Axo: Detection and Recovery for Delay and Crash Faults in Real-Time Control Systems

Maaz Mohiuddin, Wajeb Saab, Simon Bliudze, Jean-Yves Le Boudec

Axo: Detection and Recovery for Delay and Crash 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*—Real-time control systems use controllers that compute and issue setpoints within stringent delay constraints. Failure to do so, due to a crash or delay as a result of software and/or hardware faults, can cause failure of the controlled resources. Recently, Axo, a protocol for masking crash and delay faults by replicating the controller, was proposed. Axo provides safety by discarding delayed setpoints, and it relies on the presence of valid setpoints for providing availability. To ensure that enough valid setpoints are issued, faulty controller replicas need to be detected and recovered. We present a mechanism for detection and recovery of delay- and crash-faulty replicas under the Axo framework. These mechanisms were designed to be soft state (i.e., their state can be reconstructed from received messages) to enable seamless additions of new replicas. Besides presenting the design, we analytically characterize the time to detect and recover a faulty replica, and we validate them experimentally. We demonstrate the performance of Axo by using two case studies: the first provides a stability analysis of an inverted pendulum system with Axo, and the second shows the fault-tolerance performance of Axo through a deployment on a real-time control system that controls a CIGRÉ low-voltage benchmark microgrid.**

***Index Terms*—Reliability, delay faults, fault detection, fault recovery, real-time**

I. Introduction

A. Problem Description

Real-time control systems (RTCSs), such as control of electrical grids [1], autonomous vehicles [2], and manufacturing processes [3], increasingly rely on software-based controllers and commercial off-the-shelf (COTS) components. Such COTS-based RTCSs, hereafter cb-RTCSs, are susceptible to faults due to their software and hardware [4]. Also, many RTCSs are mission-critical and the manifestation of faults leads to failures causing damage to property and life [5], [6].

An RTCS consists of sensors that measure the state of processes and send this state, as *measurements*, to the controller. The controller uses these measurements, in addition to others received from process agents (PAs), to compute *setpoints* that are then issued to PAs. PAs *implement*, through actuators, the received setpoints in the respective process. Setpoints issued by the controller have strict real-time constraints, i.e., setpoints delayed by more than a *validity horizon* are invalid and should not be implemented. Such delays can occur due to software or hardware faults in the controller or in the network. We refer to such faults as *delay faults*. A special case of delay faults is *crash faults*, whereby a setpoint is indefinitely delayed. Crash faults do not violate the delay constraint, but result in the loss of a controller, thereby leading to an indefinite loss of setpoints issued. Both delayed setpoints (as a result of delay faults) and continuously absent setpoints (as a result of crash faults) potentially lead to instability, as shown in Section VI.

Traditional benign fault-tolerance protocols tolerate crash-only faults and Byzantine fault-tolerance (BFT) protocols tolerate faults concerning the value of a setpoint but not the timing aspects of it. The existing solutions for delay-fault tolerance [7], [8] require the controller, either in entirety or in part, to be implemented using specialized strictly real-time hardware and software. The large code-base, extensive use of third-party libraries and off-the-shelf hardware of cb-RTCSs make these protocols unsuitable for the cb-RTCSs.

Recently, Axo, a protocol for masking delay faults in cb-RTCSs, was proposed [9]. Axo applies to all RTCSs that can handle duplicate setpoints and have a well-defined validity horizon, a deadline after which setpoints are considered invalid. For these RTCSs, Axo masks delay faults from the PA by discarding invalid setpoints. Axo enables the use of active replication, in which several copies, hereafter *replicas*, of the same controller actively compute and issue setpoints. With active replication, the presence of one non-faulty replica is enough to deliver a valid setpoint to the PA. Axo also uses a thin software-layer to perform fault masking, thereby hiding the effects of delay faults from PAs. Hence, Axo remains agnostic to the inner working of the RTCS and applies to a wide range of RTCSs with minor additions to the controller.

The two main masking properties of fault-tolerance protocols are safety and availability [10]. Axo guarantees safety for all PAs [9], where safety is the property in which invalid setpoints are never received by PAs. Availability, however, requires the existence of at least one non-faulty replica at all times. Delay faults can be transient or persistent in nature, and if persistently delay-faulty replicas are not detected and recovered in time, the system will end up having no non-faulty replicas, and availability suffers. Hence, to continuously mask delay faults, faulty replicas need to be detected and recovered.

B. Challenges

Delay faults are an end-to-end phenomena, the two ends being the controller and the PAs. Existing fault-detection mechanisms (see Section II) rely on polling the state of the controller hence cannot be used for detecting delay faults.

To detect delay faults in the controller replicas, we introduce feedback, from the PAs, about the validity of the

setpoints received. If a PA receives a valid setpoint, then the corresponding controller is deemed to not have a delay fault, and *vice versa*. However, the main challenge in developing such a design is the possibility of the messages being lost, reordered, retransmitted, or delayed. Also, after a single setpoint computation, a controller with multiple PAs will asynchronously receive multiple feedbacks, one from each PA. These potentially different feedbacks need to be efficiently aggregated for fault detection.

The design of a detection and recovery protocol also faces several other challenges. First, the transient nature of delay faults makes their detection nontrivial. For instance, a replica might experience a delay fault for one setpoint only. In such cases, it is not only nontrivial to detect the transient delay-fault but also superfluous, as it would be more advantageous to avoid recovering that replica. Second, cb-RTCSs do not often use dedicated communication links hence are susceptible to packet losses, messages delays, retransmissions, and reordering. This could affect recovery, causing a replica to reboot multiple times for a single fault, or not at all.

Furthermore, as Axo operates without the knowledge of the inner workings of the RTCS, the rate at which an RTCS controller issues setpoints is not known. Therefore, conventional techniques for detecting crash faults, such as monitoring the frequency at which setpoints are issued, are ineffective.

C. Contributions

In this paper, we present algorithms for the detection and recovery of delay and crash faults in RTCSs that use Axo for fault masking. The algorithms integrate with the fault-masking features of Axo, thereby enabling Axo to continue masking faults despite one or more controller replicas turning faulty. The detection and recovery algorithms are designed to be soft state [11], i.e., their state can be reconstructed from received messages after a reboot. This enables the seamless addition and removal of replicas. Additionally, the algorithms are designed to keep Axo agnostic to the RTCS, in order for it to apply to a wide range of RTCSs.

II. Related Work

To the best of our knowledge, this is the first work that addresses delay-fault detection for cb-RTCSs.

Previous work in the literature studied the problem of detecting *timing faults*: faults that affect the timing attributes of a setpoint rather than its value. For instance, the authors of [12] propose a method of detecting timing faults under the framework of the timely computing base (TCB) [8]. Such a framework, however, presupposes the encapsulation and rewriting of the core RTCS functionality in a strict real-time component. This encapsulation cannot be applied to cb-RTCSs, as they have a large code-base mainly consisting of third-party libraries. A similar drawback is prevalent in [13], as the method applies to RTCSs that are developed under the Time-Triggered architecture (TTA) [7]. Our solution assumes the use of Axo, a thin layer of software that can be deployed on RTCSs without rewriting the original code [9].

In order to detect timing faults, the aforementioned work also require additional information from the RTCS. Timing-failure detection under the TTA framework requires *a priori* knowledge of intended send and receive instants of messages [13]. Similarly, detection under the TCB framework requires a known bound on the computation times of the time-critical functionalities of the RTCS [12]. Such requirements cannot be met for the cb-RTCSs considered in this paper, and our solution does not require such information.

