Reliability Mechanisms for Controllers in Real-Time Cyber-Physical Systems

THÈSE N^{O} 8804 (2018)

PRÉSENTÉE LE 26 OCTOBRE 2018
À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE POUR LES COMMUNICATIONS INFORMATIQUES ET LEURS APPLICATIONS 2
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Maaz MASHOOD MOHIUDDIN

acceptée sur proposition du jury:

Prof. V. Kuncak, président du jury
Prof. J.-Y. Le Boudec, directeur de thèse
Dr Y.-A. Pignolet, rapporteuse
Dr S. Bliudze, rapporteur
Prof. R. Guerraoui, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2018

We can only see a short distance ahead,
but we can see plenty there that needs to be done.
- Alan Turing

To my mother…

Acknowledgements

I would like to begin by thanking my thesis director, Prof. Jean-Yves Le Boudec, for his immense guidance and support throughout the duration of my PhD. He has been a constant source of learning, and has left a lasting impression on my professional and personal life through the way he conducts himself. I am privileged to be his student.

I would also like to express my gratitude to Dr. Yvonne-Anne Pignolet, Dr. Simon Bliudze, Prof. Rachid Guerraoui and Prof. Viktor Kuncak for agreeing to serve on my PhD committee. I am thankful for their feedback and the fruitful discussions. A special thanks to Simon for serving as my unofficial co-advisor, for playing a significant role in building the abstractions, and for polishing the results.

When I look back at the last five years at EPFL, the contributions of two peers stand out: Wajeb and Miroslav. Wajeb, my PhD twin, has been with me through the highs and lows of research, the startup, and life outside the office. We managed to make some excellent memories and I think we can both be happy with what we achieved. Miroslav has been my mentor since day one at EPFL. I am indebted to him for guiding me through my initial blues during the internship, launching me into the PhD through a great collaboration and orchestrating my post-PhD plans. I am truly blessed to have these guys in my life.

Being a part of LCA2 was a great experience. The lab is a rich melting pot of cultures, backgrounds and perspectives, with so much to explore. In the first phase, I had some great times with Miroslav, Nadia, Tech and Dan, who provided a much-needed outlet for coping with the work-stress through our lazy outings. Sergio was the go-to-guy for networking, who made TAing TCP/IP a lot easier. In the second phase, Wajeb, Roman formed a good addition to the LCA2 family and helped continue the work-life balance.

A key role in keeping LCA2 the well-oiled machine it is, are the secretaries Patricia, Angela and Holly, and the system administrators Marc-André and Yves. They work tirelessly so that we can focus exclusively on things we are good at. Without Holly, I would have a lot more hyphens and instances of *there by* and *where as* in my papers. If it were not for Patricia, I would probably be living under a bridge.

I was fortunate to have several collaborations during the course of my PhD. A special thanks to Elena, Mia and Ehsan who were instrumental in helping me end my doctoral studies with a flourish. I would also like to thank Prof. Mario Paolone and the DESL crew for providing me with a real opportunity to test out my power-point research on their gear. I was also lucky to supervise several students who helped me

offload parts of my work and try out some eccentric ideas.

I would like to thank all my friends in Switzerland for making my stay a pleasant one. The board games, hikes and dinners with Kassir, Wajeb, Priyanka, Taha, Sanket, Harshal, Czuee and Anwar in Lausanne served as refreshing palate-cleansers after a tough week at the office. I would like thank Hend for getting me up to speed with the bare minimum level of French required to survive in Lausanne. The base in Zürich served as a perfect retreat to a second home in the final years of my PhD. The boys from IITH (Adi, Nagi and Aniket) who were in similar situations in the USA, helped form an effective support group.

Lastly, I would like to thank my family for supporting me throughout my time as a doctoral student, either through calming phone sessions or through care-packages from Hyderabad. I am grateful for having a loving sister who has stood by me through thick and thin. Without the constant perseverance of my parents, I could not have dreamt of pursuing a PhD at EPFL. I am thankful to them for instilling the *never-say-no* attitude in me.

Lausanne, 1st October 2018 M. M.

Abstract

Cyber-physical systems (CPSs) are real-world processes that are controlled by computer algorithms. We consider CPSs where a centralized, software-based controller maintains the process in a desired state by exchanging measurements and setpoints with process agents (PAs). As CPSs control processes with low-inertia, e.g., electric grids and autonomous cars, the controller needs to satisfy stringent real-time constraints.

However, the controllers are susceptible to delay and crash faults, and the communication network might drop, delay or reorder messages. This degrades the quality of control of the physical process, failure of which can result in damage to life or property. Existing reliability solutions are either not well-suited for real-time CPSs or impose serious restrictions on the controllers. In this thesis, we design, implement and evaluate reliability mechanisms for real-time CPS controllers that require minimal modifications to the controller itself.

We begin by abstracting the execution of a CPS using events in the CPS, and the two inherent relations among those events, namely network and computation relations. We use these relations to introduce the intentionality relation that uses these events to capture the state of the physical process. Based on the intentionality relation, we define three correctness properties namely, state safety, optimal selection and consistency, that together provide linearizability (one-copy equivalence) for CPS controllers.

We propose a labeling mechanism called intentionality clocks that can be used by controllers and PAs to express the intentionality relation. Intentionality clocks are robust to delays and can be used with replicated controllers to establish a total order between the events. We use intentionality clocks to design a CPS controller and PAs that guarantee the state safety and optimal selection properties.

Then, we design Quarts, an agreement algorithm that guarantees the consistency property among replicated controllers. Inconsistent controllers can issue setpoints that result in incorrect control of the process. So, agreement is typically achieved using consensus between the replicated controllers. This can add an unbounded-latency overhead. Quarts leverages the properties specific to CPSs to perform agreement using pre-computed priorities among sets of received measurements, resulting in a bounded-latency overhead with high availability. Using simulation, we show that availability of Quarts, with two replicas, is more than an order of magnitude higher than consensus.

Intentionality clocks and Quarts together provide linearizability that enables active replication of the central controller. Using this result, we design Axo, a fault-tolerance architecture for delay and crash faults in real-time CPSs. Axo also adds new mechanisms to detect and recover faulty replicas, and provide timeliness that requires delayed setpoints be masked from the PAs. We study the effect of delay faults and the impact of fault-tolerance with Axo, by deploying Axo in two real-world CPSs.

Then, we realize that the proposed reliability mechanisms also apply to unconventional CPSs such as software defined networking (SDN), where the controlled process is the routing fabric of the network. We show that, in SDN, violating consistency can cause implementation of incorrect routing policies. Thus, we use Quarts and intentionality clocks, to design and implement QCL, a coordination layer for SDN controllers that guarantees control-plane consistency. QCL also drastically reduces the response time of SDN controllers when compared to consensus-based techniques.

In the last part of the thesis, we address the problem of reliable communication between the software agents, in a wide-area network that can drop, delay or reorder messages. For this, we propose iPRP, an IP-friendly parallel redundancy protocol for 0 ms repair of packet losses. iPRP requires fail-independent paths for high-reliability. So, we study the fail-independence of Wi-Fi links using real-life measurements, as a first step towards using Wi-Fi for real-time communication in CPSs.

Key words: cyber-physical system, real-time, mission-critical, control system, smart grid, autonomous car, reliability, delay, crash, software-fault, software-base control, commercial off-the-shelf, intentionality, redundancy, consistency, linearizability, availability, consensus, agreement, bounded latency-overhead, timeliness, fault-detection, fault-recovery, fault-masking, software-defined networking, control-plane, industrial communication, low-latency, fail-independent paths, packet replication, real-time communication, Wi-Fi, measurements, performance evaluation.

Résumé

Les systèmes cyber-physiques (CPS) sont des processus d'interaction avec le monde réel contrôlés par des algorithmes informatiques. Nous considérons des CPS où un contrôleur logiciel centralisé échange des mesures et des valeurs de consigne avec des agents de processus (AP) afin de maintenir un processus dans un état souhaité. Comme les CPS contrôlent des processus à faible inertie, tels que les réseaux de distribution électrique et les voitures autonomes, le contrôleur doit satisfaire des contraintes strictes en temps réel.

Les contrôleurs logiciels sont cependant susceptibles de subir des retards et des pannes, et le réseau de communication risque d'abandonner, de retarder ou de réordonner les messages. Ceci entraîne une dégradation de la qualité du contrôle du processus physique, pouvant entraîner des dommages physiques ou mettre des vies en danger. Les solutions de fiabilité existantes ne sont pas bien adaptées aux CPS en temps réel ou imposent de sérieuses restrictions aux contrôleurs CPS. Dans cette thèse, nous concevons, implémentons et évaluons des mécanismes de fiabilité pour les contrôleurs CPS en temps réel qui nécessitent le minimum de modifications au contrôleur lui-même.

Nous commençons par formaliser l'exécution d'un CPS en termes d'évènements dans le CPS, et par définir les deux types inhérents de relations entre ces événements : les relations de réseau et les relations de calcul. Nous utilisons ces relations pour introduire la relation d'intentionnalité, qui saisit l'état du processus physique grâce à ces évènements. Sur la base de la relation d'intentionnalité, nous définissons trois propriétés d'exactitude : la sécurité d'état, la sélection optimale et la cohérence. Ces trois propriétés garantissent la linéarisabilité (ou équivalence à une copie) des contrôleurs CPS.

Nous proposons un mécanisme d'étiquetage appelé « horloges d'intentionnalité » qui peut être utilisé par les contrôleurs et les AP pour exprimer la relation d'intentionnalité. Les horloges d'intentionnalité sont robustes aux retards et peuvent être utilisées avec des contrôleurs répliqués pour établir un ordre total entre les évènements. Nous utilisons les horloges d'intentionnalité afin de créer un contrôleur CPS et des AP qui garantissent les propriétés de sécurité d'état et de sélection optimale.

Nous proposons ensuite Quarts, un algorithme d'accord garantissant la propriété de cohérence entre contrôleurs répliqués. Des contrôleurs incohérents peuvent émettre des valeurs de consigne qui entraînent un contrôle incorrect du processus. L'accord est généralement obtenu par consensus entre les contrôleurs répliqués, ce qui peut rajouter un coût de latence non bornée. Quarts utilise les propriétés propres aux CPS pour établir un accord en utilisant des priorités précalculées parmi des ensembles de mesures, garantissant ainsi une latence bornée et une haute disponibilité. Nous démontrons via des simulations que pour le cas de deux réplicas, Quarts offre une disponibilité supérieure de 10x à celle garantie par la solution de consensus.

Ensemble, les horloges d'intentionnalité et Quarts garantissent la linéarisabilité et permettent donc une réplication active du contrôleur central. Sur la base de ces résultats, nous proposons Axo, une architecture à tolérance de pannes pour les erreurs de retard et de panne dans les CPS en temps réel. Axo offre également de nouveaux mécanismes pour la détection et la récupération des réplicas erronées, ainsi qu'une garantie de rapidité moyennant le masquage des valeurs de consigne tardives par rapport aux AP. Nous déployons Axo dans deux CPS du monde réel pour étudier l'effet des erreurs de retard et l'impact de la tolérance d'erreurs d'Axo.

Nous revenons ensuite sur les mécanismes proposés dans cette thèse et réalisons qu'ils s'appliquent également aux CPS non conventionnels tels que le réseau défini par logiciel (SDN), où le processus contrôlé est le tissu de routage au sein d'un réseau. Nous montrons que dans le contexte SDN, l'absence de cohérence peut entraîner l'implémentation de politiques de routage incorrectes. Nous utilisons donc Quarts et des horloges d'intentionnalité pour créer et implémenter QCL, une couche de coordination pour les contrôleurs SDN qui garantit la cohérence du plan de contrôle. Par rapport aux techniques de consensus, QCL offre une réduction drastique du temps de réponse des contrôleurs SDN.

Dans la dernière partie de la thèse, nous abordons le problème de la communication fiable entre les contrôleurs et les agents de processus, dans un réseau étendu qui peut abandonner, retarder ou réordonner les messages. Pour cela, nous proposons iPRP, un protocole de redondance parallèle IP-compatible pour réparer les pertes de paquets en 0 ms. iPRP exige des routes à défaillance indépendante pour garantir une haute fiabilité. Nous utilisons des mesures du monde réel pour étudier l'indépendance de défaillance des liaisons Wi-Fi, en tant que première étape de l'utilisation de iPRP sur le Wi-Fi, pour la communication dans un CPS à temps réel.

Mots clés : système cyber-physique, temps réel, mission critique, système de contrôle, réseau intelligent, voiture autonome, fiabilité, retard, accident, défaillance logicielle, contrôle de base de logiciel, commercial, intentionnalité, redondance, cohérence, linéarisation, disponibilité, consensus, accord, temps de latence borné, rapidité, détection de pannes, récupération de pannes, masquage de pannes, mise en réseau logicielle, plan de contrôle, communication industrielle, faible latence, chemins à défaillance indépendante, réplication de paquets, communication en temps réel, Wi-Fi, mesures, évaluation des performances.

List of Abbreviations

API	Application Programming Interface
BFT	Byzantine Fault-Tolerance
CCDF	Complementary Cumulative Distribution Function
CDF	Cumulative Distribution Function
COTS	Commercial Off-the-Shelf
CPS	Cyber-Physical System
DDR	Double Data Rate
DTLS	Datagram Transport Layer Security
ECDF	Empirical Cumulative Distribution Function
EV	Electric Vehicle
FLP	Fischer Lynch Paterson
GA	Grid Agent
GPS	Global Positioning System
HMAC	Hash-Based Message Authentication Code
ICB	iPRP Control Block
ID	Identifier
IEEE	Institute of Electrical and Electronics Engineers
IMB	iPRP Monitoring Block
IND	iPRP Network Discriminator
IP	Internet Protocol
iPRP	IP-friendly parallel redundancy protocol
IRB	iPRP Receiving Block
ISB	iPRP Sending Block
LAN	Local-Area Network
LQR	Linear–Quadratic Regulator
MAC	Media Access Control
MPLS	Multiprotocol Label Switching
MPLS-TP	Multiprotocol Label Switching - Transport Profile
MPTCP	Multi-Path Transmission Control Protocol

MTTF	Mean Time to Instability
MTTR	Mean Time to Repair
NFV	Network Function Virtualization
NTP	Network Time Protocol
OSPF	Open Shortest Path First
PA	Process Agent
PC	Personal Computer
PDC	Phasor Data Concentrator
PDF	Probability Density Function
PLC	Programmable Logic Controller
PLR	Packet Loss Rate
PMU	Phasor Measurement Unit
PRP	Parallel Redundancy Protocol
PTP	Precision Time Protocol
PV	Photovoltaic
QCL	Quick Coordination Layer
QCP	Quick Controller Proxy
QSP	Quick Switch Proxy
QUIC	Quick UDP Internet Connections
RAM	Random Access Memory
RIP	Routing Information Protocol
RPL	Routing Protocol for Low-Power and Lossy Networks
RSTP	Rapid Spanning Tree Protocol
RTO	Retransmission Timeout
RTT	Round-Trip Time
SDN	Software Defined Networking
SN	Sequence number
SNSID	Sequence Number Space ID
SoC	State-of-Charge
TCB	Timely Computing Base
TCP	Transmission Control Protocol
TTA	Time-Triggered Architecture
UDP	User Datagram Protocol
VLAN	Virtual Local-Area Network
VPN	Virtual Private Network
WAN	Wide-Area Network
WCET	Worst-Case Execution Time

Contents

Acknowledgements **i**

Abstracts (English and French) **iii**

List of Abbreviations **vii**

List of Figures **xiii**

List of Tables **xvii**

1 Introduction **1**
- 1.1 Background 2
- 1.2 Global Trends and Reliability Issues in CPSs 4
  - 1.2.1 Real-Time Control 4
  - 1.2.2 Complex Control and COTS Components 5
  - 1.2.3 IP-Based Communication 6
  - 1.2.4 Mission-Critical Applications 8
- 1.3 Challenges 9
- 1.4 Contributions 10
- 1.5 Roadmap 13

2 State of the Art **15**
- 2.1 Controller Reliability 15
  - 2.1.1 Passive Replication 16
  - 2.1.2 Active Replication 18
- 2.2 Communication Reliability 20

3 CPS Model and Intentionality **23**
- 3.1 Introduction 23
  - 3.1.1 State of the Controlled Process 25
  - 3.1.2 Need for a New Relation 26
  - 3.1.3 Contributions 27
- 3.2 Related Work 27
- 3.3 High-Level Model of the Software Agents in a CPS 28

- 3.3.1 Controller Model . . . 29
- 3.3.2 PA Model . . . 30
- 3.4 Events in a CPS . . . 31
- 3.5 Basic Relations between CPS Events . . . 33
  - 3.5.1 Network Relation . . . 33
  - 3.5.2 Computation Relation . . . 33
- 3.6 Intentional Equivalence and Intentionality . . . 35
  - 3.6.1 Intentional Equivalence Relation . . . 35
  - 3.6.2 Intentionality Relation . . . 36
- 3.7 Correctness Properties . . . 37
- 3.8 Conclusion . . . 39

4 Ordering Events Based on Intentionality **41**
- 4.1 Clock-Consistency Condition . . . 42
- 4.2 Clock Consistency Using Existing Solutions . . . 43
  - 4.2.1 Using Physical Time . . . 43
  - 4.2.2 Using Logical Clocks . . . 45
- 4.3 Intentionality Clocks . . . 46
- 4.4 CPS Design with Intentionality Clocks . . . 49
  - 4.4.1 Controller Design . . . 49
  - 4.4.2 PA Design . . . 51
- 4.5 Formal Guarantees . . . 52
- 4.6 Case Study: CPS for Scheduling EV Charging . . . 59
  - 4.6.1 CPS Design for EV Charging . . . 60
  - 4.6.2 Deadlock Due to Violation of Optimal Selection . . . 61
  - 4.6.3 Design with Intentionality Clocks . . . 62
- 4.7 Conclusion . . . 64

5 Quick Agreement among Replicated CPS Controllers **67**
- 5.1 Importance of Consistency in a CPS . . . 68
- 5.2 Related Work . . . 70
- 5.3 CPS Properties Required for Quarts . . . 71
- 5.4 Quarts Design . . . 73
  - 5.4.1 Protocol Walk-Through . . . 74
  - 5.4.2 Collection Phase . . . 76
  - 5.4.3 Voting Phase . . . 78
- 5.5 Formal Guarantees . . . 81
- 5.6 Performance Evaluation . . . 85
  - 5.6.1 Performance Metrics . . . 85
  - 5.6.2 Agreement Protocols . . . 86
  - 5.6.3 Simulation Methodology . . . 87
  - 5.6.4 Results . . . 90
- 5.7 Quarts+: Incorporating Asynchronous Sensors . . . 94

5.7.1 Challenge with Asynchronous Sensors 95
5.7.2 CPS Requirements for Quarts+ 96
5.7.3 Quarts+ Design 97
5.7.4 Formal Guarantees 100
5.7.5 Performance Evaluation 103
5.8 Conclusion 106

6 Axo: Tolerating Delay and Crash Faults in Real-Time CPSs 107
6.1 Timeliness in a CPS – Definitions and Challenges 109
6.2 Related Work 111
6.3 Axo 112
6.3.1 Assumptions 112
6.3.2 Design 113
6.4 Implementation 123
6.5 Performance Guarantees 124
6.5.1 Timeliness Guarantees 124
6.5.2 Bounds on Recovery Time 127
6.6 Case Study I: COMMELEC 135
6.6.1 COMMELEC Background 135
6.6.2 Test Setup 136
6.6.3 Importance of Fault-Tolerance 136
6.6.4 Effect of Inconsistency 138
6.7 Case Study II: Inverted Pendulum 139
6.8 Conclusion 143

7 QCL for Consistent-Controller Replication in SDN 145
7.1 Introduction 146
7.1.1 Problem 146
7.1.2 Our Approach and Challenges 148
7.1.3 Contributions 149
7.2 System Model 150
7.3 QCL Design 150
7.3.1 QCL Messages & Functions 152
7.3.2 Ordering QCL Messages 153
7.3.3 Exporting Switch State by QSP 154
7.3.4 Agreement at QCP 154
7.3.5 Data-plane Consistency 156
7.4 Formal Guarantees 160
7.4.1 Control-Plane Guarantees 160
7.4.2 Enforceable Safety Policies 161
7.4.3 Safety Policy Guarantees 162
7.5 Performance Evaluation 163
7.5.1 Simulation Setup 163

7.5.2 Replication Schemes 163
7.5.3 Safety Policy Results 165
7.5.4 Liveness Policy Results 165
7.6 System Implementation 172
7.7 Related Work 173
7.8 Conclusion 174

8 Reliable Real-Time Communication in Unreliable Networks 175
8.1 Introduction 176
8.2 Related Work 179
8.3 Technical Challenges in iPRP Design 180
8.4 iPRP Design 182
8.4.1 Protocol Walk-Through 183
8.4.2 IP Multicast 187
8.4.3 Crash of a Host 189
8.4.4 Addition or Removal of Network Interfaces 189
8.4.5 Duplicate Discard Algorithm 190
8.4.6 Security Considerations 191
8.5 Correctness of the Discard Algorithm 191
8.6 iPRP Diagnostic Toolkit 194
8.7 Measurements with Directional Wi-Fi Links 196
8.7.1 Description of the Test Bed 196
8.7.2 Experimental Methodology 197
8.7.3 Measurement Results 199
8.8 Are direction Wi-Fi links fail-independent? 201
8.8.1 Estimating the Distribution of Losses 202
8.8.2 Information-Theoretic Test of Independence 203
8.8.3 Impact of Dependent Losses on P_{rep} 205
8.9 Conclusion 205

9 Conclusion and Open Questions 209

A Discussion on Theorems in Section 3.6 215
A.1 Proof of Theorem 3.6.1 215
A.2 Proof of Theorem 3.6.2 216
A.3 Proof of Theorem 3.6.3 216

Bibliography 219

List of Publications 237

Curriculum Vitae 239

List of Figures

1.1 Architecture of a CPS 2
1.2 Response time of the COMMELEC controller as a function of time . . . 6
1.3 Tracking a dispatch plan with the COMMELEC controller 7
1.4 Effect of network losses on COMMELEC's ability to track a given dispatch plan 7
1.5 Energy mismatch between the dispatch plan and that provided by COMMELEC during a period of five minutes 8
1.6 Pairwise scatter plots of response times of replicas of the same COMMELEC controller running on different virtual machines on the same physical machine 9

3.1 Architecture of a CPS without asynchronous sensors and actuators that directly communicate with the controller 24
3.2 Illustration of why the happened-before relation is not suitable for ordering events in a CPS in the presence of replication 26

4.1 Difference between temporal order and intentionality due to replication of the controller. The notation (1, 2, 3) represents the vector clock of an event in the format (PA:1, C_1:2, C_2:3) 44
4.2 Difference between temporal order and intentionality due to delays . . 44
4.3 Example of computation at a controller as a result of a timeout indicating the clock at the agents after each event 48
4.4 Difference between Lamport clocks and intentionality clocks – An example of erroneous incrementation of Lamport clocks with two PAs 49
4.5 Architecture and information flow of a CPS for EV charging [1] 60

5.1 Architecture of the CPS indicating the messages exchanged with round numbers 71
5.2 Gilbert-Elliot model for simulating delay and crash faults 88
5.3 Unavailability with varying g and varying p. Unavailability of Quarts (Q) with more than 3 replica is less than 4×10^{-10} 90
5.4 Mean and 99^{th} percentile of latency in different scenarios 92
5.5 Mean and 99^{th} percentile of messaging cost in different scenarios . . . 93

5.6 Architecture of a CPS with PAs and asynchronous sensors that directly communicate with the controller 94
5.7 Inconsistency of Quarts with 2 replicas as a function of μ for different values of m 104

6.1 Axo design 114
6.2 Time to recover from delay-faults for varying θ 134
6.3 Time to recover from crash-faults for varying θ 134
6.4 COMMELEC architecture with 3 resources and PAs 136
6.5 13-bus CIGRÉ low-voltage benchmark grid controlled using COMMELEC to provide autonomy by minimizing the total export or import of power at bus B01. Grey areas are unused 137
6.6 Timeliness guarantee of Axo 137
6.7 Energy mismatch over time in COMMELEC 139
6.8 Snapshot of the inverted pendulum from experiments 140
6.9 Step response of the pendulum with a single controller for different values of controller delay 141
6.10 Illustration of the inverted pendulum CPS with two controllers and one PA (the pendulum) 142
6.11 Stability of the pendulum with a replicated controller 142

7.1 Example of edge disjoint isolation violation 147
7.2 Components in QCL 151
7.3 Unsafety in SCL and Passive schemes. Unsafety of QCL is 0 164
7.4 Representative scenario (Basic): ft16 with $g = 2$, $\delta_n = 0.5\, ms$ and parameter setup `normal` 170
7.5 CCDF of response and convergence times for the basic scenario 171
7.6 CCDF of convergence time for AS 1221 with parameter set `normal`, $g = 2$ and $\delta_n = 10\, ms$. 171
7.7 Median and tail (at 99^{th} percentile) of 4-port fat-tree with 2 and 3 replicas in ms 173

8.1 Overview of the iPRP design indicating the flow of a packet 182
8.2 Example of a network with two connected subclouds performing iPRP 184
8.3 Placement of iPRP in the networking stack 187
8.4 Map of the campus with the antenna locations. 197
8.5 Hardware used for the test-bed. 198
8.6 Box plot of the PLRs for link AB, link AC and after replication for scenarios with and without MAC-layer retransmissions, shown in log-scale. . . . 200
8.7 Two-state Gilbert model for bursty losses 202
8.8 Box plot of parameters of the two-state Gilbert model for links AB and AC, p^i_{ab}, q^i_{ab}, p^i_{ac}, q^i_{ac}, shown in log scale 203

8.9 QQ-plot of the number of losses on each link showing the non-stationary of distribution of losses across scenarios 203

List of Tables

5.1 Unavailability results for the chosen scenarios 91
5.2 Inconsistency results for the chosen scenarios. Inconsistency of A and AC is zero. * No inconsistency was observed in 10^{10} runs 91
5.3 Mean and 99^{th} percentile of latency for select scenarios 92
5.4 Mean and 99^{th} percentile of messaging cost for select scenarios 93
5.5 Unavailability results for the chosen scenarios. * No event in 10^{10} runs 105
5.6 Latency results for the chosen scenarios 105

6.1 Instability of an inverted pendulum in selected scenarios with varying θ_d 142

7.1 Median and tail (at 99^{th} percentile) response times for different scenarios. The column Setup shows the difference in each scenario from basic setup seen in Figure 7.4 167
7.2 Median and tail (at 99^{th} percentile) convergence times for different scenarios. The column Setup shows the difference in each scenario from basic setup seen in Figure 7.4 167

8.1 PLR statistics obtained from measurements 199
8.2 99^{th} percentile value of PLRs with and without MAC-layer retransmissions 200
8.3 Latency statistics with MAC-layer retransmissions 201

1 Introduction

This is your last chance.
After this, there is no turning back.
You take the blue pill – the story ends,
you wake up in bed & believe what you want.
You take the red pill – you stay in Wonderland,
and I show you how deep the rabbit hole goes.
— Morpheus, The Matrix

According to the Oxford English dictionary [2], the term *cyber* is defined as “relating to or a characteristic of the culture of computers, information technology, and virtual reality”. The term *physical* is defined as “relating to physics or the operation of natural forces generally”. The marriage between these two domains has resulted in CPSs that are physical processes whose operations are monitored, coordinated and controlled by a computing and communication core [3].

In contrast to traditional computer systems, such as database systems where computers manage and control data, *i.e.,* bits of information, CPSs control elements of physical processes such as electric power and voltages in smartgrids [1, 4, 5, 6, 7], position and velocity in autonomous vehicles [8, 9, 10, 11, 12], and positions of objects on a conveyor belt in manufacturing processes [13, 14, 15]. Consequently, the physical and software components are deeply intertwined, interacting with each other in a myriad of ways that change with context [16].

These new interactions have led to new performance requirements, previously unbeknownst to traditional computer systems. In this thesis, we identify some of the requirements pertaining to the reliability of the controllers and their communication network, we propose mechanisms that realize those requirements, and we demonstrate the efficacy of the mechanisms through results from practical deployments.

1.1 Background

Figure 1.1 shows the architecture of a CPS as seen in the domains of electric grids [1, 4, 5, 6, 7], in manufacturing processes [13] and in autonomous vehicles [8, 9, 12], among others. It consists of the physical elements and cyber components. The physical elements are the *controlled process* and *controlled resources*. The cyber components are the *controller*, *PAs*, *sensors* and *actuators*. We will describe these components in more detail using the COMMELEC [4] CPS as an example.

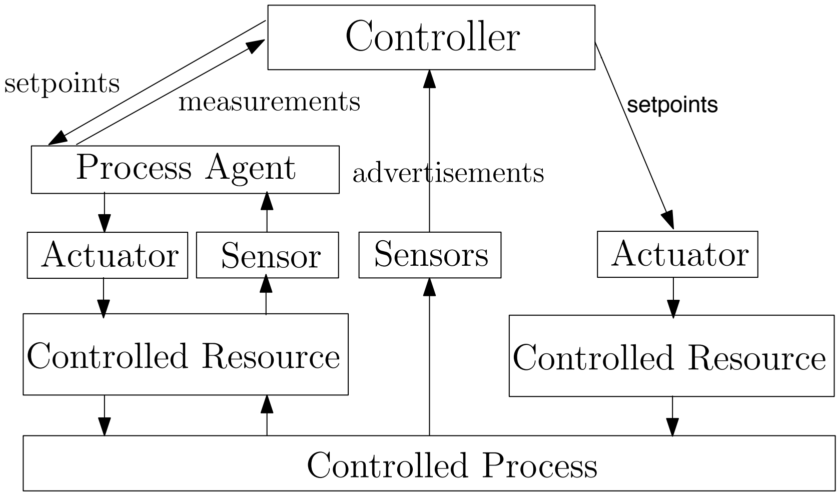


Figure 1.1 – Architecture of a CPS

COMMELEC is a CPS for real-time control of electric grids. The controlled process is the global state of the electric grid. This state consists of the current in all the lines and the voltages at all the buses. The goal of COMMELEC is to implement one or more auxiliary policies and to ensure that the currents are within the ampacity limits of the lines and that the voltages are maintained within the respective nodal voltage limits. The auxiliary policies include providing primary frequency-support to an upper-level grid [17], acting as a virtual power-plant [18], or following a dispatch signal from an external source. The process is controlled by altering the state of controlled resources. In COMMELEC, these resources include batteries, PV panels, heat-pumps, charging stations for EVs, etc. These resources are controlled by modulating the active- and reactive- power injections.

The sensors and actuators are low-level agents that interface with the physical process. The sensors read the state of the process and send it to the controller. A message sent by the sensor to a controller indicating the state of the process is called an *advertisement*. Actuators receive setpoints from the controllers or PAs and change the state of the controlled resource. Alternatively, PAs and controllers are high-level software agents that perform more complex control than sensors and actuators.

A constraint of the controller is to issue setpoints that keep the controlled process in a desired state. In COMMELEC, this amounts to maintaining the currents and voltages in the electric grid within their respective limits. A PA is responsible for satisfying the control requirements of a single resource, i.e., controlling one part of the controlled process. For a battery resource in COMMELEC, this amounts to ensuring that the SoC of the battery never falls below a certain threshold. The PA for the battery resource sends the SoC of the battery, along with the power injected by the battery, as a part of its measurement to the controller. PAs interface with the sensors and actuators, as shown in Figure 1.1. A PA uses its actuator to implement the setpoints received from the controller. Also, a PA uses the sensor to read the state of the resource and send the read state as measurements to the controller.

The controller receives measurements from PAs, and advertisements from sensors. It computes and issues setpoints to PAs and actuators. We make a distinction between measurements from PAs and advertisements from sensors, because they have different synchrony properties. On the one hand, measurements are synchronously generated, *i.e.,* follow a particular pattern of generation because the PAs respond to the controller only when they receive setpoints. On the other hand, advertisements can be asynchronously generated in no particular relation with setpoints. Notice that PAs are synchronous, whereas controllers can be asynchronous, as controllers issue setpoints whenever required by the process, for instance due to timeouts.

In COMMELEC, the sensors are PMUs [19, 20] that stream voltage and current phasors to the controller every $20\ ms$. PMUs are periodic sensors, hence follow some synchrony. Examples of asynchronous sensors are found in manufacturing plants, where temperature and pressure sensors send advertisements only when the reading crosses their threshold. Examples of actuators in COMMELEC are converters on batteries or PV panels, and circuit breakers.

We limit ourselves to CPSs that use a centralized controller, as shown in Figure 1.1. CPSs with multiple decentralized controllers that perform distributed control, such as [21,22,23,24], are not considered. Distributed control schemes are more scalable and robust to the failure of one of the controllers than their centralized counterparts are. However, the analysis and design of distributed control schemes is more complicated than that of the centralized control schemes [25]. Specifically, a major concern of the distributed schemes mentioned in [26] is that the control performance might be suboptimal when compared to centralized scheme, because the distributed controllers do not have as much information regarding the state of the process as the centralized controller does. Alternatively, as the controller in the centralized scheme is a single point of failure and the communication with the PAs is key to achieving the desired control, such CPSs require the additional reliability mechanisms proposed in this thesis.

We notice that some computer systems, such as SDN [27,28] and NFV [29,30], also follow the architecture in Figure 1.1 – except these systems do not control a physical process but control the packet-forwarding rules in a network. For instance, in [31], the PAs are switches that send their routing tables and port information in the form of measurements to the controller that responds with routing updates to be installed on the switches as a part of setpoints. Thus, the reliability mechanisms proposed in this thesis are also extended to apply in the context of SDN.

1.2 Global Trends and Reliability Issues in CPSs

Here, we present the motivation for our research. We discuss the current trends in how CPSs are designed and what kind of applications they are used for. We argue that the new applications have stricter real-time and reliability requirements, whereas the trends in CPS design make it difficult to achieve these requirements by introducing unreliable components. Therefore, we need additional mechanisms to achieve the desired reliability in real-time.

1.2.1 Real-Time Control

Emerging CPSs are *real-time* systems [32, 33]. The frequency of control by a CPS that qualifies for real-time depends on the dynamics of the physical process. If the frequency of control is faster than or comparable to the rate at which the process demonstrates an observable change, then the CPS is said to be a real-time system. For instance, the state in electric grids is observable every 20 ms. In non-real-time CPSs, proposed in early 2010s [5,34], the control is done every few seconds or minutes, whereas recent CPSs [1, 4, 7] perform sub-second control, making them real-time systems.

The real-time paradigm shift is brought about by applications such as unintentional islanding [35], virtual power-plants [18] and teleprotection [36]; in contrast to less latency-critical applications such as demand response [34] and voltage control [5]. Similar trends are also being observed in autonomous cars with the move towards level 5 autonomy (under all roadway and environmental conditions that can be managed by a human driver) requiring sub-second control [37].

This trend poses stringent timing requirements on the control action by the CPS. For example, teleprotection applications require that the setpoints be implemented within $4\ ms$ after a voltage violation in the grid [36].

1.2.2 Complex Control and COTS Components

The CPS controllers are now solving complex problems in the real-time path. Some examples of these problems are load-flow computations [38, 39], gradient descent [7], linear projections [4], non-linear optimization, pattern recognition [8, 12] and machine learning, all of which require significant computing power, memory and high-level mathematical abstractions. In contrast, previous control methods [40, 41] were less computation and memory intensive. This trend is driven by the ability of the new algorithms to simultaneously address several aspects of real-world problems. Moreover, the steep rise in computing capabilities has made such algorithms feasible in real-time [42].

In order to implement the aforementioned complex control, CPS controllers exploit the computing power and high-level abstractions of COTS hardware and software components [43]. These controllers are implemented using industrial PCs such as CompactRIO (from National Instruments) [44], DAP server (from Alstom) [45], MGC600 (from ABB) [46], Automation PC (from B&R Automation) [47], as opposed to low-level PLCs [48]. These controllers run off-the-shelf Linux operating system with a real-time patch [49], and the controller software is developed in high-level programming languages [50] such as C, C++ and Python.

As the controllers use COTS hardware and software components they might sometimes miss the deadline for performing the control action. This is evident from Figure 1.2 that shows the response time of the COMMELEC [4] controller controlling a single PA and running on an laptop computer with 2.3 GHz Intel Core i7 processor and 4 GB DDR3 RAM. The data represents an execution trace of 160 days, which amounts to 140 million samples. The median computation time is $0.47\ ms$ and the 99^{th} percentile value is $1.09\ ms$. Whereas the 99.99^{th} percentile of the response time $(1.89\ ms)$ is only 4 times the median, the 99.999^{th} percentile is $305\ ms$, hence higher than the median. The probability that the response time in a control cycle is larger than $40\ ms$ (the deadline in COMMELEC) is 10^{-5}, which amounts to 0.36 deadline violations/hr, *i.e.,* about 8 deadline violations a day. Such deadline violations are termed as delay faults. The instances of these delay faults are usually bursty, thus making the controller unavailable for several consecutive rounds.

Modern CPSs can tolerate such deadline violations because they are *soft real-time* systems: Infrequent deadline misses are tolerable and the usefulness of a setpoint degrades after the deadline has passed; as opposed to *hard real-time* systems, *i.e.,* missing a deadline causes a system failure [51]. However, in soft real-time systems, as the duration since the deadline increases beyond a threshold, the setpoint becomes *unsafe* to implement. For example, in COMMELEC, a setpoint computed for a resource with a certain knowledge of the line currents in the grid becomes unsafe to implement once the line currents have sufficiently changed. Thus, tolerating delay faults involves

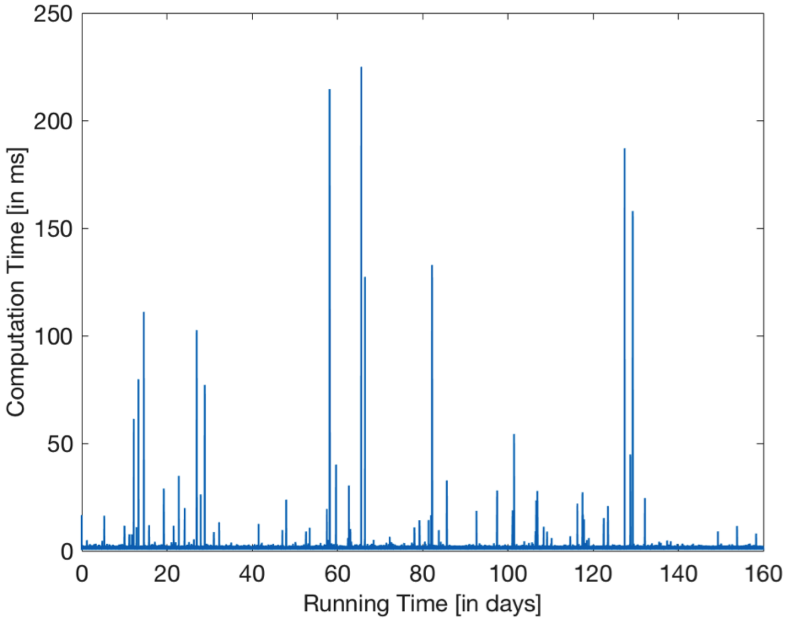


Figure 1.2 – Response time of the COMMELEC controller as a function of time

not only ensuring that setpoints are issued within a deadline but also ensuring that unsafe setpoints are not implemented.

1.2.3 IP-Based Communication

CPSs are being used to control physical processes that are spread over larger distances, and the communication between different agents takes place over several LANs; thus making it one larger IP-based WAN [52]. For instance, in [53], the real-time monitoring infrastructure is spread over a campus-wide IP network. Furthermore, these networks use COTS hardware (such as off-the-shelf switches and routers) that cannot provide the same real-time packet-delivery guarantees as traditional communication frameworks in control systems (CAN bus [54], FlexRay [55], AFDX [56], time-triggered Ethernet [57]). IP-based networks are probabilistic synchronous [58], *i.e.,* when the network conditions are “good” the messages are delivered within a bounded-delay, but when the network conditions are “bad”, the messages might be dropped, delayed or reordered. Communication networks built from COTS hardware have a PLR of 10^{-3} losses/hr and component failures at a rate of 10^{-5} failures/hr [59, 60].

To illustrate the effect of packet loss on the control by a CPS, we present results from a case study performed with the COMMELEC CPS using T-RECS [61], a virtual-commissioning system for the real-time control of electric grids. In this study, the COMMELEC controller is required to track an external dispatch signal that specifies the grid prosumption in intervals of five minutes. To this end, it must follow the dispatch plan as closely as possible while satisfying the constraints of the grid and a single bat-

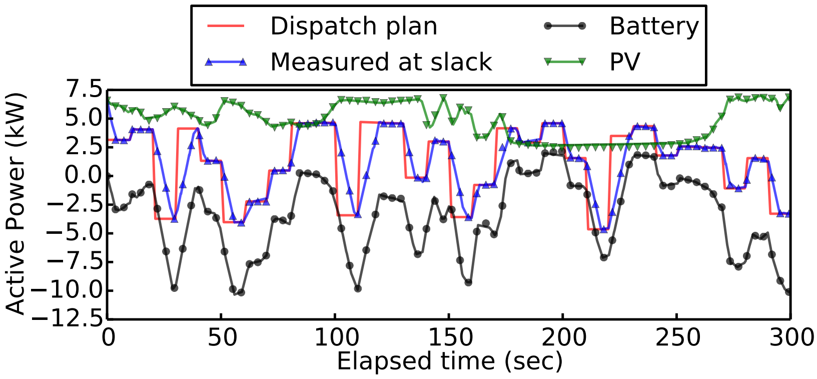


Figure 1.3 – Tracking a dispatch plan with the COMMELEC controller

tery resource, in the presence of an intermittent PV resource. The performance metric is energy mismatch that is defined as the cumulative absolute difference between the dispatch plan and the power prosumed by the grid. The maximum tolerable energy mismatch is taken as 50 kWh in a day, which amounts to 0.17 kWh in five minutes.

Fig. 1.3 shows the dispatch plan and the actual power measured at the slack bus for a duration of five minutes. To track this dispatch plan, the GA accommodates the variations in PV production by changing the power injected by the battery. Positive power represents production. Notice that the dispatch plan is discrete and has abrupt changes, whereas the path followed by COMMELEC is smooth to accommodate the inertia of the grid and the resources.

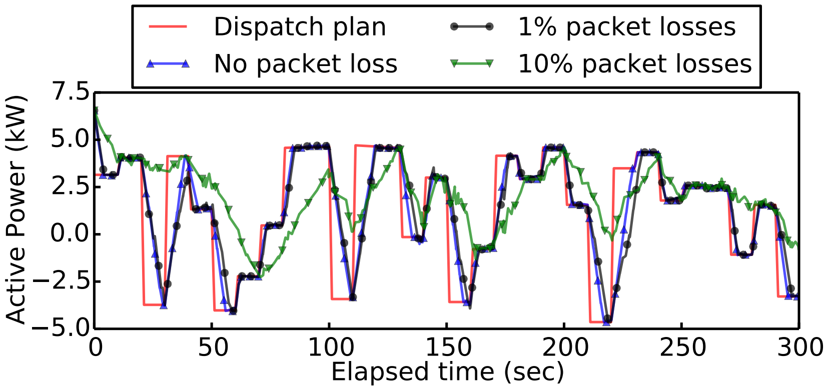


Figure 1.4 – Effect of network losses on COMMELEC's ability to track a given dispatch plan

Fig. 1.4 shows the effect of packet losses in the communication between the GA and PAs on the tracking performance. Although the requested power is tracked with high accuracy in the absence of packet losses and there is a slight deviation (when compared to scenario with no packet loss) when there is a 1% packet loss. This deviation is more profound with a higher loss rate of 10%. Figure 1.5 shows the energy mismatch as a function of the PLR. We see that the energy mismatch grows steadily with a higher PLR. At 7% packet loss, the energy mismatch exceeds the maximum energy mismatch of 0.17 kWh, during an interval of five minutes. Although this violation has no effect on the safety of the grid, it could have economic repercussions for the operating authority.

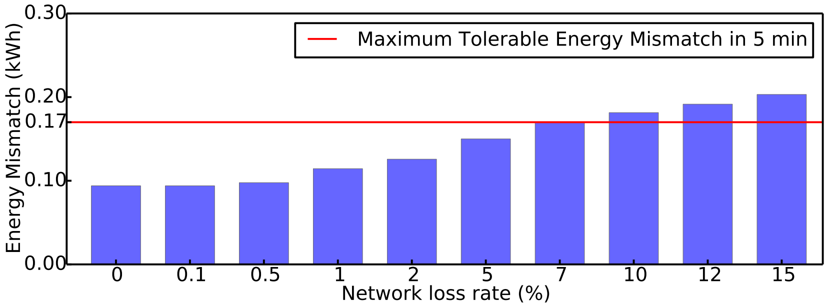


Figure 1.5 – Energy mismatch between the dispatch plan and that provided by COMMELEC during a period of five minutes

In the absence of cabling infrastructure, some CPSs also use Wi-Fi and LTE based communications [62, 63, 64] that have a higher PLR than wired networks, thereby adversely affecting the quality of the control. Therefore, the need to provide reliable communication with fast repair of packet-losses (so that the quality of control by the CPS is unaffected) is becoming more pressing.

1.2.4 Mission-Critical Applications

The loss of control by a CPS, due to the inability of the controller to perform the desired control action within the required deadline, can cause damages to life, property and business. The Northeastern Blackout of 2003 resulted in a 2-day power-outage in USA and Canada that caused 6 billion dollars in economic loss and 11 deaths [65]. One of the many causes of this blackout was a large delay in the energy management system [65, 66]. In factory automation, the average cost of downtime (the time during which the normal operation of a plant is halted) is estimated to be $12,500\$/hr$ [67, 68]. Similar threats are faced by autonomous cars [69, 70, 71].

We find, based on these trends, that CPS applications are becoming mission-critical and have stricter real-time requirements. Alternatively, they are opting for flexibility of COTS components, which makes them susceptible to crashes and delays of the controllers, and to packet-losses and delays in networks. These two conflicting trends result in the need for mechanisms that enable highly reliable, real-time CPS from COTS components. For mission-critical applications, high reliability is measured in terms of the number of *nines* of availability [72], for instance 5 nines of availability implies that the CPS is available 99.999% of the time. This implies that the downtime does not exceed 5.26 minutes a year, *i.e.*, $864\ ms$ a day.

1.3 Challenges

Achieving high reliability entails (1) ensuring enough redundancy of the controller and avoiding a single point of failure in the communication network so that control of the process continues despite faults, (2) detecting and repairing controller faults and packet losses as quickly as possible, and (3) ensuring that the faults in the software agents and the communication network do not effect the correctness of the control. Furthermore, the item 3 above involves ensuring the following properties. First, PAs might receive setpoints that have violated the deadline, hence they are unsafe and must be prevented from being implemented. Second, controllers must be prevented from using measurements that represent an older state of the process (as the state of the process changes with both time and implementation setpoints). Lastly, avoiding *split-brain* [73] among the replicated controllers to prevent the implementation of inconsistent setpoints at different resources.

Realizing (1) requires that the redundant components follow design diversity [74, 75], *i.e.,* are developed independently to ensure fail-independence. This is often done in controller design for airplanes [76, 77], where three different teams program three different controllers in different operating systems and programming languages. This approach is both expensive and practically infeasible for COTS-based CPSs, because most of the off-the-shelf operating systems share the same code-bases and many software libraries are shared between different high-level languages [78].

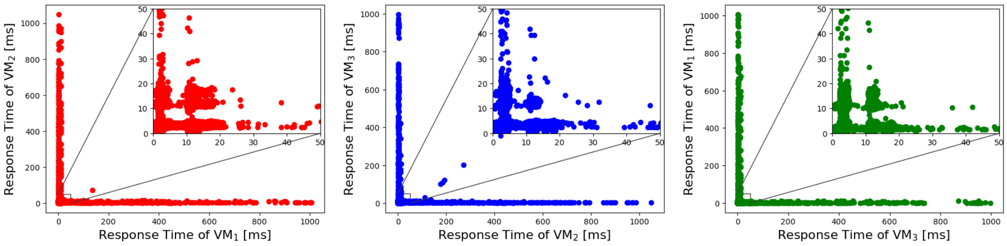


Figure 1.6 – Pairwise scatter plots of response times of replicas of the same COMMELEC controller running on different virtual machines on the same physical machine

Design diversity is a prerequisite for tolerating Byzantine faults [79], such as the controller replicas issuing incorrect setpoints or security attacks [80] such as a compromised controller maliciously steering the process into an undesirable state. In this thesis, we do not address such faults and focus solely on crash and delay faults. Fortunately, the large variations in the timing performance of modern software and the several layers of the hardware-software stack enables us to circumvent this challenge, and thus provide delay and crash fault-tolerance. Figure 1.6 shows the scatter plot of response times of the three identical replicas of the COMMELEC controller that runs on three different virtual machines hosted on the same physical machine. We find that there is little correlation between the large delays of the controllers; this indicates that identical replicas can be used to mask delay faults.

A key challenge in avoiding a single point of failure in the communication network in (1) is working with IP networks. The traditional approach to this problem is to have cloned disjoint networks [81]. However, IP networks are often connected for ease of management and diagnostics. It is desirable to make little or no modifications to the existing communication infrastructure. Hence, the communication reliability mechanisms must provide high-reliability, despite heterogeneous non-fail-independent networks. Furthermore, the software agents in a CPS often use IP multicast for communication. The challenge then is to provide reliability with multicast, and to be transparent to the network.

Realizing (2) is particularly challenging because of the transient nature of delay faults that makes them difficult to detect. Although aggressive fault-detection improves availability by faster detection, it might introduce false positives that reduce availability, as a correct controller replica might be stopped to be recovered. Alternatively, conservative fault-detection ensures that only truly faulty replicas as detected, but it induces large detection delays which reduce availability.

Mechanisms for realizing (3) have been the focus of several decades of research in distributed systems [82,83,84,85,86,87]. The purpose of these mechanisms is to ensure agreement across replicated controllers in order to avoid the split-brain syndrome. Agreement is often achieved through consensus among the controllers. According to the FLP impossibility result [88], consensus in an asynchronous system can take an unbounded amount of time to terminate. CPSs are asynchronous and require strong consistency (or simply consistency) among the controller replicas provided by consensus, as opposed to eventual consistency [89] provided by other approaches [90,91]. However, as CPSs have real-time constraints, they cannot tolerate unbounded latency due to consensus mechanisms. Hence, new mechanisms need to be developed in order to provide the desired consistency with a low latency-overhead.

Lastly, the physical process and the controlled resources are integral components of a CPS and introduce new requirements that do not exist in classic computer systems. This introduces other challenges, such as formalizing the correctness criteria of the physical process, and abstracting the state of the physical process for use by the reliability mechanisms.

1.4 Contributions

Our contributions in this thesis can be summarized as follows.

1. We describe high-level abstract models of the software agents in a CPS, namely

controller and PAs. These models apply to a wide-range of CPSs in the domain of electric grids [1,4,5,6], manufacturing processes [13], autonomous vehicles [8,9,12] and SDN [27]. They can be used to characterize both new and existing CPSs in various domains. These models enable us to better understand the reliability requirements and design the reliability mechanisms.

2. We abstract the execution trace of a CPS by using the events in the CPS and two basic relations between those events, namely, the network relation and the computation relation. We use these relations to define a new relation called the intentionality relation; it better expresses the ordering of events among themselves and with respect to the state of the controlled process. Thus, it is useful to formalize and to prove the correctness properties of a CPS. We use the abstract model from item 1 and the intentionality relation to formalize four correctness properties of a CPS. These are state-safety, optimal selection, consistency and timeliness.
3. We present intentionality clocks, a labeling mechanism that uses logical clocks adapted from Lamport clocks [85] for ordering events with respect to the intentionality relation. We present the new design of a controller and a PA that uses intentionality clocks for guaranteeing the state-safety and optimal selection correctness properties. We formally prove the guarantees provided by the new design.
4. Through a case study with an existing CPS for the optimal charging of EVs [1], we show how violating the state-safety or optimal-selection correctness properties can affect the CPS. In this case, the CPS enters a deadlock due to violation of optimal selection. We show that such situations can be avoided by following our design that requires minor modifications to the existing design.
5. To ensure consistency in the presence of replicated controllers, we propose Quarts, a low-latency agreement algorithm that uses the existence of labeled measurements from intentionality clocks in item 3 to perform agreement on the measurements used for computation by the controller, as opposed to performing agreement on setpoints done in conventional agreement mechanisms. Quarts is designed for CPS with only controllers and PAs. We also present, Quarts+, an extension to Quarts that maintains similar performance with addition of asynchronous sensors.
6. We prove that Quarts and Quarts+ guarantee the consistency correctness property and that they add a bounded latency-overhead. Moreover, through extensive simulations, we demonstrate that Quarts and Quarts+ improve the availability of the CPS by over an order of magnitude, when compared to other consistency-guaranteeing agreement mechanisms.
7. We use the controller consistency provided by Quarts from item 5 to design Axo, a delay- and crash-fault tolerance architecture that uses active replication of the

controllers. We formally prove that Axo guarantees the last remaining correctness property, *i.e.,* timeliness, and maximize availability.

8. We derive bounds on the time Axo takes to detect and recover delay-faulty and crash-faulty controller replicas. We validate the derived bounds by implementing Axo on software-based controllers and measuring the time Axo takes to detect and recover the replicas in a virtualized environment.

9. We implement the reliability mechanisms, intentionality clocks (item 3), Quarts, Quarts+ (item 5) and Axo (item 7), in the COMMELEC controller [4]. Through this exercise, we demonstrate that the proposed reliability mechanisms can be easily implemented on CPS controllers with minimal modifications to the existing software.

10. We deploy the new, reliable COMMELEC controller in a full-scale replica of the CIGRÉ low-voltage benchmark microgrid on the EPFL campus [92], and also in the T-RECS virtual-commissioning system for real-time control of electric grids. We compare the performance of COMMELEC, with and without the reliability enhancements.

11. Examining the model of the CPS from item 1, we identify that SDN fits this model and that the proposed reliability mechanisms can be extended to design a highly available SDN control-plane. To this end, we present QCL, a transparent coordination layer between single-image SDN controllers such as POX [93] and RYU [94] and SDN switches, which guarantees control-plane consistency with low latency-overhead. We prove the consistency guarantees of QCL. We show that it can be used to correctly enforce any routing policy or controller application that is supported by the underlying single-image controller.

12. We evaluate the performance of QCL through simulation and show that QCL drastically reduces the tail response-times and convergence-times in an SDN network when compared to other consistency-guaranteeing replication mechanisms. Reducing tail latency is one of the key requirements of modern datacenter networks [95]. We implement QCL along with the POX single-image controller and study its performance in 4-port fat-tree datacenter network [96] in a virtualized environment [97].

13. To provide reliable communication between the software agents in an IP-based network, we propose iPRP, an IP-friendly parallel redundancy protocol. The key contribution in the iPRP design is achieving replication of packets such that it is transparent to both the control application and the network.

14. We study the fail independence of directional Wi-Fi links, through a measurement campaign from a deployment of redundant Wi-Fi links on the roof-tops of the EPFL campus. We show that, although the links are not strictly fail-independent,

the aggregate PLR with two links is very close to what it would have been if they were fail-independent. Hence, we conclude that iPRP can be used to enhance the reliability of Wi-Fi links, in CPSs where wired communication is not possible.

1.5 Roadmap

In Chapter 2, we review the literature on existing reliability mechanisms developed for distributed systems. Then, in Chapter 3, we introduce the high-level abstractions for a controller and a PA; and we define the intentionality relation and three of the correctness properties of a CPS that we will be addressing in the rest of this thesis. These properties are *state safety*, *optimal selection* and *consistency*.

In Chapter 4, we present intentionality clocks that can be used to order events in a CPS according to the intentionality relation. We present new designs of CPS controller and PAs that implement intentionality clocks, and we prove that this design guarantees state safety and optimal selection. We conclude this chapter with a case study of a CPS for charging EVs.

In Chapter 5, we present Quarts, a bounded latency-overhead agreement mechanism that guarantees the consistency correctness property in a CPS with one or more delay- or crash-faulty controller replicas and any number of PAs. Then, to continue providing the same guarantees in the presence of asynchronous sensors, we propose an extension, Quarts+. We prove the consistency and bounded latency-overhead properties of Quarts and Quarts+, and we compare their performance with existing reliability mechanisms for guaranteeing consistency.

In Chapter 6, we define the last correctness property addressed in this thesis, namely timeliness. Then, we fit all the reliability mechanisms together to obtain Axo, a delay and crash fault-tolerance mechanism for CPS controllers. Axo uses active replication of the controllers and relies on Quarts for controller consistency. It exploits the fact that timing profiles of different controllers are not significantly correlated. Thus, when one of the controller replicas is delay faulty, another replica can continue to provide the desired control. Besides improving availability, Axo guarantees the timeliness property that requires that non-timely setpoints (unsafe) are never implemented. Additionally, it detects and recovers delay-faulty and crash-faulty controller replicas. We conclude this chapter with results from deployment of the reliability mechanisms with the COMMELEC controller and an LQR [98] controller for an inverted pendulum.

In Chapter 7, we apply the proposed reliability mechanisms to provide control-plane consistency with low latency-overhead in SDN. We present the design of this new system, QCL, and we prove that it achieves the strong consistency guarantees. Through an implementation of QCL, we show that QCL is transparent to the SDN controllers and switches. We evaluate its performance in simulation and present results from deployment in a virtualized datacenter network.

In Chapter 8, we present iPRP, an IP-friendly parallel redundancy protocol for repairing packet-losses in real-time CPSs. We also evaluate the feasibility of using iPRP to provide high-reliability with unreliable, directional Wi-Fi links, through a 45-day long measurement campaign on the EPFL campus.

In Chapter 9, we conclude the thesis by summarizing our findings. We also present our vision for the future of COTS-based real-time CPSs and the research challenges that need to be addressed in realizing that vision.

2 State of the Art

Lives of great men all remind us
We can make our lives sublime,
And, departing, leave behind us
Footprints on the sands of time;
— H. W. Longfellow, A Psalm of Life

In this chapter, we review the current mechanisms for providing controller reliability and communication reliability. We discuss their inadequacies in addressing the requirements of real-time CPSs, specifically in addressing delay faults in controllers and fast packet repair in IP networks, thereby underpinning the need for new reliability mechanisms.

2.1 Controller Reliability

Reliability in the presence of software and hardware faults of the CPS controller is provided by redundancy, *i.e.,* replication of the controller. The replication techniques can be broadly classified into two categories: (1) primary-backup replication [99], also called passive replication, and (2) state-machine replication [100], also called active replication. These techniques are discussed in the context of generic distributed systems in detail in [101]. Here, we will discuss these techniques in the context of CPSs.

The basic correctness criterion for replication techniques is linearizability, [102] also called one-copy equivalence. This gives the illusion, to the PAs, of a single controller by ensuring that all the controller replicas process the measurements in the same order and issue the same setpoints, thereby making the replication transparent for the PAs.

The first property of linearizability is *order*: If two controller replicas both perform two computations using one of the measurements m_1 and m_2 in each computation, then they use m_1 and m_2 in the same order. Linearizability in classic distributed systems, such as database systems, also includes the atomicity property: if one controller replica performs computation, then all correct controller replicas perform the same computation. CPSs do not need atomicity, rather a weaker property, *consistency*. It states that if two controller replicas issue setpoints for the same PA in the same control round, then the issued setpoints are the same. Consistency is easier to provide than atomicity. This enables low-latency reliability mechanisms that satisfy the real-time constraints of CPSs.

Next, we will describe passive and active replication techniques in detail.

2.1.1 Passive Replication

Passive-replication techniques [86,99,103,104] are also called primary-backup techniques because they use one primary controller to receive measurements from the PAs and issue setpoints to the PAs, and one or more backup (or standby) controllers that monitor the primary replica for faults.

When the primary controller receives measurements, it computes setpoints and sends the computed setpoints to the backups to synchronize the state of the primary replica and of the backups. Before issuing setpoints, the primary controller waits for acknowledgments of state synchronization from the backups. In the absence of faults, this scheme has a latency overhead of one RTT and trivially ensures linearizability, because a single replica (the primary) is handling all the measurements.

The backups use a failure detector [58,105] to check if the primary replica is correct. The failure detector is implemented by heartbeat mechanisms [106,107]. When the primary replica is detected as faulty, a new primary is elected through a leader election mechanism [108, 109]. Leader election requires that all the backups agree on the new chosen leader. This is a consensus problem and can take unbounded time in an asynchronous network, as discussed below.

Consensus is a fundamental problem in distributed computing that is the basic building block of several abstractions such as leader election, failure detection [105], group membership [110]. Solving a consensus problem requires agreement on a data value among a number of (faulty or non-faulty) agents. Protocols that solve consensus, must satisfy the following properties [111]:

- **Termination:** Every correct agent decides some value
- **Integrity:** If all the correct agents proposed the same value v, then any correct process that decides a value, decide v
- **Validity:** If a process decides a value v, then v must have been proposed by some correct process

- **Agreement:** All correct processes must agree on the same value

According to the FLP impossibility result [88], the termination and agreement property cannot be guaranteed together in an asynchronous setting. CPSs communicate with an asynchronous network that might drop, delay or reorder messages, and the software agents might take unbounded time to respond due to software faults. Thus, all problems that require solving consensus, cannot guarantee termination in a bounded time. As CPSs have real-time requirements, the mechanisms that rely on consensus are often not suitable. In the case of leader election for choosing a new primary in passive replication, the CPS remains unavailable during the time it takes for consensus to terminate, thereby reducing availability.

Additionally, designing a perfect failure detector (an instance of the consensus problem) is also proven to be impossible in an asynchronous setting [105]. Therefore, the CPS can have multiple primary replicas in case of a partition. Also, most failure detectors work best for crash-only faults. However, delay faults follow the crash-recovery model (are transient), wherein a replica that could not issue setpoints due to a delay fault can issue the same setpoints after the fault has passed. In this case, a primary replica that was detected as faulty by the failure detector can issue setpoints after it stops being delayed. As a primary replica does not expect to have other primaries at the same time, it does not perform agreement to ensure the order or consistency properties. Hence, having multiple primaries might violate linearizability. We explore this in detail in Chapters 4 and 5.

Moreover, implementation of a setpoint issued by a delayed controller can be unsafe for the CPS. This is because the state of the process used for computing the setpoint might be significantly different from the state of the process at the time of implementing the setpoint, possibly resulting in a incorrect behavior. We further explore this problem in Chapter 6.

