrites r. An example of such a function is the time-alignment function in phasor data concentrators upon receiving time-stamped measurements from phasor measurement units (PMUs) [2].

When sufficient measurements of label r are received or a timeout occurs, the `ready_to_compute` function (line 14) allows the controller to begin computing setpoints. The computations are performed in a strictly increasing order in r.

The computation of setpoints $\mathbf{X}$ corresponding to label r (line 16) depends on measurements of label r ($\mathbf{Z}$), state ($\mathbf{H}$), and correction factor ($r - r^-$). The correction factor indicates when the last setpoint was computed and is used to handle intermittent measurements, as discussed in Property 2. Also, each computation results in a new state (line 17).

Given that the state is used in the computation of setpoints (line 16), Quarts requires replicas to agree on the state. To this end, the following property is expected to hold.

Property 1. *The state used by the controller for computing the setpoints can be known exactly.*

This property is satisfied by a wide range of controllers including controllers using Kalman filters [12] where the state is the gain matrix, or more sophisticated controllers like [18], where the state is the time of the most recent voltage violation.

As the controller is composed of COTS components, it is susceptible to faults. In addition to crash faults, considered within the fault models of classic agreement solutions [7], [19], we include delay faults. The controller is marked *delay faulty* between two consecutive issuings of setpoints if its delay in issuing the former setpoint is greater than a pre-defined value [5]. A crash fault is a special case of delay fault with the controller's delay being infinite.

The intermittent nature of delay faults makes real-time agreement more challenging. As a controller can be delay faulty for an arbitrarily long time, solutions that rely on failure detection will incur a high latency-overhead for each setpoint. In order to tackle the challenges of such intermittent faults, we require the controller to satisfy the following property.

Property 2. *The controller can compute correct setpoints in the presence of intermittent measurements.*

This property holds for controllers that receive measurements and send setpoints via a non-ideal network, e.g., wide-area networks. Specifically, this applies to RTCSs that use a Kalman filter that accounts for intermittent measurements [11].

Controllers of RTCSs that exhibit this property can compute and issue setpoints despite intermittent faults, either in the network (losses of measurements) or in the controller (delay faults). On the contrary, an RTCS that does not exhibit Property 2, will eventually permanently fail to compute and issue setpoints, even in the absence of crash faults. This is because a controller that fails to compute setpoints for a label r, cannot compute setpoints for all labels greater than r.

From this property, we find that a controller might compute setpoints for label $k < r-1$, be delay faulty for some time and then compute setpoints for label r, by using the state corresponding to label k, without the knowledge of the intermediate states. Note that the resulting setpoints would be correct, but might be sub-optimal when compared to those computed using the state corresponding to label $r-1$.

Lastly, as we intend to use active replication, multiple replicas will issue setpoints of the same label to the actuators. Therefore, we require the following property to hold.

Property 3. *Actuators are able to handle duplicate setpoints.*

This property is generally exhibited in RTCSs that use absolute, rather than differential, setpoints. An example of absolute setpoints is an electric-grid controller instructing a battery agent that is injecting 8kW to '*set the injected power to 10kW*', rather than to '*increase the injected power by 2kW*'. Receiving identical duplicates of the former has the same effect as receiving a single setpoint. For RTCSs with differential setpoints, Property 3 can also be achieved by de-duplication using the labels of the setpoints.

IV. Quarts Design

We take a top-down approach to describing the design of Quarts. In Section IV-A, we give a walkthrough of a typical operation of Quarts by using, as an example, an RTCS for control of electric grids that uses a Kalman-filter controller [2]. Then, in Sections IV-B, IV-C, we detail the design of the individual building blocks of Quarts.

Algorithm 2: `collect_and_vote`($\mathbf{Z}$, $\mathbf{H}, r, r^-$)

```
// Part 1: Collection
$\mathbf{Z}_{coll} \leftarrow$ collect_missing_measurements($\mathbf{Z}, r$);
$\mathbf{H}_{coll}, r^-_{coll} \leftarrow$ collect_missing_state($\mathbf{H}, r^-$);
$S \leftarrow$ indices of entries in $\mathbf{Z}_{coll}$;
send digest($S, r^-_{coll}, r$) to all voters;

// Part 2: Voting
success, $S_{chosen}, r^-_{chosen} \leftarrow$ vote($r$);
if success and $r^-_{chosen} = r^-_{coll}$ and $S_{chosen} \subseteq S$ then
    return True, $\mathbf{Z}_{coll}[S_{chosen}], \mathbf{H}_{coll}, r^-_{coll}$;
else
    return False, $\mathbf{Z}_{coll}[S_{chosen}], \mathbf{H}_{coll}, r^-_{coll}$ ;
end
```

A. Protocol Walkthrough

Quarts is applied to an RTCS by replicating the controller, adding the red part (Algorithm 1 lines 11-13, 15), and implementing the `collect_and_vote` function (Algorithm 2) that itself implements Algorithms 3, 4 and 5. In order to guarantee consistency, this function performs agreement between the controller replicas and overwrites the set of measurements $\mathbf{Z}$, the state $\mathbf{H}$, and the label of the last setpoint computation r^-. The function returns False in case this replica should not compute for this label. The `collect_and_vote` function has two parts, Collection and Voting, described in Section IV-B and IV-C, respectively.

The novelty of Quarts is to perform agreement on measurements $\mathbf{Z}$ and the state $\mathbf{H}$ before computation, as opposed to agreement on setpoints done by existing solutions. By agreement on $\mathbf{H}$ for label r, replicas explicitly agree on the correction factor, $r - r^-$. Our choice stems from the observation that performing agreement on the input allows for an optimization that increases the probability of a successful agreement. This optimization is referred to as the collection phase, and involves replicas exchanging measurements and state so as to minimize the missing information in each replica.

Agreement is done in the subsequent voting phase, where replicas exchange a digest of the measurements and the state they have. The subset of measurements corresponding to the most-common digest is chosen for computation. The details of the digest and the voting phase are explained in Section IV-C.