Existing mechanisms for fault detection rely on monitoring the replica, such as using heartbeats [14], or on probing the replica for its current state so as to detect inconsistencies [15]. Such mechanisms target crash-only faults and cannot be extended to delay faults that are inherently an end-to-end property. Replicas themselves do not contain any state to indicate whether or not they are delay faulty, hence probing or monitoring the replicas will not provide insight for delay-fault detection. Our solution makes use of the PAs in order to correctly detect delay faults.

Another approach to detecting faults is through detailed modelling of the controller under faulty and non-faulty conditions [16], [17]. The trained models are then used to classify a replica as faulty during run-time. Such methods are prone to modelling errors and are limited to RTCSs that have constant workloads, making them unsuitable for generic RTCSs where the workload of the controller is not known *a priori*.

Fault recovery in distributed systems shares similarities with the consensus problem. Message losses, delays, and reordering poses a challenge to the requirement of ensuring that each replica is recovered only once after detection. However, consensus does not guarantee termination under these conditions [18], and the delays brought about by the multiple rounds of message exchange in state-of-the-art consensus protocols [19] are not suitable for RTCSs. To avoid using consensus, our solution makes use of the partial order provided by the time synchronization of the RTCS, and it recovers a replica with minimal delay, at most once after detection.

III. System & Fault Model

The fault-detection and fault-recovery algorithms described in this paper are built for the Axo fault-tolerance protocol. Hence, they require the same properties of the RTCS needed by Axo for fault masking as described in [9].

Axo applies to RTCSs that exhibit two main properties: (1) There exists a known validity horizon (τ_o) and (2) PAs are capable of handling duplicate setpoints. The validity horizon is specific to each RTCS and depends on the inertia of the underlying process. Axo requires τ_o as an input in order to perform fault masking and detection. The second property is inherent in RTCSs that use absolute, rather than differential, setpoints. Absolute setpoints are usually idempotent: implementing two or more duplicate setpoints has the same effect on the system as implementing only the first one received. An example of absolute setpoints would be an electric-grid controller that instructs a battery agent that is injecting 8 kW to '*set the injected power to 10 kW*', rather than a differential setpoint that would be to '*increase the injected power by 2 kW*'.

Additionally, for each setpoint, we assume that there exists a *"conception time"* t_c at which the controller begins processing the measurements used to compute this setpoint. Note that the conception time is the time at which the preconditions of computing a setpoint are met and is different from the time at which the computation of setpoints actually begins. A COTS-based controller cannot reliably estimate the conception time, in particular in the presence of delay faults. A mechanism to reliably obtain a conservative estimate (t_c^*) of the conception time from a controller in the presence of delay faults is described in Algorithm 1 in [9].

Definition 1 (Valid Setpoint)**.** *A setpoint is* valid*, if and only if, at the time of reception (t_r) at a PA, $t_r \leq t_c + \tau_o$, where t_c is the conception time of this setpoint and τ_o is the validity horizon. Else, it is* invalid*.*

We consider that a setpoint is sent to one PA, hence the setpoint has a well-defined time of reception. If a controller sends setpoints to different PAs after a computation, we consider them to be different setpoints, all with the same conception time.

We target delay and crash faults that affect the controller and the network. Therefore, our fault model consists of delayed operations, crashes in the controller, and an asynchronous network where packets can be delayed, lost, retransmitted or reordered. Generic Byzantine faults, such as wrong computations or setpoint contaminations, are not considered.

Definition 2 (Faulty Controller)**.** *A controller C is faulty at time t, if all the setpoints whose conception time equals the latest conception time at or before t are invalid.*

As RTCSs perform real-time operations on distributed nodes, they naturally have a global notion of time obtained by GPS or network protocols such as network-time protocol (NTP) and precision-time protocol (PTP). Let δ_s be an upper bound on the inaccuracy of the time-synchronization protocol.

IV. Axo Design for Detection and Recovery

Axo uses active replication of the controller and requires $g + 1$ replicas to tolerate g delay and crash faults. In traditional active-replication protocols, all the replicas independently receive measurements from sensors, compute and hold a consensus before sending one chosen setpoint to the PA. The replication of the controller is masked from the PAs, along with the faults in the replicas. In Axo, to avoid the latency overhead of consensus, the controller replicas do not directly communicate with each other. They send the computed setpoints to the PAs, where delayed setpoints are discarded. As the cb-RTCSs, for which Axo is designed, can handle duplicate setpoints, the delay faults in replicas are masked, whereas the replication is exposed. So, when at least one of the replicas is non-faulty, the PAs will receive valid setpoints, thus providing availability with minimal replication overhead.

In order to guarantee safety by discarding delayed setpoints, Axo uses a *controller library* on each controller replica and a *PA library* on each PA, as shown in Figure 1. The controller library is composed of three modules: the *tagger* for tagging

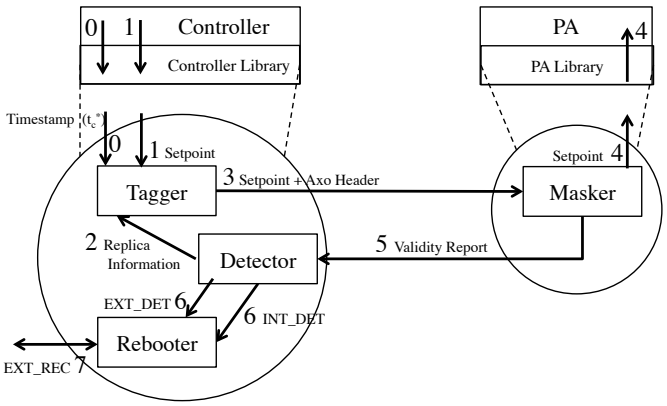


Fig. 1. Axo Architecture

outgoing setpoints with the timestamp of the conception time, the *detector* for detecting faulty replicas, and the *rebooter* for recovering the replicas that were marked faulty. The PA library has the *masker* that discards invalid setpoints. The design of the tagger and masker was introduced in [9].

Here, we introduce the components for fault detection and fault recovery, namely detector and rebooter, respectively. Besides describing their design and operation in Sections IV-C and IV-D, respectively, we restate the design of the tagger and the masker highlighting the extensions added for fault detection and recovery in Section IV-B.

A. Protocol Walkthrough

Each controller replica is assigned a unique replica ID that serves as its identifier for all ensuing Axo-related communication. Furthermore, the code of the controller replicas is instrumented such that they send t_c^*, an estimate of the conception time of the setpoint, to the controller library before beginning the computation of each setpoint. When a controller replica computes and issues a setpoint, it is intercepted by the tagger on this same replica. The tagger uses the last received t_c^* from the controller replica and the replica information received from the detector (elaborated in Section IV-C) to form the 20-byte Axo header. The utility of the different fields in the header are explained in Section IV-B. The tagger sends the setpoint prepended with the Axo header to the masker of the original PA destination.

On receiving the setpoint, the masker uses the timestamp t_c^* in the Axo header to check if the setpoint is valid according to Algorithm 2. The masker forwards the setpoint to the PA if it is valid and discards it otherwise. The masker then sends a validity report to the detectors of all replicas.

As explained in Section I, delay faults cannot be detected using conventional techniques such as time-out or heartbeat between the replicas. Furthermore, feedback from the PA is needed for fault detection, as the validity of a setpoint, and consequently the presence or absence of a delay fault, can only be established at a PA. To this end, the detectors use validity reports to check if the replica is crash or delay faulty, as in Algorithm 3. If the detector on a replica detects the same replica as faulty, then an internal detection message (INT_DET) is sent to the local rebooter. This triggers the rebooter to initiate the recovery of this replica according to

Algorithm 1: Tagger

```
1  t_c^* ← 0;
2  for each message received do
3      if message is timestamp then
4          Update t_c^*;
5      else if message is setpoint SP then
6          Prepend Axo header and send SP to the masker of the PA;
7      else if message is from detector then
8          Update t_d and h;
9      end
10 end
```

Algorithm 5. If the detector detects another replica to be faulty, then an external detection message (EXT_DET) is sent to the local rebooter that then sends an external recovery (EXT_REC) message to the rebooter of the faulty replica.