In passive replication, the backup replicas cannot issue setpoints to the PAs. Furthermore, as the process of detecting a fault and performing a failover (by leader election) is time-consuming, the CPS is unavailable is during the time the primary is faulty. We conclude that passive-replication techniques are not best suited for tolerating delay faults. The availability can be improved by faster failure-detection. However, a faster detection can lead to more false positives, whereby a non-faulty primary is wrongly detected as faulty. This increases the chances of multiple primaries at the same time, thereby violating linearizability.

In practice, passive replication can use hot or cold standbys. Hot standbys are faster in resuming control after a failure of the primary but need longer state-synchronization after each computation. Both these techniques suffer from possible violations of consistency and poor availability in the presence of delay faults.

Moreover, availability of hot standbys can be also increased by checkpointing techniques [112], where the state of the primary is constantly logged and the newly elected primary is instantiated form the last logged state. This state includes the internal controller state such as received measurements and the state external to the control logic such as that of the operating system (memory address space, file descriptors, signal handlers, etc.). However, these techniques add latency in the real-time path and do not address the issue with multiple primaries.

Lastly, we note that passive-replication techniques need only one primary to perform computation and issues setpoints. Thus, they need $f+1$ replicas to tolerate f faulty replicas.

2.1.2 Active Replication

In active-replication techniques [76, 100, 113, 114, 115] all replicas receive measurements from the PAs and compute and issue setpoints to the PAs. As the state is replicated across at each replicas, these techniques are also called state-machine replication techniques. Thus, when a replica is delay or crash faulty, others replicas continue to control the CPS, thereby remaining available.

In contrast to passive replication, active-replication techniques require that the CPS controller be deterministic. This property is satisfied by most CPS controllers [1, 4, 7, 13, 41, 93, 94, 116], thus allowing for active replication techniques to be applied to CPSs.

In order to satisfy the linearizability correctness criteria, active-replication techniques need to ensure order and consistency. One approach to provide order is to ensure that all non-faulty replicas receive measurements from the PAs in the same order. This is achieved by total-order multicast [117] that relies on a failure detector and that in turn uses consensus. Alternatively, the replicas might also choose one replica to issues setpoints to the PAs, using a leader-election mechanism that also relies on consensus.

Order can also be achieved through a labeling scheme obtained using timestamps from time-synchronized physical clocks, or logical clocks such as scalar clocks [85] or vector clocks [118]. Timestamps or clocks are used to label messages/events, and thus order them, according to a relation. The conventional relation used in distributed systems is the “happened-before” relation introduced in [85]. However, this relation is unable to relate the events to the evolution of the state of the physical process as shown in Chapter 4. Thus, the resulting ordering of events is not applicable to the CPSs. We address this problem by introducing a new relation, called the intentionality relation, and propose a labeling scheme named intentionality clocks in Chapters 3 and 4.

To provide consistency among replicated controllers, active-replication schemes perform consensus to reach agreement. They either choose the setpoints that will be issued to the PAs or choose the replica that will send the setpoints to the PAs. As mentioned in Section 2.1.1, consensus in an asynchronous system has a poor latency performance and in the worst case can add unbounded latency as proven in [88]. We found that for a large class of CPSs, the termination property of consensus can be relaxed. In other words, a CPS controller can compute and issue correct setpoints in a round, even if it failed to compute in the previous round. Thus, for such CPSs, the FLP impossibility result can be avoided, and agreement can be provided without using conventional consensus. This is property is used in Chapter 5 to design a low-latency agreement mechanism to ensure consistency.

Some systems such as transactional database systems require a weaker form of consistency called eventual consistency [89]. Eventual consistency ensures that after a sufficiently long quiescent period during which no events occur, all controllers will eventually have the correct view of the system, thereby ensuring consistency. However, there might exists periods of time during which consistency might be violated. Mechanisms that ensure eventual consistency typically have a low latency-overhead. This property is used by SCL [31], to design a low-latency fault-tolerance protocol for SDNs. However, as we describe in Chapter 7, SDNs (CPSs in general) require the strong-consistency model, and cannot benefit from eventually consistent schemes.

Note that, active-replication techniques require $2f+1$ controller replicas, to tolerate f faulty replicas, which is f replicas more than passive replication. However, as only active replication can tolerate delay faults, we will use it in Chapter 6 and rely on a the new, low-latency agreement mechanism that has a lower replication-cost that conventional consensus.

In active replication, when the controller replicas are assumed to be non-malicious, like in the case of delay and crash faults, the PAs accept the first received measurement and implement it. However, active replication can also tolerate malicious or Byzantine faults [119] using special techniques under the class of BFT protocols [79, 120, 121, 122]. BFT protocols add a minimum of 2 RTT delay and suffer from the same unbounded-latency problem as non-BFT active-replication techniques. As delay faults follow the crash-recover model and not fail-stop model of crash-only faults, BFT protocols could be a possible solution to tolerate delay faults. Such a scheme could mark a delay-faulty setpoint as unsafe and treat it as a setpoint issued by a malicious agent. However, their poor latency performance makes using BFT protocols inefficient.

Availability in active replication can be increased by proactive recovery techniques [123, 124] that recover (or reboot) a replica before it is detected as faulty in order to maintain enough non-faulty replicas at all times. These techniques monitor, over a long period of time, the performance of the software and use software aging techniques

[125, 126] to decide if the replica must be rebooted. Although such techniques have been successfully applied to study aging in the operating systems [127] and web servers [124, 128, 129], the impact of software aging on latency performance of CPS controller is not well-understood. The authors in [130] show that delay faults in a CPS do not follow any particular pattern, making it hard to decide when to perform proactive recovery. In any case, proactive recovery and software aging techniques do not remove the need for agreement between the controller replicas, which is the central problem with active replication.

Instead of tolerating faults by redundancy, a different school of thought aims to prevent faults by designing controllers that never exceed a certain WCET. Such techniques [131, 132] require using specialized hardware and software components. For example, the authors in [132] propose the TCB architecture, that employs a hard real-time module to implement the time-critical operations on the CPS controller. In this way, core components of the CPS controllers are designed to never experience delay faults. A similar approach is taken by TTA [131], where all the operations of the controller are appropriately scheduled based on their WCETs such that the no deadlines are violated. While these approaches prevent delay faults in theory, they are inapplicable to COTS-based CPSs that heavily rely on general-purpose, off-the-shelf hardware and software components.

2.2 Communication Reliability

The most common technique for providing reliable communication is to use the TCP transport layer that ensures reliable in-order delivery of messages by detecting packet losses and repairing them by retransmissions. The in-order delivery mechanism of TCP prevents packets from being forwarded to the application layer at the receiver until all previously sent packets have been delivered. The stalling of new packets is detrimental in CPSs where newer packets supersede older ones as they represent the more recent state of the controlled process. This issue, called head-of-line blocking [133], renders TCP unsuitable for such real-time traffic. Consequently, solutions based on TCP, such as MPTCP [134], also suffer from the same issue.

Recently, a UDP-based reliable transport protocol, QUIC [135, 136] was proposed. QUIC uses multiple streams and provides in-order delivery of packets within a stream, thereby alleviating the head-of-line-blocking problem. In both QUIC and TCP, the time after which a packet is retransmitted is determined by the RTO that is used to estimate the RTT of the network. The RTO is doubled after each packet loss [137]. Thus, in cases of bursty losses or component failures, the time to repair a packet loss increases indefinitely [138]. Moreover, both TCP and QUIC, *i.e.,* temporal redundancy techniques, on their own, cannot provide reliability against component failures that require sending packets along a different functional path. Other variations of temporal redundancy include coding techniques [139] that also suffer from the same issues as TCP and QUIC, in addition to added encoding and decoding delay. Lastly, TCP and QUIC are connection-oriented transport protocols and do not support connectionless IP-multicast communication, a key requirements in many CPSs where multiple receivers can asynchronously tune-in to a multicast group.

To address component failures, spatial redundancy is exploited wherein, when a failure is detected, the retransmitted packets are routed through a different path. The simplest way to achieve this is through routing protocols such as OSPF [140], RIP [141] and RSTP [142]. These protocols detect a link or router failure through signaling mechanisms and take few seconds to converge on the new route, thus restoring connectivity. The time to restore connectivity and repair packet losses can be further improved by pre-computing the backup routes. This is done by in RPL [143] and MPLS-TP [144], which perform a switchover to the pre-computed path when a failure is detected. The most promising of these solutions, MPLS-TP can provide fail-over times as low as $50\ ms$ that is still high for several real-time CPSs applications, for example high-speed teleprotection, load shedding, and PMU data delivery in smartgrids [52].

To repair packet-loss in $0\ ms$, PRP [145, 146] was conceived in the context of real-time communication in substation automation. The end-hosts that use PRP either have two interfaces connected to two fail-independent networks or provide the two required interfaces by using a specialized multi-interface middle box called red box [146]. PRP requires that the two fail-independent networks be clones of each other, each network be a single LAN with no routers, and the two interfaces of an end-host have the same MAC address. Each MAC frame at the sender is transmitted over the two networks, along with a sequence number. The receiver forwards the first received copy to the application. At the receiver, a MAC frame with a previously registered source MAC address and a sequence number pair is discarded, thereby ensuring that only one copy of a message is received by the application.

Although PRP provides the desired fail-over time, it is not natively applicable in CPSs that communicate in an IP-based network, *i.e.,* a WAN, for the following reasons. First, PRP inserts the sequence numbers and other control information as a trailer to the MAC layer. Each router in IP network strips the MAC header and creates a new one for the next hop. Thus, the PRP control information is lost and a duplicate discard cannot be performed. Second, as PRP is a MAC-layer solution, it does not natively support IP multicast. These two issues can be remedied by creating an overlay network using VLANs [147] or virtual private network (VPN) [148]. These however, add extra latency in the real-time path. Another drawback of PRP is that its duplicate-discard algorithm ensures that at most one copy of a packet is forwarded to the application only when messages are delivered without reordering [81], as found in a LAN environment. However, in WANs, middle-boxes such as scrubbers and load-balancers might cause packets of the same flow to be delivered out-of-order [149]. The PRP discard algorithm

[81] can deliver duplicate copies to the application, in scenarios with packet reordering.

The authors of [150] propose modifications to PRP so that it can be applied in WANs. However, these modifications were neither fully designed or implemented, and do not address the the comparability with the TCP/IP stack. Moreover, these modifications require that the intermediate routers be modified to accommodate the replication header, making it difficult to deploy in existing networks. We address these problems through iPRP in Chapter 8, that is IP friendly and designed to be transparent to the network.

3 CPS Model and Intentionality

I may not have gone where I intended to go,
but I think I have ended up where I needed to be.
— Douglas Adams, The Long Dark Tea-Time of the Soul

In Chapter 1, we introduced the architecture of a CPS with one central controller, one or more PAs, and one or more asynchronous sensors. In this chapter, we present a formal model of the software agents, namely the controller and the PAs. We use this model to create an abstraction of a CPS, based on the events that occur in the CPS. Using this event-based abstraction, we characterize the interactions between the software agents and qualify how they affect the underlying controlled process. We define the desired correctness properties in a CPS. We only consider CPSs with one or more PAs, and no asynchronous sensors that directly talk to the controller. These are explicitly handled in Chapter 5.

3.1 Introduction

Figure 3.1 shows the architecture of a CPS with a central controller, two PAs, and no asynchronous sensors or actuators that speak directly with the controller. Recall from Chapter 1 that the sensors and actuators interact with the physical process, by reading and altering its state, respectively. Together, the software agents, namely controller and PAs, achieve the desired control of the physical process.

PAs are software agents responsible for controlling a sub-process, i.e., one part of the controlled process. PAs interface with the sensors and actuators, as shown in Figure 3.1. They implement the setpoints received from the controller through their actuators, read the state of the resources through sensors, and send them as measurements to the controller. PAs are assumed to be synchronous, in the sense that they do not issue measurements out of turn, but only do so in response to a setpoint

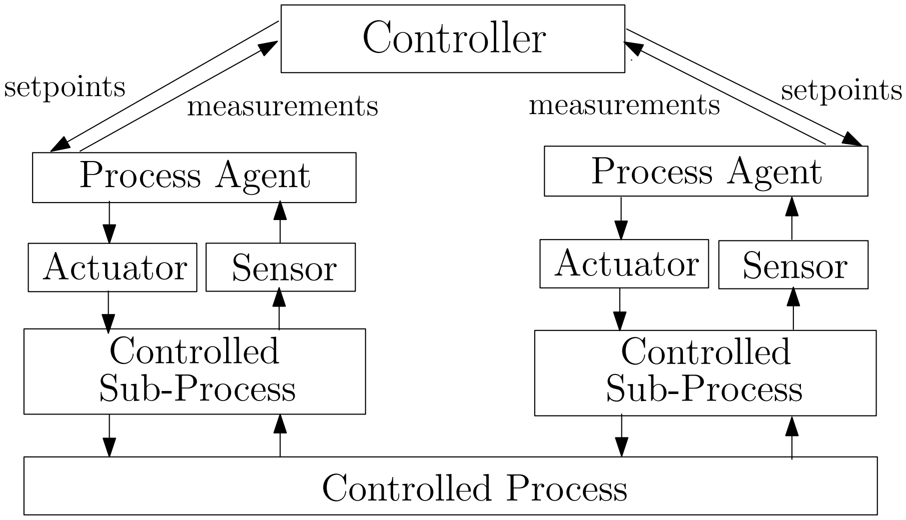


Figure 3.1 – Architecture of a CPS without asynchronous sensors and actuators that directly communicate with the controller

implementation. This makes the modeling and analysis of a CPS more tractable. In Chapter 7, we show that our techniques are amenable application for the control of CPSs with asynchronous PAs with minor modifications.

A central controller performs the high-level control of the physical process by receiving measurements from PAs and by sending setpoints to PAs. The purpose of the controller is to keep the controlled process in a desired state. To this end, it receives measurements from the PAs, performs computation, and issues setpoints to the PAs. The central controller is a single point of failure and is replicated for high availability. We assume the controller to be susceptible to crash and delay faults. We do not account for Byzantine faults, where a controller can behave arbitrarily in the event of a fault and might produce malicious setpoints in order to jeopardize the safety of the CPS. The delay and crash fault-model is formally defined in Chapter 5.

The controller is asynchronous, i.e., one controller replica can choose to issue setpoints at a time when another replica does not. This can occur due to timeouts. For example, the COMMELEC controller in [4] issues setpoints if $500\ ms$ have elapsed since the last time it did so, whereas normally, a control cycle occurs every $60 - 100\ ms$.

The software agents exchange measurements and setpoints using a network that can drop delay, and/or reorder messages. We assume the absence of malicious alterations or corruptions in the messages.

For the ease of explanation, we assume that the number of PAs and the number of controller replicas in the CPS is known and a constant. The addition and removal of software agents is assumed to be performed on the non-real-time path. In CPSs, where the churn of the software agents is high, the number of agents can be obtained in an out-of-band fashion by using non-group membership algorithms such as [110, 151]. In the interval between addition or removal of an agent and the new group membership reliably installed on all agents, the reliability mechanism developed in this dissertation will still continue to guarantee their correctness properties, albeit with reduced liveness.

3.1.1 State of the Controlled Process

The state of the sub-process controlled by a PA is altered by implementation of a setpoint. The new state is captured in the resulting measurement. The controller uses measurements from different PAs to recreate the new state of the entire process that it then uses to compute setpoints. The setpoints computed with this state are only valid as long as the recreated state reflects the actual state of the process. Any subsequent setpoint implementations change the process state, making the former setpoints *unsafe* for implementation. Therefore, to achieve the desired control, the state of the process at the time of setpoint implementation must be the same as that used by the controller for computing the corresponding setpoints. Hence, before implementing a setpoint, the PA must be able to ascertain whether the setpoint reflects the state it last advertised. In other words, it must be able to infer if a received setpoint was *caused by* the measurement it last advertised.

Similarly, a controller must be able to ascertain whether a measurement received from a PA represents the most recent state of that sub-process or the state corresponding to earlier setpoint implementations. This causal relationship can be better understood using the notion of control rounds. Software agents must be able to attribute a round number to received messages, to compare it with the round number they are currently executing, and to treat the message appropriately.

Note that, we use the term “state of the process” as a proxy for the state of the PAs. Although the state of the physical process is continuous and evolving, the state of the PAs is discrete and only changes upon a setpoint implementation. In real-time control, the CPS issues setpoints at a rate faster than the dynamics of the underlying process. Hence, the evolution of the state of the process between two setpoint implementations is minimal, thereby justifying our usage of the term. In cases where the latency between consecutive setpoint implementations is larger than the desired rate of control, due to a delayed controller for instance, the state of the process between two-setpoint implementations is significantly different. Such scenarios are avoided by satisfying the timeliness properties of the CPS, as discussed in detail in Chapter 6.

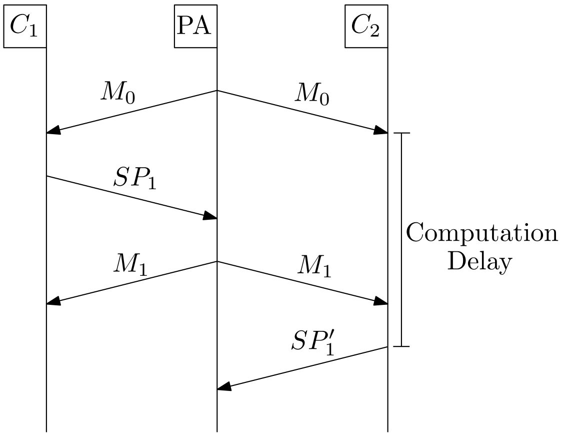


Figure 3.2 – Illustration of why the happened-before relation is not suitable for ordering events in a CPS in the presence of replication

3.1.2 Need for a New Relation

When the controller is replicated, assigning a consistent round number to events traditionally requires consensus between the replicas. For example, to order events across replicated processes in classic distributed systems, the processes in [86] undertake consensus with several rounds of message exchanges. Due to network losses and delays, and due to software faults, consensus might require unbounded time [88], making it unsuitable for real-time systems such as CPSs.

In the literature, this problem is circumvented by using message labels (that represent the causal order between the messages) to infer the round number. The causal order between the messages is derived using the happened-before relation [85]. In the presence of replication of the controller, or of random network or computation delays, messages that intend to cause a certain effect do not necessarily succeed, due to competing messages. This is illustrated through the example in Fig. 3.2.

In Fig. 3.2, the controller is replicated, and its two replicas C and C' receive the measurement M_0 sent by the PA. This measurement is used by each replica in the computation of a setpoint, resulting in SP_1 and SP_1' in C and C', respectively. In such a scenario, SP_1 and SP_1' belong to the same "generation" or the same "control round", and are said to be equivalent. SP_1 is received by the PA, and its implementation results in M_1. However, due to a delay at C', M_1 is received before SP_1' is issued. Although M_1 can be said to have happened before SP_1', M_1 is nonetheless caused by an equivalent of SP_1', namely SP_1.

In the previous example, we say that SP_1' intends to have caused M_1, but it did not succeed because a competing equivalent event (SP_1) was received earlier. In order to formally capture this phenomenon, we introduce the *intentionality* relation. This relation enables us to implement a solution to the ordering problem. The implementation is described in Chapter 4.

3.1.3 Contributions

Our main contributions in this chapter can be summarized as follows.

First, we describe a high-level abstract models of the software agents in a CPS. These models apply to a wide range of CPSs in the domain of electric grids [1, 4, 5, 6], manufacturing processes [13], autonomous vehicles [8, 9] and SDN [27]. As a result, they can be used to characterize both new and existing CPSs in various domains.

Second, we describe an abstraction of CPSs with one, possibly replicated, central controller, and one or more PAs. This abstraction models the execution trace of a CPS by using two basic relations: (1) the network relation ($\xrightarrow{n}$, read network) used to represent message exchange and (2) the computation relation ($\xrightarrow{c}$, read compute) used to represent the computation by software agents.

Third, we formally define the intentionality relation ($\rightarrow$, read intends) by using the network relation and the computation relation. We also describe an intuition for competing events and formally define this notion under the name, *intentional equivalence*. The intentionality relation accurately captures the state of the controlled physical process and is useful in proving properties of a CPS.

Lastly, we use the intentionality to formalize the correctness properties of a CPS, which are provided by the solutions in the rest of the dissertation, namely, state safety, optimal selection and consistency.

3.2 Related Work

To the best of our knowledge, we are the first to present a formal model of a CPS that captures the evolution of the state of the controlled physical process, along with an abstraction of the software agents in the CPS. Previous models of CPSs can be broadly divided into three categories: (1) domain-specific models that describe in detail, the various domain-specific operations carried out by the software agents [1, 4, 5, 8, 13], (2) models from control theory that capture the physics of the controlled process in great detail [40, 152, 153, 154], and (3) schedulability-based models that capture the deadlines of tasks in a CPS [155, 156].

The domain-specific models [1, 4, 5, 7, 8, 13] give detailed descriptions of the operations required to achieve the desired control. For example, [7] describes a very high-level exchange of messages between the agents and a very detailed discussion of the optimization problem to be solved in order to steer an electric grid into the desired state. Such models do not lend themselves well for description of the network asynchrony, or the delay- and crash-faulty nature of the software agents. Therefore, they are not suited for reasoning about the reliability of the CPSs considered in this work.

CPSs have previously been modeled and studied in the control theory under the names of networked-controlled systems [157] or real-time control systems [40]. These models concern with details of the physics of the process and focus on studying the efficacy of a control policy in maintaining stability of the process [152, 153, 154].

Lastly, the models in [155, 156] view the CPS as a series of tasks that need to be scheduled within a certain deadline. Such models use a synchronous model of computation for the software agents and are not applicable for studying delay and crash faults. Moreover, these models do not describe the various software functions implemented by the software agents.

In our proposed models, we use the events that occur in a CPS, such as sending or reception of a message, or a timeout at a software agent. Such event-based abstractions are a standard practice in classic distributed systems, such as [158], where the authors use events to qualify database systems. However, it is the formalization of the relations between these events that truly characterizes a system. In our case, these relations are the network relation and the computation relation that model the most basic operations performed by the software agents in the CPS.

We introduce the intentionality relation in order to capture the state of the controller process and track how it evolves with the implementation of setpoints. We explain the need for such a relation by demonstrating the difference between happened-before [85] and intentionality through the example in Section 3.1.2.

3.3 High-Level Model of the Software Agents in a CPS

In this section, we describe the model of the software agents in a CPS, namely controllers and PAs. These models are an abstraction of the various software functions implemented by the agents. The models were developed by studying a wide range of real-world CPSs, such as CPS for the real-time control of electric grids [1, 4, 5, 6], those used in manufacturing processes [13] and in autonomous vehicles [8, 9]. In these CPSs, the controller process is a physical process such as the voltages and power in an electric grid, the production line in the manufacturing process, and the speed and trajectory of a self-driving car. Alternatively, the CPS model also applies to SDN [27], where the controller process is “virtual”, i.e., the routing of flows in a communication network.

Although the models of the controller and PA described here only capture the internal working of the agent, they cause events in a CPS such as the sending or reception of a measurement or a setpoint. These events are related to each other through the state of the controlled process. We use the models developed here to abstract out these relations, in Sections 3.5 and 3.6.

3.3.1 Controller Model

Algorithm 3.1: Abstract model of a controller

```
1  $\mathcal{M} \leftarrow \emptyset$;          // Set of measurements received
2  $\mathbf{Z} \leftarrow [\,]$;          // Vector of measurements used in a computation
3  $\mathbf{X} \leftarrow [\,]$;          // Vector of setpoints issued
4  $T_{now}$;          // Current absolute time
5
6  on reception of a measurement $m$
7      $\mathcal{M} \leftarrow$ aggregate_received_measurements($\mathcal{M}$, m) ;
8  end;
9
10 repeat
11     decision, $\mathbf{Z} \leftarrow$ ready_to_compute($\mathcal{M}$, $T_{now}$);
12     if decision then
13         $\mathbf{X} \leftarrow$ compute($\mathbf{Z}$) ;
14         issue($\mathbf{X}$);
15     end
16 forever;
```

Algorithm 3.1 shows the abstract model of a single, non-replicated controller. It implements two routines, one to receive measurements from the PAs, and another one to compute and issue setpoints to the PAs. The two routines might interleave.

In the first routine, it implements the function `aggregate_received_measurements` in Algorithm 3.1 line 7. This function updates $\mathcal{M}$, the set of measurements received by the controller from all PAs. $\mathcal{M}$ represents the state of the physical process as observed by the controller. The function `aggregate_received_measurements` could be implemented in several ways. For instance, a controller might decide to keep only the latest message received from each PA, in which case, the $\mathcal{M}$ comprises one entry from each PA. This is observed in the the COMMELEC CPS [4]. Alternatively, a controller might choose to keep any subset of received messages.

In the second routine, the controller implements three functions: `ready_to_compute` in Algorithm 3.1 line 11, `compute` in Algorithm 3.1 line 13, and `issue` in Algorithm 3.1 line 14.

The function `ready_to_compute` decides if the controller can compute and issue setpoints by using the current state of the physical process that it observes through $\mathcal{M}$

and based on the current time T_{now}. For example, each time it receives a measurement, an SDN controller [27] decides to compute. Alternatively, the controller in [4] performs a computation either when it receives a new measurement from all PAs or when it a certain time has elapsed since the last time it issued setpoints. As a result, a controller can decide to perform a computation without having received any new measurement.

When `ready_to_compute` returns true, it also returns the vector of measurements to be used in ensuing computation ($\mathbf{Z}$). $\mathbf{Z}$ can contain any number of measurements from each PA. However, if a controller decides to compute without any measurement from a PA, then in order to guarantee correctness, it must account for this missing information. The controllers that do so, for example, [159, 160], are robust to message losses. The controllers that fail to account for missing measurements must wait for all the information before deciding to compute, in order to guarantee correctness.

The function `compute` uses the vector $\mathbf{Z}$ to compute the vector of setpoints for the PAs ($\mathbf{X}$). Once again, there are different ways in which a controller might implement the `compute` function. On the one hand, $\mathbf{X}$ can contain one setpoint for each PA as seen in [4]. In [4], the controller regularly sends a setpoint to each PA as a part of a keep-alive mechanism even if there is no change in the operating point of a PA. On the other hand, $\mathbf{X}$ can contain many setpoints for a subset of the PAs as seen in [27], where the controller sends setpoints to only the switches that require new updates after a network event.

The function `issue` sends the setpoints in $\mathbf{X}$ to the respective PAs by implementing a network call. All the functions implemented by the controller are assumed to be deterministic, i.e., they return the same output if given the same input.

From Algorithm 3.1, we notice that the controller can be stateful, as dictated by the semantic of the `aggregate_received_measurements` function that maintains $\mathcal{M}$. Furthermore, the controller is asynchronous as dictated by the semantic of the `ready_to_compute` function that might choose to return *true* any time.

Note that splitting the main routine of the controller into the two functions `ready_to_compute` and `compute` instead of just one compute function makes our model more expressive. This split enables us to reason about timeliness properties of the CPS, as discussed further in Chapter 6.

3.3.2 PA Model

Algorithm 3.2 describes an abstract model of a single PA, unaware of controller replication. As the PA is responsible for a single sub-process, as opposed to the controller that is responsible for the entire physical process, the PA is simpler with a single routine. For each received setpoint, the PA implements the setpoint using the function

Algorithm 3.2: Abstract model of a PA

```
1 on reception of a setpoint $s$
2     implement_setpoint(s);
3     $z \leftarrow$ create_measurement ;
4     issue($z$);
5 end;
```

`implement_setpoint`, through its actuator. In this way, it alters the state of the sub-process and, consequently, affects the state of the entire physical process through the actuator.

Then, the PA reads the new state of the sub-process through the `create_measurement` function that returns the measurement z. Finally, z is sent to the controller using the function `issue`.

From Algorithm 3.2, we notice that a PA is stateless. Furthermore, we notice that a PA is synchronous and responds when triggered by a setpoint.

Examples of PAs are the resource agents in the COMMELEC CPS [4] that are responsible for control of individual electrical resources such as batteries, PV panels, and heat-pumps. These PAs receive active and reactive power setpoints form the central controller and implement them through actuators such as battery converter or PV inverter. For the `create_measurement` function, these PAs read the state of their resource such as the SoC of a battery or the current power-injection of the PAs.

Alternatively, in SDN [27], the PAs are switches of the network that advertise their current link-status and routing-table entries. The setpoints in this CPS are the also routing rules to be implemented by the switches.

3.4 Events in a CPS

We abstract the execution of a CPS as a trace of events occurring on different software agents. We define three types of events that can occur on a software agent: *sending event*, *reception event*, and *timeout event*. When software agent A executes the function `issue` (Algorithm 3.1 line 14 or Algorithm 3.2 line 4), we say that A experiences a sending event. As the network might drop messages or delay messages, the message sent by A might either be received timely or the intended receiver B times out before the reception. Upon successful reception of the message by B, we say that B experiences a reception event. Alternatively, if B times out beyond a deadline, then B experiences a timeout event.

A timeout event could also be caused by firing of a timer by the internal logic of a software agent. Consider the case where agent A sends a message to B and is expecting a reply within some deadline. If B does not receive this message, then, no response is generated for A. Thus, A will eventually timeout after the deadline and is said to experience a time-out event. Hence, a time-out can occur on an agent even if no message is sent by any agent.

An event is uniquely represented by a 4-tuple (sa, pa, m, l) where (1) sa is the unique identifier of the agent on which the event occurs, (2) pa is the identifier of either the PA on which the event occurred, or the PA for which the event is intended, (3) m is the message encapsulated in the event, and (4) l is an event label given by the software agent on which the event occurs. For sending or reception events, the encapsulated message is the measurement or setpoint exchanged, and timeout events encapsulate $\perp$, representing the absence of a message.

Let $\mathcal{C}$, $\mathcal{P}$ be the sets of identifiers of all controller replicas and PAs, respectively. Then, the set of identifiers of all software agents is $\mathcal{S} = \mathcal{C} \cup \mathcal{P}$. Let $\mathcal{M}$ be the set of all messages with $\perp \in \mathcal{M}$, and $\mathcal{L}$ be a partially ordered set of labels. We denote the set of all events that occur in an execution trace by $\mathcal{E} \subset \mathcal{S} \times \mathcal{P} \times \mathcal{M} \times \mathcal{L}$. No two distinct events have the same 4-tuple (sa, pa, m, l). Furthermore, we require that the abstract labeling scheme used to obtain $\mathcal{L}$ ensures that labels of events occurring on the same software agent, for the same PA, are different. In practice, such a labeling of events is achieved through physical timestamps, a permanent sequence numbering scheme, Lamport clocks [85], Vector clocks [118], etc. Also, for CPSs that do not implement any labeling mechanism on events, the model still applies by successively numbering all events of each software agent with increasing integers.

Sending events are generated as a result of a computation by a software agent (Algorithm 3.1 line 13 or Algorithm 3.2 lines 2-3). Thus, they are termed as *output events*. Reception and timeout events are considered *input events*. This dichotomy of input-output events is similar to that of sending-receiving events used in classic distributed systems literature. We use a different name because we also have timeout events in our model.

The set of input events $\mathcal{E}^i$ (which includes reception and timeout events) and the set of output events $\mathcal{E}^o$ (which includes sending events) are such that $\mathcal{E} = \mathcal{E}^i \cup \mathcal{E}^o$ and $\mathcal{E}^i \cap \mathcal{E}^o = \emptyset$. Note that, due to network retransmissions, a single output event can result in different input events at the same PA, as each of these input events will have different labels l.

To bootstrap the CPS, we assume that a controller starts with p sending events, one for each PA. These events are called *initial sending events*. The set of all initial sending events is represented by $\mathcal{I}$.

3.5 Basic Relations between CPS Events

In this section, we define two relations, namely the *network relation* and the *computation relation*, that exist between the events in a CPS. For a relation $\xrightarrow{\text{r}}$ and an event a, we denote by $r(a)$ and $r^{-1}(a)$ the image and pre-image of a by $\xrightarrow{\text{r}}$, respectively.

3.5.1 Network Relation

Software agents exchange messages by using a communication network. Thus, a network relation ($\xrightarrow{\text{n}}$) exists between events at different agents. This relation maps an output event (sending event) at one agent to an input event (reception/timeout event) at another agent. Formally, we abstract the properties of a network relation as follows.

Definition 3.1 (Network Relation)**.** $\xrightarrow{n}$ *is a network relation iff* $\xrightarrow{n} \subset \mathcal{E}^o \times \mathcal{E}^i$ *and*

- *for any* $a \in \mathcal{E}^o$*, there exists* $b \in \mathcal{E}^i$ *s.t.*
  1. $a \xrightarrow{n} b, b.pa = a.pa$*, and* $n^{-1}(b) = \{a\}$
  2. *If* $a.sa \in \mathcal{C}$*, then* $b.sa = a.pa$
  3. *If* $a.sa \in \mathcal{P}$*, then* $b.sa \in \mathcal{C}$
- *for any* $b \in \mathcal{E}^i$*, there exists* $a \in \mathcal{E}^o$ *s.t.* $n^{-1}(b) = \{a\}$

Intuitively, $a \xrightarrow{\text{n}} b$ (read "a network b") if a is a sending event and b is its corresponding reception event or the corresponding timeout event that occurs on the intended destination. Notice that for a sending event that occurs on a controller, the corresponding input event occurs on a PA, and for a sending event that occurs on a PA, the corresponding input event occurs on a controller.

At a controller, the output of the function `issue` is a sending event that is an input to $\xrightarrow{\text{n}}$. The corresponding output of $\xrightarrow{\text{n}}$ is a reception event at a PA that occurs on Algorithm 3.2 line 1.

At a PA, the output of the function `issue` is a sending event that is an input to $\xrightarrow{\text{n}}$. The corresponding output of $\xrightarrow{\text{n}}$ is either a reception event at the controller or a timeout event at the controller at lines 6 or lines 11 in Algorithm 3.1. Although the former is evident from the pseudo-code, the latter case is when a controller times out and decides to compute without the latest measurement from this PA.

3.5.2 Computation Relation

A computation performed by a software agent can be represented as a mapping from a set of input events to a set of output events. As noted in Section 3.3.1, the `compute` function can take as input any subset of the set of received measurement $\mathcal{M}$, but if it does decide to compute without a measurement from a PA, then, in order to ensure correctness, the controller must appropriately account for that PA. This is characterized by timeout events from those PAs.

Alternatively, if a controller uses several measurements from a PA or issues several setpoints for one PA after a computation, then it can be viewed as one aggregate measurement or setpoint. As a result, we can simplify the computation of the controller as a follows. The `compute` function at a controller uses p measurements, one from each PA, and computes p setpoints, one for each PA.

From Algorithm 3.2, we know that upon reception of a setpoint, a PA implements it through the actuator by using the function `implement_setpoint` in line 2. Then it reads the state of the sub-process through a sensor by using the function `create_measurement` in line 3. The combination of two functions is viewed as a computation at a PA that takes a singleton set of input events and produces a singleton set of output events.

We abstract the properties of a computation relation ($\xrightarrow{c}$) as follows.

Definition 3.2 (Computation Relation). $\xrightarrow{c}$ *is a computation relation iff* $\xrightarrow{c} \subset \mathcal{E}^i \times \mathcal{E}^o$ *and*

1. *for any* $a \in \mathcal{E}^i$
   (a) *If* $a.sa \in \mathcal{P}$*, then* $\exists b \in \mathcal{E}^o : a.sa = b.sa$,
   $a.pa = b.pa, c(a) = \{b\}$*, and* $c^{-1}(b) = \{a\}$
   (b) *If* $a.sa \in \mathcal{C}$*, then*
      - $\forall\, i \in \mathcal{P},\ \exists!\, b \in c(a) : b.pa = i$
      - $\forall\, b \in c(a), b.sa = a.sa$
      - $\forall\, b, b' \in c(a), c^{-1}(b) = c^{-1}(b')$
      - $\forall\, b \in c(a),\ \forall\, i \in \mathcal{P},\ \exists! a' \in c^{-1}(b) : a'.pa = i$
2. *for any* $a \in \mathcal{E}^o \setminus \mathcal{I}$*, there exists* $b \in \mathcal{E}^i$ *s.t.* $b.pa = a.pa$ *and* $b \in c^{-1}(a)$.

In rule (1), we see that at the PA, the computation relation takes one input event and produces one output event, as evident from the functions `implement_setpoint` and `create_measurement`. However, at the controller, for every input event that is an input to the computation relation, there exist $p - 1$ other input events, all from different PAs that are also an input. Furthermore, the result is also p output events. This is because the function `compute` takes vector **Z** as input that has a size p and returns vector X that also has a size p.

3.6 Intentional Equivalence and Intentionality

In this section, we formalize the notion of intentionality, a relation between events in a CPS that captures the order between measurements and setpoints. To this end, we first define an equivalence relation between events, called *intentional equivalence*.

3.6.1 Intentional Equivalence Relation

In the presence of controller replication and message retransmission, certain events that occur in the same "control round" in a CPS are functionally the same, *i.e.,* they steer the physical process to a similar state. The *intentional equivalence* relation ($\equiv$) captures this.

We list the properties that are used to define this relation. The intentional equivalence relation is defined as the smallest relation satisfying the following properties.

1. If a and b are the initial sending events at controller replicas, and $a.pa = b.pa$ then $a \equiv b$
2. If $a \xrightarrow{\text{n}} b$ and $a \xrightarrow{\text{n}} c$, then $b \equiv c$
3. If $a \equiv \tilde{a}$, $a \xrightarrow{\text{n}} b$ and $\tilde{a} \xrightarrow{\text{n}} c \implies b \equiv c$
4. If $a \xrightarrow{\text{c}} b$, $a \xrightarrow{\text{c}} c$ and $b.pa = c.pa$, then $b \equiv c$
5. If $a \equiv \tilde{a}$, $a \xrightarrow{\text{c}} b$, $\tilde{a} \xrightarrow{\text{c}} c$ and $b.pa = c.pa \implies b \equiv c$

The intuition behind rule (1) is that initial sending events are computed without any knowledge of the state of the system. They are polling events and do not steer the system in any direction. Hence, those sent to the same PA are equivalent.

For rule (2), the underlying intuition is that reception events and their corresponding timeout events convey similar information to the controller, as do multiple reception events corresponding to the same sending events (i.e., retransmission). A reception event informs the controller of the state of the process, whereas the corresponding timeout event forces the controller to estimate the missing state before computation. Recall from Section 3.3.1 that, when a controller of a CPS computes using timeout events, it accounts for the missing information, in order to ensure correctness. Thus, reception events and corresponding timeout events are equivalent.

From Definition 3.2, there exists a single output event resulting from a computation relation for each PA. Rule (4) states that if an event causes, after computation, two events for the same PA, then the resulting events are equivalent. In fact, the resulting events are the same event, and are therefore equivalent by the reflexive property of the intentional equivalence relation (Theorem 3.6.1). Finally, rules (3) and (5) mean that equivalent events, when subject to the same relation, result in equivalent events.

To better understand the intuition behind equivalent events that result in similar changes to the state of a CPS, consider a controller that receives partial information from its PAs. A well-designed controller would compute only if it can reconstruct the missing information from the partially received information, thus resulting in safe setpoints. For example, a controller with two PAs, one with a 1 Hz update rate and another with a 10 Hz update rate, would require a measurement from the latter every 100 ms, whereas from the former only once every 1 s as it knows that the state of the slow PA has not changed during that second. Thus, a timeout event on the measurement from that PA is equivalent to a reception event.

Events resulting from retransmissions encapsulate the same message verbatim, and are hence deemed equivalent.

The following theorem states that the intentional equivalence is, indeed, an equivalence relation.

Theorem 3.6.1. *"Intentional equivalence" is reflexive, symmetric and transitive.*

The proofs of Theorems 3.6.1, 3.6.2, and 3.6.3 are not central to our discussion and proving them in the general case requires a tedious enumeration of several cases, in a manner similar to the one followed in the proofs of the main results of our solution to provide intentionality in a CPS (Theorems 4.5.1 and 4.5.2). However, from the result of Theorem 4.5.1, these theorems can be easily shown to hold for CPSs that implement the Intentionality clocks described in Chapter 4. These proofs are presented in the Appendix A.

3.6.2 Intentionality Relation

We define the intentionality relation ($\rightarrow$), where $a \rightarrow b$ is read as "a intends b", by using the relations defined in the previous sections. It is the smallest relation satisfying the following properties:

1. If $a \overset{\text{n}}{\rightarrow} b$, then $a \rightarrow b$
2. If $a \overset{\text{n}}{\rightarrow} b$ and $\tilde{a} \equiv a$, then $\tilde{a} \rightarrow b$
3. If $a \overset{\text{c}}{\rightarrow} b$, then $a \rightarrow b$
4. If $a \overset{\text{c}}{\rightarrow} b$ and $\tilde{a} \equiv a$, then $\tilde{a} \rightarrow b$

5. If $a \to b$ and $b \to c$, then $a \to c$

Based on these properties, the following theorems hold.

Theorem 3.6.2. *For any two events* a, b*:* $a \to b \implies b \not\to a$.

Theorem 3.6.3. *For any two events* a, b*, such that* $a.pa = b.pa$*:* $(a \not\to b \land b \not\to a) \iff a \equiv b$.

Let $\mathcal{E}_p = \{e \in \mathcal{E} | e.pa = p\}$. Recall that $\mathcal{E}_p/{\equiv}$ represents the factorization of the set $\mathcal{E}_p$ by the relation $\equiv$. Then, as a consequence of Theorem 3.6.3, the intentionality relation induces a total order on $\mathcal{E}_p/\equiv$, for any $p \in \mathcal{P}$. In other words, the intentionality relation induces a total order on any set of events concerning the same PA and belonging to different equivalence classes.

3.7 Correctness Properties

As noted in Section 3.1.1, the state of the physical process changes after every setpoint implementation. Thus, when a controller uses two input events $e : (sa_1, pa_1, m_1, l_1)$ and $e' : (sa_1, pa_1, m_2, l_2)$ such that $e \to e'$, the message m_2 reflects a more recent state of the sub-process controlled by PA$_1$ than the message m_1. If the controller uses the "old" event e in the computation of setpoints, then the resulting setpoints might not be compatible with the most recent state of the process, thereby causing incorrect control of the CPS. Hence, a controller must only use those measurements from a PA in computation that reflect the current state of the CPS, i.e., a state that was the result of the last setpoint implementation by a PA. Similarly, a PA must only implement those setpoints that were computed by accounting for its most recent state. This notion is formally captured by the correctness property of *state safety.*

Definition 3.3 (State Safety)**.** *A CPS satisfies the* state safety *property iff, whenever a software agent uses an event* a *as input for computation, then the last sending event* b *that occurred on this software agent, where* $b.pa = a.pa$*, is such that* $b \to a$.

Notice that discarding all messages can trivially satisfy the state-safety property. However, this will render the physical process uncontrolled by a complete loss of control. Therefore, the selection of events must be optimal, i.e., only those events that violate the safety property must be discarded. This property is formally defined as follows.

Definition 3.4 (Optimal Selection)**.** *A CPS satisfies the* optimal selection *property iff an event* a *on a software agent is discarded only if there exists another event* b*, with* $b \not\to a$ *that: (1) occurred on this software agent or (2) this software agent was informed that* b *occurred on its replica.*

The first part of the optimal selection property, on its own, states that an event a must be accepted by a software agent if the last sending event b on this software agent intended to cause a. Recall that if $b \rightarrow a$, then all events that occurred on this software agent before b also intend to cause a. Thus, a presents new information, i.e., information about the state of the process after the implementation of the last-sent setpoint. In the presence of replication, however, an event that was intended by the last sending event, does not necessarily present new information, as seen in the following example.

A controller replica that last computed setpoints in round l might receive two measurements from the same PA: one from round $l+1$ and another from round $l+2$, before it is ready to compute. This can occur due to another controller replica driving the CPS into round $l+2$. In this scenario, the first controller might ignore the message from round $l+1$ as the message from round $l+2$ supersedes it. The controller will, therefore, compute using reception events from round $l+2$ and declare timeout events for PAs from which it has not received measurements from that round. This condition is captured by the second part of the optimal selection property.

The intentionality relation relies on the intentional equivalence relation. In the presence of replication, however, different replicas might send different sets of setpoints in a given "control round". This can happen as a result of the replicas using different combinations of reception and timeout events in their computation. Although these input sets are deemed equivalent in our model, the encapsulated setpoints might not be the same. In practice, to ensure that the messages encapsulated in equivalent output events are the same, the controller replicas in a CPS must satisfy consistency as defined below.

Definition 3.5 (Consistency). *A CPS satisfies the* consistency *property iff, for two events* $a, \tilde{a} \in \mathcal{E}^o$*, if* $a \equiv \tilde{a}$*, then* $a.m = \tilde{a}.m$*.*

Consistency can be provided by controller replicas performing an agreement, either matching their output, as done in consensus protocols [86, 161], or their input, as done in Quarts [162], described in Chapter 5.

The three properties mentioned here, namely state safety, optimal selection and consistency, are not the only correctness properties desired in a CPS. For instance, as CPSs are often real-time systems, they also need to satisfy some timeliness properties which are addressed in Chapter 6. There are also other correctness properties that are important, but beyond the scope of this dissertation. One such key example is safety in the presence of malicious controller replicas, i.e., PAs never implement a malicious setpoint.

Note that the three properties considered here are all safety properties and none are liveness properties, as described in the distributed systems literature [83]. Thus, the FLP impossibility result [88] does not prevent simultaneously guaranteeing all three of them. Moreover, all these properties can be satisfied by making the controllers perform consensus. However, as consensus cannot be performed in bounded time in an asynchronous system [88], these methods are not applicable to real-time CPSs. Hence, the challenge is to satisfy these correctness properties with low or bounded-latency overhead. We present mechanisms that guarantee state-safety and optimal selection with zero latency-overhead in Chapter 4, and a bounded latency-overhead mechanism for providing consistency in Chapter 5. The newly proposed mechanisms are presented as modifications to the design of the software agents (Algorithms 3.1, 3.2), so that they can be implemented with minor modifications to existing software agents.

If a software agent uses an event a for computation, then the last sending event b that occurred on this software agent, where $b.pa = a.pa$, is such that $b \rightarrow a$.

An event a must only be discarded if there exists another event b, such that $b \not\rightarrow a$ that: (1) occurred on this software agent, or (2) this software agent was informed that b occurred on its replica.

For two events $a, \tilde{a} \in \mathcal{E}^o$, if $a \equiv \tilde{a}$, then $a.m = \tilde{a}.m$.

3.8 Conclusion

We presented a formal model of a CPS, based on the events that occur in the CPS and the relations among these events. We formalized the basic relations namely, network relation and computation relation. We used these relations to define the intentionality relation. Similar to the happened-before relation, the intentionality relation imposes an ordering between events. However, if $a \rightarrow b$, then it does not necessarily mean that a happened-before b. Instead, b could have resulted from an event $\tilde{a}$ that happened before a and b. The newly defined intentionality relation captures the evolution of the state of the controlled physical process in a CPS. It is hence useful in defining and proving the properties required for correct operation of a CPS. We defined three such properties namely, state safety, optimal selection and consistency.

4 Ordering Events Based on Intentionality

> *Ever since the dawn of civilization,*
> *people have not been content to see*
> *events as unconnected and inexplicable.*
> *They have craved an understanding*
> *of the underlying order in the world.*
> *— Stephen Hawking, A Brief History Of Time*

In Chapter 3, we described the intentionality relation and the associated correctness properties that are desired in a CPS. In order to satisfy these properties, software agents in the CPS need to order, according to the intentionality relation, the events that occur in the CPS. In this chapter, we describe *intentionality clocks*, a labeling mechanism based on logical clocks, and adapted from Lamport clocks [85]. These clocks accurately describe the intentionality relation and can be used by software agents to achieve the required ordering. For this, we first define the clock-consistency condition and the strong clock-consistency condition that will be used to prove the one-to-one correspondence between intentionality clocks and the intentionality relation. Using examples, we show the inability of physical time, and other existing solutions, in achieving the strong clock-consistency condition.

We present the design of the CPS software agents (controllers and PAs) that uses the intentionality clocks to achieve two of the desired correctness properties, namely state safety (Definition 3.3) and optimal selection (Definition 3.4) defined in Chapter 3. The state safety property ensures that the computation of setpoints by a controller occurs using the most recent state of the CPS; and the setpoints implemented by PAs are those computed using the most recent state of the CPS. The optimal-selection policy ensures that the software agents of the CPS do not erroneously discard measurements or setpoints containing useful information.

Several industrial solutions circumvent the ordering problem by using frameworks, such as TTA [131] and TCB [163]; they provide synchrony guarantees. However, 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 lends to an ease of deployment in existing CPSs.

Lastly, we analyze a CPS for the optimal charging of EVs [1] that uses a form of labeling mechanism that violates the optimal-selection property. We show how this can cause a deadlock in the CPS and propose modifications to the design.

4.1 Clock-Consistency Condition

In order for a labeling mechanism to accurately capture a relation, it must both *describe* and *infer* the relation. For two events a and b, we say that a clock mechanism C describes a relation $\xrightarrow{x}$ if the event labels are such that $C(a) < C(b) \implies a \xrightarrow{x} b$, where $C(a)$ and $C(b)$ are the labels of events a and b, respectively. Moreover, we say that the clock mechanism infers a relation $\xrightarrow{x}$ if $a \xrightarrow{x} b \implies C(a) < C(b)$. If the labeling mechanism both describes and infers the relation $\xrightarrow{x}$, it is said to satisfy strong clock-consistency under the relation $\xrightarrow{x}$. Alternatively, if a labeling mechanism only infers the relation, then it is said to satisfy the clock-consistency condition under relation $\xrightarrow{x}$ [85].

For example, under the “happened-before” relation that provides causal order, Lamport clocks [85] satisfy the clock-consistency condition, as they can only infer the relation, i.e., $a \xrightarrow{h} b \implies LC(a) < LC(b)$, where $\xrightarrow{h}$ is the happened-before relation and $LC(a)$ is the label of event a obtained using Lamport clocks. As Lamport clocks cannot describe the happened-before relation, Vector clocks [118] were introduced. Vector clocks both describe and infer the happened-before relation, thus satisfy the strong-clock consistency condition.

Under the happened-before relation, if two events have the same label, they are not comparable or concurrent. However, under the intentionality relation, if two events have the same label, they are intentionally equivalent. Thus, the definition of strong clock consistency needs to be modified to account for equivalent events as follows. For two events a and b, we say that a clock mechanism C describes the intentionality relation if the event labels are such that $C(a) < C(b) \implies a \rightarrow b$ and $C(a) = C(b) \wedge a.pa = b.pa \implies a \equiv b$. We say that the clock mechanism infers the intentionality relation if $a \rightarrow b \implies C(a) < C(b)$ and $a \equiv b \implies C(a) = C(b)$. The strong clock-consistency condition is satisfied if a clock mechanism both describes and infers the intentionality relation.

Traditional distributed systems provide causal order according to the happened-before relation. This is achieved through one of several mechanisms such as timestamps, Lamport clocks [85], or vector clocks [118]. In Section 4.2, we describe the inability of timestamps and vector clocks to satisfy the strong clock-consistency condition under the intentionality relation.

4.2 Clock Consistency Using Existing Solutions

4.2.1 Using Physical Time

Here, we answer the question, "Is physical time sufficient to guarantee the strong clock-consistency condition under the intentionality relation?"

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 clock-synchronization or network-based clock-synchronization (e.g., PTP [164], NTP [165]). These time-synchronization solutions provide a synchronization accuracy δ_s that ranges from sub-microsecond to one millisecond.

CPSs often use the availability of synchronized physical clocks to reason about the temporal ordering of events. However, as we see in the following examples, the temporal ordering of events does not coincide with intentionality. We use the notation $TS(a)$ to denote the timestamp of event a obtained using physical time at a software agent. If two events a and b occur on the same software agent such that b is temporally after a, then $TS(a) < TS(b)$.

Consider a CPS with two controller replicas and a single PA, as shown in Figure 4.1. We will consider a perfect time-synchronization ($\delta_s = 0$). Since $a \to b$ and $a \equiv \tilde{a}$, we have $\tilde{a} \to b$, by rule 4 of the intentionality relation (Section 3.6.2). However, $TS(a) < TS(b) < TS(\tilde{a})$. Thus, we have $\tilde{a} \to b$ and $TS(b) < TS(\tilde{a})$. Therefore, we conclude that, on their own, physical clocks cannot infer intentionality.

Another example emphasizing the difference between temporal order and intentionality, shown in Figure 4.2, concerns a CPS with one non-replicated controller and two non-replicated PAs. Due to network delay, the reception event b_2 occurs on PA$_2$ much later than the reception event b_1 at PA$_1$. As a result, the reception event d_1 from PA$_1$ occurs much earlier than the corresponding reception event $\tilde{d}_2$ from PA_2. Instead, the controller moves on with a timeout event d_2 for PA_2. Consequently, the events e_1, f_1, g_1 and h_1 take place. Then, we have $a_2 \to d_2 \to e_1 \to h_1$. But, $d_2 \equiv \tilde{d}_2 \implies \tilde{d}_2 \to h_1$. However, on the controller, the time of occurrence of h_1 occurred is less than that of $\tilde{d}_2$, i.e., $TS(h_1) < TS(\tilde{d}_2)$. Once again, the temporal order does not coincide with intentionality.

Hence, we conclude that on their own, physical clocks are not sufficient to provide

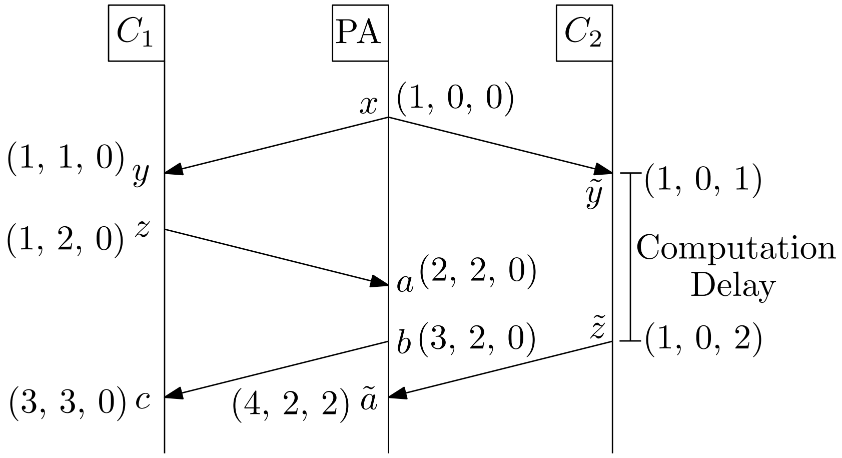


Figure 4.1 – Difference between temporal order and intentionality due to replication of the controller. The notation (1, 2, 3) represents the vector clock of an event in the format (PA:1, C_1:2, C_2:3)

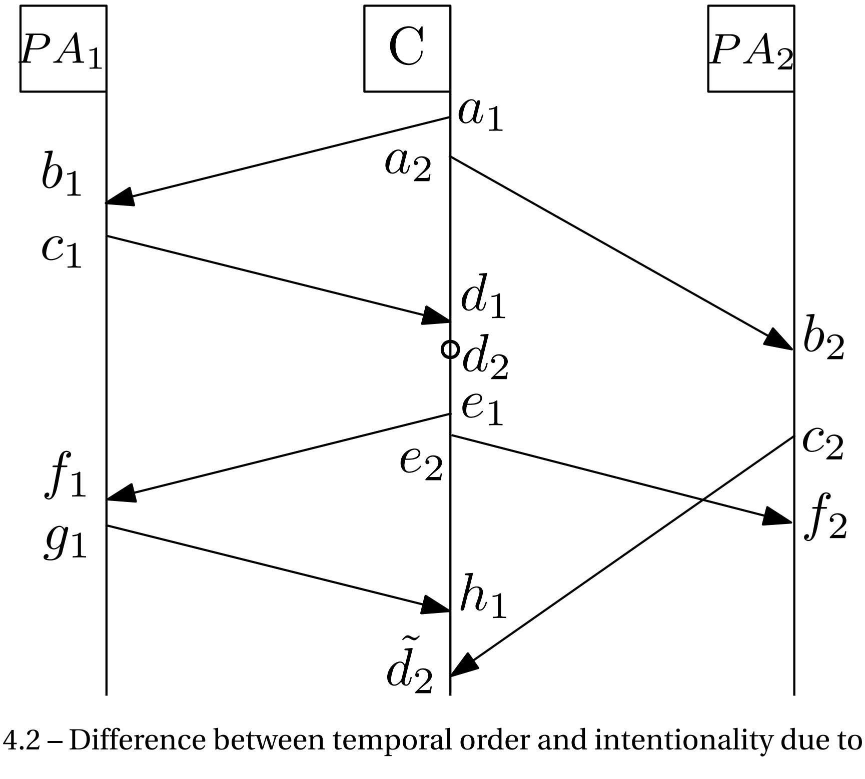


Figure 4.2 – Difference between temporal order and intentionality due to delays

strong-clock consistency under the intentionality relation and require an additional mechanism to do so. In fact, the problem is not that it fails to either infer or describe the relation, but that temporal order does not coincide with intentionality. As a result, if physical clocks are used by the software agents to order events, the CPS might violate the safety properties such as state safety and optimal selection. In Section 4.3, we present a mechanism that describes the intentionality relation by using logical clocks instead of physical clocks. This mechanism has an added benefit that it does not require synchronized physical time.

4.2.2 Using Logical Clocks

To demonstrate the inability of Lamport clocks or vector clocks in providing strong clock-consistency condition, we will use the example in Figure 4.1 and attribute the message labels using these clock mechanisms, instead of physical time.

In order to obtain labels that use Lamport clocks, we use the algorithm presented in [85]; it is given by three simple rules:

1. A software agent increments its counter before each event in that software agent
2. When a software agent sends a message, it includes its counter value with the message
3. Upon receiving a message, the counter of the recipient is updated, if necessary, to the greater of its current counter and the timestamp in the received message. The counter is then incremented by 1, before the message is considered received.

As a result, we obtain the following allocation of labels: $x.l = 1$, $y.l = \tilde{y}.l = 2$, $z.l = \tilde{z}.l = 3$, $a.l = 4$, $b.l = 5$ and $c.l = 6$. Moreover, $\tilde{a}.l = \max(\tilde{z}, b) + 1 = 6$. Thus, we have $LC(b) < LC(\tilde{a})$. However, $\tilde{a} \equiv a$ and $a \to b$. Thus, by rule 4 of the intentionality relation, $\tilde{a} \to b$. Therefore, strong-clock consistency condition is violated.

Vector clocks, in contrast to scalar Lamport clocks, use a vector of logical clocks, with one clock per software agent. The vector clocks are maintained according to the following rules:

1. Each time a process experiences an internal event, it increments, by 1, its own logical clock in the vector.
2. Each time a process sends a message, it increments, by 1, its own logical clock in the vector and then sends a copy of its own vector.
3. Each time a process receives a message, it increments, by 1, its own logical clock in the vector by one and updates each element in its vector by taking the

maximum of the value in its own vector clock and the value in the vector in the received message (for every element).

Applying these rules to the example in Figure 4.1, we obtain the vector clocks of the events as shown in Figure 4.1. The comparison on vector clocks is defined as $VC(x) < VC(y) \iff \forall z\,[VC(x)_z \leq VC(y)_z] \wedge \exists z'\,[VC(x)_{z'} < VC(y)_{z'}]$, where $VC(x)_z$ is the value of the logical clock of event x in position z. For example, in a distributed system with three agents A, B, C with vector clock $v_1 = \langle A:2, B:3, C:4\rangle$, $v_2 = \langle A:3, B:3, C:4\rangle$, $v_3 = \langle A:3, B:0, C:4\rangle$, we have:

- $v_1(A) < v_2(A)$, $v_1(B) = v_2(B)$ and $v_1(C) = v_2(C)$. Therefore, $v_1 < v_2$.
- $v_1(A) < v_3(A)$, $v_1(B) \not< v_3(B)$ and $v_1(C) = v_3(C)$. Therefore, $v_1 \not< v_3$ and $v_1 \not> v_3$. v_1 and v_3 are incomparable.

Thus, for the events in Figure 4.1, we obtain $VC(a) < VC(b) < VC(\tilde{a})$. But, we have from intentionality that $\tilde{a} \rightarrow b$. Hence, the strong clock-consistency condition is violated.

The central problem is that Lamport clocks and Vector clocks were designed to infer or describe the happened-before relation that is different from the intentionality relation because of intentional equivalence. Moreover, vector clocks for two events a and b on two different software agents are sometimes not comparable because one entry in a might be larger than the corresponding entry in b, and another entry might be smaller. Such events are labeled concurrent events under the happened-before relation. However, such events under the intentionality relation might be equivalent or might be causally related. Therefore, to avoid the issue of incomparable vector clocks, we use scalar clocks in our design of intentionality clocks.

Other clock mechanisms derived from vector clocks such as plausible clocks [166], dynamic vector clocks [167] and interval tree clocks [168] address the scalability issues in vector clocks. When it comes to intentionality, they suffer from the same limitations as vector clocks.

4.3 Intentionality Clocks

Intentionality clocks is a mechanism for maintaining, updating and synchronizing logical clocks across all software agents in a CPS with one or more replicas of a central controller and, with one or more PAs. It is adapted from the Lamport clocks abstraction that was designed for general distributed systems, to accommodate the specificities of events in CPSs.

Similar to Lamport clocks, each agent maintains and updates a local logical clock (C). The agent also maintains a set of the labels of all the events it has recorded from other agents so far $\mathcal{L}$. The labels of events are assigned using both C and $\mathcal{L}$, such that they guarantee the strong clock-consistency condition. These labels are communicated, by the software agent, along with the message encapsulated in the sending event. The labels of the reception and timeout events corresponding to that sending event are updated according to the following rules:

1. The event label of a sending event at an agent is the value of its logical clock, right before the sending event.
2. If a is a sending event and b a reception or a timeout event, such that $a \xrightarrow{\text{n}} b$, then $b.l = a.l + 1$.
3. The logical clock of an agent is never decremented.
4. The logical clock of an agent is only incremented by 1 after a computation and as follows before a computation.
   (a) If $C < \max(\mathcal{L})$, then $C \leftarrow \max(\mathcal{L})$
   (b) If $C >= \max(\mathcal{L})$, then $C \leftarrow C + 3$

Rule 1 of intentionality clocks states that a software agent attaches the value of its current clock C to the sending event, so that other agents understand the state of the controller process as seen by this software agent at the logical time instant C. Rule 2 follows from the definition of the network relation. Rule 3 is a result of the fact that the state of the controlled process evolves only in one direction. In other words, time does not go backwards.

