Axo: Masking Delay Faults in Real-Time Control Systems

Maaz Mohiuddin, Wajeb Saab, Simon Bliudze, Jean-Yves Le Boudec
School of Computer Science and Communication Systems
École Polytechnique Fédérale de Lausanne, Switzerland
{firstname.lastname}@epfl.ch

***Abstract*—We consider real-time control systems that consist of a controller that computes and sends setpoints to be implemented in physical processes through process agents. We focus on systems that use commercial off-the-shelf hardware and software components. Setpoints of these systems have strict real-time constraints: Implementing a setpoint after its deadline, or not receiving setpoints within a deadline, can cause failure. In this paper, we address delay faults: faults that cause setpoints to violate their real-time constraints. We present Axo, a fault-tolerance protocol that guarantees safety and improves availability for a class of such systems that exhibit two main properties: the setpoints must have a known validity horizon, and process agents must be capable of handling duplicate setpoints. To reason about delay faults, and consequently design Axo, we present an abstraction of a controller; the abstraction applies to a wide range of real-time control systems. We prove guarantees of safety and availability. Finally, we present an implementation of Axo and the results of the tests performed with Commelec, a real-time control system for electric grids.**

I. Introduction

Real-time control systems (RTCSs) are systems in which a software controller continuously applies control to a set of processes, thereby changing or maintaining the system state. Some common examples are RTCSs for electric grids [1]–[3], manufacturing processes [4] and autonomous vehicles [5], [6].

Many of these systems are mission critical: their failure can lead to serious damage [7]. Yet, there is a trend in increasingly relying on commercial off-the-shelf (COTS) hardware such as cRIO (from NI), DAP server (from Alstom), and MGC600 (from ABB). Also, as RTCS controllers have a large size, they often use third-party libraries, including COTS software. Such COTS-based RTCSs (cb-RTCSs) are susceptible to faults incurred by their hardware and software components [8].

Figure 1 depicts a model of RTCS architecture. It has a set of controlled processes, each with an actuator and a software process agent (PA). Also, it consists of sensors that read the state of controlled processes and other uncontrollable parts of the system, and that send this state as *measurements* to the controller. In some systems, PAs also send some information as measurements. The controller makes use of the measurements received from sensors and PAs, in order to compute *setpoints* that are sent to the PAs. To control its process, each PA *implements* the received setpoint through the corresponding actuator, which results in changes to the state.

Setpoints are computed and issued by the controller of an RTCS in order to drive the state of the system in a

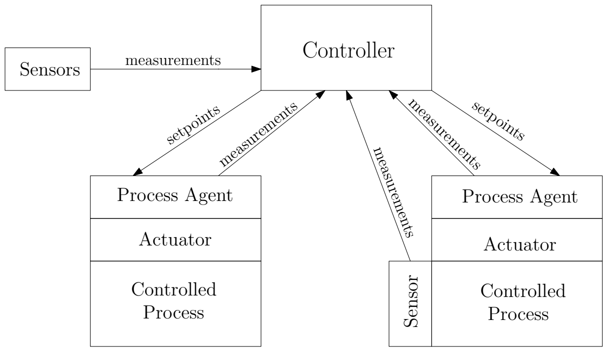

Fig. 1: Model of RTCS architecture

certain direction. Therefore, the setpoints are a function of the perceived current state of the system and will become invalid when the state drifts by some threshold. This threshold is specific to each RTCS and can be mapped, based on the inertia of the underlying system, to a *validity horizon*: a time beyond which setpoints are invalid and must not be implemented.

A validity horizon implies that setpoints are subject to strict real-time constraints. Therefore, a delay incurred due to software/hardware faults in the controller, or due to transmission delays in the network, could violate such constraints and lead to failure. We refer to such faults as *delay faults*. It is worth noting that crash faults are a special case of delay faults, where the delay is infinite.

Traditional fault-tolerance protocols can be classified as benign-fault tolerance [9], [10] and Byzantine-fault tolerance (BFT) [11]–[13]. Benign-fault tolerance handles crash-only faults and is thus not designed for delay faults. Whereas BFT protocols are designed for arbitrary kinds of faults, they focus on faults affecting the value of a setpoint, rather than its timing attributes.

An existing approach for dealing with delay faults in a real-time setting [14] proposes encapsulating all the time-critical functions of the controller into a strictly real-time component: the timely computing base (TCB) [15]. However, this is not a viable option for cb-RTCSs, as their controllers often execute complicated optimization functions. Moreover, the proposed approach requires specifying a bound on the execution time of the time-critical functions, which is not feasible in the presence of COTS software.

We designed Axo, a fault-tolerance protocol that targets delay faults in RTCSs. Axo is transparent to the RTCS, i.e., it imposes no interference with the RTCS functionality. Axo can be used with minor additions to the controller software; they are elaborated further in Section IV.

Axo requires the RTCS to exhibit the following properties:

Property 1 (P_1)**.** *There exists a known validity horizon (τ_o).*

Property 2 (P_2)**.** *PAs are able to handle duplicate setpoints.*

P_1 requires RTCSs to provide the value of τ_o as an input to Axo. P_2 requires process agents to handle duplicate setpoints, a property generally exhibited in RTCSs that use absolute rather than differential setpoints. An example of absolute setpoints would be an electric grid controller instructing a battery agent that is injecting 8kW to '*set the injected power to 10kW*', rather than a differential setpoint that would be to '*increase the injected power by 2kW*'.