Axo modifies the setpoints sent by the controllers to the PAs (by adding an Axo header), which requires the PA library on all PAs. This thin layer is transparent to the PA, in particular, any authentication and encryption of the setpoint is left untouched. However, as additional Axo messages are exchanged, these messages need to be authenticated and encrypted in order to avoid spurious recoveries. This can be achieved by using general-purpose security libraries such as datagram transport layer security. Furthermore, Axo only requires the controller to send t_c^* to the tagger. Consequently, it remains agnostic to the changes in the controller and PA.

B. Fault Masking: Tagger & Masker

Together, the tagger and the masker achieve fault masking. The design of the tagger and the masker are shown in Algorithms 1 and 2, respectively. Except for the lines in blue (Algorithm 1 lines 7-9 and Algorithm 2 line 5) that have been added to enable fault detection, the design is same as in [9].

For each setpoint, the tagger (Algorithm 1) receives the timestamp t_c^* from the controller and intercepts the setpoints sent by the controller to the PA. Then, it uses the most recently received t_c^* to populate the 20-byte Axo header that comprises the replica ID (1 byte), the destination port of the original setpoint (2 bytes), the timestamp t_c^* (8 bytes), the timestamp at which the detector was last active t_d (8 bytes), and the replica's health `h` (1 byte). The last two fields are continuously updated at the tagger by messages from the detector, as discussed in Section IV-C. The tagger prepends the Axo header to the setpoint and sends it to the masker of the PA.

At the masker (Algorithm 2), the replica ID is used to identify the sending controller replica, and the destination port number is used to forward the setpoint to the PA. The timestamp t_c^* present in the Axo header is used, along with the validity horizon τ_o, to check if a setpoint is valid. To account for the inaccuracy of time-synchronization protocol δ_s, and 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 an 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).

Lastly, the masker sends a validity report to the detectors of all controller replicas. The fields t_d and `h` present in the Axo header are echoed in the validity report to start detection.

Algorithm 2: Masker

```
1 for each setpoint received do
2     if current time ≤ t_c^* + τ then
3         Remove Axo header and send setpoint to PA;
4     end
5     Send validity report to detectors of controllers;
6 end
```

Algorithm 3: Detector

```
1  initialize DB[myID];
2  for each report VR received do
3      if VR.id ∉ DB or VR.ts > DB[VR.id].ts then
4          if VR.id ∉ DB then
5              create DB[VR.id];
6              DB[VR.id].ts ← VR.ts;
7              DB[VR.id].td ← VR.td;
8              DB[VR.id].nf ← VR.v;
9              DB[VR.id].h ← H_max;
10         else
11             DB[VR.id].h ← min(VR.h, DB[VR.id].h);
12             updateDB(DB[VR.id],VR);
13         end
14         DB[myID].td ← max(DB.ts);
15         Send DB[myID].h, DB[myID].td to tagger;
16         if VR.id ≠ myID and DB[VR.id].h ≤ H_ext then
17             EXT_DET(DB[VR.id].ts,VR.id);
18             delete DB[VR.id];
19         else if DB[myID].h ≤ H_int then
20             INT_DET(DB[myID].ts);
21         end
22         for each id in DB do
23             if max(DB.ts) − DB[id].ts > τ_c or
               max(DB.td) − DB[id].td > τ_c then
24                 if id = myID then
25                     INT_DET(max(DB.ts));
26                 else
27                     EXT_DET(max(DB.ts), id);
28                     delete DB[id];
29                 end
30             end
31         end
32     else if VR.ts = DB[VR.id].ts then
33         DB[VR.id].nf ← DB[VR.id].nf ∨ VR.v;
34     end
35 end
```

C. Fault Detection: Detector

The detection mechanism (Algorithm 3) is triggered at a replica when a validity report ($\mathbf{VR}$) is received from the masker. As no assumptions about synchronicity are made on the communication network, validity reports can be delayed, lost, retransmitted or reordered, making detection challenging.

Each validity report consists of the five following fields: (1) the timestamp t_c of the setpoint ($\mathbf{VR}.ts$), (2) the ID of the replica that issued the setpoint ($\mathbf{VR}.id$), (3) the health of the replica that issued the setpoint, as seen internally by that replica ($\mathbf{VR}.h$), (4) the detector timestamp computed as the highest setpoint timestamp that the detector of the issuing replica had processed ($\mathbf{VR}.td$), and (5) a flag that shows whether the setpoint received at the masker was valid ($\mathbf{VR}.v$).

The detector maintains a database $\mathbf{DB}$ of replicas, indexed by replica IDs. For a replica with ID id, the fields of the database are (1) the highest received setpoint timestamp (t_c) $\mathbf{DB}[id].ts$, (2) the highest received detector timestamp $\mathbf{DB}[id].td$ (3) the replica's health as seen by this replica $\mathbf{DB}[id].h$, and (4) a flag, denoting whether the replica is

Algorithm 4: Function to update detector database

```
1  function updateDB(db,VR)
2  db.ts ← VR.ts;
3  db.td ← VR.td;
4  if db.nf then
5  |  db.h ← α × db.h + (1 − α) × H_max;
6  else
7  |  db.h ← α × db.h − (1 − α) × H_max;
8  end
9  db.nf ← VR.v;
```

considered non-faulty for the setpoint with $t_c = \mathbf{DB}[id].ts$, $\mathbf{DB}[id].nf$. We denote one record of $\mathbf{DB}$ with $\mathbf{db}$.

When the detector boots, it initializes the fields in the database corresponding to its replica's ID (line 1). The health field (h) is set to its maximum value H_{max}, the setpoint timestamp (ts) and the detector timestamp (td) are set to the current time, and the non-faulty flag (nf) field is set to true.

A validity report ($\mathbf{VR}$) is identified by its setpoint timestamp field ($\mathbf{VR}.ts$). This enables aggregating the reports that originated from a single computation. Then, a replica will be considered to have made a faulty computation if and only if all of the reports received corresponding to that computation are invalid (the $\mathbf{VR}.v$ flag is set to false). To achieve this, the detector performs a logical OR of the $\mathbf{VR}.v$ flags received in reports from the same replica with the same setpoint timestamp (lines 32-34). The database is updated (according to Algorithm 4) only after a new report is received, i.e., report with a higher ts field. Consequently, the replica is not penalized when it is actually not delay faulty, but a few of the setpoints that it sent experience a high network-delay.

In Algorithm 4 lines 2-3, the timestamps ts and td are set to the corresponding ones received in the validity report. Lines 4-8 show how the health of a replica is updated. If the replica was delay-faulty in its last computation, i.e., the non-faulty flag set to false, then the updated health is computed by exponentially averaging the old health with a penalty of $-H_{max}$ and a parameter α, else the updated health is computed with a bonus of $+H_{max}$. The exponential averaging serves two purposes: (1) It smoothens out the health and dampens the effect of outliers thereby preventing the triggering of fault recovery due to transient delay faults, and (2) it keeps the health between $-H_{max}$ and $+H_{max}$ thereby avoiding overflow. The value of α ($0 \leq \alpha \leq 1$) represents the weight assigned to the history. Lastly, the non-faulty flag is set according to the valid flag of the newly received validity report (line 9).

In Algorithm 3 at line 14, the detector timestamp of this replica is computed as the highest timestamp processed by this detector. This field is used by other detectors to detect crashes of this detector. The detector timestamp and the health of this replica is sent to the tagger as a part of the Axo header, and is in turn echoed in the validity report so that newly added replicas can learn of existing replicas. Furthermore, when a validity report corresponding to a replica that is already present in the database is received (lines 10-13), the health in the database is updated to the minimum of the existing health and the received health. In this way, a newly added replica learns of the health of existing replicas and instantly joins the detection process, thereby making detection soft state.