Rule 4 dictates that, in order for the clock to represent a *"control round"*, the clock must be incremented only when a computation happens. Before a computation using timeout events, an agent must account for all the missing information. In order to properly label the timeout events as belonging to the new control round, the agent must increment its clock to the highest label it has seen so far. This is a proxy to the most recent state of the process recording by this software agent. This includes two cases: (1) either it has recorded the most recent information from other agents, i.e., $C < \max(\mathcal{L})$ or (2) this agent has the most recent state of the controlled process. In case (1), the clock takes the value of the received labels.

Case (2) is an example of a software agent that decides to compute as a result of a timeout, for example, a COMMELEC controller [4] performing a computation because of a timeout since the last time it performed computation. Figure 4.3 shows the example of one such computation. In this case, the controller must account for

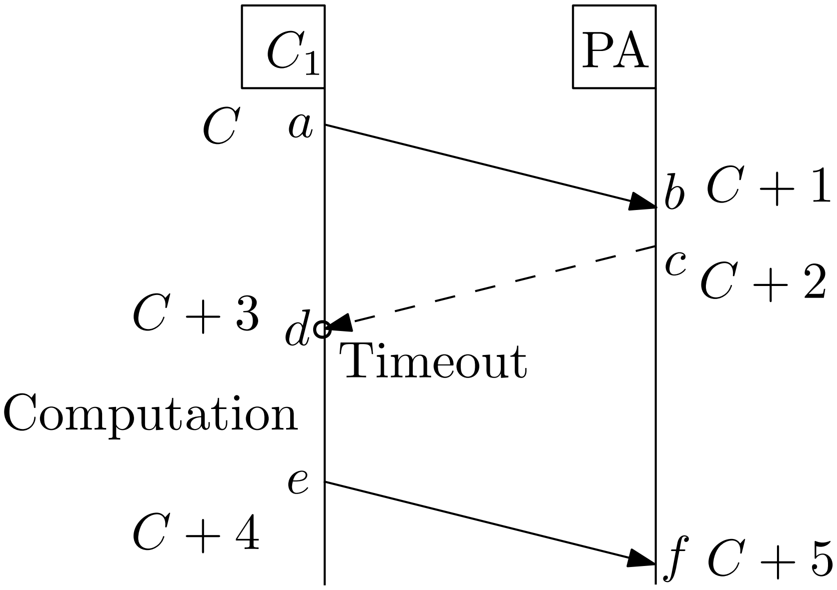


Figure 4.3 – Example of computation at a controller as a result of a timeout indicating the clock at the agents after each event

any possible setpoint implementations that could have occurred since its last setpoint issued (a). To this end, it uses the timeout a timeout event (d) for the measurement sending event (c) that would have resulted form the implementation of its last setpoint. From Figure 4.3, we see that $a \overset{\text{n}}{\to} b \overset{\text{c}}{\to} c \overset{\text{n}}{\to} d$. If the clock at event a is C, then it will be $C+1$, $C+2$ and $C+3$ at events b, c, d, respectively. The clock is incremented by 1 after the computation, to result in the setpoint sending event e.

Compared to Lamport clocks [85], there are two main distinctions in the design of intentionality clocks. First, in our solution, the logical clock at an agent is incremented only when the agent performs a computation. This enables the controller to infer the intentionality relation by using its local logical clock, and to have a notion of the control round. In contrast, the Lamport clock at an agent is updated after every event that occurs at that software agent. Thus, the agents lose the information required to infer the intentionality relation and to have a notion of the control round. For example, in Figure 4.4, a CPS with two PAs that uses Lamport clocks increments its clock each time it receives a message from a PA. As a result, it will infer the relation between events b_1 and b_2 as $b_1 \to b_2$. A similar problem arises in a CPS with replicated controllers, where a PA erroneously increments its clock twice every time it receives setpoints from two different controller replicas.

Second, in intentionality clocks, the label of a reception event is one more than the label of the corresponding sending event. In contrast, in Lamport clocks, for a reception event b that occurs when the value of the logical clock of the agent is C is $b.l = \max(a.l, C) + 1$, where a is the corresponding sending event. Due to the presence of delayed controller replicas or message retransmissions by the network, reception

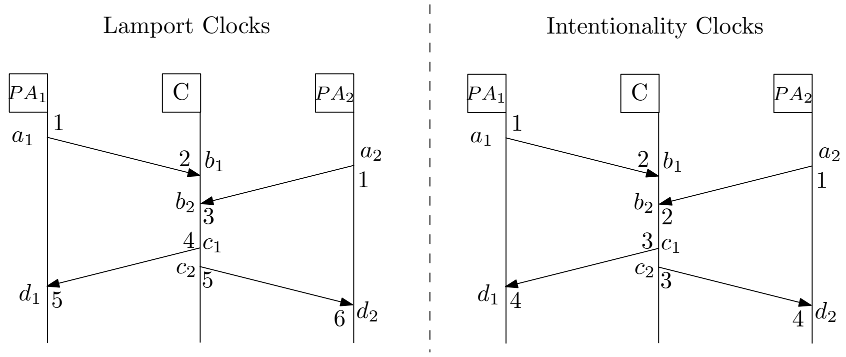


Figure 4.4 – Difference between Lamport clocks and intentionality clocks – An example of erroneous incrementation of Lamport clocks with two PAs

events from previous control rounds might have a label higher than reception events from current control round. Consequently, it cannot guarantee the strong clock-consistency condition.

Next, we present the design of software agents that use intentionality clocks. We will prove in Section 4.5 that such a design satisfies strong-clock consistency and guarantees state safety and optimal selection.

4.4 CPS Design with Intentionality Clocks

4.4.1 Controller Design

Algorithms 4.1 and 4.2 describe the design of a controller with intentionality clocks; this design satisfies the state safety and optimal-selection properties. The model of the controller is the same as the one given by Algorithm 3.1. The parts in red are our modifications for satisfying the aforementioned properties (together with the implementation of Algorithm 4.3).

Each controller maintains a logical clock (C), in addition to the set of received measurements ($\mathcal{M}$) and a set of labels of received measurements $\mathcal{L}$. Upon receiving a measurement from a PA, the controller declares a reception event with a label, one more than the label of the received message (Algorithm 4.1, line 9). The controller adds the measurement to $\mathcal{M}$ and adds the label of the reception event to $\mathcal{L}$.

The controller also occasionally checks if it has accumulated, through the measurements, enough information about the state of the physical process required to compute setpoints. To this end, it uses the `ready_to_compute` function. As noted in Section 3.3.1, the controller can choose to start a computation of setpoints by considering any form of information provided by measurements from $\mathcal{M}$. Moreover, we can augment the `ready_to_compute` function with the labels in $\mathcal{L}$, as they expose additional information, by giving insight into the intentionality relation between the events.

Algorithm 4.1: Abstract model of a controller with intentionality clocks

```
$\mathcal{M} \leftarrow \emptyset$ ;                      // Set of measurements received
$\mathbf{Z} \leftarrow [\,]$ ;                      // Vector of measurements used in a computation
$\mathbf{X} \leftarrow [\,]$ ;                      // Vector of setpoints issued
$T_{now}$ ;                         // Current absolute time
$C \leftarrow 0$;                       // Intentionality clock on this controller
$\mathcal{L} \leftarrow \{\}$;                       // Set of labels of received measurements

on reception of a message $m$ with label $l$ from a PA $i$
    Declare reception event $a : (my\_id, i, m, l + 1)$;
    $\mathcal{M} \leftarrow$ aggregate_received_measurements($\mathcal{M}$, m) ;
    $\mathcal{L} \leftarrow a.l \cup \mathcal{L}$;
end;

repeat
    decision, $\mathbf{Z} \leftarrow$ ready_to_compute($\mathcal{M}$, $T_{now}$);
    if decision then
        $\mathbf{Z}, C, \mathcal{L} \leftarrow$ choose_measurements($\mathbf{Z}$, $C$, $\mathcal{L}$);
        $\mathbf{X} \leftarrow$ compute($\mathbf{Z}$) ;        // Using measurements with label $C - 1$
        issue($\mathbf{X}, C$);             // Sending events
    end
forever;
```

When the controller decides to compute setpoints, it must first update its clock and choose the correct measurements to compute with, using Algorithm 4.2, in order to satisfy the state safety and optimal-discard correctness properties. This function implements rule 4 of the intentionality clocks. First, the controller computes the highest label of the events it has seen: C'. This represents the most recent events the controller has encountered. Then, for all PAs from which the highest label of received events is less than C', the controller declares a timeout event (line 4), thereby explicitly acknowledging that it lacks the most recent information from this PA. The controller can then account for this missing information in the subsequent computations. The logical clock C is set as $C' + 1$ to mark the computation operation and the resulting setpoints are issued by way of sending events with the label being the current logical clock.

Note that the model of the `compute` function in Algorithm 4.1 permits a controller to send multiple setpoints for the same PA. However, the computation relation (Definition 3.2) permits only one setpoint sending event per PA. This difference between an event and the setpoint messages is reconciled by encapsulating all the setpoints for one PA, resulting form a computation, into the one event. A similar mechanism can be applied to reconcile the difference that the `compute` function permits using multiple measurements from the same PA, whereas the computation relation requires only one reception or timeout event from each PA.

Algorithm 4.2: Function: `choose_measurements`($\mathbf{Z}$, C, $\mathcal{L}$)

```
1  $C' \leftarrow \max(C+3, \max(\mathcal{L}))$;
2  for each PA $i$ do
3      if the maximum label in $\mathcal{L}$ from PA $i$ is not equal to $C'$ then
4          Declare timeout event $a : (my\_id, i, \perp, C')$;
5          $\mathbf{Z}[i] \leftarrow \perp$;
6          $\mathcal{L} \leftarrow a.l \cup \mathcal{L}$;
7      end
8  end
9  $C \leftarrow C' + 1$;
10 Return $\mathbf{Z}, C, \mathcal{L}$;
```

The controller is designed to be soft state [169]. When a controller boots or reboots after a crash, its logical clock is set to zero, and the lists $\mathcal{M}$ and $\mathcal{L}$ are reinitialized. Thus, it would use all subsequent reception or timeout events for computation, because their labels would be ≥ 0. This behavior is in accordance with the state safety property, as a freshly rebooted controller de-facto has no last setpoint sending events. However, as described in Section 4.4.2, these sending events would be disregarded by the PAs as they do not reflect their most recent state. Note that, upon booting or rebooting, the controller sends setpoints corresponding to the initial sending events, indicated by S_0, with the label 0.

From lines 1 and 9 in Algorithm 4.2, we see that the logical clock of a newly booted controller is re-synchronized with that of the other software agents before computation, by taking the maximum of the labels of the received measurements. This is discussed further, in Section 4.4.2.

4.4.2 PA Design

Algorithm 4.3 describes the design of a PA with intentionality clocks; the design complements Algorithms 4.1 and 4.2 in satisfying the state safety and optimal-selection properties. Each PA maintains a clock C that is initialized with 0 upon booting.

Upon the reception of a setpoint from a controller, the PA declares a reception event with a label one more than that received in the message. Then, the PA compares the label of the event with its local logical clock. If the reception event has a higher label, then the PA implements the setpoint, else it is discarded because it violates the state safety property. In other words, a reception event with a label less than the logical clock of the PA means that the corresponding sending event was not computed with the most recent state of this PA. In this way, setpoints from delay-faulty controllers or freshly booted controllers are not implemented, thereby upholding the state safety property.

Algorithm 4.3: Abstract model of a PA with intentionality clocks

```
1  on boot
2      $C \leftarrow 0$;
3      $z \leftarrow \perp$;
4  end;
5  on reboot
6      $C \leftarrow$ stored $C$;
7      $z \leftarrow \perp$;
8  end;
9  on reception of a setpoint $s$ with a label $l$ from a controller
10     Declare reception event $a : (my\_id, my\_id, m, l+1)$;
11     if $C < a.l$ ;
12      then
13         $C \leftarrow a.l$;
14         implement_setpoint($s$);
15         $C \leftarrow C + 1$;
16         Store $C$;
17         $z \leftarrow$ create_measurement();
18     end
19     issue($z$, $C$);
20 end;
```

After implementing the setpoint through an actuator, the PA increments its logical clock to mark the computation of a new measurement. The PA computes a new measurement through a sensor and sends the measurement to the controller by a sending event labeled with the current logical clock.

Each PA stores its logical clock before computing measurements. When a PA recovers after a crash, it initializes its logical clock to the last stored value[1]. In this way, the PA keeps track of the last state it advertised to the controller.

Controllers and PAs update their local logical clocks to reflect the labels they observe (Algorithm 4.2 line 1, Algorithm 4.3 line 10). Notice that, upon receiving a setpoint with a label lower than its logical clock, a PA also sends both the latest computed measurement and the current value of the local clock. This serves to re-synchronize the software agents that miss some control rounds due to message losses, crashes and recoveries, or delays.

4.5 Formal Guarantees

We formally prove that our mechanism of intentionality clocks and Algorithms 4.1, 4.2, and 4.3 guarantee state safety and optimal selection. The first step lies in proving that

---

[1] Constantly increasing counters might cause a counter overflow. However, a 64-bit counter incremented once every millisecond takes much longer than the lifetime of any CPS to wrap around.

the intentionality clocks under our mechanism infer and describe the intentionality relation.

Theorem 4.5.1 (Strong Clock-Consistency)**.** *In a CPS that implements Algorithms 4.1, 4.2, 4.3: for any two events* a *and* b*,*

$$C(a) < C(b) \iff a \to b$$

$$C(a) = C(b) \text{ and } a.pa = b.pa \iff a \equiv b$$

Proof. The proof follows from Lemmas 4.5.1, 4.5.2, 4.5.3 and 4.5.4 below. ☐

Lemma 4.5.1. $a \equiv b \implies C(a) = C(b)$ *and* $a.pa = b.pa$.

Proof. If $a \equiv b$, then from the properties of the intentional equivalence relation, $a.pa = b.pa$, and one of the following five cases must hold:

1. a and b are initial sending events.
2. $\exists\, c$: $c \xrightarrow{\text{n}} a$ and $c \xrightarrow{\text{n}} b$
3. $\exists\, c, \tilde{c}$: $c \equiv \tilde{c}, c \xrightarrow{\text{n}} a, \tilde{c} \xrightarrow{\text{n}} b$
4. $\exists\, c$: $c \xrightarrow{\text{c}} a$ and $c \xrightarrow{\text{c}} b$
5. $\exists\, c, \tilde{c}$: $c \equiv \tilde{c}, c \xrightarrow{\text{c}} a, \tilde{c} \xrightarrow{\text{c}} b$

We prove the statement of the lemma by induction.

Base case: c, $\tilde{c}$ are initial sending events.
From item 1: $c \equiv \tilde{c}$.
From Algorithm 4.1, line 2: $C(c) = C(\tilde{c}) = 0$.
Then, $c \equiv \tilde{c} \implies C(c) = C(\tilde{c})$.

Inductive hypothesis: $C(c) = C(\tilde{c})$. (For cases 2 and 4, $c = \tilde{c}$ and this hypothesis holds trivially).

Inductive step: We show that the statement of the lemma also holds for a and b.
In cases 2 and 3, by Lemma 4.5.6:
$C(a) = C(c) + 1 = C(\tilde{c}) + 1 = C(b)$.
In cases 4 and 5, by Lemma 4.5.5:
$C(a) = C(c) + 1 = C(\tilde{c}) + 1 = C(b)$. ☐

Lemma 4.5.2. $C(a) = C(b)$ *and* $a.pa = b.pa \implies a \equiv b$.

Proof. We prove this by induction on $l = C(a) = C(b)$.

Base case: For $l = 0$, a and b are initial sending events.
By rule 1 of intentionality: $a.pa = b.pa \implies a \equiv b$.

Inductive hypothesis: $\forall\, e,\ f : C(e) = C(f) = k - 1$ and $e.pa = f.pa \implies e \equiv f$.

Inductive step: We show that, $\forall\, a,\ b : C(a) = C(b) = k$ and $a.pa = b.pa \implies a \equiv b$.

Case 1: a and b are input events.
By Definition 3.1: $\exists\, e, f$: $e \xrightarrow{\text{n}} a$ and $f \xrightarrow{\text{n}} b$.
Then, from Definition 3.1: $e.pa = a.pa$ and $f.pa = b.pa$
Thus, $e.pa = f.pa$.
By Lemma 4.5.6: $C(e) = C(a) - 1 = k - 1$.
By Lemma 4.5.6: $C(f) = C(b) - 1 = k - 1$.
Thus, by the inductive hypothesis: $e \equiv f$.
Thus, by rule 3 of intentional equivalence: $a \equiv b$.

Case 2: a and b are output events.
By Definition 3.2: $\exists\, g, h$: $g \xrightarrow{\text{c}} a$, $h \xrightarrow{\text{c}} b$, and $g.pa = h.pa$.
Then, from Lemma 4.5.5: $C(g) = C(a) - 1 = k - 1$.
Also, from Lemma 4.5.5: $C(h) = C(b) - 1 = k - 1$.
Thus, by inductive hypothesis: $g \equiv h$.
Thus, by rule 5 of intentional equivalence: $a \equiv b$.

Case 3: a is an input event and b is an output event.
We prove that this is an impossible case.
By Definition 3.1: there exists an input event e, such that $e.pa = a.pa$ and $e \xrightarrow{\text{n}} a$.
By Definition 3.2: there exists an output event f, such that $f.pa = b.pa$ and $f \xrightarrow{\text{c}} b$.
Then, by Lemma 4.5.6: $C(e) = C(a) - 1 = k - 1$.
Also, from Lemma 4.5.5: $C(f) = C(b) - 1 = k - 1$.
Then, $C(e) = C(f)$ and $e.pa = f.pa$.
From the induction hypothesis: $e \equiv f$.
By the properties of the intentional equivalence relation, e and f are either both input events or both output events.
Contradiction. □

Lemma 4.5.3. *$a \to b \implies C(a) < C(b)$.*

Proof. By the definition of the intentionality relation (Section 3.6.2), $a \to b$ has 5 possible cases.

Case 1: $a \overset{\text{n}}{\rightarrow} b$.
Then, by Lemma 4.5.6: $C\left(b\right) = C\left(a\right) + 1$.
Thus, $C\left(a\right) < C\left(b\right)$.
Case 2: $a \equiv \tilde{a}$ and $\tilde{a} \overset{\text{n}}{\rightarrow} b$.
From Lemma 4.5.1: $a \equiv \tilde{a} \implies C\left(a\right) = C\left(\tilde{a}\right)$.
By Lemma 4.5.6: $C\left(b\right) = C\left(\tilde{a}\right) + 1$.
Thus, $C\left(a\right) < C\left(b\right)$.
Case 3: $a \overset{\text{c}}{\rightarrow} b$
From Lemma 4.5.5: $C\left(b\right) = C\left(a\right) + 1$.
Thus, $C\left(a\right) < C\left(b\right)$.
Case 4: $a \equiv \tilde{a}$ and $\tilde{a} \overset{\text{c}}{\rightarrow} b$.
From Lemma 4.5.1, $C\left(a\right) = C\left(\tilde{a}\right)$.
From Lemma 4.5.5: $C\left(b\right) = C\left(\tilde{a}\right) + 1$.
Thus, $C\left(a\right) < C\left(b\right)$.
Case 5: $a \rightarrow c$ and $c \rightarrow b$.
From cases 1-4: $C\left(a\right) < C\left(c\right)$ and $C\left(c\right) < C\left(b\right)$.
Therefore, $C\left(a\right) < C\left(b\right)$. □

Lemma 4.5.4. $C\left(a\right) < C\left(b\right) \implies a \rightarrow b$.

Proof. $C\left(a\right) < C\left(b\right) \implies C\left(b\right) = C\left(a\right) + k,\ k > 0$.
We prove the lemma by induction on k.

Base case: For $k = 1$, $C\left(b\right) = C\left(a\right) + 1$

Case 1: b is an input event.
By Definition 3.1: $\exists\, c : c \overset{\text{n}}{\rightarrow} b$.
By Lemma 4.5.6: $C\left(b\right) = C\left(c\right) + 1$.
Thus, $C\left(c\right) = C\left(a\right)$.
From Lemma 4.5.2: $c \equiv a$.
Hence, from rule 2 of intentionality: $a \rightarrow b$.

Case 2: b is an output event.
By Definition 3.2: $\exists\, c : c \overset{\text{c}}{\rightarrow} b$.
From Lemma 4.5.5: $C\left(b\right) = C\left(c\right) + 1$.
Thus, $C\left(c\right) = C\left(a\right)$.
From Lemma 4.5.2: $c \equiv a$.
Hence, from rule 4 of intentionality: $a \rightarrow b$.

Inductive hypothesis: Let, for some $k > 1$
$\forall\, e, f : C\left(f\right) = C\left(e\right) + k \implies e \rightarrow f$.

Inductive step: We show that
$\forall\, a, b : C\,(b) = C\,(a) + k + 1$, $a \rightarrow b$.
$\exists\, c : C\,(b) = C\,(c) + 1, C\,(c) = C\,(a) + k$.
From the inductive hypothesis: $a \rightarrow c$.
Also, from the base case: $c \rightarrow b$.
Therefore, from rule 5 of intentionality: $a \rightarrow b$. ☐

Lemma 4.5.5. $a \xrightarrow{c} b \implies C\,(b) = C\,(a) + 1$

Proof. In Algorithm 4.2, a is either from a reception event from a PA with label C' or a timeout event with a label C' (line 6). In line 8, we have $C = C' + 1$ and the sending event b has a label C. Thus, $C\,(b) = C\,(a) + 1$.

In Algorithm 4.3, the sending event b has a label C that is updated in lines 10 and 12. In line 10, we have $C = C\,(a)$ and in line 12, we have $C = C + 1$. Thus, $C\,(b) = C\,(a) + 1$. ☐

Lemma 4.5.6. $a \xrightarrow{n} b \implies C\,(b) = C\,(a) + 1$

Proof. **At the PA**
A sending event a at a controller replica occurs in line 17 of Algorithm 4.1.
The corresponding reception event b at a PA occurs at line 8 of Algorithm 4.3.
From Algorithm 4.3, line 8: $C\,(b) = C\,(a) + 1$.

At the Controller
Algorithm 4.1 line 9: a reception event b has a label $b.l = l + 1$, where l is the label of the corresponding sending event a at a PA (Algorithm 4.3, line 16).
Thus, $C\,(b) = C\,(a) + 1$.

Algorithm 4.2 line 4: We declare timeout events with label C', for each PA i, such that the maximum label in $\mathcal{L}$ of the events corresponding to i is different from C'.
We trace the events that caused one such timeout event b.
Since $a \xrightarrow{\text{n}} b$, it follows that a is the sending event at a PA, such that $a.pa = b.pa$ that was lost or delayed, thus causing the timeout event b.
Let g be the last sending event at a controller, such that there exists a chain of events $g \xrightarrow{\text{n}} f \xrightarrow{\text{c}} d \xrightarrow{\text{n}} c$, where c is a reception event at the controller on which b occurred, and $c.l = C'$. (Definitions 3.1 and 3.2).
Thus, from result of this lemma at the PA: $d.l = C' - 1$.
From Lemma 4.5.5: $f.l = C' - 2$.
From the earlier statement at the controller: $g.l = C' - 3$.
The controller declares timeout events to acknowledge that it lacks information from some PAs, that it has from other PAs. This information is the state of the sub-processes after the implementation of the setpoints encapsulated in the last sending events that

resulted from the same computation relation at some controller replica.
Thus, from Definitions 3.1 and 3.2: there exists another chain of events, $g \xrightarrow{\text{n}} h \xrightarrow{\text{c}} a$, such that $h.pa = a.pa$.
From result of this lemma at the PA: $h.l = C' - 2$.
From Lemma 4.5.5: $a.l = C' - 1$. ☐

Theorem 4.5.2 (State Safety & Optimal Selection)**.** *A CPS that implements Algorithms 4.1, 4.2 and 4.3 guarantees state safety and optimal selection.*

Proof. The proof has two parts: state safety and optimal selection.

State Safety

At the controller: Algorithm 4.2 line 10, the controller computes setpoints using measurements with label C'.
C' is greater than the label of the last setpoint issued at line 17 in Algorithm 4.1 with label C, as $C' = \max(C+3, \max(\mathcal{L}))$ (Algorithm 4.2, line 1).
From Theorem 4.5.1: $C\,(b) < C\,(a) \implies b \rightarrow a$.
Thus, when an event a with label C' is used by the controller for computation, and another event b was the last event issued by the controller with label C, then $C' > C \implies b \rightarrow a$.

At the PA: The computation at PA, i.e., implementation of a setpoint followed by subsequent computation of a measurement, is triggered only if Algorithm 4.3 line 9 is true.
Therefore, if an event a with label $a.l$ is used in a computation by a PA when its CPS causal clock is C, then $C < a.l$.
However, C is the label of the sending event b, corresponding to the last measurement issued.
Therefore, by Theorem 4.5.1: $C\,(b) < C\,(a) \implies b \rightarrow a$.

Optimal Selection

At the controller: Consider a reception event a with label $a.l$ at a controller, when its clock is C.

Case 1: $a.l \leq C$.
Let b is a sending event corresponding to the last setpoint sent to the same PA.
From Theorem 4.5.1: $C\,(a) \leq C \implies b \not\rightarrow a$.
From Algorithm 4.2 line 1, we have $a.l < C + 3 < C'$.
In line 10, the controller computes with events of label C'.
Thus, the event a is not used in computation, i.e., discarded.

Case 2: $a.l > C$ and $a.l \neq \max(\mathcal{L})$.
Consider an event e such that $C(e) = \max(\mathcal{L})$.
From Theorem 4.5.1: $C(a) < C(e) \implies a \to e$.
From Lemma 4.5.7: $C(a) < C(e) \implies C(e) > C(a) + 4$.
Let d be the sending event on a PA, such that $d \xrightarrow{\text{n}} e$.
Then, $C(d) = C(e) - 1$.
Let c be the reception event on the PA, such that $c \xrightarrow{\text{c}} d$.
Then $C(c) = C(e) - 2$.
Let b be the sending event on a controller, such that $b \xrightarrow{\text{n}} c$.
Then $C(b) = C(e) - 3$.
Thus, $C(b) > C(a)$.
By Theorem 4.5.1: $C(b) > C(a) \implies b \not\to a$.
Hence, a must be discarded.
From Algorithm 4.2 line 1, we have $a.l < \max(\mathcal{L}) < C'$.
In line 10, the controller computes with events of label C'.
Thus, the event a is not used in computation, i.e., discarded.

Case 3: $a.l > C$ and $a.l = \max(\mathcal{L})$
From Lemma 4.5.7: $a.l > C + 3$.
From Algorithm 4.2 line 1: $a.l = C'$.
In line 10, the controller computes with events of label C'.
Thus, the event a is used in computation, i.e., not discarded.

At the PA: An event a is discarded by the PA only if $a.l \leq C$ (Algorithm 4.3, line 9), where C is the label of the event b corresponding to the last measurement sent.
Thus, from Theorem 4.5.1, if a is discarded, $b \not\to a$. ☐

Lemma 4.5.7. *For any two events a and b occurring at the same software agent, if a and b are both input events or both output events, then* $[C(b) - C(a)] \ \% \ 4 = 0$.

Proof. We start by proving the statement for output events at the controller, by using induction.

Base case: Let a be an initial sending event. Then, $C(a) = 0$.
Let b be an event occurring on a PA, such that $a \xrightarrow{\text{n}} b$.
Let c be a sending event on the same PA, such that $b \xrightarrow{\text{c}} c$.
Let d be an event on the controller, such that $c \xrightarrow{\text{n}} d$.
Let e be the sending event on the controller, such that $d \xrightarrow{\text{c}} e$.
Then, $C(e) = 4$ and $[C(e) - C(a)] \% 4 = 0$, where e and a are both output events. Thus, the statement is true for the first two sending events on all controllers.

Inductive hypothesis: Consider an instance of the execution trace at time t_0 at which there is no event in the CPS b, such that $C(b) \geq l$. At this instant, let $\mathcal{A}$ be the set of all sending events on all controller. Clearly, $\forall a \in \mathcal{A}, C(a) < l$. We assume the hypothesis be true for all $a \in \mathcal{A}$.

Inductive step:
Consider $a_0 \in \mathcal{A}$, be such that $\forall\, a \in \mathcal{A}, C(a_0) \geq C(a)$.
By the induction hypothesis: $C(a_0) = 4k$, where $k \in \mathbb{N}$.
We show that the hypothesis is also true for the first sending event b that occurs at a controller after t_0 such that $a_0 \to b$.
$\forall\, c, (c.sa = \mathrm{PA}_1 \wedge a_0 \overset{\mathrm{n}}{\to} c) \implies C(c) = 4k+1.$
$\forall\, d, (d.sa = \mathrm{PA}_1 \wedge c \overset{\mathrm{c}}{\to} d) \implies C(d) = 4k+2.$
$\forall\, e, (e.sa \in \mathcal{C} \wedge\ e.pa = \mathrm{PA}_1 \wedge d \overset{\mathrm{n}}{\to} e) \implies C(e) = 4k+3.$
$\forall\, f, (f.sa = e.sa \wedge\ e \overset{\mathrm{c}}{\to} f) \implies C(f) = 4k+4.$
Thus, $[C(f) - C(a_0)] \,\%\, 4 = 0$.
f is a sending event on a controller and $C(f) > C(a_0)$.
Thus, $f \notin \mathcal{A}$.
Also, f is the first sending event that occurred after t_0.
Thus, the hypothesis holds for all sending events at controllers.

Sending events at controllers have label of the form $4k$.
By Lemma 4.5.6: input events at PAs have label of the form $4k+1$.
From Algorithm 4.3: output events at the PAs have label of the form $4k+2$.
By Lemma 4.5.6: input events at the controller have label of the form $4k+3$. □

4.6 Case Study: CPS for Scheduling EV Charging

We present, via a case study, a practical application of the intentionality relation and the intentionality-clocks mechanism. We take the example of a CPS for scheduling the charging of a fleet of EVs, which provides a schedule that accounts for both the vehicles' demand and the vehicle-to-grid regulation services [1].

One of the purposes of this CPS is to charge each vehicle based on the vehicle's demand, and the other purpose is to provide frequency support for the grid. Modulating the charging schedule of the EVs, by charging at a higher or lower rate, or even discharging into the grid for some time in cases of downward excursion of frequency, can provide frequency support. In practice, the EVs must respond to regulation requests every two to four seconds [170].

The paper [1] presents a solution to the problem assuming an ideal communication, and without considering the failure of the software agents. However, such a mission-critical control requires high levels of reliability in a real deployment. It is, therefore, desirable to replicate the controller.

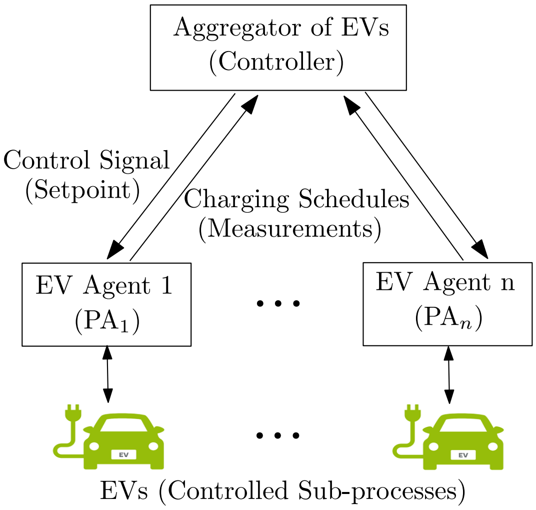


Figure 4.5 – Architecture and information flow of a CPS for EV charging [1]

In Section 4.6.2, we use the intentionality relation to analyze their CPS design for possible issues that could arise when the controller is replicated. We find that a naive extension to their design violates the optimal-selection property. Consequently, due to software and network faults, the CPS can encounter a deadlock situation, whereby frequency support will no longer be achieved. This might result in the instability of the underlying electrical grid.

4.6.1 CPS Design for EV Charging

In this section, we analyze the system model of [1] and show how our system model, presented in Section 3.3, applies.

Figure 4.5 presents an architectural view of the system consisting of EVs and the aggregator, as shown in [1]. It comprises EVs that represent the controlled sub-process from our model, the EV agents labeled 1 through n that correspond to the PAs from our model and the central aggregator that is the controller of the CPS. The controller receives a charging/discharging (henceforth referred to as charging) schedule from each PA. This schedule represents the measurements in our model. The controller computes control signals (setpoints) for each of the EVs, using the received charging schedules.

The algorithms of the controller and the PAs used for achieving the desired vehicle-to-grid regulatory service are described in [1] in Algorithms 2A and 2B. We abstract this process and summarize it, as shown in Algorithm 4.4. The controller periodically starts an iterative process that consists of several control rounds, subject to convergence of the algorithm. Each iterative process is independent from the previous one, as if the controller had a fresh start. The triggering of a new iterative process (line 2) is a timeout event in our model, which further causes n sending events, one for each

PA (line 4). These events represent the initial sending events as they are computed without using measurements.

From Algorithm 4.4, we see that the CPS in [1] uses a labeling scheme with label m. In Section 4.6.2, we show that this labeling scheme violates optimal selection (Definition 3.4), consequently the system can enter a deadlock situation. In line 6, we see that, before beginning computation in line 7, the controller waits for schedules from each PA. Thus, the `ready_to_compute` function of this CPS is the presence of one reception event corresponding to the current round with label m, from each PA. Timeout events do not occur within an iteration.

In lines 7-13, we see that the computation relation takes as input one schedule from each PA and produces one control signal for each PA. In other words, the computation relation takes as input one reception event from each PA and outputs one sending event for each PA. This is same as our computation relation (Definition 3.2).

The iterative process (lines 5-13) continues until the control has converged. It terminates with the sending of control and stop signals in the setpoints of the last computation.

Each EV agent (PA) also keeps track of the on-going control round by using label m. It only accepts charging schedules (setpoints) that belong to the current round (line 20). Upon receiving a setpoint from the current round, the PA checks if it is accompanied with a stop signal (line 22). The presence of a stop signal is an indication of the termination of the iterative process and results in implementation of the setpoint. Alternatively, when the stop signal is absent, the PA sends its new charging schedule as a measurement to the controller and increments m by one.

4.6.2 Deadlock Due to Violation of Optimal Selection

Here, we describe a scenario in which the CPS in [1] can enter a deadlock situation when the controller is replicated, due to the controller replicas having different round labels.

Consider a scenario with two replicas of the controller C_1 and C_2, with label m_1 and m_2, respectively. Consider a PA, PA$_0$, with label m_0. Consider a situation when both controllers sent out setpoints to PA$_0$ with label 5. Then, $m_1 = m_2 = 5$. PA$_0$ receives this setpoint, implements it, sends a measurement with label $m_0 = 5$ and increments its label to 6. Thus, after this round, we have $m_1 = m_2 = 5$ and $m_0 = 6$. Now, if C_2 does not receive the measurement from PA$_0$ and C_1 received measurements from all PAs, C_1 will begin computation and C_2 will be stalled. After this computation, we have $m_1 = 6$ and $m_2 = 5$. Moreover, PA$_0$ implements the setpoint and increments its label m_0 to 7.

Algorithm 4.4: Abstraction of the iterative process of scheduling EV charging [1]

```
At the Aggregator (Controller)
repeat periodically
    $m \leftarrow 0$;
    Send a request for schedules with label $m$ to each PA;
    repeat
        if schedules labeled $m$ from each PA are received then
            Perform computation of control signals;
            $m \leftarrow m + 1$;
            if control has not converged then
                Send new control signals with label $m$ to each PA;
            end
        end
    until control has converged;
    Send control signals and stop signals with label $m$ to each PA;
forever;

At the EV Agent (PA)
$m \leftarrow 0$;
on reception of a control signal with label $k$ from aggregator
    if $k == m$ then
        Compute new charging schedule;
        if stop signal received then
            $m \leftarrow 0$;
            Implement charging schedule until next control signal;
        else
            Send schedule to aggregator with label $m$;
            $m \leftarrow m + 1$;
        end
    end
end;
```

Next, let C_1 crash due to a software failure and reboot. Then, it requests for schedules with $m_1 = 0$. However, as PA_0 is expecting messages from round 7, it ignores these requests. PA_0 also discards any messages from C_2, as they will have a label 5. Moreover, C_2 discards the received measurements with label 6 because its label is $m_2 = 5$. This is a violation of the optimal-selection property by C_2. Hence, the CPS enters a deadlock state because there is no mechanism to resynchronize the label.

4.6.3 Design with Intentionality Clocks

To remedy the problem of deadlock, we altered the labeling mechanism in Algorithm 4.4 with intentionality clocks. The new design is given by Algorithm 4.5. At the controller, we resynchronize the label using Rule 4(a) of intentionality clocks, i.e., by declaring timeout events in order to move to a new round of computation. This mechanism is given by lines 6-8 of Algorithm 4.5. Furthermore, at the PA, we solve of the issue of not recording reception events with lower round labels by adding lines 33-35. Upon a reception event with a lower label, the PA must not implement the setpoint in order to satisfy state safety. Instead, the PA advertises its most recent state by sending the schedule with its current label m.

As proven earlier, our design (Section 4.4) satisfies the optimal selection property even in the presence of controller replication and of software and network faults (as shown in Theorem 4.5.2). Therefore, the problem encountered by the design in [1] can be avoided by applying intentionality clocks and tuning the design of the software agents according to Algorithm 4.5.

Algorithm 4.5: Design of scheduling EV charging in [1] with intentionality clocks

```
At the Aggregator (Controller)
repeat periodically
    $m \leftarrow 0$;
    Send a request for schedules with label $m$ to each PA;
    repeat
        if a schedule with label $k > m$ is received from a PA then
            $m \leftarrow k$;                    // Declare timeout events for all $m < k$
        end
        if schedules labeled $m$ from each PA are received then
            Perform computation of control signals;
            $m \leftarrow m + 1$;
            if control has not converged then
                Send new control signals with label $m$ to each PA;
            end
        end
    until control has converged;
    Send control signals and stop signals with label $m$ to each PA;
forever;

At the EV Agent (PA)
$m \leftarrow 0$;
on reception of a control signal with label $k$ from aggregator
    if $k == m$ then
        Compute new charging schedule;
        if stop signal received then
            $m \leftarrow 0$;
            Implement charging schedule until next control signal;
        else
            Send schedule to aggregator with label $m$;
            $m \leftarrow m + 1$;
        end
    else
        if $k < m$ then
            Send schedule to aggregator with label $m$;
              // Declare reception event
        end
    end
end;
```

4.7 Conclusion

We address the problem of enabling software agents in a CPS, namely controller and PAs, to provide a notion of rounds of computation in the presence of network losses or delays, or of the replication of the controller. We show that, in such settings, the causal (or happened-before) relation, traditionally used in distributed systems literature, does not enable the capturing of the control rounds. Instead, we find that the newly proposed intentionality relation is a good choice for enforcing such rounds.

We present a clock mechanism, intentionality clocks; it was used in the design of a controller and a PA that guarantees the strong clock-consistency condition under the intentionality relation. We consider two correctness properties, namely state safety and optimal selection: they describe how the agents must treat events in order to respect intentionality. We prove that the design guarantees these properties. Lastly, through a case study of a real-world CPS for charging EVs, we demonstrate the practical relevance of the introduced concepts.

As the design of the software agents with intentionality clocks now enables using round-numbers to order events in an otherwise orderless CPS, designing reliability mechanisms for a CPS with intentionality clocks is easier. As these clocks impose a round number on a CPS with asynchronous software agents, the asynchrony is hidden from the agents who can simply compare the labels to check whether a received message belongs to a newer control round or an older control round. Moreover, the execution trace of a CPS with intentionality clocks can be represented as a DAG with the nodes being the events and the edges being the network and computation relations between those events. Such a DAG lends itself well to automated checking of properties on a CPS.

Therefore, we can focus on ensuring the rest of the correctness properties such as consistency and timeliness within one round. In Chapter 6, we present results from the implementation of intentionality clocks, along with other reliability mechanisms, in a CPS for real-time control of electric grids [4]. We study the impact of violations of the correctness properties on the physical process, through experiments in a virtual commissioning environment [61].

One possible avenue for the extension of the theory of intentionality relation and the design of intentionality clocks is to include CPSs with hierarchy of controllers, or those with asynchronous sensors that send out-of-band measurements to the controllers.

5 Quick Agreement among Replicated CPS Controllers

To talk, or not to talk,
that is the question.

In this chapter, we present mechanisms for ensuring that the CPS guarantees the consistency property (Definition 3.5 in Section 3.7). In order to ensure consistency, the controller replicas need to perform agreement, before issuing setpoints to the PAs. Traditionally, in active-replication schemes, agreement is achieved using consensus [161]. However, consensus is known to have a poor latency-performance as the agreement requires several rounds of message exchanges between the replicas and can take an unbounded amount of time to terminate [88]. This makes consensus unsuitable for real-time CPSs, as they require consistency with a low-latency overhead.

To address the latency issues with consensus, we present Quarts; this name stands for **qu**ick **a**greement in **r**eal-**t**ime **s**ystems. In Quarts, the controller replicas perform agreement on input, *i.e.,* measurements, as opposed to agreement on output, *i.e.,* setpoints as done in some consensus mechanisms [161]. Then, certain properties of CPSs enable the controller replicas to locally choose which measurements to use in a computation without having to exchange messages ("talk") with other replicas. In Quarts, a replica can choose to not talk with other replicas and reach agreement in such a way that consistency is always guaranteed. In rounds when the replicas do not exchange messages for agreement (which are a majority of the rounds in CPSs with two replicas), the latency overhead is zero. In other cases, the latency overhead is bounded. In this way, by efficiently deciding when to and when not to talk with other replicas for agreement, Quarts provides a bounded latency-overhead and a lower average latency-overhead when compared to consensus.

Quarts is applicable to CPSs with only the controller and PAs, and not with asynchronous sensors. This is because Quarts relies on labeled measurements, where the labels are consistent across all PAs, like in the case of labels obtained from intentionality clocks (Chapter 4). Asynchronous sensors do not share the same labeling scheme, hence need to be handled differently. To this end, we propose an extension to Quarts, namely Quarts+. It ensures consistency and bounded latency-overhead similarly to Quarts at a slightly reduced availability. Both Quarts and Quarts+ have a considerably higher availability than existing passive- and active-replication techniques for consistency.

The rest of this chapter is structured as follows. We begin by explaining the need for consistency, in Section 5.1. We also elaborate on the intuition behind the definition of the consistency property (Definition 3.5) based on output events in the CPS, and we relate it to the setpoints output by the controller replicas. In Section 5.3, we list the properties of CPSs required by Quarts. We present the design of Quarts in Section 5.4, followed by the proofs of the consistency and bounded latency-overhead properties of Quarts in Section 5.5.

In Section 5.6, we use discrete-event simulation to compare the consistency, availability, latency and messaging-cost of passive and active replication schemes with that of Quarts. We find that unlike Quarts that provides 100% consistency, passive-replication schemes have non-zero inconsistency, which renders them unsafe as a replication mechanism for CPS controllers. We also find that Quarts can provide availability up to $10\times$ higher than consistency guaranteeing active-replication schemes such as consensus. Furthermore, both the average and tail-latency of Quarts are bounded and lower than consensus. Lastly, we find that Quarts has a messaging cost similar to consensus but higher than passive-replication schemes. The higher messaging-cost is the price of consistency.

In Section 5.7, to provide agreement in CPSs with asynchronous sensors, we present the extension, Quarts+. In Section 5.2, we review the related work on consistency achieving mechanisms. We summarize our findings and conclude this chapter in Section 5.8.

5.1 Importance of Consistency in a CPS

Let a and $\tilde{a}$ be two intentionally equivalent setpoints sending events occurring on different controllers. From Theorem 4.5.1 that proves the strong-clock consistency in a CPS that uses the controller and the PA design presented in Chapter 4, we have $a.l = \tilde{a}.l$. In other words, a and $\tilde{a}$ belong to the same computation round.

Although, the setpoints encapsulated by a and $\tilde{a}$ belong to the same computation round, they can result from computations that use different sets of input events (different combinations of receptions and timeout events). This is possible because of redundancy in the sensor network, or due to the controllers' ability to compute with partial information [160]. Furthermore, when the computations use different input events, the resulting output events can encapsulate different messages. Recall from Section 3.7 that this violates the consistency property that states that if two output events are intentionally equivalent, then they encapsulate the same message.

As the physical process can have several feasible operating points, it is natural for two different controller replicas to output different setpoints in the computation round, both of which are feasible. Hence, on its own, the violation of consistency property does not necessarily affect the control of the process. However, as these setpoints are sent over a network that can drop, delay, or reorder messages, their implementation might be interleaved. Hence, the control of the process resulting from the final implementation of setpoints can deviate from the correct control behavior of the CPS. In other words, issuing setpoints that violate the consistency property might result in an incorrect control of the CPS, as clarified by the following example.

Consider an operation of the COMMELEC CPS [4]: it controls a microgrid by modulating the power injections of a battery and a PV. The line connecting the microgrid to the main grid cannot support more than $100\ kW$. One controller replica might instruct the PV to inject $50\ kW$ and the battery to absorb $50\ kW$. Another controller replica, due to receiving a different subset of input measurements, might instruct the PV to inject $200\ kW$ and the battery to absorb $200\ kW$. Although setpoints from either one of the controller replicas respect the ampacity limits, the interleaving of setpoints from the two replicas does not. In other words, a sequence of message losses might result in the PV receiving only the instruction to inject $200\ kW$, and the battery receiving the instruction to absorb $50\ kW$. This causes the grid to export $150\ kW$, thus violating the limits of the main line.

Although two events a and $\tilde{a}$ have the same labels, they might not be encapsulating the same messages, thereby violating consistency for events a and $\tilde{a}$. Furthermore, the corresponding output events at PAs, c such that $a \xrightarrow{\text{n}} b \xrightarrow{\text{c}} c$ and $\tilde{c}$ such that $\tilde{a} \xrightarrow{\text{n}} \tilde{b} \xrightarrow{\text{n}} \tilde{c}$, might also not encapsulate the same messages, thereby violating consistency for events c and $\tilde{c}$. This effect can be noticed for all subsequent setpoints and advertisements, *i.e.,* violating consistency for two output events can violate consistency for subsequent output events. In order to guarantee consistency, and to be able to enforce intentional equivalence in practice, there needs to be a mechanism that ensures that when two controller replicas perform computations to issue setpoints with the same label, the setpoints for the same PA have the same value. We call this property *computation consistency*, and we show in Lemma 5.5.5 that it is sufficient to ensure consistency for all output events.

5.2 Related Work

Achieving consistency among the replicas within bounded latency is impossible, as shown in the CAP theorem [171]. The CAP theorem states that during a partition (in an asynchronous network), consistency and availability (bounded latency) cannot be achieved together. Hence, the problem of ensuring consistency among replicas involves trading off availability for consistency or vice versa under different fault models. Broadly, two types of solutions tackle this problem from different perspectives: passive replication [99] and consensus-based active replication [114, 115, 161, 172].

Passive replication [99] is commonly used in CPSs as it avoids agreement by having one primary replica compute and issue setpoints. If the primary is detected as faulty, the standbys elect a new primary by consensus. As failure detection is imperfect in an asynchronous setting [58], the standbys might incorrectly detect the primary as faulty and elect a new primary, and the original primary would not be notified, resulting in two primaries, thus leading to potential inconsistencies. Most passive-replication research considers the crash-only fault model, under which passive replication improves availability at a negligible cost of inconsistency. For real-time CPSs, however, delay faults are more relevant. The intermittent nature of delay faults causes more false positives by failure detectors, hence more inconsistencies. Imperfect fault-detection can also lead to false negatives, in which case the primary is faulty and undetected, resulting in poor availability. Moreover, the election of a new primary replica relies on consensus between the backup replicas, and consensus has poor latency performance, thereby unsuitable for real-time systems.

Active-replication solutions work by having all replicas receive inputs and compute setpoints. These replicas then hold a consensus to agree on one of two equivalent decisions: either which replica issues the setpoints, or what set of setpoints all replicas should issue. As mentioned in Section 2.1.1, consensus consists of four properties: agreement, termination, validity, and integrity [115]. Validity and integrity address issues that arise due to Byzantine faults and are not under consideration in this work. Agreement and termination are impossible to guarantee together, within a bounded delay and in an asynchronous setting [88]. As real-time CPSs require low latency, this results in low availability.

Quarts focuses on a class of CPSs for which termination is not always necessary, as they can correctly compute setpoints even if some previous computation was missed (refer to properties in Section 5.3). This enables agreement via voting that, irrespective of the success, ends after a bounded time. Moreover, performing agreement on the input, rather than over the setpoints or the issuing replicas, affords a phase of collection prior to agreement. This increases the chances of a successful vote, thereby improving availability.

We use a composite voter [173] that, in cases of no majority, selects the output of one of the replicas, based on a predetermined order. Unlike in [173], where the order is replica-based, *i.e.,* in the case of a tie, one of the replica has a priority higher that the others, we use an ordering based on the number of measurements, *i.e.,* in the case of a tie, the replica with most measurements has a higher priority. This scheme is more suited to CPSs as it results in the option with most measurements being chosen. Also, we use plurality voting or relative-majority voting [174], instead of absolute-majority voting. In plurality voting, the option with most votes is chosen irrespective of its fraction of absolute votes, thereby providing a higher chance of agreement.

5.3 CPS Properties Required for Quarts

Figure 5.1 shows the architecture of a CPS with one or more replicated controllers, m PAs, and no asynchronous sensors, and it indicates the messages exchanged with a round labeled with label r. We assume that the controllers and PAs follow the design with the logical clocks presented in Chapter 4. Thus, all messages are associated with a label derived from a logical clock of the sending software agent. The m PAs read measurements from their sensors and send them to the controllers, each measurement is marked with a label (r). To ensure agreement, Quarts requires only that the messages are consistently labeled and does not specifically require labels derived from intentionality clocks. Labeling can be also be obtained by time-synchronization or through other logical clocks [85, 118]. However, for simplicity, we consider that the labels are derived from intentionality clocks.

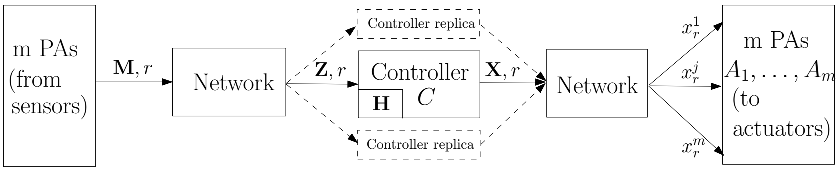


Figure 5.1 – Architecture of the CPS indicating the messages exchanged with round numbers

The model of the controller that receives and processes these measurements to produce setpoints is given by Algorithm 5.1. This algorithm is similar to Algorithm 4.2 in Chapter 4 with some key modifications. First, we express the variables in terms of measurements and setpoints, as opposed to using events, because they lend themselves well to explaining the mechanisms in Quarts. Second, we make the controller state ($\mathbf{H}$) explicit because it affects the computed setpoints, consequently affecting consistency. Third, we do not mention the set of all received measurements $\mathcal{M}$ and the fact that computations can be triggered by current time T_{now}, as these parameters have no bearing on the design of Quarts and its guarantees. Last, the function `choose_measurements` (Algorithm 4.2) that is pivotal in providing the state safety and optimal selection property is assumed to be implemented within the `compute` function (Algorithm 5.1 line 13).

Algorithm 5.1: Model of the CPS controller with labels and without events

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

Note that, the computation of setpoints $\mathbf{X}$ corresponding to label r depends on measurements of label r ($\mathbf{Z}$), state ($\mathbf{H}$), and correction factor $(r - r^-)$. The correction factor indicates when the last setpoint was computed and is used to handle intermittent measurements, as discussed in Property 5.2. Also, each computation results in a new state. To highlight the updates of the controller state in each computation, we introduce a new function `update_state` (Algorithm 5.1 line 14) that uses the vector of measurements ($\mathbf{Z}$), the previous state ($\mathbf{H}$) and the correction factor $(r - r^-)$.

The measurements and setpoints are sent over a non-ideal network that might delay or drop messages. In case of no loss, the propagation delay is bounded by δ_n. The message processing time of non-faulty replicas is negligible with respect to. δ_n. This network model is called probabilistic synchronous [58].

Given that the state is used in the computation of setpoints (line 13), Quarts requires replicas to agree on the state. To this end, the following property is expected to hold.

Property 5.1. *The state used by the controller for computing the setpoints can be known exactly.*

This property is satisfied by a wide range of controllers, including controllers using Kalman filters [41] where the state is the gain matrix, or more sophisticated controllers like [175], where the state is the time of the most recent voltage violation.

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

Property 5.2. *The controller can compute correct setpoints in the presence of intermittent measurements.*

In other words, a controller is assumed to be robust to non-idealities in the communication network or the software agents. Examples of such controllers are those that use a Kalman filter that accounts for intermittent measurements [159] and the robust COMMELEC controller that can handle missing measurements [160]. Controllers of CPSs that exhibit this property can compute and issue setpoints, despite intermittent faults, either in the network (losses of measurements) or in the controller (delay faults). On the contrary, CPS controllers that do not exhibit Property 5.2, will eventually permanently fail to compute and issue setpoints due to network losses, even in the absence of software faults. This is because a controller that fails to compute setpoints for a label r, cannot compute setpoints for all labels greater than r.

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

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

Property 5.3. *PAs are able to handle duplicate setpoints.*

This property is satisfied by the CPSs if they already satisfy the state-safety property (Theorem 4.5.2) by using intentionality clocks. Other CPSs can either use their labeling schemes to perform de-duplication of received setpoints or use absolute setpoints, rather than differential setpoints. An example of absolute setpoints is an electric-grid controller instructing a battery agent that is injecting $8\ kW$ to '*set the injected power to* $10\ kW$', rather than to '*increase the injected power by* $2\ kW$'. Receiving identical duplicates of the former has the same effect as receiving a single setpoint.

5.4 Quarts Design

We take a top-down approach to describing the design of Quarts. In Section 5.4.1, we give a walk-through of a typical operation of Quarts by using, as example, a CPS for the control of electric grids that uses a Kalman-filter based controller [176]. Then, in Sections 5.4.2, 5.4.3, we detail the design of the individual building blocks of Quarts.

5.4.1 Protocol Walk-Through

Quarts is applied to a CPS by replicating the controller, adding the red part (Algorithm 5.2 lines 12-14, 16), and implementing the `collect_and_vote` function (Algorithm 5.3) that implements Algorithms 5.4, 5.5 and 5.6. In order to guarantee consistency, this function performs agreement between the controller replicas and overwrites the set of measurements $\mathbf{Z}$, the state $\mathbf{H}$, and the label of the last setpoint computation r^-. The function returns False when this replica should not compute for this label. The `collect_and_vote` function has two parts, Collection and Voting, described in Section 5.4.2 and 5.4.3, respectively.

Algorithm 5.2: Model of the CPS controller with Quarts (in red).

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

The novelty of Quarts is that it performs agreement on measurements $\mathbf{Z}$ and the state $\mathbf{H}$ before computation, as opposed to agreement on setpoints done by existing solutions. By agreement on $\mathbf{H}$ for a label r, the replicas implicitly agree on the correction factor, $r - r^-$, which affects the subsequent `compute` and `update_state` functions. Our choice to agree on the input stems from the observation that it enables an optimization that increases the probability of a successful agreement. This optimization is referred to as the collection phase and involves replicas exchanging measurements and the state in order to minimize the missing information in each replica.

Agreement is done in the subsequent voting phase, where replicas exchange a digest of the measurements and the state they have. The subset of measurements corresponding to the most-common digest is chosen for computation. The details of the digest and the voting phase are explained in Section 5.4.3.

From Algorithm 5.2, note that the setpoints returned by the `compute` function are uniquely determined by **Z**, **H** and $r - r^{-}$. An example of such a controller is [176]: it uses a Kalman filter for estimating the state of an electric grid and uses the estimated state to compute and issue setpoints to PAs to maintain the grid in a feasible state. Here, **Z** is composed of voltage and current phasors, and **H** is the Kalman-gain matrix, and the $r - r^{-}$ represents the "jump" in the linear state of the controlled process since the last time a computation was performed by the controller. Recall that in a Kalman filter [159], the measurements, the Kalman-gain, and the correction-factor uniquely determine the output. Hence, agreeing on **Z**, **H**, and r^{-} for a given r is sufficient for agreeing on the value of the setpoints.

With Quarts, the aforementioned CPS [176] works as follows. Every 20 ms, the PAs send measurements, with a new label. When the first measurement with a new label r is received, the replica sets a timeout by which it expects to receive all measurements. One possible value of the timeout is the one-way delay of the network (δ_n). As the measurements are sent over a lossy network, the controller replicas receive different subsets of measurements. Upon timeout, each replica queries other replicas for the missing measurements. Moreover, each replica that has a state with a label less than $r - 1$ requests others for their most recent state. This way, more replicas have a similar state and a set of measurements after collection, and the subsequent voting phase is more likely to succeed.

Algorithm 5.3: `collect_and_vote(`$\mathbf{Z}$`, ` $\mathbf{H}, r, r^{-}$`)`

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

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

The collection phase in Algorithm 5.3 is realized through the functions `collect_missing_measurements` and `collect_missing_state` (Algorithms 5.4, 5.5). Each of these functions lasts for at most $2\delta_n$ at each replica, as detailed in Section 5.4.2. These functions are independent and can run simultaneously, resulting in a bounded latency of $2\delta_n$ for the collection phase.

At the end of the collection phase, each replica sends to other replicas a digest of the measurements it has obtained thus far. This begins the voting phase, in which each replica chooses, using the `vote` function (Algorithm 5.6), the set of measurements and the state to be used for computing setpoints corresponding to this label. If voting is unsuccessful, or if the corresponding replica does not possess the chosen measurements or state, then False is returned (Algorithm 5.3, line 12). This prohibits the replica from computing setpoints (Algorithm 5.2, line 16). The voting phase has a bounded duration of $3\delta_n$ and is explained further in Section 5.4.3.

Note that, the voting phase requires an upper bound on the number of controller replicas (faulty and non-faulty). This can be achieved either by statically configuring the number of replicas every time a new replica is added or removed, or by using a group membership algorithm such as [151]. As the addition of replicas is not done on the real-time path, the introduction of a new replica can wait until the termination of the group membership algorithm.

In the special case of two controller replicas, it is possible that both the collection and the voting phase at a controller might last for zero time. This case occurs when the controller receives measurements from all the PAs for this round, thereby not needing to query other controllers for measurements. Moreover, as we shall see in Section 5.4.3, receiving all measurements in a round entitles a controller in a two-replica system to decide without voting. Additionally, in CPSs where the network losses are low, this case occurs in a vast majority of the rounds for at least one of the replicas, thereby adding zero latency for agreement. Note that a replica that moves to computation on receiving all the measurements in a certain round still continues to help other replicas by participating in the collection and voting phases without delaying computation.

Lastly, a controller replica must perform agreement for a round at most once. In order to ensure this property across reboots, controllers store the highest round number in which it participated in agreement on a persistent storage (such as disk). Upon reboot, this number is retrieved and the replica does not take part in agreement for previous rounds.

5.4.2 Collection Phase

The collection phase consists of replicas exchanging measurements and state, as presented in Algorithms 5.4 and 5.5, respectively. As mentioned earlier, these can run

Algorithm 5.4: `collect_missing_measurements`($\mathbf{Z}$, r)

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

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

simultaneously. The collection phase, designed with delay faults in mind, is terminated after $2\delta_n$. Thus, non-delay-faulty replicas can proceed with the voting phase without waiting for delay-faulty ones.

The main premise of collection is that the measurements and the state with the same label have the same value. This is possible because of use intentionality clocks (described in Chapter 4). At a PA, every time it issues a measurement, it increments its local intentionality clock and tags the measurement with the value of the clock. Moreover, the intentionality clocks are always incremented and never decremented. Thus, two different measurements have different labels. Alternatively, measurements with the same label have the same value as they are generated by the same PA. The case of the state is more involved and is covered by Lemma 5.5.4.

Collecting Measurements

Each replica calls `collect_missing_measurements`($\mathbf{Z}$, r) (Algorithm 5.4). It forms S, the set of IDs of the PAs from which it has received measurements with label r. Then, it sends a *Quarts_Query* to all replicas asking for the missing measurements (entries of $\bar{S}$).

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

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

Algorithm 5.5: `collect_missing_state`($\mathbf{H}$, r^-)

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

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

Collecting State

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

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

5.4.3 Voting Phase

The collection phase ends with each replica sending its *digest* to all the voters. A digest with a label l consists of (1) an indicator set S of measurements with label l that this replica has, and (2) the state label r^- at this replica. Quarts uses a total order among digests. Consequently, we have a function $\max$ that returns the largest digest, based on the total order. For each label, there exists a largest possible digest (`full_digest`) that does not contain any missing measurements and that has $r^- = l - 1$. A possible implementation of the digest is a concatenation of the state label with the bit-mask of the measurements. For instance, computing setpoints with label $l = 25$: if r^- is 24 and a replica has received measurements from PAs 1, 2, and 5 $(m = 5)$, then its digest would be “24.11001”. The lexicographic ordering of the digest string gives the total order, with “24.11111” being the `full_digest`. We use this implementation in our simulations.

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

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

There are four cases in which the voter can choose a digest. In all other cases, the voter has to wait for more digests. (1) The number of empty cells in `v` is zero (lines 13-15). In this case, the voter has to choose one of the digests that are most common. If there is only one, it is chosen; otherwise, it chooses the largest among them. (2) There is only one most-common digest, and it will remain the most-common digest no matter what the replicas that have yet to send digests send (lines 16-18). It is chosen (as the obvious majority). (3) There is only one most-common digest, and there is at least another digest (second most-common digest), and no matter what the replicas that have yet to send digests send, any second most-common digest can be at most as common as the most common one (lines 19-24). In this case, if the most common digest is larger than all the second most common digests, the voter chooses it. (4) The `full_digest` is the only most common one and no other digest can become more common. As it is the largest by definition, it is chosen.

This approach enables the voter to successfully vote, before receiving digests from all replicas, while guaranteeing consistency and accounting for delay-faulty replicas. For instance, in a CPS with two controller replicas, when one of the replicas has `full_digest` and has not received any other digests from other replicas, we have $\mathbf{f}_{mc} = 1$, $\mathbf{f}_{sec} = 0$ and $f_0 = 1$. Thus, item 4 (lines 25-31) is triggered, and the replica successfully votes without having received any digests from the other replica. Thus a two-replica CPS is available when one of its replicas is faulty, if the other replica receives all measurements and is up-to-date on the state.