Our main contributions in this paper are as follows. First, we develop an abstraction of a controller that applies to a wide range of RTCSs. This abstraction enables us to design Axo[1] that to the best of our knowledge, is the first fault-tolerance protocol that guarantees safety and improves availability of cb-RTCSs. Then, we formally define the concepts of safety and availability as pertaining to delay faults, and we provide proofs for their guarantees. Finally, we provide an implementation of Axo as a fault-tolerance layer, and the results of testing it with Commelec [1], an RTCS for electric grids.

The rest of the paper is as follows. In Section II, we discuss related work. In Section III, we list the assumptions on the RTCS required by Axo. In Section IV, we define the controller abstraction. In Section V, we discuss the Axo design and algorithms. In Section VI, we prove the correctness guarantees of Axo. In Sections VII and VIII, we discuss the implementation of Axo, and the tests performed with Axo on Commelec. Finally, we conclude in Section X.

II. Related Work

To the best of our knowledge, Axo is the only fault-tolerance protocol that guarantees safety and improves availability for cb-RTCSs.

In the literature, delay faults for real-time systems have been studied under the name of *timing faults* [14]–[17]. The closest existing technique is the work done by Veríssimo and Casimiro on TCB [15]. They propose an architecture and programming model that can be used to provide generic delay fault tolerance for real-time systems [14]. As mentioned in Section I, this fault-tolerance approach relies on encapsulating and rewriting the time-critical functions of the real-time system in the TCB module: a strictly real-time component. This method does not apply for cb-RTCSs that are characterized by their large code-base that consists of third-party libraries and generally complex functions, for which it is not feasible to rewrite and implement in the TCB.

[1]Named after Axolotl, a type of salamanders that exhibits very quick regeneration when one or more of their organs fail.

Furthermore, several components of the TCB architecture require an implementation specific to each RTCS. This drawback is shared with other generic architectures such as the time-triggered architecture (TTA) [17]. Whereas, Axo is a layer of software that can be used on any RTCS that satisfies the assumptions (Section III) and requires only minor additions to the RTCS controller software (Section IV). This enables the deployment of Axo on existing RTCSs.

Functions to be implemented in the TCB or the TTA require a known bound on execution time. This requires static analysis of generally complex functions that might include COTS software. Additionally, we have seen that in some RTCSs [1], the execution time heavily depends on the parameters provided at run time. This would require further dynamic analysis of the execution time, a task that does not fit within the real-time constraints of RTCSs.

Other work in this field has focused on improving the quality-of-service and response-time of the systems [16]. The authors focus on transaction systems as opposed to RTCSs, and do not aim at providing hard real-time constraints as Axo does.

Traditional BFT protocols [11], [13] do not generally consider the timing attributes of the setpoints. Moreover, all BFT protocols require consensus among the replicas. The consensus can take unbounded time [18], delaying delivery of setpoints to PAs indefinitely. This property makes them unsuitable for tolerating delay faults, even in the cases when they are designed for real-time applications [13].

Active replication protocols, such as [10], use hot standbys, all of which simultaneously compute setpoints. The use of hot standbys incurs zero delay in sending the setpoints, which makes it attractive for RTCSs, especially in the context of delay faults. However, when one replica is delay faulty, it can still send a setpoint at some time after its validity horizon. This violates safety and must be explicitly handled. Axo uses active replication with an added mechanism to provide safety in the presence of delay faults.

III. Assumptions

As mentioned in Section I, we assume that the RTCS obeys P_1 and P_2 (Properties 1 and 2). Additionally, we consider that the controller is only susceptible to crash and delay faults, and that the network can only drop, delay, and reorder messages. Generic Byzantine faults, such as the controller performing an incorrect setpoint computation or the network changing the contents of messages, are not considered.

Moreover, as RTCSs perform real-time actions on distributed nodes, the controllers and PAs naturally have a global notion of time. This is realized either through a GPS-based or network-based (e.g., precise time protocol (PTP), network time protocol (NTP)) time-synchronization protocol. Such protocols are characterized by their accuracy. Henceforth, the synchronization accuracy of the time-synchronization protocol is denoted by δ_s.

IV. Abstraction of the Controller

Let t_c be the first time-instant at which the measurements used to compute a setpoint are processed by the controller. At t_c, the perceived state of the system by the controller is closest to the actual state. As we consider the controller to not be susceptible to Byzantine faults, a setpoint computed based on this perceived state will not steer the system into an infeasible state, as long as the setpoint is received within its validity horizon. To guarantee this, the validity horizon τ_o must be chosen such that it takes into consideration the difference between the perceived and the actual state of the system, at the times when the controller makes a decision to compute. Given t_c and τ_o, we define the validity of a setpoint as follows.

Definition 1 (Valid Setpoint). *A setpoint is* valid*, if and only if it is received by a PA at* $t_r \leq t_c + \tau_o$*. Otherwise, it is* invalid*.*

For RTCSs obeying P_1 (Property 1), τ_o is known. Whereas, a measure of t_c and t_r can be obtained at the controller and the PA, respectively. These measures will have an uncertainty of δ_s, the accuracy of the time-synchronization protocol.

However, the operations of obtaining these measures are susceptible to delay faults. Therefore, one can at best obtain close approximations to these values (t_c^* and t_r^*). As our aim is to guarantee that PAs receive no invalid setpoints, it is essential that the obtained approximations never result in an invalid setpoint being classified as valid. Hence, on one hand, it is imperative to obtain a $t_r^* \geq t_r$ and a $t_c^* \leq t_c$. On the other hand, these approximations have to be close, as worse approximations lead to more valid setpoints being classified as invalid, an error that must be minimized.

