Reliable and Robust Cyber-Physical Systems for Real-Time Control of Electric Grids

Thèse N° 9161

Présentée le 3 mai 2019

à la Faculté informatique et communications
Laboratoire pour les communications informatiques et leurs applications 2
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Wajeb SAAB

Acceptée sur proposition du jury

Prof. P. Thiran, président du jury
Prof. J.-Y. Le Boudec, directeur de thèse
Dr S. Bliudze, rapporteur
Dr F. Cadoux, rapporteur
Prof. M. Paolone, rapporteur

2019

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Throughout human history,
we have been dependent on machines to survive.
Fate, it seems, is not without a sense of irony.
— Morpheus, The Matrix

To my parents...

Acknowledgements

This thesis would not have been possible without the vision, supervision, guidance, and motivation of Professor Jean-Yves Le Boudec. I cannot imagine a better supervisor and role model. He was always there when we needed him, whether the situation required a three-hour meeting or simply his trust in me to choose my own research direction. His influence on me will go far beyond my PhD journey.

I would like to express my deepest gratitude to Dr. Simon Bliudze, my unofficial co-supervisor, without whom I would not have made it to this stage. I have always, and will always, hold you in the highest of regards. I would also like to thank Professor Mario Paolone, Dr. Florent Cadoux, and Professor Patrick Thiran, for being part of my PhD committee, providing valuable feedback and interesting discussions.

On the topic of interesting discussions, nothing has been more fulfilling or mentally stimulating than the numerous discussions with Dr. Maaz Mohiuddin and Professor Lorenzo Reyes. Lorenzo has constantly been the go-to person for matters of electric grids, on which I am no expert. His future PhD students will be very lucky to have him as a supervisor. Maaz has been with me throughout this journey, both as a research collaborator and a best friend. The past four years would not have been nearly as enjoyable without him.

I was lucky to be a part of the harmonious collaboration between LCA2 and DESL, in which I had the privilege of working with extraordinary people. A special mention goes to Roman for his collaboration, for the fun road trip, and for the Matryoshka.

A big part of ensuring the smooth and seamless experience of PhD students in LCA2 relies on the exemplary dedication of its backroom staff. I would like to thank the secretaries Patricia and Angela, the system administrators, Yves and Marc-André, and our amazing editor, Holly, who has read my thesis with more care than anyone else.

I would like to thank Hussein Kassir for following me everywhere I go, for the hikes, and for the board game nights with Taha, Mira, Mahdi, Hala, and Hajj. You guys were my home away from home.

My best friends away from Switzerland might not have been here, but they were the ones that kept me going. Thank you Wael, Ryan, Reem, Abed, Nadim, Tala, Lama, Raed, Rawad, Razan, and Abdullah. Thank you for your friendship, for your support, and for all that is to come. Thank you Haya for celebrating every single section of this thesis with me, and for keeping me sane throughout the process.

Acknowledgements

Last, but not least, I am nothing if not for the support of my family. Thank you Mom, Dad, and Manal for always believing in me. Thank you Jihad, Wida, Nader, Vida, Wissam, Ziad, Aida, and Amal for being in my life.

Lausanne, 15th of March 2019 W. S.

Abstract

Real-time control of electric grids is a novel approach to handling the increasing penetration of distributed and volatile energy generation brought about by renewables. Such control occurs in cyber-physical systems (CPSs), in which software agents maintain safe and optimal grid operation by exchanging messages over a communication network.

We focus on CPSs with a centralized controller that receives measurements from the various resources in the grid, performs real-time computations, and issues setpoints. Long-term deployment of such CPSs makes them susceptible to software agent faults, such as crashes and delays of controllers and unresponsiveness of resources, and to communication network faults, such as packet losses, delays, and reordering. CPS controllers must provide correct control in the presence of external non-idealities, *i.e.,* be robust, and in the presence of controller faults, *i.e.,* be reliable. In this thesis, we design, test, and deploy solutions that achieve these goals for real-time CPSs.

We begin by abstracting a CPS for electric grids into four layers: the control layer, the network layer, the sensing and actuation layer, and the physical layer. Then, we provide a model for the components in each layer, and for the interactions among them. This enables us to formally define the properties required for reliable and robust CPSs.

We propose two mechanisms — Robuster and intentionality clocks — for making a single controller robust to unresponsive resources and non-ideal network conditions. These mechanisms enable the controller to compute and issue setpoints even when some measurements are missing, rather than to have to wait for measurements from all resources. We show that our proposed mechanisms guarantee grid safety and outperform state-of-the-art alternatives.

Then, we propose Axo: a framework for crash- and delay-fault tolerance via active replication of the controller. Axo ensures that faults in the controller replicas are masked from the resources, and it provides a mechanism for detecting and recovering faulty replicas. We prove the reliable validity and availability guarantees of Axo and derive the bounds on its detection and recovery time. We showcase the benefits of Axo via a stability analysis of an inverted pendulum system.

Solutions based on active replication must guarantee that the replicas issue consistent setpoints. Traditional consensus-based schemes for achieving this are not suitable for real-time CPSs, as they incur high latency and low availability. We propose Quarts,

an agreement mechanism that guarantees consistency and a low bounded latency-overhead. We show, via extensive simulations, that Quarts provides an availability at least an order of magnitude higher than state-of-the-art solutions.

In order to test the effect of our proposed solutions on electric grids, we developed T-RECS, a virtual commissioning tool for software-based control of electric grids. T-RECS enables us to test the proper functioning of the software agents both in ideal and faulty conditions. This provides insight into the effect of faults on the grid and helps us to evaluate the impact of our reliability solutions.

We show how our proposed solutions fit together, and that they can be used to design a reliable and robust CPS for real-time control of electric grids. To this end, we study a CPS with COMMELEC, a real-time control framework for electric grids via explicit power setpoints. We analyze the reliability issues of this CPS, and show how our solutions can be used to improve its reliability and robustness. We test our proposed solutions in T-RECS, and finally deploy them in a real-scale low-voltage CIGRÉ benchmark microgrid for experimental validation.

Key words: reliability, real-time control, cyber-physical systems, centralized control, robust, electric grids, latency, non-ideal conditions, mission-critical, availability, delay faults, crash faults, lossy communication network, consistency, agreement, consensus, replication, redundancy, fault masking, fault detection, fault recovery, software agents, commercial off-the-shelf, islanding, microgrids, smartgrids, grid safety.

Résumé

Le contrôle en temps réel des réseaux électriques est une nouvelle approche pour gérer l'augmentation de la génération distribuée et volatile d'énergie due aux énergies renouvelables. Ce contrôle s'effectue dans des systèmes cyber-physiques ("Cyber-Physical System", CPS), dans lesquels des agents logiciels garantissent une exécution sûre et optimale du réseau en échangeant des messages par un réseau de communication.

Nous nous concentrons sur les CPS avec un contrôleur centralisé qui reçoit des mesures de la part des ressources dans le réseau, effectue des calculs en temps réel, et émet des valeurs de consigne. Le déploiement à long terme de ces CPS les rend vulnérables aux erreurs logicielles des agents, telles que des arrêts, des délais, et des manques de réponses, ainsi qu'aux erreurs dans les réseaux de communication, telles que des pertes de paquets, des délais, et des changements dans l'ordre des paquets. Les contrôleurs des CPS doivent fournir des instructions correctes en la présence de non-idéalités externes, c'est-à-dire être robustes, ainsi qu'en présence d'erreurs des contrôleurs, c'est-à-dire être fiables. Dans cette thèse, nous concevons, testons et déployons des solutions qui atteignent ces objectifs pour des CPS en temps réel.

Nous commençons par abstraire un CPS pour réseaux électriques en quatre couches : la couche de contrôle, la couche de réseau, la couche de détection et d'actionnement, ainsi que la couche physique. Ensuite, nous fournissons un modèle pour les composants de chaque couche, ainsi que pour leurs interactions. Ceci nous permet de définir formellement les propriétés requises pour des CPS robustes et fiables.

Nous proposons deux mécanismes — Robuster et les horloges d'intentionnalité — pour rendre un seul contrôleur robuste aux ressources qui ne réagissent pas à temps ainsi qu'aux conditions de réseau non-idéales. Ces mécanismes permettent au contrôleur de calculer et d'envoyer des valeurs de consignes même quand certaines mesures manquent, plutôt que d'attendre des mesures de toutes les ressources. Nous démontrons que les mécanismes que nous proposons garantissent la sûreté du réseau électrique et surpassent les alternatives de l'état de l'art.

Puis nous proposons Axo : une solution pour la tolérance d'arrêts et de délais par la réplication active du contrôleur. Axo garantit que les erreurs dans les répliques du contrôleur sont invisibles pour les ressources, et fournit un mécanisme pour détecter et corriger les répliques fautives. Nous prouvons les garanties de validité et la disponibilité robustes d'Axo et dérivons les bornes pour son temps de détection et de correction. Nous démontrons les avantages d'Axo par une analyse de la stabilité d'un système de

pendule inversé.

Les solutions basées sur la réplication active doivent garantir que les répliques émettent des valeurs de consignes cohérentes. Les approches traditionnelles basées sur le consensus pour atteindre cet objectif ne sont pas adaptées aux CPS en temps réel, car elles causent de hautes latences et une disponibilité faible. Nous proposons Quarts, un mécanisme de consensus qui garantit la cohérence et une faible augmentation de latence. Nous démontrons, par des simulations approfondies, que Quarts fournit une disponibilité au moins un ordre de magnitude plus élevée que les solutions de l'état de l'art.

Pour tester l'effet des solutions que nous proposons sur des réseaux électriques, nous avons développé T-RECS, un outil de mise en service virtuelle pour le contrôle logiciel de réseaux électriques. T-RECS nous permet de tester le fonctionnement des agents logiciels dans des conditions idéales ainsi que dans des conditions problématiques. Ceci nous fournit un aperçu des effets des erreurs dans le réseau, et nous aide à évaluer l'impact de nos solutions pour la fiabilité.

Nous montrons comment les solutions que nous proposons fonctionnent ensemble, et montrons qu'elles peuvent être utilisées pour concevoir un CPS fiable et robuste pour le contrôle en temps réel de réseaux électriques. à cet effet, nous étudions un CPS avec COMMELEC, une solution pour le contrôle en temps réel de réseaux électriques utilisant des valeurs de consignes explicites. Nous analysons les problèmes de fiabilité de ce CPS, et montrons comment nos solutions peuvent être utilisées pour améliorer sa fiabilité et sa robustesse. Nous testons nos solutions avec T-RECS, et les déployons dans un micro-réseau électrique CIGRE de faible voltage à l'échelle réelle afin d'effectuer une validation empirique.

Mots clés : fiabilité, contrôle en temps réel, systèmes cyber-physiques, contrôle centralisé, robuste, réseaux électriques, latence, conditions non-idéales, critique, disponibilité, erreurs de délai, erreurs d'arrêt, réseaux de communication avec pertes, cohérence, consensus, réplication, redondance, masquage d'erreurs, détection d'erreurs, récupération, agents logiciels, produit informatique COTS, isolation, micro-réseaux, réseaux intelligents, sécurité du réseau.

List of Abbreviations

API	Application Programming Interface
BFT	Byzantine Fault-Tolerance
CDF	Cumulative Distribution Function
COTS	Commercial Off-the-Shelf
CPS	Cyber-Physical System
ECDF	Empirical Cumulative Distribution Function
EV	Electric Vehicle
GA	Grid Agent
GPS	Global Positioning System
IP	Internet Protocol
iPRP	IP-friendly Parallel Redundancy Protocol
LQR	Linear Quadratic Regulator
MAC	Media Access Control
MPC	Model Predictive Control
MTTF	Mean Time to Failure
MTTI	Mean Time to Instability
MTTR	Mean Time to Repair
NTP	Network Time Protocol
PC	Personal Computer
PCC	Point of Common Coupling
PDC	Phasor Data Concentrator
PDF	Probability Density Function
PLC	Programmable Logic Controller
PMU	Phasor Measurement Unit
PRP	Parallel Redundancy Protocol
PTP	Precision Time Protocol
PV	Photovoltaic
QUIC	Quick UDP Internet Connections[1]

[1] QUIC is no longer an acronym in IETF QUIC under standardization since May 2018

RA	Resource Agent
RAM	Random Access Memory
RMSE	Root Mean Square Error
RTT	Round-Trip Time
SE	State Estimator
SoC	State of Charge
TCB	Timely Computing Base
TCP	Transmission Control Protocol
TTA	Time-Triggered Architecture
UDP	User Datagram Protocol

List of Notations

Advertisement-related notations

$\mathcal{F}$	feasibility set of a resource
$\mathcal{F}_r$	feasibility set in advertisement of round r
$\mathcal{F}^n$	long-term feasibility set, valid for n rounds
$\mathcal{U}$	uncertainty function of a resource
$\mathcal{U}_r$	uncertainty function in advertisement of round r
$\mathcal{U}^n$	long-term uncertainty function, valid for n rounds
T_{val}	validity horizon of an advertisement

Setpoint-related notations

$\mathbf{X}$	vector of setpoints
$\mathbf{X}_r$	vector of setpoints in round r; elements are x_r
$\hat{x}_r$	actual implemented setpoint by resource in round r
$\mathbf{I}_r(\mathbf{X}_r)$	set of vectors of actual setpoint implementations in round r, considering uncertainties of resources in round $r-1$
$\mathbf{J}_r(\mathbf{X}_r)$	set of vectors of actual setpoint implementations in round r, considering uncertainties of resources in round $r-1$ and possibility of missing setpoints
$\mathbf{T}_v$	vector of validity times of setpoints; elements are t_v
T_{ctrl}	control period: upper bound duration of control round
τ	validity horizon of a setpoint

Miscellaneous

δ_i	upper bound on implementation time by an RA
δ_n	upper bound on one-way network latency
p	probability of packet loss in a network

Contents

List of Figures

List of Tables

1 Introduction

Anything that can go wrong, will go wrong.
— Murphy's Law

A Cyber-Physical System (CPS) is an integration of the computation, networking, and physical domains [1]. It is a system in which software agents monitor and control physical processes and resources. Examples of CPSs can be found in autonomous vehicles [2, 3], robotics [4, 5], process control automation [6, 7], and the focus of this thesis, smartgrids [8, 9, 10].

Smartgrids are CPSs in which the state of the grid and the various electric resources is monitored and controlled in order to maintain grid safety. As these CPSs interact with the physical world, they must adhere to a set of requirements different than traditional computing systems such as databases or datacenters for Web applications.

In contrast with the requirements of web applications mentioned above, CPSs must react quickly in order to account for changes in grid conditions, as the state of the grid is constantly evolving. A response time that is sufficiently fast, compared to the rate of change of the physical process, is referred to as *real-time* [11]. Hence, latency is a key performance metric.

Furthermore, a failure in electric-grid control can have serious safety and economic consequences [12]. Hence, a CPS must handle faults both in the physical process (*i.e.*, the grid and the resources) and in the computation and networking infrastructure. Although such faults are typically rare, CPSs are deployed for long periods, making the occurrence of faults in the lifetime of a CPS inevitable. The inevitability of faults with serious implications creates a pressing demand for reliable and robust CPSs.

In this thesis, we define the requirements of reliability and robustness for real-time CPSs and propose methods to achieve these requirements. We then test our proposed methods through real-life deployments of a CPS for real-time control of electric grids.

1.1 Background

Traditional electric grids are divided into three parts: generation, transmission, and distribution. Under this neat division, energy production takes place in the generation grid and energy consumption in the distribution grid. These grids are known for their high inertia, as consumption in the distribution grids slowly changes, and as production is likewise slowly changed to match it.

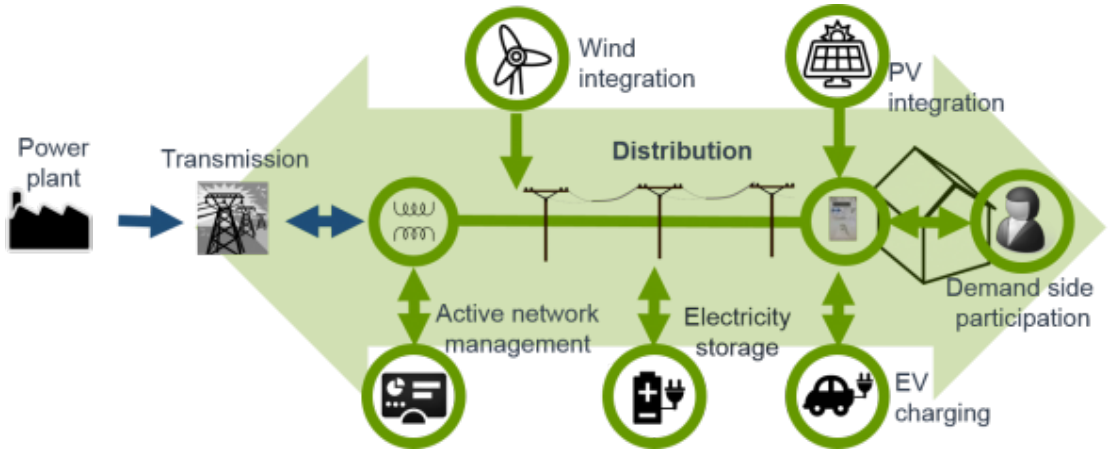


Figure 1.1 – Emerging grids

Emerging grids offer a different story. Renewable penetration, such as rooftop solar, has moved some production into the distribution grids. Furthermore, these new sources of energy are intermittent and have low inertia. For example, a passing cloud could significantly drop the power produced by a solar panel in less than a second. Consumption is also becoming less predictable and more volatile, for example, the emergence of electric vehicle (EV) charging stations in distribution grids [13, 14]. Therefore, to avoid exporting the issues to the upper-level grids, both production and consumption must now be predicted and matched at the distribution level.

This is where CPSs come into play. A network of distributed computers can be used to monitor and predict production and consumption patterns for resources in the grid. The predictions can then be used to compute a safe and desirable point of operation for the grid. Consider, for example, the grid in Figure 1.1. Assume that, at a given point, the solar panel is producing 250 kW to match the consumption of the building and the EV charging station. A change in weather conditions is about to drop the solar production to 200 kW, thus creating a deficit of 50 kW in the grid, which would create voltage violations. A CPS monitoring and controlling this grid can mitigate the imminent drop by curtailing the EV charging power accordingly. Alternatively, if the EV charging station is uncontrollable or if it is undesirable to curtail it, then an alternative source of energy, such as the battery storage device, could be utilized to close the deficit.

1.1.1 Real-Time Control of Electric Grids

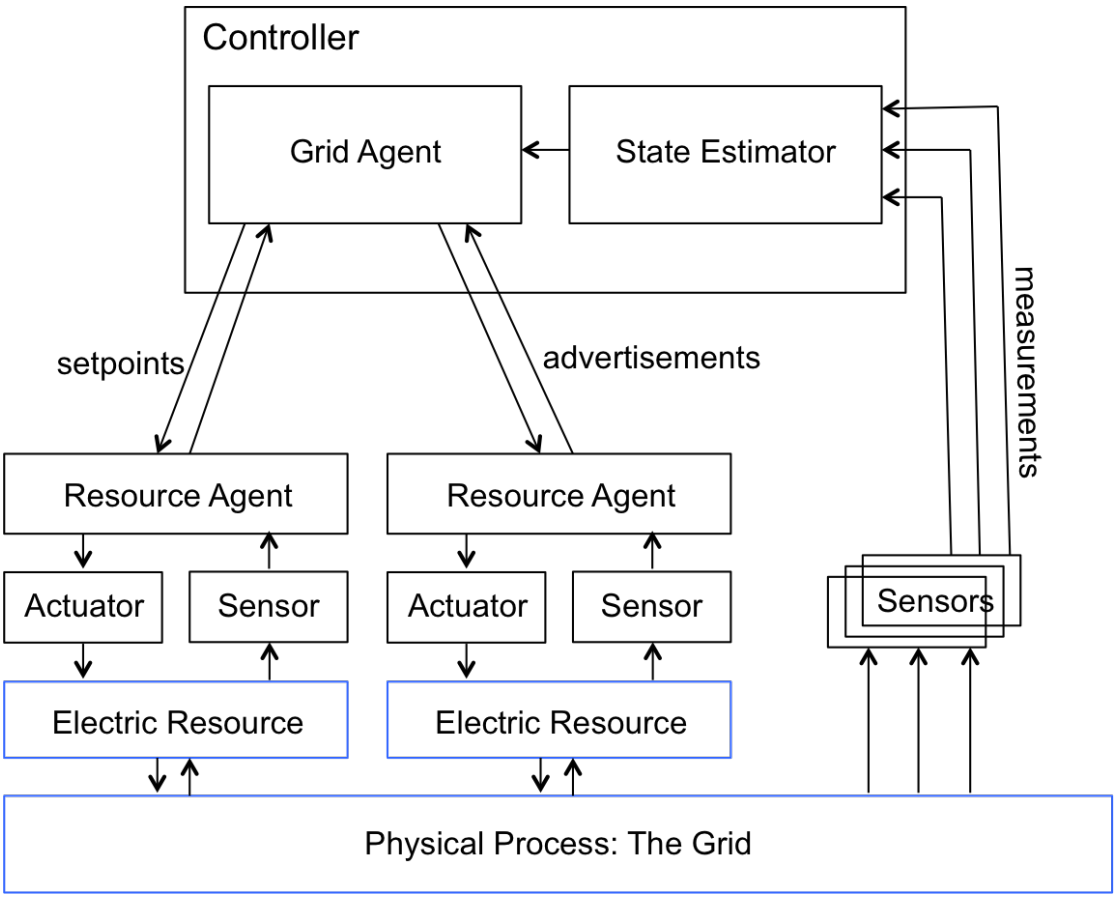


Figure 1.2 – Architecture of a CPS for real-time control of electric grids

We consider CPSs for real-time control of electric grids, with an architecture as depicted in Figure 1.2. The physical components are shown in blue and the cyber components in black. We start at the lowest level, in which the grid consists of the buses, lines, transformers, and breakers. Electric resources, such as batteries, solar panels, heaters, and residential appliances are located at different buses in the grid.

A *resource agent* (RA) monitors and possibly controls each of these resources, through its corresponding sensors and actuators. RAs communicate information about their resources as *advertisements* to a *grid agent* (GA). Additionally, sensors at various locations in the grid monitor quantities such as bus voltages and line currents. This information is sent as *measurements* to a *state estimator* (SE) that computes and sends an estimate of the entire grid state to the GA. The GA uses its inputs to compute and issue *setpoints* to the RAs, which in turn instruct the resources to implement these setpoints, if applicable.

The main use-case for real-time control is maintaining grid safety, *i.e.,* keeping the

bus voltages and line currents within given thresholds. The standards specifying such thresholds permit a maximum violation time of 500 ms [15, 16, 17]. Therefore, such a task requires low response times, especially in grids with low inertia.

We note that the discussion on grid safety, in this thesis, refers to the consequences of violating voltage and current thresholds. Such consequences involve invoking fail-safe mechanisms such as tripping of breakers, which is deemed as a failure of the control. The issues tackled, therefore, are less about ensuring quality of service, and more about protecting against non-routine system failure. While traditional grids might not suffer from such issues, emergent grids relying on real-time control require this protection.

In addition to maintaining grid safety, several ancillary services can be performed with real-time control. This includes following a dispatch signal thus acting as a virtual power plant to an upper-level grid [18], providing frequency support [19], localization of line faults [20], and enabling demand response [21].

We highlight an emerging use-case for real-time control, namely, *islanding* [22, 23]. Islanding is the process of disconnecting a microgrid from an upper-level grid, either intentionally for grid maintenance or unintentionally due to some natural disaster that causes a far reaching failure in the main grid [24]. In addition to the disconnection maneuver, the microgrid must also be operated in islanded mode and, possibly, reconnected to the main grid when the condition for islanding no longer holds. Grid safety must be maintained during such mission-critical operations.

1.1.2 Cyber-Physical Systems in the Wild

The aforementioned operations typically require performing complex computations at high levels of abstraction. These include manipulation of high-dimensional objects [23, 25], projection and gradient descent [8], non-linear optimization [26], and state-estimation [27, 28]. This complexity precludes the use of low-level computing devices, such as PLCs [29], traditionally used for non-complex and non-real-time electric grid control [30, 31].

This paradigm shift requires the use of commercial-off-the-shelf (COTS) components [32], in order to leverage their powerful computing capabilities. COTS refers to both hardware and software. COTS hardware includes computers such as the NI CompactRIO [33], B&R Automation PC [34], and the PINE ROCKPro64 [35]. The term COTS software refers to both the commodity operating system run by the COTS hardware [36], and the high-level programming languages (such as C/C++) used in developing such applications, typically using third-party software libraries.

COTS offers powerful computing, large memory, high flexibility and adaptability,

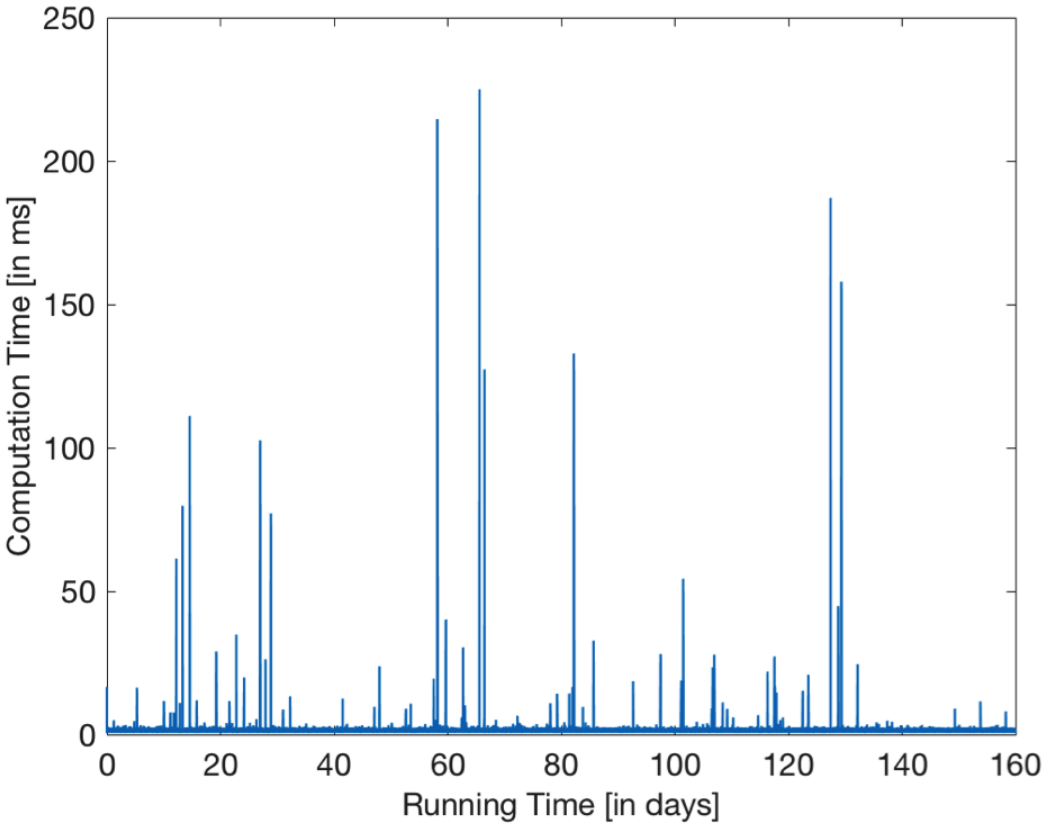


Figure 1.3 – Response time of a grid controller deployed on a CompactRIO as a function of running time

and ease of maintenance. However, these advantages are to be balanced against the lack of predictability [32]. Whereas PLCs offer strict real-time guarantees, COTS components provide best-effort performance.

Figure 1.3 shows the computation time of an NI CompactRIO device performing load flow operations [37], the deadline of which is 40 ms. We observe that although such an operation takes less than 1 ms in most cases, its duration might rise to over 200 ms. The figure includes 140 million samples, of which the median is 0.47 ms, the 99.99^{th} percentile is 1.89 ms, and the 99.999^{th} percentile is 205 ms. Although these high response times only occur at the tail of the distribution, the unpredictability of the performance of COTS components poses reliability concerns.

The above discussion is an example of delays that affect COTS-based devices. In addition to this type of fault, these devices might crash due to a software bug, an OS glitch, or a hardware failure. Moreover, CPSs rely on IP-based communication networks that are built using COTS hardware. These networks might not guarantee correct packet delivery within a given time [38].

Lastly, CPSs using COTS components are also susceptible to Byzantine faults [39].

These are faults that produce erroneous computation results or message contamination. Such faults might be unintentional, resulting from a software bug, or intentional, resulting from a compromised component in a security attack [40]. We do not consider such faults in this thesis, we instead focus on the aforementioned crash and delay faults. However, advances in correct-by-construction software design [41], formal verification techniques [42], as well as smartgrid security [43], provide orthogonal methods for eliminating the root causes of Byzantine faults.

1.1.3 Reliability and Robustness

In this thesis, we consider reliability and robustness from the point of view of the central controller. As we will see, this is enough to render the entire CPS reliable and robust.

A controller is said to be reliable, if the faults it experiences are masked from the rest of the CPS: If the crashes and delays of the controller are not observed by other components in the CPS hence do not affect grid safety or performance. For example, if the task of the controller is to maintain the bus voltages in the grid within certain limits, then a reliable controller would ensure that these limits are respected, even when it is slow to react to changes in grid conditions.

A controller is said to be robust if it can maintain grid safety and good performance in spite of non-ideal conditions resulting from faults in the rest of the CPS. In other words, unresponsive resource agents and network faults such as message losses, delays, and reordering, are handled by the controller. In the above example, the controller is said to be robust if it can maintain the voltage within its limits, in spite of missing advertisements or measurements from resources.

Reliability, therefore, is concerned with faults affecting the controller itself. Whereas robustness is concerned with faults affecting the rest of the CPS. A controller that can maintain correct operation despite both kinds of faults is said to be reliable and robust.

These definitions are re-iterated in Chapter 3 and distilled into formal requirements. We go over the challenges in achieving them, as we survey the literature in Chapter 2.

1.2 Contributions

Our contributions in this thesis are summarized as follows.

1. We provide an abstraction of a CPS for electric grids, by dividing it into four layers: the control layer, the network layer, the sensing and actuation layer, and the physical layer. We define the components of each layer, thus providing a

model for each component and for the interactions among them.

2. Using the abstraction and models in item 1, we formally define the required properties of a reliable and robust CPS. These properties can be divided into two main categories: robustness and reliability. The robustness properties are for ensuring grid safety in the presence of unresponsive resources and communication network losses, delays, and reordering. The reliability properties are for ensuring grid safety in the presence of crash and delay faults of the controller. As grid safety could be ensured by a complete shutdown of the system, all of these properties must be satisfied while preserving availability, *i.e.*, without sacrificing grid operation.

3. We propose intentionality clocks, a labeling scheme that captures the inherent round-based ordering in CPSs. Intentionality clocks enable a CPS controller to be robust against message reordering due to communication network non-idealities.

4. We propose Robuster, a mechanism that enables a CPS controller to compute and issue setpoints with partial information from the resources. Robuster guarantees that grid safety is maintained despite missing advertisements from resource. Thus, along with intentionality clocks in item 3, it guarantees all the robustness properties.

5. We propose Axo, a framework for tolerating delay and crash faults in CPS controllers by active replication of the controller. Axo enables the replication of the controller and ensures that delay and crash faults in the controller replicas are masked from the rest of the CPS. Axo also increases the long-term reliability of the CPS by detecting and recovering faulty controllers.

6. To demonstrate the necessity and improvement brought about by Axo, we perform a case study, considering an inverted pendulum system. We show how controller faults negatively affect the stability of the pendulum, and how the incorporation of Axo maintains the stability.

7. We propose Quarts, a mechanism for quick agreement among replicas in a CPS. We prove that Quarts guarantees consistency and maintains an availability higher than state-of-the-art consensus algorithms. Along with Axo from item 6, Quarts guarantees the reliability properties defined in item 2.

8. We show how Quarts can be used in CPS controllers that receive both advertisements and measurements. This enables the use of Quarts in any CPS for real-time control of electric grids.

9. We design and implement T-RECS, a virtual commissioning tool for real-time control of electric grids. Using T-RECS, we can test CPS control algorithms and

their effect on grid safety and operation. We can also study the effects of faults and non-idealities on the grid and validate the benefits of our reliability mechanisms.

10. We validate T-RECS with a real-scale microgrid, and we show that performing tests in T-RECS produces results with insignificant error.

11. We study a CPS with COMMELEC [8], a framework for real-time control of electric grids via explicit power setpoints. We highlight the reliability and robustness issues of COMMELEC, and we show how our solutions can be used to produce a reliable and robust CPS. We analyze the full system and show how the various mechanisms cooperate seamlessly both in regular grid-connected operation, and during islanding maneuvers and operation.

12. We consider a specific case of grid control in COMMELEC, namely, the islanding operation. We define the required properties for handling such an operation and propose a robust protocol for performing it. Our protocol orchestrates the selection of a slack resource, and it maintains grid safety in the presence of non-ideal conditions.

13. We implement our mechanisms and deploy them with COMMELEC, both in T-RECS and in a real-scale CIGRÈ low-voltage benchmark microgrid. Our deployments validate that the proposed mechanisms guarantee robust and reliable operation and maintain a high availability and low latency.

1.3 Roadmap

The rest of the thesis proceeds as follows.

In Chapter 2, we evaluate the literature on reliability and robustness, highlighting the drawbacks of existing solutions when dealing with real-time CPSs.

In Chapter 3, we define the abstractions of the different layers of a CPS for electric grids. We provide a model of the operation of the components and the interactions among them, and we specify the possible faults considered in each. Then, using these models and abstractions, we formally define the required properties for reliable and robust real-time CPSs.

In Chapter 4, we introduce intentionality clocks, a mechanism for labeling messages in a CPS, based on round numbers. Intentionality clocks make the CPS robust to message reordering. Then, we introduce Robuster, a mechanism to ensure that the controller is robust to the other considered non-idealities, namely, unresponsive resources and message losses and delays. We prove that intentionality clocks and Robuster guarantee the robustness properties of CPSs. Finally, we evaluate these mechanisms

in a grid deployment, and we show that they perform better than state-of-the-art alternatives.

In Chapter 5, we introduce Axo, a framework for tolerating delay and crash faults in controllers of CPSs. Axo uses active replication of the controller and a discard mechanism that together ensure that delay and crash faults in the controllers are masked from the rest of the CPS. Axo also provides a mechanism for detecting and promptly recovering faulty replicas, thereby increasing the long-term availability of the CPS. We prove the reliability properties guaranteed by Axo, and we derive bounds on its detection and recovery time. Finally, we evaluate the performance of Axo through a stability analysis of an inverted pendulum system, showing the improved metrics with Axo.

In Chapter 6, we address the consistency issue brought about by active replication. We propose Quarts, a mechanism for quick agreement between replicas in CPSs. We prove that Quarts guarantees consistency and maintains an availability higher than state-of-the-art consensus solutions. We show that Quarts is generic and can be applied to CPS controllers with different inputs.

In Chapter 7, we introduce T-RECS, a virtual commissioning tool for real-time control of electric grids. T-RECS can be used to study the effect of CPS control, non-ideal conditions, and reliability mechanisms on the grid. We present the design of T-RECS, highlighting the layering approach it follows. Then, we validate the simulated grid model in T-RECS with measurements from a real-life on-campus microgrid. Finally, we analyze the performance of T-RECS on a standard laptop, thus showing that it can be used to test CPSs for real-time control of electric grids.

In Chapter 8, we present a case study of COMMELEC [8] in a CPS for real-time control of electric grids. We analyze the reliability and robustness issues of the CPS and show how our mechanisms can be combined to transform it to a reliable and robust CPS. We showcase a deployment of our mechanisms with COMMELEC, both in T-RECS and in a real-scale low-voltage CIGRÈ benchmark microgrid [44]. Our deployment shows the significant improvement in grid safety and performance when applying our reliability and robustness mechanisms. Then, we consider operations related to islanding, namely, the islanding and reconnection maneuvers and the operation of a microgrid in islanded mode. We propose a robust protocol for performing the mission-critical task of switching the mode of operation of the resources to and from slack, in order to maintain grid safety.

Finally, in Chapter 9, we offer our concluding remarks, in addition to possible directions for future work.

2 State of the Art

Those who cannot remember the past are condemned to repeat it.
— George Santayana

In this chapter, we survey the state of the art on reliability and robustness mechanisms both for CPSs in general and for smartgrids in particular. We highlight the drawbacks of existing solutions in addressing the issues raised by real-time control of electric grids, all the while reiterating our contributions in light of this literature review.

We begin with communication network reliability, as CPSs heavily rely on the communication network for message exchange. Thus, the reliability of this network must be considered. Although it is a well-studied problem in the literature, we discuss it here for completeness.

2.1 Communication Network Reliability

Communication network reliability can be achieved through either packet retransmission or packet replication.

The traditional go-to protocol for communication network reliability via retransmission is TCP [45]. TCP is a transport layer protocol that ensures in-order packet delivery through the retransmission of lost packets and acknowledgement of received packets. However, this in-order delivery constraint leads to the head-of-line blocking phenomena, whereby later messages are delayed until earlier ones have been delivered [46, 47]. Furthermore, due to congestion control in TCP, a packet loss decreases the rate of transmission, thereby increasing the delay.

Such behavior is necessary in traditional systems, because the application requires earlier messages to be received before later ones are processed. However, this is not the case for CPSs. CPS messages are ephemeral in nature, *i.e.,* the reception of a

new message invalidates earlier ones. An example of this is the reception of voltage measurements from sensors in a grid. Once the controller receives the value of the voltage at time t_1, the value of the voltage at a time $t_2 < t_1$ becomes redundant, because the state of the grid has already changed.

QUIC [48] is a recently proposed protocol that relies on multiplexing UDP streams. Data within a stream is delivered in-order with reliability guarantees provided by QUIC. However, data across multiple streams can be delivered out of order, thereby enabling retransmission without head-of-line blocking. QUIC has been shown to improve the network reliability over TCP, decreasing latency and loss rates in an Internet-wide deployment [49]. This makes QUIC more suitable for CPSs.

Packet retransmission, however, inherently incurs additional latency, as a retransmission of a lost packet only takes place when the original packet is detected to be lost by the sender. An alternative is packet replication, whereby two or more copies of the same packet are simultaneously sent over redundant network paths to the same destination. Redundancy in network paths is essential for alleviating component failures. Hence, it will be available in networks in which reliability is important.

PRP [50, 51] is a MAC-layer solution that exploits redundant network paths for packet replication. iPRP [52] extends PRP to cover IP networks, which cover a wider area and are usually used for CPSs, especially smartgrids [53]. Packet replication over redundant network paths not only improves reliability, but also reduces the overall latency, as the first packet to be delivered will be taken. This so-called 0 ms repair time is essential for CPSs with real-time constraints.

The aforementioned protocols only serve to improve the reliability of the communication network, but packet losses, delays, and reordering can still occur. Other protocols, such as time-triggered Ethernet [54], FlexRay [55], and CAN [56], aim to eliminate such non-idealities by requiring specialized hardware and a message sending schedule from the communicating agents. Although these schemes work well in traditional real-time systems in which the agents are highly deterministic, they are not amenable to emerging CPSs, especially when computation delays are considered. Also, the guarantees provided by such protocols no longer hold when component failure is considered [57, 58].

2.2 Robust Cyber-Physical Systems

A CPS controller must handle non-idealities in the rest of the CPS to be considered robust. Such non-idealities include message losses, delays, or reordering that might result from faults in the communication network, or in the resources, RAs, and sensors it communicates with.

Several CPS controllers in the literature, particularly in the domain of grid control, are designed assuming an ideal communication network [59, 60, 61]. However, as we have seen in Section 2.1, despite advances in communication network reliability, non-idealities in a CPS deployment cannot be disregarded.

2.2.1 State Estimator Robustness

The problem of handling non-idealities in SEs has been well studied in the literature. Several approaches propose the design of robust SEs using Kalman filters in the presence of intermittent input measurements [62,63,64,65]. These enable the SE to provide an output with a bounded error, despite missing measurements that might result from non-idealities in the network or the sensors. These techniques meet the requirements for real-time control.

Another approach is to use redundancy in the sensing infrastructure, where multiple sensors have visibility of the same node in the grid [66, 67]. Thus, if a measurement from one redundant sensor is lost or delayed, the SE maintains visibility of the entire grid hence can perform correct computation. Sensor redundancy has the added benefit of enabling the SE to perform bad-data detection [68, 69].

Others consider the dual problem of missing output at the actuators (the recipients of the SE output) and propose techniques such as model predictive control (MPC) with buffered actuation [70, 71, 72]. In these approaches, the SE computes output for the next k sampling periods and sends them to the actuators. The actuators use the first output, and they buffer (save) the rest for later use, in case of missing data from the SE.

MPC relies on modeling and forecasting uncertain quantities in the grid [73, 74, 75]. Although MPC is applicable for real-time control, it can incur a high computational complexity, especially in cases with non-negligible losses in the grid [76] or large prediction horizons [73, 77].

In conclusion, the state of the art for robust SEs is solid and, with the exception of MPC in some cases, can be used for real-time control.

2.2.2 Grid Agent Robustness

In this section, we consider the robustness of the GA, a component that communicates with RAs in a round-based model, thus sending setpoints and receiving advertisements. Non-idealities in the rest of the CPS might cause the input advertisements of the GA to be lost, delayed, or reordered. It must nonetheless be able to compute setpoints to maintain control over the grid. To the best of our knowledge, we are the first to attempt to solve this problem in this context.

A prerequisite for the techniques used for SE robustness is a labeling mechanism that provides an ordering among the measurements received by the SE and the outputs it issues. Such labeling is presupposed by the SE robustness mechanisms discussed earlier and is required for GA robustness as well.

Labeling can be done using either timestamps or logical clocks. Timestamp-based approaches require time-synchronization. This can be achieved through network-based time-synchronization protocols (e.g., NTP [78], PTP [79]) or GPS-based synchronization [80]. However, timestamps cannot be used to reliably order messages in a general distributed system [81]. In specific cases, such as sensors and SEs, time-alignment can be used to provide such an order [82].

For GAs and RAs, time-alignment cannot be applied, as their form of communication is round-based. Such communication requires the use of logical clocks that capture the round numbers. In the literature, logical clocks are achieved through Lamport clocks [81], the prominent example of scalar clocks, or through Vector clocks [83].

These labeling mechanisms, however, are designed for generic distributed systems. Hence, they do not capture the inherent round-based communication between GAs and RAs. In Chapter 4, we propose a modification to Lamport clocks, called *intentionality clocks*, which suits this form of communication. Moreover, intentionality clocks work seamlessly with GA replication (required for reliability as discussed below), whereas traditional mechanisms require further modification.

Additionally, in Chapter 4, we propose *Robuster*, a mechanism that enables the GA to compute setpoints in the presence of non-idealities in the rest of the CPS. In Chapter 8, we consider the case when the GA is controlling an islanded microgrid, and we augment the robustness mechanism to work in this context.

2.3 Reliable Cyber-Physical Systems

A robust CPS controller is still susceptible to crash faults, in which it ceases to process input and issue output, and to delay faults, in which it is slow in doing so. Additionally, a CPS controller can exhibit a wide range of faults that cause it to produce erroneous output, ranging from software bugs to security attacks [40]. These faults are classified as Byzantine [39], and are handled by Byzantine-fault tolerance (BFT) techniques [84, 85, 86]. However, BFT suffers from poor latency performance, both in the best case, which requires a minimum of two rounds of message exchange, and in the worst case, which takes unbounded time. This makes BFT unsuitable for real-time CPSs.

In this thesis, Byzantine faults are out of scope, as we focus on crash and delay faults. Several approaches have been proposed to address such faults in traditional computing systems. In this section, we go over some of these approaches, and highlight

their drawbacks when applied to real-time CPSs.

2.3.1 Deterministic System Design

Delay faults in the literature are referred to as *timing faults, i.e.,* faults that affect the time at which a message is received. One approach to handling timing faults is deterministic system design, with the two prominent examples being the timely computing base (TCB) [87, 88], and the time-triggered architecture (TTA) [89, 90].

TCB is a framework that provides a strict real-time component, in which time-critical functions can be executed. In order to provide real-time guarantees, TCB presupposes that these functions can be separated from the main code base and rewritten in TCB format. TTA also requires similar modifications.

Both frameworks can be used for the detection of timing faults. However, TCB requires a known bound on the computation time of the time-critical functions [91], and TTA requires that the intended send and receive instants of messages be known *a priori* [92].

The requirements posed by these frameworks make them unsuitable for CPSs for several reasons. CPSs typically have a large code base, mainly consisting of third-party libraries. Hence, rewriting and encapsulating certain time-critical functions might not be feasible. Furthermore, CPSs execute complex functions, the execution time of which cannot always be known. Similarly, sending and reception instances at CPSs depend on both the software agents and the communication network among them, thus making them difficult to predict.

The drawbacks of these frameworks are shared with controller modeling techniques [93, 94, 95], in which a trained model of the controller classifies it as faulty or not during run-time. Such methods are prone to modeling errors and are limited to CPSs with constant or predictable workloads. Thus, they are inapplicable to the generic CPSs we consider in this work.

2.3.2 Consensus

The alternative to deterministic system design for handling crash and delay faults is redundancy through controller replication, in which two or more replicas of the same controller are used to mask, detect, and recover from faults. We go over the two main replication approaches in the next two subsections. Here, we discuss an element that is central to both, namely, consensus [96].

Consensus is at the heart of every distributed computing system. Consensus is performed among a set of distributed software agents, in which each software agent

proposes a value, and must decide on a value. Formally, consensus has four properties:

- **Agreement:** Every correct software agent that decides on a value, decides on the same value.
- **Termination:** Every correct software agent decides on some value.
- **Validity:** If a correct agent decides on a value, then this value must have been proposed by some correct agent.
- **Integrity:** If all correct agents propose the same value, then any correct agent decides on that value.

If a protocol achieves all four properties, it is said to provide consensus. The word *correct* refers to agents that do not experience a fault when consensus is being performed. Consensus protocols usually provide guarantees only when the number of correct agents is large enough. For example, consensus in BFT [84] can handle f faulty agents, if the system consists of at least $3f + 1$ agents in total.

The validity and integrity properties are specified to eliminate trivial protocol designs, such as all processes agreeing on some statically set non-proposed value, and to handle Byzantine [39] agents that behave erroneously.

Agreement is a *safety* [97] property that ensures that no two correct agents decide on different values. Termination is a *liveness* [97] property that ensures that correct agents *eventually* decide on a value. Depending on the requirements of the protocol, the termination property might have a certain deadline, given in time or in number of "consensus rounds".

Consensus (specifically, agreement and termination) is proven to be impossible to guarantee in bounded time in the presence of faults [98]. The result states that given at least one crash faulty software agent and a non-ideal communication network, consensus might require an unbounded number of steps for all correct agents to satisfy agreement and reach termination. Thus, for the fault model we consider, which includes software agent crashes and delays, in addition to network non-idealities, agreement and termination remain impossible to guarantee together.

In spite of the impossibility result, several protocols have been proposed to provide consensus [99, 100, 101], relying on the observation that faults are rare in practice. Paxos [102, 103] is a widely-used consensus protocol that guarantees agreement under a crash-only fault model. However, Paxos requires at least three rounds of message exchange between the software agents in order to reach termination. The number of rounds can grow indefinitely when the network conditions are bad.

Other protocols have been proposed that incur less minimum rounds of message exchange, but they require additional assumptions on the system or trade-off best-case performance with average-case performance [104, 105]. FastPaxos [104] is one such example that reduces the minimum number of rounds to one but has poorer latency performance than Paxos when it encounters conflicts between software agents.

In Chapter 6, we present *Quarts*, a protocol that circumvents the impossibility result by forgoing termination and focusing on agreement. We show that in CPSs that satisfy certain requirements, generally true in real-time control, termination is not necessary in every round to maintain correct control. Quarts outperforms state-of-the-art consensus protocols, providing both a lower latency and a higher availability.

2.3.3 Active Replication

Fault Masking

Active replication [106, 107, 108, 109] involves two or more replicas of the controller that simultaneously receive input, perform computations, and issue setpoints. Using such a replication technique, if one of the controller replicas crashes, then the other replicas continue operation, thus maintaining control over the system. Hence, active replication is said to increase the availability of the system, as it reduces the impact of crash faults.

The same applies for delay faults. If one replica experiences a slow computation time upon receiving input, then the presence of other replicas increases the chances that the RAs will receive setpoints with a low response time. However, in order to properly mask delay faults from RAs, the setpoints issued by the delayed replica must not be delivered to the RAs, as they represent an older state of the system. In Chapter 5, we make use of active replication and extend it with a discard algorithm in order to mask delay faults from RAs in a CPS.

Fault Detection

Additionally, in order to maintain the benefits of active replication over long periods of CPS operation, replicas that experience faults must be detected and recovered. Perfect fault detection is a problem similar to consensus [38], hence is impossible to guarantee in real-time. However, fault detection need not be on the real-time path, thus practical protocols for fault detection are suitable for CPSs.

Several techniques exist for crash-fault detection [110, 111, 112]. The most common technique is using *heartbeats*: keep-alive messages that are periodically sent by replicas in order to advertise that they are not faulty. Absence of successive heartbeat messages

from a replica is cause to believe it has crashed.

In contrast, delay-fault detection is a more difficult process. Besides the techniques in TCB [91] and TTA [92] mentioned in Section 2.3.1, this problem has not been addressed in the literature. Delay faults are an end-to-end phenomena. Hence, they can be detected only by involving the RA in the process. Furthermore, as seen in Chapter 1, delays are intermittent in nature, hence there is a trade-off between detection time and detection accuracy. In Chapter 5, we present a mechanism for crash and delay fault detection in CPSs.

Fault Recovery

Fault recovery is the process of resetting a replica that was detected as faulty into a good state. One example is rebooting the entire replica, although microreboots of one or more processes has also been suggested [113].

In a distributed system with multiple replicas attempting to reboot a faulty replica, one issue is ensuring that the faulty replica is rebooted at most once after detection. In general, this problem shares similarities with consensus and cannot be guaranteed within bounded delay. However, in the presence of a total ordering [114] in the CPS, rebooting no longer poses a difficult problem. In Chapter 5, we utilize a total ordering provided by intentionality clocks in order to reboot faulty replicas after detection.

Proactive recovery [115, 116] is an alternative to fault detection and (reactive) recovery for maintaining long-term availability of the CPS. In such a scheme, replicas are scheduled to reboot regularly, regardless of whether they experience faults. This approach is desirable if the replicas are observed to experience more faults with an increase in running time, a phenomena known as software aging [117, 118, 119]. However, as we have seen in Chapter 1, and from several of our experiments, aging has not been observed in CPS controllers. Rather, CPS controllers tend to have intermittent faults that occur at timescales that are significantly lower than aging timescales. These must be detected and reactively recovered, which causes proactive recovery to be superfluous.

Replica Agreement

An emergent issue, when using active replication, is the *split-brain syndrome* [120, 121]: whereby two (or more) replicas are sending contradictory setpoints to the RAs. This can occur due to the replicas computing with different input, a non-deterministic compute function [122], or Byzantine faults [39]. In order to maintain correct control, replicas in active replication must agree either on which replica sends the setpoints, or on what setpoints the replicas must send.

As discussed in Section 2.3.2, this form of agreement requires a consensus protocol, which is an expensive process with high tail latency. In Chapter 6, we highlight a set of properties present in CPSs, which enables us to circumvent consensus, and we present Quarts as a solution to the replica agreement problem.

2.3.4 Passive Replication

Passive replication [123, 124, 125, 126], also known as the primary-backup approach, is another form of redundancy, in which one replica serves as the *primary*, receiving input, computing, and issuing setpoints, and the others as *standby*. The primary maintains control, while the standbys monitor the primary in order to detect it as faulty. Once the primary is detected as faulty, the standbys perform leader election (via consensus) and elect a new primary amongst themselves. The faulty primary can then be recovered and set as standby.

Standby replicas in passive replication can be either hot [127, 128] or cold [129]. Hot standbys are state-synchronized by the primary after each computation, whereas cold standbys are not. On one hand, once a hot standby becomes a primary, it can immediately commence the control, as it has the most recent state. Cold standbys require additional time to get the current state, before taking part in further computation rounds. On the other hand, maintaining hot standbys adds significant latency overhead to each computation done by the primary. The choice is, therefore, a trade-off between average-case performance, and worst-case performance.

Standby replicas in passive replication monitor the primary and detect it as faulty. As mentioned earlier, perfect failure detection is impossible to guarantee in a bounded time [38]. In this setting, imperfect failure detection might result in false negatives, whereby a faulty primary is not detected, thereby reducing the availability of the system. It might also result in false positives, whereby a non-faulty primary is considered to be faulty. This is followed by the election of another primary among the standby replicas, resulting in two primaries, and possibly in the split-brain syndrome.

Although failure detection is imperfect in theory, several proposed techniques perform well in practice [110, 111, 112]. However, such techniques consider a crash-only fault model. Delay faults further exacerbate the issue of imperfect detection, as delay faults are intermittent in nature. On one hand, a primary replica that experiences intermittent delays might not be detected, thereby suffering a reduced availability. On the other hand, detecting an intermittently faulty primary and incurring the delay of choosing a new primary via leader election further reduces availability. Therefore, passive replication in the presence of delay faults is unsuitable for CPSs with real-time constraints.

3 System Model

Don't be fooled by the many books on complexity or by the many complex and arcane algorithms you find in this book or elsewhere. Although there are no textbooks on simplicity, simple systems work and complex [ones] don't.
— Jim Gray, "Transaction Processing: Concepts and Techniques"

In this chapter, we provide an abstraction of a CPS for electric grids, dividing it into four layers. We introduce the operation model for the components in each layer, the interactions among the different components, and the fault model. Finally, we formally define the required properties for reliable and robust real-time CPSs.

3.1 Model of a Cyber-Physical System for Electric Grids

Figure 3.1 shows the abstract model of a CPS for electric grids. The grid in this example comprises four buses, $B_0 - B_3$. Bus B_0 is the slack bus, and buses B_1, B_2 and B_3 have a generator (a producing resource), a battery (a prosuming resource) and a load (a consuming resource), respectively. The control of the grid and the resources is performed by software agents C1, C2, and RA1-RA3.

C1 and C2 are two controller replicas. Recall from Figure 1.2 in Chapter 1 that a controller comprises a Grid Agent (GA) and a State Estimator (SE). Controllers receive, from sensors, measurements on the state of the grid (voltage/power at each bus, current at each line). In Figure 3.1, we see two controller replicas. Recall from Chapter 2 that active replication of the controller is necessary to handle delay faults.

Software agents RA1-RA3 are resource agents that read the state of the respective resources, namely, the generator, the battery and the load, through a sensor interface provided by the resource. For example, RA3 reads the internal state of the load, such

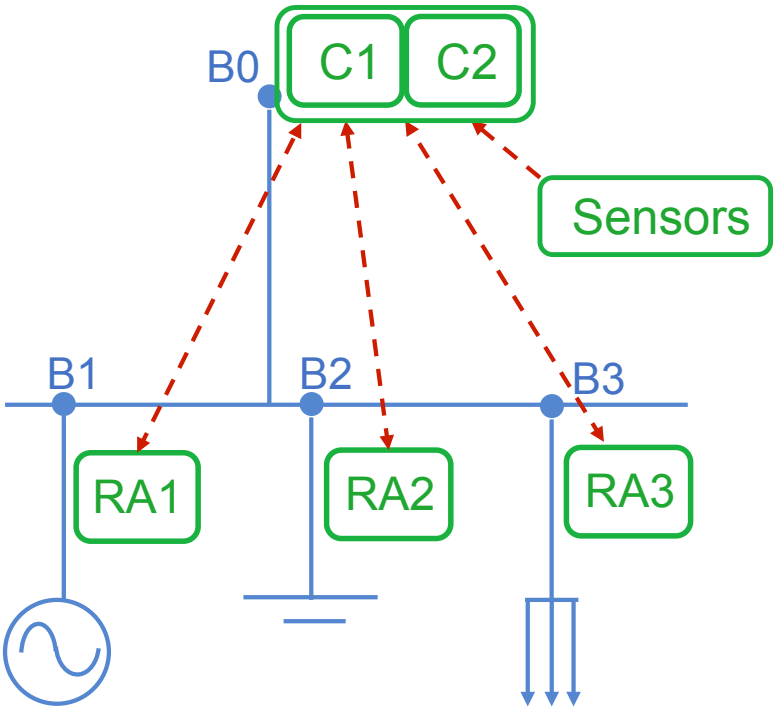


Figure 3.1 – Abstract model of a CPS for electric grids

as the temperature (for a thermal load), through the on-board thermostat.Then, the sensed quantities are used to create an advertisement that encapsulates the state of the resource. This advertisement is sent to the controllers C1 and C2.

In order to control the resources, the controllers perform computations and issue setpoints, thereby keeping the grid in a desired state. The setpoints are received by the RAs and implemented via the actuator interfaces at the resources. For example, a setpoint for changing the injected power at a battery is implemented by the converter on the battery, *i.e.,* the actuator.

We divide the various components of the CPS into 4 layers, as shown in Figure 3.2.

1. Physical layer — consists of the electric grid and the resources (thermal load, battery, generator, PV, supercapacitor, water pump).
2. Sensing and actuation layer — consists of sensors that read the state of the physical layer and actuators that alter the state of the physical layer.
3. Network layer — consists of routers, links, and switches that represent the communication infrastructure among software agents, sensors, and actuators.
4. Control layer — comprises the software agents (controllers and RAs) that receive measurements from the sensors, perform computations, exchange messages among themselves, and issue setpoints to the actuators.