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

Algorithm 5.6: `vote`(r)

```
v ← []; // vector of digests from each replica
$S_{mc}$ ← set of most common digests in v;
$S_{sec}$ ← set of second most common digests in v;
$f_{mc}$ ← count of each element of $S_{mc}$ in v;
$f_{sec}$ ← count of each element of $S_{sec}$ in v;
$f_0$ ← number of empty cells in v;
repeat
    // Receive collection message
    if digest $(S, r^-, l)$ received from replica $j$ and $l = r$ then
        v[$j$] ← $(S, r^-)$;
        update $S_{mc}$, $S_{sec}$, $f_{mc}$, $f_{sec}$, $f_0$ using v;
        // Attempt vote
        if $f_0 = 0$ then
            // All digests received, choose max of $S_{mc}$
            return True, $\max(S_{mc})$;
        else if $|S_{mc}| = 1$ and $f_{mc} > f_{sec} + f_0$ then
            // Only one most common digest, and clearly the plurality
            return True, $S_{mc}[0]$;
        else if $|S_{mc}| = 1$ and $f_{mc} = f_{sec} + f_0$ and $f_{sec} \neq 0$ then
            // $2^{nd}$ most common digests could have equal count
            if $S_{mc}[0] > \max(S_{sec})$ then
                // Most common digest is the largest
                return True, $S_{mc}[0]$;
            end
        else if $|S_{mc}| = 1$ and $f_{mc} = f_{sec} + f_0$ then
            // Other digests could have equal count
            if $S_{mc}[0]$ = full_digest then
                // Most common digest is the largest
                return True, $S_{mc}[0]$;
            end
        end
    end
until timer $3\delta_n$ expires;
return False, NULL; // Return false because vote was unsuccessful
```

5.5 Formal Guarantees

According to Definition 3.5, consistency is defined for two output events a and $\tilde{a}$ as: if $a \equiv \tilde{a}$, then $a.m = \tilde{a}.m$. To prove that Quarts guarantees consistency, we first show that the computations in the same round of controller replicas that use Quarts output the same set of setpoints. This is captured by the notion of *computation consistency* as defined below. Then we show that is sufficient to guarantee the consistent computations property in order for a CPS to ensure consistency.

Definition 5.1 (Computation Consistency)**.** *Computation consistency is said to hold for label* r *for a CPS iff all the setpoints with label* r *for the same PA have the same value.*

Theorem 5.5.1 (Computation Consistency)**.** *A CPS that satisfies the model in Section 5.3 and implements Quarts (Algorithm 5.2) guarantees computation consistency in the presence of any number of delay- or crash-faulty replicas.*

Proof. The proof requires the following lemmas.

Lemma 5.5.1. *If Algorithm 5.6 returns True for a label* r *for controller replicas* C_i *and* C_j*, then the chosen digests* S^i_{chosen} *and* S^j_{chosen} *are equal.*

Proof. For label r, one of the lines 16, 19, 24 or 30 of Algorithm 5.6 would have been triggered on the replicas.
Line 16 The number of empty cells in `v` is zero. In this case, the voter has to choose one of the digests that are most common. If there is only one, it is chosen, otherwise, it chooses the largest among them.
Line 19 There is only one most-common digest, and it will remain the most-common digest no matter what the replicas that have yet to send digests send. It is chosen (as the obvious majority).
Line 24 There is only one most-common digest, and there is at least another digest (second most common), and no matter what the replicas that have yet to send digests send, the second most-common digest will be at most as frequent as the most common. In this case, if the most-common digest is larger than all the second most-common digests, the voter chooses it.
Line 30 The `full_digest` is the only most common one and no other digest can become more common. As it is the largest by default, it can be chosen immediately. □

Let $\mathbf{Z}^i_r$, $\mathbf{Z}^j_r$ be the vector of measurements used for computation by controller C_i, C_j in round r, respectively. Then, we have the following result.

Lemma 5.5.2. $\forall\, r, i, j,\ \mathbf{Z}^i_r = \mathbf{Z}^j_r$

Proof. $\mathbf{Z}_i^r$ is the set of measurements used for computation for label r by a C_i in Algorithm 5.2 line 17.
If controller replicas C_i and C_j compute setpoints for label r, i.e., line 17 of Algorithm 5.2 is triggered, then on both C_i and C_j the function `collect_and_vote` must have returned True (Algorithm 5.3 line 10).
Therefore, the function `vote` must have returned True, for label r.
From Lemma 5.5.1, we know that, $\forall\, r$, if Algorithm 5.6 on controller C_i and C_j returns True, then $S^i_{chosen} = S^j_{chosen}$.
Additionally, as the measurements for label r received by from the same PA by C_i and C_j are identical, $S^i_{chosen} = S^j_{chosen} \implies \mathbf{Z}^i_r = \mathbf{Z}^j_r$. □

Next, we can show that two controller replicas that compute setpoints use the same state labels as shown below.

Lemma 5.5.3. *When replicas C_i and C_j compute setpoints for a label r using state labels r_i^- and r_j^- respectively, $r_i^- = r_j^-$.*

Proof. If controller replicas C_i and C_j compute setpoints for label r, i.e., line 17 of Algorithm 5.2 is triggered, then on both C_i and C_j the function `collect_and_vote` must have returned True (Algorithm 5.3, line 10).
Therefore, the function `vote` must have returned True, for label r.
From Lemma 5.5.1, we know that, $\forall\, r$, if Algorithm 5.6 on controller C_i and C_j returns True, then $S^i_{chosen} = S^j_{chosen}$.
From Section 5.4.3, we see that if two digests are equal, then they have the same state labels.
Therefore, $S^i_{chosen} = S^j_{chosen} \implies r_i^- = r_j^-$. □

Lastly, we can show that the states represented by the same state labels are identical.

Lemma 5.5.4. $\forall\, r, i, j,\ r_i^- = r_j^- \implies \mathbf{H}^i_r = \mathbf{H}^j_r$

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

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

Inductive Step: To show that, for label l, $\forall\ i,\ j\ :\ C_i, C_j$ compute setpoints for label l, $l_i^- = l_j^- \implies \mathbf{H}_l^i = \mathbf{H}_l^j$.
When a replica computes a setpoint, its state is updated by line 18 of Algorithm 5.2. When both C_i and C_j performed a computation of setpoints for label $l-1$, then the label of their state would be updated to $l-1$. Hence, $l_i^- = l_j^- = l-1$.
Additionally, as, the states and correction factors used for this computation are the same, due to the induction hypothesis, and from Lemma 5.5.2, we know that the measurements used for the computation are also the same; the states resulting from the computation are also same (Algorithm 5.2, line 18).
Therefore, we have $l_i^- = l_j^- = l-1$ and $\mathbf{H}_l^i = \mathbf{H}_l^j$
Thus, by principle of induction, $\forall\ r, i, j,\ r_i^- = r_j^- \implies \mathbf{H}_r^i = \mathbf{H}_r^j$ ☐

From Algorithm 5.2, we see that the setpoints depend entirely on the measurements $\mathbf{Z}$ used for computation, the state $\mathbf{H}$, and the state label r^-. Hence, it suffices to show that, any two replicas, C_i and C_j, which compute setpoints for label r by using measurements $\mathbf{Z}_r^i$ and $\mathbf{Z}_r^j$, states $\mathbf{H}_r^i$ and $\mathbf{H}_r^j$, and state labels r_i^- and r_j^-, respectively, will have $\mathbf{Z}_r^i = \mathbf{Z}_r^j$, $\mathbf{H}_r^i = \mathbf{H}_r^j$, and $r_i^- = r_j^-$. This is shown by Lemmas 5.5.2, 5.5.4 and 5.5.3, respectively. ☐

Now, we show that Quarts guarantees the consistency property (Definition 3.5) in a CPS that satisfies the properties mentioned in Section 5.3.

Theorem 5.5.2 (Quarts Consistency)**.** *A CPS that satisfies the model in Section 5.3 and implements Quarts (Algorithm 5.2) guarantees consistency in the presence of any number of delay- or crash-faulty replicas.*

Proof. To prove this result, we will show that for a CPS that uses intentionality clocks to label messages, computation consistency is sufficient to provide consistency in the following lemma.

Lemma 5.5.5. *A CPS that implements intentionality clocks (Algorithms 4.1 and Algorithm 4.3) and guarantees computation consistency also guarantees consistency.*

Proof. Let $a,\ \tilde{a} \in \mathcal{E}^o$ be such that $a \equiv \tilde{a}$. To show that when the CPS implements intentionality clocks and guarantees consistency, $a.m = \tilde{a}.m$
From the strong clock-consistency property of intentionality clocks under the intentional equivalence relation in Theorem 3.6.3, $a \equiv \tilde{a} \implies a.m = \tilde{a}.m$
Then, $a \equiv \tilde{a} \implies a.pa = \tilde{a}.pa$
The events a and $\tilde{a}$ can be either setpoint sending events or measurement sending events

Case 1: a and $\tilde{a}$ are setpoint sending events
Let x and $\tilde{x}$ be the setpoints encapsulated in a and $\tilde{a}$, respectively
Thus, $a.m = x$ and $\tilde{a}.m = \tilde{x}$
Also, $a.l = \tilde{a}.l \implies$ that the setpoints x and x have the same label
Moreover, $a \equiv \tilde{a} \implies a.pa = \tilde{a}.pa$
As the CPS satisfies computation consistency, $x = \tilde{x}$
Therefore, $a.m = a\tilde{.m}$

Case 2: a and $\tilde{a}$ are measurement sending events
Let us assume that a and $\tilde{a}$ are two different measurement sending events such that a occurs before $\tilde{a}$
Let t_0 and t_1 be the time instants at which the PA executes the `implement_setpoint` and `create_measurement` functions to generate event a
Then, from Algorithm 4.3, we know that the value of the intentionality clock at time t_1 is $C_{t_1} = a.l$ and at time t_0 is $C_{t_0} = a.l - 1$
Let t_2 and t_3 be the time instants at which the PA executes the `implement_setpoint` and `create_measurement` functions to generate event $\tilde{a}$
Then, from Algorithm 4.3, we know that the value of the intentionality clock at time t_3 is $C_{t_3} = \tilde{a}.l$ and at time t_2 is $C_{t_2} = \tilde{a}.l - 1$
As a occurs before $\tilde{a}$, we have $t_3 > t_1$
As Algorithm 4.3 is sequential without preemption, we have $t_0 < t_1 < t_2 < t_3$
Moreover, we have $a.l = \tilde{a}.l$
Thus, $C_{t_1} = a.l > a.l - 1 = C_{t_2}$, but $t_1 < t_2$
This is a contradiction as intentionality clocks are never decremented (Algorithm 3.2 lines 13 and 15)
Thus, a and $\tilde{a}$ are the same event. Therefore, $a.m = \tilde{a}.m$ □

In Section 5.3, we assume that the CPS implements intentionality clocks and in Theorem 5.5.1, we prove that such a CPS guarantees computation consistency. Thus, we have that it also guarantees consistency. □

CPSs aim to have minimal latency between the generation of measurements and the 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 time spent in the `collect_and_vote` function (Algorithm 5.2, line 7). It depends on the network delay between the controllers. As described in Section 5.3, we consider a lossy network with a bounded propagation delay δ_n for non-lost packets. For a CPS using such a network, Quarts guarantees bounded latency-overhead for each replica that issues a setpoint.

Theorem 5.5.3 (Quarts Bounded Latency-Overhead)**.** *When a non-faulty replica of a CPS following the model in Section 5.3 and using Quarts (Algorithm 5.2) issues a* *setpoint, its latency overhead is bounded by* $5\delta_n$.

Proof. As mentioned earlier, the latency overhead of a replica due to Quarts is the time spent in the `collect_and_vote` function (Algorithm 5.2 line 7). The `collect_and_vote` function, described in Algorithm 5.3, consists of three functions: `collect_missing_measurements`, `collect_missing_state`, and `vote`. As the statement applies only to non-faulty replicas, we know from Section 5.3, that the operations that do not involve waiting for the network to deliver messages have negligible delay.

The `collect_missing_measurements` (Algorithm 5.4) and `collect_missing_state` (Algorithm 5.5) functions can be merged into one function without requiring parallelism. The resulting function would send a *Quarts_Query* and a *Quarts_Advertisement*, then continue listening to *Quarts_Queries*, *Quarts_Responses*, *Quarts_Advertisements*, and *Quarts_Updates* until the timer of $2\delta_n$ expires. As it is designed to terminate after the timer expires, this collection function results in a bounded latency-overhead of $2\delta_n$.

The `vote` function is also designed to terminate after the timer expires. Its timer, however, is $3\delta_n$ (Algorithm 5.6 line 33). Therefore, the voting phase has a bounded latency-overhead of $3\delta_n$.

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

5.6 Performance Evaluation

We evaluate the consistency (Definitions 3.5) of Quarts and that of existing protocols [99, 161]. We also evaluate other performance metrics of CPSs, namely availability (Definition 5.2), latency (Definition 5.3) and messaging cost. The analytical evaluation of these metrics appears to be mathematically intractable. Therefore, we perform the evaluation by using discrete-event simulations.

5.6.1 Performance Metrics

From Definition 3.5, we obtain a measure of consistency as follows. If consistency holds for a CPS for setpoints with label r, then put $\gamma_r = 1$, else $\gamma_r = 0$. Note that $\gamma_r = 1$, when no setpoints are sent by a CPS with a label r, or when only one of the replicas sends setpoints for label r. The measure of consistency of a CPS is given by $\Gamma = \mathbb{E}[\gamma_r]$.

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

Definition 5.2 (Availability)**.** *Availability is said to hold for a CPS for a setpoint with label r with respect to. a PA A_j, iff A_j receives a setpoint with label r.*

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

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

Definition 5.3 (Latency)**.** *Latency of a CPS for a setpoint with label r is $\delta_r = \min_{j\in[1,g]} I_r^j - \min_{i\in[1,m]} S_r^i$.*

From Definition 5.3, we compute the mean latency of a CPS with an agreement protocol as $\Delta = \mathbb{E}[\delta_r]$. Besides having a low mean-latency, it is important that the delay distribution is light-tailed. So, we also consider the 99^{th} percentile of latency (δ_{p99}) in our evaluation.

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

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

5.6.2 Agreement Protocols

The state-of-the-art agreement protocols, against which we compare the performance of Quarts, are active replication with Fast-Paxos consensus [161] and passive replication with hot or cold standbys [99]. Hereafter, we denote them as AC, PH, and PC, respectively. We denote Quarts as Q.

Protocol AC All replicas receive measurements and perform computations. Before sending setpoints to the actuators, the replicas agree on which replica sends the setpoints for this label (which is equivalent to agreeing on which setpoint is sent), by using a consensus protocol. We choose the widely used Fast Paxos [161] protocol for consensus, as it is optimized for low latency and guarantees consistency. This is an example of an active-replication-based agreement protocol that ensures consistency by agreement on the setpoints, as opposed to agreement on the measurements, as done by Quarts.

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

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

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

5.6.3 Simulation Methodology

We consider a CPS for the control of electric grids that uses a Kalman-filter based controller [176]. In this CPS, the PAs send measurements every $T = 20$ ms. As mentioned in Section 5.3, the network is considered to be probabilistic synchronous [58], in which packets are dropped with a probability p, and are otherwise received within a delay bounded by $\delta_n = 0.5$ ms. We simulate this using a Bernoulli random variable with success probability $1 - p$. The detailed fault model is described below.

Fault Model

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

To simulate the bursty behavior of faults, we incorporate delays in the Gilbert-Elliot model [177] as follows. Each replica could be in one of two states: normal state (N) or crashed state (C), independent of other replicas. As shown in Figure 5.2, the transition probabilities from N to C and C to N are q_C and q_N, respectively. In state N, the replica is faulty (its delay exceeds a threshold τ) with a probability p_d, which simulates delay faults. The computation delay of a replica is drawn from an exponential distribution with a mean μ such that $\mathbb{P}(\text{delay} > \tau) = p_d$, where τ is the threshold for a non-faulty replica computation. The replica is considered unavailable if its delay is such that it fails to send the setpoint before new measurements arrive.

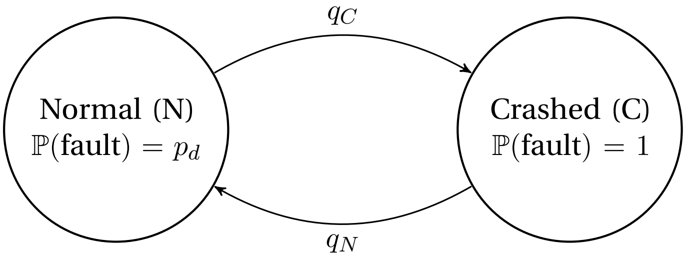

q_C
Normal (N)
$\mathbb{P}(\text{fault}) = p_d$
Crashed (C)
$\mathbb{P}(\text{fault}) = 1$
q_N

Figure 5.2 – Gilbert-Elliot model for simulating delay and crash faults

In state C, the replica is faulty with probability 1, thereby simulating crash faults. The transition from state C to state N signifies a repair after crash faults. The parameters (q_C, q_N, μ) of the model are evaluated from the desired probability of crash faults (θ_c), probability of delay faults (θ_d), MTTR from crash faults (R), and delay threshold (τ) of the CPS as follows. First, we note that the MTTR is the mean time before a replica exits state C. Thus, we have

$$\begin{aligned} R &= 1/q_N \\ \implies q_N &= 1/R \end{aligned} \tag{5.1}$$

Then, to compute θ_c, we evaluate the stationary probabilities of states N and C, π_N and π_C, respectively

$$\begin{aligned} \pi_N &= \frac{q_N}{q_C + q_N} \\ \pi_C &= \frac{q_C}{q_C + q_N} \end{aligned}$$

π_C is the total time spent by a controller replica in state C. As the replica is faulty with a probability 1 in state C, we obtain

$$\begin{aligned} \theta_c &= \pi_C \times 1 \\ \theta_c &= \frac{q_C}{q_C + q_N} \\ \implies q_C &= \frac{\theta_c}{R(1-\theta_c)} \end{aligned} \tag{5.2}$$

Similarly, we obtain p_d using the stationary probability of state N as follows

$$\begin{aligned} p_d &= \frac{\theta_d}{\pi_N} \\ &= \frac{\theta_d}{1-\theta_c} \end{aligned}$$

Moreover, p_d is also the probability that the replica takes more than the delay threshold (τ) to compute a setpoint when its delay follows an exponential distribution with mean $1/\mu$. Thus, we also have

$$\begin{aligned} p_d &= \mathbb{P}(\text{ Exp}[1/\mu] > \tau\ ) \\ &= e^{-\tau/\mu} \\ \implies \mu &= -\frac{\ln(p_d)}{\tau} \end{aligned} \tag{5.3}$$

The parameters θ_d, θ_c and R are varied across different scenarios and τ is taken as constant with value $8\ ms$ for the CPS. When the computation delay of the replica is greater than the length of the control round T, the replica is considered unavailable for the setpoint of the corresponding label. Lastly, in protocols PC and PH, the timeout after which a primary replica is detected as faulty is taken as τ, so as to allow the primary replica sufficient time to send heartbeats to the standbys. Moreover, it leaves the standys with 12 ms, i.e., 12 RTTs, to elect a new primary replica.

Quality of the Estimation

We use the relative accuracy (β) of an estimate ($\hat{x}$) of a probability (either unavailability of inconsistency) at a confidence level $1-\alpha$ as the measure of the quality of the estimate. Specifically, a $1-\alpha$ confidence interval of $\hat{x}$ has a length of η times the standard deviation of $\hat{x}$, where η is obtained from the relation $N_{0,1}(\eta) = 1-\alpha/2$. Then, c is given by $c = \frac{\sqrt{\hat{x}(1-\hat{x})/R}}{\hat{x}} \approx \frac{1}{\sqrt{R\hat{x}}}$, where R is the number of rounds. Therefore, we obtain $\beta = \frac{\eta}{\sqrt{R\hat{x}}}$. 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 $[0.95\hat{x}, 1.05\hat{x}]$.

5.6.4 Results

We simulated the 4 protocols (Q, AC, PC, PH) for several scenarios with 2 values of m (10 and 100), 5 values for each of g (1 through 5), 3 different fault models, and 10 values of p (between 10^{-4} and 0.05). Our simulation platform was a high-throughput cluster with 278 nodes; and the simulation campaign lasted a total of 10 days. We consider the following to be the nominal parameters of the CPS: $g = 2, m = 10,\ p = 10^{-3},\ \theta_c = 10^{-4}$, $\theta_d = 10^{-3},\ R = 1$ s, $\delta_n =\ 0.5$ ms, $\tau =\ 8$ ms, and $T\ =\ 20$ ms. Henceforth, unless specified otherwise, this is the set of parameters used.

Besides presenting results as a function of varying p and g, we also present results from five characteristic scenarios. Scenario #1 is the nominal scenario as mentioned above. Scenario #2 has $m = 100$ instead of $m = 10$ and shows the affect of number of PAs. Scenario #3 has a more relaxed fault model than the nominal scenario. Scenarios #4 and #5 have only crash faults and no delay faults. Moreover, Scenario #5 has $g = 3$ and shows the performance with 3 replicas in the absence of delay faults.

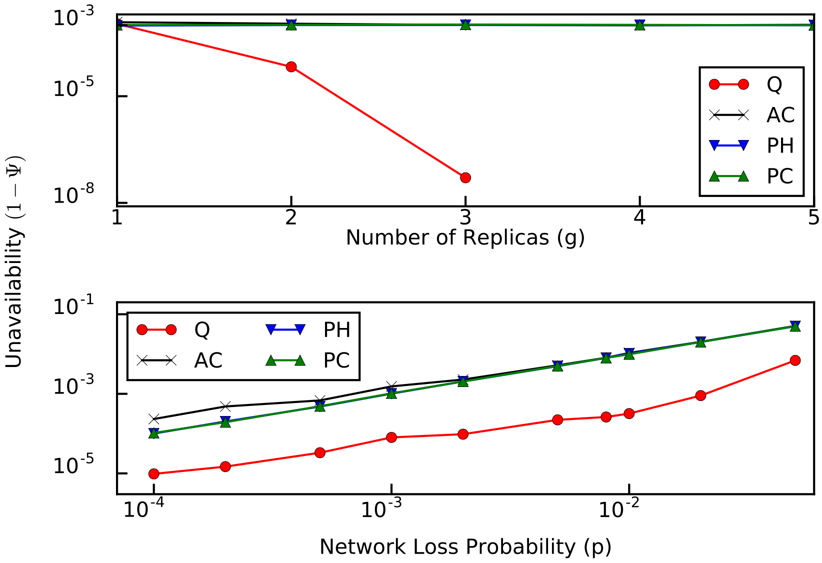


Figure 5.3 – Unavailability with varying g and varying p. Unavailability of Quarts (Q) with more than 3 replica is less than 4×10^{-10}

Figure 5.3 shows the unavailability $(1 - \Psi)$ for varying g and varying p. Tables 5.1, 5.2, 5.4 and 5.3 show the detailed simulation results for the chosen scenarios.

Finding 5.1. *Quarts provides availability higher than that of AC, PC and PH, while maintaining 100% consistency.*

	Unavailability $(1-\Psi)$			
Scenario: $(m,\ g,\ \theta_c,\ \theta_d)$	Q	AC	PH	PC
#1: $(10,\ 2,\ 10^{-4},\ 10^{-3})$	9.12×10^{-5}	1.24×10^{-3}	9.87×10^{-4}	1.02×10^{-3}
#2: $(100,\ 2,\ 10^{-4},\ 10^{-3})$	1.46×10^{-4}	1.19×10^{-3}	9.76×10^{-4}	1.03×10^{-3}
#3: $(10,\ 2,\ 10^{-5},\ 10^{-4})$	1.02×10^{-5}	9.98×10^{-4}	1.01×10^{-3}	9.96×10^{-4}
#4: $(10,\ 2,\ 10^{-4}, 0)$	8.14×10^{-5}	1.37×10^{-3}	1.01×10^{-3}	1.02×10^{-3}
#5: $(10,\ 3,\ 10^{-4}, 0)$	2.25×10^{-8}	1.01×10^{-3}	1.01×10^{-3}	9.92×10^{-4}

Table 5.1 – Unavailability results for the chosen scenarios

We simulate the protocols for $g = [1, 5]$. The first plot of Figure 5.3 shows that the unavailability (1 - Ψ) of Quarts is more than an order of magnitude lower than that of other protocols with 2 replicas, and 4 orders of magnitude lower with 3 replicas. Additionally, for more than 3 replicas, Quarts showed no unavailability in 10^{10} runs ($\sim$ 3 days of simulation). Simulating such extremely rare events requires more sophisticated techniques such as Importance Sampling and Palm Calculus [178] and is left for future work.

	Inconsistency $(1-\Gamma)$	
Scenario: $(m,\ g,\ \theta_c,\ \theta_d)$	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: $(10,\ 2,\ 10^{-4}, 0)$	$(0, 4\times10^{-10}]^*$	$(0, 4\times10^{-10}]^*$
#5: $(10,\ 3,\ 10^{-4}, 0)$	$(0, 4\times10^{-10}]^*$	$(0, 4\times10^{-10}]^*$

Table 5.2 – Inconsistency results for the chosen scenarios. Inconsistency of A and AC is zero. * No inconsistency was observed in 10^{10} runs

In order to put the availability improvement with Quarts into perspective, we analyze the MTTF [72] of the CPS, where a failure is defined as the inability of controllers to reach agreement hence not issue setpoints. For the CPS under study that sends setpoints every 20 ms, for $g = 1$, an availability of 0.9987 translates to an MTTF of 18.2 s. Using two replicas, the protocols AC, PC and PH can increase the MTTF to 18.4 s, 19.6 s and 19.9 s, respectively. By comparison, the MTTF with Quarts is 5 minutes. Furthermore, the MTTF with 3 replicas for protocols AC, PC, PH and Q is 19.7 s, 19.2 s, 19.9 s, and 4.54 days. As seen above, the existing protocols show marginal improvement in availability with each additional replica, whereas Quarts enjoys a significant increase.

The second plot of Figure 5.3 confirms Finding 5.1 for different values of p. We see that, even at extremely high-values of $p = 0.02$, Quarts has $\Psi = 0.9991$; whereas this value drops down to 0.98 for other protocols. This finding is further re-affirmed by the results of the scenarios shown in Table 5.1.

The availability improvement of Quarts comes without any consistency penalty. We find that the consistency of Quarts and AC is 1. However, the consistency guarantee of AC comes at the cost of low availability, compared to Quarts. In contrast, PC and PH have an inconsistency between 10^{-5} and 10^{-3} in the presence of delay faults, for the scenarios considered, as seen in Table 5.2. In the absence of delay faults, the probability of inconsistency for PC and PH could not be captured by our simulations in 10^{10} runs hence can be considered small. For these scenarios, however, their availability is still lower than that of Quarts.

	Latency in ms (Δ, δ_{p99})			
Scenario: $(m,\ g,\ \theta_c,\ \theta_d)$	Q	AC	PH	PC
#1: $(10,\ 2,\ 10^{-4},\ 10^{-3})$	(0.96, 3.08)	(4.28, 8.78)	(1.40, 5.59)	(1.41, 5.59)
#2: $(100,\ 2,\ 10^{-4},\ 10^{-3})$	(0.98, 3.11)	(4.27, 8.78)	(1.40, 5.58)	(1.41, 5.59)
#3: $(10,\ 2,\ 10^{-5},\ 10^{-4})$	(0.82, 2.42)	(3.91, 8.09)	(1.12, 4.26)	(1.12, 4.26)
#4: $(10,\ 2,\ 10^{-4}, 0)$	(0.39, 0.78)	(3.52, 4.18)	(0.25, 0.5)	(0.25, 0.5)
#5: $(10,\ 3,\ 10^{-4}, 0)$	(0.50, 0.84)	(3.50, 4.15)	(0.25, 0.49)	(0.25, 0.5)

Table 5.3 – Mean and 99^{th} percentile of latency for select scenarios

Finding 5.2. *Quarts has a lower average-latency and tail-latency than other consistency-guaranteeing protocols.*

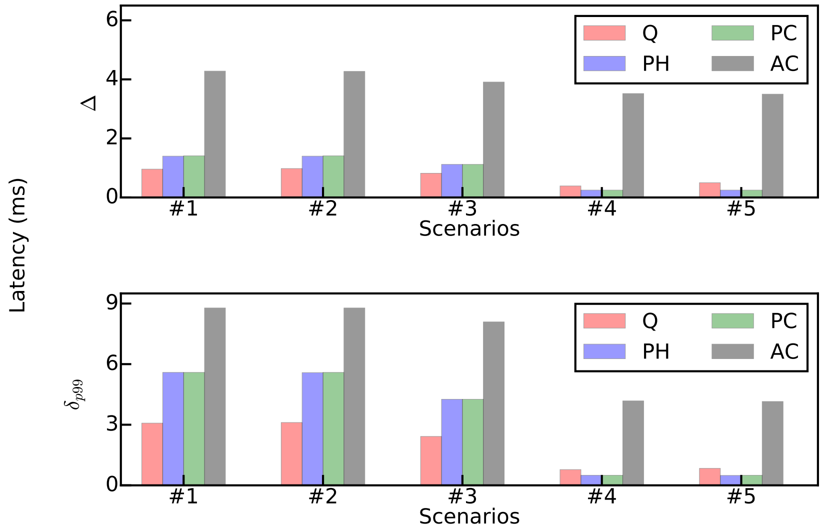


Figure 5.4 – Mean and 99^{th} percentile of latency in different scenarios

Table 5.3 and Figure 5.4 show that the mean latency of Quarts is less than that of

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

We also see in Table 5.3 and Figure 5.4 that the tail-latency of Quarts is lower than that of other protocols in the presence of delay faults, and comparable to that of PC and PH in the absence of delay faults. Such low tail-latency is due to the bounded latency-overhead of Quarts as shown by Theorem 5.5.3.

Finding 5.3. *Guaranteeing consistency comes at the marginal expense of a higher messaging cost.*

	Messaging cost in messages/label $(\Omega,\ \omega_{p99})$			
Scenario: $(m,\ g,\ \theta_c,\ \theta_d)$	Q	AC	PH	PC
#1: $(10,\ 2,\ 10^{-4},\ 10^{-3})$	(4.04, 6)	(5.17, 6)	(3.87, 8)	(3.00, 3)
#2: $(100,\ 2,\ 10^{-4},\ 10^{-3})$	(4.38, 6)	(5.17, 6)	(3.87, 8)	(3.00, 3)
#3: $(10,\ 2,\ 10^{-5},\ 10^{-4})$	(4.04, 6)	(5.10, 6)	(3.74, 7)	(3.00, 3)
#4: $(10,\ 2,\ 10^{-4}, 0)$	(4.04, 6)	(5.02, 5)	(3.50, 4)	(3.01, 3)
#5: $(10,\ 3,\ 10^{-4}, 0)$	(9.18, 15)	(9.02, 9)	(6.67, 8)	(5.01, 5)

Table 5.4 – Mean and 99th percentile of messaging cost for select scenarios

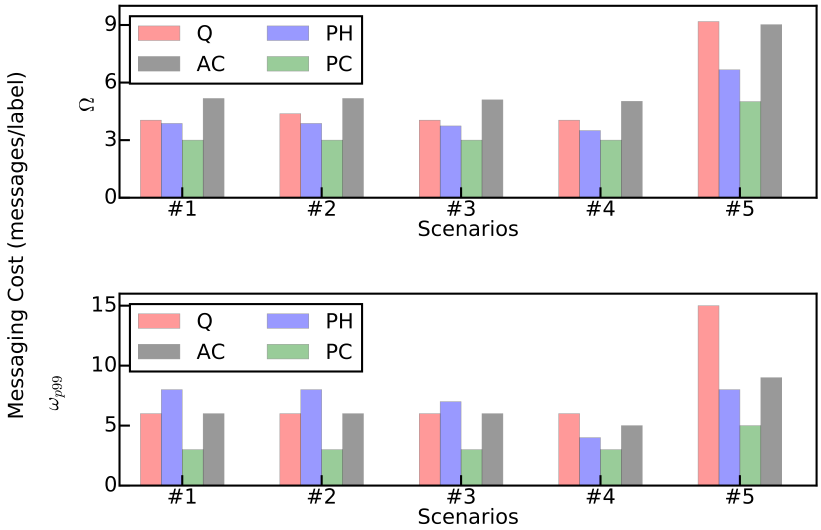


Figure 5.5 – Mean and 99^{th} percentile of messaging cost in different scenarios

The mean messaging cost of Quarts and AC is marginally higher than that of PC and PH and increases with the number of replicas. This is due to the collection phase in Quarts and the consensus employed by AC, for every setpoint. Such an exchange of messages is the price to pay for achieving consistency. Table 5.4 and Figure 5.5 show the mean and 99^{th} percentile of the messaging cost. We see that the messaging cost of Quarts is comparable to that of the other consistency-guaranteeing protocol (AC), thereby reaffirming Finding 5.3.

5.7 Quarts+: Incorporating Asynchronous Sensors

Quarts provides consistency for CPS with one or more controller replicas, one or more PAs and no sensors that directly communicate with the controllers. However, CPSs often use *asynchronous sensors* that asynchronously record the state of the process and send it to the controller in the form of out-of-band advertisements. The measurements sent by PAs are considered in-band because they are sent in response to the setpoints issued by controller replicas.

Figure 5.6 shows the architecture of such a CPS. Examples of asynchronous sensors include PMUs (in the COMMELEC CPS) that periodically sense the voltage and current at different points in the grid [53], temperature and pressure sensors in manufacturing plants, etc. As the controller uses these out-of-band advertisements in the computation of setpoints, in addition to using measurements from PAs, Quarts' agreement on measurements alone no longer ensures consistency. Consequently, in order to continue providing consistency with the addition of asynchronous sensors, the replicas also need to agree on advertisements. To this end, we propose Quarts+: it builds upon Quarts, to provide the same consistency guarantees for a wider range of CPSs.

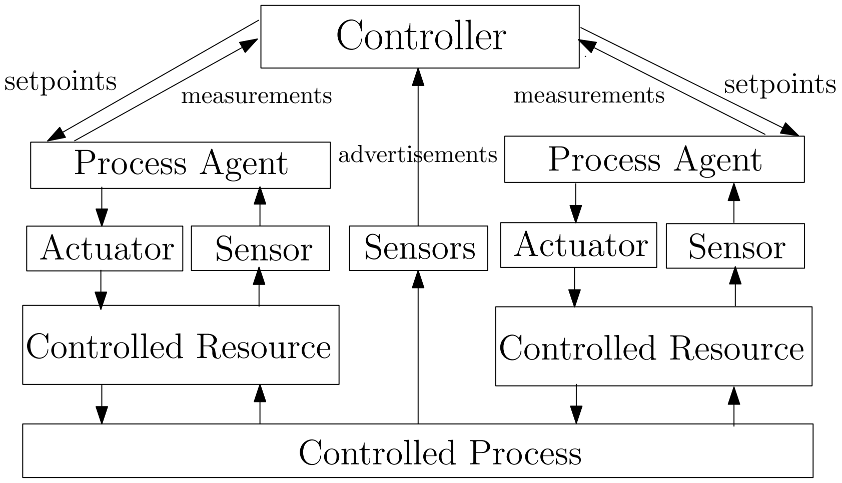


Figure 5.6 – Architecture of a CPS with PAs and asynchronous sensors that directly communicate with the controller

The central issue is that Quarts requires that all inputs share the logical label, such as Lamport clocks [85] or intentionality clocks. Logical clocks require a two-way communication between agents. As asynchronous sensors do not communicate with the PAs, nor does the controller send messages to asynchronous sensors, they cannot use logical clocks. Instead asynchronous sensors use timestamped advertisements. Hence, Quarts cannot be directly applied to CPSs that include them.

Consider an example in which two controller replicas, C_1 and C_2, reach the same round at different time instances and receive advertisements, a_1 and a_2, from the same asynchronous sensor, respectively. In order for the controllers to agree on which advertisement to use, they cannot use Quarts hence must resort to general consensus. In other words, in the absence of unified labeling schemes for all messages, namely measurements, setpoints and advertisements, the problem is to reconcile differences between the labeling scheme used by controllers and PAs, and the one used by asynchronous sensors. It is impossible to guarantee such a reconciliation within a bounded-delay [88]. In Section 5.7.1, we further detail this problem.

We observe that for asynchronous sensors with a known minimal inter-arrival time of the events (Λ), consensus can be circumvented. We specify this and other requirements on the CPS in Section 5.7.2. Using these properties, we extend Quarts to guarantee consistency within bounded-delay for CPSs with both PAs and asynchronous sensors. The new design (Quarts+) adds a bounded latency-overhead of 1 RTT to the Quarts protocol. Moreover, for a vast majority of the cycles, the added latency-overhead is 0, thereby enabling Quarts+ to benefit from the low latency and high availability enjoyed by Quarts. We present the design of Quarts+ in Section 5.7.3, and prove its consistency and bounded-latency guarantees in Section 5.7.4.

Lastly, in Section 5.7.5, we use discrete-event simulation to compare the availability, consistency, and latency of Quarts+ with Quarts and other state-of-the-art agreement schemes that were also used for evaluation of Quarts in Section 5.6.

5.7.1 Challenge with Asynchronous Sensors

The design of a deterministic voting function and the existence of a `full_digest` hinges on the ability of each controller replica to form a digest. As noted earlier, Quarts is able to do so because in each round, each PA sends exactly one measurement. Moreover, measurements in the same round are assigned the same label by using intentionality clocks. In order to create a digest with ones and zeros, a controller can simply answer the question “Did I receive a measurement from PA i in round l?”. However, a controller replica cannot answer the same question for advertisements from asynchronous sensors.

This is because asynchronous sensors do not follow the same labeling scheme as the PAs, nor do they follow a consistent labeling scheme among themselves. For example, an event-driven sensor timestamps the advertisements when the event occurs, independently from the advertisements on other sensors. Additionally, asynchronous sensors do not follow the same rounds as the PAs. For instance, in COMMELEC [4], the computation round takes about 100 ms on average, whereas the asynchronous sensors send advertisements periodically every 20 ms. Thus, on average, there are five advertisements per one measurement. Furthermore, consistently choosing one of these five advertisements in a round is another consensus problem, faced with the same impossibility of bounded-time convergence limitations as mentioned earlier [88].

Therefore, with the introduction of asynchronous sensors, Quarts can no longer form a digest and cannot vote on a digest to provide a consistent set of inputs (advertisements and measurements) to be used for computation by the controller replicas. This is the challenge in realizing low-latency agreement in the presence of asynchronous sensors. To address this challenge, we require certain properties from the CPS, as mentioned below.

5.7.2 CPS Requirements for Quarts+

In addition to the properties of a CPS required by Quarts, mentioned in Section 5.3, we also assume that all agents (controllers, PAs and sensors) are time-synchronized either by using GPS-based technique [179] or network protocols such as NTP [165] or PTP [164].

The asynchronous sensors regularly measure the state of the process and send out-of-band advertisements to the controllers. These advertisements are timestamped at the time of measuring. We assume a lower bound on the minimum interval, between two measuring events at a sensor i, is known and given by Λ_i; and we assume that the granularity of the timestamps is such that no two advertisements from the same sensor have the same timestamp. In many cases, the sensors are quasi-periodic with a small jitter [40]. In cases where the sensor is strictly periodic, Λ_i is equal to its period. We also assume that measurements from PAs are also timestamped at the time of measuring. As a result, if a PA sends the same measurement to two different controllers, then both the copies have the same timestamp.

Note that, based on these assumptions, the losses of advertisements caused by network losses or sensor delays are part of the model. Retransmissions are also permitted by the model. However, a sensor that hangs then wakes up and sends two consecutive different advertisements with timestamps less than Λ_i apart is not considered.

With the addition of asynchronous sensors, the `compute` function of the controller, described in Chapter 3 and used for Quarts in Algorithm 5.2, is adapted to include advertisements, as follows. Each computation takes as input at most one measurement from each PA and at most one advertisement from each sensor, then it produces as output exactly one setpoint for each PA. Note that the difference is that advertisements were not used in the `compute` function earlier.

5.7.3 Quarts+ Design

CPS controllers that use Quarts+ first perform Quarts to agree on the measurements from the PAs. Additionally, in order to overcome the problems with Quarts and asynchronous sensors discussed in Section 5.7.1, Quarts+ adds the following mechanisms so that controller replicas can create a digest for voting, from their advertisements and measurements. Note that this digest is functionally the same as the one used in Quarts but contains information about advertisements, in addition to the information about measurements contained in the one used by Quarts.

The only change required at the sensors and PAs is that they timestamp their outgoing measurements and advertisements, as described in Section 5.7.2. The timestamps are communicated to the controller replicas along with the measurements and advertisements. Next, the controller replicas perform agreement on measurements by using Quarts, as described in Section 5.4. As a result, a replica that successfully completes the voting phase for a round will produce a chosen digest **D** of measurements and their timestamps. Then, the successful replicas use a deterministic function ($\mathcal{F}$) to obtain a timestamp T_0 from the chosen digest **D** and the timestamps of the measurements therein.

Next, each controller replica uses T_0 and Λ_i to choose at most one advertisement from every asynchronous sensor i. If two controllers choose an advertisement from a sensor, then the mechanism guarantees that they choose the same advertisement. Then, to form a digest, the controllers use both the measurements voted upon by Quarts and the locally chosen advertisements. Each controller exchanges its digest with other controllers, and they all vote on the input vectors to choose the set of inputs (both measurements and advertisements) to be used in the computation of setpoints in this round.

Quarts+ is designed such that successful choosing of advertisements from all sensors results in a `full_digest`. Therefore, similar to Quarts, in a CPS with two controller replicas, if a controller replica has the `full_digest` (of both measurements and advertisements), it skips both the collection and voting phases and incurs zero latency-overhead. The resulting model of a controller with the Quarts+ agreement mechanism is described in Algorithm 5.7. The parts in red represent the modifications made to the controller by Quarts+.

Note that, at line 15, we use the function `quarts` that is a wrapper around the `collect_and_vote` function described in Algorithm 5.3. Similar to `collect_and_vote`, the function `quarts` receives as input the vector of received measurements **Z** and the round number r. Then, if the voting was successful, it performs the collection phase and voting phase of Quarts and returns success_quarts, along with the chosen digest and the vector of chosen measurements.

Algorithm 5.7: Controller model with Quarts+ (parts in red)

```
1  $r \leftarrow 0$;              // highest label of measurements received
2  $\mathbf{Z} \leftarrow [\,]$;  // vector of measurements with label $r$
3  $\mathbf{Y} \leftarrow \emptyset$;  // list of timestamped advertisements
4  $\mathbf{V} \leftarrow [\,]$;  // vector of inputs to be used for computation
5  $\mathbf{\Lambda} \leftarrow [\Lambda_1, ..., \Lambda_n]$; // lower bounds of inter-arrival time of sensors
6
7  repeat
8      $\mathbf{Z}, r \leftarrow$ aggregate_received_measurements($r$);
9      $\mathbf{Y} \leftarrow$ aggregate_received_advertisements();
10 forever;
11
12 repeat
13     success $\leftarrow$ False;
14     if $r > r^-$ then
15         success_quarts, $\mathbf{D}$, $\mathbf{Z} \leftarrow$ quarts($\mathbf{Z}, r$);
16         if success_quarts then
17             $T_0 \leftarrow \mathcal{F}(\mathbf{D}, \mathbf{Z})$;
18             success, $\mathbf{V} \leftarrow$ agree_on_advertisements($\mathbf{D}, \mathbf{Y}, r, T_0, \mathbf{\Lambda}$);
19         end
20     end
21     decision $\leftarrow$ ready_to_compute($\mathbf{V}, r$);
22     if success and decision then
23         $\mathbf{X} \leftarrow$ compute($\mathbf{V}$);
24         issue($\mathbf{X}, r$);
25     end
26 forever;
```

In lines 8 and 9, the controller collects measurements and advertisements, respectively. Lines 15-25 constitute the computation performed by a controller. A new round of computation begins when the controller receives measurements with a label higher than what it has seen before. This triggers Quarts in line 15, which is followed by the mechanism introduced by Quarts+: `agree_on_advertisements` in line 18. If the agreement is successful, the chosen vector of inputs $\mathbf{V}$ is used for the actual computation of setpoints in line 23.

Obtaining T_0 from Measurements

The `collect_and_vote` function of Quarts returns a unique digest $\mathbf{D}$ for a label r, within a bounded delay $5\delta_n$. If a replica is successful in choosing $\mathbf{D}$, then it has chosen the same subset of measurements as all other replicas in that round. In line 17 of Algorithm

5.7, each replica then uses a predefined deterministic function $\mathcal{F}$, such as $\max$ or $\min$, to obtain the timestamp T_0 from the set of measurements in the chosen digest $\mathbf{D}$. For instance, if $\mathcal{F} = \max$ and $\mathbf{D} = 10110$ for label $r = 5$, then the T_0 corresponding to label 5 is $\max(t_1^5, t_3^5, t_4^5)$, where t_i^r is the timestamp of the measurement from PA i in round r. As the function $\mathcal{F}$ is same on all replicas, and so is the chosen digest $\mathbf{D}$, this approach guarantees that all the replicas that choose T_0 in a particular round, choose the same T_0, without additional latency overhead.

Obtaining Consistent Advertisements from T_0

After T_0 is obtained, line 18 of Algorithm 5.7 is triggered, which in turn calls Algorithm 5.8. The first part of Algorithm 5.8 (lines 2-6) creates a digest for advertisements. Each replica locally decides if it has a particular advertisement from each sensor. To this end, in line 3, each replica uses the function `has_recent_advertisement` for each of the asynchronous sensors.

Algorithm 5.8: `agree_on_advertisements`$(\mathbf{D}, \mathbf{Y}, r, T_0, \mathbf{\Lambda})$

```
1  M[] ← 0;                                   // digest of advertisements
2  for each asynchronous sensor $i$ do
3      if has_recent_advertisement($\mathbf{Y}, T_0, \Lambda_i$) then
4          $\mathbf{M}[i] \leftarrow 1$;
5      end
6  end
7
8  Send M to all replicas;
9  Vote on received digests to choose M′ using Algorithm 5.6;
10
11 if M′ is a subset of M then
12     V ← measurements(D);
13     V.append(advertisements(M′);
14     return true, V;                        // success
15 else
16     return false;                          // failure
17 end
```

The function `has_recent_advertisement` is as described in Algorithm 5.9. First, it sub-samples, the advertisements from some sensor i from the list of all advertisements $\mathbf{Y}$, then it stores them in the list $\mathbf{Y}_i$. Then, to check for the two conditions C_1 and C_2, the controller replica uses T_0, along with the lower bound on the inter-arrival time between advertisements from that sensor (Λ_i). Condition C_1 checks if the controller replica has received an advertisement from that sensor with timestamp in the interval $[T_0, T_0 + \Lambda_i)$. Then, in condition C_2, the controller replica checks the same in the interval $[T_0, T' + 2\Lambda_i)$, where T' is the timestamp of the last received advertisement from that sensor before T_0.

Algorithm 5.9: `has_recent_advertisement(`$\mathbf{Y}, T_0, \Lambda_i$`)`

```
$\mathbf{Y}_i \leftarrow$ all advertisement from sensor $i$;
$C_1$: $\exists$ advertisement $y \in \mathbf{Y}_i$ such that $y.t \geq T_0$ and $y.t < T_0 + \Lambda_i$;
$T' \leftarrow$ highest timestamp of advertisement in $\mathbf{Y}_i$ less than $T_0$;
$C_2$: $\exists$ advertisement $y \in \mathbf{Y}_i$ such that $y.t \geq T_0$ and $y.t < T' + 2\Lambda_i$;
if $C_1$ or $C_2$ holds then
    return True;
else
    return False;
end
```

It is guaranteed that at most one advertisement lies in the union of the two intervals $[T_0, T_0 + \Lambda_i)$ and $[T_0, T' + 2\Lambda_i)$, considered by conditions C_1 and C_2, respectively. Hence, if either condition holds, the replica can choose the advertisement with a timestamp in this interval, and any replica that does so will choose the same advertisement. This is due to the lower-bound on the inter-arrival time, Λ_i, and is the subject of Theorem 5.7.1.

In other words, no two controller replicas choose a different advertisement for the same sensor in the same round. In this way, by restricting the choice of advertisement from each sensor in the same round, Quarts+ enables controller replicas to answer the question that Quarts cannot answer in the presence of asynchronous sensors in Section 5.7.1, *i.e.,* “Did I receive an advertisement from sensor i in round l?”. Consequently, the controller replicas can now form a digest for advertisements $\mathbf{M}$; this digest is then voted upon by using the function `vote` used in Quarts in Algorithm 5.6.

Upon successful voting, in lines 11-15 of Algorithm 5.8, the controller uses the chosen digest of measurements ($\mathbf{D}$) and the chosen digest of advertisements ($\mathbf{M}$) to choose from the corresponding measurements and advertisements. These measurements and advertisements are added to the input vector for computation $\mathbf{V}$ and sent to the `ready_to_compute` function of the controller at line 21 in Algorithm 5.7, thus marking the end of the Quarts+ mechanism. Note that from line 22 of Algorithm 5.7, the controller can compute and issue setpoints in a round, but only if Quarts+ completes successfully in that round. In this way, the availability of the controller is traded-off for a guaranteed consistency. In Section 5.7.5., we evaluate the availability performance of Quarts+ and compare it with that of other agreement mechanisms.

5.7.4 Formal Guarantees

Quarts+ guarantees consistency and it trades-off availability in order to maintain a bounded latency-overhead, thus satisfying the low-latency requirement of real-time

CPSs. In this section, we formally prove the consistency and bounded latency-overhead guarantees of Quarts+ for CPSs that follow the system model described in Section 5.7.2.

Theorem 5.7.1 (Quarts+ Consistency)**.** *A CPS that satisfies the model in Section 5.7.2 and implements Quarts+ (Algorithm 5.7) guarantees consistency in the presence of any number of delay- or crash-faulty controller replicas, and of any number of asynchronous sensors.*

Proof. As a result of Lemma 5.5.5, it is sufficient that computation consistency holds in order for consistency to be guaranteed
From Algorithm 5.7, we see that the computed setpoints depend entirely on the input vector $\mathbf{V}$ of measurements and advertisements (line 15)
We also see that the decision to compute depends on the boolean success returned by the function `agree_on_advertisements` (line 12) and the function `ready_to_compute` (line 13)
However, as the `ready_to_compute` depends only on $\mathbf{V}$ and r, it suffices to show that, for every round r, replicas that receive a success flag set to true have the same $\mathbf{V}$

First, in line 10, `collect_and_vote` returns the voted upon digest $\mathbf{D}$ and the chosen vector of measurements $\mathbf{Z}$
From Theorem 5.5.1, $\mathbf{D}$ and $\mathbf{Z}$ are guaranteed to be the same across all replicas that successfully voted
Else, from Algorithm 5.7 lines 16 and 13, we see that the controller will not compute in this round

Second, in line 11, $\mathcal{F}(\mathbf{D}, \mathbf{Z})$ is called
As this function is defined to be deterministic, and since $\mathbf{D}$ and $\mathbf{Z}$ are the same across all replicas, the returned T_0 is also the same across all replicas
Therefore, in line 12, we pass to `agree_on_advertisements` the same $\mathbf{D}, r, T_0$ and $\mathbf{\Lambda}$, and a possibly different $\mathbf{Y}$.

We now look at Algorithm 5.8. In line 9, we perform the deterministic voting function presented in Algorithm 5.6, and in lines 11-17, we return success only if the voted-upon digest of measurements and advertisements is a subset of the present inputs at the controller replica.
These two deterministic functions guarantee that $\mathbf{V}$ would be the same across all replicas, as long as the digests are formed correctly, i.e., that a replica sets its i^{th} bit to 1 if and only if it has the i^{th} measurement corresponding to a given round.
However, as advertisements do not belong to a round, then we must ensure that if two replicas set their i^{th} bit to 1 in round r, then both replicas have the same advertisement from sensor i.

This is achieved by the `has_recent_advertisement` function that takes as input $\mathbf{Y}, T_0$, and Λ_i; T_0, and Λ_i begin the same across replicas.

This function performs the following operation (Algorithm 5.9).
Let T' be the highest timestamp less than T_0 of a measurement in $\mathbf{Y}$ received from sensor i (lines 1, 3).
It returns true if and only if there exists a measurement from sensor i with timestamp t such that $t \in [T_0, \max(T_0 + \Lambda_i, T' + 2\Lambda_i))$.
Given that the interval is composed of deterministic linear operations on variables of common value across replicas, the problem is reduced to showing that at most one measurement exists with a timestamp in this interval.

We will prove this by contradiction.
Let $t_1, t_2 \in \mathbf{I} = [T_0, \max(T_0 + \Lambda_i, T' + 2\Lambda_i))$, such that $t_1 < t_2$.
We have, $t_1 \geq T_0$ and $t_2 \geq t_1 + \Lambda_i$.
Thus, $t_2 \geq T_0 + \Lambda_i$.
Moreover, as t_1 is the first arrival after $T', t_1 \geq T' + \Lambda_i$.
Thus, $t_2 \geq T' + 2 * \Lambda_i$ Hence, $t_2 \geq \max(T_0 + \Lambda_i, T' + 2 * \Lambda_i)$ □

Next, we prove the bounded latency-overhead of Quarts+.

Theorem 5.7.2 (Bounded Latency-Overhead)**.** *When a non-faulty replica of a CPS following the model in Section 5.7.2 and using Quarts+ (Algorithm 5.7) issues a setpoint, its latency overhead is bounded by* $6\delta_n$.

Proof. From 5.5.3,we have that when a non-faulty replica of a CPS using Quarts issues a setpoint, its latency overhead is bounded by $5\delta_n$.

Quarts+ first consists of calling Quarts, thus incurring an upper bound of $5\delta_n$ latency overhead.
Then, Quarts+ performs local computations that require negligible computation time (in the order of nano seconds), compared to network transmission (in the milliseconds).
Finally, Quarts+ performs an extra round of voting on the measurements digests (Algorithm 2 line 9).
This terminates after δ_n as described in the Algorithm 5.6.

Hence, the latency overhead is bounded by $5\delta_n + \delta_n = 6\delta_n$. □

5.7.5 Performance Evaluation

We use discrete-event simulation to evaluate the availability (Definition 5.2), latency (Definition 5.3) of Quarts+ (referred to as Q+). We compare these results with those of Quarts (referred to as Q), and other agreement protocols used in Section 5.6, namely active replication using Fast Paxos [161] consensus algorithm (referred to as AC) and passive replication using cold standbys (referred to as PC). We also show that Quarts has non-zero inconsistency (Definition 3.5), thereby underpinning the need for Quarts+. We do not report on the messaging cost of Quarts+, as this is similar to that of Quarts.

We use the same fault-model as described in Section 5.6.3, with the following parameters: probability of delay faults θ_d, probability of crash faults θ_c, and the MTTR from crash faults R.

Recall from Section 5.6.3 that the factors of the simulation that depend on the CPS are (i) the number of replica g, (ii) the number of PAs m, (iii) the communication loss probability p, (iv) the upper-bound on one-way network latency δ_n, (v) threshold for non-faulty computation τ, and (vi) the cycle time of the CPS T. In addition to these, we also have the number of asynchronous sensors n and the inter-arrival time between advertisements received from the same asynchronous sensor modeled as an exponential distribution with mean $1/\mu$ shifted by Λ. The probability density function of the inter-arrival time between two consecutive measurements from the same asynchronous sensor is given by:

$$f(x;\mu,\Lambda) = \begin{cases} \frac{1}{\mu} e^{-\frac{(x-\Lambda)}{\mu}} & , x \geq \Lambda \\ 0 & , \text{otherwise} \end{cases}$$

We use the COMMELEC CPS for real-time control of electric grids [4] as it consists of both PAs and asynchronous sensors. In COMMELEC, the cycle time T is $100\,ms$ and the asynchronous sensors are PMUs that send measurements at least 20 ms apart, thus $\Lambda = 20\,ms$.

We varied the aforementioned parameters with $n = \{5, 25\}$, $m = \{5, 10, 50, 100\}$, $g = \{2, 3\}$, $q = \{10^{-4}, 10^{-3}\}$, $\theta_d = \{10^{-4}, 10^{-3}, 0\}$, $\theta_c = \{10^{-5}, 10^{-4}\}$, $\mu = \{21, 22, 25, 30\}$ and four protocols. This results in a total of 3000+ simulations that were performed on a high-throughput cluster with 278 nodes, each simulation running either until the 95% confidence interval of the estimated value has a half width, from the central value, less than 5% or a maximum of 10^{10} runs. Hence, all our results can be interpreted with the 95% confidence interval as $[0.95\hat{x}, 1.05\hat{x}]$, where $\hat{x}$ is the reported estimate. Due to space restrictions, we show results from a few representative scenarios. We use, unless otherwise specified, $n = 5$, $m = 5$, $g = 2$, $T = 100$ ms, $\mu_i = 21$ ms, $\Lambda_i = 20$ ms, $p = 10^{-3}$, $\delta_n = 0.5$ ms, $\theta_d = 10^{-3}$, $\theta_c = 10^{-4}$, $R = 5$ s.

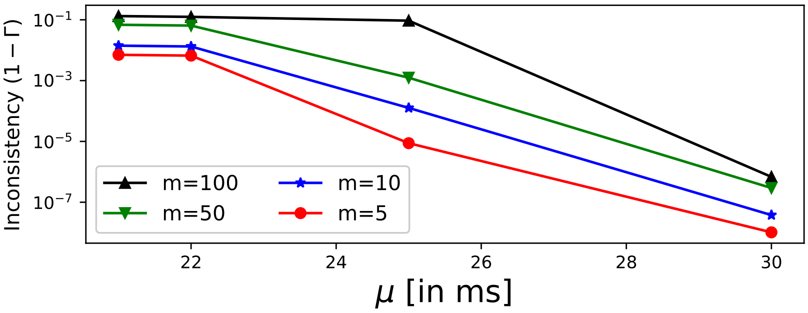


Figure 5.7 – Inconsistency of Quarts with 2 replicas as a function of μ for different values of m

Results

First, we study the inconsistency of Quarts to show the need for agreement on the measurements from asynchronous sensors. Figure 5.7 shows the inconsistency of Quarts as a function of μ for different number of sensors. We see that the inconsistency of Quarts increases with more asynchronous sensors and ranges between 10^{-2} to 10^{-8} for 5 sensors and 10^{-1} to 10^{-6} for 100 sensors. As μ increases, the measurements become less frequent, resulting in lower inconsistency.

As inconsistency increases with the number of sensors, so does the need for the extra agreement mechanism added by Quarts+. This incurs an additional availability penalty that ranges from 1-5x for 5 sensors and 5-18x for 100 sensors. Specifically, the average availability of Q+ for $n = 5$ for different values of μ is 1.2×10^{-4} and for same for Q is 2.5×10^{-5}. At $n = 100$, these values are 2.4×10^{-4} and 1.4×10^{-5}, respectively. Hence, we conclude the following.

Finding 5.4. *The inconsistency of Q increases with the number of asynchronous sensors. The price to pay to avoid this higher inconsistency with Q+ is a larger availability reduction, compared with Q.*

Next, we compare the availability and latency of Q+ with other protocols by using six characteristic scenarios, similar to those used for evaluation of Quarts in Section 5.6. The results for availability and average latency are as shown in Tables 5.5 and 5.6, respectively. Scenario #1 is the basic setup we have been using, as described in Section 5.6.3. Scenario #2 shows the impact of more controller replicas. Scenario #3 shows the impact of lower network loss rate. Scenario #4 shows the impact of lower delay fault probability. Scenarios #4 and #5 are for crash-only faults.

Finding 5.5. *Availability of Q+ is lower than that of Q but higher than AC and PC.*

Except for in Scenario #3, we see that the availability of Q+ is almost an order of a magnitude higher than that of AC and PC. In Scenario #3, the availability of Q+

	Unavailability $(1-\Psi)$			
Scenario: $(g,\ p,\ \theta_c,\ \theta_d)$	Q+	Q	AC	PC
#1: $(2,\ 10^{-3},\ 10^{-4},\ 10^{-3})$	9.9×10^{-5}	4.8×10^{-5}	1.1×10^{-3}	9.8×10^{-4}
#2: $(3,\ 10^{-3},\ 10^{-4},\ 10^{-3})$	4.62×10^{-7}	$(0, 4\times10^{-10}]^*$	9.37×10^{-4}	1.03×10^{-3}
#3: $(2,\ 10^{-4},\ 10^{-4},\ 10^{-3})$	1.18×10^{-4}	3.35×10^{-7}	1.73×10^{-4}	9.92×10^{-5}
#4: $(2,\ 10^{-3},\ 10^{-4}, 0)$	1.63×10^{-4}	1.60×10^{-5}	1.10×10^{-3}	9.80×10^{-4}
#5: $(3,\ 10^{-3},\ 10^{-4}, 0)$	3.33×10^{-7}	$(0, 4\times10^{-10}]^*$	9.93×10^{-4}	9.79×10^{-4}

Table 5.5 – Unavailability results for the chosen scenarios. * No event in 10^{10} runs

is comparable to that of PC, as PC performs very well when the network losses are extremely low. However, it is worth noting that the inconsistency of PC in Scenario #3 is 1.3×10^{-3} due to false-positives in fault-detection and the presence of multiple primary replicas, thus making it is unsafe for use.

In Scenario #4, we see that in the absence of delay faults, PC has no inconsistency and has only 6x lower availability when compared to Q+. This good performance of PC is attributed to the design of PC being tailored for crash-only faults. However, as only the primary controller replica is involved in the control, PC fails to reap the full benefits of additional replicas, as seen by the availability in Scenarios #2 and #5. Furthermore, Q+ also has a significant increase in availability when compared to AC, with each additional replica. This is similar to Q as found in Section 5.6, where agreement on input was shown to reap maximum benefits from replication.

Finding 5.6. *Average latency of Q+ is close to that of Q and significantly lower than that of other consistency guaranteeing protocol AC.*

	Average Latency (Δ)			
Scenario: $(g,\ p,\ \theta_c,\ \theta_d)$	Q+	Q	AC	PC
#1: $(2,\ 10^{-3},\ 10^{-4},\ 10^{-3})$	0.245	0.243	3.517	0.249
#2: $(3,\ 10^{-3},\ 10^{-4},\ 10^{-3})$	0.455	0.306	3.502	0.249
#3: $(2,\ 10^{-4},\ 10^{-4},\ 10^{-3})$	0.242	0.242	3.504	0.250
#4: $(2,\ 10^{-3},\ 10^{-4}, 0)$	0.169	0.167	3.515	0.250
#5: $(3,\ 10^{-3},\ 10^{-4}, 0)$	0.401	0.259	3.500	0.250

Table 5.6 – Latency results for the chosen scenarios

Similarly as in Q, the replicas in Q+ do not need to communicate in each round to reach agreement. A replica only performs agreement when it does not have the full digest for inputs. This results in a low average latency of Q+ because, in the majority of the cases, the replicas compute and issue setpoints with zero latency overhead; as opposed to AC where the replicas require consensus in each round. The tail latency of Quarts+ is bounded and is similar to that of Quarts with an extra latency of δ_n.