In the presence of delay faults, the operation of measuring t_r will return the desired t_r^*. This value is equal to t_r in non-faulty conditions, and is greater than t_r in the presence of faults. Hence, it is the closest obtainable approximation. However, it is non-trivial to obtain $t_c^* \leq t_c$. Section IV-A presents an abstract model of RTCS controllers that we have developed. This model will be essential for obtaining t_c^*, as shown in Section IV-B.

A. Controller Model

We derived an abstraction of an RTCS controller by studying several RTCSs [1]–[6]. The abstract model is shown in Algorithm 1, excluding the lines in blue (lines 2, 5, 6). The controller receives measurements from PAs and sensors. For instance, a controller for an RTCS for electrical grids receives the state of the resources (e.g., state of charge of a battery) from the PAs and the state of the grid from phasor measurement units. Measurements received by the controller are stored (lines 11-12).

Based on these measurements, the controller invokes the `readyToCompute()` function that returns true when a computation can begin (line 3). The `readyToCompute()` function might or might not utilize the available measurements (M) in its decision. Type A decisions use M to decide whether to begin a computation. An example of this decision is when the controller reacts to the state of the system, such as '*when the active power exceeds 1kW*'. Type B decisions do not use the measurements. These decisions might rely on time, as is the case with periodic RTCSs, where the output depends on the current time (TS_{now}).

Algorithm 1: Abstract Model of an RTCS Controller

```
1  while true do
2      $t_c^*$<A> ← TS$_{now}$;
3      decision ← readyToCompute(M, TS$_{now}$);
4      if decision then
5          $t_c^*$<B> ← TS$_{now}$;
6          send $t_c^*$<type>;
7          SPs = compute(M) ;
8          issue(SPs);
9      end
10 end
11 for each measurement m received do
12     M ← M ∪ $m$;
13 end
```

Note that the two types, A and B, are the types of the decisions and not of the controller. A controller might have both type A and type B decisions. In addition to covering different types of decisions, this abstraction covers all degrees of synchrony in the RTCS. For example, when `readyToCompute()` always returns true, the underlying RTCS is completely asynchronous, whereas when the function returns true only when all the measurements have been received, the underlying RTCS is completely synchronous. Thus, it is generic enough to cover a wide range of RTCSs.

The controller uses the available measurements to compute and issue setpoints to all PAs (lines 7-8). The abstraction affords any implementation of the `compute()` function, thereby remaining non-restrictive.

B. Obtaining t_c^* from the Model

Recall that t_c is the first time-instant at which the measurements used to compute a setpoint are processed by the controller. As type A decisions involve a measurement-based `readyToCompute()` function, then t_c for type A (t_c<A>) is the time of the onset of the `readyToCompute()` function. Therefore, obtaining a measure t_c^*<A> immediately before this function (Algorithm 1 line 2) results in the closest approximation of t_c<A>. This approximation never exceeds t_c<A> in the presence of delay faults, as any delay in the operation of obtaining this measure also delays the time at which the measurements are processed. This delay offsets both t_c^*<A> and t_c<A> by the same amount.

On the other hand, as type B decisions do not use measurements in the `readyToCompute()` function, the first time-instant at which the measurements are processed are in the `compute()` function itself. Hence, by the same reasoning used to obtain t_c^*<A>, the closest approximation t_c^*<B> of t_c<B> is obtained immediately before the `compute()` function (Algorithm 1 line 5).

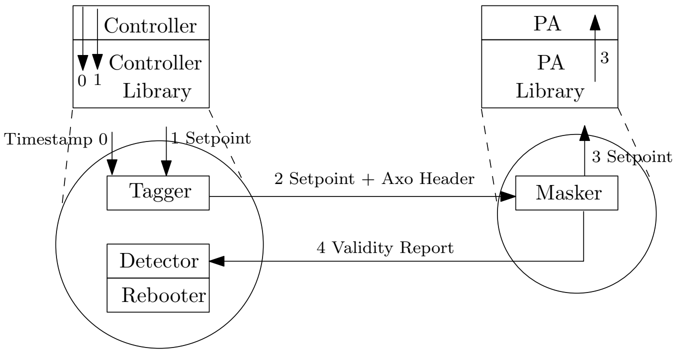

Fig. 2: Axo design

In addition to obtaining t_c^*, the controller is required to send this obtained timestamp to a specified destination (Algorithm 1 line 6). Here, the controller has to know the type of decision performed for the computation of each setpoint and send the appropriate timestamp.

V. Axo Design

Axo uses active replication of the controllers and requires $g + 1$ replicas to tolerate g delay and crash faults. In active replication, all the replicas independently receive measurements from sensors and compute and issue setpoints to the PAs. This enables Axo to tolerate faults with minimal replication-overhead. Additionally, there is no consensus between the replicas, which enables Axo to have minimal latency. However, active replication alone does not guarantee safety, as some replicas might still send delayed setpoints.

To provide safety, Axo uses the *Controller Library* and the *PA Library* as shown in Figure 2. The Controller Library comprises the *tagger*, the *detector* and the *rebooter*. The PA Library contains the *masker*. The tagger and the masker are responsible for fault masking, the detector for fault detection and the rebooter for recovery. The design of the tagger and the masker is presented below, whereas the design of the detector and the rebooter is left for future work.