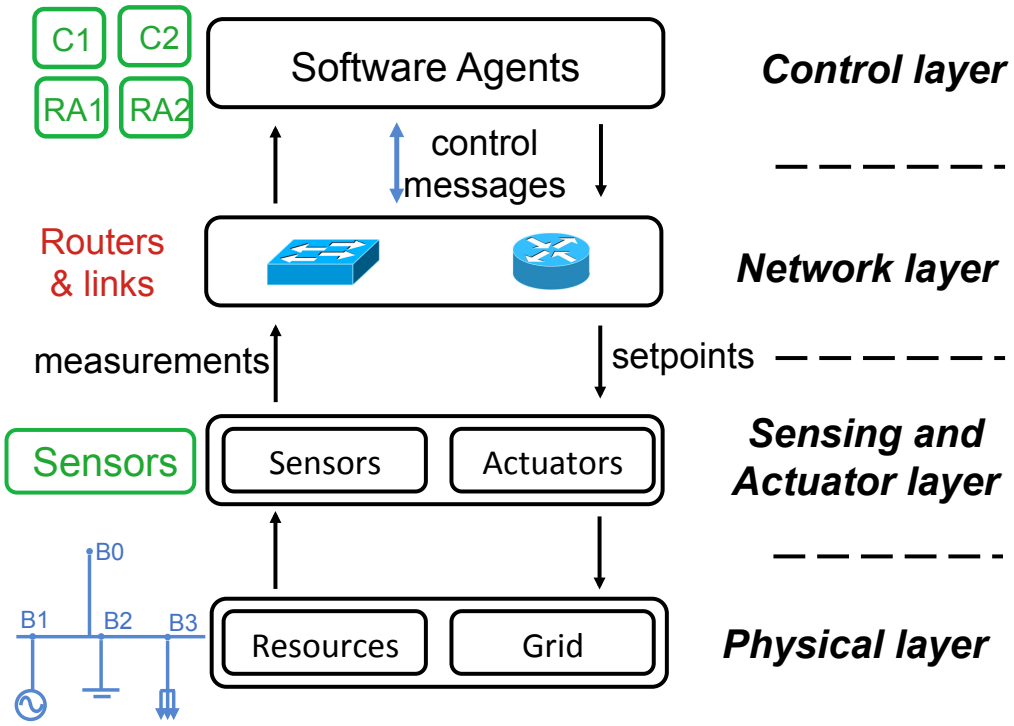


Figure 3.2 – Layers and components in a CPS for electric grids

3.2 The Bottom Three Layers

The focus in this thesis is on the control layer. The physical, sensing and actuation, and network layers are not modified. However, it is essential to list the components in these layers, to present their models, and to define the assumptions on each.

3.2.1 Physical Layer

The physical layer consists of the grid, comprising the buses, lines, transformers, and breakers, in addition to the electric resources located at the different buses.

The grid topology includes the connections of the lines and buses, the line parameters, the transformer locations and parameters, and the breaker locations and setting (on/off). These, in addition to the bus voltage magnitudes and frequencies, and the line currents, make up the state of the grid.

The electric resources are connected at different buses in the grid. They can be production, consumption, or storage devices. They can either be controllable, via setpoints sent to their actuators, or uncontrollable.

We make no assumptions on the physical layer. The grid might experience line faults, and the resources might shut down or be unresponsive to actuation requests.

3.2.2 Sensing and Actuation Layer

This layer consists of sensors and actuators, which are cyber components that interface with the physical layer. Actuators are simple devices that interface with one electric resource. They receive setpoints from RAs and implement them in their respective resource. For example, a battery converter acts as an actuator that receives power setpoints. One such setpoint might instruct it to implement +40 kW, to which it will react by instructing the battery to inject 40 kW into the grid.

Sensors are of two types: synchronous and asynchronous. Synchronous sensors are similar to actuators in that they interface with one electric resource. They receive queries from RAs and respond with measurements capturing the current state of their respective resource. For example, a battery sensor can be queried for the SoC of the battery. The term "synchronous" refers to these sensors' query-response model, *i.e.,* they only send measurements when queried by an RA.

In contrast, asynchronous sensors send measurements to the controller without being queried. We consider time-triggered asynchronous sensors that send measurements at pre-determined time intervals. These sensors are placed at different locations in the grid to ensure grid coverage or visibility [67]. The measurements they send contain information on the voltage magnitudes and frequencies at certain buses, currents at certain lines, etc.

We assume that asynchronous sensors send timestamped measurements, and that they are time-synchronized with one another. Time-synchronization can be achieved via GPS [80] for nanosecond accuracy, or network-based protocols, such as NTP [78] or PTP [79], for an accuracy that ranges from microseconds to milliseconds. The time-synchronization is assumed to be sufficient for the controller to time-align their measurements [82]. We also assume that asynchronous sensors are not malicious, *i.e.,* the measurements they send reflect the state of the grid at that time. Inaccuracies are allowed by our model, as long as the redundancy in the sensor infrastructure is sufficient for the controller to detect bad data [69].

Actuators and synchronous sensors are also assumed not to be malicious, *i.e.,* actuators do not change the setpoints received before implementation, and sensors do not contaminate the measurements before sending them to the RA. Inaccuracies in sensing and implementation are allowed, as long as they are sufficiently bounded, thus enabling the RA to account for them, as described in Section 3.3.1.

Both types of sensors can crash, be delayed, or be unresponsive to queries. Actuators can crash or omit implementation requests. Actuators might also be delayed when implementing setpoints, such as the case with thermal loads that are slow to react. However, we consider that such delays are known to the RA, enabling it to account for them in its advertisements, as seen in Section 3.3.1.

3.2.3 Network Layer

The network layer includes routers, switches, and links. It represents the infrastructure required for the communication among the software agents in the control layer, and between them and the components of the sensing and actuation layer.

The network is considered to be probabilistic synchronous [38]. That is, messages might be dropped or delayed beyond a threshold δ_n. This occurs with a probability p. Otherwise, messages are delivered within an upper bound on the one-way latency, given by δ_n. Message reordering in the network is also considered as part of the model, whereas message contamination is not.

The probability p, hereafter referred to as the loss probability, accounts for network losses and delays that result from congestion, and from link or router failures. As mentioned in Chapter 2, CPS networks can be enhanced via retransmissions using QUIC [48], or via packet replication using iPRP [52]. The loss probability decreases with the use of such protocols, but remains non-zero.

3.3 Control Layer

The control layer consists of the software agents that orchestrate the control of the physical process. Recall from Chapter 1, Figure 1.2 that these comprise the RAs and the controller. The controller is split into two components, namely, the SE and the GA, that perform different functionalities.

3.3.1 Resource Agents

An RA is a software agent assigned to one electric resource. It is responsible for monitoring the resource via the local synchronous sensor, and for controlling it via the actuator. The RA model is shown in Algorithm 3.1. For a running example, we consider an RA of a battery that receives setpoints instructing it to inject/absorb a certain current value.

The RA maintains two variables corresponding to advertisement fields (lines 1-2). $\mathcal{F}$ represents the feasibility set of the electric resource. It is the region in which the resource can operate. For example, $\mathcal{F}$ could be a current range in which the battery can operate, specifying it can inject up to 40 A and absorb up to 30 A. In such a case, $\mathcal{F}$ would contain all the numbers between -30 and +40 (positive current representing injection into the grid). Setpoints outside this range cannot be implemented.

$\mathcal{U}$ represents the uncertainty of the resource in terms of actuation. It captures the accuracy of both the actuator and the resource when implementing the setpoint. In general, $\mathcal{U}$ is a set-valued function that depends on the value of the setpoint. For each

Algorithm 3.1: Abstract model of an RA

```
1  F ← ∅;                                   // feasibility set of the resource
2  U(.);                                    // uncertainty function of the resource
3  T_val;                                   // validity horizon of advertisement fields
4
5  on inititilization
6  |  T_val ← configure();                  // initialize validity horizon from configuration file
7  end;
8
9  on reception of a setpoint sp from the controller
10 |  implement_setpoint(sp);               // send sp to the local actuator
11 |  T ← get_current_time();               // get the current time
12 |  m ← sense_state();                    // sense state of the resource via the local sensor
13 |  F, U ← update_state(m, T_val);        // compute F and U with the given validity horizon
14 |  adv ← create_advertisement(F, U, T);
15 |  issue(adv);                           // send adv to the controller
16 end;
```

setpoint u in the feasibility set, $\mathcal{U}$ returns an uncertainty set that represents the set of possible values the resource might actually implement, if instructed to implement u. For example, $\mathcal{U}(u) = \{u' \mid 0.99 \times u \leq u' \leq 1.01 \times u\}$, represents the uncertainty function of a battery that has an uncertainty of 1% with respect to the setpoint value. Alternatively, $\mathcal{U}$ could be absolute, *i.e.,* independent of the setpoint value. For example, $\mathcal{U}(u) = \{u' \mid -30\ A \leq u' \leq 40\ A\}$, represents an uncontrollable battery that might implement any value in the given range, no matter what the setpoint is. We note that setpoints might also contain active and reactive power (PQ), as shown in several examples in Chapter 4.

These advertisement fields depend on time and on the state of the electric resource. For example, the feasibility region of a battery depends on its SoC, and that of a PV depends on the irradiance. A feasibility computed given a certain SoC would no longer be valid when the SoC changes. We define the *validity horizon* (T_{val}) as the time horizon during which advertisement fields are valid. Due to the low inertia of some resources, such quantities can only be computed to be valid for a short time [130, 131]. Furthermore, as we demonstrate in Chapter 4, it is desirable to compute these fields with the minimum value of the validity horizon. The minimum value of the validity horizon depends on the period of the control round, and is discussed further in Section 3.3.3. The RA is assumed to be configured with this minimum value of the validity horizon (lines 5-7).

Note that an RA might maintain additional variables for its advertisement fields, such as the preference of the resource in terms of setpoint implementation. For example, a battery with a low SoC might prefer setpoints that instruct it to absorb current rather than to inject it, even though its feasibility set allows for either. We

focus on the aforementioned two fields, as they correspond to those required by the controller to maintain grid safety and control the resources efficiently. In Chapter 8, we discuss additional variables that might be useful in certain contexts.

The RA receives setpoints from the controller or, more specifically, the GA. Upon receiving a setpoint (lines 9-16), the RA calls the `implement_setpoint` function (line 10), which sends the setpoint to the actuator for implementation. We consider that this function only returns when the actuator finishes implementation. This enables the RA to account for the time required for the implementation to fully take place and to avoid reading a transient state from the sensor.

After implementing the setpoint, the RA measures the current time (line 11), then it calls the `sense_state` function which queries the local synchronous sensor for the state of the resource (line 12). This function returns a measurement, that is then used by the RA to compute the advertisement fields, $\mathcal{F}$ and $\mathcal{U}$, given the validity horizon T_{val} (line 13). Then, it encapsulates the fields and the measured time in an advertisement (line 14), which it sends to the GA (line 15). The time is measured after implementation, and immediately prior to sensing the state of the resource. Thus, it represents the time after the state of the resource changed due to the setpoint implementation.

Let us give an intuition behind the "meaning" of the advertisement fields. An RA sends an advertisement to the GA in response to a setpoint received. This advertisement, along with advertisements from the other RAs, will be used by the GA to compute the next setpoints that will be issued to the RAs. For each RA, the next setpoint u, computed by the GA, must belong to the advertised feasibility set $\mathcal{F}$, of that RA. The advertisement also informs the GA that, given u, the resource might actually implement any value in $\mathcal{U}(u)$, until the end of the validity horizon.

We note that the communication model between the GA and the RAs described above is round-based. That is, in each round, the GA sends setpoints to its RAs, which respond with advertisements. These advertisements are used to compute setpoints for the next round. The notion of a control round will be discussed further in Section 3.3.3.

Moreover, as we explain in Section 3.3.3, some setpoints sent by the GA are not to be implemented. Rather, they simply probe for an advertisement. For simplicity, we do not explicitly account for such setpoints in the RA model. Instead, we assume that the `implement_setpoint` function does not send these setpoint to the actuator.

Under our fault model, RAs might crash, be delayed in issuing advertisements, or be unresponsive to the GA by dropping received setpoints. Erroneous behavior due to software bugs or security attacks is not considered. Additionally, we consider that RA delays in implementing setpoints are upper-bounded with a known upper bound δ_i.

3.3.2 State Estimator

Algorithm 3.2: Abstract model of an SE

```
1  M ← ∅;                                   // set of measurements received from sensors
2  Z ← [ ];                                 // vector of measurements used in a computation
3  T_Z ← 0;                                 // timestamp of the measurements in Z
4  H ← ∅;                                   // internal state of the SE
5
6  on reception of a measurement m from an asynchronous sensor i
7  |  M ← M ∪ {(m, i)};                    // aggregate received measurements
8  end;
9
10 repeat
11 |  ready, Z, T_Z ← ready_to_compute(M);
12 |  if ready then
13 |  |  st, H ← compute_state(Z, H);
14 |  |  issue(st, T_Z);                    // send st to the GA
15 |  end
16 forever;
```

Algorithm 3.3: `ready_to_compute`($\mathcal{M}$) function in the SE

```
1 M', T_Z ← time_align(M);
2 ready, Z ← bad_data_detection(M');
3 return ready, Z, T_Z;
```

Recall from Chapter 1 that the SE is the component of the controller that is responsible for receiving measurements from the asynchronous sensors, computing the state of the grid, and sending this state to the GA. A high-level SE model is shown in Algorithm 3.2.

The SE maintains a set $\mathcal{M}$ of received measurements (line 1) that contains tuples of the form $(m,\ i)$, where m is a timestamped measurement received from the asynchronous sensor with identifier i. The SE is pre-configured with the sensor identifiers. $\mathcal{M}$ is updated upon the reception of a new measurement (lines 6-8).

Simultaneously, in the main thread (lines 10-16), the SE calls the `ready_to_compute` function (line 11) with the parameter $\mathcal{M}$. This function returns three values: (1) A $ready$ flag, indicating whether the SE should begin computation or wait for more measurements otherwise, (2) a vector $\mathbf{Z}$ containing at most one measurement per asynchronous sensor, to be used in the computation if the $ready$ flag is set to true, and (3) a timestamp $T_\mathbf{Z}$ of the measurements in $\mathbf{Z}$. If the $ready$ flag is set to true, the SE computes the current state of the grid st using $\mathbf{Z}$ and sends st and $T_\mathbf{Z}$ to the GA (lines 12-15). Note that we are considering a stateful SE, with $\mathcal{H}$ being its internal state, updated after each computation, such as the gain matrix in a Kalman filter [132].

The `ready_to_compute` function is described in Algorithm 3.3. It first time-aligns the received measurements (line 1). That is, it selects the set of measurements $\mathcal{M}'$ with the latest timestamps, selects a common timestamp $T_{\mathbf{Z}}$ for these measurements, and adjusts the values of the measurements accordingly. Then, `bad_data_detection` is called on $\mathcal{M}'$ (line 2). This function first checks whether enough measurements are available to perform a computation, and if so, whether redundant measurements are available to detect bad data and exclude it. It returns a $ready$ flag and the vector of measurements $\mathbf{Z}$ that can be used in computation if $ready$ is set to true. Note that $\mathbf{Z}$ contains the measurements in $\mathcal{M}'$, excluding any bad data detected, indexed by the identifier of the asynchronous sensor. Thus, $\mathbf{Z}$ contains at most one entry per sensor.

The SE, similarly to the GA, is susceptible to delay and crash faults. Whereas delay faults are intermittent, crash faults are permanent until actively recovered. Such a fault model can be represented by the Gilbert-Elliot model [133, 134], as shown in Figure 3.3.

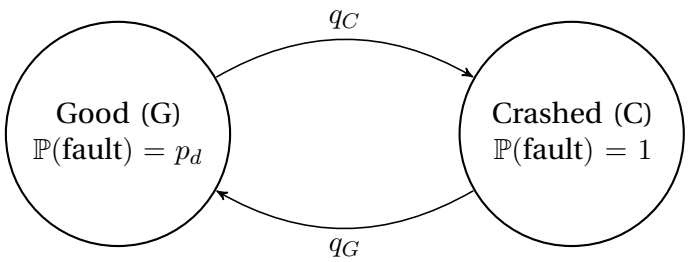


Figure 3.3 – Fault model of the SE and GA

In this model, q_C and q_G are the transition probabilities from state G to state C per computation round, and vice versa, respectively. p_d is the probability of a fault in state G, whereas the probability of a fault in state C is 1. A fault, in this case, is defined as the inability of the state estimator to issue a state to the GA within a predefined time limit.

We refer to faults occurring in state G as delay faults, and the ones in state C as crash faults. An agent (SE or GA) in state G experiences intermittent faults, whereas an agent in state C experiences faults until it is recovered, *i.e.,* moves to state G. We define θ_d and θ_c as the average probability of delay and crash faults, respectively, and R as the rate of recovery from crash faults. From this model, we have the following:

$$\theta_d = \pi_G \times p_d = \frac{q_G}{q_G + q_C} \times p_d$$
$$\theta_c = \pi_C \times 1 = \frac{q_C}{q_G + q_C}$$
$$R = 1/q_G$$

This model is used in Chapter 5 to formally characterize the benefits of Axo. It is also used to define the simulation parameters in experiments throughout this thesis.

3.3.3 Grid Agent

Algorithm 3.4: Abstract model of a GA

```
1  A ← ∅;                                   // set of advertisements received from RAs
2  S ← ∅;                                   // set of states received from the SE
3  Z_A ← [ ];                               // vector of advertisements used in a computation
4  H ← ∅;                                   // internal state of the GA
5
6  on initialization
7  |  X ← ∅;                                // initialize vector of setpoints to probes
8  |  issue(X);                             // send probes to the RAs
9  end;
10
11 on reception of an advertisement adv from an RA i
12 |  A ← A ∪ {(adv, i)};                   // aggregate received advertisements
13 end;
14
15 on reception of a state st with timestamp T from the SE
16 |  S ← S ∪ {(st, T)};                   // aggregate received states
17 end;
18
19 repeat
20 |  T ← current time;                    // get the current time
21 |  ready, timeout, Z_A, st ← ready_to_compute(A, S, T);
22 |  if ready then
23 |  |  X, H ← compute_setpoints(Z_A, st, timeout, H);
24 |  |  issue(X);                         // send X to the RAs
25 |  end
26 forever;
```

The other component of the controller is the GA. It is responsible for controlling the entire grid by coordinating the various RAs. It receives advertisements from RAs and the state of the grid from the SE, and computes and issues setpoints to the RAs. Algorithm 3.4 shows a high-level model of the GA.

The GA maintains several variables. The first is a set $\mathcal{A}$ of received advertisements (line 1) that contains tuples of the form $(adv,\ i)$, where adv is a timestamped advertisement received from the RA with identifier i. $\mathcal{A}$ is updated upon reception of a new advertisement (lines 11-13). The second is a set $\mathcal{S}$ of received states and timestamps from the SE (line 2), also updated upon reception (lines 15-17).

The main thread (lines 19-26) begins by recording the current time T (line 20), then calls the `ready_to_compute` function, with the parameters $\mathcal{A}$, $\mathcal{S}$, and T (line 21). Similar to its namesake in the SE model, this function returns a $ready$ flag and a set of variables to be used in the computation if the $ready$ flag is true (lines 22-25).

A computation returns a vector $\mathbf{X}$ of exactly one setpoint per RA (line 23), each of which is then issued to its corresponding RA (line 24). The GA can also have an internal state $\mathcal{H}$ (line 4) that influences the computation. This state is updated with each computation (line 23).

There are two ways in which a computation can be invoked: the sets of advertisements and states are enough to begin a computation or a *timeout* has occurred. The former leads to what is referred to as *implementation* setpoints, whereas the latter results in *probe* setpoints.

Implementation Setpoints

Implementation setpoints are control messages that carry information that the RA must implement, such as a power setpoint or a current injection. Such setpoints are the norm. They are computed and issued when the GA has enough information about the current state of the grid and resources. In those cases, the `ready_to_compute` function returns the following:

1. the $ready$ flag set to true, indicating that the GA can begin a computation
2. the $timeout$ flag set to false, indicating that no timeout has occurred
3. a vector $\mathbf{Z}_{\mathcal{A}}$ of at most one advertisement per RA, to be used in the computation
4. a state st to be used in the computation

The computation with these inputs returns a vector $\mathbf{X}$ containing one setpoint per RA. The implementation of this set of setpoints by the RAs serves two purposes. First, it maintains grid safety (formally defined in Section 3.4). Second, it optimizes for both the preferences of the resources and the objective given to the GA, such as following an external dispatch plan or providing frequency support to the main grid.

Under certain conditions, grid safety can be violated in order to maintain optimal performance for the resources' preference and the external request [135]. This can be done as the standards allow limited violations [15, 16, 17]. However, in general, maintaining grid safety takes precedence.

Probe Setpoints

In certain situations, and increasingly in the presence of non-idealities, the GA might not have enough information to perform a computation. That is, a computation it performs is not guaranteed to result in a vector of setpoints that maintains grid safety. However, in order to maintain real-time control, the GA sends probe setpoints in those

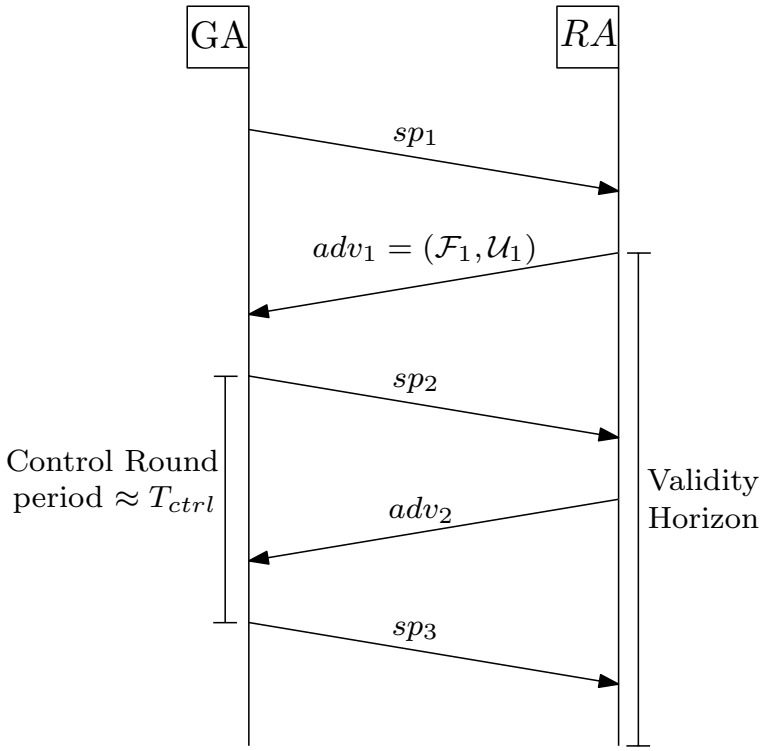


Figure 3.4 – Message sequence chart showing control rounds between a GA and an RA

situations. Probes do not carry information for implementation. Thus, RAs do not implement these setpoints, but they generate new advertisements in response to them.

In such cases, the `ready_to_compute` function returns both the $ready$ and $timeout$ flags set to true. The `compute_setpoints` function, when given a true $timeout$ flag, populates $\mathbf{X}$ with probes.

Probe setpoints are also issued when the GA is initialized (lines 6-9). The RAs respond to these initial probes with advertisements, thereby completing the first control round, and bootstrapping the CPS.

Control Rounds & Validity Horizon

As mentioned earlier, the communication between the GA and RAs occurs in rounds. Here, in order to define the notions of a control period and a validity horizon, we discuss the concept of a control round when the CPS has one GA replica. The formal definition of a control round in a CPS with multiple GA replicas is presented in Chapter 4.

As shown in Figure 3.4, a control round starts with the GA issuing setpoints to the RAs. Each RA receives its setpoint and, if it is not a probe setpoint, implements it. The RA then creates and sends an advertisement to the GA, which collects all received

advertisements, and computes setpoints for the next round.

A control round, therefore, is defined from one setpoint `issue` call at the GA to the next. In real-time control, such rounds are pseudo-periodic, *i.e.,* their duration lies in a range $[T_1, T_2]$ in non-faulty conditions. We define the period of the control round, T_{ctrl}, as the upper bound on the time between two setpoint issue calls (T_2). Equivalently, the duration of the control round can be measured from one advertisement `issue` call to the next, *i.e.,* from the point of view of RAs. This is useful for understanding the validity horizon of advertisements.

Recall from Section 3.3.1 that an RA advertisement contains the feasibility region $\mathcal{F}$ of the resource and its uncertainty $\mathcal{U}$. These values are only valid for a time horizon, referred to as the validity horizon. We re-iterate the meaning of an advertisement through the example in Figure 3.4, in order to motivate the minimum value of the validity horizon.

In Figure 3.4, the RA sends adv_1 containing $\mathcal{F}_1$ and $\mathcal{U}_1$. The GA will use this advertisement to compute sp_2. The RA, with this advertisement, is informing the GA that sp_2 must lie within $\mathcal{F}_1$, and that upon receiving sp_2, it will have an uncertainty of implementation given by $\mathcal{U}_1(sp_2)$. This is expected to hold until the following control round, when the RA receives sp_3. Therefore, the validity of the advertisement fields must last for a duration equal to two control periods. We conclude that the validity horizon T_{val} of an advertisement must be at least $2 \times T_{ctrl}$ (including some buffer). This value is configured in each RA, as mentioned in Section 3.3.1.

The minimum value of the validity horizon, however, does not take into account probe setpoints, which extend the duration of the next implementation setpoint by another period. In Chapter 4, we minimize the number of probe setpoints, by providing a mechanism for the GA to compute implementation setpoints that maintain grid safety, even when a timeout occurs.

Additionally, delay faults (in the GA, SE, RA, or network), can cause a setpoint to be received, by the RA, at a time after the validity horizon of the advertisements (used in its computation) has passed. The two problems this raises, specifically, the delayed setpoint that must be discarded and the absence of a setpoint, are discussed in Section 3.4.3, and are handled in Chapter 5.

Assumptions & Fault Model

We consider that the GA is pre-configured with the grid topology and the number of RAs, their locations, and their identifiers. As the resources and RAs are susceptible to faults, we consider that the GA has the ability to shed (disconnect) resources from the grid, e.g., by disconnecting the breaker at the corresponding bus.

Considering the non-idealities in the rest of the CPS, resulting from the fault models of the other layers and components, the GA is susceptible to omissions of states and advertisements. Additionally, it might receive them late or reordered. Moreover, as mentioned in Section 3.3.2, the GA shares the same fault model as the SE, being susceptible to delay and crash faults. Therefore, setpoints to RAs might be delayed, or not issued at all.

In spite of this, the GA must maintain grid safety, both when controlling a grid connected to an upper-level grid, and when controlling an islanded microgrid. We formally define these requirements in what follows.

3.4 Formal Requirements

As mentioned earlier, the main goal of the CPS is to maintain grid safety. We formally define grid safety as follows. Let $v_j(t),\ i_k(t)$ be the voltage and current at bus j and in line k, respectively, at time t. Let V_j and β_j be the nominal voltage and the maximum allowed voltage deviation at bus j. Let I_k be the ampacity limit for line k.

Definition 3.1 (Grid Safety)**.** *Grid safety is said to hold at time t if, and only if,*

$$\forall\, j,\ V_j - \beta_j \leq v_j(t) \leq V_j + \beta_j \tag{3.1}$$

$$\forall\, k,\ i_k(t) \leq I_k \tag{3.2}$$

Equations 3.1 and 3.2 provide bounds for the bus voltages and line currents. Typical values for voltage bounds are $\pm$10% of the nominal voltage, as specified by IEEE standards [15]. Ampacity limits depend on the dimensions and material of the line.

The grid state evolves both in time and with the implementation of setpoints at the RAs. Real-time control handles the evolution of state, by periodically sending setpoints (every T_{ctrl}) that maintain grid safety for the duration of the period.

In ideal conditions (*i.e.,* when none of the CPS components experience faults), the GA receives the state of the grid from the SE and all the advertisements from the RAs "on time". That is, it can compute setpoints that will be delivered to the RAs within the validity horizon of the advertisements used for computation. These setpoints are implemented by the RAs, thereby maintaining grid safety. Of course, setpoints under these conditions are implementation setpoints, as no timeouts occur.

The challenge is to maintain grid safety in the presence of faults that might affect the various components in all the layers.

3.4.1 Formal Computation & Implementation Model

Recall that the GA computes a vector of setpoints $\mathbf{X}$ that it issues to the RAs. Each RA, upon reception of a setpoint x_i, implements a setpoint according to its latest uncertainty $\mathcal{U}$. Therefore, the implemented setpoint $\hat{x}_i$ is such that

$$\hat{x}_i \in \mathcal{U}(x_i) \tag{3.3}$$

where $\mathcal{U}(sp)$ is the set of setpoints that the resource might implement in the validity horizon of $\mathcal{U}$ if it is instructed to implement sp.

Define $\mathbf{I}(\mathbf{X})$ as the set of vectors of implemented setpoints that are the result of applying Equation 3.3 to the elements of $\mathbf{X}$. For example, consider a grid with a battery and a load. The GA computes a vector of setpoints $\mathbf{X} = \langle 20, 25 \rangle$, instructing the battery to inject 20 A and the load to absorb 25 A. Given the uncertainty function of the battery $\mathcal{U}_b$ and the load $\mathcal{U}_l$ are such that $\mathcal{U}_b(20) = [19, 20]$ and $\mathcal{U}_l(25) = [24, 26]$, then $\mathbf{I}(\mathbf{X}) = \mathcal{U}_b(20) \times \mathcal{U}_l(25)$, *i.e.,* the cross-product of the two uncertainty sets.

Definition 3.2 (Implementation Safety)**.** *Implementation safety is said to hold for a vector of setpoints* $\mathbf{X}$ *if, and only if,*

$$\forall\, \hat{\mathbf{X}} \in \mathbf{I}(\mathbf{X}),\ \textit{implementing}\ \hat{\mathbf{X}}\ \textit{maintains grid safety}$$

That is, $\mathbf{X}$ is said to be safe to implement if implementing it maintains grid safety, even when the uncertainties of the resources are taken into consideration.

3.4.2 Robustness Requirements

Robustness is a property of the controller. Informally, a robust controller is one which produces correct output, despite a non-ideal input. In our case, the GA is said to be robust, if it computes a vector of setpoints that is safe to implement, despite missing, delayed, or reordered input (state and advertisements). Such non-idealities to the input of the GA are the result of faults that might affect the rest of the components in the CPS. Recall from Chapter 2 that the state of the art for robust SEs is solid, hence we do not discuss SE robustness.

We begin by considering input reordering that might result from network reordering, or from non-uniform delays across RAs. Recall that the communication between the GA and RAs is round-based, with advertisements being in response to setpoints in the same round. Thus, when the GA is computing setpoints, it must use the advertisements that belong to the previous round, *i.e.,* the ones that were in response to its previous setpoint. Without such a property, the control round that a setpoint belongs to offers no indication as to which advertisements were used in its computation. This

property, therefore, is necessary to enable the RAs to provide reliable ordering (Definition 3.7), implementing only the setpoints that used the latest advertisements. We formally define this as follows.

Definition 3.3 (Robust Ordering)**.** *Robust ordering is said to hold in a round* $r > 0$ *if, and only if, the computation of implementation setpoints in round* r *takes as input advertisements from round* $r - 1$.

Non-idealities in the rest of the CPS (specifically the network) do not just affect the GA input, they also affect the delivery of setpoints to the RAs. From the point of view of the GA, this affects the set of possible setpoints that the RAs might implement. It also raises other issues, which are discussed in the next subsection on reliability.

Recall that given a vector of setpoints $\mathbf{X}$, $\mathbf{I}(\mathbf{X})$ represents the set of possible vectors that the RAs might implement, considering their uncertainties at the time of reception. We note that $\mathbf{I}(.)$ depends on the uncertainties of the resources advertised in the round preceding the one in which $\mathbf{X}$ was computed. Thus, $\mathbf{I}(.)$ can be represented with a subscript r, indicating that the uncertainties used belong to round $r - 1$. The set of possible vectors in $\mathbf{I}_r(\mathbf{X})$, however, considers that all RAs will receive their setpoints. In the presence of non-idealities, some RAs might not receive their setpoints, and they will continue implementing the latest setpoint they received.

Consider a CPS in round $r - 1$ in which an RA is implementing $\hat{x}_{r-1}$. If this RA receives setpoint x_r in round r, it will implement a setpoint $\hat{x}_r \in \mathcal{U}_{r-1}(x_r)$, as described in Equation 3.3. The uncertainty is the one corresponding to the previous round, as this was the one advertised by the RA and used in the computation of x_r. It is to note that the uncertainty of each resource changes with each setpoint implementation, *i.e.,* in each round, as shown in Algorithm 3.1. However, if the RA does not receive the setpoint in round r, its resource will instead implement $\hat{x}_r \in \mathcal{U}_{r-1}(\hat{x}_{r-1})$, as $\hat{x}_{r-1}$ represents the latest setpoint it was implementing. Although $\mathcal{U}_{r-1}$ would not be computed by the RA in such a case, it still represents the state of the resource at that time.

We define $\mathbf{J}_r(\mathbf{X})$ as the set of vectors of implemented setpoints that are the result of applying Equation 3.3, and considering the possibility of missing setpoints in some RAs, to the elements of $\mathbf{X}$. More precisely, in a given round r, $\mathbf{J}_r(\mathbf{X}_r)$ is the set of all possible vectors constructed as follows. The vector size is equal to the number of RAs. Each element u of the vector corresponds to a setpoint implemented at a unique RA, and is such that:

$$u \in \mathcal{U}_{r-1}(x_r) \cup \mathcal{U}_{r-1}(\hat{x}_{r-1}), \tag{3.4}$$

where (1) $\mathcal{U}_{r-1}$ is the uncertainty of the RA/resource in question, given in the advertisement in round $r - 1$, (2) x_r is the setpoint in $\mathbf{X}_r$ issued to this RA, and (3) $\hat{x}_{r-1}$ is the actual setpoint implemented by the RA/resource at the time of the generation of the advertisement of round $r - 1$.

In other words, for each RA, we consider the union of two sets: (1) the uncertainty set of the issued setpoint in round r, considering it will be received, and (2) the uncertainty set of the previous implementation, considering the issued setpoint is not received. The assumption here, which follows from the model, is that the resource continues implementing the previous setpoint, if it is not instructed to implement a new one. The implementation, however, varies within the uncertainty set. $\mathbf{J}_r(\mathbf{X}_r)$ is, therefore, the cross product of the union of these two sets for each RA. We note that, by construction, $\mathbf{I}_r(\mathbf{X}) \subseteq \mathbf{J}_r(\mathbf{X})$.

Definition 3.4 (Robust Safety)**.** *Robust safety is said to hold for a vector of implementation setpoints* $\mathbf{X}$ *computed in round* r *if, and only if,*

$$\forall\, \hat{\mathbf{X}} \in \mathbf{J}_r(\mathbf{X}),\ \textit{implementing}\ \hat{\mathbf{X}}\ \textit{maintains grid safety}$$

Similar to implementation safety (Definition 3.2), a vector of setpoints $\mathbf{X}$ is robustly safe to implement, if implementing it maintains grid safety, even when the uncertainties of the resources, and the possibility of network losses, are taken into consideration.

We note that robust safety is defined only when a GA computes a vector of implementation setpoints. Hence, it is trivial to avoid violating this property, by never computing implementation setpoints. However, in order to maintain grid safety, the GA must compute and issue such setpoints. Therefore, we introduce the robust availability property, which requires the GA to compute implementation setpoints.

Definition 3.5 (Robust Availability)**.** *Robust availability is said to hold for a GA in a round* r *if, and only if, it can compute a vector of implementation setpoints.*

That is, despite missing or delayed advertisements, a robustly available GA, in a given round, must compute implementation setpoints, *i.e.,* it must not timeout and issue probe setpoints.

Robust ordering, safety and availability are concerned with non-idealities, and must be provided by a robustness mechanism. In addition to these properties, a robust GA must not sacrifice its operation in ideal conditions, *i.e.,* when it receives all its inputs in a given round. That is, it must compute a vector of setpoints similar to that of a GA that does not account for non-idealities.

Definition 3.6 (Robust Optimality)**.** *Robust optimality is said to hold for a GA in a round* r *in which advertisements from all RAs are received if, and only if, its vector of computed setpoints is the same as that computed by a GA in a CPS that implements Algorithms 3.1-3.4.*

Given the robustness properties (Definitions 3.3, 3.4, 3.5, 3.6), a robust GA can be

defined as one that guarantees robust ordering, safety and optimality in all rounds, and preserves robust availability, given the aforementioned guarantees.

In Chapter 4, we provide mechanisms for designing a robust GA for grids in grid-connected mode, namely intentionality clocks and Robuster. In Chapter 8, we describe the additional mechanisms required to handle islanded grids.

3.4.3 Reliability Requirements

Reliability is a also a property of the controller. Informally, a reliable controller is one the faults of which are masked from the rest of the CPS. In our case, the GA is said to be reliable if the RAs receive setpoints within the validity horizon of the advertisements used in their computation. This must happen despite delay and crash faults that might affect the GA, in addition to network faults that might drop, reorder, or delay the setpoints. A similar definition can be given for an SE. In this section, we discuss the requirements for a reliable GA. In upcoming chapters, we highlight how the mechanisms designed for a reliable GA also apply for an SE.

As with the previous subsection, we begin by considering the reordering of setpoints. If an RA last issued an advertisement in round r, it must not implement setpoints from rounds smaller than or equal to r. That is because an advertisement in round r follows the implementation of a setpoint from round r. Only new setpoints, *i.e.,* from higher rounds, have considered this advertisement in their computation, hence must be implemented.

Definition 3.7 (Reliable Ordering)**.** *Reliable ordering is said to hold for a GA if, and only if, the RAs never implement setpoints from a round number smaller than or equal to the round number of the last advertisement they issued.*

Delay faults in the GA, or in the network, can cause setpoints to be received by RAs at a time after the validity horizon of the advertisements used in their computation. Such setpoints are henceforth referred to as *invalid*.

Definition 3.8 (Reliable Validity)**.** *Reliable validity is said to hold for a GA if, and only if, the RAs never implement invalid setpoints.*

As reliable ordering and reliable validity are properties that involve accounting for non-idealities in the network between the GA and the RAs, they cannot be handled by the GA itself. A GA cannot determine that a vector of setpoints is invalid before issuing it, as it might become invalid only at the time of reception. It also cannot account for reordering or message retransmission that might occur in the network. Therefore, these properties can only be provided by a layer at each RA, which discards setpoints that are invalid or out of order.

Additionally, it is trivial to satisfy these two properties by simply discarding all received setpoints. This is accounted for with the following property.

Definition 3.9 (Reliable Availability)**.** *Reliable availability is said to hold for a GA in a round r if, and only if, all the RAs implement a setpoint in that round.*

That is, to preserve reliable availability, the aforementioned layer at each RA must only discard setpoints that are out of order or invalid. However, in the presence of delay and crash faults that affect the GA, the GA might not be able to compute and issue setpoints in each round. In these cases, the GA is referred to as a single point of failure. For this reason, the GA must be replicated, so that if one replica is experiencing a fault, other replicas can maintain control by computing and issuing setpoints.

As discussed in Section 3.1, we consider CPSs with active replication of the GA (see Figure 3.1). This enables handling delay faults and avoids having a single point of failure in the CPS. However, setpoints from multiple replicas might be contradictory, resulting in the split-brain syndrome (discussed in Section 2.3.3). Thus, the following property must also hold.

Definition 3.10 (Reliable Consistency)**.** *Reliable consistency is said to hold for a set of replicated GAs in round r if, and only if, any two vectors of setpoints issued by the GAs in round r have the same setpoint values.*

That is, all replicas that issue setpoints in a given round, issue the same setpoints. Given the reliability properties (Definitions 3.7, 3.8, 3.9, 3.10), a reliable GA can be defined as one that guarantees reliable ordering, validity, and consistency in all rounds, and preserves reliable availability, given the aforementioned guarantees.

In Chapter 4, we describe intentionality clocks, a mechanism that guarantees robust and reliable ordering. In Chapters 5 and 6, we introduce Axo and Quarts, which together, along with intentionality clocks, enable designing a reliable GA.

3.5 Conclusion

In this chapter, we have presented an abstraction of a CPS for real-time control of electric grids, in which we divide the CPS into four layers: the physical layer, the sensing and actuation layer, the network layer, and the control layer. We have presented the model of the components in each layer and discussed the interactions among the various components. We have also described the fault model for each of the components.

The presented models are simple enough to enable (1) analyzing the reliability and robustness problems that exist in real-time CPSs, and (2) characterizing the utility and

performance of the mechanisms introduced to mitigate these problems. The models are also expressive enough to enable the construction of formal requirements for designing reliable and robust CPS controllers. These requirements — robust ordering, safety, availability, and optimality (Definitions 3.3-3.6) and reliable ordering, validity, availability and consistency (Definitions 3.7-3.10) — are the basis for the provably correct solutions presented in the rest of the thesis.

The layering approach we have followed in this chapter was also useful in designing a virtual commissioning tool for real-time control of electric grids — T-RECS. In Chapter 7, we discuss the design of T-RECS, highlighting the same layering approach.

4 Robust Real-Time Control of Electric Grids

Success comes not from having certainty,
but being able to live with uncertainty.
— Jeffrey Fry

In Chapter 3, we introduced the properties required to design a robust GA, namely robust ordering, safety, availability, and optimality. In this chapter, we propose two mechanisms, intentionality clocks and Robuster, that together satisfy these properties.

Intentionality clocks is a labeling scheme tailored for round-based communication. It uses logical clocks to assign a label to each message exchanged between the GA and the RAs. That is, each setpoint and advertisement is assigned a label. This label reflects the round number that this message belongs to. As mentioned in Chapter 3, the notion of a round number is non-trivial in the presence of GA replication. We define this notion in this chapter.

Intentionality clocks is used to satisfy robust ordering (Definition 3.3), ensuring that the GA only uses, in its computation, advertisements that belong to the previous control round. This mechanism also satisfies reliable ordering (Definition 3.7), by discarding setpoints from previous rounds, before they reach the RAs. Recall from Chapter 3 that these properties are trivial to satisfy, by never computing at the GA for robust ordering, and by discarding all setpoints at the RA for reliable ordering. However, we show that intentionality clocks preserves robust and reliable availability (Definitions 3.5, 3.9), ensuring that in-order setpoints are always computed and never discarded.

Robuster is an advertisement generation and setpoint computation mechanism. In Robuster, the RAs create advertisement fields with both short-term and long-term validity horizons. These are sent to the GA, which can then use the short-term fields when they are available, maintaining robust optimality (Definition 3.6), and store the

long-term fields for later use. If an advertisement is not available in a certain control round from some RAs, the GA can use the long-term fields of previous advertisements from those RAs, if these are still valid. Thus, Robuster enables the GA to compute robustly safe setpoints in more control rounds, thereby increasing robust availability (Definitions 3.4-3.5).

In this chapter, we first present intentionality clocks in Section 4.1. We motivate the need for a labeling mechanism (Section 4.1.1), formally define the concept of control rounds and round numbers (Section 4.1.2), highlight the labeling mechanisms in the literature (Section 4.1.3), present the design of intentionality clocks (Section 4.1.4), and formally prove its guarantees (Section 4.1.5). Then, in Section 4.2, we present Robuster, which uses intentionality clocks to guarantee robust safety and optimality. We introduce the problem Robuster is solving (Section 4.2.1), present the method (Section 4.2.2) and the formal proofs (Section 4.2.3), and provide several examples of how Robuster can generate advertisements for different resources (Section 4.2.4). Finally, in Section 4.3, we evaluate these mechanisms through experiments in a grid deployment, and we show that they perform better than state-of-the-art alternatives. We offer concluding remarks in Section 4.4.

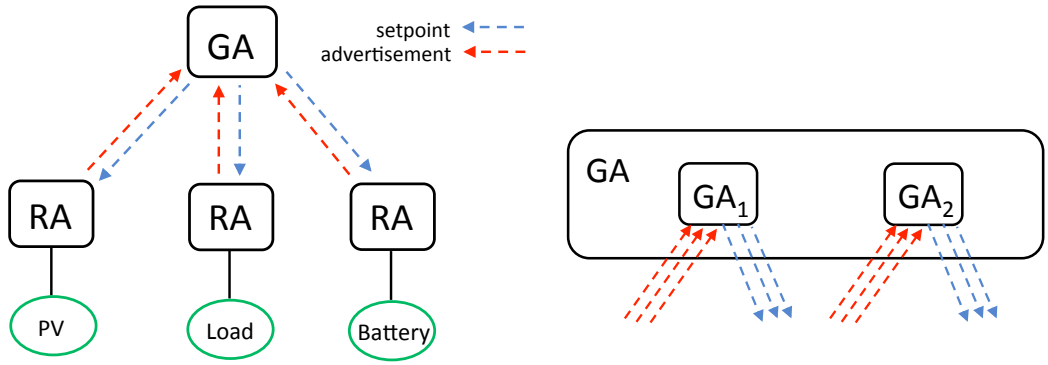
4.1 Robust and Reliable Ordering

4.1.1 Motivation

We consider the communication exchange between a GA and one or more RAs in a CPS, as shown in Figure 4.1a. Such an exchange occurs in rounds, referred to as *control rounds*. Recall from Figure 3.4 that a control round begins with the GA issuing setpoints, which is followed by the RAs receiving these setpoints, implementing them, generating advertisements, and issuing them to the GA, which in turn uses the received advertisements to compute new setpoints. The issuing of the newly computed setpoints signals the end of the previous control round, and the beginning of a new one. This is formally defined in Section 4.1.2.

On the implementation of a setpoint by an RA, the state of the resource, and consequently that of the grid, is altered. The new state is encapsulated in the subsequent advertisements sent to the GA. The GA can then use the advertisements from all the RAs to recreate the new state of the resources and grid, and subsequently compute setpoints that maintain grid safety.

If the state of the resources is different than the one the GA considers when computing, then grid safety might be violated. Therefore, the computation of setpoints in a round r must only take as input advertisements from round $r-1$, as advertisements from earlier rounds no longer reflect the actual state of the grid and electric resources.



(a) Non-replicated GA (b) Replicated GA with two replicas

Figure 4.1 – Communication between GA and RAs in a CPS

Hence, robust ordering must be satisfied (Definition 3.3).

Furthermore, an RA must only implement setpoints that were computed given the current state of its resource. That is, only certain setpoints — those the computation of which took as input the advertisement last issued by this RA — must be implemented. Hence, reliable ordering must be satisfied (Definition 3.7).

In order to guarantee the aforementioned properties, messages exchanged between the GA and RAs must be labeled with the round number they belong to. The labeling scheme designed for this purpose must maintain correct labeling in the presence of non-idealities and faults. These include the following: (1) Communication network non-idealities, namely message losses, delays, retransmissions, and reordering (as discussed in Section 3.2.3), (2) resource and RA faults, including crashes and unresponsiveness (as discussed in Section 3.3.1), and (3) GA crash and delay faults (as discussed in Section 3.3.3).

Additionally, as the GA is a single point of failure, susceptible to crash and delay faults, it is often replicated, as discussed in Chapter 3. The GA replication, shown in Figure 4.1b, is presented in Chapters 5 and 6. In this chapter, we design the labeling scheme — intentionality clocks — to be correct in the presence of GA replication.

Note that, in this discussion, we use the terms “state of the resource” and “state of the grid” as a proxy for the state of the RAs. While the state of the physical layer is continuous and evolving, the state of the RAs is discrete and only changes upon a setpoint implementation. In real-time control of electric grids, the rate at which setpoints are issued is comparable to the dynamics of the underlying grid. Thus, the evolution of the state of the grid between two setpoint implementations (*i.e.,* two control rounds) is minimal, thereby justifying our usage of the term. In Section 4.2, we revisit this issue and handle the evolution of state that occurs between control rounds, as we discuss the validity horizon of advertisements.

4.1.2 Control Rounds & Round Numbers

As mentioned in Chapter 3, a control round in a CPS with non-replicated GAs consists of the following four steps: (1) It begins with the GA issuing setpoints to the RAs, then (2) the RAs receive and implement these setpoints, then (3) the RAs issue advertisements to the GA, and finally, (4) the GA uses these advertisements to compute new setpoints. The new setpoints are issued in the next control round. All messages exchanged in steps 1-4 are said to belong to a single control round.

However, in the presence of GA replication, multiple GA replicas are computing and issuing setpoints to the RAs. Therefore, the above illustration can no longer uniquely define a control round. Instead, we require a more elaborate definition of control rounds and round numbers in such a case.

We make use of the property that GA replicas, given the same input, (*i.e.,* the same advertisements and internal state, reflecting the same state of the grid at a given point) will issue the same output setpoints. This is the reliable consistency property (Definition 3.10), which can be guaranteed as described in Chapter 6. This property enables us to define the notion of control rounds and round numbers as follows.

The following three rules define a control round in a CPS with replicated GAs:

1. Upon booting, the initial setpoints issued by a GA replica belong to round 0.

2. An advertisement issued by an RA belongs to round number r, where r is the maximum round number of a setpoint received by this RA, prior to the generation of the advertisement.

3. A setpoint issued by a GA replica belongs to round number $r+1$, where r is the maximum round number of an advertisement received by this GA replica, prior to the computation of the setpoint.

A control round, therefore, might be perceived differently at different GA replicas. An external observer can assign round numbers to messages by following a given execution trace of the CPS. However, the challenge is for distributed agents, with a limited view of the CPS, to assign such round numbers. Intentionality clocks, presented in Section 4.1.4, is a mechanism that enables distributed agents to assigned labels to messages, which reflect the round numbers they belong to.

We note that, although control rounds are perceived differently at different GA replicas, the control round period T_{ctrl} remains valid. T_{ctrl} is defined as the upper-bound between two setpoint issue calls at a *single* GA replica, in non-faulty conditions.

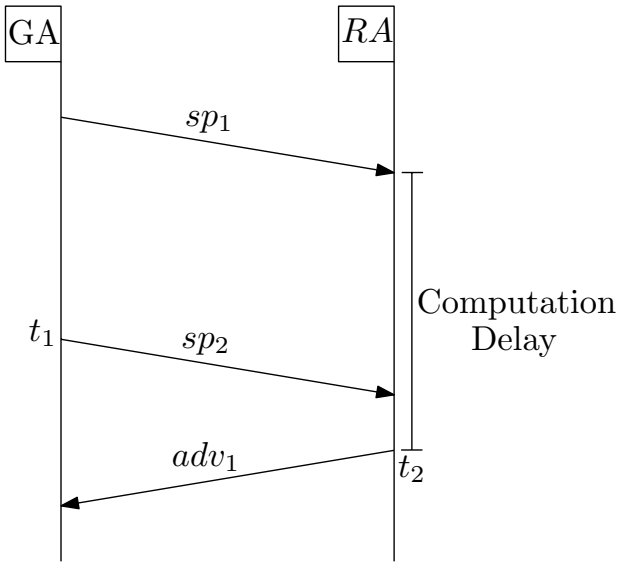


Figure 4.2 – Shortcomings of temporal order in the presence of delays

4.1.3 Labeling in the Literature

Intentionality clocks is the first mechanism that assigns round numbers to messages in CPSs with replicated controllers. Here, we summarize the different bodies of work that address the problem of labeling messages to achieve ordering in distributed systems, and we note their shortcomings when applied to round-based communication.

Temporal Order: Labeling via Timestamps

As CPSs are real-time systems, they generally keep track of physical time. To maintain synchronized global time on all software agents, their physical clocks are synchronized either using GPS-based synchronization [80] or network-based synchronization (e.g., NTP [78], PTP [79]). These time-synchronization solutions provide a synchronization accuracy δ that ranges from sub-microsecond to one millisecond.

CPSs often leverage the availability of a synchronized time to reason about the temporal ordering of messages. However, as we see in the following example, temporal ordering does not coincide with round-based ordering in the presence of delays.

Consider a CPS in which a GA is communicating with several RAs, only one of which is shown in Figure 4.2. We will consider a perfect time-synchronization ($\delta = 0$). Due to a delay at the RA shown, adv_1 was generated a time $t_2 > t_1$, with t_1 being the time at which the GA computed the setpoints for round 2. Therefore, based on the temporal ordering, adv_1 occurs after sp_2. A GA relying on timestamps and temporal ordering might use adv_1 in the computation of setpoints for round 3, thereby violating robust ordering.

Note that, in the earlier example, the GA computed setpoints for round 2 without the advertisement from the delayed RA. This is due to a timeout triggering the computation at the GA. Recall, from Section 3.3.3, that timeouts result in probe setpoints being issued. The GA, therefore, might consider that adv_1 was a response to the probe. In this chapter (Section 4.2), we present a mechanism to enable computing implementation setpoints when timeouts occur. In that case, the GA might consider that adv_1 was a response to the issued implementation setpoint.

We conclude that, on its own, timestamping is not sufficient to capture the round-based ordering between the GA and RAs. In Section 4.1.4, we present intentionality clocks, which provides such an ordering by using logical clocks instead of physical clocks. This mechanism does not require synchronized physical time.

We note that the communication model between asynchronous sensors and the SE is not round-based. Hence, measurements sent by asynchronous sensors to the SE can be labeled and ordered via timestamps. As mentioned in Section 3.2.2, we assume for this purpose that the sensors are time-synchronized with one another, and that their time-synchronization enables the SE to time-align their measurements. That is, based on the timestamps, the SE can infer a round associated with the measurements. Such a round is independent of the round of the message exchange between GAs and RAs.

Causal Order: Lamport Clocks & Vector Clocks

Ordering in traditional distributed systems is done through a causal order, captured by the *happened-before* relation [81]. Providing a causal order adheres to what is referred to as the real-time causal consistency semantic [136]. This is achieved through one of several mechanisms such as timestamps, Lamport clocks [81], or vector clocks [83].

As mentioned earlier, timestamps cannot capture the round-based ordering between GAs and RAs in the presence of delays. Here, we show that logical clocks providing a causal order also cannot capture the same, in the presence of GA replication.

Lamport clocks and vector clocks are complementary examples of logical clocks that provide causal order. Lamport clocks describe the happened-before relation, *i.e.,* if message m_1 happened before message m_2, then the label of m_1 will be less than the label of m_2. The complementary approach using vector clocks infers this relation, *i.e.,* if the label of m_1 is less than that of m_2, then m_1 happened before m_2. However, vector clocks can only provide a partial order between messages, as two labels might be incomparable. This does not suit the round-based ordering of CPSs we aim to capture, as a total order between messages is required for such an order. Therefore, we do not consider vector clocks as an avenue in what follows.

Intentionality clocks use a scalar clock inspired from Lamport clocks. In Section

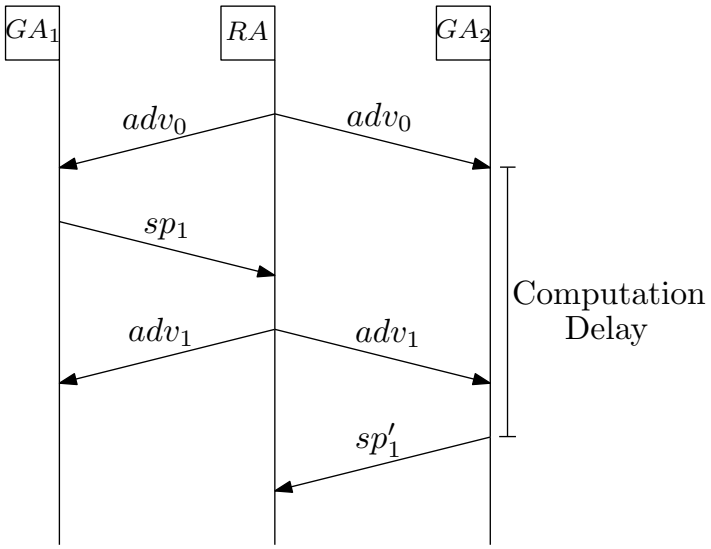


Figure 4.3 – Shortcomings of causal order in the presence of GA replication

4.1.4, we discuss the differences between intentionality clocks and Lamport clocks in greater detail. In the following example, we illustrate the shortcomings of the causal order provided by Lamport clocks when GA replication is considered.

Consider a replicated GA, with two replicas GA_1 and GA_2 controlling a single RA, as shown in Figure 4.3. GA_1 and GA_2 receive the advertisement adv_0 from the RA. This advertisement is used by each replica in the computation of a setpoint, resulting in sp_1 and sp'_1 issued by GA_1 and GA_2, respectively. In such a scenario, sp_1 and sp'_1 belong to the same control round. sp_1 is received by the RA, and its implementation results in adv_1 being issued. However, due to a delay at GA_2, adv_1 is received before sp'_1 is issued. Under the causal order, adv_1 is said to have happened before sp'_1, and a labeling mechanism based on Lamport clocks assigns adv_1 with a label smaller than that of sp'_1. However, adv_1 is in response to sp_1, the setpoint issued by the other replica in the same control round as sp'_1. Under this round-based ordering, adv_1 belongs to the same control round as sp'_1, and therefore must be considered as having happened after it, as advertisements happen after setpoints of the same round.

The contrast between causal and round-based ordering, presented in the example above, shows that Lamport clocks cannot be used for labeling messages in a CPS.

Order Without Labeling

Several industrial solutions circumvent the ordering problem by using frameworks, such as the timely computing base (TCB) [87] or the time-triggered architecture (TTA) [90], which provide synchrony guarantees. However, as mentioned in Chapter 2, using such frameworks requires specialized hardware, in addition to a complete redesigning of the application to fit the framework. In contrast, we propose a solution that requires

neither. This facilitates deployment in existing CPSs.

Another approach is to hold consensus between the GAs and RAs, to agree on which messages belong to each round. This handles the aforementioned non-idealities, faults, and GA replication. However, consensus is unsuitable for real-time CPSs, due to its high latency overhead, as mentioned in Chapter 2. In contrast, intentionality clocks performs simple operations, hence incurs negligible latency overhead.

4.1.4 Intentionality Clocks Design

Intentionality clocks is a labeling mechanism that assigns labels to outgoing messages, and that uses message labels to selectively discard incoming messages. To do this, this mechanism maintains eventually synchronized logical clocks across all software agents, namely GAs and RAs. Although it relies on eventual synchronization, intentionality clocks guarantees robust and reliable ordering, as shown in Section 4.1.5.

Intentionality clocks is an adaptation of the Lamport clocks abstraction [81]. It is designed to accommodate the round-based communication between GAs and RAs in CPSs. Being an adaptation of Lamport clocks, it uses scalar clocks, whereby each agent maintains a local logical clock C, which is a non-negative integer.

The logical clock at each agent serves two purposes. (1) The value of the logical clock immediately before sending a message is used to assign a label of that message. (2) The value of the logical clock immediately after receiving a message is used to ascertain whether or not to discard that message, based on its label.

The labels are assigned such that they reflect the control round number that the message belongs to. Messages are discarded at the GAs and RAs such that robust and reliable ordering are satisfied. Additionally, only messages that violate robust and reliable ordering are discarded, so that robust and reliable availability is preserved.