5.8 Conclusion

We considered the problem of agreement between replicated controllers in real-time CPSs. We explained the need for agreement as a requirement for ensuring consistency among the setpoints issued by controllers. We showed that failure to ensure consistency can result in incorrect control of the underlying physical process by the controller. The problem of agreement is particularly relevant when the controller replicas are susceptible to delay faults (in addition to the usual crash faults) and the CPS uses a reliable communication network that can drop, delay, or reorder messages.

We presented Quarts, an agreement protocol for CPSs among actively replicated controllers. Quarts is designed and formally proven to guarantee consistency with a bounded latency-overhead in a CPS with only PAs and no asynchronous sensors. To continue providing the same guarantees in the presence of asynchronous sensors, we presented an extension to Quarts, named Quarts+. We performed extensive performance evaluation of both Quarts and Quarts+, and compared their performance with that of existing agreement protocols using discrete-event simulation under different conditions of number of replicas, network losses, fault profiles, etc. We showed that besides guaranteeing consistency, the newly proposed mechanisms improve the availability of a CPS by more than an order of magnitude, when compared with existing agreement protocols. Moreover, Quarts and Quarts+ also improve the tail-latency performance of the CPS. These benefits come at a marginal increase in messaging cost when compared to passive replication schemes.

As a result of Quarts and Quarts+ reliability mechanisms, we have a CPS that also guarantees the the last of the desired correctness properties defined in Section 3.7, *i.e.,* consistency, in addition to state safety and optimal selection provided by intentionality clocks in Chapter 4. Together, these three properties imply linearizability, *i.e.,* one-copy equivalence. Therefore, we can implement active replication schemes with low latency-overhead such that they do not alter the control behavior of a reliable, non-replicated controller. In Chapter 6, we use these mechanisms to design Axo, a delay and crash fault-tolerance architecture for CPSs; it provides high-availability while ensuring correct control behavior. We also implement these mechanisms and demonstrate the importance of consistency in the context of COMMELEC, a CPS for control of electric grids. Moreover, we also apply these reliability mechanisms to SDN; these mechanisms have similar characteristics and requirements as CPSs. To this end, we design QCL that uses Quarts for agreement among replicated SDN controllers, and we show that it can reduce the latency in installation of routing updates in a communication network by several order of magnitude when compared to consensus-based approaches.

6 Axo: Tolerating Delay and Crash Faults in Real-Time CPSs

The highest result of education is tolerance.
– Helen Keller

A software fault is the inability of software to perform its required function. A fault in the central controller of a CPS can result in a failure, *i.e.,* the controller failing to maintain the underlying physical process in the desired state. As noted in the earlier chapters, replication of the unreliable components (controllers) is used to avoid such failures, despite faults. The ability of a CPS to continue performing the desired control of the physical process despite faults in some of its components is called fault tolerance. There are three main elements to fault tolerance: fault masking, fault detection, and fault recovery. The reliability mechanisms described in Chapters 4 and 5, namely intentionality clocks and Quarts, cannot provide fault tolerance on their own, as they are not helpful in the masking, detection or recovery from faults. However, they enabled us to design Axo[1], a delay and crash fault-tolerance scheme that uses active replication of the central controller, introduced in this chapter.

The central idea of active replication is that by having enough replicas of the controller, there will always be at least one non-faulty replica available to issue setpoints to the PAs. In active replication, it is essential to ensure that the issued setpoints are identical to those that would have been issued by a single non-faulty controller. This property is called linearizability or one-copy equivalence and requires that the fault-tolerance scheme ensure that all events in the CPS are correctly ordered (as done by intentionality clocks in Chapter 4) and that the issued setpoints ensure consistency (as guaranteed by Quarts and Quarts+ in Chapter 5). Although linearizability is a necessary correctness criteria for a fault-tolerance scheme, it is not sufficient. For instance, in a CPS with three replicated controllers, if the faulty controllers are unchecked, then over time, all three of the controllers could turn faulty and none of the replicas would then

---

[1] Named after Axolotl, a species of salamander found around Mexico: it is extensively used in scientific research due to its ability to quickly regenerate limbs.

be available to issue setpoints. Thus, the faulty replicas need to be quickly detected and recovered.

In addition to fault detection and fault recovery, as the state of the physical process in the CPS evolves with time, all measurements and setpoints have a validity horizon or an expiry time associated with them. For example, in a CPS for control of electric grids, let a controller receive measurements generated at t_0 form all the resources. The controller issues, based on the the snapshot of the grid at t_0, a setpoint to the battery instructing it to inject more power into the grid. Let the setpoint be received at the battery at time $t_1 > t_0$, such that the line connecting the battery to the grid at time t_1 is congested. It might be unsafe for the battery to inject more power into this line as this could result in breakage of the line. This characteristic of CPSs, where the same setpoint is *valid* at one time and *invalid* later on, is captured by the *timeliness* correctness property, which is orthogonal to the state safety, optimal selection and consistency correctness properties discussed earlier. We begin this chapter by formalizing the timeliness property and describing the challenges associated with ensuring timeliness in a CPS designed using COTS components that are susceptible to delay and crash faults.

Then, in Section 6.2, we review the related work on providing delay and crash fault-tolerance. This is followed by the design of Axo in Section 6.3. The Axo design comprises two libraries: the controller library and the PA library. These libraries are agnostic to the logic of the CPS controller and the PAs, and they are designed such that they can be easily integrated with an existing CPS. We demonstrate this through an open-source implementation of Axo that is described in Section 6.4. Using this implementation, we also study the latency overhead due to Axo. In Section 6.5, we prove the fault-tolerance properties of Axo. Specifically, we prove that Axo guarantees timeliness with minimal reduction in availability. Additionally, we derive bounds on the time Axo takes to detect and recover from crash and delay faults, and we validate these bounds through experiments with the Axo implementation.

Lastly, we consolidate the other reliability mechanisms, (intentionality clocks, Quarts and Quarts+) with Axo to obtain an implementation of full-tolerance protocol that guarantees both linearizability and timeliness, therefore enabling real-time CPSs to operate correctly despite delay- and crash-faulty controllers and unreliable communication. We study, through two deployments with different CPS, the impact of fault-tolerance on the control performance of CPSs. The CPSs used are:

- The COMMELEC CPS for real-time control of electric grids (Section 6.6)
- An inverted pendulum controlled by an LQR controller (Section 6.7)

Our concluding remarks are presented in Section 6.8.

6.1 Timeliness in a CPS – Definitions and Challenges

In order to formalize timeliness, we first introduce *conception time* and *validity horizon.* The conception time t_c for a setpoint is defined as the time instant at which the controller begins processing the measurements used for computing this setpoint. This time is the instant at which the controller finishes reconstructing the perceived state of the physical process, using the measurements received from the PAs and advertisements from the sensors. The perceived state is used for computation of the subsequent setpoints. Thus t_c is the time instant at which the perceived state of the physical process by the controller is closest to the actual state.

Recall that we assume a controller can only be crash or delay faulty, and not Byzantine faulty. Therefore, as the controller deems this state to be usable for computation (using the `ready_to_compute` function of Algorithm 3.1) before actually starting to compute the setpoints, the setpoints computed using this state cannot steer the physical process into an infeasible state, if they are implemented at t_c.

However, as the actual state of the physical process evolves with time and the perceived state at the controller is constant, these computed setpoints would only be valid for some time. This is called the validity horizon of the setpoints. For instance, in the COMMELEC [4] CPS, the measurements sent by the PAs are usually valid for few tens of milliseconds, as they include a short-term prediction of the state of the electric resource. In the case of a PV, this prediction indicates the power that the PV can inject in the next few milliseconds, based on the short-term prediction of the solar irradiance at that time instant. The quality of this prediction degrades with time, therefore, reducing the quality of the perceived state of the grid reconstructed using that measurement. Hence, if the setpoints computed using this measurement were to be implemented at a time when the solar irradiance is much different, thereby causing the PV to inject a much higher or lower power, then they could drive the grid into an infeasible state. Therefore, we define a valid setpoint as follows.

Definition 6.1 (Valid Setpoint)**.** *A setpoint is valid, if and only if, at the time of reception (t_r) at a PA, $t_r \leq t_c + \tau_o$, where t_c is the conception time of this setpoint and τ_o is the validity horizon. Else, it is* invalid.

Note that, as each setpoint is sent to exactly one PA, each setpoint has a well-defined time of reception, hence, a well-defined valid or invalid status. Moreover, the setpoints sent to different PAs resulting from the same computation have the same conception time, but possibly different times of reception. Thus, we define the timeliness correctness property for one PA as follows.

Definition 6.2 (Timeliness)**.** *Timeliness is said to hold for a PA P, if and only if all setpoints received by P are valid.*

In order to ensure timeliness, the software agents (controllers and PAs) in the CPS need to accurately measure t_c and t_r. However, in the presence of delay faults, an accurate measure cannot be obtained due to the non-zero execution time of the measuring function calls. Thus, the software agents can only obtain estimates t_c^* and t_r^*, respectively. These estimates should be obtained such that they do not violate timeliness. For example, if $t_c < t_c^*$, then the setpoints could be accepted until $t_c^* + \tau_o > t_c + \tau_o$, thereby possibly violating timeliness. Alternatively, if $t_c > t_c^*$, valid setpoints could be incorrectly discarded. Axo's approach to solving this issue and the associated assumptions are presented in Section 6.3.

In addition to correctly accepting and discarding setpoints, faulty controller replicas must be detected and recovered (repaired by rebooting). We define a faulty controller as follows.

Definition 6.3 (Faulty Controller)**.** *A controller C is faulty at time t, if all the setpoints whose conception time equals the latest conception time at or before t are invalid.*

As seen from Definition 6.3, a controller can be faulty for some time and then issue a setpoint that is valid at a PA, whereby it stops being faulty without any external intervention. This highlights the intermittence of delay faults that makes them difficult to be detected. Crash faults are a special case of delay faults with the duration of the fault being infinite, *i.e.,* the controller replica continues to remain faulty until repaired.

Crash faults are typically detected by probing mechanisms such as heartbeat [107]. This approach is inutile for delay faults as they are an end-to-end phenomena, with the two ends being the controller and the PA. A faulty replica can correctly send the heartbeat while incurring a delay in the computation, thereby fooling the failure detector and remaining undetected. To address this issue, we introduce a feedback mechanism, in the form of *validity reports,* from the PAs to the controller replicas, as described in Section 6.3.

A single computation can result in several setpoints, some of which are received at their respective PAs within the validity horizon and others are not. In this case, the controller replicas receive several conflicting validity reports. Additionally, as the communication network can drop, delay or reorder messages, controller replicas can lose some validity reports or receive old validity reports. The challenge then is to efficiently aggregate and reconcile recent validity reports to decide whether a replica is faulty.

Lastly, as delay faults are transient, it is non-trivial to decide when to regard a replica as faulty. For instance, if a replica is delay faulty for only one computation, it could be superfluous to detect as faulty because rebooting it would be more disadvantageous than letting that replica to continue running. Alternatively, if each delay-faulty replica is left undetected for long time, then all the replicas might be faulty at a later time, thereby resulting in loss of control by the CPS.

6.2 Related Work

To the best of our knowledge, Axo is the first scheme that addresses delay-fault tolerance for CPSs that use COTS-based hardware and software components.

In the literature, delay faults for real-time systems have been studied under the name of *timing faults* [132, 180, 181, 182]. The scheme closest to Axo is the work done by Veríssimo and Casimiro on TCB [132]. They propose the TCB architecture and an associated programming model that dictates how to encapsulate and rewrite the time-critical functions of real-time CPSs in the TCB module. The TCB module is a hard real-time component designed such that every function has a bounded WCET. This approach is not applicable to the upcoming COTS-based 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. Moreover, several components of the TCB architecture require an implementation specific to each CPS, whereas Axo is a layer of software that can be used on any CPS that satisfies the assumptions (Section 6.3.1) and requires only minor additions to the CPS controller software. This enables the deployment of Axo on existing CPSs. We demonstrate this by deploying Axo on two different CPSs.

Another approach to delay-fault tolerance is the timing-failure detection under the TTA that also relies on bounding the WCETs of the different function calls. This requires the static analysis of generally complex functions that might include COTS software. Additionally, we have also seen that in some CPSs such as COMMELEC [4], the execution time heavily depends on the parameters provided at run time. This would require further dynamic analysis of the execution time, a task that does not fit within the real-time constraints of CPSs. Moreover, TTA requires *a priori* knowledge of all time instants at which a controller or a PA sends and receives messages in a round [183]. This is a much stronger requirement than TCB and cannot be applied to COTS-based CPSs.

Other work [180] in this field has focused on improving the QoS and response time of the systems. The authors focus on transaction systems as opposed to CPSs, and do not aim at providing hard real-time constraints, as Axo does.

Active replication protocols, such as [76, 100, 113, 114], use multiple replicas, all of which simultaneously compute setpoints. When one of the replicas is delay faulty, it can still send a setpoint at some time after its validity horizon. The implementation of such a setpoint violates the timeliness property hence jeopardizes the control of the process done by other correct controller replicas. To discard such setpoints and provide timeliness in the presence of delay faults, Axo uses active replication with an added mechanism. Active replication techniques rely on replica determinism (or linearizability) for correct control [184]. We use Quarts and intentionality clocks to provide replica determinism.

To detect inconsistencies [86], mechanisms for fault detection rely on monitoring the replica, such as using heartbeats [99], or on probing the replica for its current state. Such mechanisms target crash-only faults and cannot be extended to delay faults that are inherently an end-to-end property. Replicas themselves do not contain any state to indicate whether or not they are delay faulty, hence probing or monitoring the replicas will not provide insight for delay-fault detection. Our solution makes use of the PAs in order to correctly detect delay faults.

Another method for to detecting faults is through the detailed modeling of the controller under faulty and non-faulty conditions [125, 185]. 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*.

6.3 Axo

We first describe the assumption on the controller and PAs required by Axo in Section 6.3.1, followed by the design of Axo in Section 6.3.2.

6.3.1 Assumptions

We assume that the software agents in the CPS, namely the controller replicas and the PAs, are time-synchronized. As CPSs perform real-time operations on distributed nodes, they naturally have a global notion of time either obtained by GPS [179] or network protocols such as NTP [165] or PTP [164]. The inaccuracy of the time-synchronization protocol is upper-bounded and the bound (δ_s) is known to Axo.

We also assume that there exists a known validity horizon τ_o for each setpoint. Axo requires τ_o as input in order to perform fault masking and fault detection.

We consider the same controller model, as in the previous chapters (introduced in Algorithm 3.1). In order to have linearizability, Axo requires that the controllers implement intentionality clocks (Chapter 4) for the ordering of events, and Quarts and Quarts+ (Chapter 5) for consistency. The resulting design of the control is augmented with a new mechanism to provide timeliness, as described in Section 6.3.2. The controller is assumed to be susceptible to crash or delay faults. Byzantine faults [119] that result in malicious setpoints are not considered.

The PAs are assumed to be non delay-faulty. Unlike the central controller, the PAs are not single-points of failure of the CPS. When a PA fails, one controlled resource is affected but not the entire physical process. Moreover, the functions implemented by PAs are often simpler than the those implemented by the controller. Thus, they are less susceptible to delay faults.

As in previous chapters, we assume a probabilistic synchronous network [58] that can drop, delay, reorder or retransmit messages.

6.3.2 Design

Overview

Axo uses the active replication of the central controller with $2g + 1$ replicas to tolerate g crash and delay faulty replicas. In the special case of $g = 1$, Axo only requires $g + 1 = 2$ replicas. This is because fault masking, fault detection and fault recovery in Axo can be performed with $g + 1$ replicas, but the agreement mechanism (Quarts) generally requires $2g + 1$ replicas, except in the case of $g = 1$.

Each controller is assigned a unique replica ID that serves as an identifier for all ensuing Axo-related message exchanges. Each controller independently receives measurements and advertisements, performs agreement on the measurements by using Quarts, performs agreement on advertisements using Quarts+, computes and issues setpoints. In order to ensure timeliness, Axo needs to discard invalid setpoints at the PA. To this end, it performs timestamping of outgoing setpoints at a controller. The code of the controller is instrumented to obtain an estimate (t_c^*) of the conception time and to send this estimate to Axo. At the controller, Axo attaches the received timestamp to the outgoing setpoint. At the PA, the setpoint is accepted or discarded based on its validity. All these operations are done transparently with respect to the controller and the PAs. For this, Axo uses one library at the controller replica and one at the PA namely, the *controller library* and the *PA library* (as shown in Figure 6.1).

The controller library comprises three components: the tagger, the detector and rebooter. The PA library consists of the masker. Together, the tagger and masker achieve fault-masking, *i.e.,* discard invalid setpoints. The tagger receives t_c^* from the controller. It also intercepts all outgoing setpoints at a controller replica and appends each setpoint with the last received timestamp t_c^*. Additionally, the tagger creates the Axo header that contains other information regarding the state of this replica. This includes the health of the replica (`h`), a score that indicates the recent latency performance of the replica and that is used to decide if the replica is delay faulty (see Section 6 for more details). The health is constantly updated using its current value and the validity of new setpoints. Thus, a newly added replica, or a replica that lost some messages, can seamlessly synchronize with its peers and reconstruct the

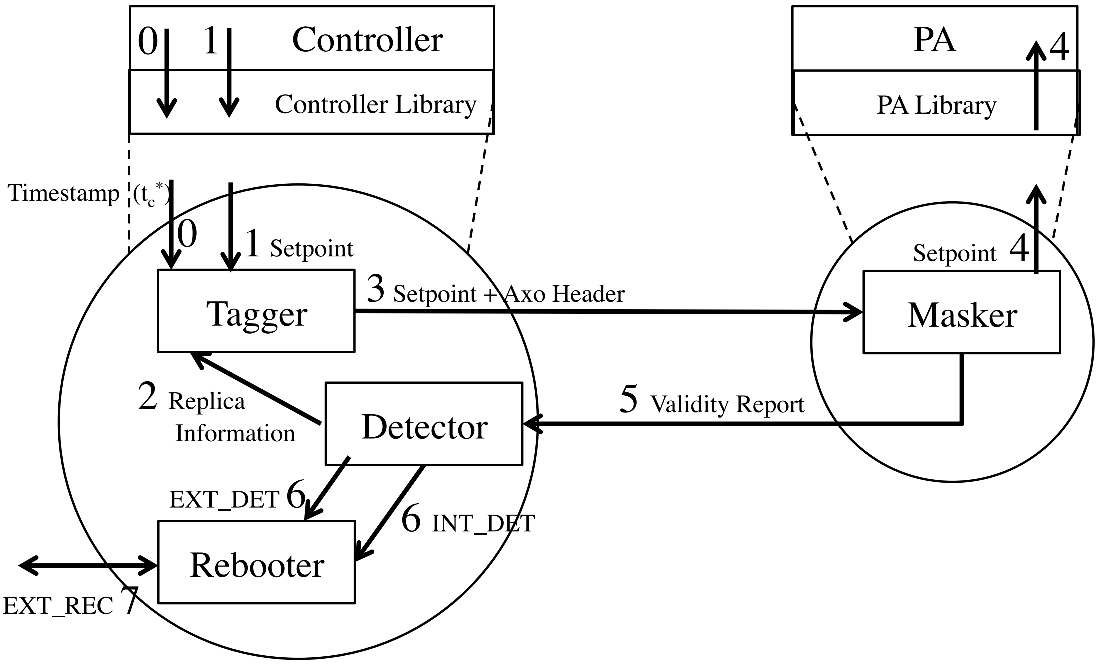


Figure 6.1 – Axo design

latest state of the all other replicas in the CPS. In this way, Axo exploits the inherent communication between PAs and the controller replicas in a CPS to maintain the the detection mechanism soft-state [169]. The setpoint, along with the Axo header, containing the information required for fault-masking and recreating the original setpoint, is sent to the masker on the corresponding PA.

At the masker, the setpoint is accepted or discarded based on the time of reception, the validity horizon and t_c^*. Then, the masker sends a validity report to all the controller replicas, indicating whether the received setpoint was valid or not. This enables them to decide whether the controller replica that sent the setpoint is non-faulty or not. If the setpoint is valid, the masker reconstructs the original setpoint and forwards it to the PA that receives and implements it as if it were sent directly by a controller.

The detector on each replica processes the received validity reports to update its health and that of other replicas, in order to detect a crash- or delay-faulty replica. When a replica C_i is detected as faulty by another replica C_j, the detector on C_j informs its rebooter to recover replica C_i by rebooting it. As noted in earlier works on software faults [124, 128, 186], simply rebooting a machine resets the state of the operating system and the software stack, thereby making it non-faulty.

Next, we described the fault-masking, fault-detection and fault-recovery mechanisms in detail.

Fault Masking: Tagger and Masker

Algorithm 6.1: Controller model with all the reliability mechanisms namely intentionality clocks, Quarts, Quarts+ and Axo. The parts in red show the modifications to the controller made by Axo

```
1  $r \leftarrow 0$;                              // highest label of measurements received
2  $\mathbf{Z} \leftarrow \emptyset$;             // vector of measurements with label $r$
3  $\mathbf{Y} \leftarrow \emptyset$;             // list of timestamped advertisements
4  $\mathbf{V} \leftarrow \emptyset$;             // vector of inputs to be used for computation
5  $\mathbf{\Lambda} \leftarrow [\Lambda_1, ..., \Lambda_n]$;   // lower bounds of inter-arrival time of sensors
6
7  repeat
8      $\mathbf{Z}, r \leftarrow$ aggregate_received_measurements($r$);
9      $\mathbf{Y} \leftarrow$ aggregate_received_advertisements();
10 forever;
11
12 repeat
13     success $\leftarrow$ False;
14     if $r > r^-$ then
15         success_quarts, $\mathbf{D}$, $\mathbf{Z} \leftarrow$ quarts($\mathbf{Z}, r$);
16         if success_quarts then
17             $T_0 \leftarrow \mathcal{F}(\mathbf{D}, \mathbf{Z})$;
18             success, $\mathbf{V} \leftarrow$ agree_on_advertisements($\mathbf{D}, \mathbf{Y}, r, T_0, \mathbf{\Lambda}$);
19         end
20     end
21     $t_c^* \leftarrow \mathrm{T}_{now}$;
22     decision $\leftarrow$ ready_to_compute($\mathbf{V}, r, T_{now}$);
23     if success and decision then
24         send $t_c^*$ to the tagger ;
25         $\mathbf{X} \leftarrow$ compute($\mathbf{V}$);
26         issue($\mathbf{X}, r$);
27     end
28 forever;
```

Fault masking is achieved by the tagger and the masker, using the controller to obtain an estimate t_c^* of the conception time of a setpoint t_c. To explain this mechanism, we present the consolidated model of the controller in Algorithm 6.1. It comprises all the reliability mechanisms introduced in this thesis, namely intentionality clocks, Quarts, Quarts+ and Axo. The label r is obtained using intentionality clocks. Lines 13-18 represent Quarts and Quarts+. The newly added lines in red (lines 21 and 24) are due to Axo and will be the focus of our attention.

The purpose of the extra lines added to the controller by Axo is to obtain an estimate t_c^* such that $t_c^* < t_c$ (to ensure timeliness) and that t_c^* is as close as possible to t_c to minimize reduction in availability, as discussed in Section 6.1. The increased expressiveness of our controller model due to the `ready_to_compute` function enables us to obtain such an estimate. As the controller deems the perceived state of the process sufficient to compute setpoints at Algorithm 6.1 line 22, it is therefore the conception time of the subsequent setpoints. As the controller is susceptible to delay faults, it might incur a large delay in the `ready_to_compute` function. So, we measure t_c^* at line 21. The recorded timestamp is sent to the tagger at line 24.

Algorithm 6.2: Tagger

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

The tagger is as described in Algorithm 6.2. For each setpoint, it receives the timestamp t_c^* from the controller and intercepts setpoints on their way to the PA. Then it uses the t_c^* to create the 20-byte Axo header that comprises

- Replica ID (1 byte)
- Destination port of the original setpoint (2 bytes)
- Setpoint timestamp: estimate of the conception time of this setpoint t_c^* (8 bytes)
- Detector timestamp: computed as the last time at which its detector processed a validity report t_d (8 bytes)
- The health of the replica `h` (1 byte)

The last two fields of the Axo header are obtained from the detector. These values are regularly updated every time the detector processes a validity report received from the PAs (Algorithm 6.2 line 8). The tagger prepends the Axo header to the setpoint and sends it to the masker.

Algorithm 6.3: Masker

```
for each setpoint received do
    if $T_{now} \leq t_c^* + \tau$ then
        Remove Axo header and send setpoint to PA;
    end
    Send validity report to detectors of controllers;
end
```

The design of the masker is given by Algorithm 6.3. For each received setpoint, the masker records the time of reception. Then, it compares the t_c^* in the Axo header of the received setpoint with τ, where τ is a conservative measure of the validity horizon. It is computed using the upper-bound on the inaccuracy of the time-synchronization (δ_s) and the upper-bound on the computation time at the masker between performing the validity check (line 2) and sending the setpoint to the PA (line 3). τ is given by $\tau_o - (2\delta_s + \delta_m)$. As proven in Theorem 6.5.1, this approach guarantees timeliness.

If the setpoint is valid, the masker removes the Axo header, modifies the destination port of the message using the destination port in the Axo header, and forwards the setpoint to the PA. It also uses the other fields in the Axo header to create and send a validity report to all controller replicas. A validity report $\mathbf{VR}$ consists of the following five fields, first four of which are same as found in the Axo header:

- timestamp t_c^* of the setpoint ($\mathbf{VR}.ts$)
- Replica ID ($\mathbf{VR}.id$)
- The health of the replica ($\mathbf{VR}.h$)
- Detector timestamp of the issuing replica ($\mathbf{VR}.td$)
- A flag indicating if the setpoint was valid at the time of reception ($\mathbf{VR}.v$)

Fault Detection: Detector

The design of the detector is given by Algorithm 6.4. It consists of five main blocks: initialization (line 1), the aggregation of validity reports (lines 5-15 and 38-41), update of the detector timestamp (lines 18-19), delay-fault detection (lines 21-25), and crash-fault detection (lines 28-34).

The detector on each replica maintains a database $\mathbf{DB}$ with one record for each replica, including itself. We denote each record in the database by $\mathbf{db}$. The record $\mathbf{db}$ for a replica with ID id is obtained as $\mathbf{db} = \mathbf{DB}[id]$ and contains the following fields:

- Setpoint timestamp: the highest t_c^* in all the validity reports processed for that replica ($\mathbf{db}.ts$)
- Detector timestamp: the highest received detector timestamp ($\mathbf{db}.td$)
- The health of that replica with ID id, as seen by current replica ($\mathbf{db}.h$)
- Non-faulty flag: A flag showing if the replica was non-faulty in the last validity report ($\mathbf{db}.nf$)

When the detector boots, it initializes the record in the database with its own ID (line 1) as follows. The setpoint timestamp (ts) and the detector timestamp (td) are set to the current time. The heath field for a healthy replica is set to its maximum value H_{max}. The non-faulty flag (nf) field is set to true as the replica is considered non-faulty at boot.

The detection mechanism is triggered at a replica, when it receives a validity report ($\mathbf{VR}$). The first part of the detection mechanism is the aggregation of validity reports (lines 5-15 and 38-41). As validity reports can be delayed, lost, retransmitted, or reordered, aggregating them efficiently is tricky. To efficiently aggregate the reports, we identify each report with its setpoint time $\mathbf{VR}.ts$. As all the reports from a single computation have the same conception time, the validity reports resulting from all the PAs corresponding to this computation have the same ts. Alternatively, we could also use the label of the setpoint obtained from the intentionality clock. When a new validity report is received, the database for this record is updated according to the function `updateDB` given by Algorithm 6.5.

In `updateDB` function, the setpoint timestamp $\mathbf{db}.ts$ and the detector timestamp $\mathbf{db}.td$ of the database record for the replica in the validity report are set to the respective values in the validity report. This indicates that the replica in the validity report was active at $\mathbf{VR}.ts$ and that its detector was active at $\mathbf{VR}.td$, thereby other replicas are not to mark it as crash faulty. Then, the health of the replica is updated using exponential averaging with a parameter α, and penalty $-H_{max}$ when the replica was faulty, or a bonus $+H_{max}$ when the replica was non-faulty. The exponential averaging serves three purposes:
(1) It smoothens out the health and dampens the effect of outliers, thus preventing short and transient delay faults from resulting in costly rebooting.
(2) It captures the recent history of replica, thus a marking a replica that has a high frequency of delay faults, as faulty.
(3) It keeps the health between $-H_{max}$ and $+H_{max}$, thereby preventing overflow.

By using exponential averaging, Axo marks a replica as delay faulty based on the frequency of delay faults rather than the presence or absence of the faults. The value α $(0 \leq \alpha \leq 1)$ represents the weight given to the historical performance of the replica and the instantaneous frequency of delay faults that is deemed as faulty by the CPS. The last operation in the `updateDB` function is to set the non-faulty flag of the controller replica to the value of the valid flag present in the current validity report. Notice that we update the health, based on the previous value of the non-faulty flag. This is done to avoid penalizing a replica in the cases when some of the PAs receive invalid setpoints and others receive valid setpoints. Recall from Section 6.1 that a replica must not be detected as faulty if only some of the PAs receive invalid setpoints for a given label. To this end, the detector performs a logical OR of the $\mathbf{VR}.v$ flags received in reports with the same setpoint timestamp (lines 38-41). Thus, if at least one of the PA receives a valid setpoint in a round from the controller replica, the controller is deemed non-faulty. As the replica needs to be deemed non-faulty even if the last validity report returns with the valid flag set to true, we delay the update of health until the next validity report for this setpoint. This adds extra latency in the detection of a faulty replica, which is a small price to pay to avoid costly rebooting of non-faulty replicas.

After processing the report and updating the database, the detector timestamp of the replica is updated at Algorithm 6.4 lines 18. Then, the detector timestamp and the health of this replica are sent to the tagger; they are then sent to the PAs in subsequent Axo headers, and in turn echoed in the validity reports. In this way, newly added replicas can learn about the existing replicas, the last time they were active, and their health score; thereby making detection soft state [169]. When a validity report corresponding to a replica that is already present in the database is received (lines $14-15$), the health in the database is updated to the minimum of the existing health and the received health. Thus, replicas do not overestimate the health of other replicas, hence a truly faulty replica is quickly detected.

Once the reports have been aggregated and the tagger has been informed of the new database information, the detector can perform the actual fault-detection. A replica is detected as delay faulty (lines 21-25) 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 as described in the section on fault recovery.

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.

For crash-fault tolerance, we use the parameter τ_c that is the time of relative inactivity of a software agent after which it is considered crash faulty. Thus, if a replica is stalled for longer than τ_c, it is deemed faulty. In Axo, we compare the inactivity of a replica with that of its peers, instead of using a constant time of inactivity to detect a crash fault. This is particular useful in CPSs with a non-constant rate of issuing of setpoints, where waiting for a constant time of inactivity of a replica can result in incorrect fault-detection. For each replica in its database, the detector compares the value of ts with the maximum of all ts's. If the difference is greater than τ_c, then that replica is deemed crash faulty. In this way, a replica is only considered to be crash faulty if it has been inactive for a period of τ_c while other replicas have been active. A similar check is done for td's, to detect detector crashes.

Fault Recovery: Rebooter

The design of the rebooter is given by Algorithm 6.6. The rebooter reacts to two types of detection messages from its local detector, namely internal detection message (`INT_DET`) and external detection message `EXT_DET`. To communicate with other rebooters in order to perform remote recovery, the rebooter uses the external recovery request message (`EXT_REC`) and `ACK` messages. The messages from the detector to the rebooter (`INT_DET` and `EXT_DET`) are timestamped with the setpoint timestamp present in the validity report that caused the detection event.

When a replica C_i detects another replica C_j as faulty, by using Algorithm 6.4, the detector on C_i sends an `EXT_DET` to its local rebooter. This triggers the rebooter on C_i to initiate recovery of the faulty replica C_j by sending external recovery requests (EXT_REC) as seen in Algorithm 6.6 lines 8-17. In order to confirm that C_j has indeed rebooted due to this detection event, C_i waits for an `ACK` message for a time T_r after sending the `EXT_REC`. In order to maximize the chances of successful recovery, in the presence of network losses, C_i repeats this process until the `ACK` is received, with a maximum of `maxSend` number of tries, as configured. Furthermore, when C_j accepts the reboot request from C_i, it sends an `ACK` with a timestamp $t + \tau_{reboot}$ to all controller replicas, where τ_{reboot} is a parameters that dictates the minimum interval between consecutive reboots. When a replica that is currently performing external recovery of C_j receives an `ACK` with a timestamp larger than that of the detection event, it marks C_j are recovered and stops the recovery protocol.

The threshold τ_{reboot} is also used to minimize superfluous reboots in the presence of message losses: for example, in a case when C_i and C_k both detect C_j as faulty, but in different rounds. This can happen due to network losses causing some validity reports being lost at C_i, thus making C_i and C_k have different timestamps of detection events. As a result, they will send `EXT_REC` messages with different timestamps, triggering different reboots. This is avoided by using τ_{reboot}.

Alternatively, when a replica detects itself as faulty, it sends an `INT_DET` to its rebooter. When a rebooter receives an `INT_DET` with a timestamp higher than the last time that the replica rebooted, the replica is immediately rebooted. In contrast to external recovery, internal recovery does not incur any latency or messaging overhead. Therefore, it is preferred over external recovery. Hence, the threshold for external recovery (H_{ext}) is chosen to be less than that for internal recovery (H_{int}).

In order to avoid multiple reboots from the same detection event, the rebooter stores the last time it rebooted (`lastReboot`) in a persistent storage (such as disk) and reads it at the time of initialization.

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

Algorithm 6.4: Detector

```
initialize DB[myID];
for each report VR received do
    if $VR.id \notin DB$ or $VR.ts > DB[VR.id].ts$ then
        // New report
        if $VR.id \notin DB$ then
            // New replica
            create $DB[VR.id]$;
            $DB[VR.id].ts \leftarrow VR.ts$;
            $DB[VR.id].td \leftarrow VR.td$;
            $DB[VR.id].nf \leftarrow VR.v$;
            $DB[VR.id].h \leftarrow H_{max}$;
        else
            // Existing replica
            $DB[VR.id].h \leftarrow \min(VR.h, DB[VR.id].h)$;
            updateDB(DB[VR.id],VR);
        end
        // Update and send detector timestamp to the tagger
        $DB[myID].td \leftarrow \max(DB.ts)$;
        Send $DB[myID].h$, $DB[myID].td$ to tagger;
        // Delay fault detection
        if $VR.id \neq myID$ and $DB[VR.id].h \leq H_{ext}$ then
            EXT_DET($DB[VR.id].ts$,$VR.id$);
            delete $DB[VR.id]$;
        else if $DB[myID].h \leq H_{int}$ then
            INT_DET($DB[myID].ts$);
        end
        // Crash fault detection
        for each $id$ in DB do
            if $\max(DB.ts) - DB[id].ts > \tau_c$ or $\max(DB.td) - DB[id].td > \tau_c$ then
                if $id = myID$ then
                    INT_DET($\max(DB.ts)$);
                else
                    EXT_DET($\max(DB.ts), id$);
                    delete $DB[id]$ ;
                end
            end
        end
    else if $VR.ts = DB[VR.id].ts$ then
        // Possibly new report from a different PA
        $DB[VR.id].nf \leftarrow DB[VR.id].nf \lor VR.v$;
    end
end
```

Algorithm 6.5: updateDB(**db**,**VR**): Function to update detector database

```
db.$ts$ ← VR.$ts$;
db.$td$ ← VR.$td$;
if db.$nf$ then
    db.$h \leftarrow \alpha \times$ db.$h + (1-\alpha) \times H_{max}$;
else
    db.$h \leftarrow \alpha \times$ db.$h - (1-\alpha) \times H_{max}$;
end
db.$nf$ ← VR.$v$;
```

Algorithm 6.6: Rebooter

```
lastReboot ← loadLastReboot();
for each message received do
    if message is INT_DET(t) then
        if $t >$ lastReboot $+ \tau_{reboot}$ then
            saveLastReboot(t) ;
            reboot the replica;
        end
    else if message is EXT_DET($t_1$, ID) then
        sentCtr ← 0;
        while sentCtr < maxSend do
            send EXT_REC($t_1$) to replica ID;
            sentCtr++;
            listen for $T_r$ ;
            if (ACK($t_2$) received) and ($t_2 \geq t_1$) then
                break;
            end
        end
    else if message is EXT_REC(t) then
        if $t >$ lastReboot $+ \tau_{reboot}$ then
            send ACK(t + $\tau_{reboot}$) to all replicas;
            saveLastReboot(t);
            reboot the replica;
        else
            send ACK(lastReboot + $\tau_{reboot}$);
        end
    end
end
```

6.4 Implementation

We developed an open-source proof-of-concept implementation of Axo in C++[2]. In order to demonstrate how Axo can be easily integrated with an existing CPS, we designed an API in C++ that can be used to instrument a CPS controller to measure t_c^* and send it to the tagger, as described in Algorithm 6.1.

The API provides two functions for recording t_c^*, (`get_timestamp_ptp` and `get_timestamp_gps`) and one function (`send_timestamp`) to send it to the tagger in the desired format. We make a distinction between the CPSs that use PTP- or GPS-based time-synchronization because PTP-based time-synchronization has a much lower clock-update rate, when compared to GPS. This can result in a time-synchronization inaccuracy of $\delta_s \sim 5\ ms$. Thus, in order to be able to mask delays in the range of milliseconds with PTP-based time-synchronization, we use the offset of the slave clock from the master clock to correct the timestamp, by using the function `get_offset`. This function is called periodically with a period T_{offset} (default 1 s). This approach lowers the time-synchronization inaccuracy to $\delta_s \sim 100\ \mu s$. The three functions need to be inserted in the controller at the appropriate locations, as required by Algorithm 6.1. Besides these modifications to the controller using the Axo API, the installation of Axo is plug-and-play. It requires no more information about the inner workings of the controller or the PAs, hence is agnostic to the CPS.

In order to characterize the latency overhead due to the Axo API, we profiled its function calls on the NI cRIO-9068 industrial computer that is commonly used for in-field deployments of CPSs. The average time spent in the `get_timestamp_gps` function is $\Delta_{gps} = 14.26\ \mu s$, and the time spent in the `send_timestamp` function is $\Delta_{send} = 33.18\ \mu s$. The time spent in the `get_timestamp_ptp` function, however, depends on the period (T_{offset}) with which the `get_offset` function is called. The `get_offset` function takes about $194\ \mu s$ on average. Thus, the average time taken by the `get_timestamp_ptp` function (Δ_{ptp}) is in the range $[194\ \mu s, 209\ \mu s]$. Therefore, the latency-overhead due to the Axo API when the CPS uses GPS is $\Delta_{gps} + \Delta_{send} = 46.44\ \mu s$ and, the latency-overhead when the CPS uses PTP is in the range $[227.18\ \mu s, 242.18\ \mu s]$.

To achieve a plug-and-play design, the tagger needs to intercept setpoints sent from the controller to the PAs. To this end, we use the `libnetfilter_queue`[3] (NF_QUEUE) framework from the Linux *iptables* project. NF_QUEUE is a user-space library that exposes a network-layer interface that enables filtering and modifying packets, transparent to the application layer. We use this interface to filter packets, based on the destination IP address of the PAs and the destination port number of the PA application, thus, limiting the footprint to only the relevant packets. In the NF_QUEUE framework all filtered packets are queued until further dequeued by a user-space application.

---

[2] Available at https://github.com/Wajeb/axo.

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

Thus, once the filtering rules are in place, all the packets remain in the queue until they are dequeued by the tagger. Therefore, if the tagger crashes, the setpoint cannot bypass Axo, thereby upholding the timeliness property by trivially discarding all setpoints at the controller.

To ensure that the rebooter always functions correctly despite crashes, in order to be remotely recovered by other replicas, we implement a heartbeat mechanism between the detector and rebooter. When the detector loses five consecutive heartbeats, it attempts to reinstantiate the rebooter. If it fails, the detector will reboot the machine.

We also incorporated Quarts and intentionality clocks along with the Axo implementation, as libraries that can be used by a CPS for ordering of events and controller consistency. The proof-of-concept implementation currently supports up to 256 controller replicas. However, in practice most CPSs do not use more than 3 or 4 replicas. We deploy this implementation with two CPS and present the results of the case studies in Sections 6.6 and 6.7. We also use the same implementation with SDN in Chapter 7.

6.5 Performance Guarantees

In this section, we will prove that the Axo design presented in Section 6.3 guarantees timeliness and maximizes availability. We also derive and experimentally validate bounds on the detection and recovery time of Axo, for delay and crash.

6.5.1 Timeliness Guarantees

Recall from Definition 6.2 that timeliness requires that the PAs never implement invalid setpoints. For Axo, we have the following result.

Theorem 6.5.1 (Axo Timeliness)**.** *If all controller replicas of a CPS use the controller library and all PAs use the PA library described in Section 6.3.2, then timeliness is guaranteed for all PAs.*

Proof. Let s be a setpoint computed at some controller replica C with a timestamp $t_c^* \leq t_c$, where t_c is the first instant at which the measurement to compute this setpoint were processed. (refer Section 6.1)
As C contains the controller library, then s will be intercepted by the tagger.
If P does not contain an PA library, it will never receive s, thereby upholding timeliness.
Otherwise, the tagger will forward s to the masker of P.
Recall from Section 6.1 that the time at which the setpoint is received at P is t_r.
By Definition 6.1, s is valid if and only if $t_r \leq t_c + \tau_o$.
Based on Algorithm 6.3 line 2, the masker of P will only accept s at $t_r^* \leq t_c^* + \tau$.
Recall that both t_c^* and t_r^* are measured locally at C and P, respectively.

Since the inaccuracy in time-synchronization protocol is δ_s, the true time at which the setpoint is received at the masker is $t'_r \le t^*_r + \delta_s$.
Similarly, the true time at which the setpoint is first valid $t_c \ge t^*_c - \delta_s$. Therefore, $t^*_r \le t^*_c + \tau \implies t'_r \le t_c + \tau + 2\delta_s$
Since the processing time of the masker is bounded by δ_m, then $t_r \le t'_r + \delta_m$.
Thus, any accepted setpoint will arrive at P at $t_r \le t_c + \tau + 2\delta_s + \delta_m$.
But $\tau = \tau_o - 2\delta_s - \delta_m$
So, the masker of P will only accept and forward s to P if $t_r \le t_c + \tau_o$.
Therefore, any setpoint s received by P will be valid. □

Note that a CPS, where the PAs discard all setpoints, also trivially satisfies timeliness. However, such a CPS would fail to control the physical process as no setpoints are implemented. Therefore, besides providing timeliness, a CPS must be available to perform the control of the physical process. A measure of how available a CPS is when providing timeliness is given by *Timely Availability* as follows.

Definition 6.4 (Timely Availability)**.** *Timely availability is said to hold for a PA P in an interval $[a,b]$, if and only if P receives at least one valid setpoint in $[a,b]$ from a set of controller replicas $\mathcal{C}$. Consequently, $\mathcal{C}$ is said to* provide timely availability *for P in $[a,b]$.*

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

Recall from Section 6.3.1 that the upper-bound on the synchronization inaccuracy of the time-synchronization protocol is δ_s. Thus, any measurement of time has an uncertainty of δ_s. Consequently, the uncertainty in recording the end-to-end delay of a setpoint is $2\delta_s$. Therefore, to guarantee timeliness, any fault-tolerance protocol needs to conservatively discard setpoints with an end-to-end delay greater than $\tau_o - 2\delta_s$. Moreover, this deadline needs to be further offset, in order to account for the computation time of the fault-tolerance protocol (δ_f). Hence, in order to tolerate delay faults, all fault-tolerance protocols need to discard potentially valid setpoints whose end-to-end delay lies in the uncertainty interval $[\tau_o - 2\delta_s - \delta_f, \tau_o]$, thereby reducing availability. For Axo, we have the following result that shows how Axo provides timely availability when any other fault-tolerance protocol with the same timeliness guarantees also provides timely availability.

Theorem 6.5.2 (Axo Timely Availability)**.** *Consider an interval $[a,b]$ and a fault-tolerance protocol f that, using a set of g replicas $\mathcal{C}$, guarantees timeliness for a PA P. If f provides availability for P in $[a,b]$, and the time taken by f to process a setpoint is at least as much as that taken by Axo, then Axo, using $\mathcal{C}$, also provides availability for P in $[a,b]$.*

Proof. Consider a CPS with a PA P.
Let $\mathcal{C} = \{C^1, ..., C^g\}$ be a set of g replicas of the controller of this CPS.

Let $\mathcal{S}$ be the set of setpoints sent by all controllers in $\mathcal{C}$ in the interval $[a - \tau_o, b]$.
Consider a fault-tolerance protocol f, applied to $\mathcal{C}$, that guarantees timeliness for P.
Denote by $\mathcal{C}_f$ the set of controllers $\mathcal{C}$ when f is applied to them.
Let $\mathcal{R} = \{s \in \mathcal{S} : s$ is received by P in $[a, b]$ and s is valid$\}$.
Formally, $f : \mathcal{S} \mapsto \mathcal{R}_f \subseteq \mathcal{R}$.
In other words, $\mathcal{R}$ is the set of valid setpoints sent by $\mathcal{C}$ and received by P in $[a, b]$. $\mathcal{R}_f$ is the subset of $\mathcal{R}$ that are received by P when f is applied to $\mathcal{C}$. Therefore, f provides availability for P in $[a, b]$, if and only if $|\mathcal{R}_f| > 0$. Note that we only consider the interval $[a - \tau_o, b]$ for the set $\mathcal{S}$, since setpoints sent outside this interval can never be valid if received in $[a, b]$.
We define the following operations and sets:
Let $\mathcal{S}_f \subseteq \mathcal{S}$, be the set of setpoints, sent by $\mathcal{C}$ in $[a - \tau_o, b]$, that f permits to be sent to P.
That is, $\mathcal{S} \setminus \mathcal{S}_f$ is the set that f discards before sending.
Let $\alpha_f : \mathcal{S} \mapsto \mathcal{S}_f$
Let $\mathcal{N}_f \subseteq \mathcal{S}_f$ be the set of setpoints that the network delivers to P in $[a, b]$, when f is applied.
Then, $\mathcal{N}_f = \{s \in \mathcal{S}_f : s$ is delivered to P in $[a, b]\}$
Let $\gamma : \mathcal{S}_f \mapsto \mathcal{N}_f$
Then, $\mathcal{R}_f \subseteq \mathcal{N}_f$
Let $\beta_f : \mathcal{N}_f \mapsto \mathcal{R}_f$
Therefore, $f = \beta_f \circ \gamma \circ \alpha_f$
Intuitively, α_f is the operation of discarding setpoints before sending them, and thus depends on the fault-tolerance protocol. γ is the operation performed by the network, which is considered to be transparent to the fault-tolerance protocol. β_f is the operation of discarding setpoints before they are received at P, in order to guarantee timeliness.
Let $\mathcal{F}_g$ be the class of fault-tolerance protocols that guarantee timeliness to hold for P by using controllers in $\mathcal{C}$. We consider all $f \in \mathcal{F}_g$ to have at least as much processing time for each setpoint as Axo.
As Axo guarantees timeliness (Theorem 6.5.1), Axo $\in \mathcal{F}_g$
As Axo uses active replication, it ensures that all g controller replica in $\mathcal{C}$ are active and send setpoints to P.
Furthermore, as the tagger (Algorithm 6.2) never discards setpoints, all the setpoints sent by the controllers are forwarded to P.
However, the tagger incurs a processing time to each setpoint, thus not all setpoints will still be sent in $[a - \tau_o, b]$.
This processing time is also incurred by all $f \in \mathcal{F}_g$, therefore

$$\forall f \in \mathcal{F}_g : \mathcal{S}_f \subseteq \mathcal{S}_{Axo} \subseteq \mathcal{S} \tag{6.1}$$

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

Then, it follows from Equation 6.1, that

$$\forall f \in \mathcal{F}_g : \mathcal{N}_f \subseteq \mathcal{N}_{Axo} \tag{6.2}$$

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

$$\mathcal{R}_f = \beta_f(\mathcal{N}_f) = \{s \in \mathcal{N}_f : \text{end-to-end delay of } s \leq \tau_f\} \tag{6.3}$$

Note that,

$$\forall \mathcal{A}, \forall f_1, f_2 \in \mathcal{F}_g, \tau_{f_1} \leq \tau_{f_2} \implies \beta_{f_1}(\mathcal{A}) \subseteq \beta_{f_2}(\mathcal{A}) \tag{6.4}$$

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

$$\forall f \in \mathcal{F}_g, \tau_f \leq \tau \tag{6.5}$$

Then, using Equations 6.2, 6.3, 6.4, 6.5, we conclude that

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

□

6.5.2 Bounds on Recovery Time

Now, we derive upper and lower bounds on the distribution of recovery time with Axo. The recovery time is the time taken for a replica to be detected and recovered, after it starts being faulty. Evaluating the exact expression for recovery time seems to be mathematically intractable. Hence, we derive the bounds and validate them through experimental results of fault recovery obtained using the implementation of Axo, as described in Section 6.4 along with a test CPS controller. Besides validating the computed bounds, this also serves as a validation of the proof-of-concept implementation of Axo.

Analytical Controller and Fault Model

Although the fault model used in simulation studies for Quarts in Section 5.6 expresses both delays and crash faults, along with the delays in each round, it does not lend itself well for mathematical analysis. We will use a simplified fault model described below.

The faults on a controller replica are independent of the faults on other replicas. Each controller replica is in one of two-states faulty or non-faulty. As a controller replica experiences different delay in the faulty and non-faulty states, the inter-arrival times of setpoints issued by the setpoints follow different distributions in the two states. The inter-arrival time of setpoints follows a Poisson process with rate λ_n, when the replica is in the non-faulty state; and a rate λ_f, when the replica is in the faulty state. As the delays in the faulty state are larger, the frequency of setpoints in the faulty state is lower, *i.e.,* $\lambda_f < \lambda_n$. Furthermore, the probability that a setpoint issued by a controller replica takes more than τ, *i.e.,* the probability that a computation results in invalid setpoints is θ. Then, the stationary probability of a replica being in the non-faulty state is $\pi_n = \frac{\lambda_f(1-\theta)}{\lambda_f(1-\theta)+\lambda_n\theta}$; and the average rate of sending setpoints is $\lambda_0 = \lambda_f\,(1-\pi_n)+\lambda_n\,\pi_n$.

As in previous chapters, the communication network is assumed to be probabilistic synchronous [58] with a loss probability p and the maximum one-way network latency of the messages received messages δ_n.

Lastly, we assume that at least one of the PAs is non-faulty and capable of receiving setpoints. A CPS without any non-faulty PAs would remain uncontrolled with or without Axo, and is thus uninteresting. Note that additional non-faulty PAs increase the chances of detection and recovery, thereby improving the derived bounds.

Analytical Results

Theorem 6.5.3 (Delay-Fault Recovery). *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 as follows:*

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

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

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

where,

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

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.

$$\begin{aligned} &\mathbb{P}^*_{det}(t_1) = \mathbb{P}(C_0 \text{ issuing } i \geq N \text{ setpoints in } [0, t_1] \text{ that are} \\ &\text{received by } C_1) = 1 - \sum_{i=0}^{N-1} \frac{(\lambda_f (1-p)^2 t_1)^i e^{\lambda_f (1-p)^2 t_1}}{i!} \end{aligned} \tag{6.6}$$

Equation 6.6 is based on the model of the controller described in the previous section: it gives the cumulative distribution function (CDF) of a Poisson distribution, where the parameter 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. N can be derived from α, H_{ext}, and H_{max}, from Algorithms 6.4, 6.5, and is as mentioned in statement of the theorem.

The PDF of the above expression can be obtained by taking the derivative, resulting in the Erlang distribution.

$$\mathbb{P}_{det}(t_1) = \frac{d}{dt_1} \mathbb{P}^*_{det}(t_1) = \frac{(\lambda_f (1-p)^2)^N t_1^{N-1} e^{-\lambda_f (1-p)^2 t_1}}{(N-1)!} \tag{6.7}$$

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 $\delta t = t_2 - (t_1 + d)$.

$$\begin{aligned} \mathbb{P}_r(\delta t) &= \mathbb{P}(C_0 \text{ receives one reboot message in } [t_2, t_2 + dt]) \\ &= \frac{d}{d\delta t}(1 - e^{\frac{-(1-p)\delta t}{T_r}}) = \frac{1-p}{T_r} e^{\frac{-(1-p)\delta t}{T_r}} \end{aligned} \tag{6.8}$$

Equation 6.8 can be obtained by modeling the process of receiving reboot messages (Algorithm 6.6) 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 of the Poisson process. This approximation facilitates the derivation of the above expression.

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

$$\begin{aligned} \mathbb{P}_{nf}(\delta t) &= \mathbb{P}(C_1 \text{being non-faulty at } t = 0 \text{ and in } (0, \delta t]) \\ &= \pi_n e^{-\lambda_n \theta \delta t} \end{aligned} \tag{6.9}$$

In Equation 6.9, we consider 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 6.7, 6.8, 6.9, 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:

$$\mathbb{P}_d^l(t) = \int\limits_{t_1=0}^{t-2\delta} \mathbb{P}_{det}(t_1) \int\limits_{t_2=t_1+2\delta}^{t} \mathbb{P}_r(t_2 - t_1 - 2\delta)\mathbb{P}_{nf}(t_2 - 2\delta)\, \mathrm{dt}_2\, \mathrm{dt}_1$$

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 in the actual probability. Therefore, the upper bound is justified. It is given as follows:

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

$$\mathbb{P}_1(t) = \int\limits_{t_1=0}^{t} \mathbb{P}_{det}(t_1) \int\limits_{t_2=t_1}^{t} \mathbb{P}_{rec}(t_2 - t_1)\,\mathrm{dt}_2\,\mathrm{dt}_1$$

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

Theorem 6.5.4 (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 ($\mathbb{P}^l_c(t)$) and upper bound ($\mathbb{P}^u_c(t)$) on the probability that it is recovered by time t is given as follows:*

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

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

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

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

where,

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

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

Proof. We first derive the following probabilities.

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

$$\begin{aligned}\mathbb{P}_a(\delta t) &= \mathbb{P}(C_0 \text{ issues a setpoint of conception time } t_0 - \delta t, \text{which is received by } C_i) \\ &= \lambda_0(1-p)^2 e^{-\lambda_0(1-p)^2 \delta t} \end{aligned} \tag{6.10}$$

Equation 6.10 considers the controller model described earlier and uses the time-reversal property of Poisson processes.

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

$$\begin{aligned}&\mathbb{P}_c(\delta t, g) = \mathbb{P}(C_j \neq C_0 \text{ conceivs a setpoint at } t_1 - d, \text{the report of which is received by } C_i \text{ at } t_1) \\ &= \begin{cases} 0 & \delta t < \tau_c \\ (g-1)\lambda_n(1-p)^2 e^{-(g-1)\lambda_n(1-p)^2(\delta t - \tau_c)} & \delta t \geq \tau_c \end{cases}\end{aligned} \tag{6.11}$$

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.

Next, we derive a lower bound and upper bound on the probability of recovering from a crash fault by using Equations 6.10 and 6.11. We will also use $\mathbb{P}_r$ and $\mathbb{P}_{nf}$ from Equations 6.8 and 6.9, respectively.

The conditions for lower and upper bound are similar to those presented in the proof of Theorem 6.5.3. 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.

$$\mathbb{P}_c^l(t) = \int\limits_{t_0=\max(0,\tau_c-t)}^{\tau_c} \int\limits_{t_1=\tau_c-t_0}^{t-2\delta} \int\limits_{t_2=t_1+2\delta}^{t} \mathbb{P}_a(t_0)\mathbb{P}_c(t1+t_0,2)\times$$
$$\mathbb{P}_r(t_2-(t_1+2\delta))\mathbb{P}_{nf}(t_2+(t_0+2\delta))\,\mathrm{d}t_2\,\mathrm{d}t_1\,\mathrm{d}t_0$$

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

$$\mathbb{P}_c^u(t) = 1-(1-\mathbb{P}_3(t))^{g-1}$$
$$\mathbb{P}_3(t) = \int\limits_{t_0=\max(0,\tau_c-t)}^{\tau_c} \int\limits_{t_1=\tau_c-t_0}^{t} \int\limits_{t_2=t_1}^{t} \mathbb{P}_a(t_0)\mathbb{P}_c(t1+t_0,g)\times\mathbb{P}_r(t_2-t_1)\,\mathrm{d}t_2\,\mathrm{d}t_1\,\mathrm{d}t_0$$

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

Experimental Validation

We use three controller replicas ($g = 3$): each with a test controller and the controller library, and one PA with the PA library. The setup includes two computers connected by an Ethernet link. For the PA, we use a Lenovo T410 laptop with a 2.67 GHz Intel Core i7 processor with 4 GB RAM running a 64-bit Ubuntu operating system. For the controller replicas, we use 64-bit Ubuntu Virtual Machines that are each configured with 1 GB RAM using VirtualBox. The virtual machines run on a MacBook Pro with MacOS 10.10.5, a 2 GHz Intel Core i7 processor and 16 GB RAM. All three controller replicas are configured to be in the same LAN that is bridged to the physical Ethernet interface in order to communicate with the PA.

The times of computation of the test controller are drawn from a Poisson process with a rate $\lambda_n = 1/100\ s^{-1}$ when the controller is non-faulty and $\lambda_f = 1/200\ s^{-1}$ when the controller is faulty. The validity horizon τ_o is $17\ ms$, the upper-bound on the masker's computation time is $\delta_m = 0.1\ ms$, the upper-bound on the synchronization inaccuracy is $\delta_s = 1\ ms$, and the upper-bound on the network latency is $\delta_n = 2\ ms$. Lastly, the threshold, after which an inactive controller is considered crash faulty, τ_c is

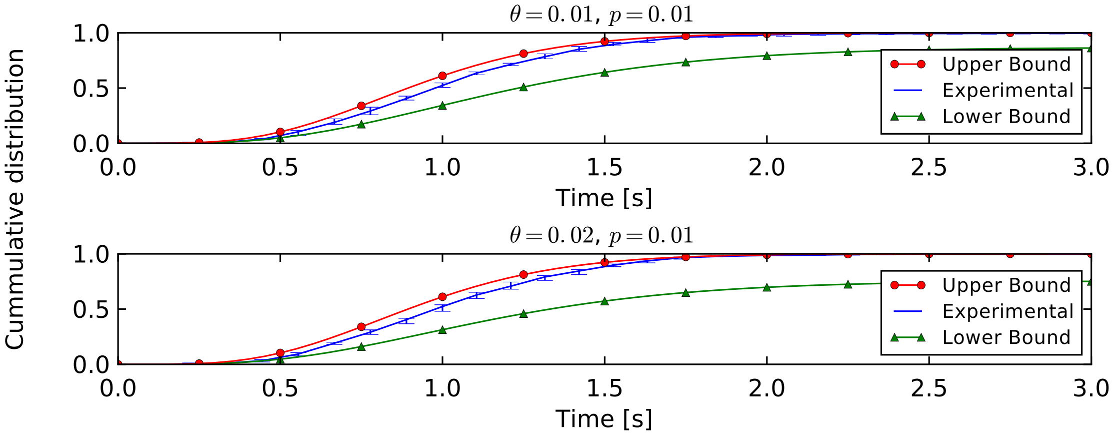


Figure 6.2 – Time to recover from delay-faults for varying θ

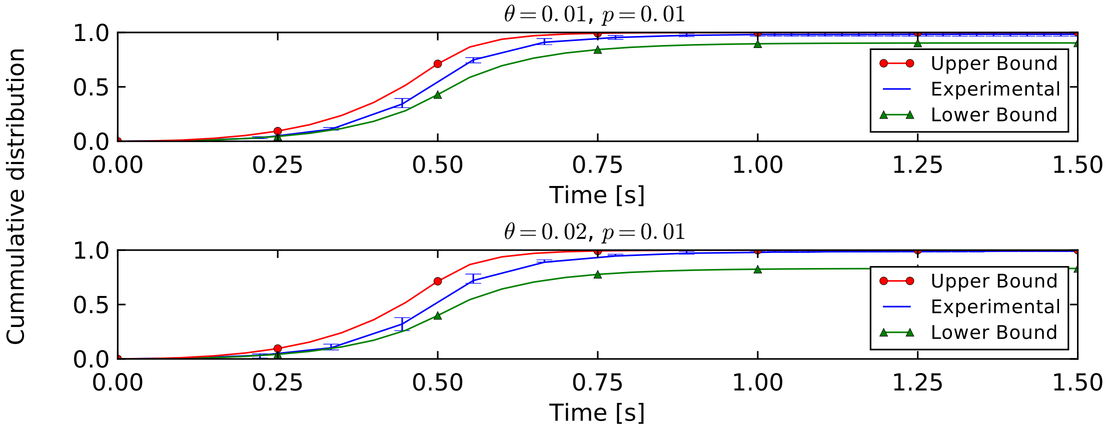


Figure 6.3 – Time to recover from crash-faults for varying θ

taken as 500 ms.

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. The time at which C_0 starts being faulty and the time at which it is recovered is recorded, and their difference is reported as the recovery time. We repeat each experiment 10 times to obtain the confidence intervals on the various quantiles of the distribution of recovery times.

We performed experiments for different values of p and θ. We noticed, from both our experiments and the analytical lower and upper bounds, that the packet loss probability p did not have a major effect on the probability of detection in the range of 0% and 2% loss probability (a realistic range of loss probabilities for CPSs). This result shows that the detection and recovery algorithms of Axo are resilient to network losses in this range.

However, the dependence on θ is significant. Figure 6.2 shows the results of the experimental simulation of a delay-faulty C_0, with $p = 1\%$ and $\theta = 0.01, 0.02$. Figure 6.3 shows the same for a crash-faulty C_0. We notice that the experimental distribution is within the analytically computed bounds. 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.