Together, the tagger and the masker achieve fault masking. The design of the tagger is shown in Algorithm 2. The tagger receives two types of messages from the controller. First, it receives the timestamp (t_c^*) that the controller records just before deciding to compute as shown in Algorithm 1. t_c^* is a lower bound on t_c, as mentioned in Section IV.

Second, the tagger intercepts the setpoint sent by the controller to the PA and prepends it with an Axo header. The Axo header consists of a 1-byte replica ID (a unique replica ID that serves as its identifier for all Axo-related communication), a 2-byte destination port of the original setpoint and an 8-byte timestamp. For the timestamp, it uses the most recently received t_c^* from the controller. Besides this information, the tagger also adds the replica's status that is later used for fault detection by other replicas.

The setpoint, along with the Axo header, is sent to the masker that uses Algorithm 3 to process it. As the validity horizon (τ_o) is measured from t_c, the setpoint is valid at all times $t \leq t_c^* + \tau_o \leq t_c + \tau_o$. To account for the accuracy of time-synchronization protocol δ_s, and for the computation time following the validity check at the masker, we use τ, instead of τ_0. Here, $\tau = \tau_o - (2\delta_s + \delta_m)$, where δ_m is the upper bound on the computation time at the masker between performing the validity check (line 2) and sending the setpoint to the PA (line 3).

Algorithm 2: Tagger

```
1  $t_c^* \leftarrow 0$;
2  for each message received from the controller do
3      if message is a timestamp then
4          $t_c^* \leftarrow$ timestamp received;
5      end
6      else if message is a setpoint {SP, dest} then
7          populate axoHeader;
8          Send {axoHeader, SP} to masker of PA dest;
9      end
10 end
```

Algorithm 3: Masker

```
1  for each message {axoHeader, SP} received do
2      if $\mathtt{TS}_{\mathtt{now}} \leq \mathtt{axoHeader}.t_c^* + \tau$ then
3          Remove axoHeader and send SP to PA;
4          valid ← true;
5      else
6          valid ← false;
7      end
8      Send validity report to detectors of controllers;
9  end
```

As mentioned in Section I, our goal is to tolerate delay faults in the controller and the network. However, Axo cannot mask delay faults that occur in the masker, after the validity check is performed. Therefore, the only operation performed after this check, in both the design and implementation of the masker, is the forwarding of the setpoint to the PA. Then, this part of the masker can be regarded as a thin layer that is not susceptible to delay faults. This is in line with previous work in fault-tolerance literature [11], [13].

If the time of reception of the setpoint $t \leq t_c^* + \tau$, then the setpoint is considered to be valid. When a setpoint is to be forwarded to the PA (Algorithm 3, line 2), the destination in the Axo header is used to recreate the original setpoint. At this point, the setpoint sent to the PA is indistinguishable from that sent by the controller. Hence, Axo remains transparent to the RTCS and need not be aware of the messaging format, encryption, or authentication measures of the setpoint payload used by the RTCS.

Besides discarding the delayed setpoints, the masker prepares a validity report for each setpoint. The masker sends this report to the detector on all replicas, which will use it in the detection process. The mechanism for detection is left for future work.

As Axo modifies the setpoints sent by the controllers to the PAs (by adding an Axo header), PAs that do not contain the PA library cannot process the modified setpoints. Therefore, Axo is not backward compatible: the PA library must be present on all PAs.

VI. Formal Correctness Guarantees

In this section, we provide correctness guarantees for Axo, where correctness is defined as Safety (Definition 2) and Availability (Definition 3). Recall the definition of a valid setpoint (Definition 1) from Section IV and the assumptions required for Axo from Section III.

Definition 2 (Safety). *Safety is said to hold for a PA P, if and only if all setpoints received by P are valid.*

Theorem 1 (Safety). *If all controller replicas have an Axo controller library, then safety is guaranteed for all PAs.*

The proof is presented in Appendix A.

Definition 3 (Availability). *Availability is said to hold for a PA P in an interval $[a, b]$, if and only if P receives at least one valid setpoint in $[a, b]$ from a set of controller replicas $\mathcal{C}$. Consequently, $\mathcal{C}$ is said to* provide availability *for P in $[a, b]$.*

Remark. When the replicas in $\mathcal{C}$ use a fault-tolerance protocol f – such as Axo – to provide availability, we say that *f provides availability*.

Recall from Section III the synchronization accuracy δ_s of the time-synchronization protocol. Any measurement of time has an uncertainty of δ_s. Consequently, the uncertainty in recording the end-to-end delay of a setpoint is $2\delta_s$. Therefore, to guarantee safety, any fault-tolerance protocol needs to conservatively discard setpoint with an end-to-end delay greater than $\tau_o - 2\delta_s$. Furthermore, this deadline needs to be further offset in order to account for the computation time of the fault-tolerance protocol (δ_f). Hence, in order to tolerate delay faults, all fault-tolerance protocols need to discard potentially valid setpoints whose end-to-end delay lies in the uncertainty interval $[\tau_o - 2\delta_s - \delta_f, \tau_o]$, thereby reducing availability. However, Axo is designed to minimize this uncertainty interval and provide maximum availability under these constraints.