From Algorithm 1, note that the setpoints returned by the `compute_setpoints` function are uniquely determined by $\mathbf{Z}$, $\mathbf{H}$ and $r - r^-$. An example of such a controller is [2], which uses a Kalman filter for estimating the state of an electric grid, and uses the estimated state to compute and issue setpoints to actuators to keep the grid in a feasible state. Here, $\mathbf{Z}$ is composed of phasors received from PMUs, and $\mathbf{H}$ is the Kalman-gain matrix. Recall that in a Kalman filter [11], the measurements, Kalman-gain, and correction-factor uniquely determine the output. Hence, agreeing on $\mathbf{Z}$, $\mathbf{H}$, and r^- for a given r is sufficient for agreeing on the value of the setpoints.

With Quarts, the aforementioned RTCS [2] works as follows. Every 20 ms, the sensors (PMUs) send measurements, with timestamps that are used as labels for identification. When the first measurement with a new label r is received, the replica sets a timeout by which it expects to receive all measurements. One possible value of the timeout is the one-way delay of the network (δ_n). As the measurements are sent

Algorithm 3: `collect_missing_measurements`($\mathbf{Z}$, r)

```
S ← indices of entries in Z;
send Query<$\bar{S}$, $r$> to all replicas; // Ask for missing indices
repeat
    if Query<$Q$, $l$> received and $l = r$ then
        // Received query, send response
        send Response<$\mathbf{Z}[Q \cap S]$, $r$> to all replicas;
    else if Response<$\mathbf{P}$, $l$> received and $l = r$ then
        // Received response, update set of measurements
        Update $\mathbf{Z}$ to include $\mathbf{P}$;
        Add received entries to $S$;
    end
until timer $2\delta_n$ expires;

return $\mathbf{Z}$; // Return set of measurements after collection
```

Algorithm 4: `collect_missing_state`($\mathbf{H}$, r^-)

```
send Advertisement<$r^-$> to all replicas; // Advertise state label
repeat
    if Advertisement<$l$> received and $l < r^-$ then
        // Received advertisement with smaller label, send my state
        send Update<$\mathbf{H}$, $r^-$> to all replicas;
    else if Update<$\mathbf{H}'$, $l$> received and $l > r^-$ then
        // Received state with higher label, update my state
        $\mathbf{H} \leftarrow \mathbf{H}'$;
        $r^- \leftarrow l$;
    end
until timer $2\delta_n$ expires;

return $\mathbf{H}$, $r^-$; // Return state and label after collection
```

over a lossy network, the controller replicas receive different subsets of measurements. On timeout, each replica queries other replicas for the missing measurements. Moreover, each replica that has a state with a label less than $r - 1$ requests others for their most recent state. This way, more replicas have a similar state and set of measurements after collection, and the subsequent voting phase is more likely to succeed.

The collection phase in Algorithm 2 is realized through the functions `collect_missing_measurements` and `collect_missing_state` (Algorithms 3, 4). Each of these functions lasts for at most $2\delta_n$ at each replica, as detailed in Section IV-B. These functions are independent and can run simultaneously, resulting in a bounded latency of $2\delta_n$ for the collection phase.

At the end of the collection phase, each replica sends to other replicas a digest of the measurements it has obtained thus far. This begins the voting phase, in which each replica chooses, using the `vote` function (Algorithm 5), the set of measurements and the state to be used for computing setpoints corresponding to this label. If voting is unsuccessful, or if the corresponding replica does not possess the chosen measurements or state, then False is returned (Algorithm 2, line 12). This prohibits the replica from computing setpoints (Algorithm 1, line 15). The voting phase has a bounded duration of $3\delta_n$ and is explained further in Section IV-C.

Note that, the voting phase requires an upper bound on the number of controller replicas (faulty and non-faulty). This can be achieved either by statically configuring the number of replicas every time a new replica is added or removed, or by using a group membership algorithm such as [20]. As addition of replicas is not done on the real-time path, the introduction of a new replica can wait until the termination of the group membership algorithm. During this time, the RTCS continues to provide consistency but with reduced availability due to a conservative count of the number of replicas.

B. Collection Phase

The collection phase consists of replicas exchanging measurements and state, as presented in Algorithms 3 and 4, respectively. As mentioned earlier, these can run simultaneously. Collection, designed with delay faults in mind, is terminated after $2\delta_n$. Thus, non-delay-faulty replicas can proceed with the voting phase without waiting for delay-faulty ones.

The main premise of collection is that measurements and state with the same label have the same value (Theorem V.1). Without agreeing on the input, this need not necessarily hold for setpoints, as replicas might use different measurements or state for computation. Hence, collection can be performed only on measurements and state, and not on setpoints.

1) Collecting Measurements: Each replica calls `collect_missing_measurements`($\mathbf{Z}, r$) (Algorithm 3). It forms S, the set of IDs of the sensors from which it has received measurements with label r. Then, it sends a *Query* to all replicas asking for the missing measurements (entries of $\bar{S}$).

For each *Query* received, the replica sends a *Response* containing the queried measurements that it has. Also, for each *Response* received, the replica updates the vector $\mathbf{Z}$ to include the newly received measurements. These exchanges include the label r to ensure that stale measurements, caused by delay-faulty replicas sending queries or responses, are ignored.

As this phase is done simultaneously by all non-delay-faulty replicas, its delay is upper-bounded by $2\delta_n$, the time required for queries and responses to be delivered.

2) Collecting State: Similarly, each replica calls `collect_missing_state`, passing it as input the current state ($\mathbf{H}$), and the label (r^-) of the measurements involved in the computation leading up to that state, henceforth referred to as the state label. The replica sends an *Advertisement* with the label of its state r^-.

If a replica receives an *Advertisement* with a label lower than its own, it sends an *Update* with its state and label to all replicas, so that they can synchronize their state accordingly. If a replica receives an *Update* with a label higher than its own, it changes its state and label.