6.6 Case Study I: COMMELEC

Here, we study the effect of the reliability mechanisms proposed in this thesis namely (Quarts, Quarts+ and Axo) on the control of an electric grid by the COMMELEC [4] CPS. We give a brief background on COMMELEC in Section 6.6.1, followed by the description of the test-setup in Section 6.6.2. Then, we present results from two experiments that demonstrate the importance of fault detection and recovery in Section 6.6.3, and the effect of inconsistency in Section 6.6.4.

6.6.1 COMMELEC Background

In COMMELEC, the electric grid is controlled by a central controller, called the GA. COMMELEC also includes resource agents that send the state of a single resource, such as a battery or a PV. These form the PAs of the COMMELEC CPS as shown in Figure 6.4. It also comprises PMUs that periodically (every 20 ms) stream the state of the grid in the form of voltage and current phasors. The PMUs constitute the asynchronous sensors in the COMMELEC CPS. In Figure 6.4, we have a COMMELEC CPS with one GA, one battery resource, one PV resource, one load resource, and three PAs. Using the measurements from the PAs and the advertisements from the sensors, the GA computes and issues setpoints. The control round lasts roughly 100 ms.

The main goal of the GA is to maintain the grid in a feasible state, i.e., respect the voltage limits at the buses in the electric grids and the ampacity limits in the lines. The GA also implements one or more auxiliary policies such as providing primary frequency support, acting as a virtual power plant, or following a dispatch signal.

The GA follows the model of the controller presented in Algorithm 3.1 that transforms to 6.1, with the inclusion of the reliability mechanisms. It is implemented in C++, uses third-party software libraries, and runs on an off-the-shelf Linux computer. It is susceptible to both crash and delay faults, as shown in Section 1.2.2.

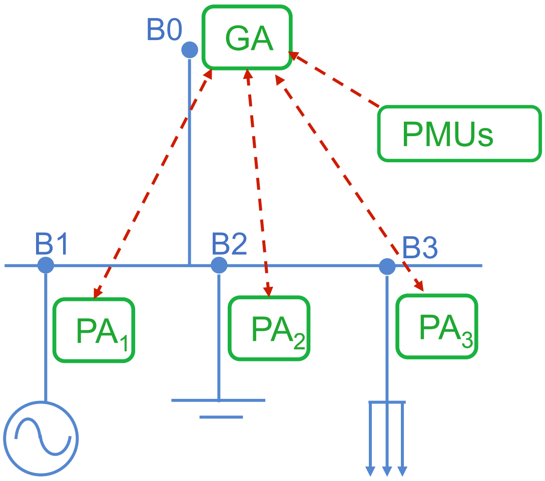


Figure 6.4 – COMMELEC architecture with 3 resources and PAs

6.6.2 Test Setup

We used the 13-bus CIGRÉ low-voltage benchmark grid, with three resources: a 15kW uncontrollable load of heaters in a building, a 25kW/25kWh battery storage, and a 20kW uncontrollable PV as shown in Figure 6.5. There are five PMUs streaming voltage and current phasor measurements to the GA, at different locations in the grid. In this case study, we configured the GA to assist autonomy, i.e. the power imported from, and exported to, the upper-level grid should be minimized at all times.

6.6.3 Importance of Fault-Tolerance

In this experiment, we use COMMELEC with two physical replicas, each running one instance of the GA on an off-the-shelf Scientific Linux (version 7.1). We use the on-campus microgrid facility at EPFL [92]; it is a 1-1 scale replica of the the 13-bus CIGRÉ low-voltage benchmark grid shown in Figure 6.5.

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

In order to demonstrate the fault tolerance of Axo, we artificially reduce τ_o to 7 ms, increasing the number of faults. We use Axo with two controller replicas (C_1, C_2). The time after which a replica is considered crash faulty (τ_c) is 500 ms. The time between successive recovery messages (T_r) is 1 ms. For time synchronization between the

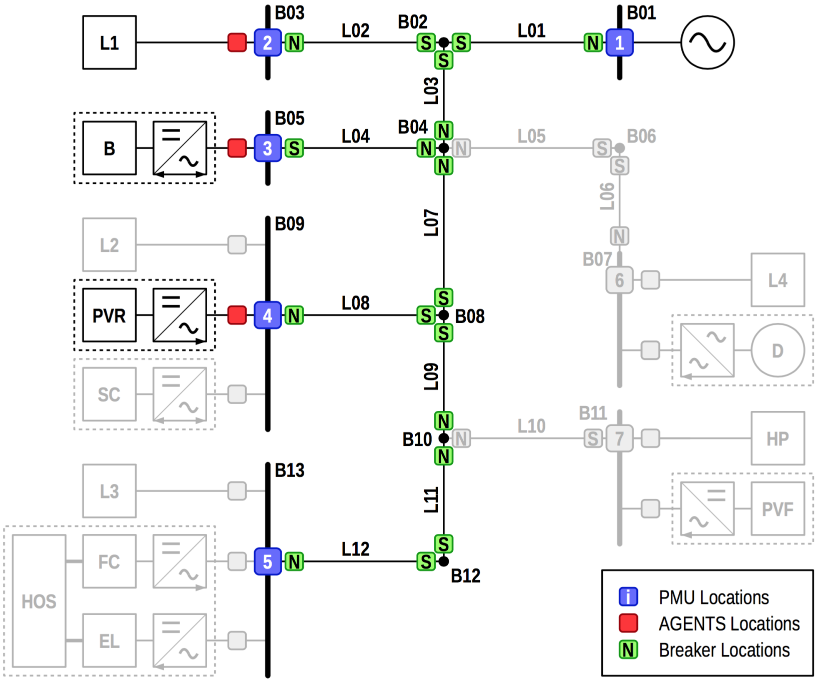


Figure 6.5 – 13-bus CIGRÉ low-voltage benchmark grid controlled using COMMELEC to provide autonomy by minimizing the total export or import of power at bus B01. Grey areas are unused

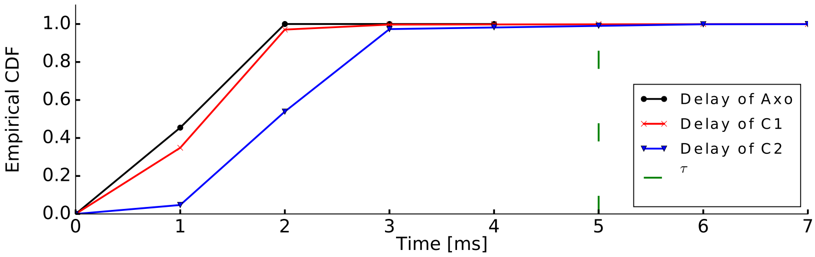


Figure 6.6 – Timeliness guarantee of Axo

replicas and the PAs, we use PTP that has a synchronization inaccuracy (δ_s) of 1 ms. Lastly, the upper bound on the computation time of the masker (δ_m) is 0.1 ms. This leaves us with $\tau = 4.9$ ms.

Figure 6.6 shows the empirical CDF of delays of setpoints sent by C_1 and C_2, measured at the masker. It also shows the effective delay of setpoints at the PAs, after the unsafe ones were discarded by the Axo masker. We observe that, although the setpoints sent by controllers have delays > 4.9 ms $(= \tau)$, the setpoints eventually received at the PAs are all valid, i.e., have an end-to-end delay < 4.9 ms < 7 ms $(= \tau_o)$, thereby demonstrating the timeliness property of Axo.

Additionally, the set of controller replicas is said to be available if the PAs receive a setpoint every 100 ms. We find that the availability with C_1 alone is 97.36% and with C_1 alone is 97.54%, which amounts to about 38 and 35 minutes of downtime. With Axo, however, the availability was found to be 99.86%, which is only 1.5 minutes of downtime. In these experiments, the controller replicas were recovered 38 times, thereby demonstrating the importance of fault recovery in providing high availability.

6.6.4 Effect of Inconsistency

In this experiment, we study the effect of inconsistent setpoints on the autonomy performance of COMMELEC by experimenting with and without Quarts+. Inconsistent setpoints might cause undesirable effects in the the grid and the resources, hence are unsafe to be tested in the real-grid. We used T-RECS [187], a virtual commissioning tool for studying the control performance of CPSs for electric grids in the presence of network and software non-idealities. T-RECS has been validated to give results that are reproducible on the campus microgrid, as shown in [187]. Therefore, the results obtained from T-RECS can be considered to be similar to those that would have been obtained from the real grid.

In order to highlight the effect of inconsistency, we performed stress tests, in which we subject the GA replicas to artificial delays in computation and a burst of extreme network conditions of 10% loss rate. The aim is to understand the effect of inconsistency on a CPS, without running experiments over several days or weeks, as would be required due to the low probability of inconsistency observed in Section 5.7.1.

In each experiment, we record the power at the slack bus (connecting the microgrid to the upper-level grid), and compute the energy mismatch in order to quantify the error in the control performance of the COMMELEC GA. The energy mismatch, in this case, is the integral of the absolute value of the power profile observed at bus B01. We also compute the maximum deviation from zero power at any given point.

First, we benchmark the performance of a single non-replicated GA that is not exposed to delay faults or message losses. Then, we compare this to two cases, one in which the two replicated GAs do not perform agreement, and one in which they perform agreement with Quarts+.

We observe, in Fig. 6.7, the energy mismatch of the three scenarios discussed earlier. The ideal case provides the benchmark tracking-performance level, and it encounters some mismatch due to the unpredictable nature of real-time PV production and load consumption. For the entire duration of the experiment, Quarts+ shows a better performance than the non-agreeing case.

Furthermore, we recorded the worst-case deviation from the tracking signal (zero

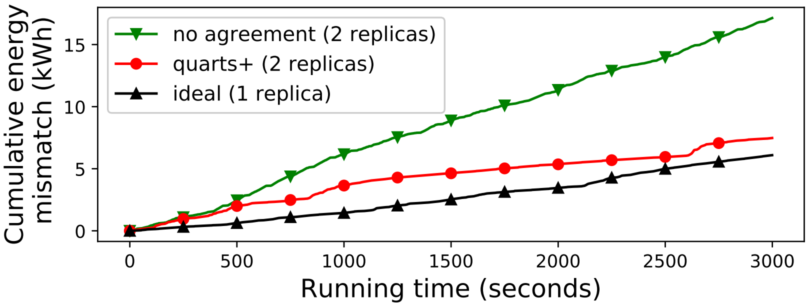


Figure 6.7 – Energy mismatch over time in COMMELEC

power) for each scenario. We observe that the non-agreeing case has a value of 33 kW: this exceeds the ampacity limit of the line connected the microgrid to the upper-level grid. Note that a single replica would never compute a set of setpoints that results in such a violation. The inconsistency between the replicas producing different outputs led to this. We do not observe such violation when using Quarts+.

Finally we observed, in 4% of the rounds, potential inconsistencies that would have arisen had we not performed agreement on advertisements. This highlights the importance of including asynchronous sensors in the agreement protocol.

6.7 Case Study II: Inverted Pendulum

In this section, we demonstrate how applying Axo affects the stability of an inverted pendulum system, a common control problem as seen in a motorized two-wheeled, self-balancing personal transporter such as a Segway. We use the example in [188], of an inverted pendulum mounted on a motorized cart, in an LQR controller attempts to maintain the pendulum upright by modulating the voltage of the motor, thus changing the force applied on the cart. The controller periodically receives the state of the pendulum. This state includes the position of the pendulum on the x-axis (x), the pendulum angle measured from the y-axis (ϕ) and the derivatives of the x and ϕ. The state information forms the measurement that is sent by the pendulum that is the PA. The goal of the LQR controller is to keep the pendulum upright with $\phi = 0°$ and pivoted at $x = 0\, m$.

For the experiments, we use Mininet [97], where the LQR controller and the PA are hosted on different nodes. The controller and the PA communicate by using a communication network with a loss probability $p = 0.001$ and the upper bound on the one-way delay $\delta_n = 0.5\, ms$. The controller operates at 100 Hz, resulting in a control cycle of 10 ms. Figure 6.8 shows a snapshot of the inverted pendulum setup from the experiments. We record the x and ϕ in each experiment.

To highlight the effect of delay on the stability of the pendulum, we evaluate the

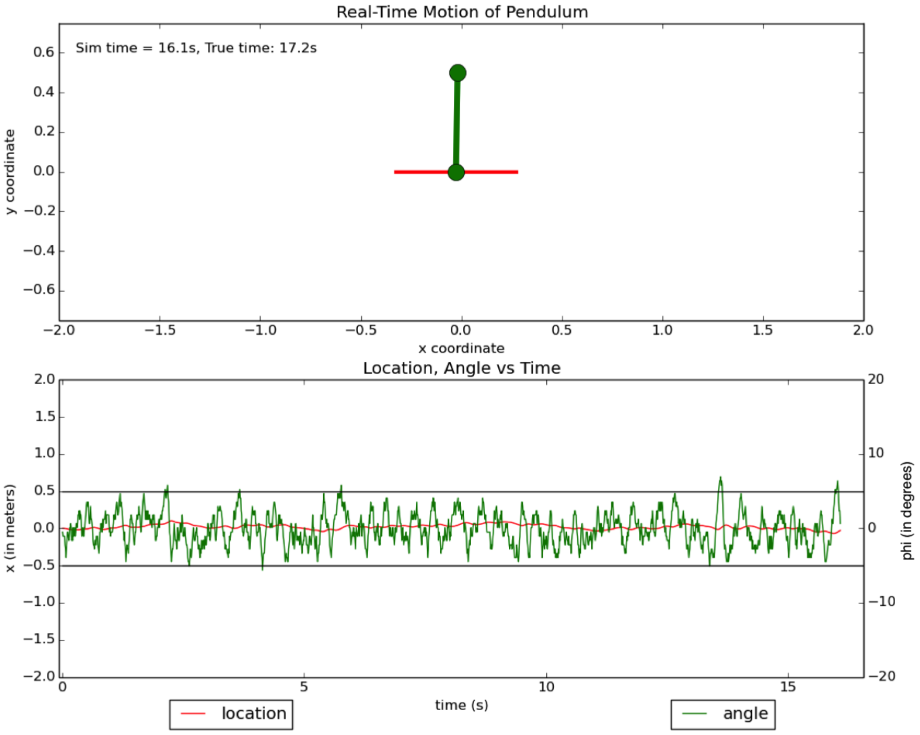


Figure 6.8 – Snapshot of the inverted pendulum from experiments

step response of the CPS with a single controller, when a step of $1\ N$ is applied as an external force. Figure 6.9 shows the step response for different values of controller delay. We see that ϕ and x experience a higher overshoot and a longer settling 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, hence the applicability of Axo in masking, detecting, and recovering from delay faults.

Next, we study the impact of Axo on the stability. For this, we use two replicas of the LQR controller as shown in Figure 6.10. As the CPS consists of only one PA, each controller issues only one setpoint in a round. Thus, this CPS does not require Quarts. Hence, we use only Axo for these experiments. The LQR controller is written in Python. Therefore, we created an Axo API in Python and integrated it with the LQR controller on each replica. The Python API is only 15 lines of code, thereby indicating the ease of deployment of Axo with a new controller.

We evaluate three metrics: the instability rate, MTTI, and MTTF. Instability rate is the fraction of the time the pendulum experiences an overshoot ($\phi > 20°$ 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 ($\phi > 35°$).

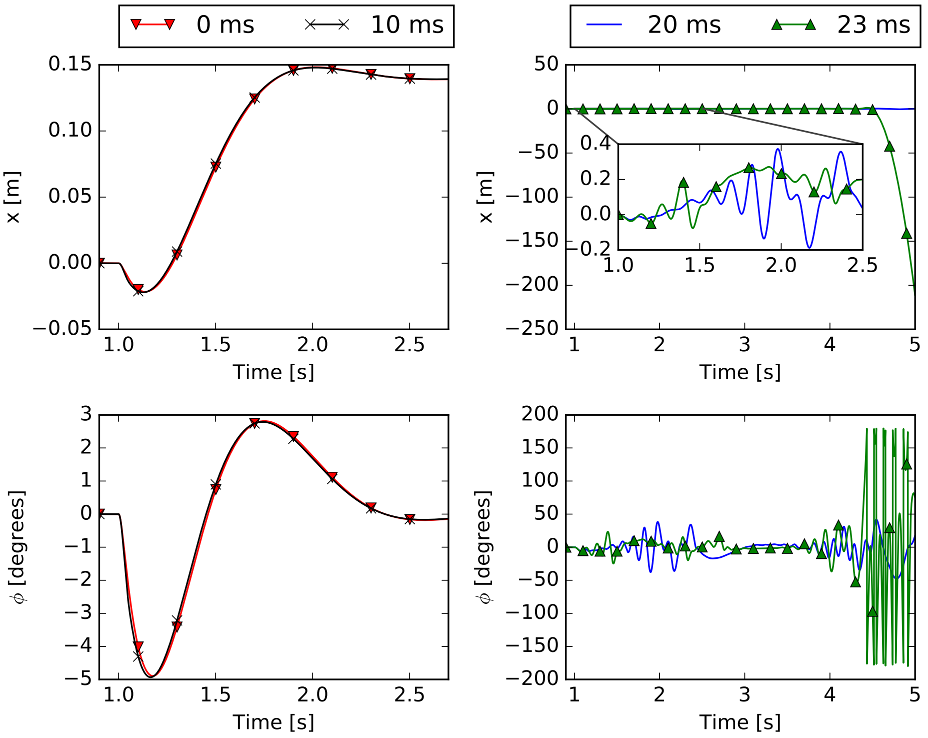


Figure 6.9 – Step response of the pendulum with a single controller for different values of controller delay

We use a bursty delay-fault model. Crash faults are not introduced because the system of two replicated controllers without Axo fails to control the pendulum after some time, as the replicas are not recovered. The delay of each controller 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.

Figure 6.11 shows the additional stability brought about by using Axo for a representative fault-scenario ($\theta_d = 10^{-3}$). Table 6.1 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.

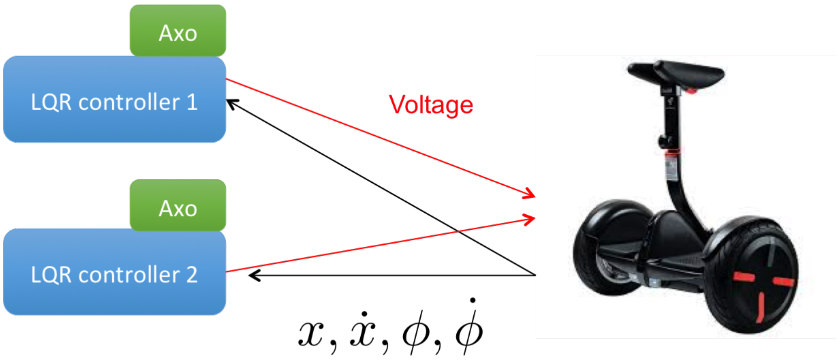


Figure 6.10 – Illustration of the inverted pendulum CPS with two controllers and one PA (the pendulum)

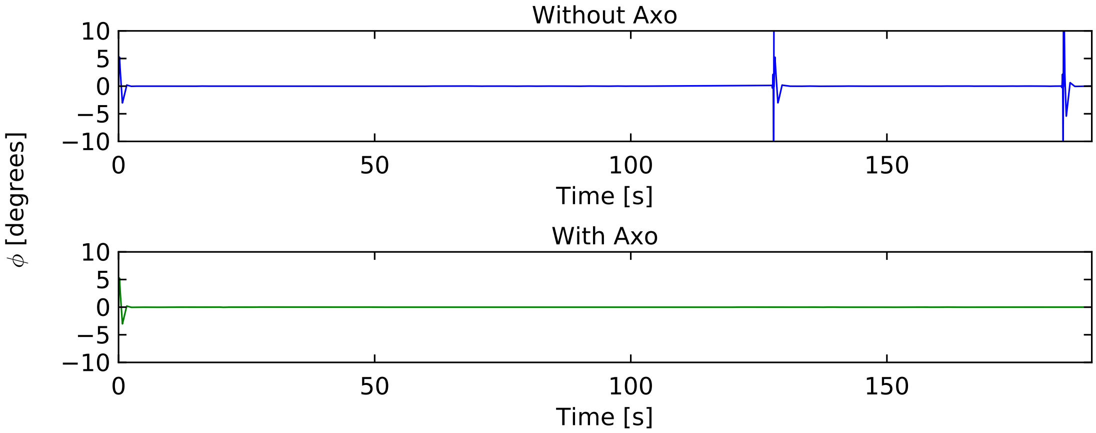


Figure 6.11 – Stability of the pendulum with a replicated controller

Scenario (θ_d)	Instability (%)		MTTI (s)		MTTF (s)	
	No Axo	Axo	No Axo	Axo	No Axo	Axo
#1: 1×10^{-3}	19.56	1.86	57.89	79.16	73.30	118.32
#2: 2×10^{-3}	23.93	2.78	25.33	31.70	22.29	47.06
#3: 5×10^{-3}	54.04	6.25	7.31	7.42	1.28	32.73

Table 6.1 – Instability of an inverted pendulum in selected scenarios with varying θ_d

6.8 Conclusion

In this chapter, we presented Axo, the first protocol for tolerating delay faults in real-time CPSs that use COTS-based hardware and software components. We describe the masking, detection, and recovery mechanisms for delay and crash faults. These mechanisms are designed to be soft state to enable the seamless addition of new replicas and removal of faulty replicas. Moreover, Axo is designed to be controller-agnostic, enabling easy deployment to a wide range of existing CPSs.

We formally proved that Axo guarantees the timeliness correctness property. In conjunction with the Quarts and intentionality clocks presented in earlier chapter, the consolidated design of a CPS with all the reliability mechanisms provides delay and crash fault tolerance. Specifically, this includes linearizability provided by Quarts and intentionality clocks, and timeliness provided by Axo. We also analytically characterized the time to recover a faulty replica by Axo and experimentally validated the evaluated expressions.

We presented an open-source implementation of Axo in C++ and an API that can be used by existing CPSs to instrument their controller in order to use Axo. We used this implementation to demonstrate the fault-tolerance properties of Axo through two case studies with real-world CPS namely, the COMMELEC CPS for real-time control of electric grids and an inverted pendulum CPS. In the first study, we deployed Axo and Quarts with COMMELEC in a full-scale microgrid test-bed on the EPFL campus and demonstrated that the resulting CPS can continue to perform the desired control even in the presence of delay and crash faults. We also demonstrated the importance of Quarts+ by showing how inconsistent setpoints issued by disagreeing controller replicas can adversely affect the control policies of the COMMELEC controller. In the second study, we showed that tolerating delay faults with Axo improves the stability of an inverted pendulum and increases its MTTI.

In this way, we designed reliability mechanisms for real-time CPSs, formally proved their correctness properties, and demonstrated their reliability properties through implementation and deployment with real-world CPSs.

7 The Quick Coordination Layer for Consistent-Controller Replication in SDN

Quick decisions are unsafe decisions.
- Sophocles, 450 BC

Challenge accepted!
- Quarts, 2018 AD

In the previous chapters, we designed reliability mechanisms for CPSs, *i.e.,* computer systems that control a physical process. Now, we notice that communication networks such as SDNs also fit the model of CPSs. The difference in SDNs is that the controlled process is not a physical process, rather the routing rules in the communication network. We find that SDNs conform to the model of the CPS presented in Chapter 3, as follows. The CPS controller is the centralized SDN controller and the PAs are switches. The switches send information about the events in the communication network to the controller in the form of measurements. The controller receives these events, computes routing updates and sends them as setpoints to the switches. The switches implement the routing updates in order to realize the desired communication policies in the network.

The SDN controller is a single point-of-failure that is susceptible to crash and delay faults and is often replicated for high availability. The replicated controllers need to coordinate in order to achieve control-plane consistency. This problem is similar to the agreement problem discussed in Chapter 5. Failure to perform agreement before installing the routing updates, potentially violating control-plane consistency, can result in the violation of certain safety policies of the SDN controller such as edge-isolation, edge-disjoint isolation. Alternatively, the conventional approach to achieving control-plane consistency uses consensus that adds some latency on average and a very high latency in the tail. In this chapter, we use Quarts (Chapter 5) and intentionality clocks (Chapter 4) to design QCL for achieving agreement with low latency-overhead. With QCL, we can quickly and safely install updates on the switches. This greatly improves both the average and tail-latency of replicated SDN controllers, in comparison with replication schemes that use consensus.

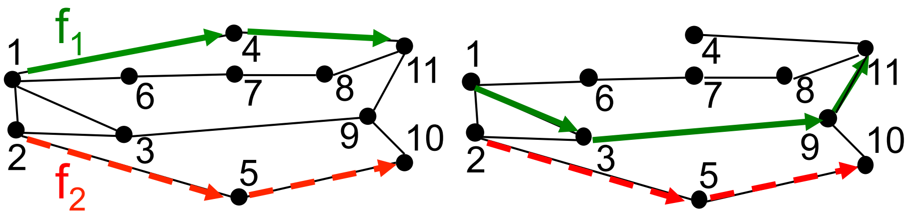
7.1 Introduction

SDN relies on a "logically-centralized" controller with a global view of the network to manage the network state and to implement networking policies. In practice, to address controller failures, a highly available logically centralized controller is implemented by replicating a single-image controller. Implementing a logically-centralized controller also involves using multiple controllers for scalability, e.g., by sharding the state between controllers, but this is orthogonal to the issue of state-machine replication for high reliability studied in our work. The challenge with replication is to ensure control-plane consistency with low latency.

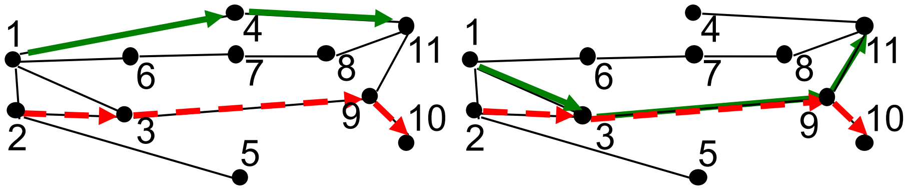
In the literature, two types of approaches are proposed to ensure control-plane consistency for replicated single-image controllers. On the one hand, there are high-latency consensus-based approaches that guarantee (strong) consistency, such as Onix [189], Ravana [190] and Onos [191]. On the other hand, we have SCL [31] that only provides eventual consistency guarantees [89], in order to provide low latency and high availability. Eventual consistency ensures that in absence of network events, all controllers will eventually have the correct view of the network, *i.e.,* topology, desired policy and the forwarding rules installed in the network will eventually be the same as those computed by a single-image controller that uses the same view. As a result, replication schemes that guarantee eventual consistency do not wait for *agreement* between replicas; they proceed with an update installation, as soon as one replica is aware of an event.

7.1.1 Problem

Although eventual consistency is sufficient to provide some safety policies such as way-pointing and node isolation [31], it cannot guarantee a large number of safety policies, such as edge isolation, node disjoint isolation, edge disjoint isolation [192], and other policies that refer to more than one flow at a time. These safety policies are required by applications that need isolated resources (switches or physical links) in order to exchange privacy-sensitive data while sharing network infrastructure, as traffic patterns can reveal information. Existing solutions fall short in cases when these policies also require low latency. For instance, isolated delivery of sensitive data is a



(a) Initial Flow-Paths.
(b) Flow-Paths by C_1.



(c) Flow-Paths by C_2.
(d) Installed Flow-Paths.

Figure 7.1 – Example of edge disjoint isolation violation

common security requirement for data exchange between financial institutions.

We show an illustration of the problem with eventual consistency and one such safety policy: edge disjoint isolation. Consider the network state shown in Figure 7.1(a). We have two flows, f_1 and f_2, starting from switches 1 and 2, and terminating at switches 11 and 10, respectively. Two controllers, C_1 and C_2, initially have the same view of the network. They both implement shortest-path routing, as well as edge disjoint isolation safety policy, *i.e.,*, the paths of flows f_1, f_2 should not share a common link. The initial routing paths of flows f_1 and f_2 are $1-4-11$ and $2-5-10$, respectively.

Let us assume that two events occur: links $1-4$ (event e_1) and $5-10$ (event e_2) break almost simultaneously (Figure 7.1(d)). Due to different network delays, C_1 first learns only about e_1, and C_2 first learns about e_2. They do not ensure agreement but rather each issues updates to modify the routing tables of the switches involved in the new paths and in the old paths. We assume that a switch uses the last installed update for a flow to serve packets of that flow. Moreover, in SCL [31], the controller uses the last known state from a switch to compute updates. As a result, C_1 and C_2 compute the new routes as shown in Figures 7.1(b) and 7.1(c), respectively, and issue updates for the switches. Each controller considers the edge disjoint isolation to be satisfied, but, an interleaving of the updates from C_1 and C_2 can violate the safety policy as shown in Figure 7.1(d). The state in Figure 7.1(d) could be reached by the following order of events: C_1 installs its updates on all involved switches, then C_2 installs its updates, but the update from C_2 for switch 1 is delayed due to e.g., network losses. Conversely, if the controllers had a consistent view of the network, *i.e.,* were in agreement, the computed routes could be: f_1 via $1-6-7-8-11$ and f_2 via $2-3-9-10$.

Additionally, existing eventually consistent schemes such as SCL [31] are unable to support reactive applications, such as NATs and firewalls, for which the first packet triggers an update installation and subsequent packets have to be handled by rules that causally follow from the first rule. Although vector-clock based labeling is a possible solution for achieving causal order, it can only provide partial order, as two vector clocks can be incomparable. Achieving total order would require the controllers to agree on the labels of sent updates. Hence, satisfying all safety policies in a general SDN and implementing all networking applications, requires consistency that typically relies on consensus. However, forming consensus is known to suffer from high latency. In this chapter, we achieve the best of both worlds, *i.e.,* guaranteeing consistency (like consensus mechanisms), while obtaining low latency (like eventually consistent schemes).

7.1.2 Our Approach and Challenges

We answer the question: *When can a controller replica proceed with computing the updates without agreement with other replicas and without violating control-plane consistency?* The challenge lies in the fact that even-though the ground truth of the SDN lies in the switches, it is difficult to consistently retrieve it on all the replicas; even in a failure-free scenario due to unavoidable network and processing delays.

In our work in Chapters 4 and 5, we have shown that for a special class of real-time CPSs, it is possible to forgo agreement in most cases and to use light-weight agreement instead of generic consensus. These systems have a known number of replicas controlling a known number of distributed agents; and the controller performs deterministic computation. Although these prerequisites are satisfied by SDN, some key challenges need to be addressed before the solutions can be applied.

In this work, we apply Quarts for agreement, as proposed in Chapter 5. Recall that Quarts is a low-latency agreement mechanism, which requires a consistent labeling of events in the SDN. Labeling is also needed to have a total order among updates sent, in order to implement reactive controller applications such as NAT. For this, we use intentionality clocks proposed in Chapter 4 that can express the relationships among SDN events better than the classic logical clocks such as Lamport clocks [85] or vector clocks [118]. However, intentionality clocks apply only to systems where the distributed agents are synchronous or round based. We extend the labeling solution to SDN, where the switches are asynchronous.

Lastly, Quarts performs best with regards to latency when all the switches advertise their state at the same time. This is not the norm in SDN, where only switches experiencing an event communicate with the controller. We address this problem with a probing-mechanism (details in Section 7.3.4).

7.1.3 Contributions

We present QCL, a distributed coordination layer connecting single-image controllers replicas and switches in SDN. QCL builds upon the eventually consistent scheme of SCL [31] and extends it with the Quarts low-latency agreement mechanism to addresses the aforementioned challenges. As a result, QCL guarantees strong control-plane consistency at a small cost in latency, compared to eventually-consistent solutions.

We formally prove that QCL can implement all the safety policies and networking applications like the underlying single-image controller. On top of that, we characterize the set of safety policies that can be satisfied by the eventually-consistent schemes and show that the corresponding set is larger for QCL. Finally, we prove that the worst-case latency overhead of agreement in QCL is upper-bounded.

We evaluate QCL through a simulation and study availability, response latency and consistency for various network policies. We compare the performance of QCL to that of SCL [31] and consensus-based schemes [190, 191] in both, datacenter and ISP topologies. We find that, with QCL, the median data-plane convergence time after an event is $0.6\times$ that of Ravana [190] and $0.5\times$ that of Consensus [191]. The latency improvement with QCL is more profound at the 99^{th} percentile of convergence time, with QCL being $160\times$ faster than Consensus and Ravana. When compared to the eventually-consistent scheme SCL, the median latency of QCL is double that of SCL and the tail latency is comparable. However, SCL can violate the safety policy of edge disjoint isolation with a probability in $[6 \times 10^{-7}, 4 \times 10^{-4}]$ for different values of network loss rate and number of replicas (see Section7.5). The low median-latency and drastically lower tail-latency render QCL an ideal replication scheme when strong consistency is required.

We show the applicability of our solution through a proof-of-concept implementation with the POX SDN controller [93]. We compare our implementation with SCL and show that our solution has comparable performance with SCL in both response and convergence time.

The rest of the chapter is organized as follows. We introduce the system model in Section 7.2. We present the design of QCL in Section 7.3 and elaborate the formal consistency and latency guarantees of QCL in Section 7.4. We evaluate QCL's performance in detail through simulation in Section 7.5, and we present implementation results in Section 7.6. Finally, we review related work in Section 7.7 and provide our conclusions in Section 7.8.

7.2 System Model

We consider that the SDN setup comprises multiple replicas of a single-image controller (e.g., POX [93], Ryu [94]) that communicate with and install updates on switches. In this section, we describe our assumptions about the communication network, the switches, and the requirements on the single-image controller. The network can drop, delay and reorder packets, and links in the network can fail at any point in time. Similar to previous chapters, the one-way propagation delay between any two end-points is bounded by δ_n – everything beyond is considered to be a delay fault.

A controller can suffer at any point in time from a crash fault or a delay fault. Byzantine faults are not considered. We require the following properties from the single-image controller.

- *Controller applications are deterministic.* If given the same network state, each replica will compute the same set of updates. This is the key requirement present in all the previous works [31, 162, 190].

- *Controllers trigger computation upon receiving an event.* Like in SCL, the controllers do not incrementally update the retained state. Instead, they recompute the state based on received messages from the switches. This makes the controller stateless and enables QCL to perform low-latency agreement.

We relax some constraints of SCL. In addition to proactive controller applications that compute updates, based on the network state, we enable reactive controller applications that respond to individual packet-ins, e.g., NAT and firewall.

As normally done in SDN deployments, we assume that switches communicate only with controllers and not with each other. Idempotent behavior is required when a switch installs updates, i.e., installing the same update twice has the same result as installing it once. QCL ensures control-plane consistency by performing agreement. However, for data-plane consistency, we rely on existing mechanisms for consistent-update installation, which are included in the design for completeness, but we claim no novelty. Specifically, we integrate the labeling-based mechanism proposed in [193].

7.3 QCL Design

QCL acts as a coordination layer for single-image controllers (e.g, POX, Ryu). The design of QCL is inspired by that of SCL [31]. Similarly to SCL, QCL consists of two components: the QSP and the QCP, shown in Figure 7.2. There is one QCP on each controller and one QSP on each switch. We use the term *proxy* to refer to either a QSP or a QCP. Algorithms 7.1 and 7.2 describe the design of QCP and QSP, respectively.

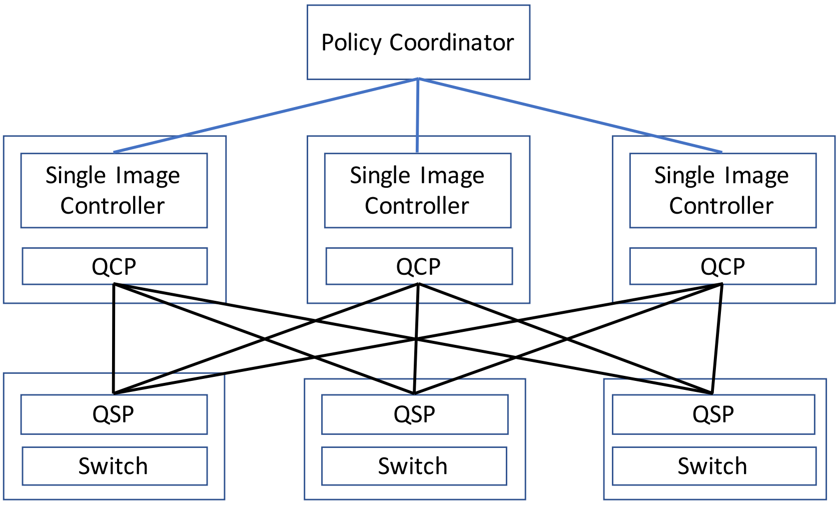


Figure 7.2 – Components in QCL

A QSP receives OpenFlow [194] events from its corresponding switch and then relays the state of the switch to the QCPs. The events can either be network events that depend on the topology of the network, such as a change of port status, or packet-ins. The two types of events are handled differently by the QSP, as described in Section 7.3.3. All events are labeled according to the labeling scheme described in Section 7.3.2.

A typical example of the working of QCP is as follows. When an event occurs at a switch, it is relayed to all the QCPs by the QSP of that switch. When a QCP receives this message, it probes all other QSPs for the most recent state of their switches. All QSPs that receive a probe, respond with their most recent state. The mechanism used by a QSP to export the state of a switch is described in Section 7.3.3. The QCPs wait for the responses from the QSPs, for a fixed time after sending out the probes. Once this time passes or all responses from all QSPs are received, the QCPs perform agreement on the received messages by communicating with each other using Quarts proposed in Chapter 5.

When two or more QCPs use Quarts, each QCP either decides on a set of inputs to be used in the update computation by its controller or decides not to let the controller compute. Furthermore, if two or more QCPs decide to compute, they will necessarily choose the same set of inputs. QCPs that decide to compute, send the chosen input upstream to their SDN controllers. Although it is possible that two QCPs decide differently, i.e, to compute or not to compute, these difference do not affect controller agreement and control-plane consistency. This is because a QCP deciding to not compute is equivalent to the controller not issuing any updates, thereby not conflicting with other compute updates. Moreover, agreement on input as opposed to agreement on the updates as done in consensus results in optimizations that enable low-latency response to network events by QCL. The agreement mechanism used by QCL, that uses Quarts at the heart of it, is given by Algorithm 7.3 and detailed in Section 7.3.4.

When a controller sends the computed updates to its QCP, the QCP relays them to the QSPs with appropriate labels. Additionally, in order to distinguish between messages from different controllers, each proxy is given a unique id. If a QSP does not receive an update in response to one or more of its events, either because of network losses or as a result of the controllers not computing due to Quarts, the QSP will trigger a new round of agreement after a timeout of T_{ret}. To this end, after the timeout, the QSP will send the same events with a higher label to the QCPs. Similarly, if an update received by the QSP does not use the most recent state of the underlying switch, then this update this discarded, and the same events are sent later with the higher label. This is discussed further in Section 7.3.5.

The number of QCPs (`n_qcp`) and QSPs (`n_qsp`) are assumed to be constant and are known to all the QCPs. Addition and removal of both switches and controllers, and planned policy changes are all assumed to be infrequent; they are handled by a policy coordinator, using a non time-critical mechanism such as two-phase commit [195] between the policy coordinator and the controller replicas.

7.3.1 QCL Messages & Functions

QCL uses three types of messages for communication between a QSP and a QCP, namely *Status*, *Probe* and *Update*. Each message has a label l. The other fields in the individual messages are described below.

Status<l, i, m> is the message used by a QSP to advertise the state of the corresponding switch, when an event occurs on the switch or is sent by a QSP in response to Probe. i is the id of the QSP and m is the body of the message.

Probe<l> is the message used by a QCP to query the current state of a switch from the QSP.

Update<l, ack, m> is a routing update to be installed on the switch, sent by a QCP to a QSP. ack indicates to the switch if its Status message was used in computation of this update, in which case, ack = True, else it is False. m is the body of the message that contains the actual routing updates.

In addition to the these messages, the QCPs also exchange other messages for agreement, as described in Section 7.3.4.

The four main functions performed by a QCP are (i) receiving status messages from QSPs and initiating Quarts (first function of Algorithm 7.2), (ii) performing Quarts to obtain the agreed upon status messages (Algorithm 7.2 line 11), (iii) sending the output of Quarts to the controller for the computation of the updates (second function of Algorithm 7.2, and (iv) sending the Update messages, including the updates received from their controllers, to the related QSPs (third function of Algorithm 7.2).

The four main functions performed by a QSP are (i) sending Status messages to QCPs when receiving an event from the switch (first function of Algorithm 7.1), (ii) responding to Probe messages received by QCPs with Status messages (second function of Algorithm 7.1), (iii) sending to QCPs Status messages if there exist events for which it has not received an update (third function of Algorithm 7.1) after a timeout of T_{ret}, and (iv) receiving and checking the Update messages and sending them to the switch (fourth function of Algorithm 7.1).

7.3.2 Ordering QCL Messages

Ensuring control-plane consistency requires that all the proxies have the same message ordering. The QSPs export a snapshot of the SDN to the QCPs using Status messages. A snapshot consists of the network topology and the state of the forwarding tables at all switches. The snapshot is recreated at the controllers from received Status messages. Each time a switch event occurs at a QSP, the snapshot of the SDN changes. Additionally, the snapshot also changes every time a new update is installed.

We employ a labeling mechanism that uses scalar logical-clocks to label all outgoing messages. The purpose of the labeling mechanism is to ensure that messages belonging to the same snapshot have the same label. For instance, if event e_2 occurs after e_1 on the same switch, then the corresponding Status messages must have different labels and the label of e_2 must be higher. This property is called local causality. If a controller computes two updates for two different switches in response to the same snapshot, then they must have the same label because they belong to the same snapshot.

To achieve this behavior, we use intentionality clocks (Chapter 4) that are adapted from Lamport clocks [85] to capture the notion of a snapshot in a system. Each QSP and QCP maintains a local logical-clock. Each outgoing message is tagged with a label. The label is obtained as the value of the local logical-clock right before sending the message. The logical clock is never decremented. It is incremented by the following rules as mentioned in Section 4.4: (1) each QSP increments its logical clock by one, when it receives an event from the switch or when it experiences a timeout (Algorithm 7.1 lines 7 and 27). (2) On receiving a message, the logical clock at each recipient is updated to the maximum of the local clock and the received label.

Thus, the labels of the outgoing messages from a QSP (Status) follow the same causal order as that of the events on the switch. This also enables QCL to support reactive controller applications such as NAT and firewall. The labels ensure that message ordering at the controllers is the same as the causal order at each switch. To ensure that the total order is maintained across QSP reboots, we augment the label with an epoch that is incremented at each QSP reboot. We assume that the epoch is stored in a persistent storage.

7.3.3 Exporting Switch State by QSP

Recall that the QSPs export the state of their switch by using Status messages so that the controllers can recreate the state of the network in order to compute updates in SDN. This approach is similar to SCL [31]. Although SCL allows only for network events that concern topology changes, we also allow for packet events (PACKET_IN in OpenFlow [194]) that require a per-packet update from the controllers.

The Status message after a network event includes the new state of the switch that comprises port status and flow-table entries. As a result, a new network event at a switch can overwrite an older network event. In contrast, packet events do not overwrite themselves. The Status after a packet event includes the OpenFlow PACKET_IN message, in addition to port status and flow-table entries as before. Moreover, all subsequent Status messages from this QSP must include this information until the corresponding Update has been installed on the switch. For this, we update the vector packet_events when a packet event occurs and an update is installed, in Algorithm 7.1 lines 9 and 37, respectively.

In Algorithm 7.1, we see that the current state of the switch that is exported in Status messages is only updated with the clock at lines 11, 20 and 28. This ensures that all outgoing messages from a QSP with the same label have the same body, a key property to ensure control-plane consistency as seen in Section 7.3.4.

7.3.4 Agreement at QCP

We use the Quarts agreement algorithm at the QCPs. Quarts performs agreement on the input of the QCPs, i.e., Status messages, as opposed to agreement on the output, i.e., Update messages, as done by consensus mechanism [115]. To do so, Quarts requires labeled messages from each QSP such that any two messages with the same label from the same QSP have the same body. In QCL, this is provided by the labeling scheme described in Section 7.3.2.

In order to minimize latency, the QCPs using Quarts first decide whether they need coordinate with others to reach agreement or can locally decide on the set of messages that will be agreed upon by other QCPs. The deciding condition is the presence of the most recent state from each QSP. When an event occurs on a switch in SDN, only that switch communicates with the controllers. In QCL, this would mean that only one switch will send a Status message with a new label to the QCP. In such a scenario, for each new label, the QCPs need to agree and cannot decide locally, as no QCP would have the Status from all QSPs corresponding to the new label. Hence, we introduce a probing mechanism before beginning agreement at line 13 in Algorithm 7.2. When a QCP receives a Status message, it sends a Probe message to all QSPs asking for their state corresponding to the label of the received Status. The QSPs reply with the Status message. This mechanism is similar to probing in SCL where the controller proxies periodically probe the switch agents, except that probing in QCL is lazy, i.e., initiated only when an event occurs. Lazy probing has a lower bandwidth requirement.

The probing phase lasts for a fixed time of $2\delta_n$, where δ_n is the bound on the one-way network latency as described in Section 7.2. At the end of the probing phase, Quarts is initiated with the vector of received Status messages (Algorithm 7.3 line 11) and the value of the local logical-clock. Quarts is round based, where a round is indicated by a label (e.g., C in Algorithm 7.3, line 11). Each round has two bounded-latency phases, namely collection and voting. During collection, each QCP collects missing Status messages from other QCPs. This also lasts for a period of $2\delta_n$, during which if a QCP misses a Status message a QS, then it queries other QCPs for this message, and receives the message from QCPs that have it. In the voting phase, each QCP shares a digest with its peers: this is a message indicating the QSPs from which it has received the Status messages. Then, each QCP uses the received digest to choose one digest by using a deterministic voting function. If, at a QCP, two digests are tied with the same number of votes, then the digest with the highest number of messages is chosen. The chosen set of messages M_{agreed} is forwarded to the respective controllers for computation.

Note that a possible outcome of the `collect_and_vote` function in Algorithm 7.3 line 11 that implements Algorithm 5.3 from Quarts is `success` = False. In this case, the QCP will not forward the Status messages to the controller (Algorithm 7.2 line 15) and prepare for the next round. Conversely, if `success` is True on two QCPs for in the same round, then Quarts [162] guarantees that their M_{agreed} is identical.

Recall that the controller is assumed to recompute the state of the network from the received inputs. Thus, if M_{agreed} does not have the Status message from a switch, this switch is treated to be a part of network partition. As we shall see in Section 7.4, this behavior does not impact safety policies such as edge-disjoint isolation and only affects liveness policies such as shortest-path routing. As similar phenomenon is observed in SCL when all the SCL probe replies from a switch are lost in a round.

As the controller is assumed to be deterministic (see Section 7.2), when two controllers compute updates with the same set of Status messages, the resulting Update messages will be identical. Also, as network partitions and failures are infrequent, most of the times, all QCPs have all the Status messages after the probing phase. Therefore, most of the times the QCPs will skip the collection phase and go straight to voting. In the special case of two controller replicas, if a QCP has all the messages after the probing phase, it skips both collection and voting, and forwards the messages to the controller without any added-latency due to agreement. This is possible because in rounds when the other replica has fewer messages, the replica with fewer messages will choose not to compute as a result of voting phase, because the digest corresponding to all messages has the highest priority.

Note that we use Quarts (without agreement on controller state) as described in Chapter 5, without any modifications to the collection and voting phase. In Chapter 5, the QCPs perform agreement only once, as it was designed for real-time CPSs, where performing agreement on messages after their real-time deadline is passed is superfluous. This however, is not true in SDN. Thus, if a QCP does not reach agreement after a first attempt, more attempts for agreement can be performed. As the QCPs exchange Status messages with each other in the collection phase, it increases the chances that all QCPs have the whole set of messages. Thus, performing more collection rounds improves the chances of success of the subsequent voting.

In Algorithm 7.2, for ease of presentation, the QCP does not process Status messages if the controller is busy computing. However, this can be optimized for even lower latency, by performing agreement among QCPs for a higher label while the controller computes updates for an older label.

7.3.5 Data-plane Consistency

The agreement mechanism described earlier guarantees control-plane consistency. However, in order to enforce the safety policies, the updates also need to be installed consistently on all the switches, i.e., data-plane consistency is needed. For consistent update-installation, we rely on prior work [193], where a packet is tagged with a label by the first switch in its data path and all subsequent switches serve the packet with a rule with the same label. Thus, at the time of installing the updates, we use the label in the Update messages sent by the QCP to tag the rules as belonging to a particular snapshot of the network.

Additionally, from Algorithm 7.1 line 35, we see that an Update is only forwarded to the switch, if it was computed with the most recent state of the switch. Although we have not found any safety policies that could be violated by not performing this check before installing the update, it certainly is a good practice for liveness. For instance, the switch could be instructed to forward packets on a port that is down, in which case the update will be installed and the QSP will not trigger new computations, as the `acked` field will be set to true, thereby violating connectivity (a liveness policy).

Algorithm 7.1: QSP Design

```
C ← reads epoch from persistent storage;
$S$ ←⊥;                                        // Current status
acked ← True;                                  // Was the last event acked?
Set event_timer to $T_{ret}$ ;                 // Timer to send a Status message
packet_events ← [ ] ;                          // Unacknowledged packet events
on event $e$ received from the switch
    $C \leftarrow C + 1$;
    if $e$ is a packet event then
        Add $e$ to packet_events ;
    end
    $S$ ← Updated current status ;
    $acked \leftarrow False$ ;
    Send $Status$< C, id, S>to all QCPs;
    Set event_timer to $T_{ret}$;              // Send Status if fires before Update
end;
on receive $Probe$<l>
    if $C \leq l$ then
        if $C < l$ then
            $C \leftarrow l$;
            $S$ ← Updated current status ;
        end
    end
    Send $Status$< C, id,  S>to all QCPs;
    Set event_timer to $T_{ret}$;
end;
on event_timer fires and acked = False
    $C \leftarrow C + 1$ ;
    $S$ ← Updated current status ;
    Send $Status$< C, id,  S>to all QCPs;
    Set event_timer;
end;
on receive $Update$<l, ack, m>
    if $C \leq l$ then
        $C \leftarrow l$;
        if ack then
            Send $m$ to the switch;
            Remove all acked events from packet_events ;
            acked ← True;
            Cancel event_timer;
        end
    end
end;
```

Algorithm 7.2: QCP Design

```
$C$ ← 0;                                    // Logical clock use to label events
S_Crt ← [ ];                                // Vector of current status messages
ACK ← [ ];                                  // ACKs to be sent to the switches
controller_free ← True;
compute ← False;
qcp_free ← True;
on reception of Status<$l$, $i$, $m$> and qcp_free = True
    if $C < l$ then
        $C \leftarrow l$;                   // New status; update clock
        S_Crt ← [ ];
        S_Crt[$i$] ← $m$;
        qcp_free ← False;
        Send Probe<$C$> to all QSPs ;
        success, S_Crt ← Agreement(S_Crt, $C$, n_qsp);
        if success then
            for $1 \le i \le$ n_qsp do
                if S_Crt[$i$] =⊥ then
                    ACK[i] ← False;
                else
                    ACK[i] ← True;
                end
            end
            compute ← True;
        else
            qcp_free = True;
        end
    end
end;
on controller_free = True and compute = True
    controller_free ← False;
    compute ← False;
    Send S_Crt to the controller;
end;
on reception of computed updates from the controller
    for each update $m$ for QSP $i$ do
        Send Update< $C$, $ACK[i]$, $m$> to QSP $i$;
    end
    qcp_free ← True;
    controller_free ← True;
end;
```

Algorithm 7.3: Agreement($\mathbf{S\_Crt}$, C, `n_qsp`)

```
repeat
    on reception of Status<$l$, $i$, $m$>
        if $C = l$ then
            $\mathbf{S\_Crt}[i] \leftarrow m$;
        end
        if $\sum_i(\mathbf{S\_Crt}[i] \neq \perp) = n\_qsp$ then
            break;
        end
    end;
until timer $2\delta_n$ expires;
success, $\mathrm{M}_{agreed} \leftarrow$ collect_and_vote($\mathbf{S\_Crt}$, $C$);        // Algorithm 10 of Quarts
return success, $\mathrm{M}_{agreed}$;
```

7.4 Formal Guarantees

In this section, we provide formal guarantees for QCL concerning control-plane consistency and enforcing safety policies. We define control-plane consistency and prove that QCL guarantees consistency. To this end, we first provide the definition of enforceable safety policies. This is followed by proving that QCL guarantees the enforceable safety policies.

Furthermore, as QCL requires agreement on events before deciding on updates for switches, when agreement is not successful, no updates are sent to the switches. However, events that are unacknowledged at a switch, trigger a retransmission in order to re-initiate a computation. Moreover, when an event is a network event, it might be overwritten by a more recent network event, e.g., a port down event followed by port up event is essentially a port up event. We call such events converse events. We show that all events are eventually acknowledged by QCL, unless their converse events take place.

Lastly, we compare the set of enforceable safety policies by QCL and SCL. We show that the set of safety policies that can be satisfied by QCL is a superset of the corresponding set of SCL.

7.4.1 Control-Plane Guarantees

Recall from Section 7.3, that the QCPs receive events as Status messages, perform agreement using Quarts, forward the vector of agreed Status messages to their respective controllers, receive the updates issued by their controllers and forward them to the QSPs. Consistency is said to hold in SDN for label r if and only if the updates with label r sent by QCPs to QSPs have the same value for the same QSP.

Theorem 7.4.1 (QCL Consistency). *The design of QCP presented in Algorithm 7.2 guarantees consistency in the presence of any number of delay- or crash-faulty replicas.*

Proof. From Lemma 5.5.2, Quarts ensures that if two QCPs, Q_i and Q_j, return `success` = True for label C (Algorithm 7.3 line 11), then they have the same M_{agreed}. Furthermore, as controllers are stateless and deterministic, agreement on the inputs used for updates' computation is sufficient to guarantee that the resulting updates received by Q_i and Q_j from the respective controllers are identical. Lastly, the value of the label does not change while the controller is computing, i.e., while `qcp_free` =False (Algorithm 7.2 lines 7, 12 and 38). Hence, the outgoing updates have the same label. □

In addition to providing control-plane consistency, it is desirable to have a low-latency overhead due to an agreement mechanism among QCPs, which is implemented by Algorithm 7.3. The latency overhead of a QCP is defined as the time spent by the QCP in Algorithm 7.2 line 14.

Theorem 7.4.2. *(Bounded Latency-Overhead) The latency overhead of a non-faulty QCP is less than or equal to* $7\delta_n$*.*

Proof. The duration of Quarts at a QCP is at most $5\delta_n$ (From Theorem 5.5.3). Moreover, the probing phase (Algorithm 7.3 lines $1-10$) takes at most $2\delta_n$. □

7.4.2 Enforceable Safety Policies

Safety policies must never be violated by a controller. We define a safety policy s as *enforceable* if there exists a single-image controller C that never violates s. Then, s is said be enforceable by C or C enforces s. Let S_C be the set of the enforceable safety policies by the single-image controller, C. Moreover, we say that s is enforceable by a logically centralized controller LC with one or more controller replicas of C, if LC never violates s. The set of all safety policies enforceable by LC is S_{LC}.

From SCL [31], we have that S_{scl} (the set of safety policies enforceable by SCL) includes a policy s if and only if it can be expressed entirely as a condition on exactly one path, i.e., the path violates or obeys the policy regardless of other existing paths in the network. For example, way-pointing states that the packets of a flow should follow a path that includes a given set of switches in the network. On the contrary, the definition of safety policies in S_C can concern multiple flows, hence multiple paths. For example, edge disjoint isolation for flows f_1 and f_2 states that their paths p_1 and p_2 should not share a link. Note that $S_{scl} \subset S_C$.

From prior literature on distributed systems [192], we know that when the control-plane is sharded across different controllers, it is impossible to enforce safety policies in S_C. However, we do not consider sharding in this chapter, thereby the impossibility result does not apply.

From the example of edge disjoint isolation in Figure 7.1, we notice that policies in S_C require control-plane consistency. Specifically, the controllers C_1, C_2 that are replicas of C, compute two different sets of paths for flows f_1, f_2, each set satisfying the safety policy. However, the true paths followed come from different controllers, and this interleaved set of paths does not satisfy the safety policy. In the absence of control-plane consistency, there needs to be a mechanism so that all switches, serve all flows, by using the updates from the same "chosen" controller; this is in fact a consensus problem and is faced with the same drawbacks of doing consensus in control-plane.

7.4.3 Safety Policy Guarantees

We prove that for any single-image controller C, each safety policy in S_C is enforceable when replicating C with QCL.

Theorem 7.4.3. *(QCL Safety) For any single-image controller C, each $s \in S_C$ is enforceable by a logically-centralized controller obtained by replicas of C using QCL, under the model described in Section 7.2.*

Proof. As s is enforceable by C, the set of updates for label r, U_r^0, computed by C, does not violate s. Let U_r^i be the set of updates computed by the i^{th} replica of C, C_i. Control-plane consistency implies that $U_r^i = U_r^0$ for any C_i. Furthermore, as updates are assumed to be idempotent, an installation of U_r^i for each C_i has the same result as an installation of U_r^0. □

Note that, as $S_{scl} \subset S_C$ and $S_{qcl} = S_C$, QCL enforces more safety policies than those enforced by SCL.

As QCL requires agreement on Status messages before deciding on updates for switches, when agreement is not successful, no updates are sent to the switches. Also, as explained in Section 7.3.3, a network event that overwrites another network event is denoted as converse event. For instance, a port up event overwrites a previous port down event. QCL ensures that events are not lost. This is an important property showing that consistent updates will be eventually computed, thereby providing liveness. For each unacknowledged event at a switch, computation of updates will be re-triggered by Status messages from its QSP, until an update is received.

Theorem 7.4.4. *(No Lost Events) QCL guarantees that every event will be eventually acknowledged except if the converse event takes place.*

This result derives from the fact that the QSP (i) sets the `acked` field to false when an event occurs (Algorithm 7.1 line 12), (ii) tracks if the last event emerged at the switch has been acknowledged (variable $acked$ at line 26 of Algorithm 7.1) and (iii) installs updates only if they have been computed accounting for the current state of the switch (checked at the receive Update function of Algorithm 7.1, line 35).

Moreover, in order to correctly implement reactive controller applications such as NAT and firewall, it is important to ensure that all packets in the flow (after the first) are handled by updates that causally follow after the first update. This requires that any two updates received at a switch (possibly from different controllers) are comparable, i.e., there must exist a total order among updates. The labeling mechanism and control-plane consistency in QCL guarantees that two updates u_1 and u_2 with labels l_1 and l_2, respectively, received at a switch, are identical if $l_1 = l_2$. Alternatively, if $l_1 < l_2$, then u_1

is time-wise before u_2, proving total order. Consequently, we conclude that QCL can be used to implement reactive controller applications.

7.5 Performance Evaluation

We use discrete-event simulation to compare the performance of QCL with other replication schemes [99, 161, 190, 191]. We begin by describing our simulation setup in Section 7.5.1, followed by description of the replication schemes in Section 7.5.2. Finally we present the results of the study on safety policy (edge disjoint isolation) and liveness policy (shortest path) in Section 7.5.3 and Section 7.5.4, respectively.

7.5.1 Simulation Setup

The data-plane is modeled as a graph of switches and the corresponding flows, and the control-plane is modeled as g independent controllers whose computation time is drawn according to the combined delay- and crash-fault model introduced in Section 5.6 for the performance evaluation of Quarts. This fault model is an adaption of the Gilbert-Elliot model [177], where the controller is either in the state of normal operation N or crashed C. When in state N, a controller can either finish its computation within a deadline τ or be delayed with probability θ_d and exponentially distributed delay time. When in state C, the controller is crashed and does not issue updates. The controller enters this state with a probability θ_c and returns to state N after a MTTR of $30\ s$. We take $\tau = 10\ ms$ as the mean update computation time of a non-faulty controller.

We use an out-of-band communication network between the data-plane and control-plane that is modeled as probabilistic synchronous [58]. The out-of-band network will drop or delay messages, with a probability p. Messages that are delivered have a maximum delay of δ_n.

We do two types of experiments. First, in order to highlight the additive value of QCL, we study the probability of a safety policy violation under eventual consistency, namely edge disjoint isolation. We use the topology shown in Figure 7.1(a). Second, we compare liveness policies and delay properties of the replication schemes by simulating two datacenter fat-tree topologies [96] with 8 and 16 port switches referred to as ft8 and ft16, respectively. We also simulate an ISP topology, AS 1221, mapped in the Rocket-Fuel Project [196]. We also vary the MTTF λ_f and MTTR λ_r of the links in the network.

7.5.2 Replication Schemes

We also study the following replication schemes.

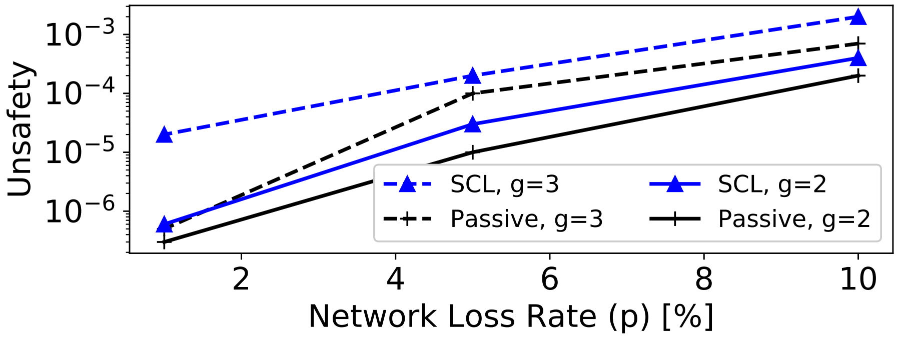


Figure 7.3 – Unsafety in SCL and Passive schemes. Unsafety of QCL is 0

Ravana [190]: Ravana guarantees consistency and uses primary-standby replication [99]. Before responding to an event, the primary replica synchronizes with the standby replicas using view-stamped replication [86]. The latency overhead in Ravana is caused by the state-synchronization mechanism. We implement the “totally ordered events” flavor of Ravana from [190] to compare with QCL that also provides total ordering among events.

Passive: We implement the “weakest” flavor of Ravana [190], which provides lower latency than Ravana, at the cost of eventual consistency. The state synchronization between the primary and standbys is performed using an unreliable mechanism, where the standbys might be out-of-sync for some time. If the failover from primary to standby happens during this period, then safety rules can be violated. Moreover, when the primary is detected as faulty by one or more backups, a new primary is elected using a leader election mechanism that employs consensus, which adds latency.

SCL [31]: SCL is an active replication scheme that provides eventual consistency. To provide low latency, the controllers forgo agreement. As a result, the response time of SCL is the smallest among all the replication schemes. Any possible state conflicts among the controller replicas are resolved via a periodic-gossip mechanism, and safety policies might be violated during the period of conflict.