Theorem 2 (Availability). *Consider an interval $[a, b]$ and a fault-tolerance protocol f that, using a set of g replicas $\mathcal{C}$, guarantees safety for a PA P. If f provides availability for P in $[a, b]$, and the time taken by f to process a setpoint is at least as much as that taken by Axo, then Axo, using $\mathcal{C}$, also provides availability for P in $[a, b]$.*

The proof is presented in Appendix B.

Note that, Theorem 2 already accounts for faults in the network and guarantees the best possible availability. Hence, Axo is able to give guarantees on both safety and availability without being in violation of the impossibility result in [18].

VII. Implementation

We developed a proof-of-concept implementation of Axo in C++; it is publicly available at https://goo.gl/MFdu28. We also make available an API in C++ that can be used by the controller in order to record and send timestamps to the tagger as described in Algorithm 1.

The API provides the controller with three functions: `get_timestamp_ptp()`, `get_timestamp_gps()` and `send_timestamp()`. Depending on the underlying time-synchronization protocol of the RTCS, one of the first two functions would be used to record the timestamp. The first function implements this for systems using PTP, whereas the second implements it for systems using GPS. The key difference is that the former accounts for the offset in time from the time-synchronization server when obtaining the timestamp, whereas the latter has no such offset. The functions need to be inserted in the controller at the correct locations, based on type of decision performed, as described in Section IV.

In the design of Axo, the tagger intercepts messages sent from the controller to the PAs. For this purpose, we use the `libnetfilter_queue`[2] (NF_QUEUE) framework from the Linux *iptables* project. NF_QUEUE is a userspace library that provides an interface to handle packets queued by the kernel packet filter. We use this framework to filter packets sent from the controller to the PAs (setpoints), and from the controller to the tagger (timestamps), using iptables' rules based on the destination IP address and the port number of the packets. Then, the tagger receives packets filtered by these rules, and handles them accordingly. As the packets filtered by iptables are only processed by the tagger, then if the tagger crashes, the setpoints are dropped and not sent to the PAs. In this way, the setpoints cannot bypass Axo, thereby upholding safety.

The API and the Axo implementation currently support only UDP-based IPv4 communication between the controller and the PAs. Additionally, our implementation uses a 1-byte replica ID that is derived from the IP address of the controller. Specifically, we use the last octet of the IPv4 address as the replica ID. Therefore, our implementation supports up to 256 replicas.

Besides the additions to the controller needed to record and send timestamps to the tagger using the Axo API, the installation of Axo is plug-and-play. It requires neither any information about the inner workings of the controller nor any modifications to the PAs.

VIII. Case Study: Commelec Controller

To demonstrate the usability of Axo with an existing RTCS, we use Commelec [1], an RTCS for control of electrical grids.

A. Applicability of Axo to Commelec

Recall that Axo requires two properties from an RTCS, namely P_1 and P_2 (see Properties 1 and 2). Also, for Axo to be applicable, it should be possible to model the RTCS controller as shown in Algorithm 1. Here, we show how Commelec fares for these assumptions and for the controller model.

The Commelec controller receives measurements concerning the electrical grid's state from phasor measurement units

[2]http://www.netfilter.org/projects/libnetfilter_queue/

(sensors). It also receives another form of measurements from PAs, referred to as *advertisements*. The controller computes and sends setpoints to the PAs, the implementation of which maintains the electrical grid in a feasible state. It follows that the implementation of a late setpoint, the computation of which was performed using a state of the electrical grid that no longer applies, could destabilize the electrical line or more adversely, result in cascading failures that could trip the entire grid. Such failures are more likely to arise in cases of an islanding maneuver of electrical grids. Therefore, we conclude that there exists a validity horizon for Commelec setpoints, which means that Commelec satisfies P_1. In our experiments, we consider a test grid with a battery and a photovoltaic cell, and the validity horizon τ_o is known to be 95 ms.

Commelec makes use of absolute setpoints. Hence, the implementation of multiple setpoints has the same effect as the implementation of last implemented setpoint. Therefore, the Commelec controller also satisfies P_2, which requires PAs to handle duplicate setpoints.

The Commelec controller continuously receives measurements from sensors that are stored and used in subsequent setpoint computations. It is the reception of advertisements from all PAs that onsets setpoint computation. After a computation, existing advertisements are flushed and new ones are awaited. If, however, a sufficiently long time elapses after sending a setpoint and not all advertisements were received, it sends empty setpoints to request for advertisements.

The above algorithm can be mapped to the controller model presented in Algorithm 1 as follows. The `readyToCompute()` function corresponds to both type A and B decisions described in Section IV. When sufficient advertisements are received, a type A decision is taken and the advertisements are used for computation of meaningful setpoints. Alternatively, when a timeout occurs, a type B decision is taken and "computation" of empty setpoints is performed.

In the former case, the controller registers a timestamp immediately before checking whether all advertisements have been received and sends this timestamp when the check returns true. In the latter case, the controller registers and sends a timestamp just before preparing and sending the empty setpoints.

Based on the discussion above, we conclude that Axo can be applied to the Commelec RTCS.