C. Voting Phase

The collection phase ends with each replica sending its *digest* to all the voters. A digest with a label l consists of (1) an indicator set S of measurements with label l that this replica has, and (2) the state label r^- at this replica. Quarts uses a total order among digests. Consequently, we have a function max that returns the largest digest, based on the total order. For each label, there exists a largest possible digest (`full_digest`) that does not contain any missing measurements and that has $r^- = l - 1$. A possible implementation of the digest is a concatenation of the state label with the bitmask of the measurements. For instance, computing setpoints with label $l = 25$: if r^- is 24 and a replica has received measurements from sensors 1, 2, and 5 ($m = 5$), then

Algorithm 5: `vote(r)`

```
1  v ← []; // vector of digests from each replica
2  $\mathbf{S}_{mc}$ ← set of most common digests in v;
3  $\mathbf{S}_{sec}$ ← set of second most common digests in v;
4  $f_{mc}$ ← count of each element of $\mathbf{S}_{mc}$ in v;
5  $f_{sec}$ ← count of each element of $\mathbf{S}_{sec}$ in v;
6  $f_0$ ← number of empty cells in v;
7
8  repeat
9      // Receive collection message
10     if digest $(S, r^-, l)$ received from replica $j$ and $l = r$ then
11         v[j] ← $(S, r^-)$;
12         update $\mathbf{S}_{mc}, \mathbf{S}_{sec}, f_{mc}, f_{sec}, f_0$ using v;
13         // Attempt vote
14         if $f_0 = 0$ then
15             // All digests received, pick max of $\mathbf{S}_{mc}$
16             return True, max($\mathbf{S}_{mc}$);
17         else if $|\mathbf{S}_{mc}| = 1$ and $f_{mc} > f_{sec} + f_0$ then
18             // Only one most common digest, and clearly the majority
19             return True, $\mathbf{S}_{mc}[0]$;
20         else if $|\mathbf{S}_{mc}| = 1$ and $f_{mc} = f_{sec} + f_0$ and $f_{sec} \neq 0$ then
21             // 2nd most common digests could have equal count
22             if $\mathbf{S}_{mc}[0] > \max(\mathbf{S}_{sec})$ then
23                 // Most common digest is the largest
24                 return True, $\mathbf{S}_{mc}[0]$;
25             end
26         else if $|\mathbf{S}_{mc}| = 1$ and $f_{mc} = f_{sec} + f_0$ then
27             // Other digests could have equal count
28             if $\mathbf{S}_{mc}[0]$ = full_digest then
29                 // Most common digest is the largest
30                 return True, $\mathbf{S}_{mc}[0]$;
31             end
32         end
33     end
34 until timer $3\delta_n$ expires;
35
36 return False, NULL; // Return false because vote was unsuccessful
```

its digest would be “24.11001”. The lexicographic ordering of the digest string gives the total order, with “24.11111” being the `full_digest`. We use this implementation in our simulations.

Each replica maintains a list v of digests received with label r from other replicas, with at most one entry per replica. From v, it attempts to vote and choose a digest, such that all replicas choosing a digest with label r, choose the same digest.

The voter also maintains two lists and three integers that are updated when a new digest is received and stored in v. As the same digest can be received from several replicas, we count the number of occurrences of each unique digest in v. We store the digests with the highest frequency in $\mathbf{S}_{mc}$, and their count in f_{mc}. Similarly, the digests with the second highest frequency are stored in $\mathbf{S}_{sec}$, and their count in f_{sec}. Finally, the number of empty elements in v, that is the number of replicas from which the voter has not received digests, is stored in f_0.

There are four cases in which the voter can choose a digest. In all other cases, the voter has to wait for more digests. (1) The number of empty cells in v is zero (lines 14-16). In this case, the voter has to choose one of the digests that are most common. If there’s only one, it is chosen, otherwise, it chooses the largest among them. (2) There is only one most common digest, and it will remain so no matter what the replicas that have yet to send digests send (lines 17-19). It is chosen (as the obvious majority). (3) There is only one most common digest, and there is at least another digest (second most common), and no matter what the replicas that have yet to send digests send, any second most common digest can be at most as frequent as the most common one (lines 20-25). In this case, if the most common digest is larger than all the second most common digests, the voter chooses it. (4) The `full_digest` is the only most common one and no other digest can become more common. Since it is the largest by definition, it is chosen.

This approach enables the voter to successfully vote, before receiving digests from all replicas, while guaranteeing consistency, thus accounting for delay faults. Specifically, item 4 (lines 26-32) enables a two-replica RTCS to be available when one of its replicas is faulty, if the other replica receives all measurements and is up-to-date on the state (`full_digest`).

The voter returns unsuccessfully after $3\delta_n$, which is enough time to allow the non-faulty replicas to send their digests.

V. Performance Guarantees

We first formalize the notion of consistency.

Definition 1 (Consistency)**.** *Consistency is said to hold on label r for an RTCS iff all the setpoints with label r for the same actuator have the same value.*

Theorem V.1 (Consistency)**.** *Quarts guarantees consistency in the presence of any number of delay- or crash-faulty replicas.*

Proof. From Algorithm 1, we see that the setpoints depend entirely on the measurements $\mathbf{Z}$ used for computation, the state $\mathbf{H}$, and the state label r^-. Hence, it suffices to show that, any two replicas, C_i and C_j, that compute setpoints for label r, using measurements $\mathbf{Z}_r^i$ and $\mathbf{Z}_r^j$, states $\mathbf{H}_r^i$ and $\mathbf{H}_r^j$, and state labels r_i^- and r_j^-, respectively, will have $\mathbf{Z}_r^i = \mathbf{Z}_r^j$, $\mathbf{H}_r^i = \mathbf{H}_r^j$, and $r_i^- = r_j^-$. This is shown by the following lemmas.

Lemma V.1. $\forall\ r, i, j,\ \mathbf{Z}_r^i = \mathbf{Z}_r^j$

Lemma V.2. *When replicas C_i and C_j compute setpoints for a label r using state labels r_i^- and r_j^- respectively, $r_i^- = r_j^-$.*

Lemma V.3. $\forall\ r, i, j,\ r_i^- = r_j^- \implies \mathbf{H}_r^i = \mathbf{H}_r^j$

The proofs for the lemmas are in Appendix A. ☐

RTCSs aim to have minimal latency between generation of measurements and issuing of setpoints. Hence, agreement protocols for RTCSs must have a low latency-overhead. The latency overhead of a replica due to Quarts is the time spent in the `collect_and_vote` function (Algorithm 1, line 7). It depends on the network delay between the controllers. As described in Section III, we consider a lossy network with a bounded propagation delay δ_n for non-lost packets. For an RTCS using such a network, Quarts guarantees bounded latency-overhead for each replica that issues a setpoint.