Consensus [191]: To study the latency performance of consensus, we simulate ONOS [191]. However, instead of using Paxos [115] for consensus, we use Fast Paxos [161] that is optimized for lower latency. ONOS is a consistency-guaranteeing replication scheme, but suffers from higher response time due to consensus latency.

In QCL, if a switch does not receive updates for an event for a period $T_{ret} = 1\ s$ since the last time a Status message was sent, it resends the Status to all the controllers to re-initiate the computation of the updates. In SCL, the gossip mechanism among QCPs is performed every $T_{ret} = 1\ s$.

7.5.3 Safety Policy Results

We quantify the safety violation by SCL and Passive schemes when the safety policy is edge disjoint isolation for flows f_1 and f_2 in Figure 7.1. We notice that violation of the edge disjoint isolation can be observed in ISP-like topologies where redundant paths of different hop-count are available for each flow, as opposed to a datacenter topology where the redundant paths are mostly of similar hop-count.

In each iteration, we create two simultaneous link events: breaking of link $4-11$ and link $5-10$. Once the rules from all the controller replicas are installed, we check if the paths violate edge disjoint isolation. Unsafety is defined as the fraction of iterations in which edge disjoint isolation is violated. We use $\theta_d = 1 \times 10^{-3}$, $\theta_c = 1 \times 10^{-4}$, $\delta_n = 1\, ms$; and vary g as 2 or 3, and p as $1\%, 5\%$ or 10%.

Figure 7.3 shows the unsafety of SCL and Passive as a function of network loss rate p for different number of replicas g. We see that, for both schemes, unsafety increases with p. For SCL, unsafety with two replicas is 6×10^{-7} at $p = 1\%$ and 4×10^{-4} at $p = 10\%$, whereas for Passive it is 3×10^{-7} at $p = 1\%$ and 4×10^{-4} at $p = 10\%$. The unsafety of Passive is slightly lower than SCL because it does unreliable state synchronization as opposed to no-agreement in SCL. As noted earlier, a safety policy must *never* be violated by an SDN controller; QCL adheres to this property (Theorem 7.4.3). An unsafety of 4×10^{-4} with SCL motivates control-plane consistency.

The unsafety of SCL increases sharply with more replicas, as the probability of disagreement increases with more replicas. Hence, as more replicas are used for higher availability, the agreement between the replicas becomes critical.

7.5.4 Liveness Policy Results

We measure *response time* as the time between the occurrence of an event at a switch and the installation of the new update on that switch. During this interval, a new event might occur on this or another switch, i.e., the network snapshot is modified. This will trigger a computation of new updates. Hence, installation of an update at a switch does not necessarily imply that the required liveness (shortest path) policy is satisfied. Also, partial installation of updates i.e., at a subset of the switches involved after a network event does not necessarily imply that the required liveness policy is satisfied. We say that the data-plane has *converged* if both safety and liveness policies are satisfied after an event. Thus, *convergence time* is time between occurrence of an event and the convergence of the data-plane. We show the median and tail (99^{th} percentile) response and convergence times. We also measure *unavailability* as the fraction of time during which the shortest path liveness policy is violated for at least one flow.

In this section, we use two sets of parameters: `normal` and `aggressive`. In the `normal` setup, we use $p = 1 \times 10^{-3}$, $\theta_c = 1 \times 10^{-4}$, $\theta_d = 1 \times 10^{-3}$, $\lambda_f = 24$ hrs, $\lambda_r = 12$ hrs. In the `aggressive` setup, we use $p = 1 \times 10^{-2}$, $\theta_c = 1 \times 10^{-3}$, $\theta_d = 1 \times 10^{-2}$, $\lambda_f = 12$ hrs, $\lambda_r = 6$ hrs. With these values of λ_f and λ_r, in ft16 with 3072 links, there is an event every $15\ s$ and $7.5\ s$ in the `normal` setup and the `aggressive` setup, respectively. We vary the number of controller replicas g and network delay δ_n.

We use ft16 with the `normal` parameter set, $g = 2$ and round-trip time of $1\ ms$, i.e., $\delta_n = 0.5\ ms$ as a basic scenario to highlight the main findings. We do sensitivity studies for varying δ_n, g, different topologies and fault profiles. Figure 7.4 shows the ECDF of response times and convergence times, along results on unavailability for different schemes.

Finding 7.1. *The median response and convergence time with QCL is lower than that of consistency-guaranteeing schemes and higher than that of eventually-consistent schemes.*

Taking a closer look, we see from Scenario 1 in Table 7.1 that the median response time with QCL is $0.6\times$ that of Ravana and $0.5\times$ that of Consensus. To better visualize the improvement in the tail, we show on the log scale the CCDF of the response and convergence times in Figure 7.5. From Scenario 1 in Table 7.1, we see that the median and the tail response times of QCL are $1.2\times$ and $1.1\times$ that of SCL, respectively. In contrast, the tail response time of Ravana, Passive and Consensus is $2\times$ that of QCL, whereas the tail convergence time is *two orders of magnitude* ($\sim 160\times$) larger than that of QCL. This makes QCL clearly a better choice for datacenter networks that aim for low tail-latency [95].

Finding 7.2. *The tail response and convergence time with QCL is comparable to that of eventually-consistent schemes and drastically lower than consistency-guaranteeing schemes.*

The improvement in response time due to QCL is attributed to the unbounded latency overhead of some form of consensus mechanism used in Passive, Ravana and Consensus. Also, the underlying agreement algorithm of QCL, has a higher probability of reaching an agreement than Consensus.

Due to its lower response and convergence time, QCL has an availability three orders of magnitude higher than Passive, Ravana, and Consensus, as shown in Figure 7.4(c). The unavailability of QCL (7.1×10^{-6}) is comparable to that of SCL (5.9×10^{-6}); this highlights the efficacy of QCL in maximizing liveness and providing safety guarantees. The drastically high availability of QCL and SCL is also the reason for the low tail-latencies measured at 99^{th} percentile. Note that the tail measured at 99.99^{th} percentile and the maximum response time and convergence time for SCL and QCL also show a similar trend. Findings 1 and 2 are true also for other scenarios (see Tables

		Median and Tail Response Time [ms]				
#	Setup	SCL	QCL	Passive	Ravana	Consensus
1	Basic	0.73, 3.86	1.9, 4.56	1.3, 7.01	2.82, 9.35	3.77, 8.91
2	$g = 3$	0.57, 2.56	1.91, 3.65	1.27, 6.84	2.81, 8.29	3.65, 8.44
3	$\delta_n = 0.1\,ms$	0.55, 3.18	0.74, 3.57	1.05, 7.39	1.38, 6.61	2.19, 7.66
4	$\delta_n = 1\,ms$	1.05, 3.89	3.22, 6.1	1.48, 6.44	4.7, 11.49	7.01, 12.92
5	$g = 3, \delta_n = 1\,ms$	0.83, 2.84	3.42, 5.23	1.62, 7.77	4.65, 9.64	7.03, 12.37
6	ft8	0.79, 3.39	1.52, 4.1	1.27, 6.59	2.78, 7.84	3.72, 9.12
7	`aggressive`	0.76, 3.72	2.16, 5.3	1.33, 7.18	2.83, 8.33	3.77, 11.1
8	g=3, `aggressive`	0.59, 2.27	1.89, 3.92	1.27, 7.27	2.83, 7.76	3.65, 8.79

Table 7.1 – Median and tail (at 99^{th} percentile) response times for different scenarios. The column Setup shows the difference in each scenario from basic setup seen in Figure 7.4

		Median and Tail Convergence Time [ms]				
#	Setup	SCL	QCL	Passive	Ravana	Consensus
1	Basic	0.77, 5.01	1.93, 5.32	1.39, 815.79	2.89, 759.48	3.84, 855.36
2	$g = 3$	0.59, 2.71	1.92, 3.89	1.41, 858.44	2.91, 861.6	3.69, 791.75
3	$\delta_n = 0.1\,ms$	0.6, 3.71	0.75, 4.0	1.13, 825.56	1.42, 759.94	2.21, 829.39
4	$\delta_n = 1\,ms$	1.09, 4.53	3.25, 6.87	1.53, 688.12	4.82, 764.15	7.05, 854.38
5	$g = 3, \delta_n = 1\,ms$	0.86, 2.99	3.45, 5.68	1.71, 880.37	4.73, 875.17	7.08, 786.05
6	ft8	0.82, 3.88	1.54, 4.4	1.35, 749.94	2.83, 617.33	3.75, 813.46
7	`aggressive`	0.88, 6.07	2.28, 10.29	1.53, 887.37	2.98, 922.55	3.98, 971.8
8	g=3, `aggressive`	0.63, 2.93	1.97, 4.6	1.49, 912.17	2.96, 967.87	3.72, 865.85

Table 7.2 – Median and tail (at 99^{th} percentile) convergence times for different scenarios. The column Setup shows the difference in each scenario from basic setup seen in Figure 7.4

7.1 and 7.2). Next, we study the variation in the latency improvement of QCL with different parameters. Specifically,

Finding 7.3. *The latency improvement of QCL increases with more replicas and decreases with larger network delay and larger network size.*

From Scenarios 1 and 2 in Tables 7.1 and 7.2, we see that the response and convergence times of all schemes reduces with more replicas. The addition of replicas makes the controller more “available”, in the sense that the controller becomes more reactive [111]. As QCL can reach an agreement with fewer available replicas, its probability of agreement increases faster than Consensus and Ravana, by adding more replicas. Hence, with $g = 3$, the latency improvement increases on average by $2.2\times$ for tail response time and by $200\times$ for tail convergence time. A similar trend is seen in Scenarios 4 and 5.

In Scenarios 3 and 4, we vary the one-way network latency δ_n to $0.1\ ms$ and $1\ ms$, respectively, from $0.5\ ms$ in the basic scenario. As QCL performs agreement, its latency is limited by the latency of the out-of-band control network. Thus, we find that average convergence time of QCL increases as δ_n increases. However, the difference in increase is much lower than that of Ravana and Consensus because, on average, the agreement mechanism of QCL exchanges less traffic than Consensus and Ravana. As a result, for $\delta_n = 0.1\ ms$, we see that the tail-latency improvement with QCL, with respect to Consensus and Ravana, is $2.1\times$ for response time and $205\times$ for convergence time, on average. For $\delta_n = 1\ ms$, these drop to $1.9\times$ and $110\times$, respectively.

We also find that at lower δ_n QCL closely follows SCL, whereas the difference is larger with high values of δ_n. This is because SCL is even less network-intensive in responding to events as it does not perform agreement. Hence, the response time of SCL is very close to δ_n. As seen earlier, the performance of QCL can be further improved to be closer to SCL with more controller replicas. When more replicas cannot be used, a marginally higher convergence time is a small price to pay for guaranteed control-plane consistency. Alternatively, for SDN controllers that do not require control-plane consistency and do not implement applications based on packet events such as NAT and firewalls, SCL could be used instead of QCL, for lower latency. However, we remark that this seriously reduces the utility of the SDN controller.

As QCL performs agreement on `n_qsp` number of Status messages, its performance depends on the number of QSPs. We see from Scenarios 1 and 6 that, as the size of the network increases, the convergence time of QCL increases. The same effect is also observed in Consensus schemes, albeit for a different reason. In Consensus schemes in large networks, more events might occur while the controllers are agreeing, thereby increasing the convergence time.

As noted earlier, the performance of QCL is dictated by the performance of agreement algorithm Quarts. Thus, the convergence time of QCL depends on the success of Quarts, which depends on the fault parameters θ_c and θ_d, and the network loss rate p. Scenario 7 with the `aggressive` setup shows the impact of these parameters. We see that the performance of all schemes is affected by increased fault-rate; and the low-latency schemes QCL and SCL are affected the most. This can be remedied by appropriately dimensioning the degree of replication when the controller is expected to be faulty, as shown by the similarity in performance of Scenarios 1 and 8. For a more details on the performance of Quarts for a wider range of fault profiles, see Chapter 5.

ISP Topology In order to study the performance of QCL in a non-datacenter topology, we simulated the AS 1221 ISP topology with $\delta_n = 10\ ms$ in Scenario 7. Figure 7.6 shows that although the tail convergence time of QCL is increased when compared to SCL due to larger one-way latency, the large improvement when compared to

Consensus, Ravana and Passive is still preserved. Moreover, as ISP networks have multiple, redundant paths of different costs, there is a higher chance of violating safety policies, such as edge disjoint isolation, than in datacenter networks, as explained in Section 7.5.3. Therefore, such networks can benefit from both low latency and zero safety violation, offered by QCL.

Lastly, to characterize the quality of service of the shortest path policy, we measured *path-inflation*, i.e., the relative increase in number of hops in the observed path of the flow when compared to the actual shortest path. We did not observe any difference between the path-inflation with different schemes. This can be attributed to the large number of equal cost redundant paths in datacenter topologies. For the ISP topology AS1221 in setup 9, we notice that the 99^{th} percentile of the maximum path-inflation across all flows, was 25% for SCL and QCL, 75% for Ravana, and 100% for Consensus. The median value of the same metric was same for all schemes. We attribute the slightly better path-inflation for SCL and QCL due their higher availability and quick response.

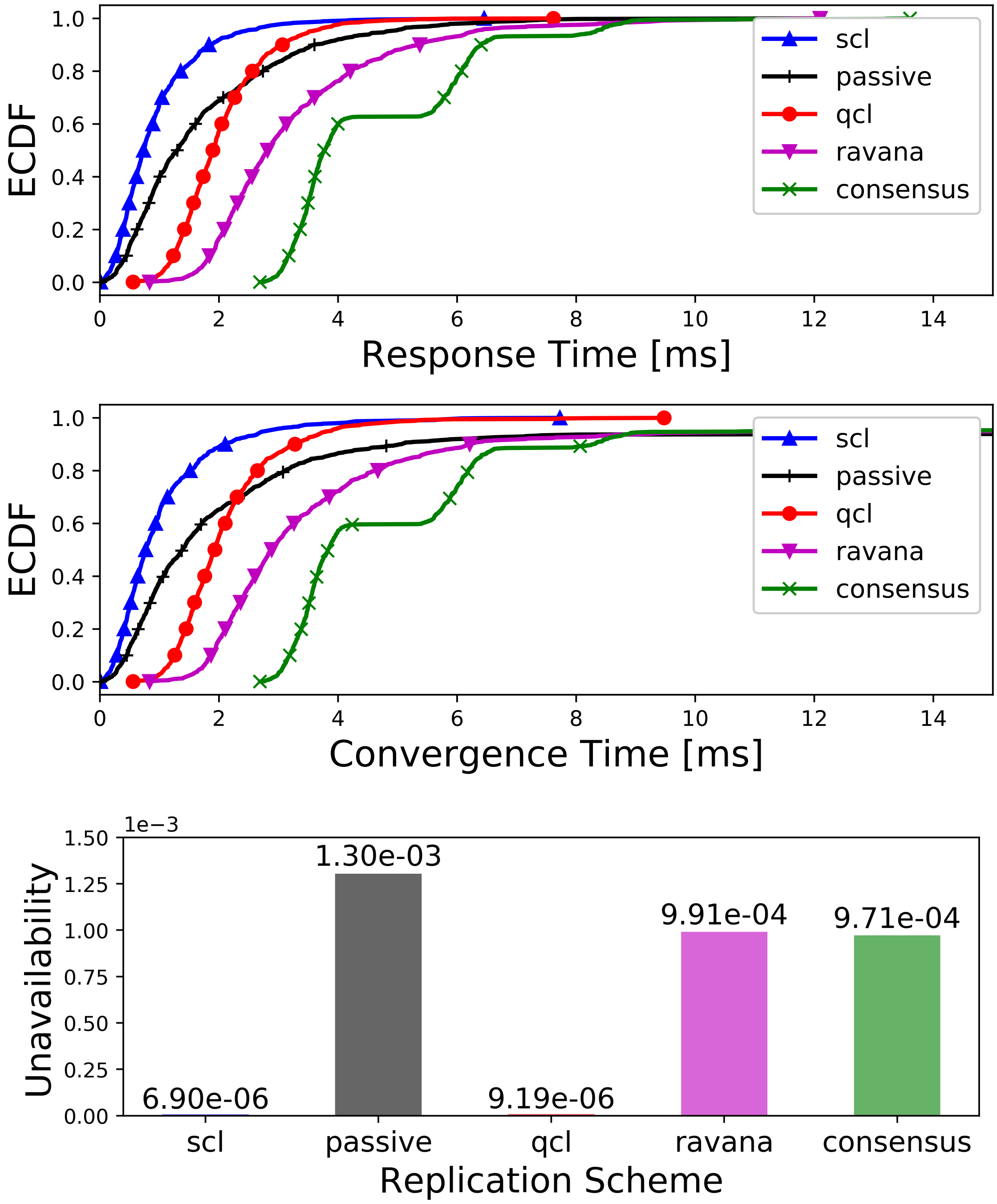


Figure 7.4 – Representative scenario (Basic): ft16 with $g = 2$, $\delta_n = 0.5\, ms$ and parameter setup `normal`

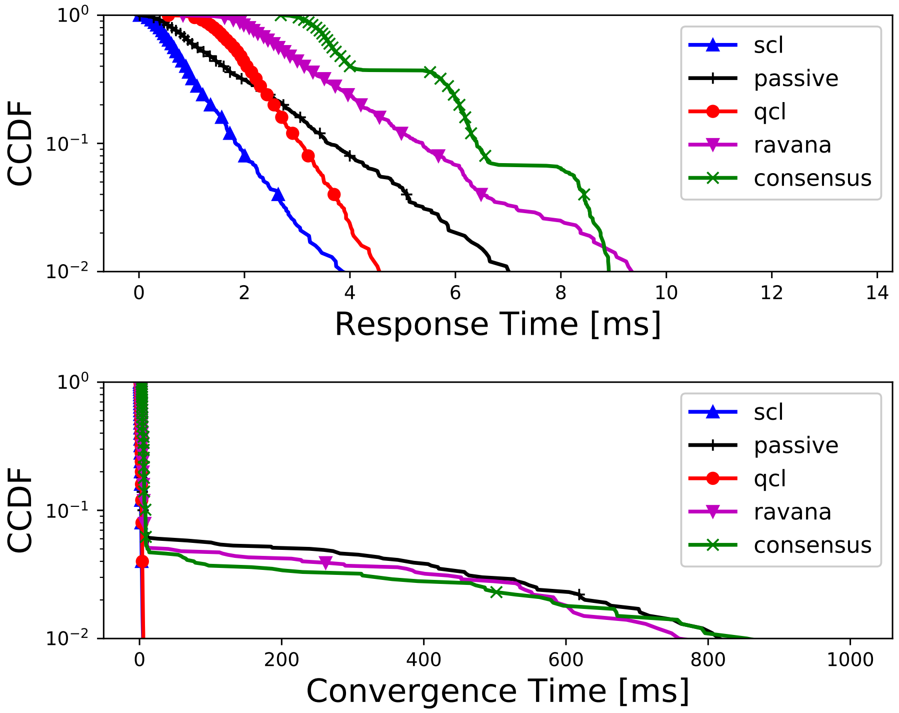


Figure 7.5 – CCDF of response and convergence times for the basic scenario

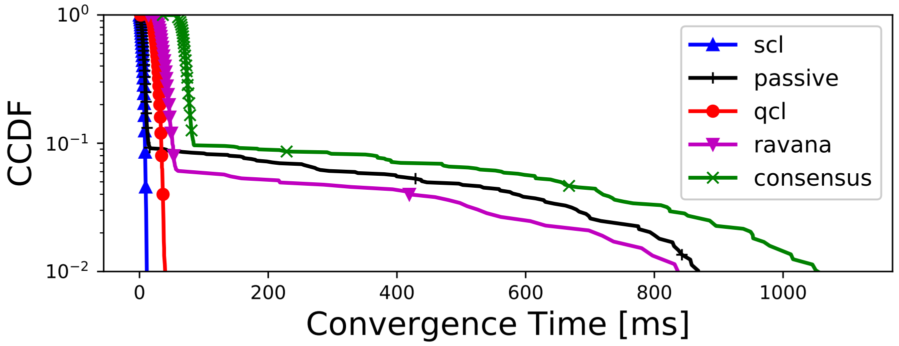


Figure 7.6 – CCDF of convergence time for AS 1221 with parameter set `normal`, $g = 2$ and $\delta_n = 10\,ms$.

7.6 System Implementation

We evaluate our proof-of-concept implementation of QCL against the original SCL implementation used in [31]. We run the two systems in the same environment with Mininet 2.2.1 [97]: 20 switches and 16 hosts are connected via 48 1Gb links (with the default Mininet behavior) forming a 4-port fat-tree. We use a single machine with two Intel Xeon E5-2680 (Haswell) processors with a total of 24 cores and 48 hyperthreads running at 2.5 GHz, and 256GB of main memory. We run Ubuntu LTS 16.04.2 distribution with the Linux kernel version 4.4.0. In both QCL and SCL we run POX controllers with shortest-path application and pair them with their corresponding QCPs via OpenFlow 1.0 [197]. On the switch side we run unmodified Open vSwitch [198] (version 2.5.2) that communicates with the QSP layer, also via OpenFlow 1.0.

Note that QCP (Algorithm 7.2) needs to differentiate between updates received from controllers as a result of two successive computations, in order to label them with the correct label. To this end, it uses the flag `controller_free` to check if the single-image controller finished computing, before sending new Status messages, thereby ensuring that the received updates correspond to the last Status messages. We implement this mechanism by sending an OpenFlow `handle_QUEUE_GET_CONFIG_REQUEST` message from the controller to the QCP, when the controller completes a computation. When this message is received, the `controller_free` flag is set to True, to indicate that the controller is now available to perform a new computation.

We analyze the response and convergence times (defined in Section 7.5) of both QCL and SCL across 200 random single-link failures. The time between consecutive events is one minute. This scenario is similar to the one used for the evaluation of SCL in [31].

In Table 7.7 we show our results with two and three single-image controller replicas. We find that the QCL and SCL are comparable in both response and convergence times. This is due to the fact that in good network conditions, QCL does not need to perform the collection phase of Quarts and can go straight to the voting phase after the probing, as described in Section 7.3.4. The probing phase lasted an average of $20\ ms$ during which all the Status messages were received. In cases where the controllers entered Quarts, the additional latency was $3\ ms$. Notice that while the response time of QCL is slightly higher than SCL due to the probing delay, the convergence time of QCL is slightly better than SCL due to control-plane consistency in QCL ensuring that different controllers do not install conflicting updates. As a result, the data-plane converges faster with QCL in our implementation.

From evaluation in [31] we know that for the same 4-port fat-tree topology ONOS is expected to be $1.24\times$ and $1.63\times$ slower than SCL, for the tail response and convergence time respectively. Similar results are expected for QCL. Note that the large latency-

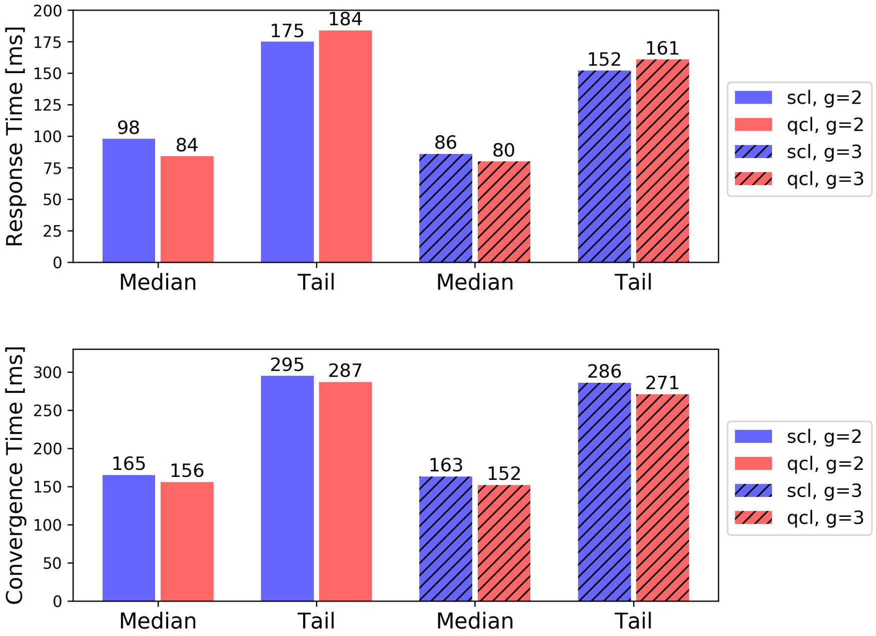


Figure 7.7 – Median and tail (at 99^{th} percentile) of 4-port fat-tree with 2 and 3 replicas in ms

improved observed in Section 7.5 is not visible in this implementation study because the high latency with Consensus (ONOS) is observed in cases of network partitions not studied here.

7.7 Related Work

Numerous related works evaluate the trade-off between the inconsistent global view network and latency performance of SDN controller applications [199, 200].

On the one hand there are numerous approaches that focus on ensuring control-plane consistency, but impose high delays in responding to network events. For instance, Hyperflow [201] that passively synchronizes network-wide views of OpenFlow controllers. Furthermore, ONOS [191] uses Paxos, that is known to have high latency and complicated implementations [31, 111]. Raft is designed to be easy to understand, but it is equivalent to Paxos in terms of correctness and performance - therefore the approaches that use Raft (e.g. ONIX) [189, 202] have the same latency issues like those that use Paxos. Ravana comes in many different flavors that offer different levels of consistency guarantees. The strongest one offers exactly-once event processing and exactly-once execution of commands that comes at a big latency price [190]. On the other hand approaches like SCL chose the opposite trade-off, as we saw in Section 7.1. The trade-offs in SCL [31], Ravana [190] and Consensus [191] have been extensively studied in our performance evaluation study in Section 7.5.

QCL managed to get the best of both worlds by using Quarts to perform agreement among controller replicas only when needed.

An orthogonal problem is to control-plane consistency is the consistency of updates under asynchronous installation of updates to the switches [193, 203, 204, 205, 206]. For that purpose QCL uses packet labeling (e.g., as in [193], [206]) as described in Section 7.3.5.

7.8 Conclusion

We have presented QCL, a coordination layer for replicated single-image controllers that guarantees control-plane consistency with low-latency. Through simulation, we find that QCL provides a $160\times$ improvement in tail latency when compared to other consistency-guaranteeing mechanisms in datacenter topologies. We have formally proven that the QCL can be used to enforce all safety policies that are enforceable by the underlying single-image controller. Also, it can be applied for both stateless and reactive controller applications. Note that, in this work QCL is applied to stateless controller applications. However, Quarts can handle state and we are planning to adapt QCL for stateful controller applications in our future work.

8 Reliable Real-Time Communication in Unreliable Networks

Look, if you had one shot
Or one opportunity
To deliver a packet
One moment
Would you capture it,
or just let it slip? Yo
– (inspired by) Eminem, Lose Yourself.

Real-time CPSs rely on reliable communication to achieve their timely control of the physical process. Even the reliability mechanisms presented in the previous chapters are inutile in the absence of a good communication network. For example, in a control round where the controller replicas issue consistency-guaranteeing setpoints well within the validity horizon, and the network drops these setpoints, the desired control cannot be achieved.

One example of such a time-critical CPS is the real-time streaming of phasor measurements in electric grids [207]. This streaming demands extremely low PLR and latency from the communication network [208]. The phasor measurements are used for a wide variety of applications. Some applications such as energy metering can tolerate a PLR of 10^{-3} and latencies up to 1 s [209]. But other applications such as PMU based state-estimation and wide-area protection require a PLR of 10^{-5} and latencies < 4 ms [52]. Mission-critical PMU-streaming applications require an availability greater than five-9's (> 0.99999). In such cases, the software agents do not have the time budget to repair the losses with retransmissions, as done in TCP. They only have a short window during which the message has to be delivered to the application; if this fails, the measurements are are rendered unusable.

8.1 Introduction

Traditionally, highly reliable communication in CPSs is achieved using wired infrastructure. However, wired networks are often laden with slow deployment and high-installation costs. Consequently, wireless infrastructure, Wi-Fi (IEEE 802.11) in particular, has recently gained traction [210, 211, 212]. The main reserve in the use of wireless infrastructure is the low QoS in terms of losses and latency, due to low reliability of wireless links.

Typically, reliable communication is achieved by using TCP. TCP uses retransmissions to repair packet losses on the fly. TCP also ensures in order delivery of packets. Thus, more recent packets are only delivered to the application after older packets have been successfully delivered. Using this approach in real-time CPSs is detrimental as newer data in CPSs is more valuable than older data because it represents a more recent state of the physical process. Hence, for real-time data, UDP is preferred as a transport protocol over TCP. See Section 8.2 for more details.

As UDP does not provide reliable transport, there needs to be other mechanisms to ensure the reliability of real-time flows. In the literature, the desired QoS target is achieved by replication of packets over two or more fail-independent paths, for example, using PRP at the MAC-layer [81, 145]. Parallel replication not only provides reliable communication in cases of packet losses due to congestion or disturbance in some of the links, it also handles network component failures. It can also serve as a method to continue uninterrupted operation during the times of scheduled maintenance. The purpose of PRP is to ensure that message losses, due to packet drops or component failures in the network, are repaired instantly ("0 ms repair"). To this end, PRP replicates packets over two disjoint cloned networks and appends the MAC header with a PRP header. The PRP header contains control information such as which of the cloned network to which a packet belongs and a sequence number. The sequence number is used by the PRP module at the receiver to de-duplicate packets and to deliver them to the application.

Currently, there is an increasing use of IP-based communication in CPSs [52]. PRP has a few drawbacks with regards to applicability in IP-based networks that use routers, thereby constituting a WAN in contrast to a single LAN. Primarily, the control information in PRP is put in the MAC header, and IP routers strip the MAC header at each hop, thus losing this critical information. Modern CPSs [4, 207] often use IP multicast for communication, where receivers might asynchronously join or leave an existing transmission. PRP does not natively provide reliability for IP multicast traffic. Moreover, in PRP, all traffic (including non real-time traffic such as management traffic and software updates) between a sender a receiver is replicated. This is both superfluous and undesirable for WANs where links are often bandwidth limited. We further discuss these issues and review other related work in Section 8.2.

Another key drawback of PRP concerns its duplicate-discard algorithm presented in [81]. It ensures that PRP forwards at most one copy of a packet to the application as long as the network does not reorder the messages. Although this assumption is true in a single LAN, packets of the same flow in a WAN might be reordered due to middleboxes such as scrubbers and load-balancers, as discussed in [149]. In scenarios when the messages are re-ordered, the PRP duplicate-discard algorithm [81] can forward duplicates to the application. This property of the discard algorithm might violate the state-safety correctness property of the CPS, especially when the CPS expects to receive only one copy of a packet.

To address the issues with PRP and provide 0 ms repair for CPSs that communicate over a WAN, we propose iPRP, an IP-friendly parallel-redundancy protocol. iPRP provides $1+n$ redundancy and is a software solution that only requires installation on the end-hosts (or simply a host, used to identify senders and receivers of the data) and no changes to the intermediate routers. It performs selective replication for time-critical UDP flows by appending the UDP header with an iPRP header. At the receiver, the information in this header is used to recreate the original packet and to forward at most one copy of the packet to the application. Besides the replication and deduplication, iPRP uses a signaling mechanism to constantly check for the failure of a host (to stop replication); availability of new paths (to start replication for increased reliability) and for the joining of new receivers in multicast communication (to stream replicated traffic to them). iPRP is transparent to the network and agnostic to the application. Another advantage of being IP friendly is that iPRP can use a wide range of TCP/IP diagnostic utilities such as `ping` and `traceroute` to ease the debugging of network issues. We also have a few iPRP specific tools and present a detailed iPRP diagnostic toolkit. We implemented iPRP[1] and have deployed this implementation on a campus-wide communication network for real-time streaming of PMU data [53].

Needless to say, both PRP and iPRP rely on the existence of fail-independent paths to reap maximum benefits from the replication. In wired networks, fail-independent paths are obtained by using physically separated networks or by appropriate routing rules in a larger, parent network. Hence, by ensuring that the replicas of the packets share no common links or devices, it is easy to guarantee that the loss of one packet does not affect the reception of its replicas.

Obtaining fail-independent paths, however, is no longer trivial in wireless networks, as the Wi-Fi links share a common medium. Consequently, the replication of packets over redundant Wi-Fi does not necessarily guarantee that the QoS requirements of mission-critical PMU-streaming applications will be satisfied. Hence, to be able to use Wi-Fi for PMU-streaming applications, the performance of parallel redundancy protocols over Wi-Fi needs to be experimentally characterized and compared against

---

[1] Available at https://github.com/LCA2-EPFL/iprp

the desired QoS requirements.

Recently, simulation- [213, 214, 215] and measurement-based [216, 217] studies were employed to quantify the loss and latency performance of packet replication over redundant Wi-Fi paths. The simulation-based studies fail to capture the real performance of redundant Wi-Fi paths. The measurement-based studies have two shortcomings: (1) The measurements were conducted in a laboratory environment and do not represent a real-life setting. (2) The measurement studies focus on generic real-time applications. Thus, the traffic profiles used are quite different from those of PMU-streaming applications, thereby rendering these studies non-representative.

We perform measurements on a test bed that is designed to closely imitate a real-life deployment of a campus-wide active distribution-network that uses PMU-based state estimation as described in [53]. Concretely, using commodity hardware, we designed a test bed that uses Wi-Fi technology (IEEE 802.11b standard). The test bed consists of nodes communicating with each other by using two spatially co-located Wi-Fi links that use directional antennas. The traffic profile used in the test bed is the same as that of PMUs in the distribution-network. Furthermore, the sending and receiving nodes are placed at the same locations as the PMUs and PDC in the distribution-network.

Using the measurements, we quantify the PLR, the end-to-end latency, jitter in latency and the availability of the effective channel, as perceived by the receiving application after replication. Using statistical inference techniques, we verify if the two links are truly fail-independent. From the setting we evaluated, we find that the losses on the two links are in fact not independent. However, the effective PLR is similar to what it would be if they were to be independent. Consequently, we conclude from the setting we evaluated, that using replication over redundant Wi-Fi paths with PRP or iPRP, is a viable option for streaming mission-critical PMU data.

The rest of this chapter is structured as follows. We review the existing solutions for reliable real-time communication in Section 8.2. In Section 8.3, we present the challenges in realizing a practically viable design of iPRP followed by a detailed description of the iPRP design in Section 8.4. This is followed by the proof of correctness of the discard algorithm in Section 8.5. We present the iPRP diagnostic toolkit in Section 8.6. Then, we present the description of the measurement test-bed to asses the viability of using directional Wi-Fi links for parallel redundancy protocols and the associated results in Section 8.7. We present the statistical test for fail-independence of directional Wi-Fi links in Section 8.8 and our concluding remarks in Section 8.9.

8.2 Related Work

PRP [81, 145] is the go-to-protocol to achieve 0 ms repair of real-time CPS traffic. As mentioned in Section 8.1, PRP is faced with several drawbacks in terms of applicability to WANs. These drawbacks were noted by the authors of [150], where modifications to PRP are proposed so that it can be applied to WANs. However, these modifications are neither fully designed or implemented. The design proposed in [150] requires that the intermediate routers in a WAN preserve the PRP trailer. This requires changes to all the intermediate routers, which makes the solution not transparent to the network and difficult to deploy. Moreover, the solution in [150] does not address the use of IP multicast nor does it support a diagnostic toolkit, a key utility in IP networks. Our transport layer solution avoids these drawbacks.

Another approach to achieving reliability through transport-layer redundancy is MPTCP [134]. MPTCP uses multi-homed devices to setup parallel channels that are then used to either send different packets (in case of the normal packet scheduler) or the same packet replicated over different networks (in case of the redundant scheduler [218]). Losses on each path are repaired independently, through retransmissions. However, MPTCP suffers from the same drawbacks as TCP. First, the packets are delivered in-order to the application, which causes the head-of-line blocking issue [133], where more recent packets are queued until older packets are delivered. This is counter-productive for traffic in CPSs because older data is subsumed by newer packets as it represents a more recent state of the physical process. Second, packet are detected as lost after the RTO which doubles after every loss, after which retransmissions are used to repair losses. Although retransmissions are often unsuitable for real-time traffic, doubling the RTO after every loss is highly undesirable as it adds more delay after each loss. Third, TCP and MPTCP do not support IP multicast. Lastly, TCP does not handle the case of service outage due to a component failure in the network and due to scheduled maintenance, where sending packets on the same path does not help with reliability. Some of these drawbacks are addressed by the upcoming QUIC transport protocol [135, 136], but support of real-time traffic and parallel redundancy is still in its nascent stages.

Routing protocols such as RIP [141] and OSPF [140] can be used to repair losses due to link or component failures. They rely on finding alternate paths, which can take a few seconds to converge, during which the real-time traffic is not delivered to the CPS application. A similar approach is used in used by MPLS-TP, where the backup path (called the protection path) is pre-computed. However, MPLS-TP takes about $50\ ms$ to perform a failover from the working path to the protection path. This makes it unsuitable for time-critical CPS applications (such as teleprotection) that require delivery of packets in under $4\ ms$.

When two or more network subclouds are available for replication, an alternative to replicating all packets on all the subclouds is to selectively replicate packets only on the networks that have a high probability of delivering it. This approach is explored in [219], where the replication mechanism monitors the state of the network (good or bad) and defers transmission on a network until it is in a good state. This approach is complementary to iPRP and can reuse the key elements of iPRP design, such as the iPRP session creation and maintenance, or de-duplication algorithm.

All these protocols rely on having a sound underlying network and solutions like PRP and iPRP require fail-independent paths. To facilitate the wide-spread adoption of Wi-Fi for mission-critical applications, several studies in the literature have evaluated the loss and latency performance of individual Wi-Fi links through measurements [220, 221]. However, there exist very few measurement-based studies [216, 217] that characterize the performance of replicated Wi-Fi links in the context of real-time applications. Both these measurement-based studies [216, 217] share similar shortcomings that render the results non-representative of streaming applications for mission-critical PMU data.

First, the measurements in [216,217] were conducted under a controlled, laboratory environment that is radically different from an in-field deployment. In contrast, our measurements were conducted using directional antennas on campus roof-tops. The locations of the measurement sites are the same as those of the PMUs in an existing campus-wide distribution-network that relies on the mission-critical PMU data [53].

Second, as the existing studies do not focus on validating the feasibility of using Wi-Fi for PMU-based applications, the traffic profiles used are quite different from those of PMU-streaming applications. Incidentally, the main finding of these papers is that the PLR is strongly dependent on the traffic profile. In light of these results, we employ the same traffic profile as used by the PMU-based state-estimation in [53]. Recently, traffic from streaming applications was used for measurements in a similar study [222]. However, we perform a more formal statistical analysis on the data to test the fail independence of the two links.

Lastly, in contrast to the short duration (of a few days) of these measurement campaigns, our measurements were conducted over period of 45 days, thereby increasing the amount of data at hand. The longer duration also captures a wider spectrum of fading effects, electro-magnetic disturbances, cross-talk, etc.; these effects are likely to surface in real deployments.

8.3 Technical Challenges in iPRP Design

Although the idea of replicating packets over two or more paths is straight-forward, there are several non-trivial challenges in designing iPRP. They are as follows:

1. Obtaining a network-transparent and application-agnostic design of iPRP. This is an important feature that determines the ease of adoption of iPRP. An ideal solution works without any specialized network hardware and without any modifications to existing CPS applications.

2. As discussed earlier, the selective replication of only the real-time traffic is a desirable feature. However, this is difficult to achieve without any modifications to the applications.

3. The communicating agents in the CPS might crash/reboot or have addition/removal of network interfaces at any time. These dynamic changes in the sender or the receiver affect the replication of the traffic. For instance, if both the end hosts have a new interface available for replication, then the two ends must implement a protocol to agree to perform replication on this interface. This is done through a handshake between the sender and receiver(s).

4. The application must receive at most one copy of a packet. This requires designing a discard algorithm that does not forward duplicates to the applications even in the presence of packet reordering. As the de-duplication needs to be in the real-time path, the algorithm must have a low time-complexity. Furthermore, as an agent might be communicating with several agents at the same time, each requiring a different instance of the discard algorithm, it is desirable to have a low memory footprint for each instance of the algorithm.

5. As many applications in a CPS use IP-multicast, iPRP must be able to natively support it. In order to do so, iPRP must permit asynchronous joining and leaving of receivers without affecting the replication for existing receivers. This is particularly challenging because of two contradicting requirements. On the one hand, applications must receive exactly one copy of the packet, thereby requiring that all packets of the real-time flow be replicated and communicated to the de-duplication algorithm at the receiver. On the other hand, in order for newly added receivers to be able to perform a handshake with the sender (to receive the relevant information to process replicated data), they first have to learn about the existence of this sender. This requires the iPRP protocol at the sender to transmit original, non-iPRP packets, in addition to the replicated traffic. In a setup with two parallel paths, this implies that the sender sends the original packet from the application, unmodified on the first network and two replicated iPRP packets on the two networks. This would lead to existing receivers receiving two copies of the packet: one original packet and one from iPRP after the de-duplication algorithm.

8.4 iPRP Design

iPRP provides $1+n$ redundancy for real-time UDP flows and does not impact TCP flows. To this end, it replicates packets of the real-time flows at the sender and performs de-duplication at the receiver. In order to ensure that the replicated packets are isolated from other traffic, all replicated traffic is sent to a pre-determined *iPRP data port* D. Once a sender and receiver start exchanging replicated traffic, they are said to be connected via an *iPRP session*. An iPRP session is uni-directional, from one sender to one or more receivers, for unicast or multicast traffic, respectively. All control traffic for initiating and maintaining an iPRP session is exchanged over a another pre-determined port called the *iPRP control port* C ($\neq D$). We require that the two ports C and D, be reserved in the operating system of the communicating hosts, exclusively for iPRP related traffic.

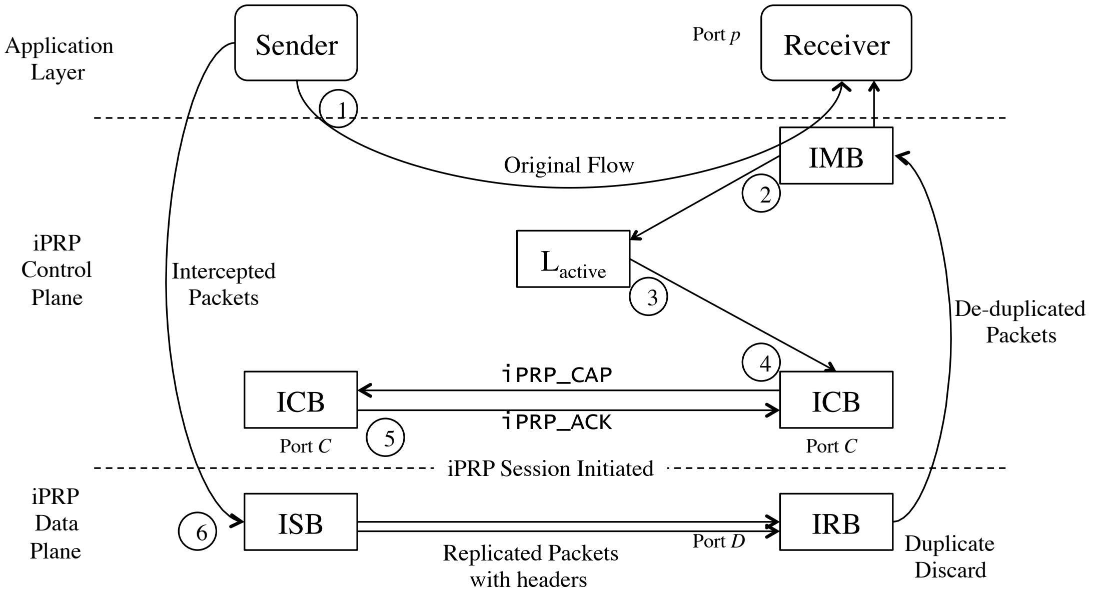


Figure 8.1 – Overview of the iPRP design indicating the flow of a packet

Figure 8.1 gives an overview of the iPRP design. It consists of four functional blocks: the iPRP monitoring block (IMB), the iPRP control block (ICB), the iPRP sending block (ISB) and the iPRP receiving block (IRB). The design of these functional blocks is given by Algorithms 8.1, 8.2, 8.3 and 8.4, respectively. The operation of iPRP can be divided into the control plane and the data plane. The control plane is responsible for monitoring new flows that need replication using the IMB at the receiver, and initiating and maintaining an iPRP session using the ICB at the sender and the receivers. The data plane is responsible for the replication of traffic using the ISB at the sender and the de-duplication of traffic using the IRB at the receiver. Note that the control plane is for non real-time iPRP control traffic, whereas the data plane is for the real-time CPS traffic.

We will explain the design of iPRP by first giving a walk-through of the normal operation of iPRP in Section 8.4.1. Then, we highlight the key elements of the design in Sections 8.4.2-8.4.6.

As mentioned earlier, we implemented iPRP and make the open-source implementation publicly available. The implementation uses the NF_QUEUE framework from the Linux iptables project because it enables filtering packets through iptables rules, and modifying them in user-space. This is the same framework used in packet mangling with Axo, in Chapter 6. This implementation is deployed on a campus-wide network for real-time streaming of PMU traffic at EPFL.

8.4.1 Protocol Walk-Through

Consider a PMU with an IP address S on one of its interfaces, streaming phasor data to the destination port p of the PDC using a UDP-based application. Sending and receiving hosts, such as PMUs and PDCs, can have any number of network interfaces. However, for the hosts to be able to use replication, it is desirable to have more than one network interface each. Moreover, in order to reap maximum-benefit from replication, it is desirable to have fail-independent paths through link- and edge-disjoint network subclouds. This is often difficult in WANs, as the users of the CPS do not have full control of the network, because a slice of the network might be a part of the public infrastructure. Additionally, most WANs have connected subclouds for the ease of management. Figure 8.2 shows an example of such a network where the PMU is connected to two network subclouds A and B, and the PDC is connected to A, B and C. Subclouds A and B use wired communication and are connected to a common maintenance server, whereas sub-cloud C uses Wi-Fi based communication. One key difference of iPRP with respect to PRP is that it is network independent and can function without cloned disjoint networks.

To start iPRP, the hosts have to initialize the ICB (Algorithm 8.2). At this point, no visible change is noticeable at the senders and receivers, and the data transfer continues unaffected. To trigger iPRP, the receiver has to mark the UDP port p as a monitored port. This is done by locally configuring it in the set of monitored ports $\mathcal{P}$, at ② in Figure 8.1. At this juncture, the ICB instantiates an IMB (Algorithm 8.1) that listens for incoming traffic on port p. The IMB maintains two lists (1) the list of active senders ($\mathbf{L}_{active}$) that keeps track of which senders are sending traffic to the ports in $\mathcal{P}$, and (2) the list of established sessions ($\mathbf{L}_{est}$) that keeps track of the flows for which an iPRP session has already been established.

When a packet on port p is received by the IMB (at Algorithm 8.1 line 3), as the source IP address of this packet is not in the currently empty $\mathbf{L}_{active}$, it is added to $\mathbf{L}_{active}$ with a timestamp called the *last-seen timer*. This timestamp is updated every

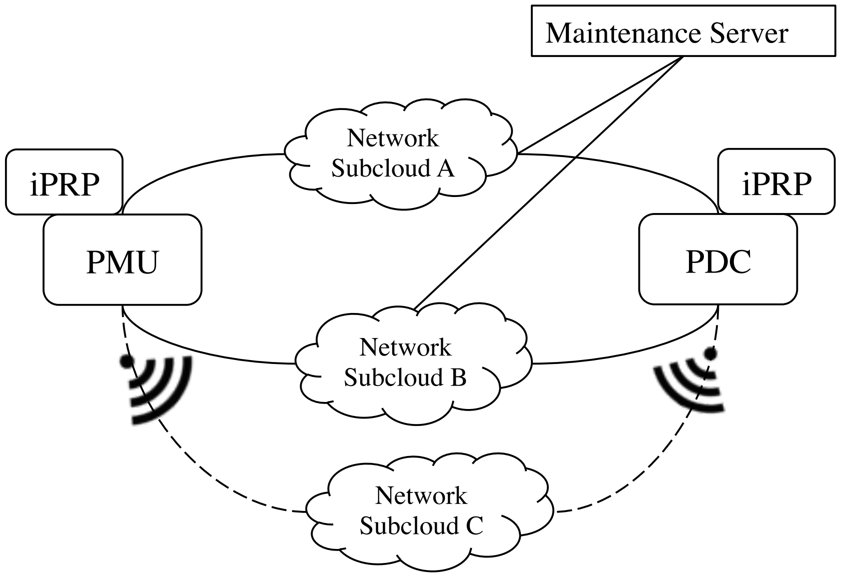


Figure 8.2 – Example of a network with two connected subclouds performing iPRP

time a packet is received (at line 7) and is used by the IMB to remove aged entries corresponding to inactive senders. An entry is considered aged if its timestamp is older than $T_{inactivity}$.

In Figure 8.1, a new entry added to $\mathbf{L}_{active}$ at ③ triggers sending of `iPRP_CAP` messages by the ICB at the receiver at ④ to the ICB at the sender. The routine for sending these messages is given by Algorithm 8.2 lines 15-18. The `iPRP_CAP` messages are non real-time control messages used to advertise to the host that is sending real-time traffic, the networks currently available for replication at a host receiving the real-time traffic. In this case, the `iPRP_CAP` messages contain the three networks A, B and C, encoded using their pre-configured 4-bit iPRP network discriminators (INDs). The `iPRP_CAP` messages also include the port number p that triggered its sending. The IP address of the PDC and the port number p are used to uniquely determine an iPRP session at the PMU. The ICB also starts the IRB (Algorithm 8.4) that listens for replicated iPRP packets on port D.

When the ICB at the PMU receives this message at ⑤ in Figure 8.1 and at Algorithm 8.2 line 2, it checks if there already exists an iPRP session for this flow. In this case, as there is no iPRP session, it records the creation of a new iPRP session and sends an acknowledgment (`iPRP_ACK`). In the case of multicast, to avoid a barrage of `iPRP_CAP` messages from all the receivers, the `iPRP_ACK` are particularly useful, as discussed further in Section 8.4.2. In order to identify the network subclouds on which the packets must be replicated, the ICB performs IND matching by taking an intersection of its own INDs with the ones received in the `iPRP_CAP` message (Algorithm 8.2 line 4), which returns the INDs A and B in this case.

Note that `iPRP_CAP` messages are sent periodically as a part of a keep-alive mechanism for the iPRP session, and aged sessions are continuously deleted from the senders record. Similarly, if a receiver does not receive iPRP messages from a sender for some time, it marks the session as dead, and updates $\mathbf{L}_{active}$ and $\mathbf{L}_{est}$. These messages also serve as a mechanism to address dynamic changes in the hosts as described in Section 8.4.4. Furthermore, all control messages are exchanged on the reserved port C and are isolated from the real-time traffic.

Algorithm 8.1: IMB: monitor for new flows at the receiver

```
$\mathbf{L}_{active} \leftarrow [\,]$;                                  // List of active senders
$\mathbf{L}_{est} \leftarrow [\,]$;                                     // List of established sessions
for each packet received on a port in $\mathcal{P}$ with source IP address $S$ do
    if $S \notin \mathbf{L}_{active}$ then
        Add $S$ to $\mathbf{L}_{active}$;                                // To send IPRP_CAP messages
    end
    Update last-seen timer of $S$;
    if $S \in \mathbf{L}_{est}$ and the packet was not received from IRB then
        Discard the packet;                                              // Already receiving replicated packets
    else
        Accept the packet;
    end
end
while true do
    Add entries for new iPRP sessions to $\mathbf{L}_{est}$;
    Remove aged entries from $\mathbf{L}_{active}$ and $\mathbf{L}_{est}$;
end
```

Once the iPRP session is created, the ICB at the sender starts the ISB (Algorithm 8.3) that intercepts all outgoing traffic for the flow corresponding to the iPRP session. In this case, the ISB at the PMU intercepts all traffic directed towards the IP address of the PDC and the port p as shown at ⑥ in Figure 8.1. The ISB replicate these packets, creates an iPRP header for each packet, and sends the replicated packets on the two matched network subclouds A and B, but to the iPRP data port D instead of the original destination port p. The iPRP header is attached as a trailer to the UDP header. This places iPRP between the UDP layer and the IP layer, making it a transport layer solution, as show in Figure 8.3. Given that iPRP is required to be transparent to the network, *i.e.,* the routers and switches in the network have to be unaware of iPRP, the header needs to be above the IP layer. This leaves two choices: the transport layer or the application layer. We chose the transport layer as it also satisfies the other requirement to be transparent to the application. This enables a design that is agnostic to the application and does not require any modifications to the existing applications.

The iPRP header contains all information required by the receiver to reconstruct the original packet and forward it to the application. Specifically, it comprises:

- SNSID that is used to uniquely identify an iPRP session. It is formed from the desti-

Algorithm 8.2: ICB: Initiate and maintain iPRP session

```
// At the sender
for each iPRP_CAP message received for port $p$ from a source IP address $S$ do
    if no iPRP session is established for $(S, p)$ then
        if IND matching is successful then
            Record creation of an iPRP session for $(S, p)$;
            send iPRP_ACK message;
        end
    else
        Update the keep-alive timer for the iPRP session $(S, p)$ ;
    end
end

// At the receiver
while true do
    compute $T_{backoff}$ (Section 8.4.2) ;
    listen for iPRP_ACKs until $T_{backoff}$ expires;
    send iPRP_CAP messages to all hosts in $\mathbf{L}_{active}$ from which no iPRP_ACKs are
      received;
    sleep $T_{CAP} - T_{backoff}$;
end
```

nation IP address of the PDC and the port p (160-bit)
- Sequence number of the packet (32-bit)
- Original destination port p (16-bit)
- IND of the network subcloud (4-bit)
- HMAC for authentication (160-bit)

When the replicated packets are received by the IRB at the PDC, they are checked to verify whether they they are new or duplicates by using the `isFreshPacket` function described in Algorithm 8.5. This function is the de-duplication or discard algorithm of iPRP. If the packet is indeed the first received copy, the payload is extracted, the original packet is reconstructed using the original destination port and IP address in the iPRP header. Then, the original packet is forwarded to the IMB, which uses it to update the last-seen timer, and forwards it to the application. More details on the discard algorithm are presented in Section 8.4.5.

Notice that by filtering packets based on the destination IP address and destination port p, iPRP achieves selective replication of only the real-time flows.

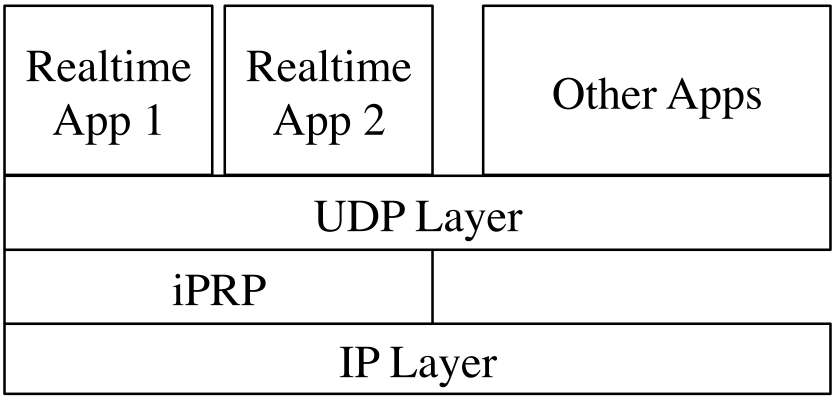


Figure 8.3 – Placement of iPRP in the networking stack

Algorithm 8.3: ISB: Replicate packets

```
1  for each intercepted packet for IP address $S$ and port $p$ do
2      if there exists an iPRP session for $(S, p)$ then
3          Replicate the payload;
4          Append iPRP headers;
5          Discard original packet;
6          Send replicated packets to the matched INDs;
7      else
8          Forward the packet unchanged;
9      end
10 end
```

8.4.2 IP Multicast

As noted earlier (Section 8.1), natively supporting IP multicast is one of the key requirements for CPSs. At the outset, we note that as iPRP performs selective replication of only the real-time traffic, it does not affect protocols such as IGMP [223] and MLD [224], which are used to form multicast forwarding trees. Therefore, the network-layer operation of IP multicast and iPRP are independent.

One of the challenges in supporting multicast is to permit newly joined receivers to benefit from replicated traffic, as discussed in Section 8.3. The issue here is to ensure that the application layer on all receivers is presented with at most one copy of the packet, despite sending both: the replicated traffic to port D for receivers that have already established an iPRP session, and non-replicated traffic to port p for new receivers to establish an iPRP session. This is achieved as follows.

When using multicast, by selectively executing Algorithm 8.3 line 5, the ISB periodically lets an original unmodified packet through, on port p. Thus, once in a while, multicast receivers that have already established an iPRP session receive a packet on the receiving application's port p, thereby triggering Algorithm 8.1 line 3 at the IMB. However, as this source is already in $\mathbf{L}_{est}$, the packet is discarded. Only the packets

Algorithm 8.4: IRB: Discard duplicate and forward packets to the application

```
for every packet received on iPRP data port do
    get sequence number space ID (SNSID);
    get sequence number (SN);
    if it is the first packet from this SNSID then
        SNSID.HighSN← SN;                // Bootstrap
        remove iPRP header;
        reconstruct original packet;
        forward to IMB;
    else
        if isFreshPacket(SN, SNSID) then
            remove iPRP header;
            reconstruct original packet;
            forward to IMB;
        else
            silently discard the packet;
        end
    end
end
```

received from IRB that have already been cleared by the discard algorithm are forwarded to the application. Alternatively, on new receivers, the reception of a packet on port p triggers a session established by the IMB. Thus, they can shortly start receiving replicated packets on port D.

Recall that all the receivers that have established an iPRP session, periodically advertise their capabilities through the `iPRP_CAP` messages. This periodic timer is reset to the same value at all the receivers in the multicast group when an `iPRP_ACK` is received. Thus, the timers on all receivers fire close to each other, causing all the receivers to simultaneously send `iPRP_CAP` messages. This can cause a "message implosion" at the source, as the size of the multicast groups can vary from a few tens to several millions. This problem is avoided by waiting at the receivers for a time $T_{backoff}$ after the timer fires and before sending the messages. If during this time, an `iPRP_ACK` message is received from the sender, then the sending of `iPRP_CAP` at this receiver is suppressed in this round. $T_{backoff}$ is computed using the approach in [225], where a similar problem was studied in the context of reliable multicast for video streaming. In this approach, the waiting time is drawn from a modified exponential distribution, such that with a very high probability, all but one or two hosts receive the `iPRP_ACK` messages before sending out their `iPRP_CAP` messages.

Algorithm 8.5: `isFreshPacket(CurrSN, SNSID)`: Function to determine whether a packet with sequence number `CurrSN` corresponds to a fresh packet in the sequence number space ID `SNSID`. The test "`x` follows `y`" is performed for 32-bit unsigned integers using subtraction without borrowing as "`(x-y)>>31==0`".

```
if CurrSN==SNSID.HighSN then
    return false ;                                   // Duplicate packet
else if CurrSN follows SNSID.HighSN then
    put SNs [SNSID.HighSN+1, CurrSN-1] in SNSID.ListSN;
    remove the smallest SNs until SNSID.ListSN has MaxLost entries;
    SNSID.HighSN ← CurrSN ;                          // Fresh packet
    return true;
else
    if CurrSN is in SNSID.ListSN then
        remove CurrSN from SNSID.ListSN;
        return true ;                                // Delayed packet
    else
        return false;                                // Already seen or very late
    end
end
```

8.4.3 Crash of a Host

A challenge in designing a replication protocol (such as iPRP) is to cause minimal disruption to the real-time traffic, either when one of the host or iPRP on one of the host crashes.

When the receiver with an established iPRP session crashes, it can no longer send `iPRP_CAP` messages. Thus, the keep-alive timer at the sender is no longer updated (Algorithm 8.2 line 9). This forces the deletion of the session after $T_{inactivity}$, thereby causing the replication to end. Alternatively, if the sender crashes, the receiver no longer receives data. Consequently, it does not update the last-seen timer for that flow, thereby eventually resulting in removal of this flow from the lists $\mathbf{L}_{active}$ and $\mathbf{L}_{est}$.

Note that when we say a receiver crashes, either the host can crash (in which case the application is no longer able to send or receive data) or only iPRP can crash (in which case, the session is automatically deleted and normal communication is restored). A similar behavior is observed at the time of session establishment, when only one of the hosts is iPRP capable.

8.4.4 Addition or Removal of Network Interfaces

CPSs that are deployed in the field often undergo network maintenance, when new network subclouds are added or existing network subclouds are removed. This amounts to network interfaces being modified at one or more of the communicating hosts.

When a new network interface is added or an existing network interface is removed on one of the end hosts, the subsequent `iPRP_CAP` messages are changed to reflect this information. This in turn causes a new IND matching, and the newly added network subcloud is accepted for replication or a previously removed network subcloud is no longer chosen for replication.

In multicast operation, due the random $T_{backoff}$ in sending the $iPRP\_CAP$ messages, the sender might receive different INDs each time, as different receivers might be connected to different sets of network subclouds. It is undesirable that the a network subcloud that is connected to a vast majority of the receivers, but not connected to a handful is not used for replication. Therefore, we take a conservative approach in removing a network subcloud by waiting for confirmation from several receivers, whereas the addition is instantaneous.

8.4.5 Duplicate Discard Algorithm

The duplicate discard algorithm (Algorithm 8.5 forwards the first copy of a replicated packet to the IMB and discards all subsequent packets. It is used by the IRB (Algorithm 8.4) in the form of the function `isFreshPacket` that returns a decision, whether to accept or reject a packet.

Note that, the discard algorithm proposed for PRP [81] fails are preventing duplicates from being forwarded to the application when packets are received out of ordered. As discussed earlier, packet reordering cannot be excluded in IP networks due to middle-boxes such as load-balancers and scrubbers.

In order to accept at most one packet, the `isFreshPacket` function makes use of the variable `HighSN` that represents the highest sequence number of a packet received before the current packet, and the list `ListSN` that is the list of sequence numbers of delayed packets. A packet with sequence number i is counted as delayed if no replicate of this packet has been seen and if a packet with sequence number $j > i$ has been seen.