The design of intentionality clocks is given in Algorithms 4.1-4.3. The rules governing intentionality clocks are the following:

1. The value of the logical clock C at an agent, immediately before an `issue` call, is assigned as the label of outgoing messages in that call.
2. At an RA with logical clock C, messages received with label $\ell \leq C$ are discarded.
3. At a GA with logical clock C, messages received with label $\ell < C$ are discarded.
4. On reception of a message with a label ℓ at a software agent with logical clock C, C takes the maximum value of C and ℓ.
5. At a GA, the logical clock is incremented once before each computation.

The main distinction between intentionality clocks and Lamport clocks is that, in intentionality clocks, the logical clock is only incremented at a GA, and only when the GA performs a computation. This enables the software agents to infer the control round that the message belongs to. In contrast, Lamport clocks increment the logical clock at each agent, every time that agents receives a message. First, this causes the agent to lose the notion of a control round, as messages are not labeled accordingly. Second, with Lamport clocks, in the presence of GA replication and network retransmissions, the reception of multiple messages in the same control round causes the clock at the receiving agents to diverge indefinitely. Consequently, Lamport clocks cannot be used to guarantee robust and reliable ordering.

GA Design

Algorithm 4.1 describes the design of a GA with intentionality clocks. This is the same as Algorithm 3.4, with the additions in red serving to integrate intentionality clocks. Refer to Section 3.3.3 for an overview of the specifics of that algorithm. Together with the complementary part at the RAs (see Algorithm 4.3), this algorithm guarantees robust and reliable ordering, and preserves robust and reliable availability.

Each GA replica maintains a logical clock C, initialized to zero, when that GA boots or reboots (line 5). C represents the current control round of the CPS, from the point of view of this GA. Upon reception of an advertisement with label ℓ, this advertisement is aggregated into the set of received advertisements $\mathcal{A}$, along with its label (line 13). Note that although we do not explicitly discard advertisements with labels less than C, we do not use them in computation, as shown later.

Following rule #4, C is set to the maximum of ℓ and C (line 14). This serves to re-synchronize delayed GA replicas. It is also applicable in the case of no replication, when the GA reboots, which will be further discussed when describing Algorithm 4.3.

In the computation thread, the function `choose_advertisements` is called, passing it as parameters $\mathcal{A}$ and C. This function, described in Algorithm 4.2, essentially returns the subset of received advertisements with label equal to C, as these are the only advertisements that can be used in the computation of setpoints for round $C+1$. This is in accordance with rule #3 of intentionality clocks, as advertisements with smaller labels will not be used in subsequent computations. The chosen subset of advertisements $\mathcal{A}'$ is passed as a parameter to the `ready_to_compute` function (line 24), which is left unchanged. Note that in this case, each element in $\mathbf{Z}_{\mathcal{A}}$ will either contain an advertisement with label C, or be empty otherwise.

The algorithm proceeds as described in Section 3.3.3. That is, if the $ready$ flag is set to true, either because of a timeout or because the advertisements in $\mathcal{A}'$ are sufficient to begin a computation, then a computation is performed (lines 25-29).

Algorithm 4.1: Abstract model of a GA with intentionality clocks. The parts in red are added to Algorithm 3.4

```
1  A ← ∅;                                   // set of advertisements received from RAs
2  S ← ∅;                                   // set of states received from the SE
3  Z_A ← [ ];                               // vector of advertisements used in a computation
4  H ← ∅;                                   // internal state of the GA
5  C ← 0;                                   // logical clock on this GA
6
7  on initialization
8  |  X ← ∅;                                // initialize vector of setpoints to probes
9  |  issue(X, C);                          // send probes to the RAs
10 end;
11
12 on reception of an advertisement adv with label ℓ from an RA i
13 |  A ← A ∪ {(adv, i, ℓ)};                // aggregate received advertisements
14 |  C ← max(C, ℓ);
15 end;
16
17 on reception of a state st with timestamp T from the SE
18 |  S ← S ∪ {(st, T)};                    // aggregate received states
19 end;
20
21 repeat
22 |  A' ← choose_advertisements(A, C);
23 |  T ← current time;                      // get the current time
24 |  ready, timeout, Z_A, st ← ready_to_compute(A', S, T);
25 |  if ready then
26 |  |  C ← C + 1;
27 |  |  X, H ← compute_setpoints(Z_A, st, timeout, H);
28 |  |  issue(X, C);                       // send X to the RAs
29 |  end
30 forever;
```

Algorithm 4.2: `choose_advertisements`$(\mathcal{A},\ C)$ function in the GA

```
1 A' ← ∅;
2 for each (adv, i, ℓ) ∈ A do
3 |  if ℓ = C then
4 |  |  A' ← A' ∪ {(adv, i)};
5 |  end
6 end
7 return A';
```

Before the computation, C is incremented[1] (line 26), in accordance with rule #5. After the computation, as mentioned in rule #1, the setpoints are issued with a label of C.

The selection of the state of the grid st, to be used in computation, is left for the GA to decide in its `ready_to_compute` function. This selection depends on the internals of the GA computation, and how it uses the state of the grid. We note that in certain situations, a specific state of the grid — one which corresponds to the state of the resources, as captured by the advertisements in that round — must be used. We discuss how to handle such a situation in Chapter 6.

Note that timeouts are still a part of this model. Hence, probe setpoints might still be issued. As probes in this case will have a label corresponding to the actual round number, an RA receiving such a probe does not violate reliable ordering. Moreover, probes are not implemented by RAs, so it is safe not to discard them. However, probes are also issued by newly booted or rebooted replicas, with a label of 0. Such probes would be discarded at the RA if the CPS has already been bootstrapped. We discuss this case in what follows.

RA Design: Masker

At each RA, we propose an external component, which maintains the logical clock, intercepts the received setpoints, discards out-of-order setpoints, and forwards in-order ones. We call this component the *masker*, and its design is presented in Algorithm 4.3.

The masker is a component or a layer that resides at each RA. It maintains the logical clock C of the RA, initialized to -1 (line 1) and the last advertisement issued by the RA (line 2). The masker intercepts communication to and from the RA.

When the masker intercepts a setpoint with a label ℓ from a GA replica to the RA, it discards it if $\ell \leq C$. Otherwise, the setpoint is forwarded to the RA (line 10). After this step, the RA is assumed to have implemented that setpoint. Therefore, the logical clock is updated to reflect the round number of the last setpoint implemented by the RA (line 11). Then, the masker waits to intercept the advertisement issued by the RA to the GA (line 13), and issues that advertisement, assigning it a label of C (line 14). This procedure follows rules #1, #2, and #4 of intentionality clocks.

The special case, when $\ell = 0$ and $\ell \leq C$, occurs when the masker has already received and forwarded some setpoints to the RA ($C > -1$), and receives a setpoint with label 0 (line 15). This case arises (1) when a GA is rebooted, initializing its logical clock to zero and sending probe setpoints, (2) when a GA replica newly joins the system,

[1]The logical clock is constantly increasing. In theory, this might cause an overflow during deployment. However, in practice, a 64-bit counter that is incremented once every millisecond (a comfortable lower bound on T_{ctrl}) requires over 500 million years to overflow.

Algorithm 4.3: Masker component at an RA

```
1  C ← −1;                                  // logical clock on the corresponding RA
2  adv ← ∅;                                 // last issued advertisement by the RA
3
4  on reboot
5  |  C ← stored C;                         // read value of C from hard storage
6  end;
7
8  on reception of a setpoint sp with label ℓ from the GA
9  |  if ℓ > C then
10 |  |  forward(sp);                       // forward sp to the RA
11 |  |  C ← ℓ;
12 |  |  store C;                           // store value of C in hard storage
13 |  |  adv ← receive_from_RA();           // wait for adv from the RA
14 |  |  issue(adv, C);                     // issue adv to the GA
15 |  else if ℓ = 0 then
16 |  |  issue(adv, C);                     // re-issue previous advertisement to the GA
17 |  end
18 end;
```

and does the same, or (3) when a retransmission causes an initialization probe to be sent again. In such cases, the masker re-issues the last issued advertisement, assigning it the same label that was assigned when it was last issued (line 16). This serves to re-synchronize delayed GAs, by providing them with the latest advertisement and round number. It also responds to probes without invoking the RA, thereby maintaining reliable ordering.

As the RA must never receive (and implement) setpoints with labels smaller than the round number of an implemented setpoint, the masker must keep track of the highest round number seen. This must hold in spite of faults that might cause the masker to reboot. Although this is not explicitly part of the system model, we account for this possibility. To do so, we store the value of C in hard storage (*i.e.,* on disk) after each update (line 12), and read this value on reboot (line 5).

Note that although we say that the masker at each RA issues an advertisement to "the GA", we consider GA replication. Thus, we consider that the `issue` call sends the message to all GAs, either using multicast or several unicasts. This can be done at the masker without giving the RA knowledge of the GA replication, thereby maintaining the design of the RAs unchanged.

4.1.5 Formal Guarantees

We formally prove that intentionality clocks, implemented in Algorithms 4.1-4.3, guarantees robust and reliable ordering. Then, we show that robust and reliable availability

are preserved with intentionality clocks. For these, we make the assumption that, in the presence of GA replication, setpoints issued by different replicas with the same label to the same RA have the same value. That is, we assume that reliable consistency holds (Definition 3.10). This allows us to consider advertisements in a given round to be in response to setpoints of that round from any GA replica. In Chapter 6, we show how this can be guaranteed.

Theorem 4.1 (Robust & Reliable Ordering)**.** *A CPS that implements Algorithms 4.1, 4.2 and 4.3 guarantees robust and reliable ordering. A message label provided by intentionality clocks is the round number of the message.*

Proof. The proof uses strong induction on the control round r. We prove the following:
(1) The GAs do not violate robust ordering.
(2) The maskers do not violate reliable ordering.
(3) Setpoints with label $\ell = r$ belong to round r.
(4) Advertisements with label $\ell = r$ belong to round r.

Base Case:

Recall, from Definition 3.3, that robust ordering is defined for control rounds $r > 0$.
The computation of setpoints in the initial round ($r = 0$) does not use advertisements, as they are probe setpoints with label 0 (Algorithm 4.1, lines 7-10).
Initially, the RAs have not issued any advertisement, so the reception of these probes in round 0 does not violate reliable ordering (Definition 3.7).
These probes are forwarded to the RAs, and advertisements with label 0 are issued in round 0 to the GAs (Algorithm 4.3, lines 10-14).

For a summary of the base case:
(1) The GAs do not violate robust ordering in round $r = 0$.
(2) The maskers do not violate reliable ordering in round $r = 0$.
(3) Setpoints with label $\ell = 0$ belong to round $r = 0$.
(4) Advertisements with label $\ell = 0$ belong to round $r = 0$.

Inductive Step:

We proceed to show that these four statements hold for round r, assuming they hold for all rounds $r' < r$.

From Algorithm 4.1 lines 22, 26, 28, we see that setpoints issued with a label $\ell = r$ use, in their computation, advertisements with label $\ell = r - 1$.
From the inductive assumption (4), these advertisements belong to round $r - 1$.
Recall from Section 4.1.2 that setpoints using advertisements from round $r - 1$, belong to round r.
Therefore, setpoints with label $\ell = r$ belong to control round r.

This proves statement (3) and, consequently, statement (1).

From Algorithm 4.3 lines 11, 14, the masker assigns a label ℓ to advertisements that are in response to setpoints with label ℓ.
Using statement (3), if $\ell = r$, then these setpoints belong to round r.
Therefore, the advertisements belong to round r.
This proves statement (4).

From Algorithm 4.3 line 9, the RAs receive setpoints only if their label is larger than C.
From Algorithm 4.3 lines 11, 14, C is the label of the last advertisement issued.
Using statements (3) and (4), reliable ordering is maintained in round r.
This proves statement (2). □

As mentioned in Chapter 3, robust ordering can be trivially satisfied by never computing at the GA. Similarly, reliable ordering can be trivially satisfied by always discarding setpoints, before implementation, at the RA. However, such a trivial mechanism violates robust and reliable availability (Definitions 3.5, 3.9).

Robust and reliable availability cannot be guaranteed, as they depend on several factors. For example, a network partition would cause the RAs to never receive setpoints, thereby violating reliable ordering. Similarly, all GA replicas might crash, thereby violating robust ordering. Therefore, the best a labeling scheme can achieve is preserving robust and reliable availability if they hold.

However, as seen in Section 4.1.4, intentionality clocks discards some advertisements rather than use them in computation, and it discards some setpoints rather than forward them to the RAs for implementation. In the following two theorems, we show that intentionality clocks only discards advertisements and setpoints that violate robust and reliable ordering. We say that this preserves robust and reliable availability.

Theorem 4.2 (Robust Availability: Intentionality Clocks)**.** *Intentionality clocks only discards advertisements from a computation if using these advertisements violates robust ordering.*

Proof. In Algorithm 4.1, line 22, the `choose_advertisements` function discards some advertisements.
These advertisements are discarded so they will not be used in computation of setpoints with label $C + 1$ (lines 26-27).
Hence, these setponits belong to round $C + 1$ (Theorem 4.1).
In Algorithm 4.2, we see that the only advertisements discarded are ones with label $\ell < C$.
Hence, these advertisements belong to rounds less than C (Theorem 4.1).
Therefore, using these advertisements would violate robust ordering. □

Theorem 4.3 (Reliable Availability: Intentionality Clocks)**.** *Intentionality clocks only discards setpoints if implementing these setpoints violates reliable ordering.*

Proof. In Algorithm 4.3 line 9, the masker discards setpoints with label $\ell \leq C$.
C is the highest label of an advertisement issued (lines 11, 14).
Hence, discarded setpoints belong to round numbers smaller or equal than the round number of the last advertisement issued by the RA (Theorem 4.1).
Therefore, implementing these setpoints violates reliable ordering. □

4.2 Robust Safety, Availability, and Optimality

4.2.1 Overview

In Section 4.1, we handled the case of message reordering affecting the GA and RAs. Here, we handle the case of message losses or delays. During long-term deployment of a CPS, and due to the reliance of real-time CPSs on COTS components [32], communication network non-idealities arise. This, in addition to faults affecting the resources and the software agents, might cause certain messages to be omitted, lost, or delayed. Such non-idealities increase the uncertainty in the operation of the GA, and limits its ability to maintain a feasible control over the grid resources, *i.e.,* to maintain grid safety. Given the mission-critical nature of real-time grid control, and the possible consequences that might arise in case of failure [12], it is essential for GAs to be robust in the presence of such non-idealities and uncertainties.

As an example, consider a grid-connected microgrid, as shown in Figure 4.1a, that consists of a battery, a PV panel, and a load. Let us suppose that the GA has the objective of providing primary frequency support to the main grid by controlling the battery power flow injection/absorption. It also needs to ensure that the bus-voltage and line-current magnitudes are within the safety limits, despite the stochastic profile of the PV injections and load consumption. A quick change in the frequency signal, PV production, or load consumption, coupled with a loss of advertisements, renders the GA unaware of the present and future state of the grid resources, and thus incapable of computing setpoints that maintain grid safety in the next round.

Several controller designs presented in the literature assume an ideal communication network [8, 59, 60]. However, despite advances in improving the resiliency and reliability of communication in power grids [52, 137, 138], non-idealities cannot be eliminated due to the stochastic nature of wide-spread communication networks. This is especially true for real-time applications, in which low latencies are required.

As mentioned in Chapter 2, the state of the art on SE robustness is solid. Therefore, we consider that state-of-the-art techniques for handling missing state (from the SE) at

the GA can be used. In this section, we reiterate the issue and the solution to missing setpoints at the RAs, that was discussed in Chapter 3. The main focus, however, is on handling missing advertisements at the GA.

Recall, from Chapter 3, that the `ready_to_compute` function at the GA decides when the GA has sufficient information to begin a computation. More specifically, it checks whether the set of received advertisements can be used to compute a vector of implementation setpoints that maintain grid safety. When advertisements are occasionally lost to the GA, the `ready_to_compute` might deem the set of advertisements insufficient to begin a computation, eventually signaling a timeout. Timeouts result in probe setpoints being issued, decreasing the availability of the CPS.

In Section 4.1, we limit the advertisements that a GA can use in its computation. More specifically, to compute a setpoint in round r, a GA must only use advertisements from round $r-1$. The reason behind this restriction is that the validity horizon of advertisements is such that they will not be valid beyond the next round, after they are generated and issued at the RAs. Increasing the validity horizon of all advertisements would allow the GA to use advertisements from earlier rounds. However, as mentioned in Chapter 3, this results in advertisements with larger uncertainties, which affects the optimality of the computation. This is not desirable in the average case, when all advertisements are received. Formally, this violates robust optimality (Definition 3.6).

We propose Robuster, a mechanism for generating advertisements and computing setpoints that maintains robust safety and optimality, and increases robust availability (Definitions 3.4-3.6). Robuster uses a technique similar to buffered actuation [70,71,72] used by SEs. It enables the RAs to compute advertisement fields with both short-term and long-term validity horizons. Short-term fields are valid for one control round, as earlier, and are used by the GA when they are present, thereby maintaining robust optimality. Long-term fields are valid for the n controls rounds, and can be used by the GA in rounds when an advertisement from that RA is missing. This enables the GA to compute implementation setpoints in those rounds, providing robust availability. Robust safety is not violated, as the validity horizon of the long-term fields extends to those control rounds.

We note that by allowing the GA to use advertisements from older rounds, we are relaxing the robust ordering property. However, we ensure that the advertisements used in a given computation round are valid for that round. We also note that we refer to validity horizons in terms of both time and control rounds. From the point of view of RAs, validity horizons are computed to be valid for a certain horizon measured in time, for example $2 \times T_{ctrl}$ for the short-term fields. This time is selected such that it represents a given number of control rounds, one control round in this case. From the point of view of the GA, validity horizons are only treated in terms of rounds, assuming no delays have occurred or will occur. In Chapter 5, we discuss how the RA discards

setpoints that are received after the validity horizon (in terms of time) has passed.

4.2.2 Robuster Design

Handling Missing Setpoints

Recall, from Chapter 3, the definition of $\mathbf{J}(\mathbf{X})$ (Equation 3.4). Given for a vector of implementation setpoints $\mathbf{X}$, $\mathbf{J}_r(\mathbf{X})$ is a set that includes all the possible vectors of actually implemented setpoints at the RAs, after $\mathbf{X}$ is issued by the GA. For each RA, this includes the uncertainties associated with its uncertainty field $\mathcal{U}$, in addition to the possibility of the setpoint in $\mathbf{X}$ not being received at the RA.

Given that the setpoints represent power, voltage, or current values, that are real numbers, the cardinality of the set of possible implemented setpoints (u) for each RA might be infinite. In turn, the set of possible vectors of implemented setpoints ($\mathbf{J}_r(\mathbf{X}_r)$) might have infinite cardinality. However, we do not claim that the GA must store or even compute this set. We state that the GA, given the required input, can compute a vector of setpoints $\mathbf{X}_r$, such that robust safety holds. That is, the GA can guarantee that every possible vector in $\mathbf{J}_r(\mathbf{X}_r)$ maintains grid safety if it represents the actual implementation. This, so-called *admissibility test* in the literature, can be performed without computing the infinite set, and in real-time [139, 140].

In what follows, we assume that the `compute_setpoints` function at the GA performs this admissibility test. If the test fails, the GA might respond by sending probe setpoints, waiting for the conditions to change, or by shedding resources in order to ensure grid safety. If the test succeeds, then the vector of computed setpoints guarantees robust safety. This, however, only holds if the validity horizon of the advertisements used in computation is valid. Thus, for each RA, the advertisement from the previous control round is required to compute a robustly safe vector of implementation setpoints. Next, we show how we can construct and use long-term fields in advertisements, in order to enable the GA to compute setpoints that maintain robust safety, even when some advertisements are missing.

Properties of Long-Term Fields of an Advertisement

Here, we consider that the feasibility region is a set, and that the uncertainty is a set-valued function with its domain being values from the feasibility region. Such a representation is generic, as any form of feasibility and uncertainty can be transformed into this representation. We note that we do not discuss the preference of the resource. We do not provide guarantees on the preference of resources when advertisements from those resources are missing. Instead, in those cases, we only guarantee that robust safety is maintained, and that robust availability is preserved.

We augment the advertisements at the RAs to include three new fields: (1) the measured implemented setpoint at the resource ($\hat{x}_r$ in Equation 3.4) after the RA calls `implement_setpoint` (Algorithm 3.1), (2) a long-term feasibility region $\mathcal{F}^n$ valid for the next $n > 1$ control rounds, and (3) a long-term uncertainty function $\mathcal{U}^n$ also valid for the next n rounds.

The original feasibility region ($\mathcal{F}$) and uncertainty function ($\mathcal{U}$) are henceforth referred to as *short-term fields*. As mentioned earlier, the short-term fields have an associated validity horizon $\lambda = 2 \times T_{ctr}$, measured in time. In other words, $\mathcal{F}(t)$, measured and computed at time t, is only valid in the range $[t,\ t+\lambda]$. This is used by the masker at the RA in Chapter 5 to discard setpoints that are received outside that range. We say $\mathcal{F}(t) = \mathcal{F}[r]$ if $\mathcal{F}(t)$ belongs to an advertisement from round r. The validity horizon for short-term fields is one control round. This is used by the GA to ascertain whether an advertisement is still valid to be used in a computation.

Short-term fields estimate the behavior of the resource in the horizon that the next control action (*i.e.,* setpoint) is expected to be implemented. Hence, λ is equal to twice the control period ($2 \times T_{ctrl}$). In general, this horizon should be short enough to allow the GA to cope with the fastest dynamics in the system, and long enough considering the computing capabilities of the GA and the latency of the communication network.

Long-term fields must be valid for a longer horizon Λ. Thus, we say $\mathcal{F}^n(t)$, measured and computed at time t, is only valid in the range $[t,\ t+\Lambda]$. Λ must be such that the range $[t,\ t+\Lambda]$ includes the next n expected setpoint implementations (assuming no delays). Hence, following a similar line of reasoning as in Figure 3.4, $\Lambda = (n+1) \times T_{ctrl}$. Therefore, $\mathcal{F}^n(t) = \mathcal{F}^n[r]$ can be used in the computation of setpoints in rounds $r' \in [r+1,\ r+n]$, *i.e.,* it has a validity horizon of n rounds.

Long-term fields estimate the behavior of the resource, including the uncertainties, in the larger validity horizon. Additionally, as long-term fields span multiple control rounds, they must account for all the possible setpoint implementations that take place during these rounds. That is, for controllable resources, the state of the resource might change because of a setpoint implementation in one of the control rounds captured by the validity horizon. This is to handle cases when the RA implements a setpoint sent by a GA in round r, but the advertisement issued by the RA is not received by the GA. In such cases, the GA will use the long-term fields of a previous advertisement from this RA, for the next setpoint computation. These fields must account for both the possibilities that the setpoint was, or was not, implemented.

Formally, the following properties must hold for the long-term fields.

Property 4.1 (Long-Term Feasibility)**.**

$$\forall\, r' \in [r, r+n-1],\ \mathcal{F}^n[r] \subseteq \mathcal{F}[r']$$

In other words, the long-term feasibility should be a subset of all short-term feasibility sets that lie within its horizon. This equates to the intersection of all short-term feasibility sets of the next n control rounds. This property ensures that any setpoint in $\mathcal{F}^n[r]$ lies within the feasibility region of the resource for the entire long-term horizon. Therefore, if the GA chooses a setpoint from $\mathcal{F}^n[r]$ for some RA, it is guaranteed to be within the actual feasibility region of the RA, if it is received at a time $t' \leq t + \Lambda$, where t is the timestamp of the advertisement in round r, and $\Lambda = (n+1) \times T_{ctrl}$.

Property 4.2 (Long-Term Uncertainty)**.**

$$\forall\, r' \in [r, r+n-1],\ \forall\, u \in \mathcal{F}^n[r],\ \mathcal{U}[r'](u) \subseteq \mathcal{U}^n[r](u)$$

As the uncertainty function encapsulates the uncertainty of the resource when instructed to implement a setpoint u, the uncertainty set of a setpoint should contain all the short-term uncertainty sets of that setpoint in the long-term horizon. This equates to the union of all short-term uncertainty sets of that setpoint in the next n rounds. This ensures that any actual implementation in the next n rounds lies within the long-term uncertainty set of the issued setpoint. Therefore, if the GA computes a setpoint, in round r, that passes the admissibility test (*i.e.,* maintains robust safety) when considering the uncertainty advertised in $\mathcal{U}^n[r]$, then it maintains robust safety given the actual uncertainty $\mathcal{U}(t')$, for $t' \leq t + \Lambda$, where t is the timestamp of the advertisement in round r, and $\Lambda = (n+1) \times T_{ctrl}$.

Note that the condition in Property 4.2 must hold for all u in $\mathcal{F}^n[r]$. Given that the domain of $\mathcal{U}^n[r]$ is $\mathcal{F}^n[r]$, and the domain of $\mathcal{U}[r']$ is $\mathcal{F}[r']$, then Property 4.1 guarantees that all the elements of $\mathcal{F}^n[r]$ are in the domain of $\mathcal{U}[r']$ as well, for all valid values of r'.

The choice of the long-term horizon (n or Λ) is a trade-off between several factors. A smaller horizon requires less time to compute, provides a more accurate prediction, and exports less uncertainty to the GA. However, as the horizon decreases, the GA becomes less robust to losses, as it is only robust to $n-1$ consecutive losses from a single RA. Sending both fields allows us to take advantage of the accuracy of the short-term prediction, and the robustness of having a longer-term horizon. In general, one can compute and send fields for several long-term horizons in each advertisement, and use the most accurate available one at the GA for computation. We show the case of only one long-term and one short-term horizon, as the general case becomes a simple extension. Note that the value of n could be different for different RAs. For simplicity, we maintain a single value when describing the design.

GA Design

The design of the GA for Robuster remains largely unchanged from Algorithm 4.1, except for two functions. The first is `compute_setpoints` (line 27). This function must

Algorithm 4.4: `choose_advertisements`($\mathcal{A}$, C) function in the GA with Robuster

```
1  A' ← ∅;                                   // set of chosen advertisements
2  L ← {0};                                  // vector of highest label seen for each RA
3  n;                                        // pre-configured value of long-term horizon
4
5  for each (adv, i, ℓ) ∈ A do
6      if ℓ > C − n and ℓ > L[i] then
7          if ℓ = C then
8              adv' ← (adv.F, adv.U, adv.T);
9          else
10             adv' ← (adv.F^n, adv.U^n, adv.T);
11         end
12         Remove advertisement with identifier i from A';
13         A' ← A' ∪ {(adv', i)};
14         L[i] ← ℓ;
15     end
16 end
17
18 return A';
```

perform the admissibility test [139, 140], as mentioned in the previous subsection. We do not go into the details of the test, as it is beyond the scope of this thesis.

The second function is the `choose_advertisements` function, previously described in Algorithm 4.2. The modified function is presented in Algorithm 4.4. In this function, the GA maintains, in addition to the set $\mathcal{A}'$ of chosen advertisements, a vector $\mathbf{L}$ of labels, corresponding to the highest label stored for each RA in $\mathcal{A}'$ (lines 1-2). We consider that the GA is pre-configured with the long-term validity horizon, given as n to this function.

The function iterates through the set $\mathcal{A}$ of all advertisements, only selecting advertisements with labels greater than $C - n$ (line 6), as other advertisements are no longer valid. It also does not consider advertisements from an RA which already has an advertisement in $\mathcal{A}'$ with a higher label (line 6). Thus, for each RA, only the latest valid advertisement, if any, is finally stored in $\mathcal{A}'$.

Among the latest valid advertisements for each RA, if the label of the advertisement is equal to C, *i.e.,* if the advertisement belongs to the immediately previous control round, then the short-term fields of that advertisement are stored in adv' (line 8), as their validity horizon has not passed. Otherwise, the advertisement belongs to an earlier round, but its long-term fields are still valid. In these cases, the long-term fields are stored in adv' (line 10). We note that adv' always contains the timestamp of the advertisement, and any other field that the RA includes in the advertisement, such as the preference. As mentioned earlier, the preference does not affect grid safety, and can be considered of secondary importance when an advertisement loss occurs.

After computing adv', it is inserted into $\mathcal{A}'$, replacing any advertisement from that RA in $\mathcal{A}'$, if any (lines 12, 13). Then, **L** is updated to reflect that the latest label for the RA in question has changed (line 14).

Recall, from Algorithm 4.1 that the returned $\mathcal{A}'$ is passed as a parameter to the `ready_to_compute` function, which in turn decides whether the advertisements present are sufficient for a computation. Robuster makes no changes to that function. Rather, it maximizes the chances that that function returns a $ready$ flag set to true, as it maximizes the number of available advertisements that are valid, hence safe to use.

4.2.3 Formal Guarantees

We say a CPS implements Robuster if its RAs generate short-term and long-term fields for advertisements, satisfying Properties 4.1, 4.2, and its GAs are designed as described in the previous subsection, specifically Algorithm 4.4. We show that a CPS that implements Robuster guarantees robust safety and robust optimality. As Robuster enables the GA to compute setpoints in control rounds in which it could not previously compute, we say Robuster increases robust availability.

Theorem 4.4 (Robust Safety)**.** *A CPS that implements Robuster guarantees robust safety.*

Proof. Given that the GA implementing Robuster performs the admissibility test before issuing a vector of implementation setpoints, it suffices to show that the advertisements used in the computation of that vector are valid in the round of computation.

From Algorithm 4.4 line 6, we see that $\mathcal{A}'$ returned by the `choose_advertisements` function only contains advertisements with label $\ell > C - n$.
From Algorithm 4.1 line 22, in a computation for round r, the parameter C passed to the `choose_advertisements` function is equal to $r - 1$.
Therefore, only advertisements with label $\ell > r - n - 1$ are used.
Also $\ell < r$, as we are computing for round r.
Therefore, advertisements used have a label $\ell \in [r - n,\ r - 1]$.

If $\ell = r - 1$, the short-term fields of this advertisement are used (line 8).
These fields are valid for round r by construction.
Otherwise, when $\ell \in [r - n,\ r - 2]$, long-term fields are used (line 10).
From Properties 4.1, 4.2, long-term fields for an advertisement from round r' are valid to be used in computation for rounds $I(r') = [r' + 1,\ r' + n]$.
$\forall\, \ell \in [r - n,\ r - 2],\ r \in I(\ell)$. □

Robust optimality (Definition 3.6) requires that the vector of computed setpoints be unaffected by Robuster when all advertisements are received. However, Robuster

requires an admissibility test to be performed by the GA. Although such a test is to be performed by a non-robust GA, Robuster increases the set of possibilities by considering setpoint losses. Even when all advertisements are received, Robuster must maintain this consideration. Therefore, in some rounds, the admissibility test might fail due to this additional consideration, thus affecting the vector of computed setpoints. Hence, robust optimality can only be guaranteed when the admissibility test succeeds.

Theorem 4.5 (Robust Optimality)**.** *A CPS that implements Robuster guarantees robust optimality in all rounds in which the admissibility test succeeds.*

Proof. From Algorithm 4.1, the vector $\mathbf{X}$ of computed setpoints in a round r depends on parameters that are the output of the `ready_to_compute` function (lines 24, 27). Robuster only modifies one parameter of the `ready_to_compute` function, namely $\mathcal{A}'$. Therefore, it suffices to show that when all advertisements are received, $\mathcal{A}'$ is left unchanged by Robuster. In that case, the `ready_to_compute` function would return the same values, and the `compute` function would compute the same vector of setpoints, if the admissibility test succeeds.

We compare Algorithm 4.4 with Algorithm 4.2 when all advertisements from round $r-1$ have been received.
If the GA is computing setpoints for round r, then the `choose_advertisements` function is passed $C = r - 1$.
Algorithm 4.2, therefore, returns a set $\mathcal{A}'$ that contains one advertisement from each RA with label C (line 3).
These advertisements only contain short-term fields, as no robustness mechanism is implemented.

Algorithm 4.4 will find an advertisement with label C from each RA (line 7).
It inserts that advertisement into $\mathcal{A}'$, including only its short-term fields (lines 8, 13).
It also removes any advertisement from that RA with a smaller label (line 12).
Therefore, Algorithm 4.4 and Algorithm 4.2 return the same $\mathcal{A}'$. □

4.2.4 Construction of Long-Term Fields

The construction of long-term fields of advertisements depends on the specifics of both the CPS and the resource. Here, we show how these can be constructed for three types of resources in the COMMELEC framework [8]. COMMELEC is described in detail in Chapter 8. For the purpose of this section, it suffices to know that in COMMELEC, the GA sends AC active and reactive power setpoints.

The three types of resources considered here are (1) a fully controllable battery, (2) an uncontrollable PV, and (3) an uncontrollable load. The methods presented for

Algorithm 4.5: Power bounds for the short-term feasibility of a battery

Function: getShortTermBounds($P,\ Q,\ T_{ctrl},\ SoC$)

1: $\lambda = 2 \times T_{ctrl}$
2: Use a converter model to get the corresponding DC power p from the AC setpoint $(P,\ Q)$ implemented at time t
3: Use p and SoC to estimate the SoC^{λ}: the SoC of the battery at $t + \lambda$
4: Use SoC^{λ} to compute the DC power bounds, p^{λ}_{min}, p^{λ}_{max}, that respect the DC voltage and current limits, as in [141]
5: Use the converter model to get P^{λ}_{min}, P^{λ}_{max} from p^{λ}_{min}, p^{λ}_{max}
6: **return** P^{λ}_{min}, P^{λ}_{max}, SoC^{λ}

Algorithm 4.6: Power bounds for the long-term feasibility of a battery

Function: getLongTermBounds($P,\ Q\ , T_{ctrl},\ SoC,\ n,\ S$)

1: $[P^{\lambda}_{min}, P^{\lambda}_{max}, SoC^{\lambda}]$ = getShortTermBounds($P,\ Q,\ T_{ctrl},\ SoC$)
2: $SoC^{\lambda}_{min} = SoC^{\lambda}_{max} = SoC^{\lambda}$
3: **for all** $i \in [1, n-1]$ **do**
4: $\quad [P^{(i+1)\lambda}_{min}, -, SoC^{(i+1)\lambda}_{min}]$ = getShortTermBounds($P^{i\lambda}_{min},\ Q,\ T_{ctrl},\ SoC^{i\lambda}_{min}$)
5: $\quad [-, P^{(i+1)\lambda}_{max}, SoC^{(i+1)\lambda}_{max}]$ = getShortTermBounds($P^{i\lambda}_{max},\ Q,\ T_{ctrl},\ SoC^{i\lambda}_{max}$)
6: **end for**
7: $P^{\Lambda}_{min} = \max_{i\in[1,n]} P^{i\lambda}_{min}$
8: $P^{\Lambda}_{max} = \min_{i\in[1,n]} P^{i\lambda}_{max}$
9: **return** P^{Λ}_{min}, P^{Λ}_{max}

constructing long-term fields builds on the methods for constructing short-term fields for these resources in COMMELEC, as defined in [141].

Controllable Batteries

To compute the short-term fields, the battery RA makes use of the battery model proposed in [142]. The batteries are considered to be ideal, *i.e.,* their uncertainty function $\mathcal{U}(u) = u$, for all $u \in \mathcal{F}$. Thus, the long-term uncertainty is the same as the short-term uncertainty. The feasibility region of the batteries depends on the SoC, and can be summarized with two limit values, providing the minimum and maximum AC active power, P_{min} and P_{max}. Formally, the feasibility and uncertainty are defined as follows:

$$\mathcal{F} = \{(P,Q) \in \mathbb{R}^2 | P_{min} \leq P \leq P_{max}, \sqrt{P^2 + Q^2} \leq S\}, \tag{4.1}$$

$$\mathcal{U}(P,Q) = \{(P,Q)\}, \tag{4.2}$$

where S represents the rated power of the battery converter.

Given the power setpoint $(P,\ Q)$, the control period T_{ctrl}, and the state-of-charge

of the battery SoC, the battery RA can compute P_{min} and P_{max}, for a time horizon $\lambda = 2 \times T_{ctrl}$. It will also estimate the resulting SoC at the end of the time horizon (SoC^{λ}). The pseudo-algorithm of this process is detailed in Algorithm 4.5.

In order to compute the long-term feasibility for a horizon of n control rounds, we follow the procedure described in Algorithm 4.6, in which we call the function in Algorithm 4.5 a total of $2n-1$ times.

First, we compute the short-term power bounds P^{λ}_{min} and P^{λ}_{max}. This gives us the minimum and maximum power setpoints that the battery might implement in the next time horizon $[t, t+\lambda]$, where t is the timestamp of the advertisement being computed. Then, we follow two different tracks of computation. The first assumes that the RA will receive minimum possible power setpoint to implement in the next control round P^{λ}_{min}. We use that as input to the function in Algorithm 4.5, to get the new minimum for the next round. This process is repeated $n-1$ times. The second does the same assuming the maximum power setpoint is received in each round.

After computing the n values of P_{min} — one for each of the following n control rounds — we take the maximum of these n values to be the long-term minimum bound P^{Λ}_{min} (line 7). The same is done for the maximum bound, except the minimum of the n values is taken (line 8). In certain situations, the power bounds are monotonic with time, thereby removing the need to perform this last step. However, we maintain it to satisfy Property 4.1 in the general case.

Uncontrollable PV

For uncontrollable resources, the value of the setpoint issued by the GA is largely irrelevant, as the resource is uncontrollable, and will inject/absorb power based on its internal state. The setpoints acts as a probe, instructing the RA to issue an advertisement to the GA. The feasibility, therefore, is irrelevant. The uncertainty, however, must be advertised, as it affects the admissibility test that the GA performs, thus affecting the value of setpoints to other RAs of controllable resources.

In COMMELEC, the short-term fields of an uncontrollable PV are defined as follows:

$$\mathcal{F} = \{(P_f, Q_f)\}, \tag{4.3}$$

$$\mathcal{U}(P_f, Q_f) = \{(P, Q) \in \mathbb{R}^2 | P_{min} \leq P \leq P_{max}, Q_{min} \leq Q \leq Q_{max}\}, \tag{4.4}$$

where (P_f, Q_f) are the forecasted power injection in the next control round. Essentially, the feasibility region attempts to inform the GA about the implementation in the next control round, and the uncertainty provides a two-dimensional range around that point.

The method for computing the short-term uncertainty limits is presented in [143]. It relies on a forecasting method for solar irradiance that uses past measurements, sampled at the desired horizon, as training data. As the method is parameterizable with respect to the desired horizon, the RA can run two instances of the method, thereby computing both the short-term and long-term uncertainties.

In practice, the long-term PV dynamics are larger than the short-term ones. However, to ensure Property 4.2 is satisfied in all cases, the long-term uncertainty should be set to be the union of itself and the short-term uncertainty.

Uncontrollable Load

The case of the uncontrollable load is very similar to that of the PV. Being an uncontrollable resource, its advertisement is defined as in Equations 4.3, 4.4, with (P_f, Q_f) representing the current power absorption instead.

However, the forecasting method for the uncertainty limits is different, as consumption profiles cannot be predicted using the same approach in [143]. Instead, the persistence method is used, whereby the uncertainty limit is given as follows:

$$\begin{aligned} P_{min}(t+\lambda) &= (1-\alpha_P^\lambda)\hat{P}(t), \\ P_{max}(t+\lambda) &= (1+\alpha_P^\lambda)\hat{P}(t), \\ Q_{min}(t+\lambda) &= (1-\alpha_Q^\lambda)\hat{Q}(t), \\ Q_{max}(t+\lambda) &= (1+\alpha_Q^\lambda)\hat{Q}(t), \end{aligned} \tag{4.5}$$

where $(\hat{P}(t), \hat{Q}(t))$ is the measured power at time t, and $\alpha_P^\lambda,\ \alpha_Q^\lambda \in (0,1]$ are the forecasting parameters, obtained *a priori* using a model of the load.

The long-term uncertainty limits can be computed in a similar way, by obtaining α_P^Λ and α_Q^Λ. In order to guarantee Property 4.2, it must hold that $\alpha_P^\Lambda \geq \alpha_P^\lambda$ and $\alpha_Q^\Lambda \geq \alpha_Q^\lambda$.

4.3 Experimental Comparison & Validation

We implemented intentionality clocks and Robuster for the COMMELEC framework [8]. In this section, we experimentally validate our methods and compare them to alternative techniques for achieving robustness. We perform our experiments using T-RECS (refer to Chapter 7 for its design), a virtual commissioning tool that simulates the CIGRÉ benchmark low-voltage microgrid [44] consisting of a battery, a PV plant, and a load. T-RECS enables us to emulate non-ideal network conditions and study the behavior of the actual COMMELEC implementations under such conditions. Our methods are then deployed on a real-scale microgrid for a 24-hour validation.

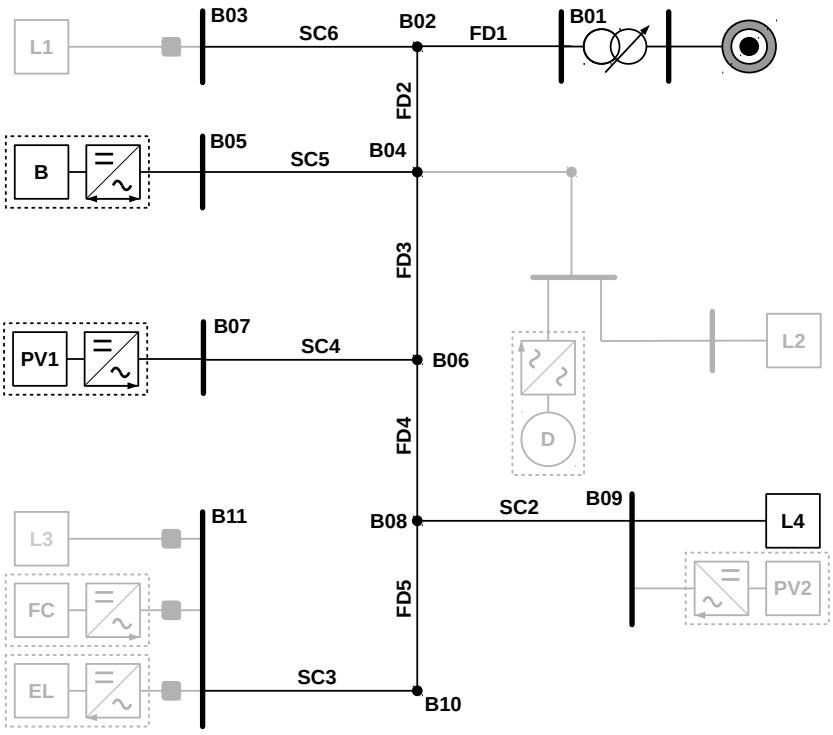


Figure 4.4 – CIGRÉ low-voltage benchmark microgrid. The resources not used for our experiments are greyed-out

In our experiments, the GA is instructed to provide frequency support [19], or to track a dispatch plan signal [18] at the point of common coupling (PCC). This is in addition to maintaining grid safety and yielding to the preference of the resources.

4.3.1 Experimental Setup

We test our method on the CIGRÉ benchmark low-voltage microgrid [44], shown in Figure 4.4. The microgrid is connected to the main grid, and consists of a $25\ kW$ uncontrollable PV, a $30\ kW$ / $90\ kWh$ battery, and a $5\ kW$ uncontrollable load. As mentioned earlier, the GA is instructed to track a pre-determined power profile, and, in some cases, to additionally provide frequency support to the main grid.

We modify the code of the COMMELEC RAs to include the long-term fields in the advertisements. We also modify the GA code, implementing intentionality clocks and Robuster, as in Algorithms 4.1, 4.4. Finally, we implement and deploy a master at each RA, as described in Algorithm 4.3.

In our setup, we compare four different alternatives of the COMMELEC GA, all of which implement intentionality clocks. (1) The *Normal* GA, which is the original implementation that requires the most recent advertisement from each RA. Otherwise,

it sends probes after $5 \times T_{ctrl}$. (2) The *Robust* GA, which implements Robuster. This GA replaces any missing short-term advertisement with valid long-term fields from that resource, if available. (3) The *Only-long* GA, which is a variation of the Robust GA, in that it only uses long-term fields throughout its operation (*i.e.,* just by replacing λ by Λ in the *Normal* GA). This decreases the size of an advertisement and simplifies the design of the GA. However, it violates robust optimality. (4) The *Previous-short* GA, which replaces any missing advertisement with the latest previously received advertisement from that RA. This eliminates the need to construct, send, or handle long-term fields. However, it violates robust safety.

In COMMELEC, the control round occurs roughly every $100\ ms$. In our experiments, we consider $T_{ctrl} = 100\ ms$, and $n = 10$. This gives us a validity horizon of 10 control rounds. From these, we get $\lambda = 200\ ms$ and $\Lambda = 1100\ ms$.

T-RECS enables us to use the actual GA and RA code and a simulated version of the resources and the grid. The messages are exchanged over an emulated communication network, the topology of which consists of one router, with each software agent (GA and RAs) on a different subnet. Resources are run on the same host machine as the RAs. With T-RECS, we are able to vary the link loss rate, and we analyze different values between 0% and 20%.

We use the root mean square error (RMSE) as a metric to measure the performance of the different GA implementations. The RMSE is calculated between the measured power at the slack and the requested frequency support signal (or dispatch plan signal). This shows how well each implementation can track the signal, and how robust each is to message losses.

4.3.2 Results

In this section, we illustrate the performance of our mechanisms under different conditions, and compare the results for the four different GAs described in Section 4.3.1. Several scenarios are considered in order to highlight the conditions under which each GA alternative performs well.

Frequency Support with Non-Binding Grid Constraints

We first study the performance of the methods when the grid state is far from the operational limits in terms of bus voltages and line currents. The GA is instructed to provide frequency support to the main grid, based on the frequency signal of Figure 4.5, which represents quick dynamics. As we are interested in studying the effects of the losses in the network, we vary the link loss rate between the GA and the RAs in the range [0%, 20%].

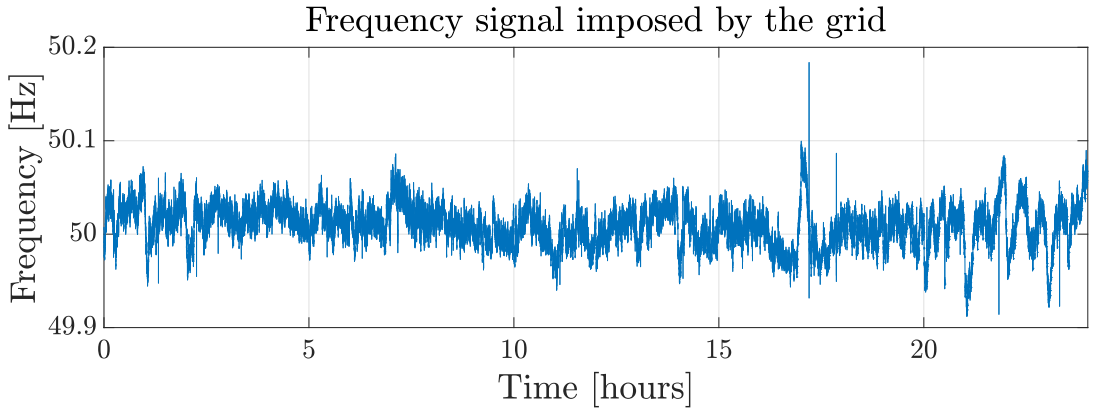


Figure 4.5 – Frequency signal imposed by the main grid used to provide frequency support

Method / Loss rate	0%	5%	10%	15%	20%
Normal	121.65	198.95	317.48	541.52	1442.98
Robust	120.44	129.03	147.25	150.55	188.12
Only-long	121.65	130.36	153.84	154.99	178.85
Previous-short	121.65	122.18	139.40	154.77	451.17

Table 4.1 – Root mean square error (in Watts) between the real power at the slack bus and the requested tracking signal, for a 10-minute interval

Table 4.1 and Figure 4.6 show the resulting RMSE for the different methods across the different link loss rates, for an interval of 10 minutes. The RMSE is calculated between the actual power at the slack, and the result of $S = -\sigma(f - f_0)$, where S is the expected power at the PCC when providing frequency support, computed by multiplying the droop parameter σ with the divergence of the grid frequency f from the reference frequency $f_0 = 50$ Hz.[2]

We observe that the performance of the *Normal* GA rapidly deteriorates as the link loss rate increases. This follows directly from the fact that it is extremely sensitive to the amount of available information, and fails to follow the request in our quick dynamic scenario, as it suffers from decreased availability. The *Previous-short* GA maintains a good level of tracking until the loss rate is too high. This is expected as the information it uses in case of a loss (the previous advertisements) is invalid, and as the loss rate increases, tracking the quick frequency changes becomes increasingly unlikely. In other words, violating robust safety not only makes it susceptible to violating grid safety, it also affects performance.

The *Robust* and *Only-long* GA manage to provide frequency support even under 20% link loss rate, although the *Only-long* GA obtains slightly worse performance,

[2]We take $\sigma = 100$ kW/Hz in our experiments.

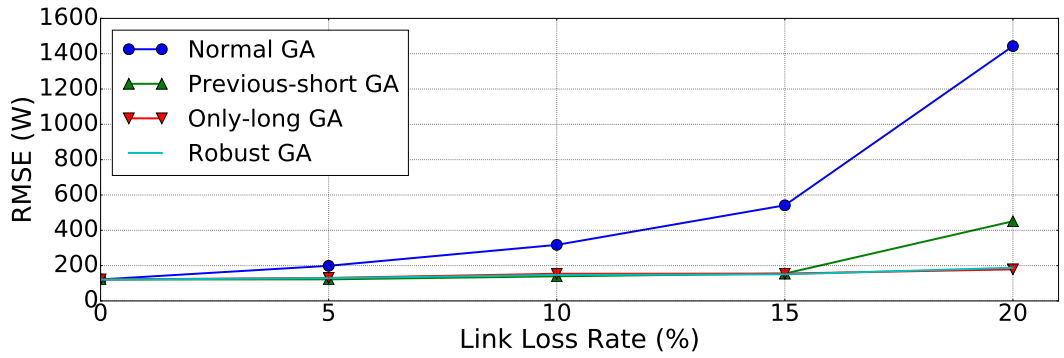


Figure 4.6 – Root mean square error (in Watts) between the real power at the slack bus and the requested tracking signal, for a 10-minute interval

especially for lower loss rates. This stems from the fact that its computations are always conservative, as they all use advertisements with a long-term horizon Λ. The effects of this are not drastic in such a scenario, but will appear when the grid conditions are binding, as presented in the next section.

Tracking a Dispatch Plan with Binding Grid Constraints

In order to study the behavior of the *Robust* and *Only-long* GA under binding grid conditions, we consider a scenario in which the GA is instructed to follow a pre-computed dispatch plan. The slower dynamics in this experiment allow us to better visualize the tracking performance. Moreover, we artificially limit the ampacity of the line connecting the microgrid to the main grid (FD1) to $16\ A$, i.e. a power limit of c.a. $11\ kVA$. Note that, even though the dispatcher is allowed to request an active power close to $11\ kW$, the GA, due to the combined uncertainty of the PV and the load, might prefer not to track the plan to avoid limit violations. This conservative behavior is accentuated when the uncertainty is larger, as when using the long-term advertisements.

Figures 4.7 and 4.8 show the tracking results of *Only-long* GA and *Robust* GA, respectively, when the link loss rate is 2%. We observe that, although both manage to track the $10\ kW$ request fully (as it results in a current far away from the ampacity limit of line FD1). However, only the *Robust* GA manages to track the $11\ kW$ signal. The *Only-long* GA uses advertisements with larger uncertainty, and is thus conservative in order to avoid current violations. The *Robust* GA maintains tracking as it can safely do so without risking violation, due to the accuracy of the short-term advertisements it uses. The RMSE values are also indicative, with the *Robust* GA having $191.54\ W$, and the *Only-long* GA having $320.91\ W$. This highlights the importance of the robust optimality property.

The *Previous-short* GA is not conservative, and thus maintains tracking (under low

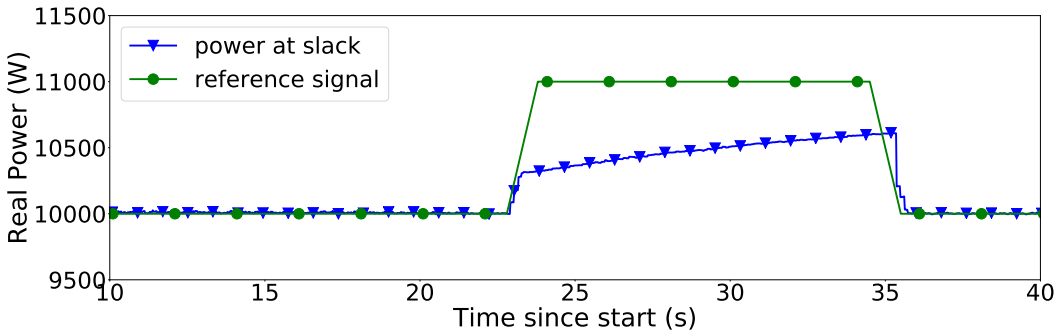


Figure 4.7 – Tracking experiment of *Only-long* GA with binding grid conditions and a 2% loss rate

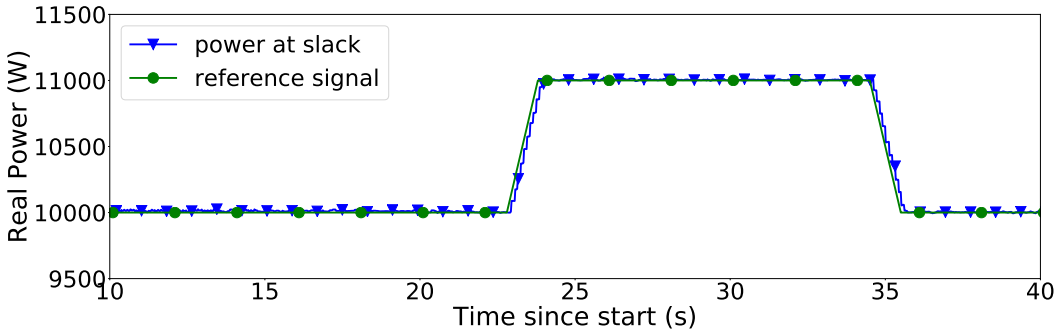


Figure 4.8 – Tracking experiment of *Robust* GA with binding grid conditions and a 2% loss rate

loss rates) even in binding grid conditions. However, as it uses invalid information, it might cause voltage and/or current violations, as mentioned earlier.

Experimental Validation in a Real-Scale Microgrid

As mentioned earlier, the experiments in previous subsections were performed in T-RECS. Here, we experimentally validate our mechanisms through a deployment in a real-scale microgrid on-campus. The microgrid has the same specifications as the one used in earlier experiments and presented in Figure 4.4.

We validate the *Robust* method via a 24-hour frequency support experiment with a 2% link loss rate. The 24-hour experiment, with the profiles of the PV and the load taken from an actual experimental run, enables us to see the performance under different and realistic grid conditions. In this particular case, the battery power is used by the GA as the slack variable, compensating for the PV and load power-variations and adapting to the frequency signal, in order to provide frequency support. The initial state of charge of the battery (20%) is pre-defined by the forecasted PV and load profiles on the day before.

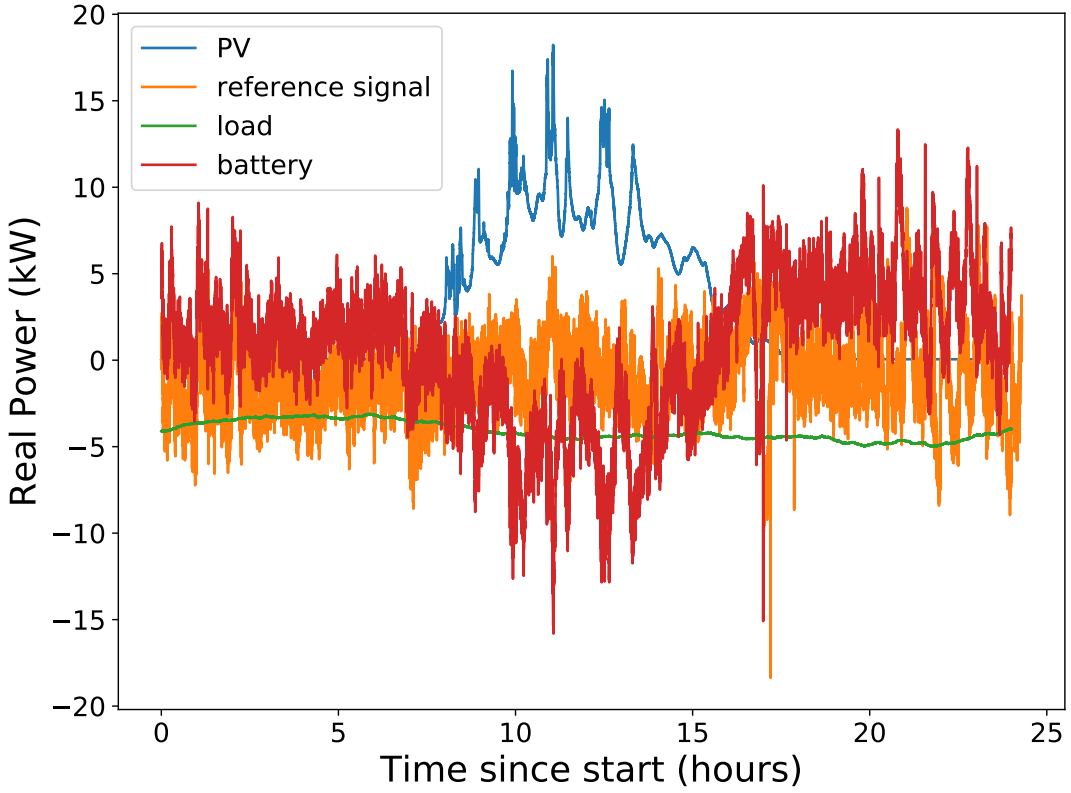


Figure 4.9 – *Robust* GA 24-hour frequency support experiment with a 2% link loss rate. The power at the slack is not shown as it would be hidden by the reference signal

Figure 4.9 shows the results of the tracking, in addition to the power at the buses of the battery, the PV, and the load. The measured power at the slack would not be visible if shown, as it would be hidden by the reference signal. Instead we compute the RMSE, which turns out to be 145.67 W for the entire day. We also measure the RMSE over a rolling window of 20 minutes, and the resulting average and maximum RMSE are 142.40 W and 263.53 W, respectively. This shows the robustness of our method throughout the daily cycle. Furthermore, the SoC of the battery during the experiment is shown in Figure 4.10, showing the capabilities of the battery to provide such an ancillary service to the main grid.