Theorem V.2 (Bounded Latency-Overhead)**.** *When a non-faulty replica of an RTCS using Quarts issues a setpoint, its latency overhead is bounded by $5\delta_n$.*

The proof of Theorem V.2 is in Appendix B.

VI. Performance Evaluation

We evaluate the consistency (Definitions 1) of Quarts and that of existing protocols [7], [13]. We also evaluate other performance metrics of RTCSs, namely, availability (Definition 2), latency (Defintion 3) and messaging cost. The analytical evaluation of these metrics appears to be mathematically intractable. So, we perform the evaluation using simulations.

A. Performance Metrics

From Definition 1, we obtain a measure of consistency as follows. If consistency holds for an RTCS for setpoints with label r, then put $\gamma_r = 1$, else $\gamma_r = 0$. Note that $\gamma_r = 1$, when no setpoints are sent by an RTCS with a label r, or when only one of the replicas sends setpoints for label r. The measure of consistency of an RTCS is given by $\Gamma = \mathbb{E}\left[\gamma_r\right]$.

In addition to providing consistency, a controller of an RTCS is required to have high availability.

Definition 2 (Availability). *Availability is said to hold for an RTCS for a setpoint with label r w.r.t. an actuator A_j, iff A_j receives a setpoint with label r.*

If availability holds for an RTCS for setpoints labelled r for an actuator A_j, then put $\psi_r^j = 1$, else $\psi_r^j = 0$. Then, a measure of the availability for setpoints with label r is $\psi_r = \frac{1}{h}\sum_{j=1}^{h} \psi_r^j$, and the availability of an RTCS is $\Psi = \mathbb{E}\left[\psi_r\right]$.

Besides high availability, RTCSs require low latency. Let $S_r^i, 1 \leq i \leq m$, be the time instant at which sensor i sends measurement with label r and $I_r^j,\ 1 \leq j \leq g$, be the time instant at which the controller C_j issues the setpoints with label r. Then, latency is defined as follows.

Definition 3 (Latency). *Latency of an RTCS for a setpoint with label r is $\delta_r = \min_{j\in[1,g]} I_r^j - \min_{i\in[1,m]} S_r^i$.*

From Definition 3, we compute the mean latency of an RTCS with an agreement protocol as $\Delta = \mathbb{E}[\delta_r]$. Besides having a low mean-latency, it is important that the delay distribution is light-tailed. So, we also consider the 99th percentile of latency (δ_{p99}) in our evaluation.

Another important performance metric for RTCSs is the messaging cost needed to provide consistency and high availability. For simplicity, we use the number of messages exchanged as an indicator of messaging cost. Specifically, the messaging cost (ω_r) of an RTCS for computing and sending setpoints corresponding to label r is evaluated as the total number of messages labelled r exchanged by the replicas among each other and with the actuators. We consider the mean (Ω) and 99th percentile (ω_{p99}) of messaging cost.

In summary, the metrics of interest are consistency (Γ), availability (Ψ), mean and 99th percentile of latency (Δ, δ_{p99}), and mean and 99th percentile of messaging cost (Ω, ω_{p99}).

B. Agreement Protocols

The state-of-the-art agreement protocols against which we compare the performance of Quarts, are active replication with Fast-Paxos consensus [13] and passive replication with hot or cold standbys [7]. Hereafter, we denote them as AC, PH, and PC, respectively. We denote Quarts as Q.

a) Protocol AC: All replicas receive measurements and perform computation. Before sending setpoints to the actuators, the replicas agree on which replica sends the septoints for this label (which is equivalent to agreeing on which setpoint is sent), by using a consensus protocol. We choose the widely used Fast Paxos [13] protocol for consenus as it is optimized for low latency and guarantees consistency. This is an example of an active-replication-based agreement protocol that ensures consistency by agreement on the septoints, as opposed to agreement on the measurements, as done by Quarts.

b) Protocol PC: This is a passive-replication scheme, in which only the primary replica sends setpoints to the actuators. In PC, the standbys are cold, i.e., only the primary receives measurements and computes setpoints. After each computation, the primary sends heartbeats to the standbys, the absence of which is used to detect the failure of the primary. The heartbeats also serve as a mechanism for synchronization of the state of the standbys with that of the primary. When the primary is detected as faulty, the standbys elect a leader among themselves by using Fast Paxos to hold a consensus. As the cold standbys do not receive measurements, the newly elected primary can only begin computing setpoints for next label.

c) Protocol PH: This agreement protocol is the same as protocol PC, except that the standbys in protocol PH are hot, i.e., they receive measurements and compute setpoints but do not send them to the actuators. As a result, when a new primary is elected after the failure of the existing primary, it can issue setpoints for the current label and does not have to wait for the reception of measurements of the next label.

PC and PH are expected to have lower latency and higher availability than the active-replication scheme AC, as the replicas do not hold consensus for every label. For this reason, passive replication is the traditional choice of agreement protocol for RTCSs and is included in our evaluation.

C. Simulation Methodology

We consider an RTCS for the control of electric grids that uses a Kalman-filter based controller [2]. In this RTCS, PMUs send measurements every $T = 20$ ms. As mentioned in Section III, the network is considered to be probabilistic synchronous [9], in which packets are dropped with a probability p, and are otherwise received within a delay bounded by $\delta_n = 0.5$ ms. We simulate this using a Bernoulli random variable with success probability $1 - p$, and a uniformly distributed delay in the range $(0, 0.5]$ ms when the packet is delivered. The detailed fault model is described below.