A replica is detected as delay faulty (line 16-21) when its health in the database falls below a threshold. For a replica to be detected as delay faulty by its own detector, the threshold H_{int} is used. Whereas, detecting other replicas makes use of the threshold $H_{ext} < H_{int}$. This enables a replica to be detected by its own detector, before it is detected by others. This is particularly useful, as the routine for internal recovery is quicker than that for external recovery (see Section IV-D).

The parameter α and the two thresholds for health, H_{int} and H_{ext}, can be varied to trade-off speed of detection for tolerance of transient delay-faults. A higher α gives less weight to the penalty term causing slower detection, and *vice versa*. On the contrary, a higher H_{int} or H_{ext} reduces the number of invalid setpoints permitted by a replica before being deemed faulty, causing faster detection.

Crash faults are detected as shown in lines 22-31. The detector compares, for each replica in its database, the value of ts with the maximum of all ts's. If the difference is greater than τ_c, then that replica is deemed crash faulty. In this way, a replica is only considered to be crash faulty if it has been inactive for a period of τ_c, *while other replicas have been active*. This relative comparison allows incorporating RTCSs with a non-constant rate of issuing setpoints. A similar comparison is done for td's, to detect crashes in the detector.

D. Fault Recovery: Rebooter

The rebooter, according to Algorithm 5, (1) reboots its own replica when it receives an internal detection (`INT_DET`) message from the local detector, (2) reboots its own replica when it receives an external recovery (`EXT_REC`) messages from another rebooter, and (3) sends `EXT_REC` messages to another rebooter when it receives an external detection (`EXT_DET`) message from the local detector.

The messages received by the rebooter contain a timestamp that corresponds to the report that triggered the detection (see Algorithm 3, lines 17, 20, 25, 27). The rebooter saves this timestamp on disk when it triggers a reboot, and loads it upon booting. The loaded timestamp `lastReboot` is used to order the received messages and to avoid rebooting multiple times for the same detection. This enables the algorithm to deal with message delays and reordering without using consensus. A threshold of τ_{reboot} is also used to avoid multiple reboots in the presence of message losses. For example, a lost report to C_1 might cause replicas C_1 and C_2 to detect a delay fault in C_0 at different times, as C_2 would detect it immediately but C_1 will only detect it after it receives the next report. So, they will send `EXT_REC` messages with different timestamps.

When the rebooter receives `EXT_DET` messages from the local detector, it sends `EXT_REC` messages to the faulty replica, until it receives an `ACK` (Algorithm 5, lines 9-17). This is done at most `maxSend` times, once every T_r. The `ACK`s sent also contain the timestamp of the received `EXT_REC` message, plus the threshold τ_{reboot}. This enables the rebooter issuing the `EXT_REC` messages to differentiate the `ACK`s that correspond to the current exchange from the delayed ones.

From Algorithm 5, note that internal recovery is faster than external recovery, as it is performed locally and does

Algorithm 5: Rebooter

```
1  lastReboot ← loadLastReboot();
2  for each message received do
3      if message is INT_DET(t) then
4          if t > lastReboot + τ_reboot then
5              saveLastReboot(t) ;
6              reboot the replica;
7          end
8      else if message is EXT_DET(t_1, ID) then
9          sentCtr ← 0;
10         while sentCtr < maxSend do
11             send EXT_REC(t_1) to replica ID;
12             sentCtr++;
13             listen for T_r ;
14             if (ACK(t_2) received) and (t_2 ≥ t_1) then
15                 break;
16             end
17         end
18     else if message is EXT_REC(t) then
19         if t > lastReboot + τ_reboot then
20             send ACK(t + τ_reboot) to all replicas;
21             saveLastReboot(t);
22             reboot the replica;
23         else
24             send ACK(lastReboot + τ_reboot);
25         end
26     end
27 end
```

not require communication between the rebooters over the network. This makes it more desirable and is the reason behind setting $H_{ext} < H_{int}$, as mentioned in Section IV-C.

Given that the rebooter needs to respond to remote reboot requests, it needs to be non-susceptible to crash faults. Else, the replica cannot be recovered, as it is not possible to remotely reboot an unresponsive machine. In our analysis, we assume that the part of the rebooter that handles external recovery requests is non-faulty. In our implementation, we achieve this by using a simple heartbeat mechanism on the detector that monitors the rebooter and re-instantiates it in case of faults.

V. Performance Analysis

In this section, we analytically characterize recovery time, i.e., the time taken to detect and recover a faulty replica by Axo. This time depends on the number of replicas, the fault-arrival rate, the frequency of computations by the RTCS, and the delays and losses in the network. We derive the relationship between the time to detect and recover, and the aforementioned parameters. Then, we compare and validate these expressions with results obtained from experiments.

Consider an RTCS consisting of controller replicas that sends setpoints according to a Poisson process of rate λ_n when non-faulty and rate λ_f when faulty. Due to delays induced by a fault, a faulty replica might continue its current computation while other replicas begin a new computation. Hence, $\lambda_n > \lambda_f$. Furthermore, we consider the probability that a computation results in an invalid setpoint, i.e., that the time taken for computation and delivery of a setpoint is greater than τ, to be θ. Additionally, the network between controller replicas and the PAs is considered to have a round-trip time upper bounded by 2Δ and a packet loss probability of p. Lastly, we assume that at least one PA, including its masker, is non-faulty. This assumption is valid as it would not make sense to speak of fault tolerance when all PAs are faulty: the processes would be uncontrollable. Additional non-faulty PAs will only improve the time for detection and recovery.

Additionally, we derive the stationary probability of a replica being in the non-faulty state $\pi_n = \frac{\lambda_f(1-\theta)}{\lambda_f(1-\theta)+\lambda_n\theta}$ and the average rate of sending setpoints $\lambda_0 = \lambda_f\ (1-\pi_n) + \lambda_n\ \pi_n$.

A. Analytical Evaluation of Recovery Time

The exact evaluation of the expression of recovery time appears to be mathematically intractable and is beyond the scope of this paper. Instead, we derive a lower bound and an upper bound on the recovery time. In Section V-B, we validate these bounds by comparing them with experimental results.

Theorem V.1 gives the expressions for the detection and recovery of a delay-faulty controller replica. Theorem V.2 gives the same for a crash-faulty controller replica. The proofs of these theorems can be found in Appendix A and Appendix B, respectively.

Theorem V.1 (Delay-Faulty Controller)**.** *In an RTCS with g controller replicas, if a replica C_0 starts to be delay faulty at time $t = 0$ and remains faulty till time t, then a lower bound $(\mathbb{P}^l_d(t))$ and upper bound $(\mathbb{P}^u_d(t))$ on the probability that it is recovered by time t is given as follows:*

$$\mathbb{P}^l_d(t+2\Delta) = \frac{\gamma^N \pi_n \beta}{\beta+\eta} \times \left[\frac{1}{(\gamma+\eta)^N}\left(1-\frac{\Gamma(N,\gamma t)}{(N-1)!}\right) - \frac{e^{-(\beta+\eta)t}}{(\gamma-\beta)^N}\left(1-\frac{\Gamma(N,(\gamma-\beta)t)}{(N-1)!}\right)\right]$$

$$\mathbb{P}^u_d(t) = 1-(1-\mathbb{P}_1(t))^{g-1}$$

$$\mathbb{P}_1(t) = 1-\frac{\Gamma(N,\gamma t)}{(N-1)!} - \frac{\gamma^N}{(\gamma-\beta)^N} e^{\frac{-(1-p)}{T_r}t}\left[1-\frac{\Gamma(N,(\gamma-\beta)t)}{(N-1)!}\right]$$

$$\beta = (1-p)/T_r, \quad \gamma = \lambda_f(1-p)^2, \quad \eta = \lambda_n\theta$$

$$N = \frac{log(\frac{1}{2}(\frac{H_{ext}}{H_{max}}+1))}{log(\alpha)}, \quad \Gamma(x,s) = \int_s^\infty t^{x-1}e^{-t}dt$$