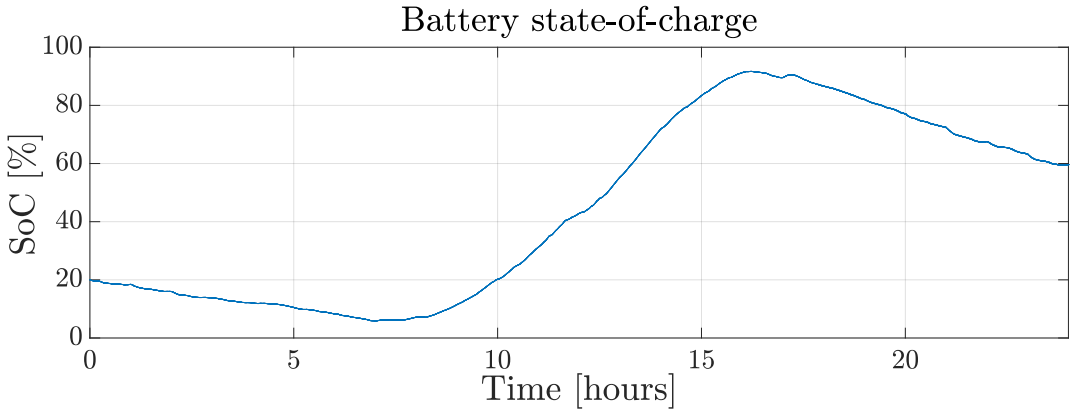


Figure 4.10 – Battery state-of-charge (SoC) during the 24-hour experiment

4.4 Conclusion

In this chapter, we have presented two mechanisms, intentionality clocks and Robuster, that together enable the design of a robust GA.

We have highlighted the need for robust and reliable ordering in a CPS for real-time control of electric grids, and have shown how state-of-the-art ordering mechanisms fail to support the round-based communication scheme between the GA and the RAs. Then, we have discussed the design of intentionality clocks, including its integration with the GA and RA models presented in Chapter 3. Intentionality clocks was proven to satisfy robust and reliable ordering, and to preserve robust and reliable availability.

Using intentionality clocks, we were able to design Robuster, a mechanism for generating advertisements and computing setpoints that enables the GA to maintain grid safety in the presence of non-idealities in the communication network, RAs, or resources. The RAs — in a CPS implementing Robuster — generate and issue advertisements that contain fields with a long-term validity horizon, in addition to the short-term fields previously included. The GA then uses the short-term fields when an advertisement from the previous round is available, and the long-term fields from earlier advertisements otherwise. The long-term validity of the new fields is guaranteed by their generation at the RAs.

We have presented the design of Robuster, highlighting the required properties on the long-term fields of an advertisement, and showing the minimal changes required at the GA in order to implement it. Then, a formal proof has been presented that Robuster guarantees robust safety and optimality. Robuster increases robust availability, as it enables the GA to compute setpoints in more control rounds. We have discussed several examples of how the long-term fields can be generated in different kinds of resources in the COMMELEC framework.

Finally, we have experimentally validated our methods, via a deployment both in a

real-scale microgrid and in T-RECS, a virtual commissioning tool. Our comparative analysis has shown that by maintaining robust safety and optimality, Robuster outperforms state-of-the-art alternatives under various binding and non-binding grid conditions, and under varying rates of losses in the communication network. A full-day validation has shown that Robuster maintains its performance in a long-running experiment with realistic conditions.

This chapter has dealt with the operation of a GA when controlling a grid that is connected to an upper-level main grid at a PCC. However, the GA might be requested to, or even decide to, disconnect the two grids. Such a condition is referred to as islanding. Islanding introduces several challenges that must be addressed by the GA. In Chapter 8, we go over these challenges, and discuss the additional mechanisms necessary to design a robust GA that is capable of performing islanding maneuvers.

5 Axo: Tolerating Delay Faults in Cyber-Physical Systems

You may delay, but time will not.
— Benjamin Franklin

In previous chapters, we have discussed mechanisms for controller robustness, which is concerned with handling non-idealities in the various CPS components, except the controller. In this chapter, we discuss controller reliability, which is concerned with handling controller faults. Specifically, we focus on delay faults incurred when the GA is computing a vector of setpoints. Such faults might result in the setpoints reaching the RAs at a time later than the validity horizon of the advertisements used in the computation. This raises two issues, one being the loss of reliable availability due to the absence of a valid setpoint, and the other being the violation of reliable validity due to the presence of an invalid setpoint.

We introduce Axo, a framework for tolerating delay and crash faults in CPS controllers. Axo uses active replication of the controller to handle the former problem of missing valid setpoints. In active replication, if one of the replicas experiences a delay fault, the RAs might still receive valid setpoints computed and issued by the other replicas. Axo also extends the masker component, presented in Chapter 4, to discard invalid setpoints before they reach the RA. Thus, Axo guarantees reliable validity and preserves reliable availability (Definitions 3.8, 3.9).

Over long periods of CPS deployment, controller replicas might crash or experience an increased probability of incurring delay faults. Moreover, multiple replicas might become faulty, rendering the CPS uncontrollable. Axo also provides a mechanism to detect and recover faulty replicas, ensuring the long-term reliability of the CPS.

We provide an overview of Axo, and of the problems addressed in this chapter in Section 5.1. In Section 5.2, we survey the related work on tolerating delay and crash faults. In Section 5.3, we define the fault model and the required properties

on a CPS for Axo. We present the design of Axo in Section 5.4, and prove its formal guarantees in Section 5.5. Then, we analyze the performance of Axo and derive bounds on the detection and recovery time in Section 5.6. In Section 5.7, we highlight the generic nature of Axo, by applying it to an inverted pendulum system, and we show the improved stability metrics it provides. We conclude this chapter in Section 5.8.

5.1 Overview

5.1.1 Problem Description

We consider CPSs as described in Chapter 3, and as shown in Figure 1.2, consisting of a controller, RAs, and sensors. The model and architecture applies to a wide range of CPSs. In addition to real-time control of electric grids [8, 9, 144], this includes manufacturing processes [145], autonomous vehicles [146, 147], and robotics [4, 5].

As mentioned in Chapter 1, such CPSs are mission-critical: their failure can lead to serious damage [12, 148]. 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 CPS controllers have a large code base, they often use third-party libraries, including COTS software. Such CPSs are susceptible to faults incurred by their hardware and software components [32].

Recall that setpoints are computed and issued by the controller of a CPS in order to drive the state of the system in a 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 is captured by the validity horizon of the advertisements used in the computation of these seetpoints. Each RA computes its advertisements such that they are valid for a certain time, which is based on the inertia of the underlying grid or process. Setpoints received beyond the validity horizon must not be implemented. This motivates the need for reliable validity (Definition 3.8).

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 an extreme case of delay faults, where the delay is infinite.

Delay faults not only might violate reliable validity, they might also result in the loss of reliable availability (Definition 3.9). Reliable availability states that all RAs must receive and implement setpoints in a given control round. Therefore, we must handle both the problem of receiving delayed setpoints and the problem of not receiving setpoints at all.

We designed Axo, a fault-tolerance protocol that targets delay faults in real-time CPSs. Axo is transparent to the CPS, i.e., it imposes no interference with the CPS functionality, and it can be used with minor additions to the controller software, as elaborated in Section 5.4. It is applicable to a wide range of CPSs that exhibit the properties described in Section 5.3.

Axo uses active replication of the controller, whereby multiple replicas of the controller simultaneously receive advertisements and measurements, compute setpoints, and issue the computed setpoints to the RAs. The presence of multiple replicas increases the chances that setpoints are received by RAs. That is, if one of the replicas experiences a delay, the other replicas are simultaneously computing and issuing setpoints, essentially masking this fault from the RAs.

However, to truly mask the fault, RAs must not receive the setpoint issued by the delayed replica, if it is delayed beyond the validity horizon. Axo uses a component at each RA, namely the *masker* (proposed in Chapter 4), to discard delayed setpoints. This enables Axo to guarantee reliable validity, and to preserve reliable availability.

5.1.2 Challenges in Delay Fault Detection & Recovery

Axo guarantees reliable validity despite any number of faults affecting the controller or other CPS components. However, in order to preserve reliable availability, and maintain real-time control, it must ensure the reception of setpoints at the RAs. Consequently, it must ensure the existence of at least one non-faulty replica at all times.

Although transient in nature, delay faults can occur more frequently over long periods of CPS deployment, if not detected and recovered. In such cases, the CPS might end up with no non-faulty replicas, and availability will be lost. Hence, to continuously mask delay faults, faulty replicas need to be detected and recovered.

Delay faults are an end-to-end phenomena, the two ends being the controller and the RAs. Existing fault-detection mechanisms (see Section 5.2) rely on polling the state of the controller hence cannot be used for detecting delay faults.

To detect delay faults in the controller replicas, Axo introduces feedback, from the RAs, about the validity of the setpoints received. If an RA receives a valid setpoint, then the corresponding controller is deemed to not have a delay fault, and *vice versa*. The main challenge in developing such a design is the possibility of the messages being lost, reordered, retransmitted, or delayed. This can be handled with a partial order provided by temporal ordering via timestamps, or with a total order provided by intentionality clocks. Also, after a single setpoint computation, a controller with multiple RAs will receive multiple feedbacks, one from each RA. 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, and then proceed with non-delayed operation. 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, and losing availability in the meantime. Second, as CPS communication networks are non-ideal, they 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 CPS, the rate at which a CPS controller issues setpoints is not known. For real-time control of electric grids, we can rely on the pseudo-periodic operation of the GA. However, in general, conventional techniques for detecting crash faults, such as monitoring the frequency at which setpoints are issued, are ineffective.

The detection and recovery algorithms provided by Axo are designed to be soft state [149], i.e., their state can be reconstructed from received messages after a reboot. This enables the seamless addition and removal of replicas, as discussed in Section 5.4.

Note that throughout this chapter, we discuss controllers and controller replicas. The replication involves the entire controller, with both its components: the SE and the GA. Recall that the SE performs an intermediate computation, then sends the state of the grid to the GA. The GA, therefore, is the only component issuing messages (setpoints) to the RAs, and any faults affecting the computation of the SE also affects that of the GA. The masking, detection, and recovery algorithms, presented in this chapter, tolerate delay and crash faults that occur in either component.

5.2 Related Work

To the best of our knowledge, Axo is the first fault-tolerance protocol that addresses delay faults in real-time CPSs composed of COTS components.

5.2.1 Masking Delay Faults

In the literature, delay faults for real-time systems have been studied under the name of *timing faults* [87, 88, 92, 150]. The closest existing technique is the work done by Veríssimo and Casimiro on the timely computing base (TCB) [87]. They propose an architecture and programming model that can be used to provide generic delay-fault tolerance for real-time systems [88]. As mentioned in Chapter 2, 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 CPSs 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.

Furthermore, several components of the TCB architecture require an implementation specific to each CPS. This drawback is shared with other generic architectures such as the time-triggered architecture (TTA) [92]. In contrast, Axo is a layer of software that can be used on any CPS that satisfies the system model (see Section 5.3) and requires only minor additions to the CPS controller software (see Section 5.4). This enables the deployment of Axo on existing CPSs.

Other work in this field has focused on improving the quality-of-service and response time of the systems [150]. However, that work focuses on traditional transaction-based systems, which have different requirements and pose different challenges than those associated with CPSs. Specifically, the method presented does not aim at providing hard real-time guarantees, such as the reliable validity property that Axo satisfies.

Traditional Byzantine-fault tolerance (BFT) protocols [84, 151] do not generally consider the timing attributes of the setpoints. Moreover, as mentioned in Chapter 2, all BFT protocols require consensus among the replicas. The consensus can take unbounded time [98], delaying delivery of setpoints to RAs indefinitely. This property makes them unsuitable for tolerating delay faults, even in the cases when they are designed for real-time applications [151].

Active replication protocols, such as [123], use multiple controller, all of which simultaneously compute setpoints. The use of multiple replicas incurs zero delay in issuing the setpoints, which makes it attractive for CPSs, especially in the context of delay faults. Although active replication preserves reliable availability, when one replica is delay faulty, it can still send a setpoint at some time after its validity horizon, thereby violating reliable validity. Axo uses active replication with an added mechanism to discard invalid setpoints before they reach the RA, in order to provide reliable validity in the presence of delay faults.

5.2.2 Detection & Recovery of Delay Faults

In order to detect timing faults, the aforementioned TCB and TTA frameworks also require additional information from the CPS. Timing-failure detection under the TTA framework requires *a priori* knowledge of intended send and receive instants of messages [92]. Similarly, detection under the TCB framework requires a known bound on the computation times of the time-critical functionalities of the CPS [91]. This requires static analysis of generally complex functions that might include COTS software. Additionally, as we have seen in Chapter 1, the execution time of real-time CPSs is generally variable and unpredictable. This would require further dynamic analysis of

the execution time, a task that does not fit within the real-time constraints of CPSs. In contrast, Axo provides delay-fault detection, and does not require such information.

Existing mechanisms for fault detection rely on monitoring the replica, such as using heartbeats [124], or on probing the replica for its current state so as to detect inconsistencies [123]. 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. Axo makes use of the round-based communication scheme between GAs and RAs 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 [93, 94]. The trained models are then used to classify a replica as faulty during run-time. Such methods are prone to modeling errors and are limited to CPSs that have constant workloads, making them unsuitable for generic CPSs where the workload of the controller is not known *a priori*.

Fault recovery is a challenging problem in the presence of message losses, delays, and reordering. These non-idealities pose a challenge to the main requirement of fault recovery: ensuring that each replica is recovered only once after detection. This type of problem can be solved using consensus. However, as mentioned in Chapter 2, consensus does not guarantee termination under these conditions [98]. Furthermore, the delays brought about by the multiple rounds of message exchange in state-of-the-art consensus protocols [152] are not suitable for CPSs. To avoid using consensus, Axo makes use of the total order provided by intentionality clocks (Chapter 4), enabling it to recover a faulty replica with minimal delay, at most once after detection.

5.3 System Model

5.3.1 Required CPS Properties

As mentioned in Section 5.1, Axo applies to a wide range of CPSs [8, 9, 144, 145, 146, 147]. However, it requires that the CPS exhibit the following properties that are satisfied by the model presented in Chapter 3:

Property 5.1. *Advertisements have a known validity horizon (T_{val}), measured in time.*

Property 5.2. *RAs are able to handle duplicate setpoints.*

The validity horizon, mentioned in Property 5.1, is specific to each CPS, and it depends on the inertia of the underlying process. In Chapter 3, we have discussed the validity horizon for advertisements in CPSs for real-time control of electric grids.

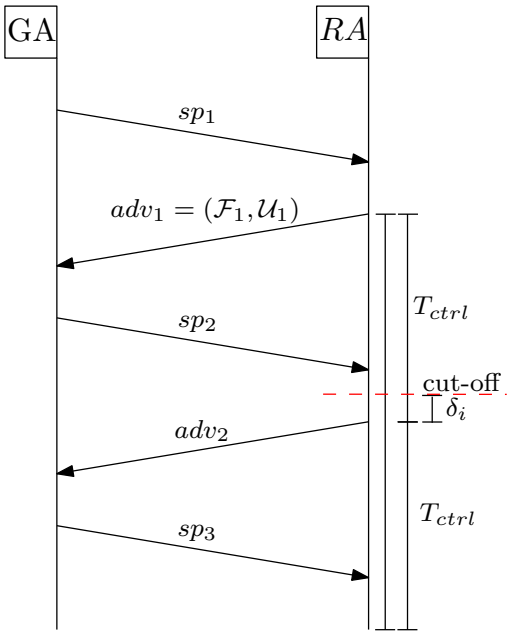


Figure 5.1 – Message sequence chart showing the cut-off for accepting setpoints as a function of the control period T_{ctrl} and the upper bound on setpoint implementation time δ_i

Additionally, in Chapter 4, we have shown how to create advertisements with two separate sets of fields, each with its own validity horizon. Here, we give the GA access to the validity horizon for each advertisement used in computation. Note that this might be different for different RAs.

Property 5.2 requires RAs to handle duplicate setpoints. This property is provided by intentionality clocks, presented in Chapter 4, as only one setpoint issued a given control round is forwarded by the masker to the RA. Alternatively, this property is inherent in CPSs that use absolute, rather than differential, setpoints. Absolute setpoints are 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 a GA that instructs a battery RA 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*'.

The computation model of the CPS is assumed to be the same as the one presented in Chapter 3. Specifically, the GA receives advertisements from RAs, uses at most one advertisement from each RA to perform setpoint computations, and issues at most one setpoint to each RA. The RAs implement the received setpoints, then issue advertisements to the GA.

5.3.2 The Validity Horizon

Recall from Chapter 3, Figure 3.4, that the validity horizon of an advertisement (with short-term fields) is $T_{val} = 2 \times T_{ctrl}$. That is, an advertisement in round r must be valid until the setpoint from round $r + 2$ is received, as shown in Figure 5.1. Although the setpoints that use this advertisement in its computation can only do so in round $r + 1$, the advertisement must be valid for an additional round, to allow the controller enough time to compute a new setpoint, ensuring that the uncertainty advertised still holds.

Therefore, there exists an inherent cut-off time, after which a setpoint in round $r+1$ must not be implemented, as shown in Figure 5.1. The time left after that setpoint finishes implementation must be enough for another control round to take place (*i.e.,* T_{ctrl}). Additionally, recall from Chapter 3 that the upper-bound on the implementation time of a setpoint by an RA is given by δ_i. Therefore, we define the validity horizon of a setpoint (τ) as follows:

$$\tau = T_{val} - T_{ctrl} - \delta_i \tag{5.1}$$

That is, the setpoint must be implemented before $T+\tau$, where T is the timestamp of the advertisement, from the same RA, that was used in the computation of the setpoint. The difference between T_{val} and τ, is to allow enough time for the implementation to take place (δ_i), and for the next setpoint to be received (T_{ctrl}). Implementing the setpoint after this time might result in the RA exceeding the validity horizon of the advertisement, before receiving a new setpoint.

This enables us to formally define the notion of valid and invalid setpoints, previously mentioned in Chapter 3.

Definition 5.1 (Valid Setpoint)**.** *Consider a setpoint sp, issued to an RA with identifier i, which used, in its computation, an advertisement adv from RA i. Let T be the timestamp of adv, and τ be the validity horizon of sp. sp is said to be* valid*, if and only if, the time of reception (t_r) at the RA is such that $t_r \leq T + \tau$. Else, it is said to be* invalid.

As each setpoint is issued to one RA, each setpoint has a well-defined time of reception. Setpoints that are never received, due to network losses or software agent omissions, are not labeled as either valid or invalid, as they are only labeled at the RA as such. Note that, as the controller issues multiple setpoints during a single computation — one to each RA — some of these setpoints might be valid, others invalid, and others not received at all.

The above definition of the validity horizon of a setpoint can be extended for advertisements that are valid for n rounds. Recall from Chapter 4, that an advertisement constructed by the RA to be valid for n rounds has a validity horizon of $T_{val} = (n+1) \times T_{ctrl}$. From Equation 5.1, we say the validity horizon of a setpoint computed using such an

advertisement is $\tau = T_{val} - T_{ctrl} - \delta_i = n \times T_{ctrl} - \delta_i$. In what follows, we denote to the validity horizon of such setpoints as τ^n.

5.3.3 Fault Model

The fault model considered for Axo is also the same as the one presented in Chapter 3. We consider that the software agents are susceptible to crash and delay faults, and that the network can 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.

We define a delay-faulty and a crash-faulty controller as follows.

Definition 5.2 (Delay Faulty Controller)**.** *Consider a setpoint-issuing instant T_1 at a controller. If all the received setpoints issued at T_1 are invalid, then the controller is said to be delay faulty in $[T_1,\ T_2)$, where T_2 is the next issuing instant at this controller.*

A delay-faulty controller is, therefore, one which has delivered invalid setpoints to all its RAs, excluding the setpoints that were lost. It remains classified as delay faulty until it issues setpoints, at least one of which is valid. In case the controller never issues setpoints after T_1, it is classified as crash faulty, as seen in the following definition.

Definition 5.3 (Crash Faulty Controller)**.** *A controller is said to be crash faulty at a time t, if and only if, $\forall T \geq t$, it does not issue setpoints at time T.*

Although crash faults can be considered as a special case of delay faults, in which the computation delay is infinite, here we consider separate the two cases as it allows us to detect them in different methods. Definition 5.2, on its own, does not suffice. A controller that issues valid setpoints at T_1, then fails to issue subsequent setpoints, is not classified as delay faulty. In contrast, a controller that is classified as delay faulty at T_1, then fails to issue subsequent setpoints, is also classified as crash faulty.

5.4 Axo Design

Axo uses active replication of the controller and requires minimal replication overhead: $g + 1$ replicas can tolerate g delay and crash faults. With active replication, all controller replicas receive measurements from sensors and advertisements from RAs, compute setpoints, and issue these setpoints to the RAs. Thus, without additional mechanisms, the replication is exposed to the RAs, which end up receiving setpoints from multiple replicas. Recall that we consider that the RAs are capable of handling such duplicate setpoints (Property 5.2).

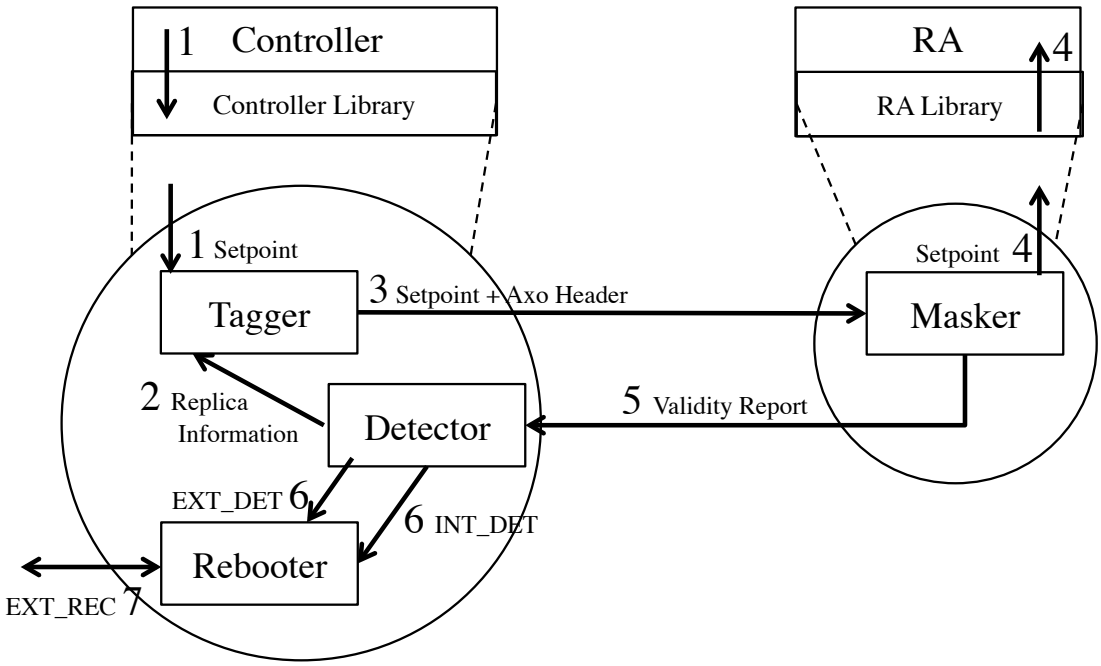


Figure 5.2 – Axo architecture and library components

We present Axo as an extension to the algorithms presented in Chapter 4, which included the two robustness mechanisms, intentionality clocks and Robuster. Recall that the intentionality clocks design (Section 4.1.4) includes a *masker* component at each RA, which serves, among other things, to mask the replication from the RA. The presence of this component makes the CPS satisfy Property 5.2, by ensuring that the RAs at most one instance of each setpoint.

However, the first setpoint received in each round might still be delayed, thereby violating reliable validity. In order to guarantee reliable validity, Axo extends the design of the masker with a mechanism to discard invalid setpoints. Thus, the masker only forwards valid setpoints to the RA.

Axo requires minor modifications to the designs of the RA and the GA, as presented in Section 5.4.1. The RA augments the timestamped advertisements it issues with the corresponding validity horizon τ. The GA can the compute the latest time at which each setpoint is valid, by adding the validity horizon to the value of the advertisement timestamp T. In addition to the vector of setpoints, the GA also issues a vector containing the latest times at which each setpoint is valid, henceforth referred to as a vector of validity times.

Axo uses a *controller library* on each controller replica and an *RA library* on each RA, as shown in Figure 5.2. The controller library is composed of three modules: (1) the *tagger* for intercepting setpoints issued by the GA and tagging them with the latest validity time, (2) the *detector* for detecting faulty controller replicas, and (3) the *rebooter*

for recovering the replicas that were detected as faulty. The RA library has the *masker* that discards invalid setpoints. The detailed design of the tagger and the masker is presented in Section 5.4.2. The operation of the detector and the rebooter are discussed in Section 5.4.3 and Section 5.4.4, respectively.

Each controller replica is assigned a unique replica identifier that serves as its identifier for all ensuing Axo-related communication. When a controller replica issues a vector of setpoints and a vector of validity times, these are intercepted by the tagger on this same replica. The tagger forms a 28-byte Axo header and prepends to each setpoint, before forwarding it to the masker of its original RA destination, according to Algorithm 5.3. The utility of the different fields in the header is explained in Section 5.4.2.

On receiving the setpoint, the masker checks the validity of the setpoint, by comparing the time in the Axo header to the time of reception, according to Algorithm 5.4. The masker forwards the setpoint to the RA if it is valid and discards it otherwise. Then, the masker then sends a validity report to the detectors of all replicas.

As explained in Section 5.1, delay faults cannot be detected using conventional techniques such as time-out or heartbeat between the replicas. Furthermore, feedback from the RA 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 the RA side (specifically, the masker). To this end, the detectors use validity reports to check if the replica is crash or delay faulty, as in Algorithm 5.5. 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 5.7. 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 RAs (by adding an Axo header), which requires the RA library to be present on all RAs. This thin layer is transparent to the RA. 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 minor modifications to the controller and RAs. Consequently, it remains agnostic to any changes, updates, or patches performed to these software agents.

5.4.1 Controller & Resource Agent Modifications

We consider that the RA augments each advertisement with the setpoint validity horizon. An RA implementing Robuster (see Chapter 4), must include two setpoint validity horizons, one for the short-term fields (τ), and one for the long-term fields of the advertisement (τ^n). Recall that the advertisement also contains a timestamp T, corresponding to the time of its generation.

Algorithm 5.1 and Algorithm 5.2 describe the modifications to the GA required for Axo. These extends Algorithm 4.1 and Algorithm 4.4 with the red lines.

The GA maintains an additional vector $\mathbf{T}_v$, indexed by the RA identifier, containing the setpoint validity times for each RA. Note that these represent the actual cut-off times, calculated by adding the advertisement timestamp to the validity horizon, as described in Algorithm 5.2. $\mathbf{T}_v$ is issued in each `issue` call (lines 10, 32).

$\mathbf{T}_v$ is updated in the `choose_advertisements` function, described in Algorithm 5.2. As mentioned earlier, this algorithm assigns, to each RA, a setpoint validity time equal to the validity horizon (τ or τ^n) added to the timestamp of the advertisement from that RA (lines 10, 10). The choice of τ depends on whether the short-term or long-term fields of the advertisement from that RA are chosen for computation.

Recall that the GA might probe setpoints, either during initialization (Algorithm 5.1 line 10), or due to a timeout flag being set (line 29). In both cases, the values of the validity horizon must be disregarded, as a probe setpoint does not use advertisements in its computation, and can always be received by the RA without violating reliable validity. We set the entries of $\mathbf{T}_v$ to all-zeros in those cases, and handle them explicitly at the masker side.

The vector of setpoints, the vector of validity times, and the message label are issued by the GA and intercepted by the tagger. We describe the operation of the tagger in the following subsection.

We note that the timestamping performed at the RAs for Axo requires no time synchronization. Each RA is comparing two instances of its local time, and the GA is only relaying the value of the timestamp after adding to it the absolute time corresponding to the validity horizon.

Recall from Chapter 2 that we do not consider, in our fault model, delays in setpoint implementation at the RA or at the actuators. Such an assumption is necessary, as one can only guarantee no time violations until the last check, which in this case is performed at the masker. Thus, the assumption of an upper bound on implementation time (δ_i) is justified. This, however, does not eliminate the possibility of delays in the RA when issuing advertisements. The presence of such delays might render the

Algorithm 5.1: Abstract model of a GA with Axo. The parts in red are added to Algorithm 4.1

```
1  A ← ∅;                                          // set of advertisements received from RAs
2  S ← ∅;                                          // set of states received from the SE
3  Z_A ← [ ];                                      // vector of advertisements used in a computation
4  T_v ← [0, 0, ...., 0];                          // vector of validity times
5  H ← ∅;                                          // internal state of the GA
6  C ← 0;                                          // logical clock on this GA
7
8  on initialization
9  |  X ← ∅;                                       // initialize vector of setpoints to probes
10 |  issue(X,T_v, C);                             // send probes to the RAs
11 end;
12
13 on reception of an advertisement adv with label ℓ from an RA i
14 |  A ← A ∪ {(adv, i, ℓ)};                       // aggregate received advertisements
15 |  C ← max(C, ℓ);
16 end;
17
18 on reception of a state st with timestamp T from the SE
19 |  S ← S ∪ {(st, T)};                          // aggregate received states
20 end;
21
22 repeat
23 |  A', T_v ← choose_advertisements(A, C);
24 |  T ← current time;                           // get the current time
25 |  ready, timeout, Z_A, st ← ready_to_compute(A', S, T);
26 |  if ready then
27 |  |  C ← C + 1;
28 |  |  X, H ← compute_setpoints(Z_A, st, timeout, H);
29 |  |  if timeout then
30 |  |  |  T_v ← [0, 0, ...., 0];                // reset to all-zeros
31 |  |  end
32 |  |  issue(X,T_v, C);                         // send X to the RAs
33 |  end
34 forever;
```

Algorithm 5.2: `choose_advertisements`($\mathcal{A}$, C) function in the GA with Axo. The parts in red are added to Algorithm 4.4

```
1  A' ← ∅;                                   // set of chosen advertisements
2  L ← {0};                                  // vector of highest label seen for each RA
3  T_v ← [0, 0, ...., 0];                    // vector of computed validity times
4  n;                                        // pre-configured value of long-term horizon
5
6  for each (adv, i, ℓ) ∈ A do
7      if ℓ > C − n and ℓ > L[i] then
8          if ℓ = C then
9              adv' ← (adv.F, adv.U, adv.T);
10             T_v[i] ← adv.τ + adv.T;
11         else
12             adv' ← (adv.F^n, adv.U^n, adv.T);
13             T_v[i] ← adv.τ^n + adv.T;
14         end
15         Remove advertisement with identifier i from A';
16         A' ← A' ∪ {(adv', i)};
17         L[i] ← ℓ;
18     end
19 end
20
21 return A', T_v;
```

computed timestamp unreliable. However, as mentioned in Chapter 3, the timestamp is taken after the implementation is performed, but before the state of the resource is measured, thus it is always conservative.

5.4.2 Fault Masking: Tagger & Masker

Together, the tagger and the masker achieve fault masking. The design of the tagger is shown in Algorithm 5.3. The tagger is configured with the unique identifier of the replica on which it resides (line 1). It also maintains two variables, ℓ_d and h, corresponding to the latest label and the latest health received from the detector, respectively (lines 2, 3). These are updated upon reception of new values from the detector (lines 5-8). They are further discussed in Section 5.4.3.

The tagger intercepts the messages issued by the controller. Specifically, for each destination RA i, the tagger receives the setpoint sp, its validity time t_v, and its label ℓ (line 10). Recall that t_v is computed as in Algorithm 5.2, and ℓ is computed as in Algorithm 4.1. The tagger then forms the `axoHeader` (line 11), and prepends it to the setpoint that it then sends to the masker of the RA i (line 12).

The Axo header consists of (1) a 1-byte unique replica identifier id, which serves to

Algorithm 5.3: Tagger

```
1  id;                                          // unique identifier of this replica
2  ℓ_d ← 0;                                     // latest label received from detector
3  h ← H_max;                                   // latest replica health received from detector
4
5  on reception of update {L, H} from the detector
6  |  ℓ_d ← L;
7  |  h ← H;
8  end;
9
10 on interception of {sp, t_v, ℓ} from the controller to RA i
11 |  axoHeader ← (id, ℓ, t_v, ℓ_d, h, i); // populate axo header
12 |  send (axoHeader, sp) to the masker of RA i
13 end;
```

identify the issuing replica and is used for fault detection, (2) an 8-byte label ℓ specifying the round number this setpoint belongs to (see Chapter 4), (3) an 8-byte timestamp t_v specifying the validity time of the setpoint, which is used at the masker for fault masking, (4) an 8-byte label ℓ_d, which along with (5) a 1-byte health field h, are later relayed by the masker to the detector and used for fault detection. Finally, (6) a 2-byte RA identifier i is included, enabling the masker to forward the setpoint to the correct RA port number. This adds up to a 28-byte Axo header.

The setpoint, along with the Axo header, is sent to the masker that processes it, as described in Algorithm 5.4. The masker was previously described in Algorithm 4.3, in which its role was to forward setpoints to the RA if the label of the setpoint is larger than the logical clock, corresponding to the highest label received. The parts in red in Algorithm 5.4 are the added parts that enable the masker to also account for the validity of the setpoint.

In line 11, the masker checks not only for the label of the setpoint, but also for the validity time. Only setpoints with a validity time greater than or equal to the reception time t_r are forwarded to the RA. This is in line with the definition of a valid setpoint (Definition 5.1). Additionally, probe setpoints with a validity time of zero are also forwarded, as discussed in Section 5.3.2.

The masker marks the validity of setpoints with the $valid$ flag (lines 10, 17, 20). This flag, along with the contents of the Axo header, except the RA identifier and the validity time, is used to form a validity report (line 23). This report is sent to the detectors of all controller replicas.

As mentioned in Section 5.1, 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

Algorithm 5.4: Masker. The parts in red are added to Algorithm 4.3

```
1  C ← −1;                                   // logical clock on the corresponding RA
2  adv ← ∅;                                  // last issued advertisement by the RA
3
4  on reboot
5  |  C ← stored C;                          // read value of C from hard storage
6  end;
7
8  on reception of a setpoint sp with header axoHeader from the GA
9  |  t_r ← current time;                    // get current reception time
10 |  valid ← false;                         // initialize the valid boolean
11 |  if axoHeader.ℓ > C and (t_r ≤ axoHeader.t_v or axoHeader.t_v = 0) then
12 |  |  forward(sp);                        // forward sp to the RA
13 |  |  C ← max(C, axoHeader.ℓ);
14 |  |  store C;                            // store value of C in hard storage
15 |  |  adv ← receive_from_RA();            // wait for adv from the RA
16 |  |  issue(adv, C);                      // issue adv to the GA
17 |  |  valid ← true;                       // setpoint was valid
18 |  else if axoHeader.ℓ = 0 then
19 |  |  issue(adv, C);                      // re-issue previous advertisement to the GA
20 |  |  valid ← true;                       // probe setpoints are always valid
21 |  end
22 |
23 |  report ← (axoHeader.id, axoHeader.ℓ, axoHeader.ℓ_d, axoHeader.h, valid);
24 |  Send report to detectors of all controller replicas;
25 end;
```

check, in both the design and implementation of the masker, is the forwarding of the setpoint to the RA. 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 [84, 151]. Furthermore, recall from Section 5.3.2 that we assume that the RA and the actuator will implement the forwarded setpoint within a duration upper bounded by δ_i. This is included in the computation of the validity horizon at the RA. Thus, reliable validity is guaranteed as shown in Theorem 5.1.

As Axo modifies the setpoints sent by the controllers to the RAs (by adding an Axo header), RAs that do not contain the RA library cannot process the modified setpoints. Therefore, Axo is not backward compatible: the RA library must be present on all RAs. Although reliable validity is guaranteed without the presence of the RA library, as no setpoints will be received by the RA, hence no invalid setpoints will be received, reliable availability suffers.

The computation of the validity times in the controller (Algorithm 5.2), the interception of the setpoint and populating the Axo header at the tagger (Algorithm 5.3), and the validity check performed at the masker (Algorithm 5.4), constitute the real-

time path of a setpoint. We see that the aforementioned algorithms perform simple operations and do not require additional communication over the network. Hence, they add negligible latency.

5.4.3 Fault Detection: Detector

The detection mechanism (Algorithm 5.5) is triggered at a replica when a validity report (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 ID of the replica that issued the setpoint ($vr.id$), (2) the label of the setpoint ($vr.\ell$), (3) the health of the replica that issued the setpoint, as seen internally by that replica ($vr.h$), (4) the detector label computed as the highest setpoint label that the detector of the issuing replica had processed ($vr.\ell_d$), and (5) a flag that shows whether the setpoint received at the masker was valid ($vr.valid$).

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 label (ℓ) $\mathbf{DB}[id].\ell$, (2) the highest received detector label $\mathbf{DB}[id].\ell_d$, (3) the replica's health as seen by this replica $\mathbf{DB}[id].h$, and (4) a flag $\mathbf{DB}[id].nf$, denoting whether the replica is considered non-faulty for the setpoint with label $\ell = \mathbf{DB}[id].\ell$. 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 (ℓ) and the detector timestamp (ℓ_d) are set to zero, and the non-faulty flag (nf) field is set to true.

A validity report (vr) is identified by its setpoint label field ($vr.\ell$). 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 $vr.valid$ flag is set to false). To achieve this, the detector performs a logical OR of the $vr.valid$ flags received in reports from the same replica with the same setpoint timestamp (lines 37-39). The database is updated (according to Algorithm 5.6) only after a new report is received, i.e., report with a higher ℓ 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 5.6 lines 1-2, the labels ℓ and ℓ_d are set to the corresponding ones received in the validity report. Lines 4-8 show how the health of a replica is updated.

Algorithm 5.5: Detector

```
1  initialize DB[myID];                                   // initialize database entry of this replica
2
3  on reception of a report vr from a masker
4      if vr.id ∉ DB or vr.ℓ > DB[vr.id].ℓ then
5          if vr.id ∉ DB then
6              create DB[vr.id];                          // create new entry in database
7              DB[vr.id].ℓ ← vr.ℓ;
8              DB[vr.id].ℓ_d ← vr.ℓ_d;
9              DB[vr.id].nf ← vr.valid;
10             DB[vr.id].h ← H_max;
11         else
12             DB[vr.id].h ← min(vr.h, DB[vr.id].h);
13             updateDB(DB[vr.id], vr);                   // update database entry
14         end
15
16         DB[myID].ℓ_d ← max(DB.ℓ);                      // update detector label
17         Send DB[myID].h, DB[myID].ℓ_d to local tagger;
18
19         if vr.id ≠ myID and DB[vr.id].h ≤ H_ext then
20             EXT_DET(DB[vr.id].ℓ, vr.id);               // trigger external delay detection
21             delete DB[vr.id];                          // delete entry from database
22         else if DB[myID].h ≤ H_int then
23             INT_DET(DB[myID].ℓ);                       // trigger internal delay detection
24         end
25
26         for each id ∈ DB do
27             if max(DB.ℓ) − DB[id].ℓ > L_c or max(DB.ℓ_d) − DB[id].ℓ_d > L_c then
28                 if id = myID then
29                     INT_DET(max(DB.ℓ));                // trigger internal crash detection
30                 else
31                     EXT_DET(max(DB.ℓ), id);            // trigger external crash detection
32                     delete DB[id];                     // delete entry from database
33                 end
34             end
35         end
36
37     else if vr.ℓ = DB[vr.id].ℓ then
38         DB[vr.id].nf ← DB[vr.id].nf ∨ vr.valid;        // update non-faulty flag
39     end
40 end;
```

Algorithm 5.6: `updateDB`(db, vr) function in the detector

```
1  db.ℓ ← vr.ℓ;
2  db.ℓ_d ← vr.ℓ_d;
3
4  if db.nf then
5  |  db.h ← α × db.h + (1 − α) × H_max;        // increase health of non-faulty replica
6  else
7  |  db.h ← α × db.h − (1 − α) × H_max;        // decrease health of -faulty replica
8  end
9
10 db.nf ← vr.valid;
```

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 enables considering the rate of delay faults, rather than the presence or absence of faults in a given round. Thus, it considers a sliding window of previous rounds, depending on the value of α. This 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 10).

In Algorithm 5.5 at line 16, the detector label of this replica is computed as the highest setpoint label processed by this detector. This field is used by other detectors to detect crashes of this detector. The detector label 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 11-14), 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 (lines 19-24) 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 5.4.4).

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 26-35. The detector compares, for each replica in its database, the value of the setpoint label ℓ with the maximum of all ℓ's. If the difference is greater than some threshold L_c, then that replica is deemed crash faulty, as it has not sent setpoints in L_c control rounds. In this way, a replica is only considered to be crash faulty if it has been inactive for a L_c rounds, *while other replicas have been active*. This relative comparison allows incorporating CPSs with a non-constant rate of issuing setpoints. A similar comparison is done for ℓ_d's, to detect crashes in the detector.

5.4.4 Fault Recovery: Rebooter

The rebooter, according to Algorithm 5.7, (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 label that corresponds to the report that triggered the detection (see Algorithm 5.5, lines 20, 23, 29, 31). The rebooter saves this label on disk when it triggers a reboot, and loads it upon booting. The loaded label ℓ_r 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 T_r is also used to avoid multiple reboots in the presence of message losses. For example, a lost report to one controller replica 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 labels.

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 5.7, lines 27-36). This is done at most $maxSend$ times, once every T_r. The `ACK`s sent also contain the label of the received `EXT_REC` message, plus the threshold L_r. 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 5.7, note that internal recovery is faster than external recovery, as it is performed locally and does 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 5.4.3.

Algorithm 5.7: Rebooter

```
1  ℓ_r ← 0;                                                  // label of last reboot
2
3  on reboot
4  |  ℓ_r ← stored ℓ_r;
5  end;
6
7  on reception of INT_DET[ℓ]
8  |  if ℓ > ℓ_r + L_r then
9  |  |  ℓ_r ← ℓ;
10 |  |  store ℓ_r;
11 |  |  reboot the replica;
12 |  end
13 end;
14
15 on reception of EXT_DET[ℓ_1, id]
16 |  sentCtr ← 0;
17 |  while sentCtr < maxSend do
18 |  |  send EXT_REC(ℓ_1) to replica id;
19 |  |  sentCtr++;
20 |  |  listen for T_r ;
21 |  |  if (ACK(ℓ_2) received) and (ℓ_2 ≥ ℓ_1) then
22 |  |  |  break;
23 |  |  end
24 |  end
25 end;
26
27 on reception of EXT_REC[ℓ]
28 |  if ℓ > ℓ_r + L_r then
29 |  |  send ACK(ℓ + T_r) to all replicas;
30 |  |  ℓ_r ← ℓ;
31 |  |  store ℓ_r;
32 |  |  reboot the replica;
33 |  else
34 |  |  send ACK(ℓ_r + T_r) to all replicas;
35 |  end
36 end;
```

Given that the rebooter needs to respond to remote reboot requests, it needs to be non-susceptible to crash faults. Otherwise, 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.

5.5 Formal Guarantees

We say that a CPS implements Axo if (1) its RAs include the validity horizon in the advertisement as described in Section 5.4.1, (2) its GA is modified according to Algorithm 5.1, (3) a tagger, detector, and rebooter are present on all GA replicas, according to Algorithms 5.3, 5.5, 5.7, and (4) a masker is present on all RAs, according to Algorithm 5.4.

In this section, we prove that Axo guarantees reliable validity by discarding invalid setpoints at the masker before they reach the RA. Then, we show that Axo only discards invalid setpoints, thereby preserving reliable availability. Recall the definitions of reliable validity (Definition 3.8), reliable availability (Definition 3.9), and valid setpoints (Definition 5.1).

Theorem 5.1 (Reliable Validity)**.** *A CPS that implements Axo guarantees reliable validity.*

Proof. Let adv be an advertisement computed with a timestamp T at some RA i.
The RA includes the validity horizon $\tau = n \times T_{ctrl} - \delta_i$ in adv (Section 5.3.2).
If a GA computes and issues a setpoint sp to RA i that uses adv in its computation, the GA includes $t_v = T + \tau$ when issuing sp (Algorithm 5.2, lines 10, 13).
If the GA does not use any advertisement from RA i in its computation of sp, then sp is a probe and cannot be invalid.

The tagger intercepts sp and t_v.
If the tagger is faulty, the setpoint does not reach the masker or the RA and cannot violate reliable validity.
The tagger preprends the Axo header to sp, including t_v in the header, and sends it to the masker or RA i (Algorithm 5.3, lines 11-12). If the message is lost, then reliable validity cannot be violated.

Otherwise, the masker computes the reception time t_r after receiving sp (Algorithm 5.4).
The masker only forwards sp to the RA in two conditions.
The first is if $t_v = 0$, which means according to Algorithm 5.1, that the setpoint is a probe issued because of a timeout.
Thus, it cannot be invalid.
The second is if $t_v \geq t_r$.

This implies that $t_r \leq T + \tau$, which is the definition of a valid setpoint (Definition 5.1). Therefore, for any setpoint sp that is invalid, sp is not forwarded to the RA.

Besides proving the statement of the theorem, we discuss the computation of the validity horizon at the RAs.
The advertisement fields are computed at time $t_0 \geq T$ (Algorithm 3.1, lines 11, 13).
Thus, if $t_r \leq T + \tau$, then $t_r \leq t_0 + \tau$.
Moreover, as discussed in Section 5.3.2, after the masker forwards the setpoint to the RA, the RA implements the setpoint after a duration upper-bounded by δ_i.
Recall that $\tau = n \times T_{ctrl} - \delta_i$.
Therefore, if an RA receives a setpoint at $t_r \leq t_0 + \tau$, it will implement it at $t_i \leq t_r + \delta_i$.
Thus, $t_i \leq t_0 + n \times T_{ctrl}$.
This ensures that the setpoint is implemented within the validity horizon of the advertisement used in its computation.
It also ensures that the RA can continue implementing the setpoint for a duration of T_{ctrl} (included in that validity horizon), until it receives the next setpoint.
Recall that the GAs account for the possibility of a fault preventing the RA from receiving the next setpoint, as they guarantee robust safety (Theorem 4.4). □

Theorem 5.2 (Reliable Availability: Axo)**.** *Axo never discards valid setpoints.*

Proof. As Axo uses active replication, and does not interfere with the issuing of setpoints to the RAs, then the maskers receive as many setpoints as with any CPS using g replicas.
The maskers discard some setpoints instead of forwarding them to the RAs.
We show that Axo minimizes the number of setpoints discarded, while guaranteeing reliable ordering and validity.

The masker for RA i discards setpoints such that their reception time $t_r > T + \tau$, where T is the timestamp of the advertisement from RA i used in the computation of the setpoint, and τ is the validity time of that setpoint.
By Definition 5.1, all such setpoints are invalid.
Therefore, Axo only discards invalid setpoints. □

5.6 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 setpoints issued by the controller replicas, 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.

We consider a CPS that consists of controller replicas that issues setpoints according to a Poisson process of rate λ_n when non-faulty and rate λ_f when faulty. This follows from the fault model described in Chapter 3, Figure 3.3. On one hand, we approximate the rate of issuing setpoints when the replica is non-faulty as $\lambda_n \approx 1/T_{ctrl}$. As the rate is small in real-time systems, this approximation is justified. It also allows us to consider slight variations in the control period, brought about by the jitter in computation and communication latency. On the other hand, when the replica is faulty, the rate of issuing setpoints decreases. Therefore, $\lambda_f < \lambda_n$.

Furthermore, recall that θ_d and θ_c represent the probability of a delay fault and a crash fault, respectively. We define θ as the probability that either fault occurs, such that $\theta = \theta_c + \theta_c$. This simplifies the Gilbert-Elliot model presented in Section 3.3.2, allowing us to derive the results in Theorem 5.3 and Theorem 5.4.

Using the previously defined terms, the stationary probability of a replica being non-faulty (π_n) can be derived as $\pi_n = \frac{\lambda_f(1-\theta)}{\lambda_f(1-\theta)+\lambda_n\theta}$, and the average rate of sending setpoints $\lambda_0 = \lambda_f\,(1-\pi_n) + \lambda_n\,\pi_n$.

The network between controller replicas and the RAs is considered to have a round-trip time upper bounded by $2\delta_n$ and a packet loss probability of p, as described in Section 3.2.3.

We assume that at least one RA, including its masker, is non-faulty. This assumption is valid as it would not make sense to speak of fault tolerance when all RAs are faulty: the processes would be uncontrollable. Additional non-faulty RAs will only improve the time for detection and recovery.

5.6.1 Analytical Evaluation of Recovery Time

The exact evaluation of the expression of recovery time appears to be mathematically intractable. Instead, we derive a lower bound and an upper bound on the recovery time. In Section 5.6.2, we validate these bounds by comparing them with experimental results.

Theorem 5.3 gives the expressions for the detection and recovery of a delay-faulty controller replica. Theorem 5.4 gives the same for a crash-faulty controller replica. The proofs of these theorems can be found in Appendix A.1 and Appendix A.2, respectively.

Theorem 5.3 (Delay-Faulty Controller)**.** *In a CPS 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 $(\mathbb{P}_d^l(t))$ and upper bound $(\mathbb{P}_d^u(t))$ on the probability that it is recovered by time t is given as follows:*

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

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

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

where,

$$\beta = (1-p)/T_r,\ \gamma = \lambda_f(1-p)^2,\ \eta = \lambda_n\theta,\ N = \frac{log(\frac{1}{2}(\frac{H_{ext}}{H_{max}}+1))}{log(\alpha)},\ \Gamma(x,s) = \int_s^\infty t^{x-1}e^{-t}dt$$

Theorem 5.4 (Crash-Faulty Controller)**.** *In a CPS 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 $(\mathbb{P}_c^l(t))$ and upper bound $(\mathbb{P}_c^u(t))$ on the probability that it is recovered by time t is given as follows:*

$$\mathbb{P}_c^l(t) = \mathbb{P}_2(t - 2\delta_n)$$

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

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

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

where,

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

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

5.6.2 Experimental Validation

We implemented the Axo components in C++, and use them along with a test controller for these experiments. We use three replicas ($g = 3$) each with a test controller and the Axo controller library, and one RA with an Axo RA 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 $\lambda_n = 1/100\ s^{-1}$ (considering that the control period T_{ctrl} is 100 ms) when the controller is non-faulty and $\lambda_f = 1/200\ s^{-1}$ when the controller is faulty. The RA 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 5.2, a replica is considered delay faulty if the last setpoint it sent to the RA is received after $T+\tau$. We configure the validity horizon to be $\tau = 97$ ms, and the one-way network latency to be $\delta_n = 2$ ms. We take the crash-fault detection threshold L_c to be 5 control rounds. Thus, $\tau_c = 5 \times T_{ctrl} = 500\ ms$.

We study the variation of the distribution of recovery time as a function of the probability of a controller fault 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.

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 CPSs. 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 5.3 shows the results of the experimental simulation of a delay-faulty C_0, with $p = 1\%$ and a varying θ. Figure 5.4 shows 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.

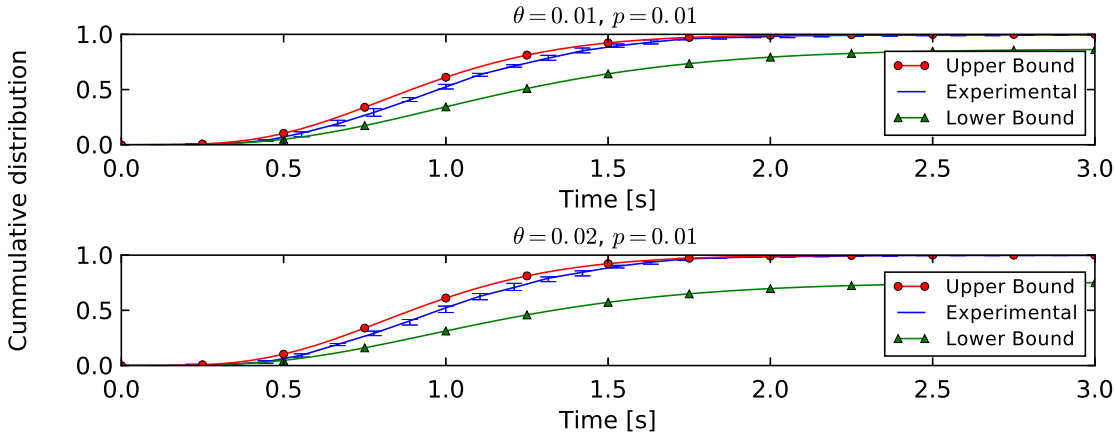


Figure 5.3 – Time to recover from delay-faults for varying fault rate θ

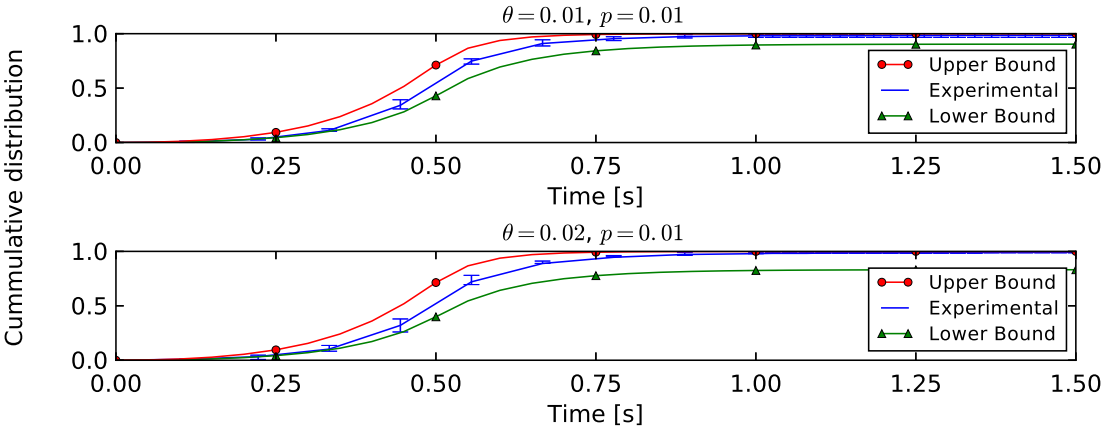


Figure 5.4 – Time to recover from crash-faults for varying fault rate θ

5.7 Stability Analysis: An Inverted Pendulum

In this section, we demonstrate how applying Axo affects the stability of an inverted pendulum system. We use the example in [153], 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 [154], 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 actual Axo code, rather than simulate it.

Figure 5.5 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

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 5.1 – Results of select scenarios with varying θ_d

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 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 ($\phi > 20^\circ$ 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 ($\phi > 35^\circ$).

Figure 5.6 shows the additional stability brought about by using Axo for a representative fault-scenario (θ_d = 10^{-3}). Table 5.1 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.

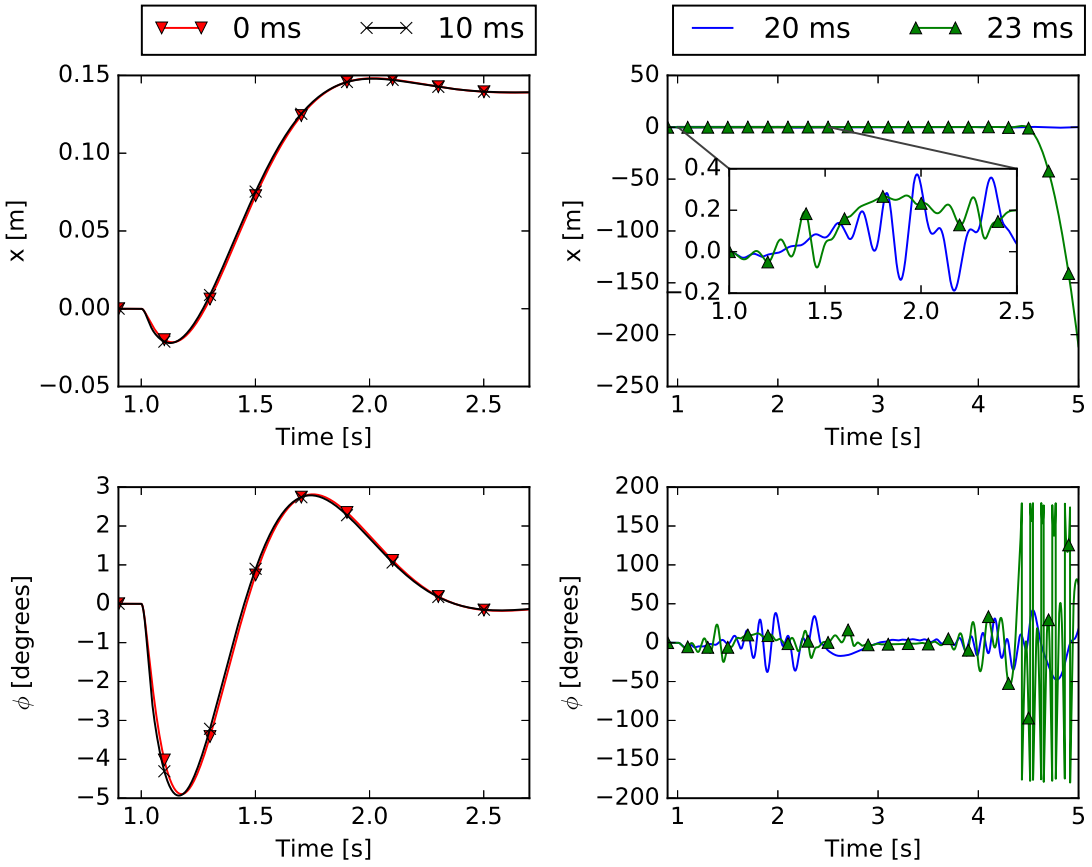


Figure 5.5 – Step response of the pendulum with a non-replicated controller

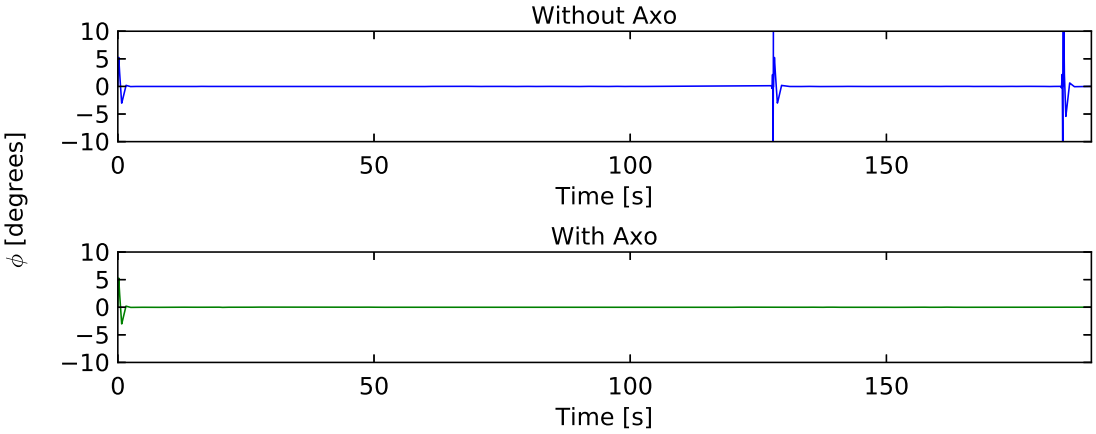


Figure 5.6 – Stability of the pendulum with a replicated controller

5.8 Conclusion

We have presented Axo, the first protocol that enables real-time CPSs to tolerate delay and crash faults of its controller. Axo uses active replication of the controller, ensuring that one controller replica is always available to receive input, perform computations, and issue setpoints to the RAs. In order to mask delay faults, Axo discards invalid setpoints at the masker, before the reach the RAs. We have proven that the fault-masking mechanism of Axo guarantees reliable validity and preserves reliable availability.