1) Fault Model: The controller replicas are considered to have independent faults, consisting of both crashes and delays. Crash faults are fail-stop, causing a replica to be faulty indefinitely until it is externally recovered. Delay faults are intermittent, i.e., a delay-faulty replica might turn non-faulty after the duration of the computation that was delayed.

To simulate the bursty behavior of faults, we use the Gilbert-Elliot model [21]. The replica could be in one of two states: Good state (G) or Bad state (B). The transition probabilities from G to B and B to G are q_B and q_G, respectively. In state G, the replica is faulty with a probability p_d, which simulates delay faults. The computation delay of a replica is drawn from

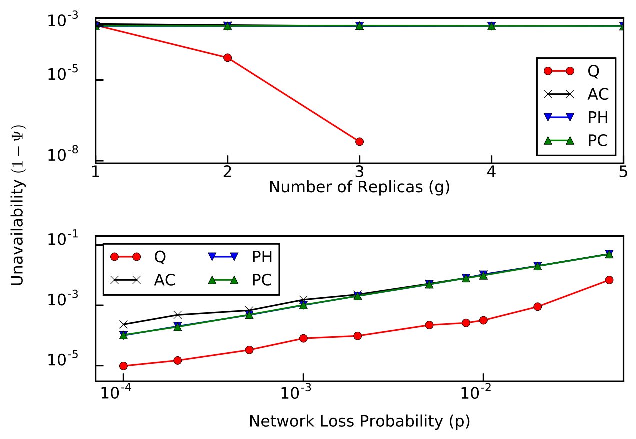

Fig. 2. Unavailability with varying g and varying p

an exponential distribution such that $\mathbb{P}(\text{delay} > \tau) = p_d$, where τ is the threshold for a non-faulty replica computation. The replica is considered unavailable if its delay is such that it fails to send the setpoint before new measurements arrive.

In state B, the replica is faulty with probability 1, thereby simulating crash faults. The parameters (q_B, q_G, p_d) of the model are evaluated from the probability of crash faults (θ_c), probability of delay faults (θ_d), and mean-time-to-repair from crash faults (R), which are chosen for each particular scenario.

2) Quality of the Estimation: We run each simulation until the 95% confidence interval of the estimated value has a half width, from the central value, less than 5%. Hence, all our results can be interpreted with the 95% confidence interval as $[0.95\hat{x}, 1.05\hat{x}]$, where $\hat{x}$ is the reported estimate.

D. Results

We simulated the 4 protocols (Q, AC, PC, PH) for several scenarios with 2 values of m (10 and 100), 5 values for each of g and h (1 through 5), 3 different fault models, and 10 values of p (between 10^{-4} and 0.05). Our simulation platform was a high-throughput cluster with 278 nodes and the simulation campaign lasted a total of 10 days. Due to limited space, we present the results of a distilled representative set of scenarios.

Henceforth, unless otherwise specified, the parameters used are $g = 2,\ h = 1, m = 10,\ p = 1E{-}3,\ \theta_c = 1E{-}4,\ \theta_d = 1E{-}3,\ R = 1$ s, $\Delta_n =\ 0.5$ ms, $\tau =\ 8$ ms, and $T\ =\ 20$ ms. Figure 2 shows the unavailability $(1 - \Psi)$ for varying g and varying p. Tables I, II show the detailed simulation results for the chosen scenarios.

Finding 1. *Quarts provides higher availability than that of AC, PC and PH, while maintaining 100% consistency.*

We simulate the protocols for $g = [1, 5]$. The first plot of Figure 2 shows that the unavailability (1 - Ψ) of Quarts is more than an order of magnitude lower than that of other protocols with 2 replicas, and 4 orders of magnitude lower with 3 replicas. Additionally, for more than 3 replicas, Quarts showed no unavailability in $1E10$ runs ($\sim$ 3 days of simulation). Simulating such extremely rare-events requires more sophisticated techniques such as Importance Sampling and Palm Calculus [22], and is left for future work.

In order to put the availability improvement with Quarts into perspective, we analyze the mean-time-between-faults (MTBF) [23] of the RTCS. For the RTCS under study that sends setpoints every 20 ms, for $g = 1$, an availability of 0.9987 translates to an MTBF of 18.2 s. Using two replicas, the protocols AC, PC and PH can increase the MTBF to 18.4 s, 19.6 s and 19.9 s, respectively. By comparison, the MTBF with Quarts is 5 minutes. Furthermore, the MTBF with 3 replicas for protocols AC, PC, PH and Q is 19.7 s, 19.2 s, 19.9 s, and 4.54 days. As seen above, the existing protocols show marginal improvement in availability with each additional replica, whereas Quarts enjoys a significant increase.

The second plot of Figure 2 confirms Finding 1 for different values of p. We see that, even at extremely high-values of $p = 0.02$, Quarts has $\Psi = 0.9991$, whereas this value drops down to 0.98 for other protocols. This finding is further re-affirmed by the results of the scenarios shown in Table I.

The availability improvement of Quarts comes without any penalty to consistency. We find that the consistency of Quarts and AC is 1. However, the consistency guarantee of AC comes at the cost of low availability, as compared to Quarts. In contrast, PC and PH have an inconsistency between $1E-5$ and $1E-3$ in the presence of delay faults, for the scenarios considered. In the absence of delay faults, the probability of inconsistency for PC and PH could not be captured by our simulations in $1E10$ runs and, therefore, can be considered small. For these scenarios, however, their availability is still lower than that of Quarts.

Finding 2. *Quarts has a lower average-latency and tail-latency than other consistency-guaranteeing protocols.*

Table II shows that the mean latency of Quarts is less than that of AC with a factor of 4. PC and PH have a comparable mean that comes the expense of inconsistency. Quarts has a better mean latency than AC, as it does not perform consensus in each cycle. Furthermore, in Quarts, when a replica is up-to-date, i.e., has all the measurements and the state label $r-1$, it can enter the voting phase without waiting for the collection timer ($2\delta_n$) to expire. In the meantime, it continues to listen for queries and sends responses to other replicas.

We also see in Table II that the tail-latency of Quarts is lower than that of other protocols in the presence of delay faults, and comparable to that of PC and PH in the absence of delay faults. Such low tail-latency is due to the bounded latency-overhead of Quarts as shown by Theorem V.2.