Theorem V.2 (Crash-Faulty Controller)**.** *In an RTCS with g controller replicas, if a replica C_0 starts to be crash faulty at time $t = 0$ and remains faulty till time t, then a lower bound $(\mathbb{P}^l_c(t))$ and upper bound $(\mathbb{P}^u_c(t))$ on the probability that it is recovered by time t is given as follows:*

$$\mathbb{P}^l_c(t) = \mathbb{P}_2(t-2\Delta)$$

$$\mathbb{P}_2(t) = \begin{cases} \frac{Be^{-(D+G)\frac{\tau_c}{T_r}}}{D+E+G}\left(\frac{A(1-p)T_r}{E+G}e^{-(E+G)\frac{t}{T_r}} + \frac{e^{D\frac{t}{T_r}}}{D+F+G}\right) \\ -\frac{Be^{-(D+G)\frac{\tau_c}{T_r}}}{F+G}\left(\frac{A(1-p)T_r}{D+F+G}e^{-(F+G)\frac{t}{T_r}} + \frac{1}{E+G}\right) & t \le \tau_c \\ e^{-(E+G)\frac{t}{T_r}}\left[\frac{AB(1-p)T_r\left(e^{-(D+G)\frac{\tau_c}{T_r}} - e^{E\frac{\tau_c}{T_r}}\right)}{(E+G)(D+E+G)}\right] & t > \tau_c \\ -e^{-(F+G)\frac{t}{T_r}}\left[\frac{AB(1-p)T_r\left(e^{-(D+G)\frac{\tau_c}{T_r}} - e^{F\frac{\tau_c}{T_r}}\right)}{(F+G)(D+F+G)}\right] \\ -\frac{B\left(e^{-(D+G)\frac{\tau_c}{T_r}} - e^{-G\frac{\tau_c}{T_r}}\right)}{(F+G)(E+G)} \end{cases}$$

$$\mathbb{P}_c^u(t) = 1 - (1 - \mathbb{P}_3(t))^{g-1}$$

$$\mathbb{P}_3(t) = \begin{cases} e^{-D\frac{\tau_c}{T_r}}\left[\frac{EJ}{(D+E)(D+J)}e^{D\frac{t}{T_r}} + \frac{DJ}{(D+E)(J-E)}e^{-E\frac{t}{T_r}} - \frac{DE}{(D+J)(J-E)}e^{-J\frac{t}{T_r}} - 1\right] & t \le \tau_c \\ \frac{DJ}{(J-E)(D+E)}(e^{-D\frac{\tau_c}{T_r}} - e^{E\frac{\tau_c}{T_r}})e^{-E\frac{t}{T_r}} & t > \tau_c \\ -\frac{DE}{(J-E)(D+J)}(e^{-D\frac{\tau_c}{T_r}} - e^{J\frac{\tau_c}{T_r}})e^{-J\frac{t}{T_r}} \\ -(e^{-D\frac{\tau_c}{T_r}} - 1) \end{cases}$$

$$A = \frac{\lambda_0}{\lambda_n(1-p)T_r - 1}, \quad B = (1-p)^3 T_r \lambda_n \pi_n, \quad D = (1-p)^2 T_r \lambda_o$$

$$E = (1-p), \quad F = \lambda_n(1-p)^2 T_r, \quad G = \lambda_n \theta T_r, \quad J = (g-1)\lambda_n(1-p)^2 T_r$$

B. Experimental Validation

1) Experimental setup: The authors in [9] provide an implementation of Axo for fault masking. We extend this implementation with a detector and rebooter for fault detection and fault recovery, respectively. The implementation, which will be made publicly available, is used for the experiments.

We use three replicas ($g = 3$) each with a test controller and the Axo controller library, and one PA with an Axo PA library. The controller replicas run on 64-bit Ubuntu Virtual Machines that are configured with 1 GB RAM using VirtualBox on a Macbook Pro with MacOS 10.10.5, a 2 GHz Intel Core i7 processor and 16 GB RAM. The test controller begins a computation according to a Poisson process with rate $\lambda_n = 1/100\ s^{-1}$ when the controller is non-faulty and $\lambda_f = 1/200\ s^{-1}$ when the controller is faulty. The PA runs on a Lenovo T410 laptop with a 2.67 GHz Intel Core i7 processor with 4 GB RAM running a 64-bit Ubuntu operating system.

As described by Definition 2, a replica is considered delay faulty if the last setpoint it sent to the PA, takes more than τ_o. We have $\tau_o = 17$ ms. Furthermore, the synchronization inaccuracy $\delta_s = 1$ ms (with PTP) and an upper bound on the computation time of the masker $\delta_m = 0.1$ ms. Also, the one-way network latency $\Delta = 2$ ms. Therefore, the replica is considered faulty if its setpoint takes more than $\tau = 14.9$ ms. Lastly, the threshold, after which an inactive controller is considered crash faulty, τ_c is taken as 500 ms.

We study the variation of the distribution of recovery time as a function of the probability of a setpoint being faulty θ. We perform experiments for $p = 0.01$ and $\theta = 0.01, 0.02$. In each experiment, C_0 is configured to start being faulty at a random time and remain so until recovered, whereas C_1 and C_2 follow the parameters of the scenario. Each time C_0 is recovered, the total time elapsed, from the time it started being faulty until the time it was recovered, is recorded as the recovery time.

2) Results: We noticed both from our experiments and from the analytical lower and upper bounds that the packet loss probability p did not have a major effect on the probability of detection. The range of values under consideration is between 0% and 2% loss probability, which is a realistic figure for RTCSs. This shows that the detection and recovery algorithms of Axo are resilient to network losses in this range.

However, the effect of fault rate of replicas (θ) is significant. Figure 2 shows the results of the experimental simulation of a delay-faulty C_0, with $p = 1\%$ and a varying θ. Figure 3 shows

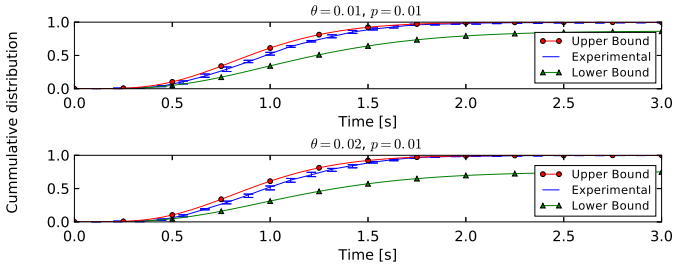


Fig. 2. Time to recover from delay-faults for varying θ

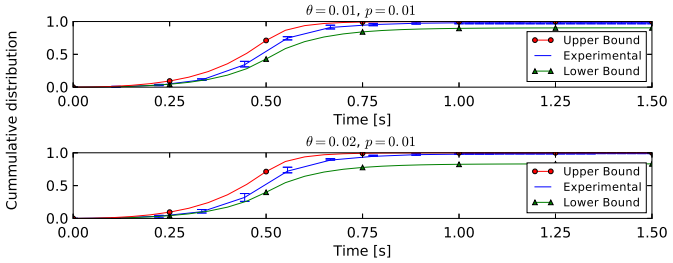


Fig. 3. Time to recover from crash-faults for varying θ

the same for a crash-faulty C_0. In addition to validating the lower and upper bounds, these results show the effect of a higher fault rate on the detection and recovery performance. We also notice that the lower and upper bounds are close to each other, which provides a good estimate of the real values.