In order to detect and recover from delay and crash faults, Axo makes use of the round-based communication scheme between the controller and the RAs, and sends validity reports from the masker to the detector of each replica. The detection and recovery mechanisms of Axo are designed to be soft state, to enable the seamless addition of new replicas and removal of faulty replicas. We have analytically characterized the time to recover a faulty replica, and have experimentally validated the expressions.

Finally, we have performed a stability analysis of an inverted pendulum system, to study the effect of delay faults on the stability. We have shown that, by detecting and recovering from delay faults, Axo improves the stability.

Axo is designed to be controller-agnostic. This enables it to be generic, and facilitates its deployment to a wide range of CPSs. In Chapter 8, we deploy Axo with COMMELEC [8], and we demonstrate the fault-tolerance properties of Axo in greater detail.

As mentioned in Chapter 2, active replication might introduce inconsistencies, resulting in the split-brain syndrome [121]. This issue is handled in Chapter 6, in which we introduce Quarts, an agreement protocol that guarantees reliable consistency in real-time CPSs.

6 Quarts: Quick Agreement in Cyber-Physical Systems

A distributed system is one in which the failure
of a computer you didn't even know existed
can render your own computer unusable.
— Leslie Lamport

In Chapter 5, we have presented Axo, a fault-tolerance solution based on active replication. With active replication, each of the controller replicas receives measurements and advertisements, computes setpoints, and issues them. In Chapter 4, we have presented Robuster, a mechanism that enables a GA to compute setpoints even when some advertisements are missing. Due to non-uniform network losses, two replicas might receive a different subset of advertisements in a given control round. With Robuster, both replicas can compute a vector setpoints that maintains robust safety (Theorem 4.4). However, these vectors might differ, and the interleaving of their setpoints at the RAs results in what is referred to as the *split-brain* syndrome [121]. The split-brain syndrome and its effect on CPS correctness are further discussed in Section 6.1.

In order to address the split-brain syndrome, we have introduced the reliable consistency property (Definition 3.10), which states that setpoints issued in the same control round to the same RA must have the same value. Traditional methods of guaranteeing consistency rely on consensus [96]. However, as we have discussed in Chapter 2, consensus mechanisms incur a high latency-overhead and suffer from a reduced availability.

In Section 6.3, we identify and present a set of properties, of the controllers and RAs, generally exhibited by CPSs for real-time control of electric grids. With these properties in mind, we are able to relax the requirements on consensus for these types of systems. Specifically, they enable us to forgo the termination requirement, which, for generic consensus protocols, requires all non-faulty replicas to decide on a value.

We designed Quarts, an agreement protocol that suits CPSs with the aforementioned properties. Quarts guarantees agreement between the replicas, and forgoes termination. That is, it does not require all replicas to decide on a value in each round, thus permitting some to not perform a computation. Instead, Quarts re-synchronizes such replicas, called *stragglers*, in the following rounds.

The design of Quarts is presented in Section 6.4. Quarts is designed to provide agreement on a vector of messages with a specific property (see Equation 6.1). This property applies to the input messages to the controller in a CPS. Thus, Quarts can be used in the replicas to agree on the input used for computation. Considering a deterministic computation function, this in turn provides agreement on the output.

In Section 6.5, we show how Quarts can be applied to CPSs. Specifically, we use Quarts to provide agreement between SE replicas, ensuring that the states they send to their corresponding GAs are consistent. Then, we use Quarts to provide agreement between GA replicas, ensuring that the vectors of setpoints they compute and issue to the RAs are consistent. Thus, the controller output is guaranteed to be consistent.

In Section 6.6, we provide a formal proof that applying Quarts to CPSs guarantees reliable consistency. Also, we show that the latency overhead incurred by CPSs due to Quarts is upper-bounded. Recall that, in contrast, consensus mechanisms incur an unbounded latency-overhead.

Finally, we use discrete-event simulation to compare the performance of Quarts with Fast Paxos [104], a state-of-the-art consensus mechanism. We also compare Quarts with passive-replication solutions that attempt to circumvent the consistency issue by having only one replica compute and issue setpoints. The performance metrics considered in our evaluation are availability, latency, and messaging cost.

We show that, as mentioned in Chapter 2, passive replication cannot guarantee consistency, as it relies on failure detection, which is inherently imperfect [38]. Passive replication was designed for a crash-only fault model, and we observe that in scenarios in which we do not consider delay faults, passive replication performs as intended. However, when delay faults are introduced, we observe inconsistencies, in addition to an increase in latency and a decrease in availability. We also observe that Fast Paxos suffers from high latency and low availability, when compared to Quarts.

Quarts provides an availability that is one order of magnitude higher than that of other solutions when two controller replicas are used, and its availability increases with the number of replicas, as opposed to other solutions. Quarts also provides lower latency than Fast Paxos, both in the average case and in the tail. The price to pay for guaranteeing consistency, both for Quarts and Fast Paxos, is a marginal increase in message cost. Our results are presented in Section 6.7.

6.1 The Split-Brain Syndrome

The split-brain syndrome [121] is a term used in distributed computing to refer to a process receiving conflicting control messages from multiple controllers. In traditional computing, this might refer to multiple computers simultaneously writing to disk, which causes data contamination. In CPSs, an example would be an autonomous vehicle receiving an instruction to turn left from one controller, and to turn right from another. This would result in the vehicle exhibiting an undefined behavior, and potentially leads to safety hazards.

6.1.1 Causes

Recall from Chapter 3 that, in this thesis, we consider software agent delays and crashes, and message losses, omissions, delays, and reordering. Delays might lead to different controller replicas receiving their input messages at different time instances. Losses, omissions, and crashes might lead to them receiving different subsets of input messages, due to non-uniform losses. With Robuster (Chapter 4), controller replicas can compute setpoints that maintain grid safety, even when some input advertisements are missing. Therefore, two different controller replicas, with a different subset of input advertisements, might both compute and issue a vector of setpoints in a given control round r. Given that these vectors were computed using a different combination of short-term fields and long-term fields of advertisements, they will not necessarily have the same setpoint values. We observe the effect this might have on the CPS in the example in the following subsection.

Several other factors might result in the split-brain syndrome, most of which we have addressed in earlier chapters. For example, a delayed or an out-of-order setpoint received at an RA might be in conflict with another valid in-order setpoint. However, intentionality clocks (Chapter 4), and Axo (Chapter 5) guarantee reliable ordering and validity (Definitions 3.7, 3.8), respectively, ensuring that such a cause does not affect CPSs that use these solutions.

Another possible cause for the split-brain syndrome are Byzantine faults [39]. Such faults are caused by software bugs, security attacks, and hardware faults, and might result in an erroneous computation by one of the replicas. Byzantine-fault tolerance (BFT) solutions address this particular cause, but, as mentioned earlier, Byzantine faults are not considered in the fault model in this thesis, and are hence not addressed.

6.1.2 Effects

Consider the example shown in Figure 6.1. Two GA replicas, GA_1 and GA_2, monitor and control a grid that consists of two resources: a battery and a PV. The main line of

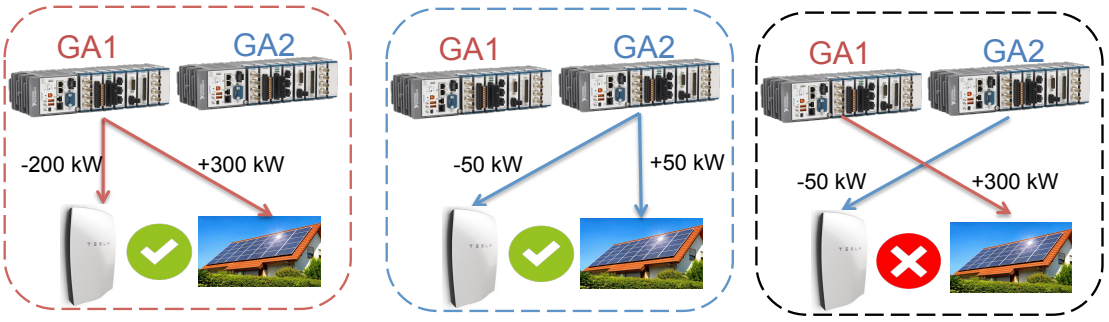


Figure 6.1 – Interleaving of setpoints as a result of the split-brain syndrome

grid has an ampacity limit of 200 A, which corresponds to approximately 140 kW. The GAs, therefore, must maintain the power imbalance in the grid below this limit.

Suppose that in control round 9, both GAs received advertisements from both RAs. The PV RA included in that advertisement, a conservative long-term uncertainty, accounting for a possible change in weather conditions in the next second.

In control round 10, GA_1 receives advertisements from both RAs. The PV RA indicates in its short-term fields that it can inject up to 300 kW, and the battery indicates it can absorb up to 200 kW. GA_1 computes and issues the vector of setpoints $\mathbf{X}_1 =< +300, -200 >$[1]. This maximizes the utilization of the PV, and maintains the imbalance to 100 kW, well under the 140 kW limit.

GA_2, in control round 10, receives an advertisement only from the battery RA. It computes, using the long-term fields of the previous advertisement of the PV, a vector of setpoints $\mathbf{X}_2 =< +50, -50 >$. This considers the possibility that the PV might not be capable of exporting its maximum of 300 kW, due to the increased uncertainty in its long-term fields. It also decreases the battery absorption to match the PV injection, and to maintain a low power imbalance.

Although an implementation of either one of these computed vectors $\mathbf{X}_1$ and $\mathbf{X}_2$ maintains grid safety, an interleaving of their setpoints at the RAs might not. Consider a sequence of message losses that causes both the following: (1) the PV RA receives only the setpoint from GA_1, instructing it to inject 300 kW, and (2) the battery RA receives only the setpoint from GA_2, instructing it to absorb 50 kW. The resulting power imbalance would be 250 kW, violating the ampacity limit, and possibility causing damage to the grid and the resources.

In Chapter 4, we have discussed that a robust GA must consider all the possibilities of implementation, taking into account the uncertainties of the resources, and the possibilities of message losses. However, a GA cannot take into account all the possible computations its replicas might perform. In order to circumvent this issue, the GA

[1]Note that the subscript in this example refers to the GA that computed the vector of setpoints, rather than the round number r.

replicas must guarantee reliable consistency (Definition 3.10). If reliable consistency were satisfied in the example above, $\mathbf{X}_1$ and $\mathbf{X}_2$ would contain the same setpoints, thereby eliminating the problem of interleaving, and consequently handling the split-brain issue.

We note that the above example is a pathological case. However, as we have mentioned, the GA can account for a loss in availability, but not for a violation of consistency. Even though consistency might be violated only in rare occasions, over the long-term deployment of a CPS, it becomes a rather likely possibility.

6.1.3 Proposed Solution: Quarts

We propose Quarts, an agreement protocol that is tailored for real-time CPSs. Quarts guarantees consistency with a low bounded latency-overhead and a high availability. Therefore, it does not suffer from the same drawbacks as traditional, more generic, consensus solutions. However, Quarts can only be used to agree on messages of a specific format, hence can only be applied to CPSs that satisfy a certain set of requirements (mentioned in Section 6.3). These requirements are generally satisfied by CPSs for real-time control of electric grids.

When applied to CPS controllers, Quarts is used to agree on the input used for computation, rather than the output issued. SE replicas use Quarts to agree on the set of measurements to be used in a computation of the state to be sent to the GAs. Similarly, GA replicas use Quarts to agree on the set of advertisements, and on the SE state, to be used in the computation of setpoints in each control round. Additionally, as SEs and GAs have an internal state (represented as $\mathcal{H}$ in Algorithms 3.2, 3.4), Quarts is also used to synchronize this internal state between the replicas. Therefore, as all the parameters used for computation are agreed upon by the replicas, the resulting output (SE state or GA setpoints) are guaranteed to be the same.

Note that Quarts assumes a deterministic computation. That is, for the same set of parameters (input messages and internal state) to the computation function, the output is the same.

6.2 Related Work

Ensuring consistency among replicated controllers is a well-studied problem in the literature. Two types of solutions tackle this problem from different perspectives: passive replication [123, 124, 125, 126] and consensus-based active replication [96, 101, 102, 104].

6.2.1 Passive Replication

Passive-replication solutions circumvent the consistency problem by having only one replica — the *primary* — receive inputs, and compute and issue setpoints. As other replicas — the *standbys* — are not issuing any output, the consistency problem is virtually non-existent. Instead of computing, standby replicas monitor the primary in order to detect it when it is faulty. Upon detecting the primary as faulty, the standby replicas elect, among themselves, a replica to act as the new primary.

This approach was designed for crash-only faults, and performs best when only crashes are considered. However, as discussed in Chapter 2, passive replication experiences deteriorating performance, in terms of consistency, availability, and latency, when delay faults occur. Most importantly for consistency, an imperfect failure detection can render the system with two primaries, resulting in potential inconsistencies, as they are not explicitly handled.

Although the rest of the chapters focus on active replication, in this chapter we cover passive replication for completeness, and to highlight its drawbacks in the presence of delay faults. In Section 6.7, we simulate two types of passive-replication schemes, which we refer to as passive hot (PH) and passive cold (PC). PH is a passive-replication in which the standby replicas are *hot, i.e.,* they are constantly synchronized with the primary. In contrast, the standby replicas in PC are not synchronized, and are referred to as *cold*. PH incurs additional latency and messaging cost in each control round with respect to PC. However, in PH, when the primary fails and a standby replica is elected as a new primary, the latter can immediately participate in the computation of the next control round, as it is already synchronized. In contrast, in PC, the new primary has to wait an additional round to synchronize, thereby suffering from reduced availability.

6.2.2 Active Replication with Consensus

With active replication, ensuring consistency requires reaching an agreement between the replicas. Traditionally, such agreement is reached using consensus [96], whereby the replicas agree on either: (1) the output to send (vector of setpoints or SE state in our case), or (2) which replica issues the output. As discussed in Section 2.3.2, consensus has four main properties: agreement, termination, validity, and integrity. Validity and integrity are mainly included to handle Byzantine faults, hence can be left aside. However, agreement and termination are impossible together, in the presence of faults [98].

The impossibility result [98] stipulates that any consensus protocol involves a trade-off between availability (termination) and consistency (agreement). Several protocols [101,102,104] have been proposed to provide consensus under different fault conditions. Fast Paxos [104] is a state-of-the-art protocol that provides a low-latency

agreement when no faults occur. However, as with any generic consensus protocol, its latency-overhead cannot be bounded under faulty conditions. Consequently, it suffers from reduced availability. In Section 6.7, we simulate Fast Paxos (FP), and compare it with Quarts and the previously mentioned passive-replication solutions.

In contrast, Quarts is not a generic consensus protocol. Quarts can only be used to reach agreement on messages of a specific format, and hence it only applies to CPSs that satisfy the requirements mentioned in Section 6.3. Controller replicas in CPSs to which Quarts applies are soft state, *i.e.,* they can be re-synchronized in one time step in case they do not take part in some control rounds. Therefore, Quarts is able to forgo termination in favor of agreement. That is, Quarts does not require all replicas to terminate in each control round. Instead, the replicas that do terminate are guaranteed to have agreement, and the other replicas do not issue any output.

Furthermore, Quarts is used to agree on the inputs used for computation, rather than on the output as in traditional consensus protocols. This is done in two phases: (1) the *collection phase* involves the replicas exchanging the input they received in each control round, and (2) the *voting phase* involves the replicas performing a vote to reach an agreement on which set of input to use in the computation.

The collection phase is used to increase the number of common messages in the replicas, thereby facilitating the voting phase and increasing the chances of a successful vote. The voter used in Quarts is plurality-based [155]. In plurality voting, also known as relative-majority voting, the option with the most votes is chosen. In contrast, (absolute-) majority voting requires the option to have more than half of the total votes to be chosen. Plurality, therefore, is more suited for CPSs in which availability hinges on a successful vote. The voter in Quarts is also composite [156], *i.e.,* when there is a tie between multiple options, one is chosen based on a predetermined priority. However, unlike in [156], in which the priority is replica-based, the priority in Quarts is based on the number of input messages received. This has two advantages: (1) it enables the CPS to give higher priority to replicas with more recent information, and (2) it does not restrict the priority to a replica that might become faulty.

6.3 Required CPS Properties for Quarts

In this section, we enumerate, and give the intuition behind, the CPS properties required to apply Quarts. The properties mentioned here all hold for CPSs for real-time control of electric grids. They are either assumed in the system model described in Chapter 3, or provided by the mechanisms described in Chapter 4. In Section 6.4, we highlight the necessity of these properties as we describe the design of Quarts.

Property 6.1. *Control messages, exchanged by software agents, are labeled such that they can be grouped into computation rounds.*

Property 6.1 refers to two types of messages. First, it refers to the measurements issued by asynchronous sensors. These must be labeled in such a way to enable the SE to group them into a vector that will be used in a given computation. As mentioned in Chapter 3, we consider that the asynchronous sensors send timestamped measurements that can be time-aligned by the SE, thereby assigning a common timestamp (label) to these measurements in each computation round.

The second type of message the property refers to are the advertisements issued by RAs. These can be labeled using intentionality clocks (Chapter 4) that assigns round-based labels to such messages. Quarts is designed to agree on a vector of messages to be used in a computation, hence this property is necessary.

Property 6.2. *The controllers can compute with intermittent input messages.*

This property is two-fold. First, it means that the controller can compute and issue output even when some input messages are missing in a certain round. Second, it means that controllers that miss a computation round, due to a delay fault for example, can still take part in the next computation round. That is, the input messages received are enough to perform a correct computation, regardless of internal state. This property enables Quarts to forgo termination of some replicas in a control round, without sacrificing availability in the next rounds.

State-of-the-art SEs, as discussed in Chapter 2, can compute with missing measurements [62, 63]. Their internal state, e.g. the gain matrix in a Kalman filter [132], serves to improve the accuracy of the computation. They can, however, perform a correct computation even if the state is not up-to-date, albeit with degraded accuracy. The input measurements, therefore, are enough to perform a correct computation.

Robuster (Chapter 4) provides a mechanism for the GA to compute with missing input advertisements. Moreover, the state in the GA with Robuster consists of the latest advertisement received from each RA (containing the long-term fields). Hence, missing a computation round does not negate the ability of the GA to take part in future computation rounds.

Property 6.3. *The internal state of a controller can be known exactly.*

As replicas applying Quarts must use the same internal state to compute a consistent output, the state of the controller must be known exactly so that it can be exchanged with stragglers. This property holds for state-of-the-art SEs using Kalman filters, as their internal state is the gain matrix. It also holds for GAs implementing Robuster, as their internal state consists of the latest advertisements received from each RA. We consider that any additional internal state that the GA maintains can be similarly known and extracted.

Property 6.4. *RAs are able to handle duplicate setpoints.*

This is the same as Property 5.2 required in Axo (Chapter 5). It is required for Quarts as well, since Quarts relies on active replication and issues multiple setpoints for each RA. However, as mentioned in Chapter 5, the masker discards duplicate setpoints before they are implemented at the RA. Therefore, this property is trivially satisfied.

Property 6.5. *RAs and asynchronous sensors are time-synchronized.*

Property 6.6. *There exists a lower bound (σ) on the time interval between two measuring events at asynchronous sensors.*

Properties 6.5 and 6.6 are essential for a GA using Quarts to group the advertisements in a given round r, to the SE state with a timestamp T. As we describe in Section 6.5, the GA uses the timestamps of the chosen advertisements in order to select a timestamp of the SE state. The time-synchronization serves to ensure that the SE state, used in a given computation, represents the same state of the grid encapsulated in the advertisements. We discuss the need for the lower bound (σ) in Sections 6.4, 6.5.

Throughout this chapter, we assume that the number of replicas (g) in the CPS is known to all controller replicas. Addition and removal of replicas occurs on the non-real-time path. When adding or removing replicas, an upper-bound on their number is maintained. This marginally reduces the availability until the new number is agreed upon, but does not affect the correctness guarantees.

6.4 Quarts Design

Quarts is an agreement protocol for replicated controllers. A instance of the Quarts protocol is instantiated at each controller replica, with an input being a set of messages of the form $(r,\ i,\ v)$, where r is a common label across all messages, i is a unique identifier in the set, and v is the value. All possible identifiers are known *a priori*, which means that an upper bound n on their number is also known. Across replicas, the sets having the same label r must maintain the following:

$$\forall\,(r,\ i_1,\ v_1),\ (r,\ i_2,\ v_2),\ i_1 = i_2 \implies v_1 = v_2 \tag{6.1}$$

In other words, for a given label r, each replica instantiates Quarts with a vector $\mathbf{V}$ of size n. Each element of $\mathbf{V}$ corresponds to one unique identifier, either containing the value of the message with that identifier, or being empty. Across replicas, non-empty elements of the same identifier must have the same value.

Consider the example shown in Figure 6.2, in which 3 controller replicas instantiate Quarts for a common label r. In the example, the number of identifiers n is 5. Initially,

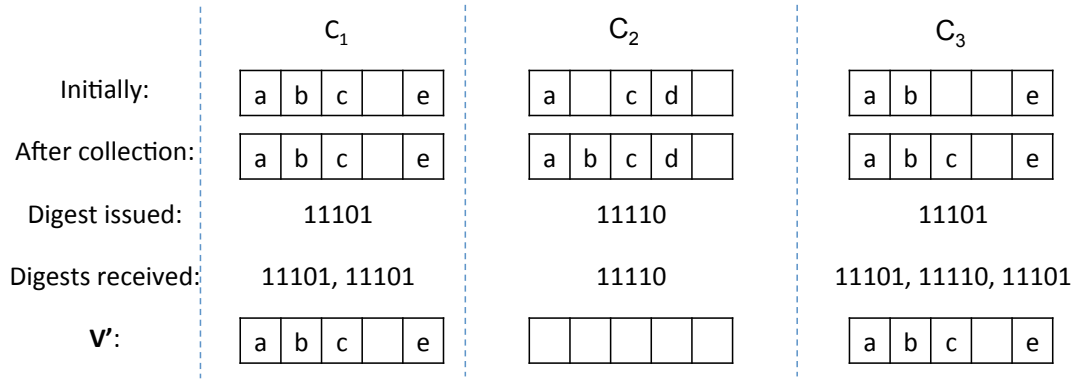


Figure 6.2 – Collection and voting phases of Quarts with three controller replicas

replica C_1 has the value of 4 messages, whereas replicas C_2 and C_3 only have the value of 3 messages. Notice that when a certain identifier is present in multiple replicas, it has the same value across these replicas.

Quarts outputs at each replica, within a bounded-latency, the following two variables: (1) a boolean flag $success$, indicating whether that replica successfully terminated, and (2) a vector $\mathbf{V}'$ containing the result of the vote if the $success$ flag is set to true. Quarts guarantees that across all replicas in which $success$ is true for a given label r, the vectors $\mathbf{V}'$ are exactly identical. For a subset of the identifiers, referred to as the *chosen* identifiers, $\mathbf{V}'$ contains the value of the message of that identifier, as present in some of the input vectors $\mathbf{V}$. The rest of the elements in $\mathbf{V}'$ are empty. In Figure 6.2, we see that replica C_2 does not terminate successfully, whereas replicas C_1 and C_3 terminate and agree on a vector containing 4 messages.

Quarts also provides the option to agree on a state variable $\mathcal{H}$ that can be included in the input with a state label r^-. In that case, Quarts additionally outputs the post-agreement state variable $\mathcal{H}'$ with its label r'. Across all replicas in which $success$ is true for a given label r, the variables $\mathcal{H}'$ and the labels r' are identical.

The Quarts protocol is composed of two phases: the *collection* phase and the *voting* phase. We present the design of each phase in what follows, and we refer back to the example in Figure 6.2 for illustration. Sections 6.4.1 and 6.4.2 present an overview of the design of these phases. Section 6.4.3 presents the modifications that can be performed during implementation, in order to optimize for the best case.

6.4.1 The Collection Phase

Collection is a bounded-latency phase in which each replica attempts to "collect", from other replicas, (1) the message values for any of its missing messages, and (2) the state variable corresponding to the most recent state, if it does not already have it. These are presented separately in Algorithms 6.1 and 6.2, respectively.

Algorithm 6.1: `collect_missing_messages`($\mathbf{V}$, r, T_{coll})

```
1  S ← identifiers of missing messages in V;
2  Send Query<S, r> to all replicas;        // request value of missing messages
3
4  repeat
5      if Query<Q, ℓ> received and ℓ = r then
6          // received query, send response
7          Send Response<V [Q \ S], r> to all replicas;
8      else if Response<P, ℓ> received and ℓ = r then
9          // received response, update set of messages
10         Update V to include P;
11         Remove received identifiers from S;
12     end
13 until timer T_coll expires;
14
15 return V;                                // return set of messages after collection
```

Collecting Messages

Algorithm 6.1 shows how each replica performs the collection phase for messages. It describes the function `collect_missing_messages` which takes as input a vector $\mathbf{V}$ of messages, a label r corresponding to the messages in $\mathbf{V}$, and an upper-bound T_{coll} on the execution time of the function.

The function first populates a set S with the identifiers that do not contain values in $\mathbf{V}$ (line 1). Then, it sends a *Query* to all other replicas, containing the elements of S and the label r (line 2). It then listens for messages from other replicas for a period of T_{coll} (line 13).

During that period, the replica might receive queries of the form *Query*<Q, ℓ>, where Q is the set of identifiers requested by some other replica, and ℓ is the label of the messages that replica is collecting (line 5). The function only responds if the labels ℓ and r are the same, in which case it sends a *Response*. This response contains the values of the identifiers in Q that this replica possesses (line 7).

The replica might also receive responses, either to its own query or to queries from other replicas, since all queries and responses are sent to all replicas (line 8). The response contains the value of some identifiers, in addition to a label. Again, the replica only accepts response that have a label equal to r, in which case it updates its vector $\mathbf{V}$ to include the values of the elements received (line 10) and removes the identifiers received from the set S (line 11). After the timer expires, the function returns the updated vector of messages $\mathbf{V}$ (line 15).

For example, in Figure 6.2, controller C_3 sends a query asking for the values of identifiers 3 and 4. We observe that it received a response from C_1 containing the value

Algorithm 6.2: `collect_missing_state`($\mathcal{H}$, r^-, T_{coll})

```
1  Send State<r⁻> to all replicas;          // advertise state label
2
3  repeat
4      if State<ℓ> received and ℓ < r⁻ then
5          // received state advertisement with smaller label, send my state
6          Send Update<H, r⁻> to all replicas;
7      else if Update<H', ℓ> received and ℓ > r⁻ then
8          // received state update with higher label, update my state
9          H ← H';
10         r⁻ ← ℓ;
11     end
12 until timer T_coll expires;
13
14 return H, r⁻;                            // return state and label after collection
```

of the 3rd identifier. In the example, either the query issued by C_3 was not received by C_2, or the response issued by C_2 was not received by C_3, since C_3 did not receive the value of the 4th identifier, present at C_2.

We note that message are always exchanged with all other replicas. This gossip-like communication scheme increases the chances of having a common vector of messages between all replicas at the end. It comes at the expense of an increased messaging cost, but as the number of replicas is usually not very high, the incurred messaging cost is only marginally higher than state-of-the-art protocols, as seen in Section 6.7.

Collecting State

Algorithm 6.2 presents the mechanism for collecting state. The `collect_missing_state` function takes as input the state variable $\mathcal{H}$, the state label r^-, and an upper-bound T_{coll} on the execution time of the function. The goal of this function is to retrieve, from the other replicas, the state variable with the highest state label. Of course, each replica must also help other replicas achieve their goal.

The replica first sends a *State* message, to all other replicas, containing its state label r^- (line 1). This serves to advertise, to the other replicas, the state label it has. Then, the replica listens for messages for a period of T_{coll} (line 12).

During that period, it might receive *State* messages from other replicas with state labels ℓ (line 4). If ℓ is smaller than r^-, then those replicas are stragglers, and must be re-synchronized. In that case, the replica sends an *Update* message containing its state variable $\mathcal{H}$, and its state label r^- to all other replicas (line 6).

If the replica receives an *Update* message with a state variable $\mathcal{H}'$ and a state label ℓ,

it checks whether ℓ is bigger than its own state label r^- (line 7). In that case, it updates its own state variable and state label (lines 9-10). After the timer expires, the function returns the updated state variable and state label (line 14).

6.4.2 The Voting Phase

After the collection phase, the replicas attempt to vote on (1) the subset of identifiers, and (2) the state variable, to be used in computation. The mechanism for performing this vote is presented in Algorithm 6.3. The `vote` function takes as input the vector of messages $\mathbf{V}$, the label of messages r, the state label r^-, and the upper bound on execution time T_{vote}.

Digests: Format, Priorities, & The Full Digest

The function begins by creating a *digest* out of $\mathbf{V}$ and r^- (line 1). A digest is an encoding of the set of identifiers, and the state label, present at this replica. It can be formed in several ways.

In our implementation, we encode the messages present as a bitstream of size n. For example, in Figure 6.2, C_1 creates a digest "11101", which represents that it has the value for messages with identifiers 1, 2, 3, and 5. State collection and voting is not shown in that example. We encode the state as follows: if r^- is 6 in the example above, then the digest would be "6.11101".

We assign priorities to digests in order to compare them. We say a digest $\mathcal{D}_1$ is larger than $\mathcal{D}_2$ if it has a higher priority. Thus, among a set of digests, the maximum is the one with the highest priority. The priority rules are given as follows. Digest $\mathcal{D}_1$ is larger than digest $\mathcal{D}_2$ if:

1. $\mathcal{D}_1$ has a higher state label than $\mathcal{D}_2$.
2. $\mathcal{D}_1$ and $\mathcal{D}_2$ have the same state label, but $\mathcal{D}_1$ contains more messages than $\mathcal{D}_2$.
3. $\mathcal{D}_1$ and $\mathcal{D}_2$ have the same state label and the same number of messages, but $\mathcal{D}_1$ has a message with a smaller identifier.

One example corresponding to each of the above conditions is listed in what follows: (1) "6.11001" > "5.11101", (2) "6.11101" > "6.10001", and (3) "6.11101" > "6.11011".

From the above definition, if we know that the most recent state label possible is ℓ, then the digest formed from ℓ and the set of all messages (all 1's in the bitstream) is the largest possible digest. Consider an example where we are agreeing on messages with label r in a round-based communication scheme. The maximum state label is

Algorithm 6.3: `vote`($\mathbf{V}$, r, r^-, T_{vote})

```
1  D_my ← create_digest(V, r⁻);          // create this replica's digest
2  Send Digest<D_my, r> to all replicas;
3  D ← [D_my, ..., ...];                 // vector of digests from each replica
4
5  D_mc;                                 // set of most common digests in D
6  D_sec;                                // set of second most common digests in D
7  f_mc;                                 // count of each element of D_mc in D
8  f_sec;                                // count of each element of D_sec in D
9  f_0;                                  // number of empty cells in D
10
11 repeat
12     // receive collection message
13     if Digest<D, ℓ> received from replica j and ℓ = r then
14         D[j] ← D;
15         Update D_mc, D_sec, f_mc, f_sec, f_0 using D;
16         // attempt vote
17         if f_0 = 0 then
18             // all digests received, pick max of D_mc
19             return True, max(D_mc);
20         else if |D_mc| = 1 and f_mc > f_sec + f_0 then
21             // only one most common digest, and clearly the majority
22             return True, D_mc[0];
23         else if |D_mc| = 1 and f_mc = f_sec + f_0 and f_sec ≠ 0 then
24             // second most common digests could have equal count
25             if D_mc[0] > max(D_sec) then
26                 // most common digest is the largest
27                 return True, D_mc[0];
28             end
29         else if |D_mc| = 1 and f_mc = f_sec + f_0 then
30             // other digests could have equal count
31             if D_mc[0] = full_digest then
32                 // most common digest is the largest
33                 return True, D_mc[0];
34             end
35         end
36     end
37 until timer T_vote expires;
38
39 return False, NULL;                   // return false because vote was unsuccessful
```

the one corresponding to the previous control round $\ell = r - 1$. Therefore, the digest composed of "ℓ.11111" is the largest possible digest any replica could have. We call this the `full_digest`. We see its significance in voting in the next subsection.

Voting

After the digest is created (Algorithm 6.3, line 1), it is sent to all replicas along with the message label r (line 2). Each replica maintains a vector $\mathbf{D}$ of size g (where g is the number of replicas). This vector contains the digest received from each replica. It is initialized to contain the digest created by this replica (line 3).

Additionally, each replica maintains the following 5 variables (lines 5-9): (1) a set D_{mc} containing the most common digests in $\mathbf{D}$, *i.e.,* the set of digests that appear most frequently, (2) a set D_{sec} containing the second most common digest in $\mathbf{D}$, (3) an integer f_{mc} representing the count (frequency) in $\mathbf{D}$ of the digests from D_{mc}, (4) an integer f_{sec} representing the count in $\mathbf{D}$ of the digests from D_{sec}, and (5) and integer f_0 representing the number of empty cells in $\mathbf{D}$, *i.e.,* the number of replicas from which we have not received a digest.

Upon reception of a digest $\mathcal{D}$ with a label $\ell = r$, the digest is inserted into $\mathbf{D}$. Then, the five aforementioned variables are updated according to the changes in $\mathbf{D}$. That is, the most common digests are placed in D_{mc}, their count in f_{mc}, and so on.

Afterwards, the replica attempts to select a digest from $\mathbf{D}$. A digest will only be selected if it is sure that no other replica will select a different digest. Otherwise, once the timer expires (line 37), the $success$ flag is returned as false (line 39).

The vote is successful in one of the following 4 conditions:

1. If a digest from all the replicas is received, *i.e.,* $f_0 = 0$ (lines 17-19), then out of the most common digests, the maximum is chosen. That is, first we check for the digest that appears most frequently, *i.e.,* that has plurality or relative-majority, then if there are multiple of those, we choose the one that has the highest priority among them.

2. In lines 20-22, If the set of most common digests has a cardinality of 1, then only one digest ($\mathcal{D}$) appears most frequently. In that case, we compare its count f_{mc} to the count f_{sec} of the second most common digest. If there are not enough empty cells in $\mathbf{D}$ to make one of the second most common digests as common as $\mathcal{D}$, then $\mathcal{D}$ is returned.

3. Alternatively, in the previous case, if there are enough empty elements to make the second most common digest appear just as frequently as $\mathcal{D}$, then $\mathcal{D}$ can only

be chosen if it is larger than all the digests in D_{sec} (lines 23-28). This assumes that D_{sec} is not empty, *i.e.*, that $f_{sec} > 0$.

4. In the previous case, if $f_{sec} = 0$, then we do not know what other digests might be received and appear just as frequently as $\mathcal{D}$. We are only sure that $\mathcal{D}$ has a higher priority than any of those digests if $\mathcal{D}$ is actually the `full_digest`. In that case, it is selected (lines 29-35).

In the example shown in Figure 6.2, replica C_3 receives all the digests, thus using condition #1, selects the most common digest that appears ("11101"). C_1 only receives the digest from C_3, in addition to its own digest. However, condition #3 applies in this case, as these digests are similar, and $g = 3$. Therefore, the digest will remain the most common digest, and can be returned. Replica C_2 does not receive digests other than its own, hence does not successfully return.

We note that in addition to the $success$ flag, the function in Algorithm 6.3 returns the digest, rather than the vector of messages and the state label. However, as we describe in the next subsection, the latter results can be reconstructed from the digest.

6.4.3 Optimization for the Best Case

The functions for collection and voting presented in Section 6.4.1 only return after the timer expires. That means that their actual execution time is always equal to the upper bound T_{coll}. However, these functions could return as soon as they have the information they require. The `collect_missing_messages` function could return as soon as it receives the values of the messages of all identifiers. Similarly, if the most recent state label is known in advance, the function `collect_missing_state` could return once it receives the state corresponding to that state label.

However, these functions also serve the purpose of helping other replicas collect messages and state. Therefore, that part must continue executing until the other replicas finish their operation. Algorithm 6.4 presents an implementation-view of Quarts collection and voting. It is optimized for the best-case scenario, which occurs often in CPSs with low fault-rates. In the best case, the replica receives all messages and already has the most recent internal state. Notice that in that case, the function incurs minimal latency-overhead, as it incurs zero overhead for collection.

The `quarts` function takes as input the following: (1) a vector $\mathbf{V}$ of messages, (2) a state variable $\mathcal{H}$, (3) a label r corresponding to the messages, (4) a label r^- corresponding to the state variable, (5) a label r^* representing the highest possible state label, and (6) T_{coll} and (7) T_{vote} representing upper-bounds on execution time.

r^* is the highest possible state label that exists at all replicas when r is the message

Algorithm 6.4: `quarts`($\mathbf{V}$, $\mathcal{H}$, r, r^-, r^*, T_{coll}, T_{vote})

```
1  S ← identifiers of missing messages in V;
2  Send Query<S, r> to all replicas;          // request value of missing messages
3  Send State<r⁻> to all replicas;            // advertise state label
4
5  // threads can continue running even if function returns
6  Call a thread to respond to queries until T_coll + T_vote;
7  Call a thread to respond to state advertisements until T_coll + T_vote;
8
9  repeat
10     if Response<P, ℓ> received and ℓ = r then
11         // collect missing messages
12         Update V to include P;
13         Remove received identifiers from S;
14     else if Update<H', ℓ> received and ℓ > r⁻ then
15         // collect missing state
16         H ← H';
17         r⁻ ← ℓ;
18     end
19 until timer T_coll expires or all messages received and r⁻ = r* ;
20
21 success, D ← vote(V, r, r⁻, T_vote); // call the vote function
22
23 if success and V contains all the messages in D and r⁻ is the state label in D then
24     return True, V[D], H;
25 else
26     return False, NULL, NULL;
27 end
```

label. In a round-based communication scheme, $r^* = r-1$, as the latest state is the one the controller has after computing in the previous round. In Section 6.5, we discuss how to compute r^* when the labels are timestamps, as in the case for SEs.

In lines 1-3, the function sends a *Query* for missing messages and a *State* advertisement for collecting state. This is as in Algorithms 6.1, 6.2. Then, it calls two threads, which take care of responding to queries and state advertisements from other replicas, as in the aforementioned algorithms (lines 6-7). These threads can continue executing even after the `quarts` function returns. However, we bound their execution to $T_{coll} + T_{vote}$, which is when other replicas would have finished executing.

Lines 9-19 perform the collection of missing messages and state. Notice from line 19 that this returns as soon as all the messages are present and the state received is the most recent state. Therefore, this step incurs zero latency-overhead if the replica already has all this information. This is optimized for the best-case scenario when no faults occur.

Afterwards, the `vote` function is called (described in Algorithm 6.3). This function returns the $success$ flag and the digest $\mathcal{D}$. The replica can compute only if all the following three conditions hold: (1) $success$ is set to true, indicating that this replica has successfully voted on a digest, (2) its vector of messages $\mathbf{V}$ contains values for all the identifiers that are included in the digest $\mathcal{D}$, and (3) its state label is that represented in $\mathcal{D}$. If all these conditions hold, then the function returns the vector of messages, only including the chosen identifiers, and the state variable (line 24). Otherwise, it returns a false flag, indicating that the replica cannot compute in this round (line 24).

6.4.4 The Two-Replica Case

For any number of controller replicas g, Quarts guarantees consistency in a bounded latency-overhead, as shown in Section 6.6. Quarts also outperforms state-of-the-art consensus protocols in terms of both latency and availability, as shown in Section 6.7. Additionally, when the number of controller replicas is two ($g = 2$), Quarts incurs zero latency-overhead in the majority of the scenarios. We illustrate this case in this subsection.

Recall from Section 6.4.3 that a replica incurs zero latency-overhead in collection, in rounds when it contains all the most recent information. In a typical CPS deployment, the fault rate in software agents and the network loss rate are both low. Therefore, in most computation rounds, replicas contain the most recent state and have the values of all messages. In addition to the zero-latency collection incurred in this case, we show that Quarts incurs zero latency-overhead when voting in a two-replica CPS.

We consider a two-replica CPS, where one replica has the most recent state and the values of all messages. This replica skips collection as mentioned earlier, and calls the `vote` function (Algorithm 6.3). During voting, it starts by creating a digest, which in this case would be the `full_digest`, described in Section 6.4.2.

Notice that case #4 immediately applies (lines 29-35). The set of most common digests D_{mc} contains only the digest of this replica. The frequency of that digest is $f_{mc} = 1$. No second most common digest exists, so $f_{sec} = 0$, and the number of empty cells is $f_0 = 1$, as only the other replica (in a two-replica system) has not sent its digest yet. Therefore, no matter what digest the other replica sends, it will never be more frequent that the digest created by this replica. Furthermore, the digest created by this replica is the `full_digest`, *i.e.,* it has the highest priority out of any digest created in round r. Thus, voting incurs zero latency-overhead as well.

In conclusion, a controller replica, in a two-replica CPS, containing all the most recent information, can begin computation immediately, without waiting for communication from the other replica, and without the risk of violating consistency.

6.5 Applying Quarts to CPS Controllers

In this section, we discuss how Quarts can be applied to CPS controllers in order to guarantee reliable consistency. First, we discuss how Quarts can be applied to GAs in Section 6.5.1. Then, we discuss the additional mechanisms required to apply Quarts to SEs in Section 6.5.2.

6.5.1 Quarts in Grid Agents

In order to apply Quarts to GAs, we need to know when to instantiate Quarts, and what parameters to instantiate Quarts with. As mentioned in Section 6.1, Quarts must first be used to agree on which set of advertisements to be used in a computation. Recall, from Chapter 4, that in a computation for round r, advertisements from round $r-1$ must be used, if available. For RAs from which the GA does not have an advertisement from round $r-1$, it must use the most recent advertisement available. This can be captured in the $\mathcal{H}$ state variable, which we assume here to hold the latest agreed upon advertisement from each resource. With this in mind, a GA replica computing in round r must select, for each RA, the advertisement from the same round as other replicas.

After agreeing on the advertisements and the state variable, the GAs must choose an SE state to use in computation. This choice must be consistent among GA replicas computing in this round. There are several methods to choose such the SE state, but we consider an additional requirement: the timestamp of the SE state chosen must be greater than the timestamps of the advertisements chosen. That is, we consider that the SE state must reflect the same state of the grid encapsulated by the advertisements used in the computation. For this step, we assume that the SE replicas send consistent SE state to the GAs, *i.e.,* SE states with the same timestamp, received at different GA replicas, have the same value. This is guaranteed by applying Quarts to SEs, as described in Section 6.5.2.

Algorithm 6.5 describes the mechanism of applying Quarts to GAs. This algorithm extends Algorithm 5.1 with the lines in red. The GA maintains 3 additional variables: (1) a vector $\mathbf{V}$ of advertisements received from RAs in the latest round (line 1), (2) an integer r corresponding to the last round number in which Quarts agreement was performed (line 7), and (3) an integer r^- corresponding to the last round number in which a computation was performed (line 8). r^-, therefore, represents the state label.

When to Instantiate Quarts: Accounting for Jitter

As mentioned in Section 6.4, Quarts must be passed a vector of messages as one of the arguments. However, the GA receives each advertisement separately. Therefore, it

Algorithm 6.5: Abstract model of a GA with Quarts. The parts in red are added to Algorithm 5.1 (Algorithm is continued on next page)

```
1  V ← [];                                   // vector of advertisements received from RAs
2  S ← ∅;                                    // set of states received from the SE
3  Z_A ← [ ];                                // vector of advertisements used in a computation
4  T_v ← [0, 0, ...., 0];                    // vector of validity times
5  H ← ∅;                                    // internal state of the GA
6  C ← 0;                                    // logical clock on this GA
7  r ← −1;                                   // last round in which Quarts was performed
8  r⁻ ← −1;                                  // last round in which we computed
9
10 on initialization
11 |  X ← ∅;                                 // initialize vector of setpoints to probes
12 |  issue(X, T_v, C);                      // send probes to the RAs
13 end;
14
15 on reception of an advertisement adv with label ℓ from an RA i
16 |  if ℓ > r then
17 |  |  Start timer for δ_n;                // reset and start the jitter timer
18 |  |  V ← [];                             // reset vector of advertisements
19 |  end
20 |  if ℓ ≥ r then
21 |  |  V[i] ← {(adv, ℓ)};                 // update vector of advertisements
22 |  end
23 |  C ← max(C, ℓ);
24 end;
25
26 on reception of a state st with timestamp T from the SE
27 |  S ← S ∪ {(st, T)};                    // aggregate received states
28 end;
```

must decide when to stop waiting for advertisements in a single control round, and instantiate a Quarts instance.

Recall, from Chapter 3, that the upper bound on one-way network latency for messages that are not lost is δ_n. Hence, the maximum time difference (or jitter) between receiving the first and last advertisement in a given control round is δ_n. In the algorithm, when an advertisement from a new round is received (line 16), a timer is started that will fire after δ_n (line 17). The vector $\mathbf{V}$ of advertisements in the latest control round is also reset, as a new control round has started (line 18). Until the timer fires, advertisements from the same control round are added to $\mathbf{V}$ (line 21).

In line 31, we see the conditions under which Quarts is instantiated. First, the largest label of advertisement received (C) must be larger than r. This is to avoid instantiating Quarts with the same label multiple times. If that holds, then Quarts is

```
29 repeat
30     success ← False;
31     if C > r and (timer expired or all advertisements received) then
32         r ← C;                                   // performing Quarts, update r
33         success, V′, H ← quarts(V, H, r, r⁻, r* = r, T_coll = 3δ_n, T_vote = 5δ_n)
34     end
35
36     if success then
37         A_Q, T_v ← select_advertisements(V′, H);
38         S_Q ← select_state(V′, S);
39         T ← current time;                        // get the current time
40         ready, timeout, Z_A, st ← ready_to_compute(A_Q, S_Q, T);
41
42         if ready then
43             C ← C + 1;
44             X, H ← compute_setpoints(Z_A, st, timeout, H);
45             if timeout then
46                 T_v ← [0, 0, ...., 0];           // reset to all-zeros
47             end
48             issue(X, T_v, C);                    // send X to the RAs
49         end
50
51         H ← update(H, A_Q);
52         r⁻ ← r;
53     end
54 forever;
```

instantiated either when the timer fires, or when all advertisements from that round are received, whichever comes first. This enables replicas that receive all advertisements to begin agreement immediately, whereas replicas with some missing advertisements wait for the remaining jitter.

Assigning Values for T_{col} and T_{vote}

Quarts requires two parameters, T_{col} and T_{vote}, representing the upper bound on the execution time of the collection and voting phases, respectively. The value of T_{col} depends on how long a controller must wait before it is sure it will not receive any responses or state updates from its replicas. Recall that the collection phase involves sending a query and state advertisement, followed by reception of responses and state updates. This message exchange involves one round-trip time (RTT). Additionally, replicas wait for a jitter of δ_n before instantiating Quarts. Hence, $T_{col} = 3\delta_n$.

Similarly, T_{vote} represents the maximum time a replica must wait for digests from other replicas, before declaring an unsuccessful vote. Again, we must account for the

slowest replica, which started Quarts after the timer fired (δ_n), and required the full T_{col} in collection phase. In contrast, the fastest replica immediately started Quarts and skipped the collection phase. The time taken for a digest to be received is upper-bounded by δ_n. Therefore, adding these terms, the fastest replica must wait a maximum of $T_{vote} = 5\delta_n$, before giving up on a successful vote.

The Most Recent State r^*

Quarts requires a parameter, r^*, representing the label of the most recent state. When agreeing on advertisements with label r (for the computation of setpoints in round $r+1$), the most recent state belongs to the replica that successfully finished Quarts in round r, *i.e.,* before issuing setpoints with label r. These replicas do not need to collect state, as they have the most recent one. Therefore, $r^* = r$.

Choosing a Consistent Set of Advertisements

The `quarts` function on line 33 returns the $success$ flag, a vector $\mathbf{V}'$ of advertisements, and an updated state variable $\mathcal{H}$. Recall from Section 6.4 that in a given round, across all replicas in which the $success$ flag is set to true, $\mathbf{V}'$ and $\mathcal{H}$ are consistent. This is formally proven in Section 6.6.

The GA must then choose a consistent set of advertisements to perform the computation with. This is done via the `select_advertisements` function (line 37). In a manner similar to the `choose_advertisements` function in Algorithm 5.1, this function populates $\mathcal{A}_Q$ with the advertisements in $\mathbf{V}'$, and for identifiers not present in $\mathbf{V}'$, the latest advertisement in $\mathcal{H}$, if any, is used as a long-term advertisement in $\mathcal{A}_Q$.

As $\mathbf{V}'$ and $\mathcal{H}$ are common across all replicas with a true $success$ flag in a given round, $\mathcal{A}_Q$ is also guaranteed to be common.

Choosing a Consistent SE State

In addition to choosing a consistent set of advertisements and a consistent state variable, the GA must choose a consistent SE state to use in computation. This is done via the `choose_state` function on line 38, which is described in Algorithm 6.6.

This function makes use of Property 6.5 and Property 6.6, described in Section 6.3. The RAs and the asynchronous sensors are considered to be time-synchronized, and the interval between two measuring instances at an asynchronous sensor is lower bounded by σ. This in turn means that the interval between any two SE state timestamps is lower bounded by σ, as the SE state timestamp is derived from the asynchronous sensors timestamps, as described in Algorithm 3.2.

Algorithm 6.6: `select_state`($\mathbf{V}'$, $\mathcal{S}$) function in the GA

```
1  T_max ← maximum timestamp of advertisements in V';
2  T_last ← maximum timestamp of states in S smaller than T_max;
3
4  I_1 ← [T_max, T_last + 2σ);          // first interval
5  I_2 ← [T_max, T_max + σ);            // second interval
6
7  if there exists st ∈ S such that st.T ∈ I_1 ∪ I_2 then
8  |  return {st};
9  else
10 |  return ∅;
11 end
```

The `choose_state` function (Algorithm 6.6), checks for an SE state with a timestamp in a given interval. As shown in Lemma 6.5, the interval chosen is guaranteed to have at most one element. Furthermore, if an SE state is found with a timestamp in that interval, it is guaranteed to be the same across all replicas. The function returns $\emptyset$ if the replica does not have the required SE state.

Computing Consistent Setpoints

After choosing a set of advertisements, a state variable, and an SE state, all consistently among all replicas, a GA calls the `ready_to_compute` function, and subsequently the `compute` function. As these functions are passed the same parameters across all replicas in a given control round, they will return the same value for all replicas. That is, all replicas that issue setpoints, will issue the same setpoints, whether these are implementation or probe setpoints.

After the computation, the state variable and the state label are updated. With this update, the state variable $\mathcal{H}$ includes the latest agreed upon advertisement from each RA. Only adding the agreed upon advertisements $\mathcal{H}$ enables subsequent control rounds to remain consistent as well.

Note that we consider that replicas that do not possess the chosen SE state do not compute in the corresponding control round, as the `ready_to_compute` function would return a false $ready$ flag. An alternative would be to perform another Quarts instance, attempting a collection phase to receive the required SE state, and subsequently voting on whether or not to use it. However, in our simulations (Section 6.7), we do not perform this additional step. The alternatives are a trade-off between availability and additional latency-overhead.

6.5.2 Quarts in State Estimators

Similar to the mechanism shown in Section 6.5.1 applying Quarts to GAs, Quarts can be applied to SEs. SEs must agree on the measurements used in computation, in addition to the state variable. The measurements, however, are timestamped, rather than labeled using logical clocks. With time-alignment, however, the measurements can be grouped into rounds [82].

The same arguments apply, as in Section 6.5.1. First, the SE must account for a jitter of δ_n. Second, the values of T_{col} and T_{vote} are the same — $3\delta_n$ and $5\delta_n$, respectively.

The main difference is the value of r^*, representing the label of the latest state in which the SE computed. Given that t represents the timestamp of the measurements passed to Quarts, and given that σ is a lower bound on the interval between two measurements at an asynchronous sensor, we conclude that the last computation performed by an SE was with a timestamp $T < t - \sigma$. Also, for any timestamp T such that, $t - 2\sigma < T < t - \sigma$, if a state label with a value of T exists, then T represents the latest state label before t. The proof is similar to the proof of Lemma 6.5.

Therefore, we pass $r^* = t - 2\sigma + \epsilon$ to Quarts, where ϵ represents an infinitesimally small number. However, we modify Algorithm 6.4 line 19, to end collection when $r^- \geq r^*$. This covers both the case of GA agreement, where r^* is an integer exactly equal to the latest possible label, and the case of the SE described here.

As the SE replicas agree on the measurements used for computation of an SE state of a given timestamp, multiple GA replicas choosing SE states with the same timestamp are guaranteed to choose the same SE state value, as shown in the next section.

6.6 Formal Guarantees

In this section, we prove that applying Quarts to CPSs guarantees reliable consistency (Definition 3.10). That is, we show that setpoints issued to the same RAs in the same round have the same value. Additionally, we prove that Quarts incurs a bounded latency-overhead.

Theorem 6.1 (Reliable Consistency)**.** *A CPS that implements Quarts, as in Algorithms 6.4-6.6, guarantees reliable consistency.*

Proof. From Algorithm 6.5, we see that the setpoints are computed in a given control round r, only if the $success$ flag returned by Quarts is true (line 36).
The computation in that case depends entirely on the values of $\mathbf{Z}_{\mathcal{A}}$, st, and $\mathcal{H}$ (line 44).
These values, in turn, depend on the values of $\mathcal{A}_Q$ and $\mathcal{S}_Q$ (lines 37-38).
The proof then follows directly from the results of Lemmas 6.1, 6.2, 6.3, 6.4, 6.5. □

Lemma 6.1. *For any two GA replicas GA_1 and GA_2, in which the function* `quarts` *(Algorithm 6.5 line 33) returns, respectively, ($success_1$, $\mathbf{V}'_1$, $\mathcal{H}_1$) and ($success_2$, $\mathbf{V}'_2$, $\mathcal{H}_2$), when called in a given round r, the following holds:*

$$(success_1 = \textit{true}) \textit{ and } (success_2 = \textit{true}) \implies \mathbf{V}'_1 = \mathbf{V}'_2$$

Proof. If $success_1$ in GA_1 is true, in a given round r, then the function `quarts`, implemented as in Algorithm 6.4, must have the following conditions:
(1) The `vote` function returned true (line 21) and
(2) the `vote` function returned a digest $\mathcal{D}$, and $\mathbf{V}$ contains advertisement values for all the identifiers in $\mathcal{D}$ (line 24).
The same applies to GA_2 in round r. The return vector $\mathbf{V}'$ contains the value of the advertisements in $\mathbf{V}$ that are also in $\mathcal{D}$.
$\mathbf{V}$ contains advertisements received from RAs in round r.
Therefore, if both replicas have an advertisement from a given RA in $\mathbf{V}$, then the values of the advertisement are the same.
Therefore, it suffices to show that $\mathcal{D}$ is the same across both replicas.

If the `vote` returns true in round r, then a digest $\mathcal{D}$ is selected only if: (1) a relative majority of the replicas proposes $\mathcal{D}$, and
(2) in case of ties, a static priority is used.
This would be trivial to show if the voter at each replica waited to receive all the other digests before selecting the result.
As Quarts is only called once in round r at each replica, this would guarantee that $\mathcal{D}$ is the same across all replicas.

Condition #1 in the voter waits for all digests to be received, then picks the one with the highest priority among the digests with relative majority (lines 17-19).
The other conditions return prematurely.
We show that they only return if, no matter what other digests are received, condition #1 will hold for the return value $\mathcal{D}$.

Condition #2 (lines 20-22) returns if there is only one digest that is most common among the received ones, and no matter what the remaining replicas send, this digest will remain the most common, with no ties.
That digest is guaranteed to have relative majority, with no ties, when condition #1 is reached.

Condition #3 (lines 23-28) returns if
(1) there is only one most common digest among the received ones,
(2) no matter what the remaining replicas send, the other received digests can only become equal in frequency to the currently most common, and
(3) the current most common has a higher priority than all the other digests that might

reach its frequency. That digest is guaranteed to have relative majority when condition #1 is reached.
It might have ties with some of the currently received digests, but it has higher priority than all of them.

Condition #4 (lines 29-35) returns if
(1) there is only one digest type $\mathcal{D}$ in $\mathbf{D}$,
(2) that digest will remain the relative majority even if all remaining digests have the same value, different than $\mathcal{D}$, and
(3) $\mathcal{D}$ is the `full_digest`.
This is similar to condition #3, except the other digests that might have the same frequency as $\mathcal{D}$ are not seen yet.
Therefore, the replica can only be sure that $\mathcal{D}$ has the highest priority if it is the `full_digest`.
The `full_digest` always has the highest priority. □

Lemma 6.2. *For any two GA replicas GA_1 and GA_2, in which the function `quarts` (Algorithm 6.5 line 33) returns, respectively, $(success_1,\ \mathbf{V}'_1,\ \mathcal{H}_1)$ and $(success_2,\ \mathbf{V}'_2,\ \mathcal{H}_2)$, when called in a given round r, the following holds:*

$$(success_1 = \textit{true}) \textit{ and } (success_2 = \textit{true}) \implies \mathcal{H}'_1 = \mathcal{H}'_2$$

Proof. From Lemma 6.1, we know that the digests, returned by the `vote` function, is the same in both replicas. From Section 6.4.2, we see that if two digest are equal, then they have the same state labels.
From Algorithm 6.4, we see that the state variable $\mathcal{H}$ is updated according to messages received from other replicas.
Therefore, it suffices to show that when two replicas pass $\mathcal{H}_1$ and $\mathcal{H}_2$ to the `quarts` function, with the same state label r^-, then $\mathcal{H}_1 = \mathcal{H}_2$.