Figure 3 shows the mean and 99th percentile of the latency of the protocols for different scenarios.

Finding 3. *Guaranteeing consistency comes at the marginal expense of a higher messaging cost.*

The mean messaging cost of Quarts and AC is marginally higher than that of PC and PH and increases with the number of replicas. This is due to the collection phase in Quarts and the consensus employed by AC, for every setpoint. Such an exchange of messages is the price to pay for achieving consistency. The 99th percentile messaging cost is also higher for Quarts, as can be seen in Appendix C.

	Unavailability $(1-\Psi)$				Inconsistency $(1-\Gamma)$		Messaging cost in messages/label (Ω)			
Scenario: $(m,\ g,\ \theta_c,\ \theta_d)$	Q	AC	PH	PC	PH	PC	Q	AC	PH	PC
#1: $(10,\ 2,\ 1E{-}4,\ 1E{-}3)$	$9.12E{-}5$	$1.24E{-}3$	$9.87E{-}4$	$1.02E{-}3$	$1.92E{-}4$	$1.28E{-}3$	4.04	5.17	3.87	3.00
#2: $(100,\ 2,\ 1E{-}4,\ 1E{-}3)$	$1.46E{-}4$	$1.19E{-}3$	$9.76E{-}4$	$1.03E{-}3$	$1.68E{-}3$	$1.50E{-}3$	4.38	5.17	3.87	3.00
#3: $(10,\ 2,\ 1E{-}5,\ 1E{-}4)$	$1.02E{-}5$	$9.98E{-}4$	$1.01E{-}3$	$9.96E{-}4$	$2.40E{-}5$	$1.38E{-}4$	4.04	5.10	3.74	3.00
#4: $(10,\ 2,\ 1E{-}4, 0)$	$8.14E{-}5$	$1.37E{-}3$	$1.01E{-}3$	$1.02E{-}3$	$(0, 3E-10]^*$	$(0, 3E-10]^*$	4.04	5.02	3.50	3.00
#5: $(10,\ 3,\ 1E{-}4, 0)$	$2.25E{-}8$	$1.01E{-}3$	$1.01E{-}3$	$9.92E{-}4$	$(0, 3E-10]^*$	$(0, 3E-10]^*$	9.18	9.02	6.67	5.01

TABLE I
SIMULATION RESULTS FOR SELECT SCENARIOS OF QUARTS (Q), ACTIVE CONSENSUS (AC), PASSIVE HOT (PH), AND PASSIVE COLD (PC)
* NO INCONSISTENCY WAS OBSERVED IN $1E10$ RUNS

	Latency in ms $(\Delta,\ \delta_{p99})$			
Scenario	Q	AC	PH	PC
#1	0.96, 3.08	4.28, 8.78	1.40, 5.59	1.41, 5.59
#2	0.98, 3.11	4.27, 8.78	1.40, 5.58	1.41, 5.59
#3	0.82, 2.42	3.91, 8.09	1.12, 4.26	1.12, 4.26
#4	0.39, 0.78	3.52, 4.18	0.25, 0.5	0.25, 0.5
#5	0.50, 0.84	3.50, 4.15	0.25, 0.49	0.25, 0.5

TABLE II
MEAN AND 99^{th} PERCENTILE LATENCY FOR THE SCENARIOS IN TABLE I

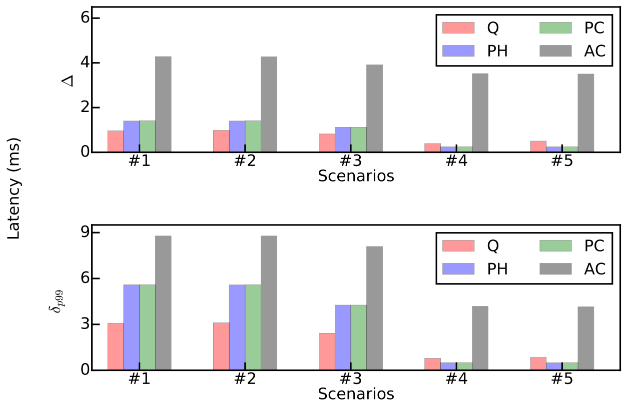

Fig. 3. Mean and 99^{th}% latency in different scenarios

VII. CONCLUSION AND FUTURE WORK

We presented Quarts, an agreement protocol for RTCSs that uses active replication of the controller. Quarts is designed and formally proven to guarantee consistency with a bounded latency-overhead. We performed an extensive performance evaluation of Quarts and existing agreement protocols through simulation under different conditions of number of replicas, network losses, fault profiles, etc. We showed that besides guaranteeing consistency, Quarts improves the availability of an RTCS by more than an order of magnitude, when compared with existing agreement protocols. Moreover, Quarts improves the tail-latency performance of the RTCS. These benefits of Quarts come at a marginal increase in messaging cost when compared to passive replication schemes.

We intend to implement Quarts and deploy it in our campus microgrid with an RTCS that performs control of electric grids [1]. As Quarts applies to Kalman-filter controllers, which have a wide range of applications, we intend to implement a Quarts API for such systems. We also intend to model and verify our design and implementation using formal modeling and verification tools, such as BIP [24], so as to facilitate the ease of adoption of Quarts for mission-critical applications.

REFERENCES

[1] Andrey Bernstein, Lorenzo Reyes-Chamorro, Jean-Yves Le Boudec, and Mario Paolone. A Composable Method for Real-Time Control of Active Distribution Networks with Explicit Power Setpoints. Part I: Framework. *Electric Power Systems Research*, 125:254–264, 2015.

[2] Styliani Sarri, Lorenzo Zanni, Miroslav Popovic, Jean-Yves Le Boudec, and Mario Paolone. Performance Assessment of Linear State Estimators using Synchrophasor Measurements. *IEEE Transactions on Instrumentation and Measurement*, 65(3):535–548, 2016.