VI. Case Studies

A. Stability Analysis of an Inverted Pendulum

In this section, we demonstrate how applying Axo affects the stability of an inverted pendulum system. We use the example in [20], of an inverted pendulum mounted on a motorized cart, in which a Linear Quadratic Regulation (LQR) controller attempts to balance the pendulum by applying a force on the cart.

For the implementation, we use Mininet [21], and we separate the controller from the actuator and have them communicate over a network with loss probability $p = 0.1\%$, and with a one-way delay of 0.5 ms in case of no loss. The controller operates at 100 Hz, resulting in a control cycle of 10 ms. Using Mininet enables us to run the Axo code (mentioned in Section V-B1), rather than simulate it.

Figure 4 shows the step response of the system for different delay profiles of a non-replicated controller, when a step of 1 N is applied as an external force. We see that the pendulum angle (ϕ) and position (x) experience a higher overshoot, and a longer resting time as the mean delay of the controller increases. For delays greater than 20 ms, the system becomes entirely unstable. This shows the real-time requirements of an inverted pendulum system, and hence the applicability of Axo in masking, detecting, and recovering from delay faults.

Next, we perform step response experiments with a replicated controller with two replicas and a bursty delay fault model. In these experiments, the delay is exponentially distributed with a mean of 2 ms in the good state and 80 ms in the bad state; the probability of transition to the bad state

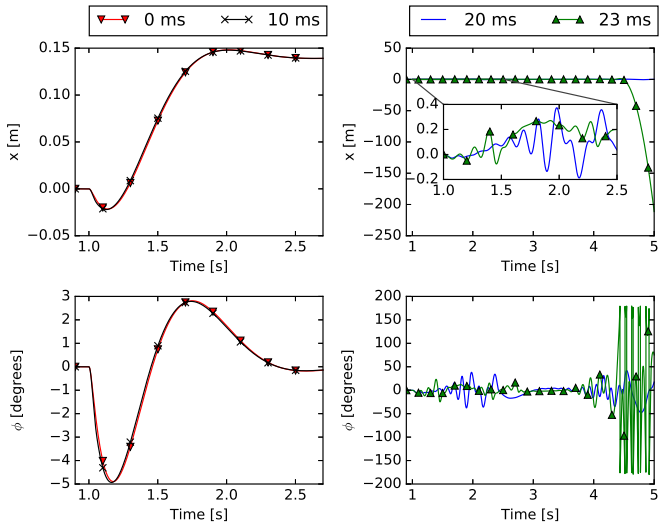


Fig. 4. Step response of the pendulum with a non-replicated controller

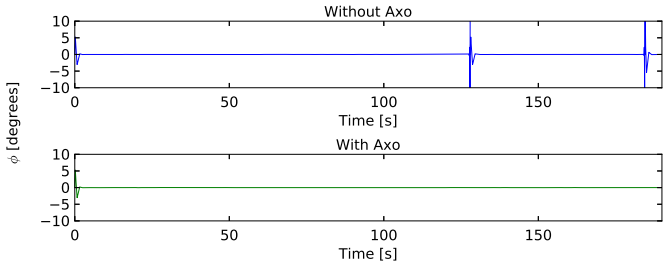


Fig. 5. Stability of the pendulum with a replicated controller

is θ_d, which is varied across several scenarios; and the mean burst length is 20 computation cycles.

We evaluate three metrics: the instability rate, mean time to instability (MTTI), and mean time to failure (MTTF). Instability rate is the fraction of the time the pendulum experiences an overshoot ($\phi > 20^o$ or $x > 0.2$ m), and the MTTI is defined as the mean time until an overshoot occurs. The MTTF is the mean time until the pendulum reaches an angle that the LQR controller is not tuned to handle ($\phi > 35^o$).

Figure 5 shows the additional stability brought about by using Axo for a representative fault-scenario ($\theta_d = 10^{-3}$). Table I shows the computed metrics after a large number of runs. The results are to be interpreted as the mean of an exponential distribution obtained by fitting. The results show that, for all scenarios, Axo improves stability in all the metrics by up to 25x, with the improvement becoming more apparent as the probability of delay faults increases.

B. Commelec Deployment

We deployed Axo for tolerating delay and crash faults in Commelec [1], an RTCS for real-time control of electrical grids. The RTCS is used for controlling a CIGRÉ low-voltage benchmark microgrid on campus. The microgrid consists of two hardware resources, a 25 kW - 25 kWh battery and a 24 kW load of smart heaters. The controller follows a given dispatch signal for active and reactive power in real-time. To this end, it receives the state of resources from the respective

Scenario (θ_d)	Instability (%)		MTTI (s)		MTTF (s)	
	No Axo	Axo	No Axo	Axo	No Axo	Axo
#1: 1E-3	19.56	1.86	57.89	79.16	73.30	118.32
#2: 2E-3	23.93	2.78	25.33	31.70	22.29	47.06
#3: 5E-3	54.04	6.25	7.31	7.42	1.28	32.73

TABLE I
RESULTS OF SELECT SCENARIOS WITH VARYING θ_d

PAs, and it computes and issues setpoints every 100 ms. The controller runs off-the-shelf Scientific Linux version 7.1.

The setpoints have a validity horizon (τ_o) of 10 ms. Consequently, the controller is designed to compute and issue setpoints within 10 ms. First, we measured the computation times for around 10 million measurements. We observe that 32 setpoints (0.00032 %) have a computation time greater than 10 ms. Therefore, we conclude that although very rare, delay faults are observed in real-life deployments of RTCS. Also, delays added due to the communication network further increase the risk of delay faults.

In order to demonstrate the fault tolerance of Axo, we artificially reduce τ_o to 7 ms, increasing the number of faults. We use Axo with two controller replicas (C_1, C_2). The time after which a replica is considered crash faulty (τ_c) is 500 ms and the time between successive recovery messages (T_r) is 1 ms. For time synchronization between the replicas and the PAs, we use PTP that has a synchronization inaccuracy (δ_s) of 1 ms. Lastly, the upper bound on the computation time of the masker (δ_m) is 0.1 ms. This leaves us with $\tau = 4.9$ ms.

Figure 6 shows the empirical cumulative distribution function (CDF) of delays of setpoints sent by C_1 and C_2, measured at the masker. It also shows the effective delay of setpoints at the PAs, after the unsafe ones were discarded by the Axo masker. We observe that, although the setpoints sent by controllers have delays > 4.9 ms ($= \tau$), the setpoints eventually received at the PAs are all valid, i.e., have an end-to-end delay < 4.9 ms < 7 ms ($= \tau_o$), thereby demonstrating the safety property of Axo.

Additionally, the set of controller replicas is said to be available if the PAs receive a setpoint every 100 ms. We find the availability to be 99.86%. In these experiments, the controller replicas were recovered 38 times, thereby demonstrating the importance of fault recovery in providing high-availability.

In order to quantify the overhead in the computation of setpoints by the controller due to the proposed detection and recovery algorithms, we measure the computation time with and without these algorithms. We find that, for non-faulty setpoints, the mean computation time with detection and recovery, measured at 95% confidence level, is 0.893 ± 0.0004 ms, and the same without detection and recovery is 0.274 ± 0.0001 ms. For delay-faulty setpoints, the mean computation time with detection and recovery, measured at 95% confidence level, is 14.18 ± 4.05 ms, and without detection and recovery is 14.45 ± 2.76 ms. We notice that the additional delay in the computation of a setpoint incurred by a non-faulty controller using Axo is sub-millisecond.