We prove this by strong induction on r, the control round in which Quarts is instantiated.
Base Case: When $r = 0$, $r_i^- = r_j^- = -1$, then the states used for computing the first setpoint are $\mathcal{H}_1^r = \mathcal{H}_2^r = \emptyset$. Thus, the statement holds for $r = 0$.

Induction Hypothesis: Let the statement hold of the labels in $[0, \ell]$, where $\ell > 0$.
Thus, $\forall\ r \in [0, \ell - 1],\ i,\ j,\ r_1^- = r_2^- \implies \mathcal{H}_1^r = \mathcal{H}_2^r$
This means that, if both replicas computed setpoints for a label $r \in [0, \ell - 1]$ and had the same state label, they had the same state.

Inductive Step: To show that, for label ℓ, if both replicas compute setpoints in round ℓ then, $r_1^- = r_2^- \implies \mathcal{H}_1^\ell = \mathcal{H}_2^\ell$.

When a replica computes a setpoint, its state is updated by lines 44, 51 of Algorithm 6.5.
As both replicas have the same state label $r^- \leq \ell - 1$, then they computed in the same previous control round, and had their state labels updated (line 52).
From the induction hypothesis, the state variable used in the computation of that round is the same.
From Lemma 6.1, the vector of advertisements used in that computation is the same.
From Lemma 6.5, the SE state used in that computation is the same.
Therefore, the updated state in line 44 is the same.
Moreover, in line 51, the state variable is updated using the set of chosen advertisements, which is also the same as shown in Lemma 6.4.
Therefore, we have $\mathcal{H}_1^\ell = \mathcal{H}_2^\ell$ □

Lemma 6.3. *For any two SE replicas SE_1 and SE_2, that issue an SE state in round r, the value of the SE state is the same.*

Proof. The proof follows the same reasoning as Lemmas 6.1, 6.2.
The SE replicas agree on what measurements and state variable to use in the computation of a given SE state.
The resulting values computed are the same. □

Lemma 6.4. *If the function* `select_advertisements` *(Algorithm 6.5 line 37) returns two sets $\mathcal{A}_{Q1}$ and $\mathcal{A}_{Q2}$, at two GA replicas, given the same input parameters $\mathbf{V}'$ and $\mathcal{H}$, then $\mathcal{A}_{Q1}$ and $\mathcal{A}_{Q2}$ have the same value.*

Proof. The function is not described explicitly, but is very similar to Algorithm 5.2.
The advertisements in $\mathbf{V}'$ are placed in $\mathcal{A}_Q$, along with the short-term validity time (check Chapter 5).
The rest of the RAs, with no advertisements in $\mathbf{V}'$, are chosen from $\mathcal{H}$, such that the advertisement with the highest label is chosen for each RA.
Given that $\mathbf{V}'$ and $\mathcal{H}$ are the same, the resulting $\mathcal{A}_Q$ is guaranteed to be the same. □

Lemma 6.5. *If the function* `select_state` *(Algorithm 6.5 line 38) returns two non-empty sets $\mathcal{S}_{Q1}$ and $\mathcal{S}_{Q2}$, at two GA replicas, given the same input parameter $\mathbf{V}'$, then $\mathcal{S}_{Q1}$ and $\mathcal{S}_{Q2}$ have the same value, regardless of the input parameter $\mathcal{S}$ at the different replicas.*

Proof. The function `select_state` is implemented in Algorithm 6.6.
As $\mathbf{V}'$ is the same for both replicas, T_{max} chosen in line 1, is also the same.
T_{max} represents the maximum timestamp of an advertisement in $\mathbf{V}'$.

Due to message losses, $\mathcal{S}$ might contain different SE states in both replicas.
However, from Lemma 6.3, if two SE states, across the two replicas, have the same timestamp, then they have the same value.

We show that if the function returns a non-empty set, then it returns a state with the same timestamp in both replicas.
The SE state chosen has a timestamp that belongs in the union of two intervals:
(1) $I_1 \leftarrow [T_{max}, T_{last} + 2\sigma)$ (line 4), and
(2) $I_2 \leftarrow [T_{max}, T_{max} + \sigma)$ (line 5),
where σ represents the minimum interval between timestamps of SE state issue by the SEs, and T_{last} represents the highest timestamp in $\mathcal{S}$ that is smaller than T_{max}.
Note that T_{last} might be different in different replicas.
However, we show that there exists at most one timestamp in $I_1 \cup I_2$ for which an SE state has been issued.
Therefore, if both replicas find an element in $I_1 \cup I_2$, it is the same element.

We prove this by contradiction.
Let $t_1, t_2 \in I_1 \cup I_2 = [T_{max}, \max(T_{max} + \sigma, T_{last} + 2\sigma))$.
$t_1 \geq T_{max}$.
As σ is the minimum interval between two timestamps, $t_2 \geq t_1 + \sigma$.
Thus, $t_2 \geq T_{max} + \sigma$.
Also, as T_{last} is also a timestamp of an SE state, and $t_1 > T_{last}$, then $t_1 \geq T_{last} + \sigma$.
Thus, $t_2 \geq T_{last} + 2\sigma$.
Hence, $t_2 \geq \max(T_{max} + \sigma, T_{last} + 2\sigma)$.
Therefore, $t_2 \notin I_1 \cup I_2$. □

A real-time CPS aims to have minimal latency between the generation of advertisements and issuing of setpoints. Hence, agreement protocols for CPSs must have a low latency-overhead. The latency overhead of a replica due to Quarts is the execution time in the `quarts` function (Algorithm 6.5, line 33). Other functions used by Quarts incur negligible latency, as they perform simple tasks without network communication. The `quarts` function is shown in Algorithm 6.4. The execution time of this function depends on the upper-bound of the one-way network latency δ_n, as discussed in Section 6.5.

Theorem 6.2 (Bounded Latency-Overhead)**.** *If the* `quarts` *function (Algorithm 6.4) returns true, its execution time is upper-bounded by* $5\delta_n$.

Proof. We consider the viewpoint of the slowest replica.
The slowest replica calls the `quarts` function after the timer fires, incurring a delay of δ_n over the fastest replica to start Quarts.
The slowest replica, then, requires $T_{coll} = 3\delta_n$ to finish the collection phase.
Afterwards, that replica sends its digest.
The digest will be received at other replicas within δ_n. Therefore, the latest digest received in a given round r, is after $5\delta_n$ from when the fastest replica started Quarts.
If the `vote` function does not successfully return after $5\delta_n$, it never will.
Hence, the upper bound on the execution time of `quarts`, if it returns true, is $5\delta_n$. □

6.7 Simulation Results

We perform discrete-event simulations to study the performance of Quarts, and to compare it to state-of-the-art protocols for agreement [104, 124]. The performance metrics used for comparison are consistency, availability, latency, and messaging cost. These are defined in Section 6.7.1. The protocols we compare are Quarts, Fast Paxos, and two flavors of passive replication. These are described in Section 6.7.2. We define our simulation methodology in Section 6.7.3, and present our results in Section 6.7.4.

6.7.1 Performance Metrics

From Definition 3.10, reliable consistency is said to hold in a round r, if no two setpoints to the same RA in round r have different values. We obtain a measure of consistency as follows. If reliable consistency holds in a round r, then we say $\gamma_r = 1$, otherwise $\gamma_r = 0$. As consistency is a safety property, if no setpoints are issued in a given round r, then consistency is not violated. In those cases, $\gamma_r = 1$. The consistency metric of a CPS execution is thus given by $\Gamma = \mathbb{E}[\gamma_r]$.

Availability is defined per RA in a given round. If an RA j receives a setpoint in round r, we say $\psi_r^j = 1$. Otherwise, $\psi_r^j = 0$. We measure the availability in a given round as the fraction of RAs that receive setpoints: $\psi_r = \frac{1}{n}\sum_{j=1}^{n} \psi_r^j$. The overall availability metric of a CPS execution is given by $\Psi = \mathbb{E}[\psi_r]$.

The latency of a CPS in a given round r is the duration from the time an RA first issued an advertisement in round $r-1$, until the time a GA replica first issued a setpoint in round r. Formally, we define $S_r^j, 1 \leq j \leq n$, as the time instant at which RA j issued an advertisement in round r, and $I_r^k,\ 1 \leq k \leq g$, as the time instant at which the GA GA_k issues the setpoints in round r. The latency in round r is defined as $\delta_r = \min_{k\in[1,g]} I_r^k - \min_{j\in[1,n]} S_{r-1}^j$. We consider two latency metrics in our analysis: (1) the mean latency, computed as $\Delta = \mathbb{E}[\delta_r]$, and (2) the 99^{th} percentile of latency (δ_{p99}), computed from the ECDF.

Finally, we consider the messaging cost of a protocol as the number of messages exchanged among the controller replicas and between the replicas and RAs. The messaging cost in a given round r (ω_r) is the number of messages exchanged that carry the label r. We consider the mean messaging cost (Ω) and the 99^{th} percentile (ω_{p99}) in our analysis.

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

6.7.2 Agreement Protocols

As mentioned in Section 5.2, we simulate, in addition to Quarts (Q), three protocols: Fast Paxos (FP), passive replication with hot standbys (PH), and passive replication with cold standbys (PC).

- **Quarts (Q):** Quarts is simulated as described in Sections 6.4, 6.5. Quarts guarantees reliable consistency, so the consistency metric will always be 1.
- **Fast Paxos (FP):** This is an active-replication protocol in which all replicas receive input, perform computations, then agree on which replica issues the output setpoints in a given round. This form of agreement is equivalent to agreeing on which setpoints to issue. Fast Paxos [104] is used to perform the agreement. It guarantees consistency, and is optimized for low latency.
- **Passive Hot (PH):** Passive replication with hot standbys is similar to active replication in that all replicas receive input and compute setpoints. However, one replica is designated as the *primary*, and only the primary issues setpoints. The other replicas serve as *standbys*. The primary sends regular heartbeat messages to the standbys. The absence of heartbeats for several rounds signals that the primary is faulty, in which case the standbys elect a new primary among themselves. Heartbeats also serve to synchronize the state of the standbys with the primary after each computation. As the standbys receive input, compute setpoints, and are state-synchronized, they can immediately take part in issuing setpoints when they are elected.
- **Passive Cold (PC):** Cold standbys in passive replication do not receive input. They only monitor the primary and perform leader election when the primary fails. In both protocols, leader election is performed using Fast Paxos in our simulations. As the standbys do not receive input and are not state-synchronized with the primary, they cannot take part in issuing setpoints in the same control round they are elected in.

PH and PC are expected to have lower latency than FP and Q, as they do not perform agreement between the replicas in each control round. However, as mentioned in Section 6.2, passive-replication protocols cannot guarantee consistency. We include them in our evaluation to compare their latency and availability performance with Quarts, highlighting their deteriorating performance when delay faults are introduced.

6.7.3 Simulation Methodology

We consider the fault model described in Chapter 3, Figure 3.3, which shows the Gilbert-Elliot model [133,134]. Each controller replica is considered to have independent faults,

that are either crashes or delays. We consider several scenarios, in which we vary the probability of crash faults (θ_c), the probability of delay faults (θ_d), the mean-time-to-recovery (MTTR) from crash faults (R). Given the values of these variables, the simulation parameters (q_C, q_N, p_d) of the Gilbert-Elliot model can be calculated, using the equations in Section 3.3.2.

We consider a delay threshold τ, *i.e.,* we consider that a replica with a computation time higher than τ in a given round is delay-faulty. We simulate the computation time as an exponential distribution with mean $1/\mu$, where μ is calculated as follows:

$$p_d = e^{-\tau/\mu} \implies \mu = -\frac{ln(p_d)}{\tau}$$

In our simulations, we consider a control round period $T_{ctrl} = 20\ ms$, and we consider the delay threshold $\tau = 8\ ms$. If a replica is delayed beyond T_{ctrl} in a given round, it is considered unavailable in that round. Additionally, for PC and PH, the standbys detect the primary as faulty if it fails to send heartbeats for a duration of τ. The upper-bound on one-way network latency is considered to be $\delta_n = 0.5\ ms$. Hence, the standbys are left with 12 round-trip times (RTTs) to elect a new primary.

Quality of Simulation Results

We use the relative accuracy (β) of an estimate ($\hat{a}$) of a probability (either availability or consistency) at a confidence level $1-\alpha$ as the measure of the quality of the estimate. Specifically, a $1-\alpha$ confidence interval of $\hat{a}$ has a length of η times the standard deviation of $\hat{a}$, where η is obtained from the relation $N_{0,1}(\eta) = 1-\alpha/2$. Then, c is given by $c = \frac{\sqrt{\hat{a}(1-\hat{a})/N}}{\hat{a}} \approx \frac{1}{\sqrt{N\hat{a}}}$, where N is the number of rounds. Therefore, we obtain $\beta = \frac{\eta}{\sqrt{N\hat{a}}}$. To obtain a relative accuracy of 10% with a 95% confidence level, we obtain $\beta = 10\%$ and $\eta = 1.96$.

We run each simulation until the relative accuracy of the estimate is less than 10% at a 95% confidence level. In other words, the 95% confidence interval of the estimated value has a half width, from the central value, less than 5%. Hence, all our results can be interpreted with the 95% confidence interval as $[0.95\hat{a}, 1.05\hat{a}]$.

Simulation Scenarios

We simulate four protocols (Q, FP, PH, PC) for several scenarios. The number of replicas (g) is varied between 1 and 5. The network loss probability (p) is varied between 10^{-4} and 0.05, for a total of 10 values. The number of RAs (n) is varied between 10 and 100, for a total of 5 values.

Scenario	n	g	θ_c	θ_d
#1	10	2	10^{-4}	10^{-3}
#2	100	2	10^{-4}	10^{-3}
#3	10	2	10^{-5}	10^{-4}
#4	10	2	10^{-4}	0
#5	10	3	10^{-4}	0

Table 6.1 – Select scenarios with their parameter values

Unless otherwise specified, the parameters take their nominal values, given by the following: $g = 2,\ n = 10,\ p = 10^{-3},\ \theta_c = 10^{-4},\ \theta_d = 10^{-3},\ R = 1\ s,\ \delta_n = 0.5\ ms$, $\tau = 8\ ms$, and $T_{ctrl} = 20\ ms$.

We present the availability as function of varying g and p. Then, we present the results of five representative scenarios, described in Table 6.1, in which we vary the number of RAs, the number of replicas, and the fault probabilities. Scenario #1 is the nominal scenario, in which the parameters take their nominal values. Scenario #2 increases the number of RAs to 100, showing the effect of a large number of RAs on the performance of the various protocols. Scenario #3 corresponds to a less severe fault model than scenario #1. Scenarios #4 and #5 correspond to a fault model with no delay faults. These show the performance of passive-replication protocols in the crash-only fault model they were designed for. Moreover, scenario #5 has an additional replica ($g = 3$), and shows the performance with 3 controller replicas.

6.7.4 Results

Figure 6.3 shows the unavailability (1 - Ψ) as a function of varying g and p. The first plot shows the results for varying the number of replicas. We see that, with 2 replicas, the unavailability of Quarts is an order of magnitude lower than that of other protocols. This difference increases to 4 orders of magnitude when the number of replicas becomes 3. The unavailability incurred by other protocols remains constant with additional replicas, whereas that of Quarts decreases. With 4 or 5 replicas, Quarts showed no unavailability after 10 billion simulation rounds (around 3 days of simulation). We conclude that the value is decreasing, but finding it requires more sophisticated techniques such as importance sampling and palm calculus [157].

The second plot shows the result of unavailability for a varying network loss probability. We see that, for all protocols, the unavailability increases as the loss probability increases. However, the unavailability of Quarts remains at least an order of magnitude lower than that of other protocols. Finding 6.1 summarizes the results of this figure.

Finding 6.1. *Quarts provides an availability higher than that of FP, PC, and PH. The availability of Quarts increases with the number of replicas.*

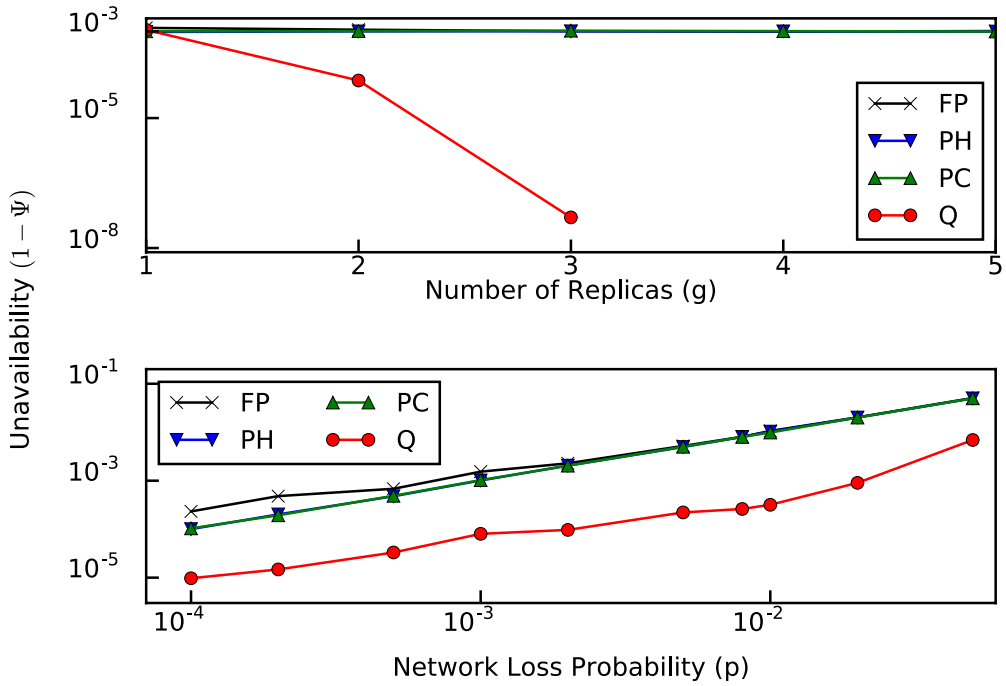


Figure 6.3 – Unavailability with varying g and varying p. Unavailability of Quarts with more than 3 replicas is less than 4×10^{-10}

Table 6.2 shows the results for inconsistency in the selected scenarios. As expected, Quarts and Fast Paxos guarantee consistency, hence provide zero inconsistency. However, as seen earlier, the consistency guarantee of Fast Paxos comes at the expense of low availability.

Additionally, we see that passive-replication protocols suffer from inconsistency in scenarios #1-#3. These are the scenarios in which delay faults are present. In scenarios #4 and #5, no inconsistency was observed in 10 billion runs, so we provide the confidence interval of the result. This reaffirms our premise that passive-replication protocols perform well when crash-only faults are considered, but their performance suffers in the presence of delay faults. These results are summarized in Finding 6.2.

Finding 6.2. *Quarts and FP guarantee consistency. PC and PH suffer from inconsistencies in the presence of delay faults.*

Figure 6.4 shows the mean and 99^{th} percentile of latency for the selected scenarios. We see that the mean latency of Quarts is less than that of Fast Paxos with a factor of 4. PC and PH have a latency comparable to Quarts, but this comes at the expense of inconsistencies as shown earlier. Fast Paxos has a high mean latency as it performs consensus in each control round. In contrast, Quarts incurs zero low latency-overhead with two replicas in most rounds, as discussed in Section 6.4.4.

	Inconsistency $(1-\Gamma)$	
Scenario: $(n,\ g,\ \theta_c,\ \theta_d)$	PH	PC
#1: $(10,\ 2,\ 10^{-4},\ 10^{-3})$	1.92×10^{-4}	1.28×10^{-3}
#2: $(100,\ 2,\ 10^{-4},\ 10^{-3})$	1.68×10^{-3}	1.50×10^{-3}
#3: $(10,\ 2,\ 10^{-5},\ 10^{-4})$	2.40×10^{-5}	1.38×10^{-4}
#4: $(10,\ 2,\ 10^{-4}, 0)$	$(0, 4\times10^{-10}]^{*2}$	$(0, 4\times10^{-10}]^{*}$
#5: $(10,\ 3,\ 10^{-4}, 0)$	$(0, 4\times10^{-10}]^{*}$	$(0, 4\times10^{-10}]^{*}$

Table 6.2 – Inconsistency results for the selected scenarios. Inconsistency of Quarts and Fast Paxos is zero

Furthermore, the tail latency of Quarts is lower than that of all other protocols in scenarios that include delay faults. In the absence of delay faults, the tail-latency for Quarts, PC, and PH is very low and comparable. Recall that the latency-overhead of Quarts is bounded, as shown in Theorem 6.2.

Finding 6.3. *Quarts has a lower average- and tail-latency than FP in all scenarios, and a lower average- and tail- latency than PC and PH in the presence of delay faults.*

Figure 6.5 shows the mean and 99^{th} percentile of messaging for the selected scenarios. We see that the results are comparable for all protocols. However, Quarts and Fast Paxos have a marginally higher messaging cost than PC and PH. Additionally, the messaging cost of Quarts and Fast Paxos increases with the number of replicas, as expected. This is due to the additional control messages being exchanged to guarantee consistency. We conclude the following.

Finding 6.4. *Guaranteeing consistency comes at the marginal expense of a higher messaging cost. This cost increases with the number of replicas.*

[2*] no inconsistency was observed in 10 billion runs

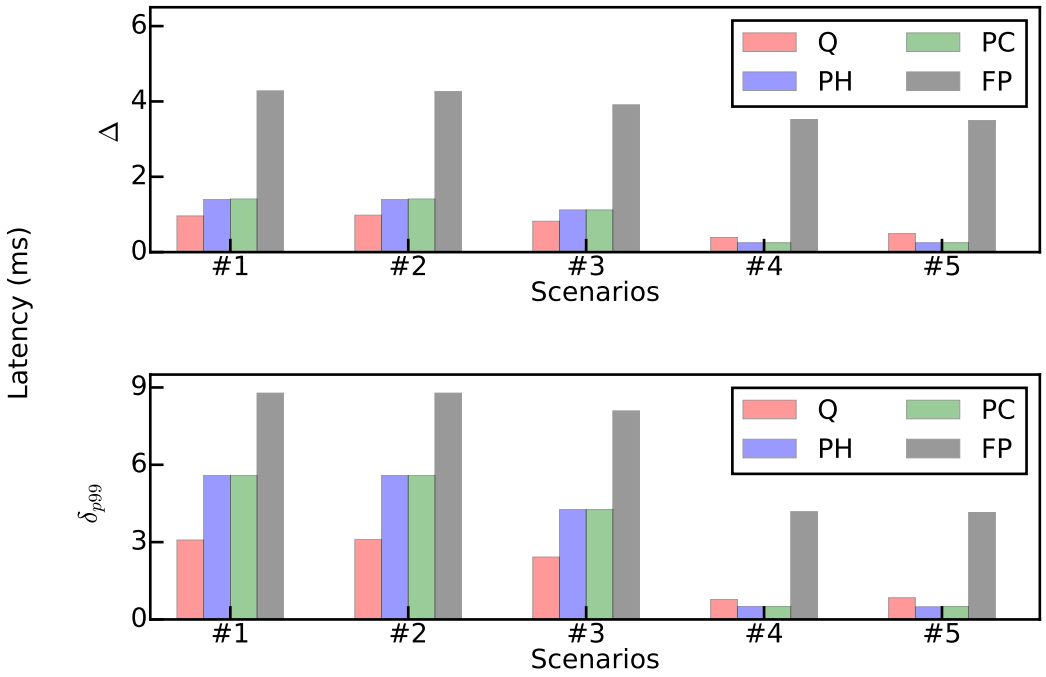


Figure 6.4 – Mean and 99^{th} percentile of latency in different scenarios

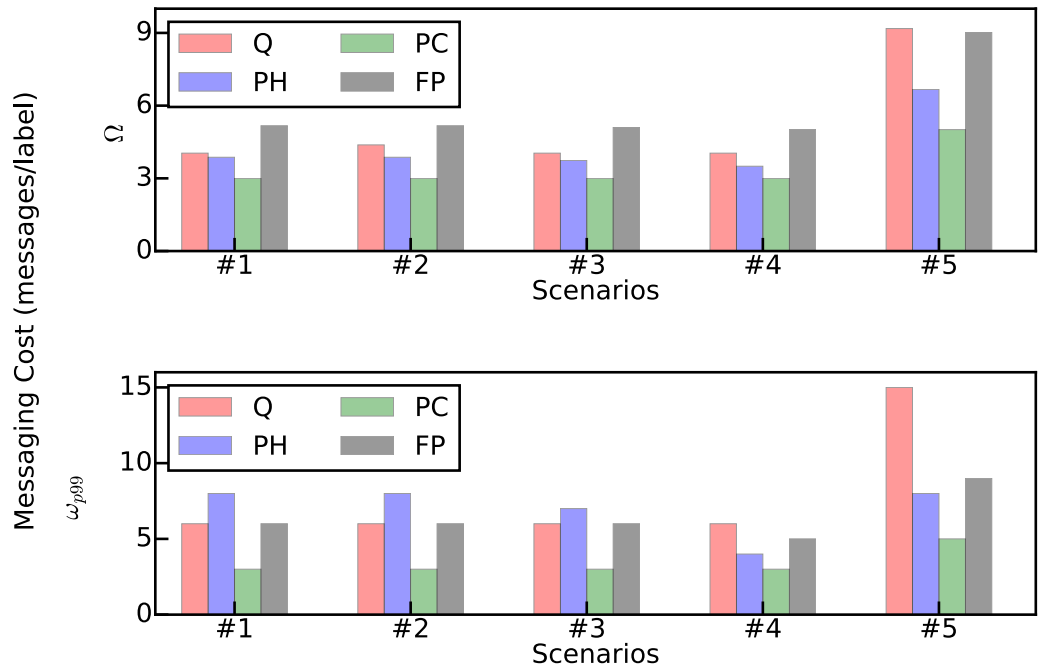


Figure 6.5 – Mean and 99^{th} percentile of messaging cost in different scenarios

6.8 Conclusion

In this chapter, we have discussed the need for reliable consistency between controller replicas in a CPS. We have discussed the split-brain syndrome, which arises when consistency is not guaranteed, and have shown its effects on a CPS for real-time control of electric grids.

We have presented the shortcomings of state-of-the-art solutions for guaranteeing consistency, which suffer from reduced availability and a high latency-overhead, especially in the presence of delay faults. We have proposed Quarts, an agreement protocol designed for real-time CPSs. We have presented the design of Quarts, and have shown how it can be applied to CPSs. We have formally proven that Quarts guarantees reliable consistency with a bounded latency-overhead.

We have performed extensive performance evaluation of Quarts, and compared its performance with that of existing agreement protocols using discrete-event simulation under different conditions of number of replicas, network losses, fault profiles, etc. The results show that besides guaranteeing consistency, Quarts improves the availability of a CPS by more than an order of magnitude, when compared with existing agreement protocols. Moreover, Quarts also improves the tail-latency performance of the CPS. These benefits come at the expense of a marginal increase in messaging cost when compared to passive-replication schemes.

Quarts combines with Axo (presented in Chapter 5) to enable the design of an actively replicated controller that tolerates delay and crash faults affecting the individual controller replicas. Thus, a reliable controller can be designed.

In Chapter 8, we deploy Quarts with COMMELEC [8], a CPS for real-time control of electric grids. We perform additional tests, both in T-RECS, a virtual commissioning tool, and on a real microgrid. These tests highlight the relevance of guaranteeing consistency in real-life CPSs.

7 T-RECS: Virtual Commissioning Tool for Real-Time Control of Electric Grids

Testing is an infinite process of comparing the invisible to the ambiguous in order to avoid the unthinkable happening to the anonymous.
— James Bach

In real-time control of electric grids using multiple software agents, the control performance depends on (1) the proper functioning of the software agents, *i.e.,* absence of software faults, and (2) the behavior of software agents in the presence of non-idealities, such as communication network losses and delays and software agent crashes and delays. To evaluate the control performance of such systems, we propose T-RECS, a virtual commissioning tool. T-RECS enables testing the performance of software-based control in-silico (before the actual deployment of software agents in the grid), saving both time and money. T-RECS can be used to study the effect of CPS control, on the grid, in both ideal and non-ideal conditions. It can also be used to study the effect of the robustness and reliability mechanisms, proposed in earlier chapters, on grid safety and operation.

Developers can run the binaries of their software agents in T-RECS where these binaries exchange real messages by using an emulated network and simulated models of the electric grid and resources. Consequently, the control of an entire microgrid can be tested on a standard computer. In this chapter, we first describe the design and the open-source implementation of T-RECS. Second, we measure its CPU and memory usage and show that our implementation can accommodate eight software agents on a standard laptop computer. Third, we validate the simulated grid used in T-RECS by replaying data collected from experiments performed in a real low-voltage microgrid. We find that the average error is 0.037% and the 99^{th} percentile of the error is less than 0.1%. This shows that T-RECS can replace in-field testing in the initial phases of development, providing accurate results for a fraction of the time and money.

7.1 Introduction

Real-time software-based systems for control of electric grids have the core of their control logic in software that is executed by multiple agents [8, 9, 144, 158]. These agents are typically distributed all over the grid and communicate using a communication network. They usually either control other lower-level agents or directly control different resources such as a battery, a super-capacitor, or an array of solar panels. As mentioned in Chapter 1, the rate of control varies depending on the system but for real-time systems, such as [8], it is typically sub-second.

7.1.1 Problem

As developers of these systems need to test their software agents before the actual deployment in the field, various testbeds [159, 160, 161, 162, 163, 164, 165] are proposed in the literature. For real-time software-based systems, the testing mainly concerns (1) the correct implementation of their distributed control logic and (2) the reliable communication among software agents. We find, however, that these testbeds are not appropriate for such testing.

First, current testbeds cannot test the final executables of software agents that are going to be deployed in the field. Instead these testbeds require either modeling of the control logic or the development of the control system in their specific language or framework. Second, for simulating the electric grid, these testbeds use physical equipment or hardware-in-the-loop, e.g., OPAL-RT eMEGAsim simulator or real-time digital simulator. This incurs a high cost and imposes serious limitations on the ease of use of such testbeds. Moreover, physical equipment cannot be used to study the grid in extreme conditions as this could cause potential damage.

To summarize, the main requirements of such a testbed are (1) the ability to use existing software agents with minimal modifications, (2) to avoid the use of physical equipment, and (3) to enable inducing non-idealities in the communication network and software agents. A testbed that satisfies these properties can be used by developers of software agents to design, test, and commission the agents before actual deployment in the field. As such tests can be performed entirely in-silico, we term such a testbed as a virtual commissioning tool.

7.1.2 Proposed Virtual Commissioning Tool

We propose a virtual commissioning tool, called T-RECS, for developers of multi-agent software-based control of electric grids. The design of T-RECS is divided into the same four layers that were discussed in Chapter 3: (1) the physical layer, (2) the sensing and actuation layer, (3) the network layer, and (4) the control layer.

The first and second layers in T-RECS are simulated in software. The physical grid in the first layer is modeled using the three-phase nodal-admittance matrix (Y-matrix) representation. The evolution of the grid is tracked through complex voltage phasors at each bus. These phasors are obtained by performing a load flow whenever there is a change in the grid state. Electric resources in the first layer, such as batteries, loads, and PV panels, are also simulated using state-of-the-art models, e.g., a battery model proposed in [142]. Sensors at the second layer are modeled in such a way that they can read the state of the grid from the simulated grid in layer one and then, can send this state to a software agent.

The third layer, *i.e.,* the communication network layer, is emulated using the Mininet framework [154]. This enables real packets to be exchanged between software agents, and we can easily study the effect of communication bandwidth, losses, and delays on the control performance. For the fourth layer, T-RECS provides users with virtual containers where software agents can be run without any modifications. Therefore, using T-RECS, we can verify whether the final executables of software agents are free from software bugs and if they correctly implement the control logic. The use of virtual containers also gives us the possibility to simulate the delays and crashes of agents or other software-related issues, hence enables developers to quickly investigate their effect on the control performance.

As described in Chapter 3, these four layers form the basic architecture of almost all multi-agent software-based control systems for electric grids. As T-RECS applies to all control systems that adhere to this architecture, it can be used seamlessly with a wide-range of control systems [8, 9, 144, 158].

Besides satisfying the requirements of a virtual commissioning tool listed earlier, T-RECS is designed to support real-time control systems. This entails fast updates of the simulated physical layer to reflect the changes in the grid and the electric resources, due to sub-second rate of control. This is possible due to our implementation of a fast, recent algorithm for solving the load-flow problem [139].

T-RECS is implemented entirely in software, and its implementation is made open-source[1]. Therefore, it reduces the barrier to study software-based control of electric grids. It is worth noting that the design of T-RECS involves the integration of several existing concepts such as Mininet [154], fast load-flow [139], and resource models, to obtain a usable and high-performing tool. This is particularly challenging in terms of interoperability between layers due to the heterogeneity of each layer.

T-RECS can only be used to study the steady-state behavior of the grid, as the software models it uses cannot capture the system transients and switching harmonics. Therefore, T-RECS cannot capture, test, or analyze the consequences of a frequency

[1] https://smartgrid.epfl.ch/?q=t-recs

Testbed	Enables inducing non-idealities	Enables using existing software	Avoids using physical equipment
[159]	✗	✓	✗
[160]	✗	✓	✗
[161, 162]	✗	✗	✗
[163]	✗	✗	✓
[164, 165]	✗	✗	✓
T-RECS	✓	✓	✓

Table 7.1 – Comparative summary of different testbeds

collapse, nor can it reflect the malfunctioning of PMUs due to such transients.

The rest of this chapter is structured as follows. In Section 7.2, we survey the literature on testbeds and compare T-RECS with the state of the art. We detail the design of T-RECS in Section 7.3. In Section 7.4, we present the results from the validation of T-RECS' grid model, by comparing its measurements to that from a real low-voltage microgrid. The successful validation confirms that T-RECS can be used to replicate results of experiments in real electric grids, thus supporting reproducible research. In Section 7.5, we evaluate the CPU and memory usage of T-RECS, in order to show that it can easily support control systems with multiple software-agents in a standard computer. Finally, we offer concluding remarks in Section 7.6.

7.2 Related Work

To the best of our knowledge, T-RECS is the first virtual commissioning tool that enables studying the performance of real-time software-based control systems, entirely in-silico.

Table 7.1 presents a comparative summary of the requirements (specified in Section 7.1) that are satisfied by T-RECS and other existing testbeds. We see that none of the existing testbeds satisfies all the three requirements of a testbed for software-based control of electric grids. Below we individually discuss the advantages and limitations of each testbed and compare it with T-RECS.

A testbed for decentralized control of active distribution networks is proposed in [161]. It consists of three main layers. The first layer performs the real-time simulation of physical power system elements in the OPAL-RT eMEGAsim simulator. The second layer requires the development of the multi-agent control system in the Java Agent Development Framework (JADE). Finally, the third layer models and simulates the communication network with OPNET Modeler. T-RECS also has these three layers, but they are managed differently. The first layer, *i.e.*, the simulation of physical power system elements, is done in T-RECS using software models instead of using the OPAL-

RT eMEGAsim simulator. This has both an advantage and a drawback. The advantage is that T-RECS is inexpensive, scalable, portable, and easily distributable as it does not require the physical equipment (OPAL-RT eMEGAsim simulator). The drawback is that T-RECS cannot study the effect of system transients or switching harmonics on the control software. This is because, in T-RECS, the software models of both the physical grid and electric resources are modeled in the phasor domain. The second layer in [161], *i.e.,* running multi-agent control software, is managed in T-RECS by using software containers provided by the Mininet framework. These containers can directly run existing or developed executables of software agents hence, as opposed to [161], T-RECS permits testing these final agent executables. Finally, the third layer, *i.e.,* the network layer, is emulated in the software using the Mininet framework. As the emulation of communication networks exchanges real packets, T-RECS enables easy and accurate study of the effects of different network bandwidths, losses, and delays in the communication network on the control performance.

Another testbed is proposed in [163]. Like T-RECS, this testbed is completely software-defined and does not involve physical equipment. However, it is not possible to test the effects of network communication technologies on the performance of software agents. This is because the communication network is neither simulated nor emulated. As today's distributed software-based control systems heavily rely on communication among different agents, the communication network is the main source of unexpected behavior of such agents, and not being able to measure it is a limitation of this testbed. Furthermore, as opposed to T-RECS, this testbed does not provide users with software containers, hence executables of multiple software agents cannot be directly run and tested with it. For example, in [163], the authors implemented their energy management software in one of the components of the testbed itself.

Another multi-agent testbed for power systems is proposed in [159]. This testbed is composed of a power-system simulator, computational platforms, and a data-communication infrastructure. As the testbed uses real hardware (computation platforms and communication infrastructure), it is neither inexpensive, portable, easily deployable, nor scalable. According to the authors of [159], these limitations can be removed if the computation platforms can somehow be virtualized or be placed in software containers and if the communications infrastructure can be emulated in the software. However, as noted in Section 7.3, this exercise poses several challenges due to the heterogeneity of the different components. In T-RECS, we divided the control framework into four manageable layers. This allows us to emulate the network infrastructure (in the network layer) and run different agents in multiple software containers (in the control layer). Thus, we overcome the limitations of [159].

A real-time testbed for operation, control and cyber-security of power systems is proposed in [160]. It targets the testing of low-level power-system control mechanisms,

such as system monitoring and fault detection. However, as opposed to T-RECS, this testbed is not software-defined and consists of many hardware devices such as the real-time digital simulator, the programmable logic controller, NI-PXI controller, and the Ethernet network. Although hardware-in-the-loop might have some benefits, it is not necessary if we target a testbed for the evaluation of effects of software and communication non-idealities on the control performance using software agents. This is the reason T-RECS is designed completely in software and has all the benefits of a pure software solution. Additionally, we find that these hardware devices, used in [160], run modified software (as compared to what runs in the real grid) or run software specifically developed for testbed purposes. This means that the testbed in [160] cannot test and validate the real software that is going to be deployed in the grid. On the contrary, T-RECS runs unmodified executables of software agents, hence a T-RECS user can easily figure out the runtime behavior and bugs of these software agents. Moreover, T-RECS support reproducible research.

In [162], the authors developed an agent-based testbed simulator for power grid modeling and control. They model the agents of the grid, instead of running the real agents in the testbed. As the modeling of software agents puts an additional burden on the testbed user and can test only the correctness of the logic, it is not enough to assess the correctness of software agents. The proposed testbed is hybrid: a part of the testbed is in software, but other parts require the presence of some minimal hardware and actual I/O signals. According to the authors, the hardware-in-the-loop is a complicated architecture and is therefore, not well suited for testing and validating the software-based multi-agent control systems that extensively rely on computational and communication technologies.

To investigate the effects of cyber-contingency on power system operations, a co-simulation model, based on information flow, is proposed in [164]. The authors model the network contingencies at a low level, e.g., delayed, disordered, dropped, and distorted information flows. The authors claim that these low-level parameters are easier to model than high-level network parameters such as denial-of-service and man-in-the-middle attacks. In contrast to this work, where they simulate the communication network with these low-level parameters, T-RECS emulates the communication network by using Mininet, which enables us to study the effects of different network bandwidths, losses, and delays corresponding to multiple real-world scenarios. As message exchanges are emulated in T-RECS, it accurately captures the real-time properties of the control protocol. Another important distinction of this work with T-RECS is that, in [164], the decision-making layer, *i.e.,* software agents, is also simulated, whereas T-RECS can run the real software agents without requiring the development of models of software agents.

To run software agents, Mosaik [165] uses process-based software containers, based on SimPy (a discrete event simulation platform). However, it has two drawbacks. First,

it does not model or emulate the communication network between different agents, thus it cannot be used to study the effect of non-ideal communication networks on the software-based control systems. Second, the control agents need to be re-written using Mosaik API, whereas T-RECS does not suffer from these limitations.

Next we discuss our choices for the different software solutions used in the T-RECS layers. To simulate the physical grid in the phasor domain, we use the three-phase load-flow algorithm proposed in [139]. We do not use the Newton-Raphson (NR) method for our load-flow computations, because the computation time of the chosen algorithm is faster than the load-flow implementation using NR (see Section 7.5). With regard to modeling the electric resources, we use existing state-of-the-art models, e.g., the battery model proposed in [142]. Moreover, T-RECS is designed such that users can plug in new models of electric resources as/if needed.

Apart from the testbeds designed for control of electric systems, there exists a vast amount of literature on modeling or simulating the electric resources and grid. Authors in [166, 167, 168] propose models for the most common resources such as loads, converters, batteries, solar panels, and electric vehicles, whereas the grid is simulated in phasor domain in [166, 167, 169]. With regard to modeling electric resources, T-RECS provides users with some state-of-the-art models but lets users plug in new models as needed. To simulate the electric grid in the phasor domain, we do not use PyPower [169] as it supports only single-phase load flow. Also, the grid model provided by GridLAB-D [166] is not appropriate because (1) it neglects phase-to-ground coupling capacitance, (2) it cannot take as input the new power setpoint at sub-second level on a given node, and (3) the inputs to the grid model are physical parameters of lines such as the type and length of wires instead of a Y-matrix. In T-RECS, to simulate the grid, we use the three-phase load flow algorithm proposed in [139].

7.3 T-RECS Design

Figure 7.1 shows the overview of T-RECS design. It follows the layering approach presented in Chapter 3, Figure 3.2. The physical layer and the sensing and actuation layer are simulated using state-of-the-art models of the grid and electric resources, respectively. To this end, T-RECS requires the grid topology, the resource types, and their parameters, as input. The details of the design of the physical layer and the sending and actuation layer are described in Sections 7.3.1 and 7.3.2, respectively.

The network layer is emulated in Mininet [154], which uses virtual switches and hosts to simulate the switches and routers in the network. For this layer, T-RECS takes as input the topology of the communication network, the bandwidths, the losses, and delays of different links. The detailed design of the network layer is presented in Section 7.3.3.

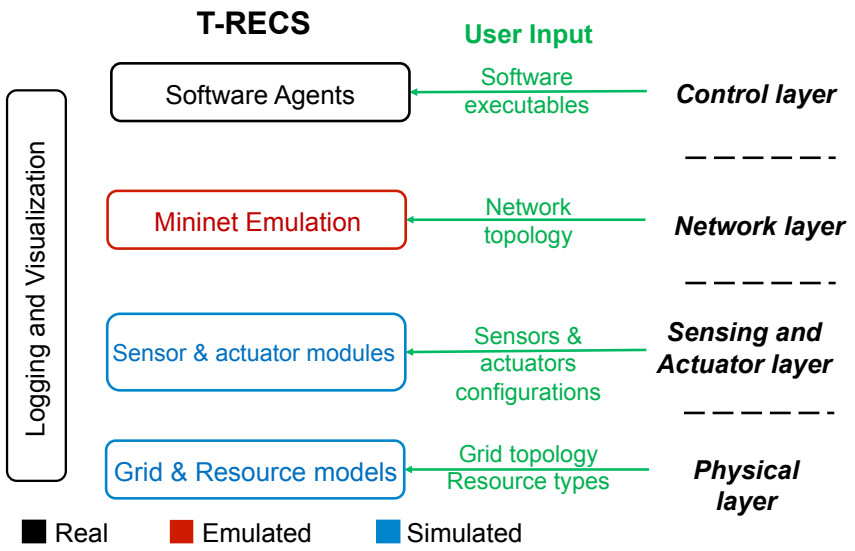


Figure 7.1 – T-RECS design highlighting the mapping between the layers and the implementation

The control layer is built through unmodified software agents that are run in Linux containers [170] provided by Mininet. The detailed design of the control layer is presented in Section 7.3.4.

In line with the requirements of a virtual commissioning tool listed in Section 7.1, the layers of T-RECS are built in software, thus making it feasible to study all elements of real-time control of the grid within a computer.

7.3.1 Physical Layer

The two components of the physical layer are the grid and the resources.

Grid

The grid is represented as a set of complex voltage phasors at each bus. To this end, the grid is modeled by its nodal-admittance matrix. We use a three-phase model in order to be able to simulate distribution networks, which are often unbalanced. The grid model takes as input the grid topology (connection between the buses and the line parameters) and computes the nodal-admittance matrix. The frequency of the grid is taken as a dynamic input that can change during the execution, e.g., it could be dictated by the slack bus or by some generator.

For each change of voltage or power at a bus, the grid model performs a three-phase load flow to obtain the voltage at all the buses and the current in all the lines of the grid. As the speed of the load-flow computation dictates the responsiveness of the grid

model, we chose to implement the method proposed in [139], as it boasts a significant decrease in computation time when compared to the classic Newton-Raphson method. In Section 7.5.2, we evaluate how the computation time of the load flow varies with different sizes of the grid.

As the load-flow analysis is a light-weight representation of the grid, it provides the steady-state information of the grid without considering system transient processes. We make this trade-off in order to be able to simulate the grid in-silico. In Section 7.4, we compare results from the load-flow analysis of the grid model to those obtained from a real grid and show that the average error is 0.037% and the 99th percentile of the error is less than 0.1%.

Resources

The resources in T-RECS are simulated using existing models of electrical resources. As mentioned in Chapter 3, there are two types of resources: controllable and uncontrollable. For example, a battery is a controllable resource if its output power can be modulated through setpoints, whereas a load that cannot be curtailed is an example of an uncontrollable resource.

In our open-source implementation, we make four resource models available: a controllable battery, an uncontrollable PV, an uncontrollable load, and a controllable flex-house. The battery is modeled using the two-time constant model described in [142]. The uncontrollable load and PV resources are simulated by replaying a timestamped trace of the power injections. The flex-house model captures the heat dynamics of buildings and is used to simulate controllable thermal loads, as described in [171]. The change in the output of a resource is reflected as a change in the state of the bus on which the resource is placed.

The open-source implementation of T-RECS enables the addition of new resource models. Each resource model performs three tasks. First, the resource periodically updates its state according to the resource dynamics. For example, the battery updates its state-of-charge (SoC) based on the elapsed time and output power. Second, when it receives a request from a sensor, it responds with its internal state. Lastly, when it receives a request from an actuator, it changes the output power and the power at the corresponding bus in the grid model.

7.3.2 Sensing and Actuation Layer

The sensors and actuators are simulated as application programming interfaces (APIs) that act as an interface between the physical layer and the control layer. There are two types of APIs: the grid API and the resource API. The grid API provides methods to get

the state of the grid and set the power at each bus (except slack bus) in the grid. The state of the grid consists of voltage phasors and active/reactive power at each bus, and line currents. The resource API provides functions for reading and changing the state of the resource model, such as the power injected by a battery. These APIs can be used to create specific sensors and actuators, as required by the user's configuration.

In the current release, T-RECS provides an implementation of a simulator for a real-time state estimator that periodically calls the grid API to get the state of the grid and then, periodically sends the state of the grid in a given format to a list of agents specified by the user. We also provide an implementation of an actuator that alters the state of the battery resource used in the experiments in Section 7.4.

Recall from Chapter 3 that sensors might be synchronous or asynchronous. Synchronous sensors are queried by software agents (specifically, the RAs) for measurements. In contrast, asynchronous sensors stream measurements to software agents (specifically, the controllers) without being queried. The measurements are sent as real packets via the network layer.

7.3.3 Network Layer

The network layer in T-RECS is emulated using virtual switches, routers, and hosts, provided by Mininet [154]. The main advantages of using an emulated network as opposed to a simulated network done by other works are the following: (1) real messages are exchanged in Mininet in contrast to the discrete-event simulation of messages, (2) the effect of bandwidth limitations can be accurately studied because real switches are emulated, and (3) newly developed network protocols can be easily studied without any modification to the protocol implementation.

The switches are realized by Open vSwitch [172], a programmable multi-layer switch that can be used to accurately emulate practically any L2 and/or L3 network topology. It can also be interfaced with a software-defined networking controller to emulate a large-scale managed network used in sub-station automation networks.

Using the rich set of tools from the Linux traffic-control suite `tc-netem` [173], we impose bandwidth and delay restrictions on the links to replicate a real-life network topology. Additionally, T-RECS enables us to use message-loss profiles provided by `tc-netem` and queuing disciplines, in order to capture the real-life network more accurately. This facilitates studying the performance of the control system in non-ideal network conditions, a requirement of such a virtual commissioning tool.

The end hosts of the network are Linux containers (provided by Mininet) that are used to run the software agents in the control layer. The network topology and the link configurations are taken as an input.

7.3.4 Control Layer

The control layer is identical to the real world. T-RECS takes the executables of the software agents as input and executes them in the software containers provided by Mininet. Just as in an actual deployment, the unmodified executables receive messages from sensors and other software agents, perform computations, exchange messages, and send setpoints.. The control layer is realized by executing one software agent in each Mininet host, with controllers receiving real measurements and advertisements and sending real setpoints, and RAs receiving measurements from sensors and setpoints from controllers, just as they would when run in the field. In this way, T-RECS recreates an environment in which the software agents can be executed and tested without modifying their code, as envisioned in our requirements for such a tool.

Although our current implementation runs on a single computer, it is straightforward to extend to run on a cluster of many computers for high scalability. A single computer in a cluster could host several software agents, and the communication links between them can be modified to reflect the real-life network by using the same set of tools from the Linux traffic-control suite.

Note that, although the focus is on control systems for electric grids, the layering scheme utilized here can be applied to other domains. Consider the example of a self-driving car [146]. The physical layer comprises the dynamics of the car and the dynamics of the environment it runs in, both of which are governed by laws of classic mechanics. The sensing and actuation layer sensors include the speedometer for the current speed, and the actuators include a PID controller that maintains the speed at a given setpoint. The control layer includes the software agents that detect objects, perform obstacle-avoidance and navigation, etc. In fact, the same design philosophy used in T-RECS can be used to design low-cost, in-silico, virtual-commissioning tools for multi-agent software-based control systems in other domains.

7.4 Validation

In this section, we validate the T-RECS grid model described in Section 7.3.1. Recall that the T-RECS grid model performs load-flow computation and updates the state of the grid whenever there is a power injection or absorption at a bus. Thus, this section aims to quantify the error committed by the load-flow solver of the T-RECS grid model as compared to the measured state of a real grid.

Figure 7.2 shows the topology of our in-house microgrid at EPFL. This microgrid is used in the experiments described below. It consists of 13 buses, labeled B01-B13, and reproduces, in real scale, the topology defined by the CIGRÉ Task Force C6.04.02 [44]. The resources used in our experiments are shown in solid points in

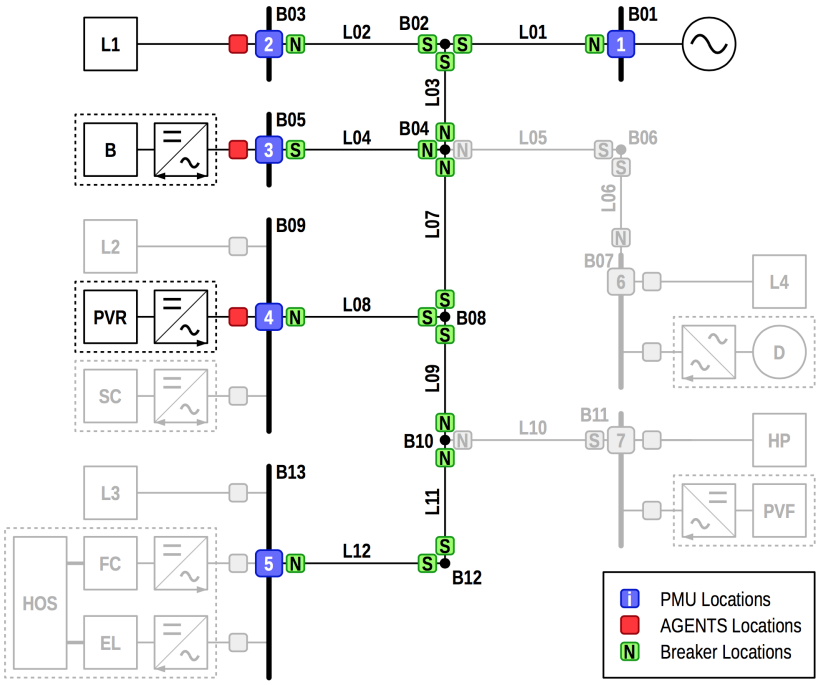


Figure 7.2 – Grid topology used for validation. The resources not used in our experiments are greyed-out

Figure 7.2. They consist of a 24 kW controllable load on bus B03, a 20 kW, 25 kWh controllable battery on bus B05, and an uncontrollable PV generator of 13 kWp on bus B09. The microgrid is monitored in real-time using phasor measurement units (PMUs), a phasor data concentrator (PDC), and a real-time state estimator (SE) [53]. From the SE, we obtain the timestamped traces of voltage and current phasors at each bus, with one measurement every 20 ms.

The traces are collected during an experimental validation [174] of a real-time control framework, called COMMELEC, further described in Chapter 8. COMMELEC is a real-time framework that controls the given resources in real-time using explicit power setpoints [8].

We validate the T-RECS grid model under two different scenarios described in [174], using two separate objectives for the GA. The first scenario is concerned with grid safety. In this scenario, the GA maintains the grid in a feasible state, *i.e.,* respects the ampacity limits of all the lines, voltage limits of all the buses, and the constraints of all the resources, in a grid with an uncertainty in power prosumption due to the load and the PV. The second scenario is real-time dispatchability, in which the GA tracks an external dispatch signal, while also maintaining the grid in a feasible state.

In order to quantify the error between the measurements from the experiments and the output of the load-flow solver in T-RECS, we use measured voltage and power

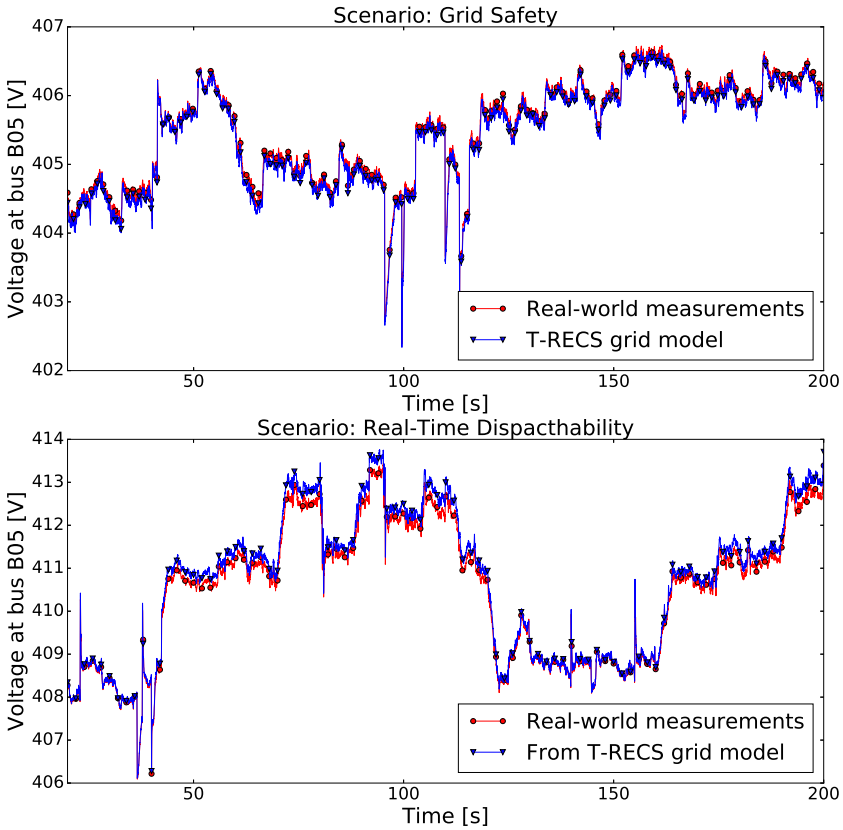


Figure 7.3 – Voltage at the battery bus (B05) obtained from measurements and from grid model

traces from the two experiments as follows. At every 20 ms timestamp, we use the power injections at all the buses and give them, along with the voltage magnitude at the slack bus, as input to the grid model. The grid model performs the load-flow computation and returns the voltage at each bus, and the power at the slack. At each bus, we compare this voltage against the voltage obtained from the measurement traces at the same timestamp.

For the two scenarios mentioned earlier, Figure 7.3 shows the voltage at the battery bus (B05) obtained from the measurements and from the grid model during a three-minute window. We see that the voltage from the grid model closely follows the measurements. However, we note that the error is relatively higher in the second scenario of real-time dispatchability. This is because the instantaneous grid parameters (resistance, reactance, and susceptance of lines) are a function of the temperature and frequency of the line. The voltage error depends on the difference between the instantaneous grid parameters of the real grid and the static grid parameters used by T-RECS. The impact of this difference in parameters is higher for higher voltage amplitudes. This error is unavoidable because it is hard to estimate the instantaneous grid parameters, but we observe that it is below 0.1%.

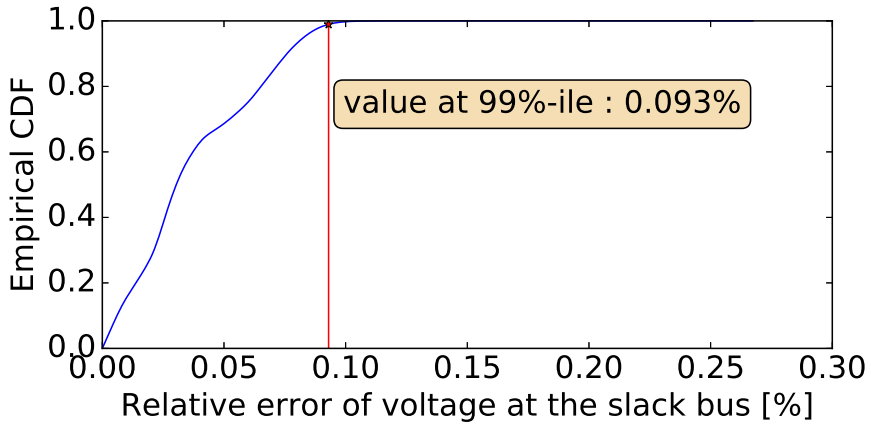


Figure 7.4 – Empirical CDF of the relative error in voltage at all the buses

Figure 7.4 shows the empirical cumulative-distribution function (CDF) of the error between the voltage measured by PMUs at each bus and the voltage obtained by load-flow from the grid model. The CDF is computed using the entire data from the experiments, which amounts to 750,000 data points. We see that the average value of the relative error is 0.037% and its value at the 99%-ile is 0.093%. Thus, we conclude that the error incurred by the grid model is negligible.