[3] Paulo Leitão. Agent-Based Distributed Manufacturing Control: A State-of-the-Art Survey. *Engineering Applications of Artificial Intelligence*, 22(7):979–991, 2009.

[4] Donald J Reifer, Victor R Basili, Barry W Boehm, and Betsy Clark. Cots-Based Systems–Twelve Lessons Learned about Maintenance. In *COTS-Based Software Systems*, pages 137–145. Springer, 2004.

[5] Maaz Mohiuddin, Wajeb Saab, Simon Bliudze, and Jean-Yves Le Boudec. Axo: Masking Delay Faults in Real-Time Control Systems. In *Industrial Electronics Society, IECON 2016-42nd Annual Conference of the IEEE*, pages 4933–4940. IEEE, 2016.

[6] Seth Gilbert and Nancy Ann Lynch. Perspectives on the CAP Theorem. Institute of Electrical and Electronics Engineers, 2012.

[7] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. The Primary-Backup Approach. *Distributed systems*, 2:199–216, 1993.

[8] Leslie Lamport et al. Paxos Made Simple. *ACM Sigact News*, 32(4):18–25, 2001.

[9] Dacfey Dzung, Rachid Guerraoui, David Kozhaya, and Yvonne-Anne Pignolet. Never Say Never–Probabilistic and Temporal Failure Detectors. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 679–688. IEEE, 2016.

[10] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[11] Bruno Sinopoli, Luca Schenato, Massimo Franceschetti, Kameshwar Poolla, Michael I Jordan, and Shankar S Sastry. Kalman Filtering with Intermittent Observations. *IEEE transactions on Automatic Control*, 49(9):1453–1464, 2004.

[12] Rudolph Emil Kalman. A New Approach to Linear Filtering and Prediction Problems. *Journal of basic Engineering*, 82(1):35–45, 1960.

[13] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[14] Brian M Oki and Barbara H Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17. ACM, 1988.

[15] Judith L Gersting, Robert L Nist, Dale B Roberts, and RL Van Valkenburg. A Comparison of Voting Algorithms for N-Version Programming. In *System Sciences, 1991. Proceedings of the Twenty-Fourth Annual Hawaii International Conference on*, volume 2, pages 253–262. IEEE, 1991.

[16] Douglas M Blough and Gregory F Sullivan. A Comparison of Voting Strategies for Fault-Tolerant Distributed Systems. In *Reliable Distributed Systems, 1990. Proceedings., Ninth Symposium on*, pages 136–145. IEEE, 1990.

[17] Leslie Lamport. Time Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[18] Roman Rudnik, Lorenzo Enrique Reyes Chamorro, Andrey Bernstein, Jean-Yves Le Boudec, and Mario Paolone. Handling Large Power Steps in Real-Time Microgrid Control Via Explicit Power Setpoints. In *PowerTech 2017*, number EPFL-CONF-226196, 2017.

[19] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[20] R. Guerraoui, D. Kozhaya, M. Oriol, and Y. A. Pignolet. Who's on Board?: Probabilistic Membership for Real-Time Distributed Control

Systems . In *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, pages 167–176, Sept 2016.

[21] Edwin O Elliott. Estimates of Error Rates for Codes on Burst-Noise Channels. *The Bell System Technical Journal*, 42(5):1977–1997, 1963.

[22] Jean-Yves Le Boudec. *Performance Evaluation of Computer and Communication Systems*, volume 12. Epfl Press Lausanne, 2010.

[23] Hubert Kirrmann. §2.5 Dependable Automation. *Collaborative Process Automation Systems*, page 100, 2010.

[24] Simon Bliudze, Alessandro Cimatti, Mohamad Jaber, Sergio Mover, Marco Roveri, Wajeb Saab, and Qiang Wang. Formal Verification of Infinite-State BIP Models. In *International Symposium on Automated Technology for Verification and Analysis*, pages 326–343. Springer, 2015.

APPENDIX A
PROOF THEOREM V.1

From Algorithm 1, we see that the setpoints depend entirely on the measurements $\mathbf{Z}$ used for computation, the state $\mathbf{H}$, and the state label r^-. Hence, it suffices to show that, any two replicas, C_i and C_j, that compute setpoints for label r, using measurements $\mathbf{Z}_r^i$ and $\mathbf{Z}_r^j$, states $\mathbf{H}_r^i$ and $\mathbf{H}_r^j$, and state labels r_i^- and r_j^-, respectively, will have $\mathbf{Z}_r^i = \mathbf{Z}_r^j$, $\mathbf{H}_r^i = \mathbf{H}_r^j$, and $r_i^- = r_j^-$. This is shown by the following lemmas.

Lemma A.1. $\forall\ r, i, j,\ \mathbf{Z}_r^i = \mathbf{Z}_r^j$

Proof. $\mathbf{Z}_i^r$ is the set of measurements used for computation for label r by a C_i in Algorithm 1 (line 16).
If controller replicas C_i and C_j compute setpoints for label r, i.e., line 16 of Algorithm 1 is triggered, then on both C_i and C_j the function `collect_and_vote` must have returned True (Algorithm 2, line 10).
Therefore, the function `vote` must have returned True, for label r.
From Lemma A.2, we know that, $\forall\ r$, if Algorithm 5 on controller C_i and C_j returns True, then $S^i_{chosen} = S^j_{chosen}$.
Additionally, as the measurements for label r recieved by from a same sensor by C_i and C_j are identical, $S^i_{chosen} = S^j_{chosen} \implies \mathbf{Z}_r^i = \mathbf{Z}_r^j$. □

Lemma A.2. *If Algorithm 5 returns True for a label r for controller replicas C_i and C_j, then the chosen digests S^i_{chosen} and S^j_{chosen} are equal.*

Proof. For label r, one of the lines 16, 19, 24 or 30 of Algorithm 5 would have been triggered on the replicas.

a) Line 16: The number of empty cells in `v` is zero. In this case, the voter has to pick one of the digests that are most common. If there's only one, it is picked, otherwise, it picks the largest among them.

b) Line 19: There is only one most common digest, and it will remain so no matter what the replicas that have yet to send digests send. It is picked (as the obvious majority).