B. Fault Tolerance with Axo

To study the fault tolerance in Commelec with Axo, we use three replicas of the Commelec controller and install Axo. Two of the replicas C_1 and C_2 are housed on virtual machines, and one replica C_3 is a Lenovo T400 laptop with 4 GB RAM and 2.33 GHz Intel i5 processor. Furthermore, there are two PAs: a battery PA and a photovoltaic PA. The controllers and the PAs are all interfaced with a simulated grid and the validity horizon of setpoints is taken to be $\tau_o = 95$ ms. We use PTP for time synchronization which has an accuracy (δ_s) of 1 ms, and we take $\delta_m = 1$ ms. Then, we have $\tau = 92$ ms. Furthermore,

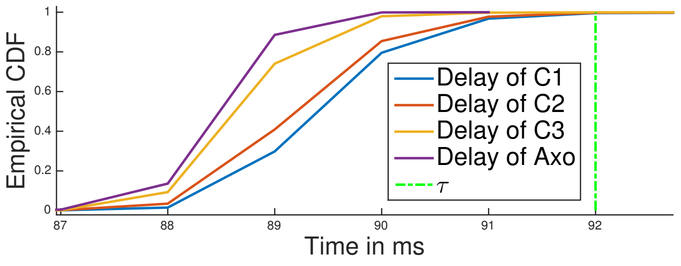

Fig. 3: Safety with Axo

the network is configured to be lossy with a loss probability of 0.01.

In order to demonstrate the safety and availability properties of Axo in the presence of a wide range of fault profiles, we inject bursty delay-faults and crash faults with different fault rates in different replicas. The probabilities that C_1, C_2 and C_3 send an invalid setpoint are 0.02, 0.01 and 0.001, respectively. C_1 has no crash faults, whereas C_2 and C_3 are crash-faulty with a probability of 0.001 and 0.0001, respectively. Furthermore, to detect and recover from faults, we have an initial version of detector and rebooter in place.

Figure 3 shows the safety property of Axo. It shows that the delay incurred by setpoints sent by controllers C_1, C_2, C_3 recorded at the masker is sometimes greater than τ. However, the delay perceived by the PAs is always less than τ. Hence, we conclude that the PAs never receive an invalid setpoint. Furthermore, we see an added benefit of Axo: the overall delay perceived by the PAs is reduced. This is because Axo uses active replication and the first valid setpoint received by the masker is forwarded to the PA. Also, Axo does not add any additional delay, except for the processing of a setpoint at the masker.

We expect the availability for C_1, C_2 and C_3 to be 98%, 99% and 99.9%. In practice, due to a reduction in availability as a result of detection and recovery time (repair time), we observed that the availability for C_1, C_2 and C_3 was 81.31%, 89.37% and 91.31%. As the faults are independent, the expected availability with Axo is 99.83%. The observed availability due to Axo was 99.97%.

C. Overhead of Axo API

In order to characterize the latency overhead due to the Axo API, we profiled its function calls on a NI cRIO-9068, a ruggedized machine commonly used for deployment of RTCSs in the field. Recall from Section VII that the Axo API provides different functions for RTCSs that use GPS- and PTP-based time synchronization.

The two functions are `get_timestamp_gps()` and `get_timestamp_ptp()`, the difference being the presence of `get_offset()` in the latter. The `get_offset()` function is called only when enough time has passed since the last time the offset was recorded. This is realized through a configurable parameter T_{offset} (default 1 s). Thus, when PTP is used, if the time between successive setpoint computations (T_{comp}) is greater than T_{offset}, the `get_offset()` function is triggered each time a setpoint is computed. Alternatively, when $T_{comp} < T_{offset}$, the `get_offset()` function is only called $\frac{100 \times T_{comp}}{T_{offset}}$ % of the setpoint computations.

Let $\xi = T_{comp}/T_{offset}$. Let the mean execution times for the `get_timestamp_gps()`, `get_timestamp_ptp()`, `get_offset()` and `send_timestamp()` be Δ_{gps}, Δ_{ptp}, Δ_{offset} and Δ_{send}, respectively. Hence, the delay due to Axo API when GPS is used is $G_{Axo} = \Delta_{gps} + \Delta_{send}$ and when PTP is used is $P_{Axo} = \Delta_{ptp} + \Delta_{send}$. Furthermore, when $\xi > 1$, $\Delta_{ptp} = \Delta_{gps} + \Delta_{offset}$ and when $\xi < 1$, $\Delta_{ptp} = \Delta_{gps} + \xi \times \Delta_{offset}$.

From our profiling, we observe that $\Delta_{gps} = 14.26$ μs, $\Delta_{offset} = 194$ μs and $\Delta_{send} = 33.18$ μs. Then $G_{Axo} = 47.44$ μs. Additionally, for the Commelec controller, we have $T_{comp} = 200$ ms $< T_{offset}$ (1 s). Therefore, with $\xi = 0.2$, we have $P_{Axo} = 144.44$ μs.

IX. Acknowledgments

This research was supported by the "SNSF - NRP 70" Energy Turnaround project. The authors would like to thank the anonymous reviewers for their valuable feedback.

X. Conclusion and Future Work

In this paper we have presented Axo that to the best of our knowledge, is the first fault-tolerance protocol for tolerating delay faults in COTS-based RTCSs. To reason about delay faults and design Axo, we developed an abstraction of controllers used in RTCSs by studying several RTCSs. The abstract model developed is generic and applies to systems with any degree of synchrony.

We have discussed the design of the fault-masking components of Axo. Furthermore, we have described the publicly available implementation of Axo and the associated API. Additionally, we have formally proven the safety and availability properties of Axo and we have validated these properties through tests with an existing RTCS for control of electrical grids.

The fault-detection and fault-recovery mechanisms are currently under consideration. Furthermore, in the future, we will analyze the bounds on the time taken for detection of a faulty replica and its recovery with Axo.