7.5 Performance Evaluation

In this section, we perform two performance studies of our T-RECS implementation. First, in Section 7.5.1, we evaluate the CPU and memory usage of T-RECS as a function of number of software agents in the control system. The goal of this performance evaluation is to show that T-RECS can easily accommodate several software agents on a standard laptop computer. Then, in Section 7.5.2, we study the execution time of the load-flow computations by the grid model for three benchmark grids of different sizes. The aim of this study is to show that due to its ability to quickly update the state of the grid, T-RECS is suitable for real-time control systems for electric grids, which have sub-second rate of control.

All experiments are performed on a Lenovo T400 laptop with 3.7 GB RAM and 2.67GHz Intel Core i7 processor. The operating system is 64-bit Ubuntu 16.04 LTS with the Intel virtualization technology enabled.

7.5.1 CPU and Memory Usage

We use the COMMELEC control system with the same CIGRÉ benchmark low-voltage microgrid used in Section 7.4. In our setup, COMMELEC consists of one GA and several RAs. Recall that each RA is attached to one resource. For example, if COMMELEC

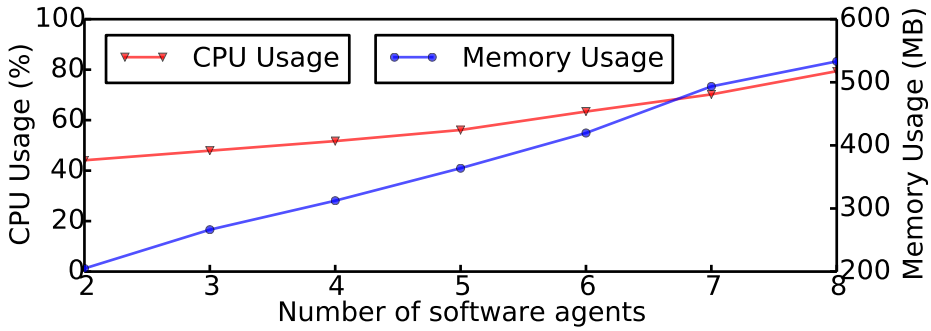


Figure 7.5 – CPU and memory usage of T-RECS, on a laptop with 3.7 GB RAM and a 2.67GHz Intel Core i7 processor, as a function of number of software agents. CPU usage in percentage is cumulative of all four CPUs of the i7 processor

monitors and controls two resources, then it has to run three software agents: one GA and two RAs corresponding to the two resources.

To better interpret our performance results, it is important to highlight that the COMMELEC control system is run at the pace of 100 ms, *i.e.,* RAs send state of their controlled resources every 100 ms to the GA, and the GA sends power setpoints to RAs every 100 ms. This has two implications for T-RECS: (1) there is a heavy load of messages to emulate in the network layer, and (2) there are frequent load-flow computations to be performed in the grid module. Additionally, as the GA requires the state of the grid every 20 ms as input, the T-RECS sensor module must send this information every 20 ms.

We run multiple experiments with different numbers of COMMELEC software agents (and corresponding resources) and record the CPU and memory usage of T-RECS in each case. The CPU usage reported below is cumulative of all four CPUs of the i7 processor and is solely due to T-RECS processes.

When T-RECS is not running, the CPU usage is 1.5% and the memory usage is 563 MB. Figure 7.5 shows how the CPU and memory usage scales with the number of software agents. We find that the CPU usage of T-RECS starts off at 44.1% in case of 2 software agents and increases linearly with a rate of 6% per additional software agent or resource. Moreover, the memory footprint of T-RECS is close to 200 MB with 2 software agents and increases at a rate of around 55 MB per additional resource or software agent.

The initial high CPU usage can be explained by the initialization of all the T-RECS components. Also, the increase in the CPU usage with the number of software agents is linear, as expected. This is because, with an additional software agent, the burden of T-RECS increases linearly in the following three directions. First, it needs to run one additional resource model of the resource managed by the new software agent. Second,

Benchmark Grid	T-RECS Load-Flow (in ms)	Newton-Raphson (in ms)
CIGRÉ 4-bus	0.75 ± 0.02	23.28 ± 0.39
CIGRÉ 13-bus	1.13 ± 0.02	232.19 ± 1.05
CIGRÉ 34-bus	7.42 ± 0.14	1594.26 ± 5.29

Table 7.2 – Average execution times (in ms) of the the load-flow implementation used in T-RECS and the Newton-Raphson method, for three different CIGRÉ benchmark grids, measured at 95% confidence

the number of load-flow computations in the grid engine increases linearly with the number of resource models because the number of updates sent to the grid module increases. Finally, the communication network emulation layer needs to emulate a fixed number of additional packets.

We see that, when running on a modest laptop, T-RECS can support up to eight software agents (1 GA and 7 RAs). To put this in perspective, a typical microgrid is controlled with one GA and about five RAs. Hence, we conclude that developers of software-based control systems can use T-RECS, for virtual commissioning, on a general-purpose desktop/laptop.

7.5.2 Load-Flow Computation

Recall from Section 7.3.1, that every time the power injected or consumed by a resource changes, a load-flow computation is triggered by the grid model. This serves to reflect the effect of the new power injection or consumption on the other buses in the grid. The time taken by the load-flow computation is, therefore, the time taken for a given update to be reflected in the simulated grid of T-RECS. Therefore, it is important to quantify the execution time of the load-flow computation performed by the grid model of our implementation. This importance is even more prominent in case of real-time control systems with a sub-second rate of control.

The load-flow computation algorithm used by T-RECS is described in [139]. We compare the time taken by the Python implementation of this algorithm in T-RECS for three different CIGRÉ benchmark grids [175]. These grids are of different sizes and consist of 4, 13, and 34 buses, respectively. In our tests, we set the desired relative accuracy of load-flow results as 10^{-6}, and we set the maximum number of iterations performed by the load-flow solvers to be 100. To highlight the improvement in computation time over traditional load-flow solvers that use Newton-Raphson method, we have also implemented the load-flow computation by using state-of-the-art Newton-Raphson algorithm in Python. We also run the same test cases with this algorithm.

In Table 7.2, for the three grids of different sizes, we report the mean computation time and the confidence interval for the mean at 95% confidence, computed from 100

samples. We see that the average computation time of the T-RECS grid model for grids of 4, 13 and 34 buses are 0.75 ms, 1.13 ms and 7.4 ms, respectively. This computation time is well below the required update time (of 100 ms) for a real-time control system like COMMELEC, thereby confirming the real-time capability of T-RECS. We also observe a sharp decrease in computation time when compared to a traditional load-flow solver that takes 23 ms, 232 ms, and 1.6 seconds for the same grids, respectively. This affirms our choice to use the fast, recent load-flow algorithm proposed in [139].

7.6 Conclusion

We have presented T-RECS, a virtual commissioning tool that is used for designing, testing, and validating multi-agent real-time control software for electric grids. It enables developers to test the executables of their software agents without requiring any modifications to them. The effect of non-ideal communication networks on the control performance can be studied using T-RECS, as real packets are being exchanged between the software agents. This is made possible by emulating the communication network layer in T-RECS using Mininet. T-RECS simulates the physical grid using a phasor-domain load-flow solver, and uses state-of-the-art models to simulate the electric resources. To the best of our knowledge, T-RECS is the first virtual commissioning tool for real-time software-based control of electric grids.

The main design criteria for T-RECS are the ability to run unmodified code, operate in real-time, and study the effect of non-ideal communication network on the control performance, all without requiring physical equipment. Indeed, T-RECS can be run entirely on a standard laptop or desktop computer. This makes T-RECS the ideal tool for reproducible research in the field of control of electric grids.

We have made available an implementation of T-RECS and have validated the load-flow solver in T-RECS by running the same experiment in a real microgrid. We have found that the tail relative error is less than 0.1% when comparing the results from the two experiments.

T-RECS was used in Chapter 4 to evaluate the benefits of applying, to COMMELEC, the robustness mechanisms proposed in that chapter. In fact, T-RECS is actively being used in the co-development of the COMMELEC framework, as it facilitates and accelerates the design-test cycles of that process. In the next chapter, we use T-RECS to study the effect of network non-idealities on the performance of COMMELEC. Then, we show, using T-RECS, how applying Axo (Chapter 5) and Quarts (Chapter 6) improves the performance of COMMELEC.

8 Case Study: COMMELEC - A Cyber-Physical System for Real-Time Control of Electric Grids

The true method of knowledge is experiment.
— William Blake

In previous chapters, we have presented the system model for CPSs for real-time control of electric grids, proposed robustness and reliability mechanisms for such CPSs in the presence of communication network non-idealities and software agent delays and crashes, and proposed T-RECS, a virtual commissioning tool for co-designing and testing such CPSs. In this chapter, we highlight a case study of an existing framework for real-time control of electric grids: COMMELEC [8].

We introduce the COMMELEC framework and briefly discuss its architecture in relation to the system model presented in Chapter 3. We discuss the operational scenarios and objectives of COMMELEC when it monitors and controls a microgrid in grid-connected mode. Then, we study the effect of non-idealities in a CPS with COMMELEC, both on grid safety and on performance in terms of meeting the objectives.

In Chapter 4, we have seen the improvements brought about by applying intentionality clocks and Robuster to COMMELEC, in the presence of non-ideal communication. In this chapter, we showcase the improvements brought about by applying Axo (Chapter 5) and Quarts (Chapter 6) to COMMELEC, in the presence of software agent crashes and delays and communication network non-idealities.

The enhanced robustness and reliability of COMMELEC enables performing mission-critical tasks with strict real-time requirements. We introduce one such use-case scenario of COMMELEC, namely the islanding maneuver and operation. The islanding functionality in COMMELEC is split into two parts: slack ranking and slack switching. We present the existing mechanism used in COMMELEC for performing slack ranking.

Then, we propose a mechanism to perform slack switching for all the islanding scenarios: disconnection, reconnection, and islanded mode. Our proposed mechanism takes into account the possible non-idealities in the communication network and in the resources being controlled. Finally, we discuss how the robustness and reliability mechanisms, proposed throughout this thesis, are applicable to the islanding scenario in COMMELEC.

8.1 The COMMELEC Framework

COMMELEC [8] is a framework for real-time control of electric grids via explicit power setpoints. Software agents in COMMELEC communicate, in a control period of 100 ms, by using an abstract device-independent language, thereby hiding the complexity of the resource they monitor and control. This enables COMMELEC to be composable and scalable, as a generic software agent implementation can be used to monitor and control grids, independent of the number and types of resources. This is further discussed as we go over the architecture and components of COMMELEC in Section 8.1.1.

The main goal of COMMELEC is to maintain grid safety, ensuring that voltage and current bounds are not violated. Additionally, it can perform various ancillary services. In Section 8.1.2, we enumerate some of these services when COMMELEC is controlling a grid-connected microgrid. In Section 8.5, we discuss, in further detail, how COMMELEC can perform islanding, and subsequently handle an islanded microgrid.

8.1.1 Architecture

As COMMELEC is a framework for real-time control of electric grids, a CPS with COMMELEC has an architecture that closely follows the system model presented in Chapter 3.

In COMMELEC, the grid is monitored by phasor measurement units (PMUs) [176], located at various buses. The PMUs are time-synchronized with one another via GPS [80], and they are configured to send a voltage-phasor measurement every 20 ms. The measurements sent by PMUs are received and aggregated by a phasor data concentrator (PDC) that time-aligns these measurements and forwards them to the state estimator (SE) component of the COMMELEC controller. The PMU/PDC pairing, therefore, acts as a time-triggered asynchronous sensor, as described in Chapter 3.

Each resource in the grid is assigned a resource agent (RA) that monitors its state through a local synchronous sensor and, possibly, controls it through a local actuator. The setpoints received by the RAs are active and reactive power values, which must be implemented in the resource. The RA translates the internal state of the resource into a device-independent language, by creating advertisements that consists of the

following three mathematical terms:

- a PQ-profile ($\mathcal{F}$): a region in the PQ-plane to which setpoints, sent to this resource, must belong.
- a belief function ($\mathcal{U}$): a function that maps a PQ power setpoint to a set that represents the uncertainty of the resource.
- a virtual cost function ($\mathcal{P}$): a function that maps a PQ power setpoint to a real number that represents the cost, or preference, of the resource in implementing this setpoint.

These terms represent the feasibility, uncertainty, and preference of the resource, respectively, as described in Chapter 3. The advertisement formed of these terms is sent to the grid agent (GA) component of the COMMELEC controller.

The GA receives advertisements from the RAs and state messages from the SE. These are used to compute setpoints that are then issued to the RAs. This control round is repeated every 100 ms.

8.1.2 Grid-Connected Operation

As mentioned earlier, the main goal of COMMELEC is to maintain grid safety (Definition 3.1). That is, the voltage at each bus in the grid must be maintained within its bounds [15], and the current in each line must be maintained below the ampacity of the corresponding line. COMMELEC ensures this by performing a load-flow computation [37], considering the set of setpoints being issued, in addition to the uncertainty of the resources in implementing them (as given in the belief functions). Such a check is called the admissibility test [139], and it can be extended to account for non-idealities, as mentioned in Chapter 4.

In addition to maintaining grid safety, COMMELEC enables a wide range of ancillary services. One example is tracking a dispatch plan in a grid-connected microgrid, thereby acting as a virtual power plant to the main grid [18]. In such a scenario, the GA is configured with a power profile that it must track at the point of common coupling (PCC). That is, the GA must steer the power of the resources in the grid in order to track the power profile requested at the slack. It must do so in the presence of variable consumption and production in the grid, and it must maintain grid safety throughout this operation. We show an example of tracking a dispatch plan in Section 8.2.

Another ancillary service is frequency support, whereby the microgrid controlled by COMMELEC can counteract frequency fluctuations in the main grid by adjusting the power imbalance at the PCC. The COMMELEC GA can perform frequency support

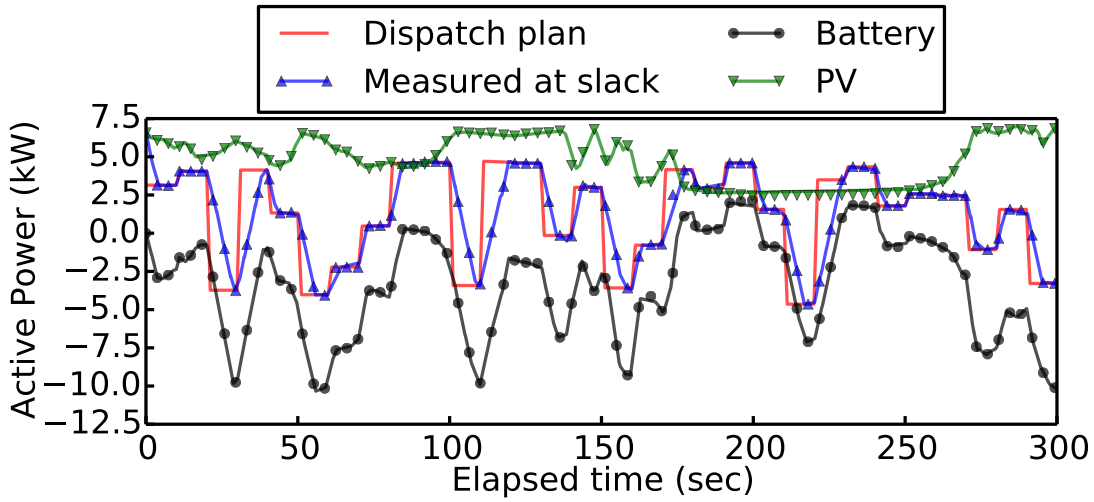


Figure 8.1 – Power profiles for the tracking scenario in COMMELEC under ideal conditions

along with the tracking of a dispatch plan. In Chapter 4, we have seen examples of COMMELEC performing frequency support.

8.2 Effect of Non-Idealities on COMMELEC

In this section, we study the effect of communication network non-idealities on the performance of COMMELEC when tracking a dispatch plan. In this experiment, we run a single non-replicated COMMELEC controller, without any robustness mechanisms. We perform the experiment in T-RECS (described in Chapter 7), and we do not subject the software agents to delays or crashes, in order to focus on the effect of message losses, in the communication network, on the performance of COMMELEC.

We study a setup that consists of two resources: a 20 kW / 20 kWh controllable battery, and an 11 kW uncontrollable PV installation. These resources are located on different buses of a CIGRÉ low-voltage benchmark microgrid [44] — the same one that was described in Chapter 7. The PV power trace is taken from real measurements.

Figure 8.1 shows the power profiles during the experiment, under ideal conditions. This includes (1) the dispatch plan for the given 5-minute interval, (2) the power trace of the PV resource, (3) the measured power at the battery bus, and (4) the measured power at the slack bus. We observe that COMMELEC attempts to counteract the variations in the PV production with the flexible battery resource, in order to track the dispatch plan at the slack bus. We see that the measured power at the slack closely follows the dispatch plan, albeit with some delay.

Figure 8.2 shows the effect of network losses on the performance of COMMELEC.

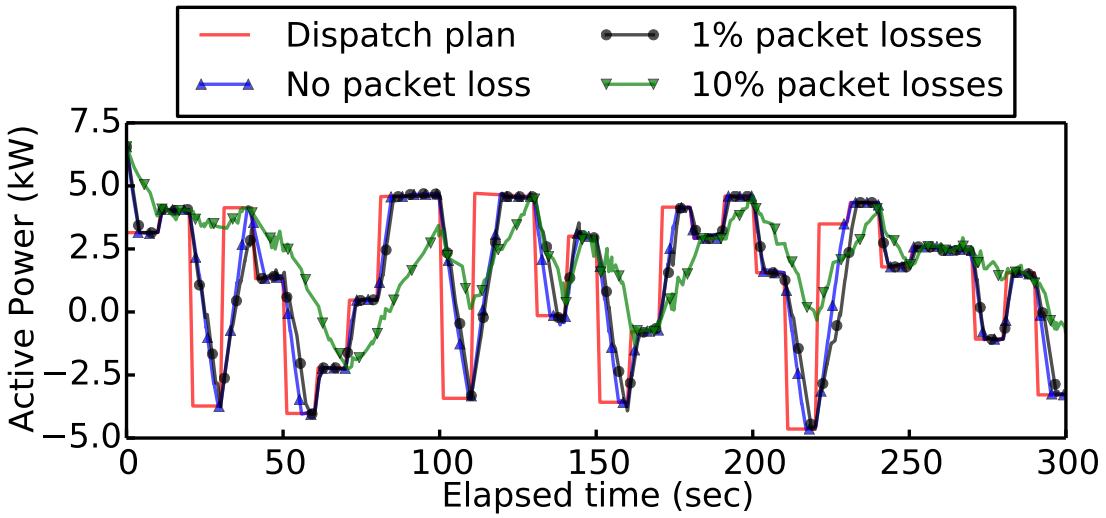


Figure 8.2 – Power profiles for the tracking scenario in COMMELEC under varying network loss probabilities

Packet loss probability	0%	0.5%	1%	5%	10%
Normal COMMELEC controller	0.09	0.10	0.12	0.15	0.18
Robust COMMELEC controller	0.09	0.09	0.09	0.10	0.10

Table 8.1 – Energy mismatch (in kWh) for the tracking scenario in COMMELEC under varying network loss probabilities. Robust COMMELEC controller refers to a controller that implements intentionality clocks and Robuster

The dispatch plan and the measured power at the slack in the absence of packet losses are the same as shown in Figure 8.1. We see that, with a 1% packet loss rate, the measured power at the slack increasingly deviates from the dispatch plan. This deviation becomes more profound with higher loss rates, as observed with the plot for 10% loss rate.

We also measure the energy mismatch, during the 5-minute interval, for various loss rates. The results are shown in Table 8.1. As expected, the energy mismatch increases with an increasing loss probability.

In Chapter 4, we have shown how applying intentionality clocks and Robuster to COMMELEC results in a robust controller that is capable of maintaining tracking performance, in spite of non-idealities. We observe this behavior as well in Table 8.1. The robust COMMELEC controller maintains a tracking performance close to ideal conditions, even under 10% packet loss probability.

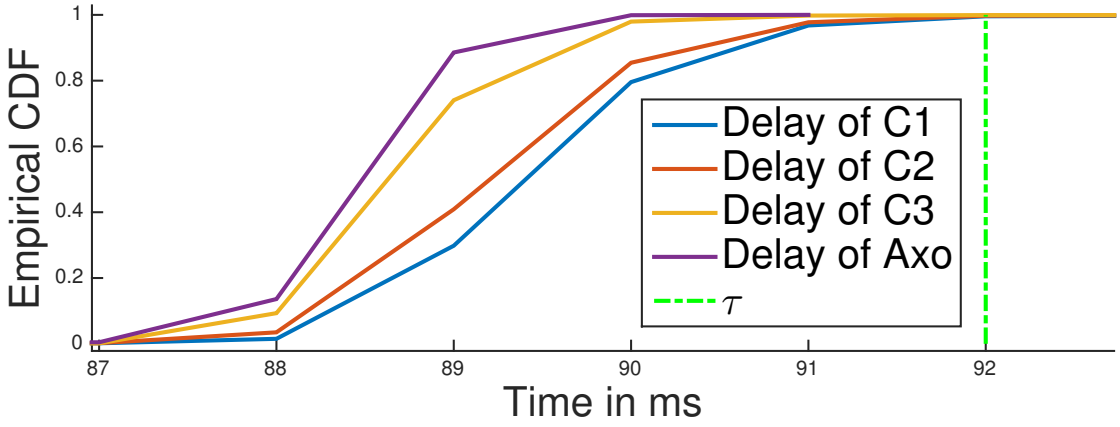


Figure 8.3 – Delay profile of the controller replicas in COMMELEC with and without Axo. τ represents the validity horizon of the setpoint

8.3 Application of Axo to COMMELEC

We study the improvements brought about by applying Axo, a fault-tolerance protocol for delay and crash faults, to COMMELEC. We perform two long-running experiments: one in T-RECS, and one in a real-life microgrid. The setup is the same as the one in Section 8.2, with a battery and a PV. However, the COMMELEC controller is replicated, an Axo controller library is implemented on all the replicas, and an Axo RA library is implemented on all the RAs.

In the first experiment, we run three replicas of the controller (C_1, C_2, C_3) in T-RECS. Recall from Chapter 5 that the validity horizon of setpoints is equal to $\tau = n \times T_{ctrl} - \delta_i$, where n is the number of rounds for which advertisements are valid, T_{ctrl} is the control period, and δ_i is the upper bound on implementation time of a setpoint at the RAs. As we only consider short-term advertisements in this experiment, $n = 1$. The control period in COMMELEC is $T_{ctrl} = 100\ ms$. The upper bound on implementation time was measured to be $3\ ms$, which results in $\tau = 97\ ms$.

First, we perform the experiment for around 10 million control rounds (approximately 12 days). We observe that 32 setpoints (0.00032%) incurred a delay greater than $\tau = 97\ ms$. Therefore, we conclude that although very rare, delay faults are observed in CPSs.

In order to demonstrate the fault-tolerance capabilities of Axo, we artificially reduce τ to $92\ ms$, thereby increasing the number of delay faults in the CPS. We also inject bursty delay faults and crash faults with different fault rates in the different controller replicas in T-RECS. The probabilities that C_1, C_2, and C_3 experience a delay fault are set as 0.02, 0.01 and 0.001, respectively. C_1 has no crash faults, whereas C_2 and C_3 have a crash fault with probabilities of 0.001 and 0.0001, respectively. Furthermore, the network is configured to have a loss probability of 0.01.

Figure 8.3 shows the ECDF of the delays of the controller replicas with and without Axo. We observe that the delays incurred by setpoints sent by the individual replicas, as measured at the maskers, is sometimes greater than $\tau = 92\ ms$. However, as seen in Chapter 5, the masker discards such delayed setpoints, so that the delay observed at the RAs is always less than the validity horizon, ensuring that the implemented setpoints are valid. We see that, in addition to ensuring that invalid setpoints are discarded, Axo decreases the overall delay perceived by the RAs, as it uses active replication and accepts the first valid setpoint in each control round. This is shown in the plot corresponding to the delay profile of Axo, which represents the delay profile of the first setpoint received in each control round.

We also measure the availability of each controller replica as the percentage of control rounds in which valid setpoints from that replica were received at both maskers. The availabilities of C_1, C_2, and C_3 were observed to be 81.31%, 89.37% and 91.31%, respectively. The loss of availability is due to the delay and crash faults incurred, in addition to the detection and recovery time for faulty replicas. The overall availability provided by Axo, measured as the percentage of control rounds in which valid setpoints from any replica are received at both maskers, was observed to be 99.97%. This highlights the improved availability brought about by active replication, in spite of the time required to detect and recover faulty replicas.

In the second experiment, we control a real-life on-campus microgrid via two controller replicas that are running on dedicated desktop computers, with off-the-shelf Scientific Linux version 7.1. The setup remains the same, with one battery and one PV resource. We observe similar results for safety and availability, as in the previous experiment. However, we take this opportunity to highlight some findings on the detection and recovery algorithms of Axo.

The experiment duration was 24 hours, during which the controller replicas were recovered 38 times, thereby demonstrating the importance of fault recovery in providing high availability. We measure the time taken to compute setpoints, at each replica, from the time of reception of the first advertisement in a control round to the time of issuing the setpoints. In order to quantify the overhead in the computation of setpoints by the controller due to the proposed detection and recovery algorithms, we repeat these measurements when the detection and recovery algorithms of Axo are not running. The reported measurements are reported at 95% confidence interval. We find that, during control rounds in which a replica is non-faulty, the mean computation time is $0.893 \pm 0.0004\ ms$ with detection and recovery, whereas it is $0.274 \pm 0.0001\ ms$ without. During control rounds in which a replica is faulty, the mean computation time is $14.18 \pm 4.05\ ms$ with detection and recovery, and $14.45 \pm 2.76\ ms$ without. As the additional delay brought about by these algorithms is sub-millisecond in the non-faulty case, we conclude that they do not affect the real-time path.

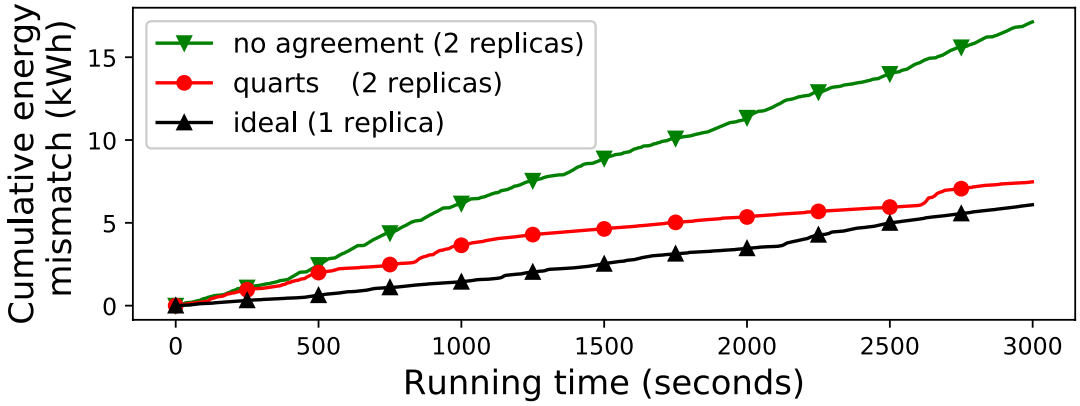


Figure 8.4 – Energy mismatch over time in COMMELEC with and without Quarts

8.4 Application of Quarts to COMMELEC

In the previous section, we have shown how Axo guarantees reliable validity by discarding invalid setpoints, and how it improves the delay profile and increases availability by using active replication of the controller. However, as discussed in Chapter 6, active replication without agreement might result in inconsistencies and, consequently, in the split-brain syndrome [121]. In this section, we show the undesirable effects, of such inconsistencies, on the underlying grid. Thus, we close the gap between the abstract concept of inconsistency and its possible effect on the control performance of a CPS for electric grids, in practice.

We perform an experiment in T-RECS, with two resources: a battery and a PV, as in the previous section. The COMMELEC GA is configured to assist in microgrid autonomy, *i.e.,* the power imported from, and exported to, the upper-level grid should be minimized at all times. This is equivalent to tracking a dispatch profile of zero Watts.

To highlight the effect of inconsistencies, we perform stress tests, in which we subject the controller replicas to artificial delays in computation and extreme network conditions of 10% loss rate. The aim is to understand the effect of inconsistencies on a CPS, without running experiments over several weeks, as would be required due to the low probability of inconsistency, as observed in Chapter 6.

We run three different scenarios in which we record the power at the slack bus and compute the energy mismatch in order to quantify the error in the control performance of COMMELEC. The energy mismatch, in this case, is the integral of the absolute value of the power profile, since the goal is to provide autonomy. We also compute the maximum deviation from zero power at any given point, in order to check for grid safety violations. The ampacity limit of the line connecting the microgrid to the main grid is equivalent to a power of 28 kW.

First, we benchmark the performance of a single non-replicated controller that is not exposed to delay faults or messages losses (ideal). Then, we compare this to two cases, one with two replicated controllers that do not perform agreement (no agreement), and one in which the two replicas perform agreement using Quarts (quarts).

We observe, in Figure 8.4, the energy mismatch of these three scenarios. The ideal case provides the benchmark tracking-performance level, and it encounters some mismatch due to the unpredictable nature of real-time PV production. We observe the performance improvement brought about by applying Quarts, as compared to the case with two non-agreeing replicas, throughout the entire duration of the experiment. This shows how Quarts closes the gap between the in-field performance and the ideal performance of COMMELEC.

Furthermore, we recorded the worst-case deviation from the tracking signal for each scenario. For the ideal scenario, the worst-case deviation was 25 kW. For the non-agreeing scenario, it was 33 kW: this exceeds the ampacity limit of the line connecting the microgrid to the upper-level grid, thereby violating grid safety. We note that a single replica would never compute setpoints that result in such a violation. The inconsistency between the replicas producing different outputs, coupled with the interleaving of setpoints, is what led to this violation. For the Quarts scenario, this violation is not observed, as the maximum deviation was recorded at 26 kW.

8.5 Islanding in COMMELEC

We now discuss the islanding operation in COMMELEC, a mission-critical operation that benefits from the aforementioned robustness and reliability properties. When a microgrid is connected to an upper-level grid, power imbalances can be exported to, or imported from, the main (upper-level) grid. Upon an intentional or emergency disconnection from the main grid, the microgrid is expected to continue working in *islanded mode*. During islanded mode, (at least) one resource needs to act as slack, *i.e.,* it must compensate for power variations to keep the power balance and ensure the security of supply.

COMMELEC, therefore, must maintain grid safety in either of the two modes of operation: grid-connected mode and islanded mode. The inner workings of COMMELEC is very similar in both modes of operation. Hence, the mechanisms for robustness and reliability, proposed in earlier chapters, can be applied in both modes. COMMELEC must also perform the transition maneuvers, successfully and safely, from one mode to another. There are three maneuvers to be performed, and these are described next.

The first is the *islanding maneuver*, also known as the disconnection maneuver.

Such a maneuver might be necessary, when the main grid experiences a fault, in order to isolate the microgrid. It might also be planned (intentional), in order to perform maintenance for example. During the islanding maneuver, the breaker at the PCC must be open, and one of the resources must be instructed to become slack.

During islanded mode, the chosen slack might lose its ability to maintain the power balance in the grid. For example, a battery resource acting as slack might fully discharge, thereby losing its ability to inject power into the grid. In such cases, an *islanded slack switching* maneuver must be performed, in which another resource is chosen to become slack. During the slack switching maneuver, the chosen resource is instructed to become slack, replacing the existing slack resource. The existing slack is, in turn, instructed to return to its original state, as a PQ resource that receives active and reactive power setpoints.

The third transition is the *reconnection maneuver*, which can be performed when the conditions that caused islanding to occur no longer hold. For example, after the grid fault is handled, in the case of unintentional islanding. During the reconnection maneuver, the existing slack resource is instructed to become a PQ resource, and the breaker at the PCC must be closed.

Although several resources might be eligible to become slack, some are more suitable than others (energy storage systems in particular) depending on the state of both the resources and the grid before the islanding transition. COMMELEC maintains a list of slack-candidate resources, ranked in real-time based on their suitability to become slack [23, 177].

In this section, we first present the architecture and components of COMMELEC required for islanding. Then, we present an overview of the operation of the slack ranking method in COMMELEC. Finally, we discuss the applicability of the reliability and robustness mechanisms, proposed throughout this thesis, to the slack ranking operation of COMMELEC. In the next section, we present our proposed protocol that performs the aforementioned three maneuvers, such that it maintains grid safety.

8.5.1 Architecture

Recall that the COMMELEC controller consists of a GA and an SE. As shown in Figure 8.5, in order to perform islanding, COMMELEC requires three additional components: the *slack ranker*, the *slack switcher*, and the *synchrocheck*. The slack ranker and the slack switcher are two sub-components of the GA. These are in addition to the main component of the GA that receives advertisements and SE states, performs computations, and issues setpoints. The synchrocheck is a separate device that is located at the PCC, and that can open or close the breaker connecting the microgrid to the main grid.

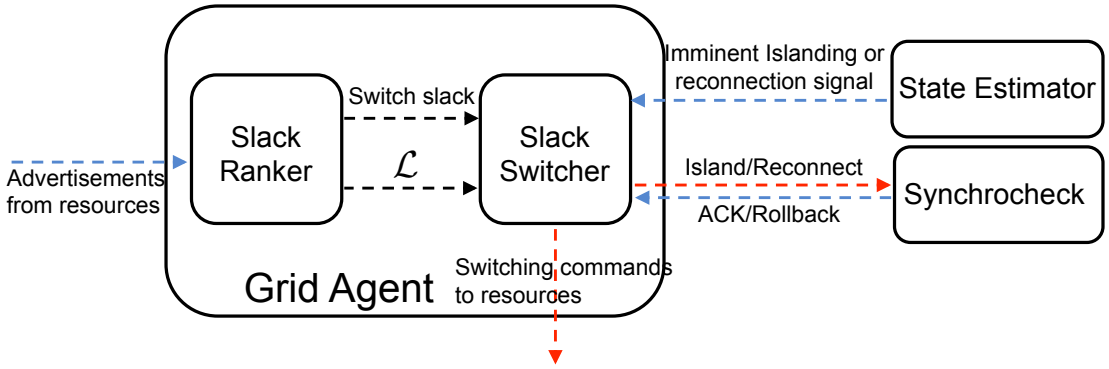


Figure 8.5 – Architecture and components of COMMELEC required for islanding. $\mathcal{L}$ is the ranked list of slack candidates.

In each control round, the slack ranker receives the set of advertisements to be used in the computation by the GA. Recall that the advertisements contain real-time information about the internal state of the resource. The slack ranker uses these advertisements to update a list $\mathcal{L}$ of slack candidates, ranked from best to worst, and it sends this list to the slack switcher. The slack ranker can also detect, by using the advertisements, when an existing slack resource has lost its ability to remain as slack. It informs the slack switcher when this occurs.

We consider that the synchrocheck sends measurements to the SE, hence acts as an asynchronous sensor. Being located at the PCC, the synchrocheck can sense an imminent islanding condition. It can also sense the absence of the condition that previously led to islanding. By sending this information to the SE, the SE can then inform the slack switcher of one of these two imminent conditions (islanding or reconnection).

The slack switcher receives, in each control round, an updated ranked list $\mathcal{L}$ of slack candidates. Additionally, as mentioned earlier, it might receive one of three signals: (1) an imminent islanding signal from the SE, which triggers the islanding maneuver, (2) an imminent reconnection signal from the SE, which triggers the reconnection maneuver, and (3) a slack switching signal from the slack ranker, which triggers the islanded slack switching maneuver. These maneuvers are described in Section 8.6.

During islanding or reconnection maneuvers, the slack switcher sends a signal to the synchrocheck informing it to open or close the breaker, at the PCC, respectively. The synchrocheck, then, performs this operation if the conditions at the PCC are suitable, in which case it replies with an acknowledgement (ACK) to the slack switcher. In case the conditions are not suitable, for example if the main grid can no longer be reconnected during a reconnection maneuver, the synchrochek sends a rollback signal to the slack switcher, informing it of the failure of the operation.

8.5.2 Slack Ranking

Ranking Metrics

The main goal of the slack ranker is to rank the microgrid resources in the order of their suitability to become slack. To do this, it uses the advertisement fields from each resource as input, and computes a set of metrics for each resource [23, 177]. These metrics are then used to rank the resources.

The metrics described in [23, 177] are the following: (1) power availability, (2) voltage feasibility, (3) current feasibility, (4) survival time, and (5) number of resources shed. In what follows, we describe the intuition behind each metric and briefly present how they can be computed.

As the slack resource must handle the power imbalances in the grid, its feasibility region (PQ profile) must include all the possible imbalances that might arise. Thus, the power availability metric is a measure of the safety margin, between the feasibility region of the resource and the maximum possible imbalance in the grid. Such a metric also captures the controllability of the resource, as uncontrollable resources in COMMELEC generally have a infinitesimally small feasibility region. The feasibility region of each resource is included in its advertisement, and the maximum possible imbalance can be computed by considering the uncertainty (belief function) of each resource, also included in each advertisement.

The rated power of the resource limits its power availability. Hence, fully controllable resources with a large rated power, such as supercapacitors, typically have a large power availability, thereby making them suitable slacks. However, the location of such resources might limit the power they can inject/absorb, as the voltage bounds at its bus, or the current limits at the line connecting it to the grid, might be violated. The voltage and current feasibility metrics provide a measure of the safety margins between the voltage and current bounds, and the voltages and currents resulting from appointing a given resource as slack, respectively. These can be computed by performing several load flow computations, by considering the extreme power injections in each resource.

A slack resource, ideally, must be capable of compensating for the power variations throughout the duration of the islanded mode operation. Such a duration is not known *a priori*. Therefore, we measure the survival time of a resource, as the duration after which it might cease to be a suitable slack candidate. Such a metric is computed by considering the rated energy of the resource, its current state-of-charge (SoC), and the maximum power imbalances it might have to inject/absorb in each control round. COMMELEC advertisements generally do not include the rated energy of the resource nor its SoC , as these are device-specific properties. However, for energy storage devices, advertisements are extended to include these two quantities that will only be used for slack ranking. The rated energy can alternatively be statically configured for each

resource, as it does not change in real-time.

The power variations in the grid might be too large for any resource to be able to handle. Hence, in order to maintain grid safety in islanded mode, the COMMELEC GA might be required to shed (disconnect) some resources upon performing the islanding maneuver. The last metric measures the number of resources shed, if a given resource is chosen as slack. Shedding fewer resources is favorable.

After computing these metrics, the slack ranker uses them to compute and output a list of possible slack candidates, ranked from best to worst. Slack candidates are resources whose metrics all exceed some minimum set thresholds. For example, resources whose survival time is less than 10 seconds are not considered as slack candidates. The remaining resources are ranked based on the metrics in the above listed order.

Thresholds for Slack Switching

During islanded mode, the existing slack resource might no longer be the best ranked resource in the slack candidate list $\mathcal{L}$. However, in order to avoid oscillations, COMMELEC does not perform slack switching in such cases. Instead, an existing slack is only replaced if one of its metrics falls below the minimum set thresholds.

In such cases, the slack ranker removes that resource from the list $\mathcal{L}$ and signals the slack switcher to perform the islanded slack switching maneuver. The slack switcher replaces the existing slack with the best ranked candidate resource in $\mathcal{L}$.

8.5.3 Applicability of Proposed Mechanisms to Slack Ranking

Here, we discuss how our proposed mechanisms for reliability and robustness can be applied for the slack ranking computation in COMMELEC.

Slack ranking is a side-computation in COMMELEC: it occurs alongside the computation of setpoints in each control round, and it takes as input the advertisements used to perform the setpoint computation. Therefore, intentionality clocks and Robuster can be easily applied. The label assigned to the list, issued by the slack ranker, is the same label of the setpoints computed in the same control round. Also, the same short-term and long-term advertisement fields used in the computation of the setpoints can be used to compute the slack candidate list. These mechanisms enable the computation of the list to be robust.

The slack list is not issued to RAs, hence fault masking is not required. However, the slack ranker might still incur delay and crash faults. These must be detected and subsequently recovered. We can use Axo to perform detection and recovery of delay

and crash faults in the slack ranker. In order to detect such faults, the slack ranker must send the latest label of the list it computes to the tagger in each control round. Thus, similar to the mechanism for detecting faults in the detector in Axo, faults in the slack ranker can be detected. Upon detection, the faulty slack ranker can be rebooted.

Finally, with active replication of the COMMELEC GA, the consistency between the slack rankers at the different replicas must be ensured. However, the input advertisements used to compute the ranked lists are the same as those used for computation of setpoints. Therefore, they are already agreed upon using Quarts, thereby guaranteeing consistency among the slack rankers. Applying both Axo and Quarts enables the computation of the slack candidate list to be reliable.

8.6 A Slack Switching Protocol for Islanding Maneuvers and Operation

The protocol performed by the slack switcher involves three main functions.

1. Islanding (disconnection) maneuver: disconnection of a grid-connected microgrid from the main grid.
2. Reconnection maneuver: reconnection of an islanded microgrid to the main grid.
3. Islanded slack switching: switching slacks during islanded operation.

These three functions involve switching the mode of operation of some resources in the microgrid. In Section 8.6.1, we describe the possible modes of operation of a resource, and we list the requirements of the slack switcher.

Although the three functions share a lot of common elements, in Sections 8.6.2-8.6.4 we describe each separately, while drawing on the parallels between them. Throughout, we highlight the design choices and the reasons behind making them.

Finally, in Section 8.6.5 we discuss the applicability of the proposed mechanisms, for reliability and robustness, to the proposed slack switching protocol.

8.6.1 Requirements

The resources in the grid operate in one of the following three modes:

- Grid-feeding mode: a grid-feeding resource is a PQ resource that follows the frequency it measures at its bus, and that is given active and reactive power

setpoints by the COMMELEC GA to implement. In grid-connected mode, all resources are in grid-feeding mode.

- Grid-forming mode: a grid-forming resource is a slack resource. It sets the voltage magnitude and frequency in the grid, as configured by the COMMELEC controller. It also compensates for the power imbalance at its bus in order to maintain the frequency.
- Grid-supporting mode: a grid-supporting resource, is also called a droop. It measures the power at its bus and changes the frequency accordingly. Its parameters are also configured by the COMMELEC controller.

In grid-connected mode, COMMELEC requires all resources to be grid-feeding. In this mode, the main grid acts as the slack, imposing the frequency, as it is generally characterized by higher inertia than the microgrid. In islanded mode, COMMELEC requires, at steady state, one resource to be in grid-forming mode, setting a fixed voltage magnitude and frequency, and all other resources to be in grid-feeding mode.

Grid-supporting mode is an intermediate state that can temporarily act as slack, albeit in degraded mode, as it does not set a fixed frequency. Multiple grid-supporting resources can co-exist in a microgrid, performing conventional droop control.

The configuration of the parameters for grid-forming and grid-supporting mode are not pertinent, hence their discussion is omitted. Instead, we focus on the protocol that orchestrates the switching of the modes during the three aforementioned maneuvers. The requirements of the protocol are the following:

1. At most one grid-forming resource at all times.
2. Zero grid-forming resources when grid-connected.
3. At least one grid-forming or grid-supporting resource at all times in islanded operation.
4. Minimize the time in which resources are in grid-supporting mode.
5. Minimize the number of mode changes in resources.

The intuition behind each of the aforementioned requirements is as follows. (1) Having multiple grid-forming resources, each imposing its own frequency on the grid, might lead to instability due to frequency fluctuations. Additionally, these resources are generally located at different buses in the grid, hence it is difficult to ensure that they impose the same frequency on the rest of the grid. (2) When grid-connected, the upper-level grid is imposing the frequency of the microgrid. Thus, having a grid-forming

resource would also lead to instability. (3) When islanded, at least one resource must be compensating for the power imbalance, be that a grid-forming or a grid-supporting resource. Coupled with the first requirement, this requirement permits multiple grid-supporting resources and at most one grid-forming resource. This is because multiple grid-supporting resources can co-exist together, and with a grid-forming resource, for a brief duration, without causing instability.

In addition to the strict first three requirements that must be guaranteed, the last two are best-effort. (4) This requirement ensures that grid-supporting resources will eventually be switched to grid-feeding or grid-forming. Hence, it eliminates multiple grid-supporting resources that might be present, thereby increasing the flexibility of the grid, and eliminating the chances of instability. (5) Excessive mode switches are undesirable as they cause unpredictable transients. Additionally, disturbing the operation of a resource might cause it to become unresponsive, in which case it would be shed.

8.6.2 Disconnection Maneuver

Here, we consider a grid-connected microgrid, with all k resources in grid-feeding, and in which the upper-level grid acts as the slack. The goal is to end up with an islanded microgrid with m ($\leq k$) resources: a single grid-forming resource and $m-1$ grid-feeding resources. The number of resources might decrease due to shedding performed during the islanding maneuver.

The protocol is presented is Algorithm 8.1. In addition to receiving a ranked list of slack-candidates from the slack ranker, the slack switcher also receives signals from the SE. In this case, it receives the "islanding" signal, with a timestamp T_0 signaling the latest time at which the microgrid will be islanded.

The best case proceeds as follows. We begin by selecting the top slack candidate from the list, and instructing it to switch from grid-feeding to grid-supporting mode. Then, the synchrocheck is instructed to preemptively island the microgrid. Finally, the grid-supporting resource is instructed to switch to grid-forming.

However, due to network non-idealities, and the possibility of delayed responsiveness from the resources (or the synchrocheck), several issues might arise. First, in order to account for transient network faults and partial unresponsive behavior, message retransmission is used. Every message is retransmitted once every τ milliseconds, until an acknowledgement is received, but at most n times. If a resource does not respond after n retransmissions, it is declared unresponsive, and corrective action is performed.

In Step 1, if the chosen candidate is unresponsive to the mode switch from grid-

Algorithm 8.1: Protocol for Islanding Maneuver

Upon receiving islanding(T_0) from SE:

$i = 0$
Step 1: Pick resource R_i from slack list
Switch R_i from grid-feeding to grid-supporting
if R_i is unresponsive **then**
    Shed resource R_i
    Increment i
    Repeat Step 1
end if

Step 2: Instruct syncrocheck to island microgrid
Wait until signal from synchrocheck or until T_0

Step 3: Switch R_i from grid-supporting to grid-forming
if R_i unresponsive **then**
    Perform Step 1 starting from R_{i+1}
    Shed resource R_i
    Repeat Step 3 for new grid-supporting resource
end if

feeding to grid-supporting, it is shed[1] (i.e. disconnected from the grid). Then, the next best candidate is selected and the step is repeated. This step is guaranteed to converge, either to a microgrid with one grid-supporting resource, or to a microgrid with all its resources shed. Note that the resource cannot be directly switched to grid-forming, as the microgrid is still connected to the upper-level grid.

In Step 2, if the synchrocheck does not perform the islanding immediately, we can wait until T_0 when islanding will occur in any case. After Step 2, the microgrid is islanded with a single grid-supporting resource acting as a slack. The microgrid cannot be islanded before Step 1 is performed, as this would violate the requirement of having at least one grid-forming or grid-supporting resources at all times in islanded mode.

In Step 3, if this resource is unresponsive to switching to grid-forming, the next best candidate is selected and switched to grid-supporting as in Step 1. Then, the original unresponsive resource is shed. At this point, the microgrid is back to having a single grid-supporting resource and the process is repeated. This process is also guaranteed to converge to a microgrid with one grid-forming resource, or to one with all its resources shed. The unresponsive grid-supporting resource cannot be shed before another resource is switched to grid-supporting, as that would leave the microgrid with no slack.

[1] Note that the load shedding discussed here is in addition to the load shedding that might be performed by the slack ranker to guarantee grid stability. The slack switcher does not know for certain what mode the unresponsive resource is in, so it must shed it.

The protocol is performed in such a way to fulfill the requirements presented in Section 8.6.1.

1. Only one resource is switched to grid-forming, specifically in Step 3. Thus, the microgrid has at most one grid-forming resource at all times.
2. Step 3 switches a resource to grid-forming after the microgrid is disconnected. Thus, the microgrid has zero grid-forming resources when grid-connected.
3. In islanded operation, *i.e.*, after Step 2, the microgrid has at least one grid-supporting resource until it is successfully switched to grid-forming.
4. A resource is only maintained in grid-supporting mode until the disconnection is successfully performed, after which it is promptly switched to grid-forming.
5. Two resource mode switches are performed, unless some resources are unresponsive. Thus, the number of mode switches is kept to a minimum.

8.6.3 Reconnection Maneuver

Here, we consider an islanded microgrid with one grid-forming resource and $k-1$ grid-feeding resources. The goal is to end up with a grid-connected microgrid with all $m\ (\leq k)$ resources in grid-feeding mode.

The protocol is presented is Algorithm 8.2. Again, we begin by analyzing the best case. Upon receiving a "reconnection" signal from the synchroncheck, the slack switcher instructs the grid-forming resource to become grid-supporting. Then, the synchrocheck is instructed to reconnect the microgrid. After receiving the success signal for reconnection, the grid-supporting resource is switched to grid-feeding.

The grid-forming resource might be unresponsive to switching to grid-supporting, which poses a problem, as we require a grid-supporting resource before reconnecting the main grid. In this case, we perform the first step of Algorithm 8.1, which will switch some grid-feeding resource to grid-supporting, and will take the priority list into account. After this step is successful, it is safe to shed the unresponsive grid-forming resource.

The reconnection maneuver might fail, in which case the synchrocheck will send a signal instructing us to rollback to an islanded microgrid with a grid-forming resource. Hence, it is required to switch a grid-supporting resource to grid-forming, which is similar to the third step of Algorithm 8.1.

In case the reconnection maneuver is successful, the grid-supporting resource must be switched to grid-feeding mode. If this resource is unresponsive, it is simply

Algorithm 8.2: Protocol for Reconnection Maneuver

Upon receiving reconnection from SE:

Step 1: Switch grid-forming resource R to grid-supporting
if R is unresponsive **then**
    Perform Step 1 of Algorithm 8.1
    R' is now grid-supporting
    Shed R
end if

Step 2: Instruct syncrocheck to reconnect microgrid
Wait until success or rollback signal from synchrocheck
if rollback signal received **then**
    Perform Step 3 of Algorithm 8.1 on R/R'
    Exit Algorithm
end if

Step 3: Switch R/R' from grid-supporting to grid-feeding
if R/R' unresponsive **then**
    Shed resource R/R'
end if

shed, resulting in a grid-connected microgrid with all the remaining resources in grid-feeding mode.

As in the previous section, it can be observed that the requirements listed in Section 8.6.1 are fulfilled.

8.6.4 Islanded Mode Operation: Slack Switching Maneuver

Finally, we consider an islanded microgrid with one grid-forming resource R and k grid-feeding resources. R can no longer sustain compensating for the power imbalance, and must be switched. The goal is to end up with one grid-forming resource $R' \neq R$, and the remaining $m \leq k$ resources in grid-feeding mode.

The protocol is presented in Algorithm 8.3.The first and third step are exactly the same as in Algorithm 8.1: the first selects the best-ranked slack candidate and switches it to grid-supporting, and the third switches that resource to grid-forming. The corrective actions are also the same in case of unresponsive resources.

The second step in Algorithm 8.1 involved disconnecting the microgrid from the main grid that was acting as slack. In this case, the second step switches the grid-forming slack resource to grid-feeding. It is shed if it is unresponsive.

Algorithm 8.3: Protocol for Slack Switching in Islanded Operation

Upon receiving switch() from slack ranker:

$i = 0$
Step 1: Pick resource R_i from slack list
Switch R_i from grid-feeding to grid-supporting
if R_i is unresponsive **then**
    Shed resource R_i
    Increment i
    Repeat Step 1
end if

Step 2: Switch R from grid-forming to grid-feeding
if R is unresponsive **then**
    Shed resource R
end if

Step 3: Switch R_i from grid-supporting to grid-forming
if R_i unresponsive **then**
    Perform Step 1 starting from R_{i+1}
    Shed resource R_i
    Repeat Step 3 for new grid-supporting resource
end if

8.6.5 Applicability of Proposed Mechanisms to Slack Switching

As in Section 8.5.3, here, we discuss the applicability of the proposed robustness and reliability mechanisms to the slack switching protocol.

The slack switching protocol involves a round-based communication with a subset of the resources. Therefore, intentionality clocks can be used to order the messages exchanged. Recall that the slack switcher receives a list $\mathcal{L}$ with a given label ℓ that represents the latest control round in COMMELEC. This label will be included in all messages in the slack switching protocol. Thus, the resources will not perform mode switches due to slack switching messages from previous rounds, which are no longer valid. An additional label can then be included to indicate the step of the process being performed, as in Algorithms 8.1-8.3. In this way, a resource that has performed Step 3 will not perform Step 1 again if it receives a retransmission or a duplicate message.

Robuster cannot be applied in this context. The messages received from the resources are simple acknowledgments, indicating the success of the switching operation. The slack switcher needs these messages in order to continue. Retransmission is, therefore, performed. Network reliability protocols, such as iPRP [52] and QUIC [48], discussed in Chapter 2, can also be used here to decrease the effect of network losses on the protocol.

Axo can be applied in the same way as discussed in Section 8.5.3. Recall that the slack ranker sends the computed ranked list $\mathcal{L}$, with a label, in each control round, to the slack switcher. The slack switcher, after processing the list, can send the label to the Axo tagger. This way, a fault in either component can be detected and recovered by Axo.

Finally, Quarts can be applied on the input acknowledgments received from the resources, in order to guarantee that the replicas consistently perform the steps of the corresponding protocol, thereby instructing the same resources to switch modes. At the beginning of Step 1, all replicas have the same input from the slack ranker and the SE, hence all perform Step 1 by sending the same message to the same resource. In subsequent steps, the controller replicas perform agreement on the received acknowledgement signals from resources and the received acknowledgment or rollback signal from the synchrocheck. By guaranteeing consistency at each step, consistency is guaranteed throughout the protocol. For example, after requesting a resource to switch modes, the replicas expect an acknowledgement from that resource. Therefore, Quarts has to wait for some time to receive that message, then agree on whether or not it was received. The following message sent will be a retransmission if it was agreed that the acknowledgement was not received. Otherwise, it would be the message corresponding to the next step in the protocol, if any.

8.7 Conclusion

In this chapter, we have presented COMMELEC [8], an existing framework for real-time control of electric grid. The COMMELEC GA sends explicit active and reactive power setpoints to the RAs, which implement them and reply with advertisements encapsulating the internal state of the resources they control. COMMELEC ensures that the grid is maintained in a feasible state and enables several ancillary services such as tracking a dispatch plan, providing frequency support to the upper-level main grid, and performing (intentional and unintentional) islanding and reconnection maneuvers.

We have presented the effect of communication network non-idealities on the performance of COMMELEC in a CPS for real-time control of electric grids. We have shown that an increase in packet loss probability results in additional deviation from the tracking signal and might result in violation of grid safety.

Table 8.2 provides a summary of the properties for robustness and reliability, presented in Chapter 3, in addition to the proposed mechanisms that achieve them. Recall that robust availability (Definition 3.5) and reliable availability (Definition 3.9) are not guaranteed by one specific mechanism. Rather, they are preserved by all applicable mechanisms.

Property	Mechanism	Impact
Robust ordering and reliable ordering (Definitions 3.3, 3.7)	Intentionality clocks	Total order of messages between software agents
Robust safety and robust optimality (Definitions 3.4, 3.6)	Robuster	Compute setpoints that maintain grid safety, despite missing advertisements
Reliable validity (Definition 3.8)	Axo	Delay and crash fault tolerance
Reliable consistency (Definition 3.10)	Quarts	Avoid the split-brain syndrome among replicated GAs

Table 8.2 – Summary of the proposed properties and the corresponding mechanisms that achieve them

We have shown the impact of applying the mechanisms proposed throughout this thesis on the performance of COMMELEC in grid-connected mode. Robuster and intentionality clocks combine to enable the design of a robust COMMELEC controller, thereby decreasing the effect of network non-idealities on the tracking performance. Axo and Quarts combine to enable the design of an actively replicated COMMELEC controller that is reliable, *i.e.*, it maintains grid safety and close to optimal tracking performance, in the presence of delay and crash faults affecting its replicas.

Then, we considered the operation of COMMELEC in islanded mode and in the transitions from grid-connected to islanded mode. We have presented the two components responsible for islanding in COMMELEC: the slack ranker and the slack switcher. The slack ranker computes a list of slack-candidate resources, ranked based on their suitability to become slack. Besides briefly presenting the design of the slack ranker, as proposed in [23, 177], we have discussed the applicability of the robustness and reliability mechanisms, proposed throughout this thesis, to the computation performed by the slack ranker. Thus, we have shown how to ensure that such a computation is performed robustly and reliably.

Finally, we designed a protocol for performing the islanding, reconnection, and slack switching maneuvers in COMMELEC. We have presented a set of requirements for such a protocol, and have shown that the design fulfills these requirements. We have also discussed how the robustness and reliability mechanisms apply in this context.

9 Conclusions and Directions for Future Work

We can only see a short distance ahead,
but we can see plenty there that needs to be done.
— Alan Turing

We have studied the problem of designing reliable and robust real-time CPSs, specifically in the context of real-time control of electric grids. To address the non-idealities in the communication network, such as message losses, delays, and reordering, we designed robustness mechanisms, called Robuster and intentionality clocks. These mechanisms combine to design a *robust* controller, *i.e.,* one that can continue to operate correctly in the presence of the aforementioned non-idealities. To address the faults that can affect software agents, such as delays and crashes, we designed reliability mechanisms: Axo and Quarts. Axo enables masking delay and crash faults in the controller from the rest of the CPS, by using active replication of the controller. Quarts ensures that controller replicas send consistent setpoints, thereby avoiding the split-brain syndrome [121]. These mechanisms combine to design a *reliable* controller, *i.e.,* one that can continue to operate correctly despite delay and crash faults.