VII. CONCLUSION

We present Axo, the first protocol for tolerating delay faults in cb-RTCSs. We describe the masking, detection and recovery

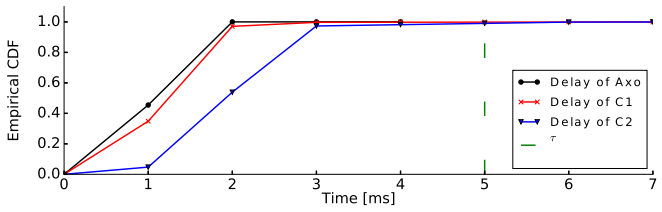


Fig. 6. Safety guarantee of Axo

mechanisms for delay and crash faults. These mechanisms were designed to be soft state to enable the seamless addition of new replicas and removal of faulty replicas. Moreover, Axo was designed to be controller-agnostic, enabling easy deployment to a wide range of existing cb-RTCSs.

We analytically characterize the time to recover a faulty replica by Axo and experimentally validate the expressions. Also, we perform a stability analysis to study the effect of delay faults on the stability of an inverted pendulum system, and we show that, by detecting and recovering from delay faults, Axo improves the stability.

We also use Axo to tolerate faults in an RTCS that controls a CIGRÉ low-voltage benchmark microgrid on campus. We observe that, from over 10 million setpoints, approximately $3.2 \times 10^{-4}\%$ setpoints were faulty. We remark that, although quite rare, such delay faults must be masked from the PAs. Lastly, we demonstrate the safety and availability properties of Axo. Currently, we are in the process of deploying Axo, along with Commelec, in a municipality-wide electrical grid, an experience through which we expect to demonstrate the fault-tolerance properties of Axo in greater detail.

VIII. Acknowledgment

This research was supported by the "SCCER - FURIES" project and the "SNSF - NRP 70" Energy Turnaround project.

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] 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.

[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] G. Andersson, P. Donalek, R. Farmer, N. Hatziargyriou, et al. 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.

[6] Michael Dunn. Toyota's Killer Firmware: Bad Design and its Consequences. *EDN Network, October*, 28, 2013.

[7] Hermann Kopetz and Günther Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

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

[9] 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.

[10] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, et al. *Fundamental Concepts of Dependability*. University of Newcastle upon Tyne, Computing Science, 2001.

[11] John CS Lui, Vishal Misra, and Dan Rubenstein. On the Robustness of Soft State Protocols. In *Network Protocols, 2004. ICNP 2004. Proceedings of the 12th IEEE International Conference on*, pages 50–60. IEEE, 2004.

[12] António Casimiro and Paulo Veríssimo. Timing Failure Detection with a Timely Computing Base. 1999.

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

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

[15] 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.

[16] Kalyanaraman Vaidyanathan and Kishor S Trivedi. A Comprehensive Model for Software Rejuvenation. *Dependable and Secure Computing, IEEE Transactions on*, 2(2):124–137, 2005.

[17] Tomasz Maniak, Chrisina Jayne, Rahat Iqbal, and Faiyaz Doctor. Automated Intelligent System for Sound Signalling Device Quality Assurance. *Information Sciences*, 294:600–611, 2015.

[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.

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

[20] Carnegie Mellon, University of Michigan. Control Tutorials for MATLAB & Simulink. http://ctms.engin.umich.edu/CTMS/index.php?example=InvertedPendulum&section=SystemModeling, 2012. Accessed: 2017-06-22.

[21] Mininet. http://mininet.org/. Accessed 2017-06-22.

Appendix A
Proof of Theorem V.1

Delay-Faulty Controller: *In an RTCS with g controller replicas, if a replica C_0 starts to be delay-faulty at time $t = 0$ and remains faulty till time t, then a lower bound $(\mathbb{P}^l_d(t))$ and upper bound $(\mathbb{P}^u_d(t))$ on the probability that it is recovered by time t is given.*

Proof. First, we derive the probability for $g = 2$.

In a two-replica RTCS, the probability that the delay-faulty replica C_0 issues enough delayed setpoints in $[0, t_1]$ to be detected by the second replica C_1, given that C_1 is non-faulty throughout is computed as follows.

$$\mathbb{P}^*_{det}(t_1) = \mathbb{P}(C_0 \text{ issuing } i \geq N \text{ setpoints in } [0, t_1] \text{ that are received by } C_1) = 1 - \sum_{i=0}^{N-1} \frac{(\lambda_f(1-p)^2 t_1)^i e^{\lambda_f(1-p)^2 t_1}}{i!} \quad (1)$$

Equation 1 is based on the model of the controller described in Section V: it gives the cumulative distribution function (CDF) of a Poisson distribution, where the rate is the rate of a faulty replica issuing setpoints (λ_f) multiplied by the probability of the corresponding report being received $(1-p)^2$. N is the number of consecutive reports, corresponding to delayed setpoints, that are sufficient to detect a delay fault, and can be derived from α, H_{ext}, and H_{max}, from Algorithms 3, 4, as shown in Theorem V.1.

The probability density function (PDF) of the above expression can be obtained by taking the derivative, resulting in the Erlang distribution.

$$\mathbb{P}_{det}(t_1) = \frac{d}{dt_1}\mathbb{P}^*_{det}(t_1) = \frac{(\lambda_f(1-p)^2)^N t_1^{N-1} e^{-\lambda_f(1-p)^2 t_1}}{(N-1)!} \tag{2}$$

In a two-replica RTCS, the probability that C_1 will recover a delay-faulty replica C_0, that was detected as faulty at t_1+d, at $[t_2, t_2+dt]$, given that C_1 is non-faulty throughout, is given by $\mathbb{P}_r(\Delta t)$, where $\Delta t = t_2 - (t_1+d)$.

$$\begin{aligned}\mathbb{P}_r(\Delta t) &= \mathbb{P}(C_0 \text{ receives one reboot message in } [t_2, t_2+dt]) \\ &= \frac{d}{d\Delta t}(1 - e^{\frac{-(1-p)\Delta t}{T_r}}) = \frac{1-p}{T_r} e^{\frac{-(1-p)\Delta t}{T_r}}\end{aligned} \tag{3}$$

Equation 3 can be obtained by modeling the process of receiving reboot messages (Algorithm 5) as a Poisson process of rate $(1-p)/T_r$, where $1-p$ is the probability of receiving a reboot message and $1/T_r$ is the rate at which they are sent. The approximation of the periodic sending process as an exponential one is justified by the low rate, and facilitates the derivation of the above expression.

The probability of C_1 being non-faulty in $[0, \Delta t]$ is:

$$\begin{aligned}\mathbb{P}_{nf}(\Delta t) &= \mathbb{P}(C_1 \text{being non-faulty at } t=0 \text{ and in } (0, \Delta t]) \\ &= \pi_n e^{-\lambda_n \theta \Delta t}\end{aligned} \tag{4}$$

Equation 4 considers the fault model of a controller replica, where π_n is the stationary probability of being in a non-faulty state. This is multiplied by the probability of not transitioning to the faulty state within a period of Δt.

Using Equations 2, 3, 4, we can define the lower and upper bounds on recovering a delay-faulty controller replica.

For a lower bound, we consider a two-replica system, the worst-case network delay, and that a faulty replica cannot help in detection and recovery. Increasing the number of replicas, decreasing the network delay, or considering the cases in which faulty replicas can take part in detection or recovery, will increase the probability. Therefore, the lower bound is justified. It is given as follows:

$$\mathbb{P}^l_d(t) = \int_{t_1=0}^{t-2\Delta} \mathbb{P}_{det}(t_1) \int_{t_2=t_1+2\Delta}^{t} \mathbb{P}_r(t_2 - t_1 - 2\Delta)\mathbb{P}_{nf}(t_2 - 2\Delta) dt_2 dt_1$$

Note that the lower bound always considers two-replica RTCSs regardless of g.