Emerging RTCSs are making use of a hierarchy of controllers. We plan to extend Axo, in such a way that it becomes composable and will therefore directly handle hierarchical RTCSs. Another possible extension of Axo is with regard to Byzantine faults. Current approaches deal with Byzantine faults using consensus, which is unsuitable for RTCSs. This is left for future work.

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] Konstantina Christakou, D-C Tomozei, J-Y Le Boudec, and Mario Paolone. GECN: Primary Voltage Control for Active Distribution Networks via Real-Time Demand-Response. *Smart Grid, IEEE Transactions on*, 5(2):622–631, 2014.

[3] Zhe Xiao, Tinghua Li, Ming Huang, Jihong Shi, Jingjing Yang, Jiang Yu, and Wei Wu. Hierarchical MAS Based Control Strategy for Microgrid. *Energies*, 3(9):1622–1638, 2010.

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

[5] Chris Urmson, J Andrew Bagnell, Christopher R Baker, Martial Hebert, Alonzo Kelly, Raj Rajkumar, Paul E Rybski, Sebastian Scherer, Reid Simmons, Sanjiv Singh, et al. Tartan Racing: A Multi-Modal Approach to the DARPA Urban Challenge. 2007.

[6] Tan Yew Teck, Mandar Chitre, and Prahlad Vadakkepat. Hierarchical Agent-Based Command and Control System for Autonomous Underwater Vehicles. In *Autonomous and Intelligent Systems (AIS), 2010 International Conference on*, pages 1–6. IEEE, 2010.

[7] G. Andersson, P. Donalek, R. Farmer, N. Hatziargyriou, I. Kamwa, P. Kundur, N. Martins, J. Paserba, P. Pourbeik, J. Sanchez-Gasca, R. Schulz, A. Stankovic, C. Taylor, and V. Vittal. Causes of the 2003 Major Grid Blackouts in North America and Europe, and Recommended Means to Improve System Dynamic Performance. *Power Systems, IEEE Transactions on*, 20(4):1922–1928, Nov 2005.

[8] 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.

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

[10] 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.

[11] Miguel Castro, Barbara Liskov, et al. Practical Byzantine Fault Tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[12] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

[13] Jonathan Kirsch, Stuart Goose, Yair Amir, Dong Wei, and Paul Skare. Survivable SCADA via Intrusion-Tolerant Replication. *Smart Grid, IEEE Transactions on*, 5(1):60–70, 2014.

[14] A. Casimiro and P. Verissimo. Generic Timing Fault Tolerance Using a Timely Computing Base. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 27–36, 2002.

[15] Paulo Veríssimo and António Casimiro. The Timely Computing Base Model and Architecture. *Computers, IEEE Transactions on*, 51(8):916–930, 2002.

[16] S. Krishnamurthy, W. H. Sanders, and M. Cukier. A Dynamic Replica Selection Algorithm for Tolerating Timing Faults. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pages 107–116, July 2001.

[17] Hermann Kopetz. Fault Containment and Error Detection in the Time-Triggered Architecture. In *Autonomous Decentralized Systems, 2003. ISADS 2003. The Sixth International Symposium on*, pages 139–146. IEEE, 2003.

[18] 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.

Appendix A
Proof of Safety: Theorem 1

If all controller replicas have an Axo controller library, then safety is guaranteed for all PAs.

Proof. Let s be a setpoint computed at some controller replica C with a timestamp $t_c^* \leq t_c$, where t_c is the first instant at which the measurement to compute this setpoint were processed. (refer Section IV)
As C contains the Axo controller library, then s will be intercepted by the tagger.
If P does not contain an Axo PA library, it will never receive s, thereby upholding safety.

Otherwise, the tagger will forward s to the masker of P.
Recall from Section IV, that the time at which the setpoint is received at P is t_r.
By Definition 1, s is valid if and only if $t_r \leq t_c + \tau_o$.
Based on Algorithm 3 line 2, the masker of P will only accept s at $t_r^* \leq t_c^* + \tau$.
Recall that both t_c^* and t_r^* are measured locally at C and P, respectively.
Since the inaccuracy in time-synchronization protocol is δ_s, the true time at which the setpoint is received at the masker is $t_r' \leq t_r^* + \delta_s$.
Similarly, the true time at which the setpoint is first valid $t_c \geq t_c^* - \delta_s$. Therefore, $t_r^* \leq t_c^* + \tau \implies t_r' \leq t_c + \tau + 2\delta_s$
Since the processing time of the masker is bounded by δ_m, then $t_r \leq t_r' + \delta_m$.
Thus, any accepted setpoint will arrive at P at $t_r \leq t_c + \tau + 2\delta_s + \delta_m$.
But $\tau = \tau_o - 2\delta_s - \delta_m$
So, the masker of P will only accept and forward s to P if $t_r \leq t_c + \tau_o$.
Therefore, any setpoint s received by P will be valid. □

Appendix B
Proof of Availability: Theorem 2

Consider an interval $[a, b]$ and a fault-tolerance protocol f, using a set of g replicas $\mathcal{C}$, that guarantees safety for a PA P. If f provides availability for P in $[a, b]$, and the time taken by f to process a setpoint is at least as much as that taken by Axo, then Axo, using $\mathcal{C}$, also provides availability for P in $[a, b]$.