c) Line 24: There is only one most common digest, and there is at least another digest (second most common), and no matter what the replicas that have yet to send digests send, the second most common digest will be at most as frequent as the most common. In this case, if the most common digest is larger than all the second most common digests, the voter picks it.

d) Line 30: The `full_digest` is the only most common one and no other digest can become more common. Since it is the largest by default, it can be picked immediately. □

Lemma A.3. *When replicas C_i and C_j compute setpoints for a label r using state labels r_i^- and r_j^- respectively, $r_i^- = r_j^-$.*

Proof. If controller replicas C_i and C_j compute setpoints for label r, i.e., line 16 of Algorithm 1 is triggered, then on both C_i and C_j the function `collect_and_vote` must have returned True (Algorithm 2, line 10).
Therefore, the function `vote` must have returned True, for label r.
From Lemma A.2, we know that, $\forall\ r$, if Algorithm 5 on controller C_i and C_j returns True, then $S^i_{chosen} = S^j_{chosen}$.
From Section IV-C, we see that if two digest are equal, then they have the same state labels.
Therefore, $S^i_{chosen} = S^j_{chosen} \implies r_i^- = r_j^-$. □

Lemma A.4. $\forall\ r, i, j,\ r_i^- =\ r_j^- \implies \mathbf{H}_r^i = \mathbf{H}_r^j$

Proof. We prove this lemma by strong induction on r, the label of setpoint being computed.
Base Case: When $r = 1$, $r_i^- = r_j^- = 0$, then the states used for computing the first setpoint are $\mathbf{H}_r^i = \mathbf{H}_r^j = \emptyset$. Thus, the statement holds for $r = 1$.

Induction Hypothesis: Let the statement hold of the labels in $[1, l]$, where $l > 1$.
Thus, $\forall\ r \in [1, l-1],\ i,\ j,\ r_i^- = r_j^- \implies \mathbf{H}_r^i = \mathbf{H}_r^j$
This means that, any two replicas that computed setpoints for a label $r \in [1, l-1]$ and had the same state label, had the same state.

Inductive Step: To show that, for label l, $\forall\ i,\ j : C_i, C_j$ compute setpoints for label l, $l_i^- = l_j^- \implies \mathbf{H}_l^i = \mathbf{H}_l^j$.

When a replica computes a setpoint, its state is updated by line 17 of Algorithm 1.
When both C_i and C_j performed a computation of setpoints for label $l-1$, then the label of their state would be updated to $l-1$. Hence, $l_i^- = l_j^- = l-1$.
Additionally, since, the states and correction factors used for this computation are the same due to the induction hypothesis, and from Lemma A.1, we know that the measurements used for the computation are also same, the states resulting from the comptuation are also same (Algorithm 1, line 17).
Therefore, we have $l_i^- = l_j^- = l-1$ and $\mathbf{H}_l^i = \mathbf{H}_l^j$
Thus, by principle of induction, $\forall\ r, i, j,\ r_i^- =\ r_j^- \implies \mathbf{H}_r^i = \mathbf{H}_r^j$ □

APPENDIX B
PROOF OF THEOREM V.2

Proof. As mentioned earlier, the latency overhead of a replica due to Quarts is the time spent in the `collect_and_vote` function (Algorithm 1, line 7). The `collect_and_vote` function, described in Algorithm 2, consists of three functions: `collect_missing_measurements`, `collect_missing_state`, and `vote`. Since, the

	Messaging cost in messages/label (Ω, ω_{p99})			
Scenario	Q	AC	PH	PC
#1	4.04, 6	5.17, 6	3.87, 8	3, 3
#2	4.38, 6	5.17, 6	3.87, 8	3, 3
#3	4.04, 6	5.10, 6	3.74, 7	3, 3
#4	4.04, 6	5.02, 5	3.50, 4	3, 3
#5	9.18, 15	9.02, 9	6.67, 8	5.01, 5

TABLE III
MEAN AND 99^{th} PERCENTILE MESSAGING COST FOR THE SCENARIOS IN TABLE I

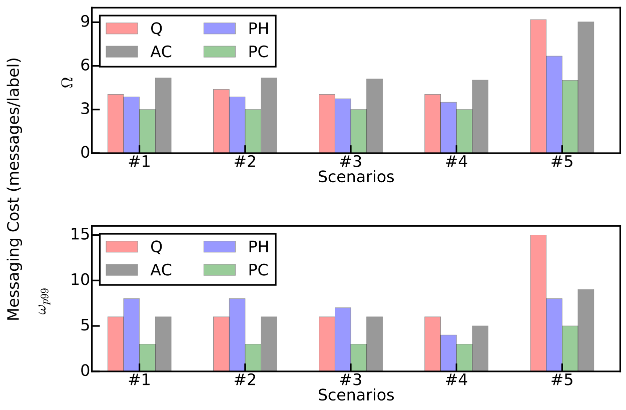

Fig. 4. Mean and 99^{th}% messaging cost in different scenarios

statement applies only to non-faulty replicas, from Section III, we know that operations that do not involve waiting for the network to deliver messages have negligible delay.

The `collect_missing_measurements` (Algorithm 3) and `collect_missing_state` (Algorithm 4) functions can be merged into one function without requiring parallelism. The resulting function would send a *Query* and an *Advertisement*, then continue listening to *Queries*, *Responses*, *Advertisements*, and *Updates* until the timer of $2\delta_n$ expires. As it is designed to terminate after the timer expires, this collection function results in a bounded latency-overhead of $2\delta_n$.

The `vote` function is also designed to terminate after a timer expires. Its timer, however, is $3\delta_n$ (Algorithm 5 line 34). Therefore, the voting phase has a bounded latency-overhead of $3\delta_n$.

As a result, Quarts is shown to have a bounded latency-overhead of $5\delta_n$ due to its collection and voting phases. □

APPENDIX C
ADDITIONAL SIMULATION RESULTS

Figure 4 and Table III show the mean and 99th percentile of the messaging cost. We see that the messaging cost of Quarts is comparable to that of other consistency guaranteeing protocol (AC), thereby reaffirming Finding 3.