For an upper bound, we relax the condition of dependence between replicas: we consider that each additional replica in the system can detect and recover C_0 independently. We also consider that all these replicas are always non-faulty, and that the network has zero delay. These relaxations always result in an increase to the actual probability. Therefore, the upper bound is justified. It is given as follows:

$$\mathbb{P}^u_d(t) = 1 - (1 - \mathbb{P}_1(t))^{g-1}$$

$$\mathbb{P}_1(t) = \int_{t_1=0}^{t} \mathbb{P}_{det}(t_1) \int_{t_2=t_1}^{t} \mathbb{P}_{rec}(t_2 - t_1) dt_2 dt_1$$

The derivation of $\mathbb{P}^l_d(t)$ and $\mathbb{P}^u_d(t)$ results in the statement of the Theorem.

□

APPENDIX B
PROOF OF THEOREM V.2

Crash-Faulty Controller: *In an RTCS with g controller replicas, if a replica C_0 starts to be crash-faulty at time $t=0$ and remains faulty till time t, then a lower bound $(\mathbb{P}^l_c(t))$ and upper bound $(\mathbb{P}^u_c(t))$ on the probability that it is recovered by time t is given.*

Proof. We first derive the following probabilities.

We define the notion of *awareness*, where a replica C_i is aware of replica C_0 at t_a, if the detector database at C_i contains an entry for C_0 at t_a. This condition is satisfied if C_0 sends a setpoint with a conception time $t_0 > t_a - \tau_c$, the report of which is received by C_i. The probability of such an event, given that C_0 conceives another setpoint at t_a and that C_i is non-faulty throughout, is given as $\mathbb{P}_a(\Delta t)$, where $\Delta t = t_0 - t_a$:

$$\begin{aligned}\mathbb{P}_a(\Delta t) &= \mathbb{P}(C_0 \text{ issues a setpoint of conception time } t_0 - \Delta t, \\ &\quad \text{that is received by } C_i) \\ &= \lambda_0(1-p)^2 e^{-\lambda_0(1-p)^2 \Delta t}\end{aligned} \tag{5}$$

Equation 5 considers the controller model from Section V, and uses the time-reversal property of Poisson processes.

We now consider a g-replica RTCS, in which C_0 crashes at t_0, and the other $g-1$ replicas are assumed to be able to detect this independently, and are all non-faulty throughout. For this, each controller can be modeled as receiving setpoints at a rate of $(g-1)\lambda_n(1-p)^2$. The network is considered to have a fixed one-way delay of d for packets that are not dropped. Under such conditions, the probability of a replica C_i detecting C_0 as crash faulty in the interval $[t_1, t_1+dt]$, given the above conditions and that C_i was aware of C_0, is given as $\mathbb{P}_c(\Delta t, g)$, where $\Delta t = t_1 - t_0 - d$.

$$\begin{aligned}&\mathbb{P}_c(\Delta t, g) = \mathbb{P}(C_j \neq C_0 \text{ conceivs a setpoint at } t_1 - d, \\ &\text{the report of which is received by } C_i \text{ at } t_1) \\ &= \begin{cases} 0 & \Delta t < \tau_c \\ (g-1)\lambda_n(1-p)^2 e^{-(g-1)\lambda_n(1-p)^2(\Delta t - \tau_c)} & \Delta t \geq \tau_c \end{cases}\end{aligned} \tag{6}$$

Note that the above expression is an upper bound when $g > 2$, but is exact when $g = 2$, since the condition of independence is not required when there is only one replica participating in detection.

In what follows, we derive a lower bound and upper bound on the probability of recovering from a crash fault using Equations 5 and 6. We will also use $\mathbb{P}_r$ and $\mathbb{P}_{nf}$ from Equations 3 and 4, respectively.

The conditions for lower and upper bound are similar to those presented in Appendix A. For a lower bound, we consider a two-replica RTCS, the worst-case network delay, and that a faulty replica cannot help in detection and recovery.

$$\mathbb{P}_c^l(t) = \int_{t_0=\max(0,\tau_c-t)}^{\tau_c} \int_{t_1=\tau_c-t_0}^{t-2\Delta} \int_{t_2=t_1+2\Delta}^{t} \mathbb{P}_a(t_0)\mathbb{P}_c(t1+t_0,2)\times \mathbb{P}_r(t_2-(t_1+2\Delta))\mathbb{P}_{nf}(t_2+(t_0+2\Delta))dt_2dt_1dt_0$$

For an upper bound, we relax the condition of dependence between replicas: we consider that each additional replica in the system can detect and recover C_0 independently. We also consider that all these replicas are always non-faulty, and that the network has zero delay. These relaxations always result in an increase to the actual probability. Therefore, the upper bound is justified. It is given as follows:

$$\mathbb{P}_c^u(t) = 1-(1-\mathbb{P}_3(t))^{g-1}$$

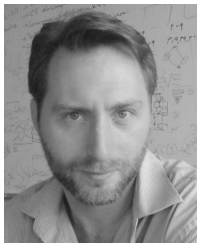
$$\mathbb{P}_3(t) = \int_{t_0=\max(0,\tau_c-t)}^{\tau_c} \int_{t_1=\tau_c-t_0}^{t} \int_{t_2=t_1}^{t} \mathbb{P}_a(t_0)\mathbb{P}_c(t1+t_0,g)\times \mathbb{P}_r(t_2-t_1)dt_2dt_1dt_0$$

The derivation of $\mathbb{P}_c^l(t)$ and $\mathbb{P}_c^u(t)$ results in the statement of the Theorem.

□

Maaz Mohiuddin is a PhD student at the School of Computer Science and Communication Systems of EPFL since January 2014. He works under the supervision of Professor Jean-Yves Le Boudec. He grew up in Hyderabad, India and attended the Indian Institute of Technology Hyderabad, where he completed his undergraduate in Electrical Engineering with a minor in Computer Science in 2013. His main research interests are fault-tolerance in real-time systems and reliable communication for smart grid networks. He has co-authored a paper presented at WFCS 2015 that won the best paper award.

Wajeb Saab is a PhD student at the School of Computer Science and Communication Systems of EPFL. He grew up in Lebanon, where he obtained his undergraduate degree in Computer and Communications Engineering in 2014. He moved to Switzerland in September 2014, where he started working towards his PhD under the supervision of Professor Jean-Yves Le Boudec. His main research interests are reliable and robust real-time control systems.



Simon Bliudze holds an MSc in Mathematics from St. Petersburg State University (Russia, 1998), an MSc in Computer Science from Université Paris 6 (France, 2001) and a PhD in Computer Science from École Polytechnique (France, 2006). He has spent two years at Verimag (Grenoble, France) as a post-doc with Joseph Sifakis working on formal semantics for the BIP component framework. After three years as a research engineer at CEA Saclay (France) and six years as a scientific collaborator at EPFL (Lausanne, Switzerland), he has recently joined INRIA Lille - Nord Europe (France).

Jean-Yves Le Boudec is professor at EPFL and fellow of the IEEE. He graduated from Ecole Normale Supérieure de Saint-Cloud, Paris, where he obtained the Agrégation in Mathematics in 1980 and received his doctorate in 1984 from the University of Rennes, France. From 1984 to 1987 he was with INSA/IRISA, Rennes. In 1987 he joined Bell Northern Research, Ottawa, Canada, as a member of scientific staff in the Network and Product Traffic Design Department. In 1988, he joined the IBM Zurich Research Laboratory where he was manager of the Customer Premises Network Department. In 1994 he became associate professor at EPFL. His interests are in the performance and architecture of communication systems and smart grids. He co-authored a book on network calculus, which forms a foundation to many traffic control concepts in the internet, an introductory textbook on Information Sciences, and is the author of the book "Performance Evaluation".