Proof. Consider an RTCS with a PA P.
Let $\mathcal{C} = \{C^1, ..., C^g\}$ be a set of g replicas of the controller of this RTCS.
Let $\mathcal{S}$ be the set of setpoints sent by all controllers in $\mathcal{C}$ in the interval $[a - \tau_o, b]$.
Consider a fault-tolerance protocol f, applied to $\mathcal{C}$, that guarantees safety for P. Denote by $\mathcal{C}_f$ the set of controllers $\mathcal{C}$ when f is applied to them.
Let $\mathcal{R} = \{s \in \mathcal{S} : s \text{ is received by } P \text{ in } [a, b] \text{ and } s \text{ is valid}\}$.
Formally, $f : \mathcal{S} \mapsto \mathcal{R}_f \subseteq \mathcal{R}$.
In other words, $\mathcal{R}$ is the set of valid setpoints sent by $\mathcal{C}$ and received by P in $[a, b]$. $\mathcal{R}_f$ is the subset of $\mathcal{R}$ that are received by P when f is applied to $\mathcal{C}$. Therefore, f provides availability for P in $[a, b]$, if and only if $|\mathcal{R}_f| > 0$. Note that we only consider the interval $[a - \tau_o, b]$ for the set $\mathcal{S}$, since setpoints sent outside this interval can never be valid if received in $[a, b]$.
We defined the following operations and sets:
Let $\mathcal{S}_f \subseteq \mathcal{S}$, be the set of setpoints, sent by $\mathcal{C}$ in $[a - \tau_o, b]$, that f allows to be sent to P. That is, $\mathcal{S} \setminus \mathcal{S}_f$ is the set that f discards before sending.
Let $\alpha_f : \mathcal{S} \mapsto \mathcal{S}_f$
Let $\mathcal{N}_f \subseteq \mathcal{S}_f$ be the set of setpoints that the network delivers to P in $[a, b]$, when f is applied.
Then, $\mathcal{N}_f = \{s \in \mathcal{S}_f : s \text{ is delivered to } P \text{ in } [a, b]\}$
Let $\gamma : \mathcal{S}_f \mapsto \mathcal{N}_f$
Then, $\mathcal{R}_f \subseteq \mathcal{N}_f$
Let $\beta_f : \mathcal{N}_f \mapsto \mathcal{R}_f$
Therefore, $f = \beta_f \circ \gamma \circ \alpha_f$
Intuitively, α_f is the operation of discarding setpoints before sending them, and thus depends on the fault-tolerance protocol. γ is the operation performed by the network, which is considered to be transparent to the fault-tolerance protocol. β_f is the operation of discarding setpoints before they are received at P, in order to guarantee safety.
Let $\mathcal{F}_g$ be the class of fault-tolerance protocols that guarantee safety to hold for P using controllers in $\mathcal{C}$. We consider all $f \in \mathcal{F}_g$ to have at least as much processing time for each setpoint as Axo.
As Axo guarantees safety (Theorem 1), Axo $\in \mathcal{F}_g$
As Axo uses active replication, it ensures that all g controller replica in $\mathcal{C}$ are active and send setpoints to P.
Furthermore, as the tagger (Algorithm 2) never discards setpoints, all the setpoints sent by the controllers are forwarded to P.
However, the tagger incurs a processing time to each setpoint, thus not all setpoints will still be sent in $[a - \tau_o, b]$.
This processing time is also incurred by all $f \in \mathcal{F}_g$, therefore

$$\forall f \in \mathcal{F}_g : \mathcal{S}_f \subseteq \mathcal{S}_{Axo} \subseteq \mathcal{S} \tag{1}$$

Now we apply γ. Since γ is transparent to the fault-tolerance protocol, then $\forall \mathcal{A}, \mathcal{B}, s \in \mathcal{A} \cap \mathcal{B}$ and $s \notin \gamma(\mathcal{A}) \implies s \notin \gamma(\mathcal{B})$.
Then, it follows from Equation 1, that

$$\forall f \in \mathcal{F}_g : \mathcal{N}_f \subseteq \mathcal{N}_{Axo} \tag{2}$$

Since all fault-tolerance protocols $f \in \mathcal{F}_g$ guarantee safety, they need to discard invalid setpoints.
Let, $\tau_f \leq \tau_o$ be the delay after which f discards setpoints.
Note that $\tau_{Axo} = \tau = \tau_o - 2\delta_s - \delta_m$.
Then

$$\mathcal{R}_f = \beta_f(\mathcal{N}_f) = \{s \in \mathcal{N}_f : \text{end-to-end delay of } s \leq \tau_f\} \tag{3}$$

Note that,

$$\forall \mathcal{A}, \forall f_1, f_2 \in \mathcal{F}_g, \tau_{f_1} \leq \tau_{f_2} \implies \beta_{f_1}(\mathcal{A}) \subseteq \beta_{f_2}(\mathcal{A}) \tag{4}$$

To guarantee safety, any fault-tolerance protocol needs to conservatively discard setpoints that lie within the uncertainty interval brought about by the uncertainty in measuring time (δ_s) and the processing time of the setpoint (δ_f). Since, $\delta_f \geq \delta_m$, then

$$\forall f \in \mathcal{F}_g, \tau_f \leq \tau \tag{5}$$

Then, using Equations 2, 3, 4, 5, we conclude that

$$\begin{aligned} &\forall f \in \mathcal{F}_g, \mathcal{R}_f \subseteq \mathcal{R}_{Axo} \\ \implies &\forall f \in \mathcal{F}_g, |\mathcal{R}_f| > 0 \implies |\mathcal{R}_{Axo}| > 0 \end{aligned}$$

□