We have presented a model of CPSs for real-time control of electric grids, which is composed of four layers. Using this model, we have formulated a set of requirements for designing robust and reliable CPSs. These requirements were the basis of the provably correct mechanisms proposed throughout the thesis.

Using the layering approach, we developed T-RECS, a virtual commissioning tool that enables testing CPSs for real-time control of electric grids in-silico, before field-deployment. Using T-RECS, we have studied the effect of network and software non-idealities on the performance of COMMELEC [8], a CPS for real-time control of grids. We have also shown, both through simulations in T-RECS and through experiments in a real-scale microgrid, the improvements in performance brought about by applying our mechanisms for robustness and reliability to COMMELEC.

In Chapter 3, we have decomposed a CPS for real-time control of electric grids into four layers: the physical layer, the sensing and actuation layer, the network layer, and the control layer. We have discussed the operation models of the components in each layer and have highlighted the interactions among them. We have also discussed the fault models considered for each of the components, enumerating the type of faults considered in this thesis as follows: (1) grid failures, such as line faults and electric resource shut down or unresponsiveness, (2) sensor and actuator delays and unresponsiveness, (3) communication network non-idealities, which include message losses, delays, and reordering, and (4) software agent crashes and delays. Other faults are not considered in this thesis, hence must be separately accounted for. In particular, Byzantine faults [39], which include erroneous computations by software agents and message contamination by the communication network, are not handled by our proposed mechanisms. These types of faults occur due to software bugs and cyber-security vulnerabilities [40], therefore can be reduced by software and system verification techniques [41, 43].

Also in Chapter 3, we have introduced and motivated a set of formal requirements for a reliable and robust CPS. In order to design a robust CPS controller, we have introduced robust ordering, safety, availability, and optimality (Definitions 3.3-3.6). A controller that guarantees robust ordering, safety, and optimality, and preserves robust availability is said to be a robust controller. In order to design a reliable CPS controller, we have introduced reliable ordering, validity, availability, and consistency (Definitions 3.7-3.10). A set of replicated controllers that guarantee reliable ordering, validity, and consistency, and preserve reliable availability are said to be reliable. These properties do not form a comprehensive list of requirements for the reliability and robustness of any CPS. Future work can extend these properties for specific classes of CPSs.

In Chapter 4, we have defined the notion of a CPS control round in the presence of controller replication. Then, we have proposed intentionality clocks, a mechanism that assigns labels to the messages exchanged between software agents in a CPS, such that each label corresponds to the round number each message belongs to. The intentionality clocks mechanism guarantees robust and reliable ordering. That is to say, for robust ordering, it guarantees that when the controllers compute setpoints for round r, they only use advertisements from round $r-1$. For reliable ordering, it guarantees that setpoints implemented by an RA belong to a round number higher than the last advertisement this RA issued. In addition to these guarantees, intentionality clocks preserves robust and reliable availability, whereby it only discards advertisements and setpoints that would otherwise violate robust and reliable ordering. One possible avenue to explore, for an extension of this work, is to consider CPSs with hierarchical controllers [144, 147]. This would require an analysis of the interdependence between the control rounds at the different hierarchies, in addition to an extension of the labeling mechanism to accommodate the new model.

Also in Chapter 4, we have proposed Robuster, a mechanism for generating advertisements at the RAs and for computing setpoints at the GA. Robuster guarantees robust safety and optimality, and it increases the number of rounds in which robust availability holds. Robuster augments the advertisements, issued by RAs, with long-term fields, *i.e.*, fields with a validity horizon that extends for the next n control rounds. These fields, in addition to the short-term fields that are valid for one control round, are used by the GA to compute setpoints. Advertisement fields are used only during rounds in which they are valid, thereby guaranteeing robust safety. Furthermore, short-term fields are always used when they are available, thereby guaranteeing robust optimality. As the number of control rounds in which a setpoint computation can be performed increases, robust availability holds in more control rounds in CPSs that implement Robuster. We have presented the formal requirements for generating long-term fields at an RA, and we have shown examples of generating such fields in RAs for several resources in the COMMELEC framework [8]. In future work, we could explore the idea of generating multiple sets of long-term fields, each valid for a different number of rounds, a technique similar to the receding horizon control [178].

In Chapter 5, we have proposed Axo, a protocol for delay- and crash-fault tolerance in real-time CPSs. Axo uses active replication of the controller, whereby multiple controller replicas simultaneously receive input, perform computations, and issue output setpoints. Active replication increases the probability of at least one non-delayed replica in a given control round. Axo guarantees reliable validity by discarding invalid setpoints before they are implemented at the RA. In order to preserve reliable availability, Axo only discards invalid setpoints. Axo also ensures the long-term reliability of a CPS by providing a mechanism for detecting and recovering from delay and crash faults. We have presented the design of Axo, have formally proven its guarantees, and have derived bounds on the recovery time from both delay and crash faults. We have highlighted the utility of Axo via a stability analysis of an inverted pendulum system. We have shown that Axo improves the stability of an inverted pendulum, in the presence of delay and crash faults. The detection algorithm in Axo uses exponential averaging, thereby declaring a delay fault when the rate of delays, in a given sliding window, is above a certain threshold. In future work, we could consider a method based on pattern recognition, in order to detect delay faults that might be missed by the exponential averaging method.

In Chapter 6, we have proposed Quarts, an agreement protocol for real-time CPSs. We have introduced the causes and effects of the split-brain syndrome [121], and have shown that traditional agreement mechanisms are not suitable for CPSs for real-time control of electric grids. We have highlighted a set of properties, generally exhibited in such CPSs, that enable developing a better tailored agreement protocol with a bounded low latency-overhead. We have described the design of the two phases of Quarts: collection and voting. Then, we have shown how Quarts can be applied to controllers of CPSs, in order to guarantee reliable consistency, thereby avoiding

the split-brain syndrome. We have performed extensive simulations of Quarts, in comparison with consensus-based and passive-replication mechanisms. Through these simulations we have reported several findings, as follows: (1) Quarts provides an availability that is several orders of magnitude higher than state-of-the-art agreement protocols, (2) the availability provided by Quarts increases with the number of replicas, as opposed to other protocols, (3) passive-replication schemes cannot guarantee consistency in the presence of delay faults, (4) Quarts has a lower average- and tail-latency than consistency-guaranteeing consensus-based schemes, and (5) providing consistency comes at the expense of a marginal increase in messaging cost. Quarts requires modifications to the controllers in order to guarantee consistency. Future work could explore the idea of designing Quarts as a layer that can be deployed alongside the controller, in a manner similar to the Axo controller library. This would facilitate the adoption of Quarts onto existing CPSs.

In Chapter 7, we have presented T-RECS, a virtual commissioning tool for software-based real-time control of electric grids. Designed entirely in software, T-RECS can be used to improve the efficiency of design-test cycles of CPSs, thereby assisting in their co-development. T-RECS was designed to follow the layering approach presented in Chapter 3. We have discussed three main requirements for such a tool: (1) the ability to induce non-idealities into the communication network and the software agents, (2) the ability to study the operation of the unmodified software agents, and (3) the ability to perform such simulations entirely in-silico, without the need for physical equipment. T-RECS simulates the electric grid and resources by using state-of-the-art resource models and a load-flow solver [139]. It emulates the communication network using Mininet [154], which also provides Linux containers [170] that can run the unmodified software agent code. We have validated the load-flow solver used in T-RECS, by comparing its results to measurements obtained from a measurement campaign in an in-house microgrid on campus. We have shown that the tail relative error is less than 0.1%, which validates that T-RECS can be used to study CPSs for electric grids in-silico. We have also evaluated the performance of T-RECS, in terms of CPU, memory usage, and load-flow computation time. T-RECS can run up to eight software agents on a standard laptop computer, and the state-of-the-art load-flow solver is faster than the Newton-Raphson method by up to two orders of magnitude. In future work, a key feature to incorporate in T-RECS would be virtual time. In its current form, T-RECS runs at a pace dictated by the software agents. Thus, in order to run a day-long experiment, T-RECS requires one day. Incorporating virtual time would enable bypassing the idle times of the software agents, without affecting their behavior. This would enable running long-term experiments at a faster pace than the one dictated by the software agents.

In Chapter 8, we have described COMMELEC [8], an existing framework for real-time control of electric grids which uses explicit power setpoints. We have described the architecture and operation of COMMELEC both in the grid-connected mode and

in the islanded mode. We have shown, via simulations in T-RECS, the effects of communication network and software non-idealities on the performance of COMMELEC, both in maintaining grid safety, and in tracking a given dispatch plan. This highlights the need for applying the robustness and reliability mechanisms, discussed in this thesis.

We have experimentally validated, both in T-RECS and in a real-scale microgrid, our mechanisms for robustness and reliability when applied to COMMELEC. In Chapter 4, we have shown the improvements brought about by applying intentionality clocks and Robuster to COMMELEC. By guaranteeing robust ordering and safety and preserving robust availability, these mechanisms provide a minimal root-mean-square error (RMSE) in a scenario for frequency support. Additionally, by guaranteeing robust optimality, these mechanisms maintain the ability to track a dispatch signal even under binding grid conditions. In Chapter 8, we have applied both Axo and Quarts to COMMELEC. We have shown that by guaranteeing reliable safety and consistency, these mechanisms maintain grid safety and improve the RMSE in dispatch plan tracking scenarios.

The work done in this thesis is complementary to the work done on improving the reliability, robustness, and resilience of the electric grid, *i.e.,* the physical layer. A potential avenue for future work is to develop a combined framework for analyzing the reliability and robustness of both the cyber and physical components in real-time control of electric grids. One work that goes in this direction is [179], in which they model the various layers of the CPS and assign probabilities of failure to events in each layer. This probabilistic study can be combined with a worst-case analysis to provide a comprehensive framework.

A Derivation of Axo Performance Analysis Results

A.1 Proof of Theorem 5.3: Delay-Faulty Controller

Delay-Faulty Controller: *In a CPS 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 CPS, 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.

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

Equation A.1 is based on the model of the controller described in Section 5.6: 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 5.5, 5.6, as shown in Theorem 5.3.

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

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

In a two-replica CPS, 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 $\Delta t = t_2 - (t_1 + d)$.

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

Equation A.3 can be obtained by modeling the process of receiving reboot messages (Algorithm 5.7) 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:

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

Equation A.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 A.2, A.3, A.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 CPSs 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:

$$\mathbb{P}_d^u(t) = 1 - (1 - \mathbb{P}_1(t))^{g-1}$$
$$\mathbb{P}_1(t) = \int_{t_1=0}^{t} \mathbb{P}_{det}(t_1) \int_{t_2=t_1}^{t} \mathbb{P}_{rec}(t_2 - t_1) dt_2 dt_1$$

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

A.2 Proof of Theorem 5.4: Crash-Faulty Controller

Crash-Faulty Controller: *In a CPS 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 $(\mathbb{P}_c^l(t))$ and upper bound $(\mathbb{P}_c^u(t))$ 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 issues a setpoint at $t_0 > t_a - \tau_c$, the report of which is received by C_i. The probability of such an event, given that C_0 issues another setpoint at t_a and that C_i is non-faulty throughout, is given as $\mathbb{P}_a(\Delta t)$, where $\Delta t = t_0 - t_a$:

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

Equation A.5 considers the controller model from Section 5.6, and uses the time-reversal property of Poisson processes.

We now consider a g-replica CPS, 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 $\Delta t = t_1 - t_0 - d$.

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

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 A.5 and A.6. We will also use $\mathbb{P}_r$ and $\mathbb{P}_{nf}$ from Equations A.3 and A.4, respectively.

The conditions for lower and upper bound are similar to those presented in Appendix A.1. For a lower bound, we consider a two-replica CPS, the worst-case network delay, and that a faulty replica cannot help in detection and recovery.

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

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:

$$\begin{aligned}\mathbb{P}_c^u(t) &= 1-(1-\mathbb{P}_3(t))^{g-1} \\ \mathbb{P}_3(t) &= \int_{t_0=\max(0,\tau_c-t)}^{\tau_c} \int_{t_1=\tau_c-t_0}^{t} \int_{t_2=t_1}^{t} \mathbb{P}_a(t_0)\mathbb{P}_c(t1+t_0,g)\times\mathbb{P}_r(t_2-t_1)dt_2dt_1dt_0\end{aligned}$$

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

Bibliography

[1] W. H. Wolf, "Cyber-Physical Systems," *IEEE Computer*, vol. 42, no. 3, pp. 88–89, 2009.

[2] K. Jo, J. Kim, D. Kim, C. Jang, and M. Sunwoo, "Development of Autonomous Car—Part II: A Case Study on the Implementation of an Autonomous Driving System Based on Distributed Architecture," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 8, pp. 5119–5132, Aug 2015.

[3] C. Berger and B. Rumpe, "Autonomous Driving—5 Years After the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System," *arXiv preprint arXiv:1409.0413*, 2014.

[4] J. Michniewicz and G. Reinhart, "Cyber-Physical Robotics—Automated Analysis, Programming and Configuration of Robot Cells Based on Cyber-Physical-Systems," *Procedia Technology*, vol. 15, pp. 566–575, 2014.

[5] J. Fink, A. Ribeiro, and V. Kumar, "Robust Control for Mobility and Wireless Communication in Cyber-Physical Systems with Application to Robot Teams," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 164–178, 2012.

[6] L. Wang, M. Törngren, and M. Onori, "Current Status and Advancement of Cyber-Physical Systems in Manufacturing," *Journal of Manufacturing Systems*, vol. 37, pp. 517–527, 2015.

[7] L. Monostori, "Cyber-Physical Production Systems: Roots, Expectations and R&D Challenges," *Procedia CIRP*, vol. 17, pp. 9–13, 2014.

[8] A. Bernstein, L. Reyes-Chamorro, J.-Y. Le Boudec, and M. Paolone, "A Composable Method for Real-Time Control of Active Distribution Networks with Explicit Power Setpoints. Part I: Framework," *Electric Power Systems Research*, vol. 125, pp. 254–264, 2015.

[9] K. Christakou, D.-C. Tomozei, J.-Y. Le Boudec, and M. Paolone, "GECN: Primary Voltage Control for Active Distribution Networks Via Real-Time Demand-Response," *IEEE Transactions on Smart Grid*, vol. 5, no. 2, pp. 622–631, 2014.

[10] Y. Tang, K. Dvijotham, and S. Low, "Real-Time Optimal Power Flow," *IEEE Transactions on Smart Grid*, vol. 8, no. 6, pp. 2963–2973, 2017.

[11] J. Martin, *Programming Real-Time Computer Systems*. Prentice Hall, 1965.

[12] 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," *IEEE Transactions on Power Systems*, vol. 20, no. 4, pp. 1922–1928, Nov 2005.

[13] O. Ardakanian, C. Rosenberg, and S. Keshav, "Distributed Control of Electric Vehicle Charging," in *Proceedings of the fourth international conference on Future energy systems*. ACM, 2013, pp. 101–112.

[14] S. Deilami, A. S. Masoum, P. S. Moses, and M. A. Masoum, "Real-Time Coordination of Plug-in Electric Vehicle Charging in Smart Grids to Minimize Power Losses and Improve Voltage Profile," *IEEE Transactions on Smart Grid*, vol. 2, no. 3, pp. 456–467, 2011.

[15] IEEE Power and Energy Society, "IEEE Guide for Voltage Sag Indices," 2014.

[16] International Electrotechnical Commission, "International Standard: Electric Cables – Calculation of the Current Rating," 2007.

[17] IEEE Power Engineering Society, "IEEE Standard for Calculating the Current-Temperature of Bare Overhead Conductors," 2006.

[18] H. Saboori, M. Mohammadi, and R. Taghe, "Virtual Power Plant (VPP), Definition, Concept, Components and Types," in *Power and Energy Engineering Conference (APPEEC), 2011 Asia-Pacific*. IEEE, 2011, pp. 1–4.

[19] A. Oudalov, D. Chartouni, and C. Ohler, "Optimizing a Battery Energy Storage System for Primary Frequency Control," *IEEE Transactions on Power Systems*, vol. 22, no. 3, pp. 1259–1266, 2007.

[20] M. Pignati, L. Zanni, P. Romano, R. Cherkaoui, and M. Paolone, "Fault Detection and Faulted Line Identification in Active Distribution Networks Using Synchrophasors-Based Real-Time State Estimation," *IEEE Transactions on Power Delivery*, vol. 32, no. 1, pp. 381–392, 2017.

[21] P. Palensky and D. Dietrich, "Demand Side Management: Demand Response, Intelligent Energy Systems, and Smart Loads," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 3, pp. 381–388, 2011.

[22] P. C. Loh, Y. K. Chai, D. Li, and F. Blaabjerg, "Autonomous Operation of Distributed Storages in Microgrids," *IEEE Transactions on Power Electronics*, vol. 7, no. 1, pp. 23–30, 2014.

[23] A. Bernstein, J.-Y. Le Boudec, L. Reyes-Chamorro, and M. Paolone, "Real-Time Control of Microgrids with Explicit Power Setpoints: Unintentional Islanding," in *PowerTech*. IEEE, 2015, pp. 1–6.

[24] C. Chen, J. Wang, F. Qiu, and D. Zhao, "Resilient Distribution System by Microgrids Formation After Natural Disasters," *IEEE Transactions on Smart Grid*, vol. 7, no. 2, pp. 958–966, 2016.

[25] S. Barot and J. A. Taylor, "A Concise, Approximate Representation of a Collection of Loads Described by Polytopes," *International Journal of Electrical Power & Energy Systems*, vol. 84, pp. 55–63, 2017.

[26] O. Sundstrom and C. Binding, "Flexible Charging Optimization for Electric Vehicles Considering Distribution Grid Constraints," *IEEE Transactions on Smart Grid*, vol. 3, no. 1, pp. 26–37, 2012.

[27] W. K. Chai, N. Wang, K. V. Katsaros, G. Kamel, G. Pavlou, S. Melis, M. Hoefling, B. Vieira, P. Romano, S. Sarri *et al.*, "An Information-Centric Communication Infrastructure for Real-Time State Estimation of Active Distribution Networks," *IEEE Transactions on Smart Grid*, vol. 6, no. 4, pp. 2134–2146, 2015.

[28] L. Zanni, S. Sarri, M. Pignati, R. Cherkaoui, and M. Paolone, "Probabilistic Assessment of the Process-Noise Covariance Matrix of Discrete Kalman Filter State Estimation of Active Distribution Networks," in *Probabilistic Methods Applied to Power Systems (PMAPS), 2014 International Conference on*. IEEE, 2014, pp. 1–6.

[29] W. Bolton, *Programmable Logic Controllers*. Newnes, 2015.

[30] T. le Fevre Kristensen, R. L. Olsen, J. G. Rasmussen, and H.-P. Schwefel, "Information Access for Event-Driven Smart Grid Controllers," *Sustainable Energy, Grids and Networks*, vol. 13, pp. 78–92, 2018.

[31] C. Lo and N. Ansari, "Decentralized Controls and Communications for Autonomous Distribution Networks in Smart Grid," *IEEE Transactions on Smart Grid*, vol. 4, no. 1, pp. 66–77, March 2013.

[32] D. J. Reifer, V. R. Basili, B. W. Boehm, and B. Clark, "COTS-Based Systems–Twelve Lessons Learned about Maintenance," in *COTS-Based Software Systems*. Springer, 2004, pp. 137–145.

[33] National Instruments, "NI CompactRIO," http://www.ni.com/compactrio/, Accessed: 2018-10-07.

[34] B&R Automation, "Automation PC 910," https://www.br-automation.com/en/products/industrial-pcs/automation-pc-910/, Accessed: 2018-10-07.

[35] PINE64, "ROCKPRO64," https://www.pine64.org/?page_id=61454/, Accessed: 2018-10-07.

[36] M. Barabanov and V. Yodaiken, "Real-Time Linux," *Linux journal*, vol. 23, no. 4.2, p. 1, 1996.

[37] C. Wang, A. Bernstein, J.-Y. Le Boudec, and M. Paolone, "Existence and Uniqueness of Load-Flow Solutions in Three-Phase Distribution Networks," *IEEE Transactions on Power Systems*, vol. 32, no. 4, pp. 3319–3320, 2017.

[38] D. Dzung, R. Guerraoui, D. Kozhaya, and Y.-A. Pignolet, "Never Say Never–Probabilistic and Temporal Failure Detectors," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 679–688.

[39] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[40] F. Aloul, A. Al-Ali, R. Al-Dalky, M. Al-Mardini, and W. El-Hajj, "Smart Grid Security: Threats, Vulnerabilities and Solutions," *International Journal of Smart Grid and Clean Energy*, vol. 1, no. 1, pp. 1–6, 2012.

[41] P. Nilsson, O. Hussien, A. Balkan, Y. Chen, A. D. Ames, J. W. Grizzle, N. Ozay, H. Peng, and P. Tabuada, "Correct-by-Construction Adaptive Cruise Control: Two Approaches," *IEEE Transactions on Control Systems Technology*, vol. 24, no. 4, pp. 1294–1307, 2016.

[42] S. Bliudze, A. Cimatti, M. Jaber, S. Mover, M. Roveri, W. Saab, and Q. Wang, "Formal Verification of Infinite-State BIP Models," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2015, pp. 326–343.

[43] T. T. Tesfay, J.-P. Hubaux, J.-Y. Le Boudec, and P. Oechslin, "Cyber-Secure Communication Architecture for Active Power Distribution Networks," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 2014, pp. 545–552.

[44] Taskforce C6.04.02, "Benchmark Systems for Network Integration of Renewable and Distributed Energy Resources," CIGRÉ, Tech. Rep., 2010.

[45] J. Postel, "Transmission Control Protocol," Internet Requests for Comments, RFC Editor, STD 7, September 1981. [Online]. Available: http://www.rfc-editor.org/rfc/rfc793.txt

[46] J. C. R. Bennett, C. Partridge, and N. Shectman, "Packet Reordering is Not Pathological Network Behavior," *IEEE/ACM Transactions on Networking*, vol. 7, no. 6, pp. 789–798, Dec 1999.

[47] M. Scharf and S. Kiesel, "Head-of-line Blocking in TCP and SCTP: Analysis and Measurements," in *GLOBECOM*, vol. 6, 2006, pp. 1–5.

[48] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-transport-15, Oct 2018, Work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-15

[49] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar *et al.*, "The QUIC Transport Protocol: Design and Internet-Scale Deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 183–196.

[50] H. Kirrmann, M. Hansson, and P. Muri, "IEC 62439 PRP: Bumpless Recovery for Highly Available, Hard Real-Time Industrial Networks," in *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, Sept 2007, pp. 1396–1399.

[51] H. Kirrmann, K. Weber, O. Kleineberg, and H. Weibel, "HSR: Zero Recovery Time and Low-cost Redundancy for Industrial Ethernet (High Availability Seamless Redundancy, IEC 62439-3)," in *Proceedings of the 14th IEEE International Conference on Emerging Technologies & Factory Automation*, ser. ETFA'09. Piscataway, NJ, USA: IEEE Press, 2009.

[52] M. Popovic, M. Mohiuddin, D. C. Tomozei, and J. Y. L. Boudec, "iPRP - the Parallel Redundancy Protocol for IP Networks: Protocol Design and Operation," *IEEE Transactions on Industrial Informatics*, vol. PP, no. 99, pp. 1–1, 2016.

[53] M. Pignati, M. Popovic, S. Barreto, R. Cherkaoui, G. D. Flores, J.-Y. Le Boudec, M. Mohiuddin, M. Paolone, P. Romano, S. Sarri *et al.*, "Real-time State Estimation of the EPFL-Campus Medium-Voltage Grid by Using PMUs," in *Innovative Smart Grid Technologies Conference (ISGT), 2015 IEEE Power & Energy Society*. IEEE, 2015, pp. 1–5.

[54] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer, *The Time-Triggered Ethernet (TTE) Design*. IEEE, 2005.

[55] F. Consortium *et al.*, "FlexRay Communications System-Protocol Specification," *Version*, vol. 2, no. 1, pp. 198–207, 2005.

[56] M. Farsi, K. Ratcliff, and M. Barbosa, "An Overview of Controller Area Network," *Computing & Control Engineering Journal*, vol. 10, no. 3, pp. 113–120, 1999.

[57] M. Suchara, D. Xu, R. Doverspike, D. Johnson, and J. Rexford, "Network Architecture for Joint Failure Recovery and Traffic Engineering," in *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '11)*. ACM, 2011, pp. 97–108.

[58] C. Sousa, G. e Souza, I. Moraes, R. C. Carrano, C. V. N. de Albuquerque, L. F. N. Passos, A. Carniato, A. L. Bettiol, R. Z. Homma, R. C. Andrade *et al.*, "Link Quality Wstimation for AMI," in *Innovative Smart Grid Technologies Latin America (ISGT LATAM), 2015 IEEE PES*. IEEE, 2015, pp. 646–649.

[59] A. Lucas and S. Chondrogiannis, "Smart Grid Energy Storage Controller for Frequency Regulation and Peak Shaving, Using a Vanadium Redox Flow Battery," *International Journal of Electrical Power & Energy Systems*, vol. 80, pp. 26–36, 2016.

[60] S. J. Crocker and J. L. Mathieu, "Adaptive State Estimation and Control of Thermostatic Loads for Real-Time Energy Balancing," in *American Control Conference (ACC), 2016*. IEEE, 2016, pp. 3557–3563.

[61] W. M. H. Heemels, A. R. Teel, N. Van de Wouw, and D. Nesic, "Networked Control Systems with Communication Constraints: Tradeoffs Between Transmission Intervals, Delays and Performance," *IEEE Transactions on Automatic Control*, vol. 55, no. 8, pp. 1781–1796, 2010.

[62] Z. Wang, F. Yang, D. W. C. Ho, and X. Liu, "Robust Finite-Horizon Filtering for Stochastic Systems with Missing Measurements," *IEEE Signal Processing Letters*, vol. 12, no. 6, pp. 437–440, June 2005.

[63] Y. Shi and H. Fang, "Kalman Filter-based Identification for Systems with Randomly Missing Measurements in a Network Environment," *International Journal of Control*, vol. 83, no. 3, pp. 538–551, 2010.

[64] A. Gomez-Exposito, A. Abur, A. de la Villa Jaen, and C. Gomez-Quiles, "A Multilevel State Estimation Paradigm for Smart Grids," *Proceedings of the IEEE*, vol. 99, no. 6, pp. 952–976, June 2011.

[65] B. Sinopoli, L. Schenato, M. Franceschetti, K. Poolla, M. I. Jordan, and S. S. Sastry, "Kalman Filtering with Intermittent Observations," *IEEE Transactions on Automatic Control*, vol. 49, no. 9, pp. 1453–1464, Sept 2004.

[66] S. Shakkottai, R. Srikant, and N. B. Shroff, "Unreliable Sensor Grids: Coverage, Connectivity and Diameter," *Ad Hoc Networks*, vol. 3, no. 6, pp. 702–716, 2005.

[67] K. Chakrabarty, S. S. Iyengar, H. Qi, and E. Cho, "Grid Coverage for Surveillance and Target Location in Distributed Sensor Networks," *IEEE Transactions on Computers*, vol. 51, no. 12, pp. 1448–1453, 2002.

[68] N. Xia, H. B. Gooi, S. Chen, and M. Wang, "Redundancy based PMU Placement in State Estimation," *Sustainable Energy, Grids and Networks*, vol. 2, pp. 23–31, 2015.

[69] J. Chen and A. Abur, "Placement of PMUs to Enable Bad Data Detection in State Estimation," *IEEE Transactions on Power Systems*, vol. 21, no. 4, pp. 1608–1615, 2006.

[70] B. Li, Y. Ma, T. Westenbroek, C. Wu, H. Gonzalez, and C. Lu, "Wireless Routing and Control: a Cyber-Physical Case Study," in *Cyber-Physical Systems (ICCPS), 2016 ACM/IEEE 7th International Conference on*. IEEE, 2016, pp. 1–10.

[71] A. Bemporad, A. Casavola, and E. Mosca, "Nonlinear Control of Constrained Linear Systems via Predictive Reference Management," *IEEE transactions on Automatic Control*, vol. 42, no. 3, pp. 340–349, 1997.

[72] A. Bemporad, "Predictive Control of Teleoperated Constrained Systems with Unbounded Communication Delays," in *Proceedings of the 37th IEEE Conference on Decision and Control (Cat. No.98CH36171)*, vol. 2, Dec 1998, pp. 2133–2138 vol.2.

[73] L. Xie, Y. Gu, A. Eskandari, and M. Ehsani, "Fast MPC-Based Coordination of Wind Power and Battery Energy Storage Systems," *Journal of Energy Engineering*, vol. 138, no. 2, pp. 43–53, 2012.

[74] I. Koutsopoulos, T. G. Papaioannou, V. Hatzi *et al.*, "Modeling and Optimization of the Smart Grid Ecosystem," *Foundations and Trends® in Networking*, vol. 10, no. 2-3, pp. 115–316, 2016.

[75] F. Sossan, E. Namor, R. Cherkaoui, and M. Paolone, "Achieving the Dispatchability of Distribution Feeders Through Prosumers Data Driven Forecasting and Model Predictive Control of Electrochemical Storage," *IEEE Transactions on Sustainable Energy*, vol. 7, no. 4, pp. 1762–1777, 2016.

[76] E. Stai, L. Reyes-Chamorro, F. Sossan, J.-Y. Le Boudec, and M. Paolone, "Dispatching Stochastic Heterogeneous Resources Accounting for Grid and Battery Losses," *IEEE Transactions on Smart Grid*, 2017.

[77] P. Patrinos, S. Trimboli, and A. Bemporad, "Stochastic MPC for Real-Time Market-Based Optimal Power Dispatch," in *Decision and Control and European Control Conference (CDC-ECC), 2011 50th IEEE Conference on*. IEEE, 2011, pp. 7111–7116.

[78] D. L. Mills, "Internet Time Synchronization: The Network Time Protocol," *IEEE Transactions on Communications*, vol. 39, no. 10, pp. 1482–1493, 1991.

[79] I. Instrumentation and M. Society, "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," *IEEE Std 1588-2002*, pp. i–144, 2002.

[80] D. W. Allan and M. A. Weiss, "Accurate Time and Frequency Transfer During Common-View of a GPS Satellite," in *34th Annual Symposium on Frequency Control*, May 1980, pp. 334–346.

[81] L. Lamport, "Time Clocks and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[82] A. Armenia and J. H. Chow, "A Flexible Phasor Data Concentrator Design Leveraging Existing Software Technologies," *IEEE Transactions on Smart Grid*, vol. 1, no. 1, pp. 73–81, 2010.

[83] C. J. Fidge, "Timestamps in Message-Passing Systems that Preserve the Partial Ordering," in *Proceedings of 11th Australian Computer Science Conference*. Australian National University. Department of Computer Science, Feb 1988, pp. 56–66.

[84] M. Castro, B. Liskov *et al.*, "Practical Byzantine Fault Tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[85] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "CheapBFT: Resource-Efficient Byzantine Fault Tolerance," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 295–308.

[86] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet, "ZZ and the Art of Practical BFT Execution," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 123–138.

[87] P. Veríssimo and A. Casimiro, "The Timely Computing Base Model and Architecture," *IEEE Transactions on Computers*, vol. 51, no. 8, pp. 916–930, 2002.

[88] 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*, 2002, pp. 27–36.

[89] H. Kopetz and G. Grunsteidl, "TTP – A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems," in *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*. IEEE, 1993, pp. 524–533.

[90] H. Kopetz and G. Bauer, "The Time-Triggered Architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.

[91] A. Casimiro and P. Veríssimo, "Timing Failure Detection with a Timely Computing Base," in *3rd European Research Seminar on Advances in Distributed Systems (ERSADS'99)*. Department of Informatics, University of Lisbon, 1999.

[92] H. Kopetz, "Fault Containment and Error Detection in the Time-Triggered Architecture," in *Autonomous Decentralized Systems, 2003. The Sixth International Symposium on*. IEEE, 2003, pp. 139–146.

[93] K. Vaidyanathan and K. S. Trivedi, "A Comprehensive Model for Software Rejuvenation," *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 2, pp. 124–137, 2005.

[94] T. Maniak, C. Jayne, R. Iqbal, and F. Doctor, "Automated Intelligent System for Sound Signalling Device Quality Assurance," *Information Sciences*, vol. 294, pp. 600–611, 2015.

[95] T. Maniak, R. Iqbal, F. Doctor, and C. Jayne, "Automated Sound Signalling Device Quality Assurance Tool for Embedded Industrial Control Applications," in *Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on*. IEEE, 2013, pp. 4812–4818.

[96] R. Guerraoui and L. Rodrigues, *Introduction to Reliable Distributed Programming*. Springer Science & Business Media, 2006.

[97] A. Avizienis, J.-C. Laprie, B. Randell *et al.*, *Fundamental Concepts of Dependability*. University of Newcastle upon Tyne, Computing Science, 2001.

[98] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[99] A. Schiper, "Early Consensus in an Asynchronous System with a Weak Failure Detector," *Distributed Computing*, vol. 10, no. 3, pp. 149–157, 1997.

[100] H. Howard, M. Schwarzkopf, A. Madhavapeddy, and J. Crowcroft, "Raft Refloated: Do We Have Consensus?" *SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 12–21, Jan. 2015.

[101] F. J. Meyer and D. K. Pradhan, "Consensus with Dual Failure Modes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 2, pp. 214–222, 1991.

[102] L. Lamport *et al.*, "Paxos Made Simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[103] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos Made Live: an Engineering Perspective," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM, 2007, pp. 398–407.

[104] L. Lamport, "Fast Paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[105] H. Howard, D. Malkhi, and A. Spiegelman, "Flexible Paxos: Quorum Intersection Revisited," in *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 70, 2017, pp. 25:1–25:14.

[106] A. L. Hopkins, T. B. Smith, and J. H. Lala, "FTMP – A Highly Reliable Fault-Tolerant Multiprocess for Aircraft," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1221–1239, 1978.

[107] F. B. Schneider, "The State Machine Approach: A Tutorial," in *Fault-tolerant distributed computing*. Springer, 1990, pp. 18–41.

[108] M. Chérèque, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron, "Active Replication in Delta-4," in *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*. IEEE, 1992, pp. 28–37.

[109] R. Baldoni, C. Marchetti, and S. T. Piergiovanni, "Asynchronous Active Replication in Three-Tier Distributed Systems," in *Dependable Computing, 2002. Proceedings. 2002 Pacific Rim International Symposium on*. IEEE, 2002, pp. 19–26.

[110] T. D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996.

[111] R. Van Renesse, Y. Minsky, and M. Hayden, "A Gossip-Style Failure Detection Service," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer-Verlag, 2009, pp. 55–70.

[112] M. K. Aguilera, W. Chen, and S. Toueg, "Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication," in *International Workshop on Distributed Algorithms*. Springer, 1997, pp. 126–140.

[113] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot–A Technique for Cheap Recovery," *arXiv preprint cs/0406005*, 2004.

[114] B. Schröder, *Ordered Sets: An Introduction with Connections from Combinatorics to Topology*. Birkhäuser, 2016.

[115] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert, "Proactive Management of Software Aging," *IBM Journal of Research and Development*, vol. 45, no. 2, pp. 311–332, 2001.

[116] M. Castro and B. Liskov, "Proactive recovery in a byzantine-fault-tolerant system," in *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*. USENIX Association, 2000, p. 19.

[117] D. L. Parnas, "Software Aging," in *Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press, 1994, pp. 279–287.

[118] Y. Bao, X. Sun, and K. S. Trivedi, "A Workload-Based Analysis of Software Aging, and Rejuvenation," *IEEE Transactions on Reliability*, vol. 54, no. 3, pp. 541–548, 2005.

[119] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Software Aging Analysis of the Linux Operating System," in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*. IEEE, 2010, pp. 71–80.

[120] S. B. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in a Partitioned Network: A Survey," *ACM Computer Surveys*, vol. 17, no. 3, pp. 341–370, Sep 1985.

[121] Wikipedia, "Split-Brain (Computing)," Accessed: 2018-10-17. [Online]. Available: https://en.wikipedia.org/wiki/Split-brain_(computing)

[122] S. Poledna, "Replica Determinism in Distributed Real-Time Systems: A Brief Survey," *Real-Time Systems*, vol. 6, no. 3, pp. 289–316, 1994.

[123] B. M. Oki and B. 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*. ACM, 1988, pp. 8–17.

[124] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The Primary-Backup Approach," *Distributed systems*, vol. 2, pp. 199–216, 1993.

[125] H. Zou and F. Jahanian, "Real-Time Primary-Backup (RTPB) Replication with Temporal Consistency Guarantees," in *Distributed Computing Systems, 1998. Proceedings. 18th International Conference on*. IEEE, 1998, pp. 48–56.

[126] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-Performance Broadcast for Primary-Backup Systems," in *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*. IEEE, 2011, pp. 245–256.

[127] D. Li, P. Morton, T. Li, and B. Cole, "Cisco Hot Standby Router Protocol (HSRP)," in *RFC 2281*, 1998.

[128] S. Rizwan, V. Khurana, and G. Taneja, "Reliability Analysis of a Hot Standby Industrial System," *International Journal of Modelling & Simulation*, vol. 30, no. 3, p. 315, 2010.

[129] L. Xing, O. Tannous, and J. Bechta Dugan, "Reliability Analysis of Nonrepairable Cold-Standby Systems Using Sequential Binary Decision Diagrams," *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, vol. 42, no. 3, pp. 715–726, 2012.

[130] E. Scolari, D. Torregrossa, J.-Y. Le Boudec, and M. Paolone, "Ultra-Short-Term Prediction Intervals of Photovoltaic AC Active Power," in *Probabilistic Methods Applied to Power Systems (PMAPS), 2016 International Conference on*. IEEE, 2016, pp. 1–8.

[131] E. Scolari, F. Sossan, and M. Paolone, "Irradiance Prediction Intervals for PV Stochastic Generation in Microgrid Applications," *Solar Energy*, vol. 139, pp. 116–129, 2016.

[132] G. Welch and G. Bishop, "An Introduction to the Kalman Filter," Department of Computer Science, University of North Carolina at Chapel Hill, Tech. Rep., 1995.

[133] E. N. Gilbert, "Capacity of a Burst-Noise Channel," *Bell Labs Technical Journal*, vol. 39, no. 5, pp. 1253–1265, 1960.

[134] E. O. Elliott, "Estimates of Error Rates for Codes on Burst-Noise Channels," *The Bell System Technical Journal*, vol. 42, no. 5, pp. 1977–1997, 1963.

[135] R. Rudnik, L. Reyes Chamorro, A. Bernstein, J.-Y. Le Boudec, and M. Paolone, "Handling Large Power Steps in Real-Time Microgrid Control Via Explicit Power Setpoints," in *2017 IEEE Manchester PowerTech*, June 2017, pp. 1–6.

[136] P. Mahajan, L. Alvisi, and M. Dahlin, "Consistency, Availability, and Convergence," University of Texas at Austin, Tech. Rep., 2011.

[137] E. Piatkowska, L. P. Bayarri, L. A. Garcia, K. Mavrogenou, K. Tsatsakis, M. Sanduleac, and P. Smith, "Enabling Novel Smart Grid Energy Services with the Nobel Grid Architecture," in *2017 IEEE Manchester PowerTech*, June 2017, pp. 1–6.

[138] Y. Wu, J. Wei, and B. M. Hodge, "A Distributed Middleware Architecture for Attack-Resilient Communications in Smart Grids," in *2017 IEEE International Conference on Communications (ICC)*, May 2017, pp. 1–7.

[139] C. Wang, A. Bernstein, J.-Y. Le Boudec, and M. Paolone, "Explicit Conditions on Existence and Uniqueness of Load-Flow Solutions in Distribution Networks," *IEEE Transactions on Smart Grid*, 2016.

[140] C. Wang, E. Stai, and J.-Y. Le Boudec, "A Polynomial-Time Method for Testing Admissibility of Uncertain Power Injections in Microgrids," *arXiv preprint arXiv:1810.06256*, 2018.

[141] L. Reyes-Chamorro, A. Bernstein, J.-Y. Le Boudec, and M. Paolone, "A Composable Method for Real-Time Control of Active Distribution Networks with Explicit Power Setpoints. Part II: Implementation and Validation," *Electric Power Systems Research*, vol. 125, pp. 265–280, 2015.

[142] M. Bahramipanah, D. Torregrossa, R. Cherkaoui, and M. Paolone, "Enhanced Equivalent Electrical Circuit Model of Lithium-Based Batteries Accounting for Charge Redistribution, State-of-Health, and Temperature Effects," *IEEE Trans. Transport. Electrific*, vol. 3, no. 3, pp. 589–599, Sept 2017.

[143] E. Scolari, F. Sossan, and M. Paolone, "Irradiance Prediction Intervals for PV Stochastic Generation in Microgrid Applications," *Solar Energy*, vol. 139, no. Supplement C, pp. 116 – 129, 2016.

[144] Z. Xiao, T. Li, M. Huang, J. Shi, J. Yang, J. Yu, and W. Wu, "Hierarchical MAS Based Control Strategy for Microgrid," *Energies*, vol. 3, no. 9, pp. 1622–1638, 2010.

[145] P. Leitão, "Agent-Based Distributed Manufacturing Control: A State-of-the-Art Survey," *Engineering Applications of Artificial Intelligence*, vol. 22, no. 7, pp. 979–991, 2009.

[146] C. Urmson, J. A. Bagnell, C. R. Baker, M. Hebert, A. Kelly, R. Rajkumar, P. E. Rybski, S. Scherer, R. Simmons, S. Singh *et al.*, "Tartan Racing: A Multi-Modal Approach to the Darpa Urban Challenge," Carnegie Mellon University, Tech. Rep., 2007.

[147] T. Y. Teck, M. Chitre, and P. Vadakkepat, "Hierarchical Agent-Based Command and Control System for Autonomous Underwater Vehicles," in *Autonomous and Intelligent Systems (AIS), 2010 International Conference on*. IEEE, 2010, pp. 1–6.

[148] M. Dunn, "Toyota'S Killer Firmware: Bad Design and its Consequences," *EDN Network, October*, vol. 28, 2013.

[149] J. C. Lui, V. Misra, and D. Rubenstein, "On the Robustness of Soft State Protocols," in *Network Protocols, 2004. ICNP 2004. Proceedings of the 12th IEEE International Conference on*. IEEE, 2004, pp. 50–60.

[150] 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*, July 2001, pp. 107–116.

[151] J. Kirsch, S. Goose, Y. Amir, D. Wei, and P. Skare, "Survivable SCADA Via Intrusion-Tolerant Replication," *Smart Grid, IEEE Transactions on*, vol. 5, no. 1, pp. 60–70, 2014.

[152] L. Lamport, "The Part-Time Parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[153] Carnegie Mellon, University of Michigan, "Control Tutorials for MATLAB & Simulink," http://ctms.engin.umich.edu/CTMS/index.php?example=InvertedPendulum&section=SystemModeling, 2012, Accessed: 2018-11-13.

[154] "Mininet," http://mininet.org/, Accessed: 2018-11-13.

[155] D. M. Blough and G. F. Sullivan, "A Comparison of Voting Strategies for Fault-Tolerant Distributed Systems," in *Reliable Distributed Systems, 1990. Proceedings., Ninth Symposium on*. IEEE, 1990, pp. 136–145.

[156] J. L. Gersting, R. L. Nist, D. B. Roberts, and R. Van Valkenburg, "A Comparison of Voting Algorithms for N-Version Programming," in *System Sciences, 1991. Proceedings of the Twenty-Fourth Annual Hawaii International Conference on*, vol. 2. IEEE, 1991, pp. 253–262.

[157] J.-Y. Le Boudec, *Performance Evaluation of Computer and Communication Systems*. EPFL Press, 2010.

[158] J. Lin, K.-C. Leung, and V. O. Li, "Optimal Scheduling with Vehicle-to-Grid Regulation Service," *IEEE Internet of Things Journal*, vol. 1, no. 6, pp. 556–569, 2014.

[159] M. J. Stanovich, S. K. Srivastava, D. A. Cartes, and T. L. Bevis, "Multi-Agent Testbed for Emerging Power Systems," in *Power and Energy Society General Meeting (PES), 2013 IEEE*, 2013.

[160] R. M. Reddi and A. K. Srivastava, "Real Time Test Bed Development for Power System Operation, Control and Cyber Security," in *North American Power Symposium (NAPS), 2010*, 2010.

[161] A. Saleem, N. Honeth, Y. Wu, and L. Nordstrom, "Integrated Multi-Agent Testbed for Decentralized Control of Active Distribution Networks," in *Power and Energy Society General Meeting (PES), 2013 IEEE*, 2013.

[162] F. Maturana, R. Staron, K. Loparo, and D. Carnahan, "Agent-Based Testbed Simulator for Power Grid Modeling and Control," in *Energytech, 2012 IEEE*, 2012.

[163] A. Ravichandran, "Software-Defined MicroGrid Testbed for Energy Management," *Master Thesis*, 2011.

[164] Y. Cao, X. Shi, and Y. Li, "A Simplified Co-simulation Model for Investigating Impacts of Cyber-Contingency on Power System Operations," *IEEE Transactions on Smart Grid*, 2017.

[165] S. Schutte, S. Scherfke, and M. Troschel, "Mosaik: A Framework for Modular Simulation of Active Components in Smart Grids," in *2011 IEEE First International Workshop on Smart Grid Modeling and Simulation (SGMS)*, Oct 2011.

[166] D. P. Chassin, J. C. Fuller, and N. Djilali, "GridLAB-D: An Agent-Based Simulation Framework for Smart Grids," in *Journal of Applied Mathematics*, 2014.

[167] S. E. Mattsson, H. Elmqvist, and J. Broenink, "Modelica - An International Effort to Design the Next Generation Modeling Language," in *Benelux Quarterly Journal on Automatic Control*, 1998.

[168] EPRI, “OpenDSS,” "http://smartgrid.epri.com/SimulationTool.aspx", Accessed: 2018-01-22.

[169] “PyPower,” "https://pypi.python.org/pypi/PYPOWER", Accessed: 2018-01-22.

[170] Wikipedia, “LXC,” https://en.wikipedia.org/wiki/LXC, Accessed: 2018-01-22.

[171] P. Bacher and H. Madsen, “Identifying Suitable Models for the Heat Dynamics of Buildings,” *Energy and Buildings*, vol. 43, no. 7, pp. 1511–1522, 2011.

[172] Linux Foundation, “OvS: Open vSwitch,” http://openvswitch.org/, Accessed: 2018-01-22.

[173] S. Hemminger *et al.*, “Network Emulation with NetEm,” in *6th Australia's National Linux Conference (LCA 2005), Canberra, Australia (April 2005)*. Citeseer, 2005, pp. 18–23.

[174] L. Reyes-Chamorro, A. Bernstein, N. J. Bouman, E. Scolari, A. Kettner, B. Cathiard, J.-Y. Le Boudec, and M. Paolone, “Experimental Validation of an Explicit Power-Flow Primary Control in Microgrid,” in *IEEE Transactions on Industrial Informatics*, no. EPFL-ARTICLE-234511, 2018.

[175] IEEE PES Distribution System Analysis Subcommittee Distribution Test Feeder Working Group, “Distribution Test Feeders,” https://ewh.ieee.org/soc/pes/dsacom/testfeeders/index.html, accessed: 2018-01-22.

[176] P. Romano and M. Paolone, “Enhanced Interpolated-DFT for Synchrophasor Estimation in FPGAs: Theory, Implementation, and Validation of a PMU Prototype,” *IEEE Transactions on Instrumentation and Measurement*, vol. 63, no. 12, pp. 2824–2836, 2014.

[177] L. Reyes-Chamorro, W. Saab, R. Rudnik, A. M. Kettner, M. Paolone, and J.-Y. Le Boudec, “Slack Selection for Unintentional Islanding: Practical Validation in a Benchmark Microgrid,” in *2018 Power Systems Computation Conference (PSCC)*, June 2018, pp. 1–7.

[178] H. Michalska and D. Q. Mayne, “Robust Receding Horizon Control of Constrained Nonlinear Systems,” *IEEE Transactions on Automatic Control*, vol. 38, no. 11, pp. 1623–1633, Nov 1993.

[179] T. Balachandran, M. H. Kapourchali, M. Sephary, V. Aravinthan, M. Ni, S. Tindemans, H. Lei, C. Singh, M. Papic, M. Ben-Idris *et al.*, “Reliability Modeling Considerations for Emerging Cyber-Physical Power Systems,” in *Probabilistic Methods Applied to Power Systems (PMAPS), 2018 International Conference on*. IEEE, 2018.

List of Publications

Following is the list of all my publications written as a PhD student at EPFL.

Accepted

1. S. Bliudze, A. Cimatti, M. Jaber, S. Mover, M. Roveri, **W. Saab**, and Q. Wang, "Formal Verification of Infinite-State BIP Models," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2015, pp. 326–343.

2. M. Mohiuddin, **W. Saab**, S. Bliudze, and J.-Y. Le Boudec, "Axo: Masking Delay Faults in Real-Time Control Systems," in *Industrial Electronics Society, IECON 2016-42nd Annual Conference of the IEEE*. IEEE, 2016, pp. 4933–4940.

3. A. Bernstein, J.-Y. Le Boudec, M. Paolone, L. Reyes-Chamorro, and **W. Saab**, "Aggregation of Power Capabilities of Heterogeneous Resources for Real-Time Control of Power Grids," in *Power Systems Computation Conference (PSCC), 2016*. IEEE, 2016, pp. 1–7.

4. **W. Saab**, M. Mohiuddin, S. Bliudze, and J.-Y. Le Boudec, "Quarts: Quick Agreement for Real-Time Control Systems," in *Emerging Technologies and Factory Automation (ETFA), 2017 22nd IEEE International Conference on*. IEEE, 2017, pp. 1–8.

5. J. Achara, M. Mohiuddin, **W. Saab**, R. Rudnik, and J.-Y. Le Boudec, "T-RECS: A Software Testbed for Multi-Agent Real-Time Control of Electric Grids," in *Emerging Technologies and Factory Automation (ETFA), 2017 22nd IEEE International Conference on*. IEEE, 2017, pp. 1–4.

6. M. Mohiuddin, **W. Saab**, S. Bliudze, and J.-Y. Le Boudec, "Axo: Detection and Recovery for Delay and Crash Faults in Real-Time Control Systems," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3065–3075, July 2018.

7. **W. Saab**, M. Mohiuddin, S. Bliudze, and J.-Y. Le Boudec, "Ordering Events Based on Intentionality in Cyber-Physical Systems," in *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*. IEEE Press, 2018, pp. 107–118.

8. L. Reyes-Chamorro, **W. Saab**, R. Rudnik, A. M. Kettner, M. Paolone, and J.-Y. Le Boudec, "Slack Selection for Unintentional Islanding: Practical Validation in a Benchmark Microgrid," in *2018 Power Systems Computation Conference (PSCC)*, June 2018, pp. 1–7.

9. **W. Saab**, R. Rudnik, J.-Y. Le Boudec, L. Reyes-Chamorro, and M. Paolone, "Robust Real-Time Control of Power Grids in the Presence of Communication Network Non-Idealities," in *2018 IEEE International Conference on Probabilistic Methods Applied to Power Systems (PMAPS)*, June 2018, pp. 1–6.

10. J. P. Achara, M. Mohiuddin, **W. Saab**, R. Rudnik, J.-Y. Le Boudec, and L. Reyes-Chamorro, "T-RECS: A Virtual Commissioning Tool for Software-Based Control of Electric Grids: Design, Validation, and Operation," in *Proceedings of the Ninth International Conference on Future Energy Systems (e-Energy '18)*. ACM, 2018, pp. 303–313.

11. S. A. Sanaee Kohroudi, J. Mostafa, M. Mohiuddin, **W. Saab**, and J.-Y. Le Boudec, "Experimental Validation of the Suitability of Virtualization-based Replication for Fault Tolerance in Real-time Control of Electric Grids," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '18)*. ACM, 2018, pp. 46:1–46:4.

Under Review

1. M. Mohiuddin, **W. Saab**, and J.-Y. Le Boudec, "CROC: Consistent Replication of Controllers in a Cyber-Physical System with Asynchronous Sensors," *IEEE Transactions on Industrial Informatics*, 2018.

2. **W. Saab**, R. Rudnik, J.-Y. Le Boudec, L. Reyes-Chamorro, and M. Paolone, "A Robust Protocol for Slack Switching During Islanded Operation and Islanding Maneuvers," *IEEE Transactions on Industrial Informatics*, 2018.

Wajeb Saab

Curriculum Vitae

Lausanne, Switzerland
+41 763 61 47 44
wajeb.saab@gmail.com
goo.gl/6FheGH

Research Interests

Cyber-Physical Systems, Distributed Systems, Reliability, Real-Time

Education

2014–present **PhD in Computer Science & Communication Systems**,
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland.
Thesis: Reliable and Robust Cyber-Physical Systems for Real-Time Control of Electric Grids
Supervisor: Professor Jean-Yves Le Boudec

2010–2014 **Bachelor in Computer & Communications Engineering**,
American University of Beirut (AUB), Lebanon.
Graduated with High Distinction (*GPA - 92/100*)

Research Experience

2014–present **Doctoral Assistant**, *Laboratory for Computer Communications and Applications (LCA2)*, EPFL, Switzerland.

- Designed state-of-the-art mechanisms for real-time distributed systems
  - A fault-tolerance protocol for delay and crash faults
  - An agreement protocol with bounded latency-overhead
  - A labeling scheme for replicated controllers in cyber-physical systems
  - A robust computation mechanism for controllers of electric grids
- Implemented proposed mechanisms in Python/C/C++
- Developed an emulation framework for testing these solutions on smart grid controllers
- Field-deployed these solutions in several smart grids in Switzerland

Summer 2013 **Research Intern**, *Rigorous System Design Laboratory (RiSD)*, EPFL, Switzerland.

- Supervised by Professor Joseph Sifakis (Turing Award 2007)
- Proposed a method for transforming BIP models into NuSMV models for model-checking
- Developed a source-to-source tool in Java for automatically performing the transformation
- The tool was used in the model-based system design course taught by Professor Sifakis at EPFL

2012–2013 **Research Assistant**, *American University of Beirut*, Lebanon.

- Proposed several new truncated digital multiplier circuit designs
- Calculated improvements in transistor count, circuit area, and energy consumption
- Evaluated their performance on JPEG compression and edge detection algorithms

Teaching Experience

2015–2017 **Teaching Assistant**, *TCP/IP Networking*, EPFL.
Responsible for lab sessions on IPv4/IPv6 networking, routing, and UDP/TCP socket programming.

2017 **Teaching Assistant**, *Performance Evaluation*, EPFL.
Responsible for lab sessions on discrete-event simulation and queuing theory.

2015 **Teaching Assistant**, *Real-Time Networks*, EPFL.
Responsible for exercise sessions on CAN and time-triggered Ethernet.

2013 **Teaching Assistant**, *Computer Networks*, AUB.
Responsible for a project on client-server communication over TCP/IP.

Professional Activities

2017–2018 **Co-Founder**, *RaaSS*.
- Co-founded a startup for enterprise reliability software for cyber-physical systems, such as smart grids, autonomous vehicles, and datacenter SDN networks.
- Carried out customer-discovery interviews to better understand the product-market fit.
- Raised capital through the EPFL enable grant.

2017–2018 **Project Demos**, *EPFL*.
- Demonstrated the islanding of an on-campus microgrid using a real-time control framework during an SNSF visit in 2018.
- Demonstrated a stability analysis of an inverted pendulum system in the presence of delays at the IC Research day 2017.

2017–2018 **Reviewer**, *EPFL*.
- Elsevier Journal on Sustainable Energy, Grids and Networks

Supervised Projects

2018 **Reliable Communication Protocol for Smart Grids**.
Semester Project, John Stephan, EPFL Master Student

Ephemeral QUIC for Distributed Real-Time Cyber-Physical Systems.
Master Thesis, Weiyu Zhang, EPFL Master Student

2017 **Implementation of a Three-Phase Load-Flow Solver for a Microgrid Simulator**.
Semester Project, Diana Rivera Villanueva, EPFL Master Student

Implementation of a Virtual Commissioning System for Electric Grids.
Semester Project, Firas Belhaj, EPFL Master Student

FRED: Fast Recovery for Ephemeral Data in Real-Time Systems.
Semester Project, Robin Solginac, EPFL Master Student

Implementation of Quarts in C++.
Summer Internship, Muhammad Tirmazi, Visiting Bachelor Student

Software Aging in Real-Time Control Systems.
Master Thesis, Jalal Mostafa, Visiting Master Student

Correlation in the Performance of Replicated Virtualized Controllers.
Summer Internship, Ali Reza Sanaee, Visiting Master Student

2016 **Proactive Fault-Recovery For Real-Time Mission-Critical Systems**.
Master Thesis, Aswin Suresh, EPFL Master Student

Modeling and Validation of a Fault-Tolerance Protocol.
Semester Project, Marco Zoveralli, EPFL Master Student

Extracurricular Activities

2015–present **PolyProg Committee Member**, *EPFL*.
- Held the treasurer seat for the fiscal year 2016-2017
- Organized three editions of HC2: the largest programming contest in Switzerland
  - Led the organization team for two editions
  - Set and tested problems for three editions
- Proposed and set several problems for a programming contest for first-year students
- Gave an algorithmic and programming seminar for first-year students

2012–2015 **ACM Programming Team Member**, *AUB/EPFL*.
- Attended weekly training sessions; participated in several programming contests such as TopCoder, Codeforces, Google Code Jam, Facebook Hacker Cup, and the IEEEXtreme
- Winner - 2013 ACM Lebanese Collegiate Programming Competition (LCPC)
- Participant - 2013 ACM Arab Collegiate Programming Competition (ACPC)
- Bronze medalist - 2015 ACM South Western European Regional Competition (SWERC)

Skills

OS	**Linux, MacOS, Windows**
Programming	**C/C++, Java, Python**
Miscellaneous	**TCP/IP, Distributed Algorithms**
Languages	**English, Arabic**

Honors and Awards

2018 **EPFL Teaching Award**, *Excellence in teaching a graduate TCP/IP networking course.*
2017 **Best Statup Pitch Award**, *Innosuisse Business Concept course, Lausanne.*
2017 **Venturelab AIT Participant**, *Top 10 Swiss researchers with entrepreneurial vision.*
2016 **Best Project Award**, *EPFL Applied Data Analysis course.*
2014 **EPFL Fellowship**, *PhD Fellowship for the Computer Science PhD program at EPFL.*

**Ce document a été imprimé au Centre d'impression EPFL,
imprimerie climatiquement neutre, certifiée myClimate.**