To better appreciate the workings of the function `isFreshPacket`, we explain through examples. A received packet with sequence number `CurrSN` falls in one of the three categories as discussed below.
Case 1: Close duplicate (line 1) In networks with symmetric delays on all network subclouds, this is the most common case. It occurs when the packet from the other network subcloud was just processed. For instance, `HighSN` = 10 and `CurrSN` = 10. Clearly, it must be discarded.
Case 2: Fresh packet (line 3) When `CurrSN` is higher than `HighSN`, the packet is new and must be accepted. However, this packet might have been received out of order. For example, `HighSN` = 10 and `CurrSN` = 15. In this case, the sequence numbers 11-14

are added to `ListSN`, as packets that are delayed and expected to arrive. Note that the size of `ListSN` is limited to `MaxLost`.
Case 3: Delayed packet When `CurrSN` is older than `HighSN` two possible sub-cases arise. Either the packet has been is very late (older than `MaxLost`) or it has already been delivered (line 13). Then, it is discarded. Alternatively, if the delayed packet has never been seen before, it is a valid packet and is accepted (line 9) and the sequence number is removed from `ListSN`.

Note that we use 32-bit arithmetic to compare sequence numbers despite wrap-around. This is captured by the related “x follows y”, as described in Algorithm 8.5.

The IRB uses one instance of `isFreshPacket` per iPRP session. This ensures isolation among packets from different flows. Moreover, in order to ensure that the correct ordering is maintained across reboots of the senders and receivers, we exploit the fact that there is a single sender of packet. To this end, we use generation numbers to differentiate the sequence numbers spaces across reboots.

The function `isFreshPacket` uses one variable `HighSN` and one list `ListSN` whose size is upper bounded by `MaxLost`. Thus, the maximum memory footprint per session is $4 \times$ `MaxLost` bytes (due to 32-bit sequence numbers). With a typical value of `MaxLost` = 1024, this amounts to $4\ KB$. Moreover, the most expensive time operation in this function is the searching of `ListSN` at line 9, which is quite low due to the small size of `MaxLost`. This can be further lowered to a constant time by using a hash table implementation, at the expense of higher memory footprint.

8.4.6 Security Considerations

Network security is critical for CPSs. We design iPRP such that it does not introduce any security vulnerabilities in the communication network. To this end, all iPRP control messages are encrypted and authenticated. In the unicast mode, a secure session is created using DTLS [226]. In multicast, iPRP relies on any primitive that establishes a secure channel with the multicast group.

In addition to the control traffic, in order to avoid replay attacks pertaining to sequence numbers, the iPRP header inserted in each message is authenticated using symmetric keys.

8.5 Correctness of the Discard Algorithm

Before stating the correctness of the algorithm, we need to introduce some definitions. We say that a received packet is *valid* if it arrives in order or if it is out-of-order, but not later than T_{late}. Formally, this means that a packet received at time t with `SN` $= \alpha$ is not valid if some packet with $\texttt{SN} = \beta > \alpha + \texttt{MaxLost}$ was received before t.

Furthermore, let Δ be an upper bound on the delay jitter across all network subclouds. Formally, for any two packets i, j sent over any two network subclouds k, l: $\Delta \geq \left(\delta_i^k - \delta_j^l\right)$, where δ denotes the one-way network delay. Also, recall from Section 8.4.1 that $T_{inactivity}$ is the time after which inactive sessions are considered aged, hence terminated.

Theorem 8.5.1 (Correctness of the discard algorithm)**.** *If $R \times \Delta < 2^{31}$ and $R \times (T_{inactivity} + \Delta) < 2^{31}$, then Algorithm 8.5 guarantees that: (1) no duplicates are forwarded to the application and (2) the first received valid copy of any original packet is forwarded to the application.*

Proof. To prove the statement of Theorem 8.5.1, we need the following lemmas.

Lemma 8.5.1. *If $R \times \Delta < 2^{31}$ and $R \times (T_{inactivity} + \Delta) < 2^{31}$, then the wrap-around problem does not exist.*

Proof. The wrap-around problem can arise in two cases.

Case 1: A late packet arrives with $\texttt{CurrSN} < \texttt{HighSN} - 2^{31}$. As $R \times \Delta < 2^{31}$, the time required by the source to emit 2^{31} packets is longer than Δ. Hence, `HighSN` cannot precede `CurrSN` for more than 2^{31} and this scenario is not possible.

Case 2: A fresh packet is received with $\texttt{CurrSN} > \texttt{HighSN} + 2^{31}$. This means that from the point of view of the receiver, there were more than 2^{31} iPRP packets lost in succession. As $R \times (T_{inactivity} + \Delta) < 2^{31}$, the time for more than 2^{31} consecutive packets to be sent is greater than $(T_{inactivity} + \Delta)$. Hence, the time between reception of any two packets differing by `SN`s more than 2^{31} is greater than $(T_{inactivity})$. Therefore, during this time the iPRP session would be terminated and a new session will be initiated when the fresh packet is received. As a result, this scenario is also not possible. $\square$

Therefore, in the rest of the proof, we can ignore the wrap-around problem and proceed as if `SN`s of received packets were integers of infinite precision. Also, the notation $\texttt{HighSN}_{t^-}$ [resp. $\texttt{HighSN}_{t^+}$] denotes the value of `HighSN` just before [resp. after] time t.

Lemma 8.5.2 (Monotonicity of HighSN)**.** *If at time t, a packet with $\texttt{SN} = \alpha$ is received, then $\texttt{HighSN}_{t^+} = \max(\texttt{HighSN}_{t^-}, \alpha)$. Therefore, `HighSN` increases monotonically with time.*

Proof. From Algorithm 8.5, when $\alpha > \texttt{HighSN}_{t^-}$ (line 3) then the value of `HighSN` is changed to α (line 6). Otherwise, when $\texttt{HighSN}_{t^-} \geq \alpha$ (lines 1 and 8), `HighSN` is unchanged, i.e., $\texttt{HighSN}_{t+} = \texttt{HighSN}_{t-}$. Combined, the two cases give $\texttt{HighSN}_{t+} = \max(\texttt{HighSN}_{t-}, \alpha)$. ☐

Lemma 8.5.3 (Fresh packet is never put in `ListSN`)**.** *If at time t, a packet with $\texttt{SN} = \alpha$ is forwarded to the application then $\alpha \notin \texttt{ListSN}_{t'+} \forall\, t' \geq t$.*

Proof. Let us prove by contradiction. Assume that $\exists\, t' > t$ such that $\alpha \in \texttt{ListSN}_{t'}$. Hence, $\exists\, t_1 \in (t, t']$ when α was added to `ListSN`. As $t_1 > t$, from Lemma 8.5.2, we conclude that $\texttt{HighSN}_{t_1-} \geq \texttt{HighSN}_{t+} \geq \alpha$. Now, from Algorithm 8.5, we know that only SNs $> \texttt{HighSN}_{t_1-}$ can be added to `ListSN`. Hence, α cannot be added to `ListSN` at time t_1. Therefore, we have a contradiction. ☐

Lemma 8.5.4. *At any time t, $\texttt{HighSN}_{t-}$ is equal to `SN` of a packet received at some time $t_0 < t$, or no packet has been received yet.*

Proof. `HighSN` is modified only at line 6, where it takes the value of the SN received. Hence, `HighSN` cannot have a value of a SN that has not been seen yet. ☐

Now, we proceed with the proof of the theorem. First, we prove statement (1). Assume we receive a duplicate packet with $\texttt{SN} = \alpha$ at time t. This means that a packet with $\texttt{SN} = \alpha$ was already seen at time $t_0 < t$. Then, from Lemma 8.5.2 it follows that $\alpha \leq \texttt{HighSN}_{t-}$. Then, either $\alpha = \texttt{HighSN}_{t-}$ (line 1) or $\alpha < \texttt{HighSN}_{t-}$ (line 9).

Case 1: When $\alpha = \texttt{HighSN}_{t-}$, the packet is discarded according to line 2.

Case 2: When $\alpha < \texttt{HighSN}_{t-}$, line 9 is evaluated as false due to Lemma 8.5.3. Hence, the packet is discarded by line 13.

Next, we prove statement (2) by contradiction. Assume we receive a first copy of a valid packet with $\texttt{SN} = \alpha$ at time t but we do not forward it. This can happen either due to line 2 (case 1) or due to line 13 (case 2).

Case 1: Statement from line 1 was evaluated as true, which means that $\alpha = \texttt{HighSN}_{t-}$. As $\texttt{SN} = \alpha$ is seen for the first time, Lemma 8.5.4 is contradicted. Hence, this case is not possible.

Case 2: Statement from line 9 was evaluated as false, which means that $\alpha < \texttt{HighSN}_{t-}$ and $\alpha \notin \texttt{ListSN}_{t-}$. We show by contradiction that this is not possible, i.e., we now assume that $\alpha < \texttt{HighSN}_{t-}$ and $\alpha \notin \texttt{ListSN}_{t-}$. Now, there are three cases when $\alpha \notin \texttt{ListSN}_{t-}$ can be true:

(i) $\texttt{SN} = \alpha$ was added to and removed from `ListSN` before time t because it was seen (line 11). This case is not possible as the packet is fresh.

(ii) $\texttt{SN} = \alpha$ was added to `ListSN` and later removed at time $t_0 < t$ because the size of `ListSN` is limited to `MaxLost` entries (line 6). This means that at time $t_0 < t$ a packet with $\texttt{SN} = \beta$ was forwarded, and $\beta - \alpha > \texttt{MaxLost}$ (line 6). However, this means that the packet with $\texttt{SN} = \alpha$ was not valid at time t_0. Therefore, it is also not valid at time $t > t_0$.

(iii) $\texttt{SN} = \alpha$ was never added to `ListSN`. Consider the set $\mathbb{T} = \{\tau \geq 0 : \texttt{HighSN}_{\tau^+} > \alpha\}$. $\mathbb{T}$ is non-empty because $t \in \mathbb{T}$, by hypothesis of our contradiction. Let $t_0 = \inf \mathbb{T}$. Then, necessarily $\texttt{HighSN}_{t_0^-} \leq \alpha < \texttt{HighSN}_{t_0^+}$ (say, $= \beta$). β is the `SN` of a packet received at time t_0. Since α is valid, $\beta - \alpha < \texttt{MaxLost}$. Otherwise, α would be invalid at time t_0, therefore at time t, which is excluded. Then we have two subcases possible:

a) $\texttt{HighSN}_{t_0^-} < \alpha$. Then, by line 4, α is added to `ListSN`, which is a contradiction.

b) $\texttt{HighSN}_{t_0^-} = \alpha$. But, by Lemma 8.5.4 a packet with $\texttt{SN} = \alpha$ must have been received before t_0, which is a contradiction because α is a fresh packet at $t \geq t_0$. □

To understand the practicality of the conditions in the theorem, note that $T_{inactivity}$ is in the order of seconds and is much larger than Δ. Therefore, the only condition to verify is $R \times (T_{inactivity} + \Delta) < 2^{31}$, which for, say $T_{inactivity} = 10s$ and $\Delta = 100ms$, requires $R < 2 \times 10^8$ packets per second – a rate much higher than ever expected.

8.6 iPRP Diagnostic Toolkit

One of the main advantages of iPRP being IP friendly is its ability to exploit the rich set of IP layer diagnostic utilities associated with TCP/IP. Such utilities are unavailable to MAC-layer redundancy solutions such as PRP. The basic tools include connectivity and path checking tools such as `ping` and `traceroute`. The more advanced iPRP specific tools enable the collection of statistics and checking for the existence of parallel paths.

The toolkit comprises the following tools:

```
iPRPtest <Remote IP Address> <Port> <Number of packets> <Time period>
iPRPping <Remote IP Address>
iPRPtracert <Remote IP Address>
iPRPsenderStats <IP Address>
iPRPreceiverStats <IP Address>.
```

Next, we describe the functioning of each tool:

- `iPRPtest` tests the unicast iPRP operation between the local and remote hosts. Firstly, it checks for the presence of an iPRP session between the two machines by querying the local peer-base and returning the peer-base entry corresponding to the iPRP session identified by the inputted IP address. This entry consists of a list of the interfaces (and their IP addresses) of the remote host connected to the networks identified by the INDs. Here is an example output if an iPRP session exists:

```
$ iPRPtest aa::1 1234 10 5
  Interface  Remote IP address  IND
     eth0          aa::1        0xa
     eth3          cc::1        0xc
```

If it does not exist, `iPRPtest` tries to establish one. It communicates the UDP port number to the iPRP-session-maintenance functional block on the remote machine. This port is then added temporarily to the set $\mathcal{P}$. After the temporary-iPRP-session establishment, `iPRPtest` sends periodic probe packets to the remote host along multiple paths, depending on the parameters *number of packets* and *time period*. Finally, the iPRP session is closed and the corresponding UDP port is removed from set $\mathcal{P}$ on the remote machine. If an iPRP session could not be established, an appropriate message is generated.

• `iPRPping` evaluates the end-to-end connectivity over multiple paths, to a remote host with an iPRP session with the local host. It exploits the ICMP `ping` and does not exercise iPRP that operates on UDP. `iPRPping` queries the local peer-base for the remote IP addresses associated with the the input IP address and uses the native ping to check connectivity over multiple paths in a round-robin fashion. `iPRPping` can also be used to obtain the path MTUs along all paths to a host by varying the size of the *ICMP echo request* packets. Finally, it reports the packet loss and RTT statistics for all the available paths. In the case of absence of an iPRP session, an appropriate message is generated.

• `iPRPtracert` enlists the routes taken by IP packets over all the paths to the remote host with the inputted IP address. It queries the local peer-base for the remote IP addresses used during an iPRP session. Then, it uses the `traceroute` from the TCP/IP suite to trace the routes over multiple paths in a sequential manner. If the remote host does not have any iPRP session with the local host, `iPRPtracert` does not attempt to establish an iPRP session and generates an appropriate message.

• `iPRPsenderStats` queries the remote IP address for packet delivery statistics associated with its iPRP session. For unicast, the argument is the IP address of the remote host with an iPRP session. In multicast, the argument is a multicast group IP address. `iPRPsenderStats` queries the remote IP address of one of the subscribers of the multicast group, for its statistics. If iPRP session does not exist, an appropriate message is generated. The reported statistics are – *PktCtrX*: Total number of packets successfully received over the network with IND X.
– *LastTimeSeenX*: UTC time stamp of last received packet over the network with IND X.
– *WrongINDX*: Number of non-IND X packets received over the IND X network. This can happen due to a common link between multiple networks or faulty cabling at the hosts. The iPRP self-configuring property makes it immune to such faults, thus enabling detection without disrupting the normal data delivery.

– *AccINDX*: Number of packets received over the IND X network and forwarded to the application. The highest AccINDX corresponds to the fastest network.

- `iPRPreceiverStats`: This tool is used to locally obtain the statistics *PktCtrX*, *LastTimeSeenX*, *WrongINDX* and *AccINDX* at the receiver. In a unicast operation, the argument is the IP address used by the sender to establish the iPRP session with local machine. In multicast operation, it is the used multicast IP address. `iPRPreceiverStats` queries the locally stored statistics table to report the above mentioned fields.

8.7 Measurements with Directional Wi-Fi Links

The ease of deploying Wi-Fi networks in comparison with wired network has led to CPSs using end-to-end Wi-Fi communication or Wi-Fi links in one of the hops. However, off-the-shelf Wi-Fi links are known to have high PLR. With the current trends, the question arises: *Are off-the-shelf directional Wi-Fi links a viable option for reliable real-time communication?* In this section, we answer this question through data collected from a 45-day long measurement campaign conducted at the EPFL campus.

8.7.1 Description of the Test Bed

Figure 8.4 shows the map of the EPFL campus with the three roof-top measurement sites, namely: A, B and C. The distances between sending and receiving antennas are 180 m (site B to site A) and 230 m (site C to site A) and there is a line of sight between the sending and receiving sites. We use directional antennas (shown in Figure 8.5) for transmission and reception, a common practice when Wi-Fi is used as a replacement for a cable in mission-critical networks. The antennas have an 8 dBi gain. The test bed is based on 2.4 GHz Wi-Fi-technology (802.11b standard) and we use different channels and polarizations for the two links to minimize the mutual interference between them.

Sites B and C are used for traffic generation, and site A is used for reception and logging. The locations of sending nodes at sites B and C are same as that of the PMUs at locations ELL and CM in the campus smart-grid described in [53]. Also, site A is present at the same location as the PDC in [53].

For reception, at site A, we use a ruggedized PC [2], equipped with two Wi-Fi cards to support the two desired wireless links. For sending, at sites B and C, we have two Alix2d2 system boards[3]. The ruggedized PC runs 64-bit Ubuntu operating system and Alix2d2's run OpenWrt 10.3 [4] operating system. The machines used in the test-bed are shown in Figure 8.5.

---

[2] http://www.logicsupply.com/da-1000/

[3] http://www.pcengines.ch/alix2d2.htm

[4] https://openwrt.org/

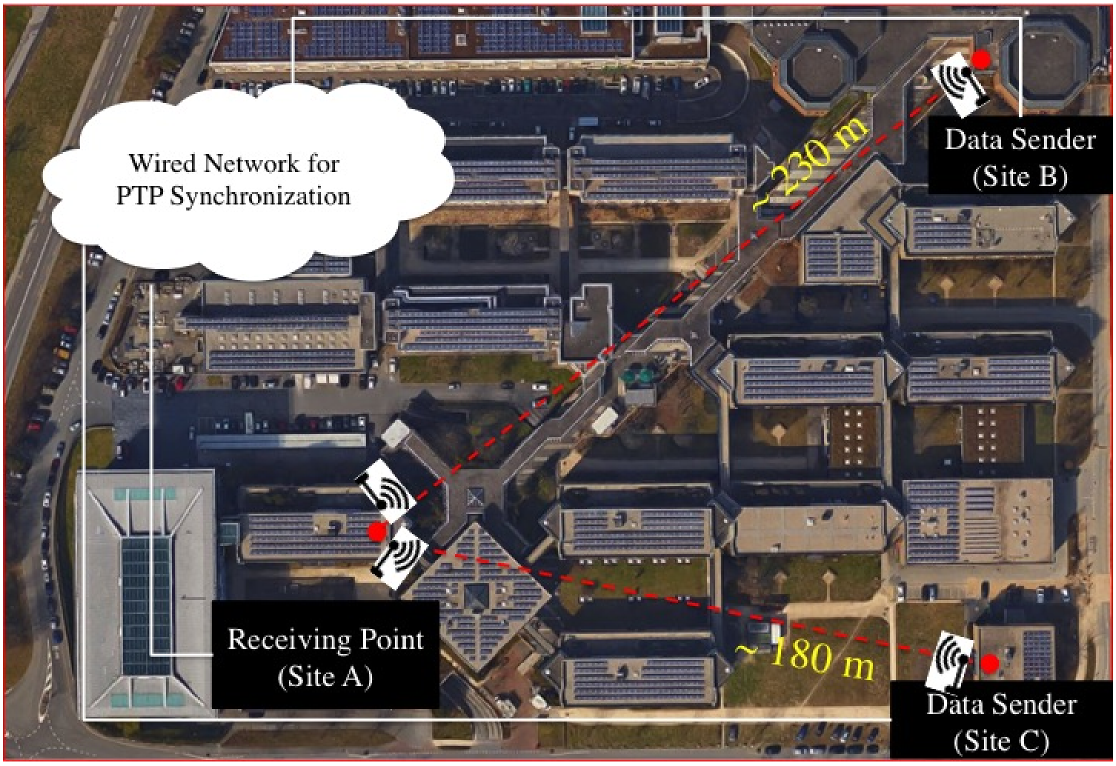


Figure 8.4 – Map of the campus with the antenna locations.

For logging all packets exchanged over both the Wi-Fi links, we use the packet capture tool `tcpdump` [5] on sites A, B and C. To measure the end-to-end latency in transmission of the packets, we have time synchronization among the machines. To this end, we connect the machines through the wired, campus network that is used for network time-synchronization with the PTP [164]. In this way, we can compute the end-to-end latency with a 0.1 ms accuracy.

8.7.2 Experimental Methodology

In order to study the effect of different Wi-Fi parameters (such as raw data-rate, beacon interval, choice of the channel number) on the recorded PLR, we carried out measurements by varying different parameters. Through preliminary measurements, we observed that the choice of the channel number and the beacon interval have practically no effect on the observed PLR and latency. Hence, we omit these factors from the analysis that follows.

The two factors taken into account in our analysis are (1) MAC-layer retransmission enabled/disabled, and (2) the raw data-rate. MAC-layer retransmissions are expected to reduce the PLR because a lost packet might be repaired by the MAC-layer in subsequent transmissions. However, it is worth noting that each retransmission adds to the

---

[5] http://www.tcpdump.org/

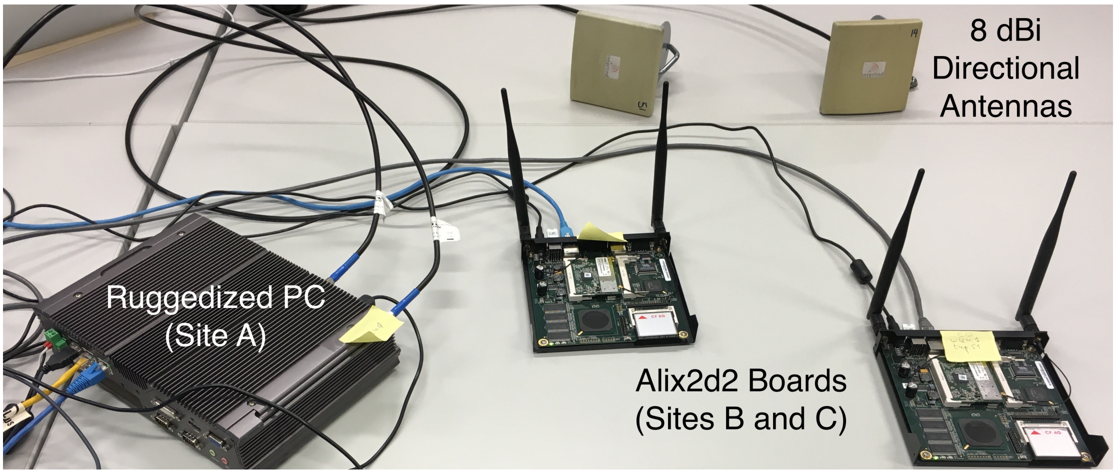


Figure 8.5 – Hardware used for the test-bed.

end-to-end latency. Hence, receiving a packet as a result of MAC-layer retransmissions is a case of trading-off latency for PLR. This trade-off is further elaborated on in Section 8.7.3.

Last, we analyze three different raw data-rates supported by the standard: 1 Mbps, 5.5 Mbps and 11 Mbps. In theory, lower raw data-rates are more robust, due to a higher degree of redundancy from channel coding. Hence, the effect of channel fading is expected to be lower for lower data-rates thereby resulting in a lower overall packet-loss probability.

These factors translate to conducting experiments in six different scenarios: presence or absence of MAC-layer retransmissions, and raw data-rates of 1 Mbps, 5.5 Mbps or 11 Mbps. We randomize these parameters in the measurement scenarios so as to normalize the bias due to individual parameters. Each scenario lasts for nearly 30 minutes, with both the senders sending approximately 90,000 UDP packets of size 300 bytes each, with a packet every 20 ms. This traffic profile is the same as that of the PMUs that stream mission-critical measurements used for state estimation in [53].

The measurement campaign lasted for 45 days resulting in approximately 1500 scenarios, which amounts to 300 million packets being transmitted over the two replicated paths. Furthermore, there were 250 instances of each scenario, amounting to nearly 50 million packets for each scenario.

The packets from both links AB and AC are independently labeled with an increasing SN and a timestamp of the instant at which each packet was generated (t_g). The time at which the packet was actually sent (t_s) is recorded by `tcpdump` at the sender. Note that, due to possible non-negligible delays in the processing of the packet by the operating system, t_s is not necessarily equal to t_g. As the aim of this measurement campaign is to characterize the latency due to Wi-Fi links, using t_s is more apt than using t_g. Hence, the use of `tcpdump` in our measurement test-bed.

At the receiver, for each received packet, the time of reception (t_r) is also logged by `tcpdump`. In the post-processing, the logs of links AB and AC are both analyzed for losses and latency, as described in Section 8.7.3.

8.7.3 Measurement Results

From the measurements of each scenario i, we evaluate the PLR for link AB (P^i_{ab}) and link AC (P^i_{ac}). We compute the availability (A) as $A = MTTF/(MTTR + MTTF) = 1-$PLR, where MTTF and MTTR are the mean-times to failure and recovery of the link, respectively. As a result, we obtain availability for link AB (A^i_{ab}) and link AC (A^i_{ac}).

Consider two packets with SNs x and y sent over links AB and AC, respectively. If $x = y$, then the two packets can be regarded as belonging to the same “generation", i.e., they mimic replicas of each other. Consequently, packet x is said to be lost after replication, if and only the packet is lost on both links AB and AC. As a result, for each scenario i, we obtain the effective PLR as seen by the receiver after replication (P^i_{rep}) and the corresponding availability (A^i_{rep}).

Table 8.1 shows mean, 95th percentile (95th percentile) and 99th percentile (99th percentile) values of P_{ab}, P_{ac}, P_{rep}. We find that the mean PLRs for individual links AB and AC are 9.69×10^{-4} and 2.4×10^{-3} respectively. As expected, these values are much higher than 10^{-5}, the PLR required for mission-critical PMU-streaming applications. However, the mean P_{rep} is 3.58×10^{-6}, is well within the acceptable value for streaming of mission-critical PMU data.

	Mean	95th percentile	99th percentile
P_{ab}	9.69×10^{-4}	0.0013	0.0293
P_{ac}	2.4×10^{-3}	0.0191	0.0466
P_{rep}	3.58×10^{-6}	1.10×10^{-5}	2.22×10^{-5}

Table 8.1 – PLR statistics obtained from measurements

The performance improvement achieved by replication over redundant Wi-Fi paths is more prominent when either of the individual links experiences high losses. We observe, in Table 8.1, that the 99th percentile values of P_{ab}, P_{ac} and P_{rep} are 0.0293, 0.0466 and 2.22×10^{-5}, respectively. We see that even the 99th percentile value of P_{rep} is comparable with the desired PLR of the PMU-streaming applications.

Furthermore, we find the mean availability $A_{rep} = 0.999996$ and its 99th percentile value is 0.99997, which is comparable to the five-9’s requirement of mission-critical PMU-streaming applications. Next, we study the variation of PLR as a function of different parameters and check if the availability can be improved by tuning the parameters of the Wi-Fi links.

Effect of MAC-Layer Retransmission

We studied the effect of MAC-layer retransmissions on loss and latency. Figure 8.6 shows the box plot of P_{ab}, P_{ac} and P_{rep}, with and without MAC-layer retransmissions. Table 8.2 shows the 99^{th} percentile PLR values with and without MAC-layer retransmissions. Although the use of MAC-layer retransmissions provides an order of magnitude improvement in PLRs over individual links, we find absolutely no effective losses over the redundant Wi-Fi paths among the 50 million transmitted. This strongly asserts the use of MAC-layer retransmissions for streaming mission-critical PMU applications.

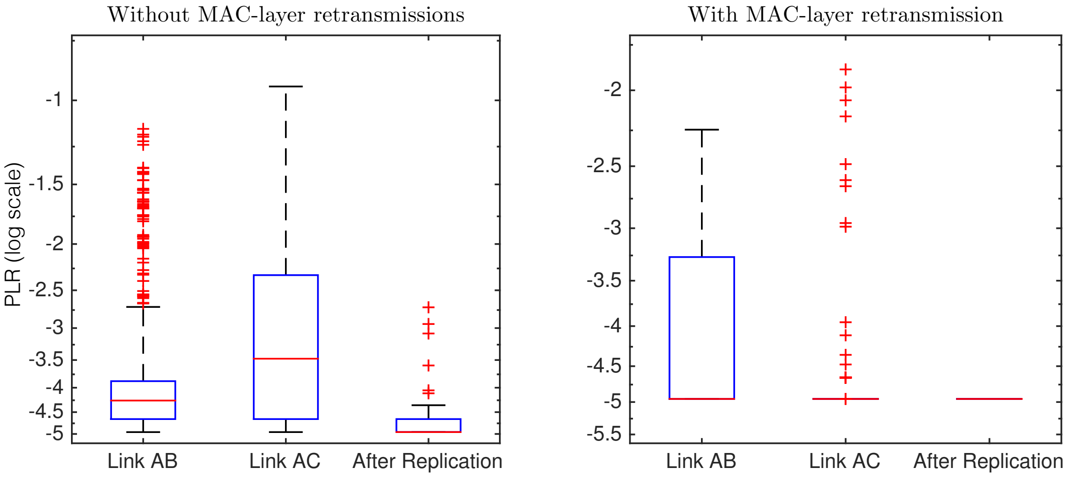


Figure 8.6 – Box plot of the PLRs for link AB, link AC and after replication for scenarios with and without MAC-layer retransmissions, shown in log-scale.

	Without Retransmissions	With Retransmissions
$P_{ab,99\%}$	0.0402	0.0010
$P_{ac,99\%}$	0.0610	0.0018
$P_{rep,99\%}$	4.43×10^{-5}	0

Table 8.2 – 99^{th} percentile value of PLRs with and without MAC-layer retransmissions

Next, we quantify the end-to-end latency and verify if the latency performance with MAC-layer retransmissions conforms to the requirements of the mission-critical PMU-streaming applications (about 4 ms). For this purpose, let the delay due to packet with SN x on link AB be given by $d^x_{ab} = t^{x,ab}_r - t^{x,ab}_s$. Similarly, $d^x_{ac} = t^{x,ac}_r - t^{x,ac}_s$. Then, the effective delay for packet x after replication is given by $d^x_{rep} = \min(d^x_{ab}, d^x_{ac})$.

Table 8.3 shows the mean and quantiles of d_{ab}, d_{ac} and d_{rep}. For most packets, the

	Mean	95^{th} percentile	99^{th} percentile	99.9^{th} percentile	99.99^{th} percentile
d_{ab} (in ms)	0.87	2.58	3.82	4.94	6.32
d_{ac} (in ms)	0.61	0.73	1.26	4.2	6.67
d_{rep} (in ms)	0.58	0.59	0.92	2.69	4.3

Table 8.3 – Latency statistics with MAC-layer retransmissions

latencies of both the individual links and the latency after replication is within the admissible latency of 4 ms, required by mission-critical PMU-streaming applications. However, the tail latencies at 99.9^{th} percentile and 99.99^{th} percentile for individual links exceed the 4 ms mark, indicating that although very rare, there are cases where the real-time requirements are not respected by individual links. Additionally, replication over redundant paths brings down the delay for most of the packets to within the admissible range, barring a minuscule fraction of 10^{-4}.

We find that the mean jitter in d_{ab}, d_{ac} is 0.455 ms and 0.055 ms, whereas the mean jitter in d_{rep} is 0.025 ms. Hence, although individual Wi-Fi links have a high jitter in latency, which is undesirable for PMU-streaming applications, replication over redundant Wi-Fi paths significantly reduces the jitter in latency.

Hence, from the setting we studied, we find that using MAC-layer retransmissions satisfies the loss- and latency-requirements of mission-critical PMU-streaming applications. Thus, we conclude that replicating packets over directional Wi-Fi links is a viable option for streaming PMU measurements.

Effect of Raw Data-Rate

To study the effect of raw data-rate on the PLR over directional Wi-Fi links, we performed experiments with three data-rates: 1 Mpbs, 5.5 Mpbs and 11 Mbps. As lower data-rates have a higher degree of redundancy due to channel-coding, the PLR for a lower data-rate is expected to be lower than that of higher data-rates.

We found that the mean P_{rep} for scenarios with raw data-rate of 1 Mbps, 5.5 Mbps and 11 Mbps are 2.21×10^{-6}, 1.77×10^{-6} and 6.52×10^{-6} respectively. The 99^{th} percentile values of the same were 2.04×10^{-5}, 1.11×10^{-5} and 3.60×10^{-5} respectively. We find that the PLR for 1 Mbps and 5.5 Mbps are quite close to each other, whereas the ones for 11 Mbps are slightly worse. As expected, we suggest the use of a lower raw data-rate.

8.8 Are direction Wi-Fi links fail-independent?

In this section, we characterize the correlation between losses on the two wireless links AB and AC in order to verify whether the directional Wi-Fi links are truly fail-independent.

As a first step towards verifying whether the losses on both the links are statistically correlated, we evaluate the cross-correlation co-efficient of losses on both links. We find the cross-correlation co-efficient to be -0.0005 with a standard deviation of 0.0504. The coefficient of variation is 1.117×10^3, indicating that this first-order statistic cannot be used for a conclusive answer and a formal statistical test, such as the likelihood-ratio test [178] needs to be applied.

The prerequisite for the application of the likelihood-ratio test is the knowledge of the distribution of losses. Furthermore, to be able to apply the test, the distributions of losses on the links AB and AC need to be stationary. The estimation of these distributions for link AB and link AC, and their stationarity is discussed below.

8.8.1 Estimating the Distribution of Losses

We know from the literature that packet losses over wireless links are bursty, i.e., losses are correlated in time. We consider the two-state Gilbert model [227] for representing the observed losses over individual wireless links. Figure 8.7 shows the two-state Markov chain representing the Gilbert model. It consists of the Good state (1) where no packets are lost and the Bad state (0) where the link drops all packets. The transition probabilities from Good state to Bad state, and from Bad state to Good state, are q and p, respectively. Then, the average PLR is given by $q/(p+q)$.

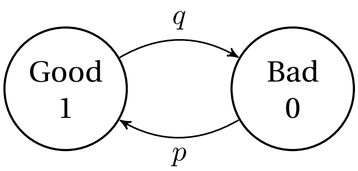


Figure 8.7 – Two-state Gilbert model for bursty losses

From the measurement, for each scenario i, we obtain the parameters of the two-state Gilbert model for link AB and AC: $p^i_{ab}, q^i_{ab}, p^i_{ac}, q^i_{ac}$. For link AB, we find that the mean p_{ab} ($\hat{p}_{ab}$) is 0.4217 and mean q_{ab} ($\hat{q}_{ab}$) is 6.99×10^{-5}. For link AC, we find that the mean p_{ac} ($\hat{p}_{ac}$) is 0.2962 and mean q_{ac} ($\hat{q}_{ab}$) is 3.27×10^{-5}. Figure 8.8 shows the boxplots of parameters for links AB and AC respectively. As seen from the plots, the variance in the parameters of the model is quite high.

Furthermore, from the QQ-plots in Figure 8.9, we see that the distribution of the observed number of losses in the n scenarios differs significantly from the distribution of number of losses if the losses on the links AB and AC were to follow Gilbert($\hat{p}_{ab}, \hat{q}_{ab}$) and Gilbert($\hat{p}_{ac}, \hat{q}_{ac}$), respectively . We conclude that the distribution of losses across

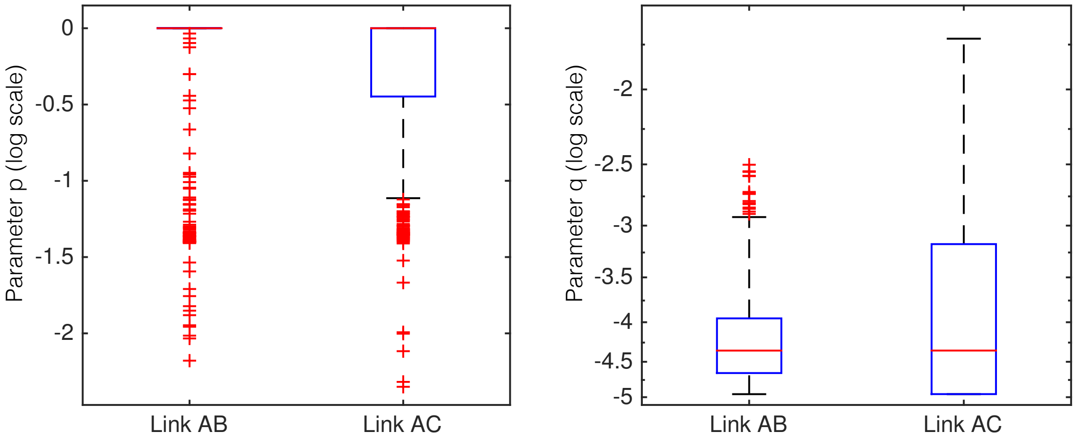


Figure 8.8 – Box plot of parameters of the two-state Gilbert model for links AB and AC, p^i_{ab}, q^i_{ab}, p^i_{ac}, q^i_{ac}, shown in log scale

scenarios is non-stationary and a standard statistical test, such as the likelihood ratio test, cannot be applied. Therefore, we develop an information-theoretic test, based on normalized mutual-information between the losses on the link AB and AC; it uses the bootstrap technique for statistical inference [228]. This information-theoretic test is described below.

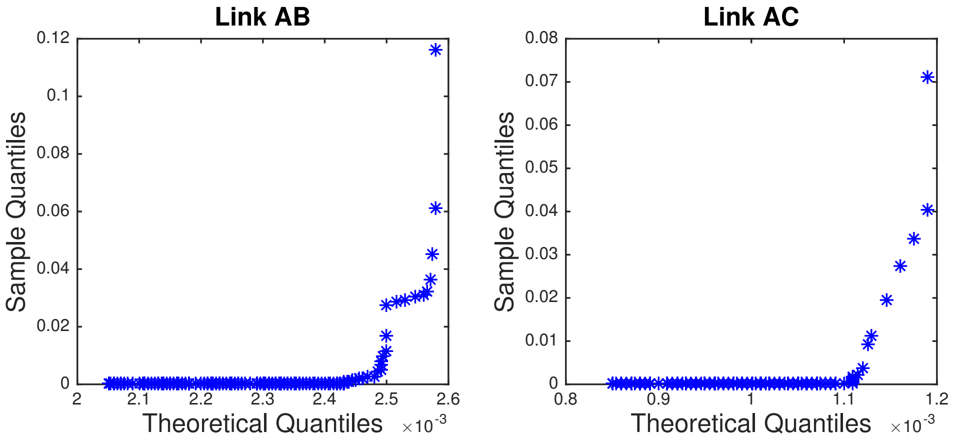


Figure 8.9 – QQ-plot of the number of losses on each link showing the non-stationary of distribution of losses across scenarios

8.8.2 Information-Theoretic Test of Independence

If the losses on the two channels were to be independent, then the Wi-Fi links AB and AC can be represented as two-state Markov chains with transition matrices given by $\alpha = \text{Gilbert}(p_{ab}, q_{ab})$ and $\beta = \text{Gilbert}(p_{ac}, q_{ac})$, respectively. Then, the joint channel is obtained by the product of the Markov chains given by $\Theta_o = \alpha \otimes \beta$, where $\otimes$ represents the tensor product. Alternatively, if the losses on the two links are dependent, then the joint channel of the two wireless links AB and AC is represented by a 4-state Markov chain with states: 1) (Good, Good) 2) (Good, Bad) 3) (Bad, Good) and 4) (Bad, Bad). Let its transition matrix be Θ.

Then, the test of independence is
H_0 : "$\alpha \sim \mathcal{F}_{ab}$, $\beta \sim \mathcal{F}_{ac}$ and the losses follow $\Theta_o = \alpha \otimes \beta$" over H_1 : "The losses follow $\Theta \neq \Theta_o$, $\Theta \sim \mathcal{G}$", where $\mathcal{F}_{ab}, \mathcal{F}_{ac}, \mathcal{G}$ are arbitrary distributions.

The test statistic is mean normalized mutual-information $\mathbb{J} = \frac{\sum_{i=0}^{n} J^i}{n}$, where (J^i) is the normalized mutual-information between the losses on both the links in the i^{th} scenario and is given as follows. Recall that the PLRs for link AB and AC in the i^{th} scenario are P^i_{ab} and P^i_{ac}, respectively. Then, the entropies of the links in the i^{th} scenario are given by

$$H^i(AB) = P^i_{ab} \log\left(\frac{1}{P^i_{ab}}\right) + (1 - P^i_{ab}) \log\left(\frac{1}{1 - P^i_{ab}}\right)$$
$$H^i(AC) = P^i_{ac} \log\left(\frac{1}{P^i_{ac}}\right) + (1 - P^i_{ac}) \log\left(\frac{1}{1 - P^i_{ac}}\right)$$

Let P^i_{xy}, where $x = \{0, 1\}$ and $y = \{0, 1\}$, represent the empirical probability of packet loss (0) and reception (1) on the both the links AB and AC in the i^{th} scenario. Then, the mutual information and the normalized mutual-information in the i^{th} scenario is given by

$$I^i(AB; AC) = \sum_{x=\{0,1\}} \sum_{y=\{0,1\}} P^i_{xy} \log\left(\frac{P^i_{xy}}{P^i_x \times P^i_y}\right)$$
$$J^i = \frac{I^i(AB; AC)}{\sqrt{H^i(AB) \times H^i(AC)}}$$

It is worth noting that normalized mutual-information, consequently $\mathbb{J}$, is the information-theoretic analogue to covariance. In the ideal case, i.e., in the absence of measurement noise, if the losses were independent, then $\mathbb{J} = 0$. Hence, if H_0 is true, we expect $\mathbb{J}$ to be small. Furthermore, $\mathbb{J} \geq 0$. Therefore, the test's rejection region is $\mathbb{J} > \eta$, where η is needed to be computed as a function of the confidence level.

For a confidence level of 99%, we have to evaluate $\eta_{0.01}$ such that, under H_0, $\mathbb{P}_{H_0}(\mathbb{J} > \eta_{0.01}) < 0.01$. However, $\mathbb{P}_{H_0}(\mathbb{J} > \eta_{0.01}) = \mathbb{P}(\mathbb{J} > \eta_{0.01} | \mathcal{F}_{ab}, \mathcal{F}_{ac})$. From the theory of bootstrapping [228], this can be approximated as $\mathbb{P}(\mathbb{J} > \eta_{0.01} | \hat{\mathcal{F}_{ab}}, \hat{\mathcal{F}_{ac}})$, where $\hat{\mathcal{F}_{ab}}$ and $\hat{\mathcal{F}_{ac}}$ are the empirical distributions of the parameters of the Gilbert model for link AB and link AC, respectively.

To evaluate $\eta_{0.01}$, we perform a generation of losses on the two links under H_0, according to the bootstrap method shown in Algorithm 8.6. In each run, we randomly

Algorithm 8.6: Bootstrapping the generation of losses

```
currentRun $\leftarrow$ 0;
$\hat{\mathcal{F}}_{ab} = \{\alpha_1, \alpha_2, ..., \alpha_n\}$;
$\hat{\mathcal{F}}_{ac} = \{\beta_1, \beta_2, ..., \beta_n\}$;
while currentRun $<$ numRuns do
    i,j $\leftarrow$ random_integer(1,n);
    $\Theta_{sim} = \alpha_i \otimes \beta_j$;
    Pkts$_{ab}$, Pkts$_{ac}$ $\leftarrow$ generate_packets($\Theta_{sim}$);
    currentRun++;
end
```

select (with replacement) one set of parameters of the Gilbert model for link AB and link AC, from among the parameters observed from real measurements. From these parameters, under H_0, we obtain Θ_{sim}, the distribution of the losses on the two channels in the current simulation run. Θ_{sim} is used to generate a large number of packets (10^6) on the two links (`Pkts`$_{ab}$, `Pkts`$_{ac}$). From the generated packets, $\mathbb{J}_{sim}$ is evaluated. This process is repeated `numRuns` times to obtain several independent values of $\mathbb{J}_{sim}$. Then, $\eta_{0.01}$ is the 99^{th} percentile value of $\mathbb{J}_{sim}$.

Using `numRuns` = 10^6, we obtain $\eta_{0.01} = 3.82 \times 10^{-5} \pm 1.4 \times 10^{-5}$ at a 99% confidence level. From the measurements, we have $\mathbb{J} = 8.91 \times 10^{-4} > \eta_{0.01}$. Hence, we reject H_0 with 99% confidence level and conclude that the two links are not fail-independent.

8.8.3 Impact of Dependent Losses on P_{rep}

In Section 8.8.2, we found that losses on the two Wi-Fi links AB and AC are not independent. The mean normalized mutual-information calculated from the measurements ($\mathbb{J} = 8.91 \times 10^{-4}$) is a measure of mutual information between the two channels. Its low value indicates that, although the losses are not independent, the dependence is low.

The low degree of dependence is also noticeable from the PLRs. From the measurements, we have $P_{ab} = 9.69 \times 10^{-4}$, $P_{ac} = 2.4 \times 10^{-3}$. Hence, if the losses were independent, the effective PLR would be $P_{indep} = P_{ab} \times P_{ac} = 2.32 \times 10^{-6}$. From the measurements we have $P_{rep} = 3.58 \times 10^{-6}$, which is close to P_{indep}. Therefore, we conclude that although the two Wi-Fi links are not fail independent, the effective loss performance, as observed after replication, is close to what it would have been if they were to be fail-independent.

8.9 Conclusion

In this chapter, we addressed the issue of reliability for real-time CPS traffic with the focus on 0 ms repair for packets exchanged in WANs. We proposed iPRP, an IP-friendly parallel-redundancy protocol. iPRP is a transport-layer protocol that selectively replicates UDP traffic over two or more fail-independent paths. It is designed to be network transparent and only requires a software installation on the end hosts. It is also designed to be application agnostic and can replicate packets, without any modifications to the existing applications.

We presented the design of iPRP and showed how it handles the support of IP multicast where several receivers can benefit from reliable communication through replicated traffic. We also designed a new de-duplication algorithm that ensures that at most one copy of a packet is forwarded to the application even in the presence of network losses, delays or packet re-orderings. We formally proved this guarantee of the discard algorithm. We also described the iPRP diagnostic toolkit that can be used to debug network issues in networks with one or more network subclouds, to collect statistics on the performance of the network subclouds, and to identify the existence of parallel fail-independent paths. We also made an implementation of iPRP publicly available.

We experimentally validated the feasibility of using Wi-Fi measurements over redundant paths for streaming mission-critical PMU data. In the measurement testbed we used, the location of the measurement sites and the traffic profile are the same as those used in the PMU-based state estimation in the campus-wide active distribution-network described in [53]. Such a setting is also commonly encountered when the last hop of a WAN is realized using Wi-Fi, where directional antennas are used to boost the reliability of individual links.

In the setting we evaluated, we conclude that although the PLRs of individual Wi-Fi links were far from admissible for streaming mission-critical PMU data, packet replication over the two Wi-Fi paths provided the desired level of reliability (PLR $\sim 10^{-5}$). The effective PLR and availability can be further improved by enabling MAC-layer retransmissions on each Wi-Fi link. The end-to-end latency was observed to be within 4 ms, the required latency by mission-critical PMU-streaming applications. The mean jitter in the latency was measured to be very low (0.02 ms), further bolstering the usability of Wi-Fi over redundant paths.

From the measurements, we observe that although the two links were found to not be fail-independent, the effective PLR as observed after replication was very close to the product of the two PLRs. Thus, for all practical purposes, the losses on the two links can be thought of as being independent.

We concluded, based on these results, that replication over redundant Wi-Fi links, formed from off-the-shelf components, is a viable option for achieving the loss and latency performance required for streaming mission-critical PMU data.

The requirement of reliable delivery of real-time traffic is not unique to CPS. Several other fields such as high-frequency trading and multiplayer online gaming require low-latency traffic and can benefit from parallel redundancy. As iPRP is designed to be application independent, it can be used in these fields without any modifications. Moreover, our open-source implementation can also be used to develop proof-of-concepts for these fields, paving the way for easy adoption.

9 Conclusion and Open Questions

The questions are always
more important than the answers.
— Randy Pausch, The Last Lecture

We have studied the problem of providing reliability for real-time CPSs in the presence of delay and crash faults due to unreliable software, and in the presence of delay and losses due to unreliable communication network. Specifically, to address the issue of delay and crash faults in the central controller in the CPS due to the use of COTS hardware and software components, we have proposed a fault-tolerance architecture (Axo) that uses active replication and ensures linearizability by using intentionality clocks for ordering messages and Quarts for ensuring replica-consistency. To address the problem of packet losses, which is exacerbated due to communication in WANs or using unreliable communication technologies such as Wi-Fi or LTE, we design iPRP, an IP-friendly parallel redundancy protocol. We also study the viability of replicating packets over directional Wi-Fi links for achieving high-reliability.

The central elements of all the reliability mechanisms developed in this thesis are the abstract models of the software agents in the CPS, namely a controller and a PA. These models have been presented in Chapter 3. These models have been developed by studying a wide-range of real-world CPSs, such as CPS for real-time control of electric grids [1, 4, 5, 6], those used in manufacturing processes [13], and in autonomous vehicles [8, 9]. The key distinguishing features of our proposed controller model is the splitting of the computation performed by the controller into two functions: the `ready_to_compute` function that decides whether the controller has enough information about the physical process to execute the `compute` function for computing setpoints. This finer view of the controller makes our model more expressive and enables us to design more efficiently reliability mechanisms.

In Chapter 3, we have also abstracted the execution trace of CPS in terms of the sending, reception and timeout events that occur at its software agents, and the network and computation relations that cause those events. We use these relations to introduce the intentionality relation that uses the events in a CPS to capture the state of the physical process. To this end, the intentionality relation views the CPS as a sequence of events, each event intending to cause a particular change in the physical process, which either succeeds or fails in causing the change due to other competing events or network losses. The intentionality relation is defined only for CPSs with controllers and PAs and no asynchronous sensors that directly communicate with the PAs. A possible avenue for future work is to extend the intentionality relation to CPSs with such sensors. The secret lies in answering the question, *What change in the physical process does an asynchronous sensor intend to cause?* Reasoning about the controller and the PAs in the light of the proposed models and the intentionality relation enables us to identify and prove the correctness properties of CPSs. The properties, for delay and crash fault-tolerance, considered in this thesis, are state safety, optimal selection, consistency and timeliness. For a different fault-model such as Byzantine faults, the question *“What are the properties of the CPS that ensure correct control of the physical process despite malicious software agents?"* remains to be explored.

In Chapter 4, we have presented intentionality clocks, a labeling mechanism that provides the strong clock-consistency property under the intentionality relation, for CPSs with one or more replicated controllers. We used intentionality clocks to design a controller and a PA that ensures the state safety and optimal selection properties. We proved these guarantees and illustrated their importance through an example of CPS for charging of electric vehicles. In this CPS, when intentionality clocks are not used, we show that the optimal selection property is violated and a deadlock state is reached. We also shown through another example that a mechanism that uses timestamps obtained from physical clocks can violate the strong clock-consistency property under the intentionality relation. As timestamps are often used in the messages exchanged by real-time CPSs, it is desirable to reuse the same timestamps for ordering, without any extra bandwidth overhead. However, there is an open question *Does there exist a mechanism using physical clocks that provides strong clock-consistency under the intentionality relation?*

In Chapter 5, we have used the intentionality clocks to design Quarts, an agreement protocol for replicated controllers that guarantees the consistency correctness property and adds a bounded-latency overhead. Quarts applies to CPSs with only controllers and PAs. We have proposed an extension, Quarts+ that ensures the same guarantees as Quarts, in CPS with PAs and sensors. Both Quarts and Quarts+, perform agreement on input to the controller replicas. For this, they use a deterministic voting mechanism that uses a predetermined priority among the sets of inputs to rank the inputs. In some cases, this enables a controller replica performing Quarts (or Quarts+), to decide on the chosen inputs, without communicating with other replicas. Through extensive simulation for different scenarios, we have shown that out proposed mechanisms increase the chances of reaching agreement by over an order of magnitude, when compared to the conventional approach, *i.e.,* consensus. In the current design of the predetermined priority list for the sets of inputs, we have considered a set with more inputs to have a higher priority. This, however, discounts the importance of some PAs and sensors that are more important for the control of the physical process at a given time. A question worth exploring, *How can the predetermined priority function be modified such that it maximizes the quality of control of the physical process?*

In Chapter 6, we have proposed Axo, a fault-tolerance architecture for delay and crash faults. Axo uses active replication, and relies on intentionality clocks and Quarts for providing linearizability, a correctness requirement for active replication. Axo adds another mechanism to provide the timeliness correctness property. It also quickly detects delay and crash faulty replicas and reboots them to maximize the number of non-faulty replicas at all times, thereby increasing availability. We have proved the timeliness guarantees of Axo and derive and validate bounds on the time Axo to detect and recover faulty replicas. Axo uses exponential averaging to keep track of the health of delay faulty replicas; and it penalizes a replica when its health falls below a threshold. Although this approach is effective in addressing some patterns with a higher rate of delay faults, it fails to catch patterns with alternating faults such as 10101010 and 110011001100, where 1 represents no fault in a round and 0 represents a fault. If such faults are left undetected, then the faults on different replicas might occur in the same rounds, and result in reduced availability. An interesting question to address would be, *How can recurring patterns in delay faults be detected to ensure that faults on different replicas do not happen in the same round?*

In Chapter 6, we have also evaluated the fault-tolerance properties of Axo, Quarts and intentionality clocks with COMMELEC and with an inverted-pendulum CPS. In each of these case-studies, we have shown that the fault detection and recovery by Axo improve the control performance, such as MTTI of the inverted-pendulum. Axo also relies on the knowledge of the validity horizon of the setpoints in a CPS. Whereas in some CPSs such as COMMELEC, the validity horizon is easy to obtain from the validity of the short-term predictions uses in the measurements, obtaining a good estimate on the validity horizon in other CPSs is non-trivial. An open question for CPSs in general is *How to estimate the validity horizon of setpoints in a CPS?* We have also shown that using Quarts to prevent inconsistency improves the ability of COMMELEC to implement an auxiliary service, such as provide autonomy in a microgrid, *i.e.,* minimize the total power imported or exported by the microgrid. Quarts prevents inconsistency at the expense of availability, inconsistent setpoints in some CPSs might be tolerable due to the nature of the physical process. In such cases, it is worth exploring *whether the consistency property of Quarts can be relaxed to further improve availability without jeopardizing the control of the physical process?*

In Chapter 7, we have proposed QCL that uses our proposed reliability mechanisms (intentionality clocks and Quarts) to improve the latency performance of SDN controllers. Existing replication mechanisms in SDN, which are designed for low-latency, only provide eventual-consistency guarantees. As a result, they limit the types of enforceable policies and do not permit reactive controller applications (such as NATs and firewalls) to be implemented by the SDN controllers. We show that QCL can provide strong consistency and can support reactive controller applications, and greatly reduce the latency of issuing updates, when compared to other such schemes. Enterprise systems such as datacenters also use other architectures similar to SDN, such as NFV [29] and OpenStack Neutron [229]; they separate the control plane from the data plane. As low-latency is highly desirable for such systems, an interesting exercise is to explore *what other applications in enterprise systems could benefit from control-plane replication using Quarts?*

In Chapter 8, we have addressed the problem of reliable real-time communication in CPSs that communicate over WANs. To this end, we have designed iPRP for replicating packets on two or more parallel paths, to ensure 0 ms repair of packets lost either due to congestion or due to component failures. We have provided an implementation of iPRP: It is plug-and-play, and only requires installation on the communicating end hosts, without any modification to the CPS applications or the interconnecting network. As iPRP and other such solutions rely on the existence of fail-independent paths and as CPSs are increasingly using Wi-Fi based communication, we have experimentally validated the fail-independence of directional Wi-Fi links, through a 45-day measurement campaign on the rooftops of the EPFL campus. We find that, although the links are not fail independent, the correlation of losses is so low that the effective PLR and delay is close to what it would have been if the links were indeed fail independent. This paves way for using directional Wi-Fi links for mission-critical real-time CPS applications.

A drawback of iPRP is that it applies only to UDP flows, which are not congestion-aware, thereby making iPRP not TCP-friendly [230]. In order for a communication protocol to be used by CPSs to communicate over a public network with TCP flows, it is desirable to have TCP-friendliness. QUIC is a UDP-based transport protocol that performs congestion-control, thus is TCP-friendly. Therefore, it is a good candidate for supporting parallel redundancy. Such a solution requires that all packets be forwarded on all network subclouds which in turn requires that the data-rates and delays on the subclouds be symmetric. However, asymmetric congestion on the paths results in different data-rates, thereby making true redundancy challenging. The question to be addressed here is, *How do we schedule packets on different network subclouds while being congestion-aware and minimizing the PLR?*

The reliability mechanisms developed in this thesis are designed to tolerate only delay and crash faults. They do not address Byzantine faults. An open question is, *Can Quarts be used to lower the latency-overhead of BFT solutions so that they can be* *applied to real-time CPSs?*

We note that the reliability mechanisms proposed in this thesis do not directly apply to CPSs with distributed controllers, rather only those with one centralized controller. However, an easy extension of this work could be in the context of CPSs with hierarchical controllers such as the composable versions of COMMELEC [4] and [13]. Quarts and Quarts+ are concerned with only one layer of control. Thus, they are agnostic to hierarchy. However, the correctness properties of Axo and intentionality clocks depend on the setpoints received from the parent controller. We hope that with some minor modifications the additional information from the parent controller can be incorporated into the design. The open question is, *What changes would be required to Axo and intentionality clocks to prove their composability?*

Lastly, we realize that although Axo and iPRP are designed to be controller-agnostic, the reliability mechanisms that provide linearizability, namely, intentionality clocks and Quarts are not controller-agnostic. However, in Chapter 7, we were able to use intentionality clocks and Quarts without any modifications to the SDN controller. The important property of the SDN controller, which enabled us to do so is that, the `ready_to_compute` function decides based only on the measurements and not on the time since the last setpoint issued (as done by COMMELEC [4]). Moreover, when the controller finishes computation, it notifies QCP of the completion. It is indeed desirable to achieve linearizability in a controller-agnostic fashion, as this would make the fault-tolerance protocol minimally intrusive. Thus, the following question arises: *What are the necessary and sufficient properties of a CPS that enable the design of a CPS-agnostic fault-tolerance scheme?*

A Discussion on Theorems in Section 3.6

Here, we derive the results of the theorems in Section 3.6. As noted earlier, we prove these theorems for CPSs that implement Algorithms 4.1, 4.2, and 4.3. For this, we use the result of Theorem 4.5.1, which follows from Lemmas 4.5.1, 4.5.2, 4.5.3 and 4.5.4. For readability, the theorem and lemmas are re-stated here.

Theorem (2.8.1)**.** *In a CPS that implements Algorithms 4.1, 4.2, 4.3: for any two events* a *and* b*,*

$$C(a) < C(b) \iff a \rightarrow b$$

$$C(a) = C(b) \text{ and } a.pa = b.pa \iff a \equiv b$$

The following lemmas hold for CPSs that implement intentionality clocks using Algorithms 4.1, 4.2, and 4.3.

Lemma (2.8.1)**.** $a \equiv b \implies C(a) = C(b)$ *and* $a.pa = b.pa$.

Lemma (2.8.2)**.** $C(a) = C(b)$ *and* $a.pa = b.pa \implies a \equiv b$.

Lemma (2.8.3)**.** $a \rightarrow b \implies C(a) < C(b)$.

Lemma (2.8.4)**.** $C(a) < C(b) \implies a \rightarrow b$.

A.1 Proof of Theorem 3.6.1

Theorem. *"Intentional equivalence" is reflexive, symmetric, transitive.*

Proof. Properties (1)-(5) of the intentional equivalence relation (Definition 3.6.1) can be observed to be reflexive, symmetric, and transitive. Rather than enumerate the several cases in order to prove this, we use the results obtained in Lemmas 4.5.1, 4.5.2 to provide a much more concise proof.

Reflexive: Consider event a.
We have $C(a) = C(a)$.
Therefore, by Lemma 4.5.2: $C(a) = C(a) \implies a \equiv a$.

Symmetric: Consider events a and b, such that $a \equiv b$.
By Lemma 4.5.1: $a \equiv b \implies C(a) = C(b)$.
Then, $C(b) = C(a)$.
Therefore, by Lemma 4.5.2: $C(b) = C(a) \implies b \equiv a$.

Transitive: Consider events a, b, c, such that $a \equiv b$, $b \equiv c$.
By Lemma 4.5.1: $a \equiv b \implies C(a) = C(b)$.
By Lemma 4.5.1: $b \equiv c \implies C(b) = C(c)$.
Then, $C(a) = C(c)$.
Therefore, by Lemma 4.5.2: $C(a) = C(c) \implies a \equiv c$. □

A.2 Proof of Theorem 3.6.2

Theorem. *For any two events* a, b*:* $a \rightarrow b \implies b \not\rightarrow a$.

Proof. We prove this by contradiction.
Let $a \rightarrow b$ and $b \rightarrow a$.
From Lemma 4.5.4, $a \rightarrow b \implies C(a) < C(b)$.
From Lemma 4.5.4, $b \rightarrow a \implies C(b) < C(a)$.
Contradiction.
Therefore, $b \not\rightarrow a$. □

A.3 Proof of Theorem 3.6.3

Theorem. *For any two events* a, b, *such that* $a.pa = b.pa$*:*
$(a \not\rightarrow b \wedge b \not\rightarrow a) \iff a \equiv b$.

Proof. The statement has two parts.
Part 1: ($a.pa = b.pa$, $a \not\rightarrow b$) and $b \not\rightarrow a \implies a \equiv b$
From the converse of Lemma 4.5.4: $a \not\rightarrow b \implies C(a) \not< C(b)$.
From the converse of Lemma 4.5.4: $b \not\rightarrow a \implies C(b) \not< C(a)$.
If $C(a) \not< C(b)$ and $C(b) \not< C(a)$, then $C(a) = C(b)$.
By Lemma 4.5.2: $(C(a) = C(b) \wedge a.pa = b.pa) \implies a \equiv b$.

Part 2: ($a.pa = b.pa$ and $a \equiv b$) $\implies$ ($a \not\rightarrow b$ and $b \not\rightarrow a$)
From Lemma 4.5.1: $a \equiv b \implies C(a) = C(b)$.
Thus, $C(a) \not< C(b)$ and $C(b) \not< C(a)$.

From the converse of Lemma 4.5.3: $C\left(a\right) \nless C\left(b\right) \implies a \not\to b$.
From the converse of Lemma 4.5.3: $C\left(b\right) \nless C\left(a\right) \implies b \not\to a$.

$\square$

Bibliography

[1] 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.

[2] O. E. Dictionary, “Oxford english dictionary online,” *Mount Royal College Lib., Calgary*, vol. 14, 2004.

[3] I. Dumitrache, “The next generation of cyber-physical systems,” *Journal of Control Engineering and Applied Informatics*, vol. 12, no. 2, pp. 3–4, 2010.

[4] 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.

[5] 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,” *Smart Grid, IEEE Transactions on*, vol. 5, no. 2, pp. 622–631, 2014.

[6] 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.

[7] A. Bernstein, N. Bouman, and J.-Y. Le Boudec, “Real-Time Control of an Ensemble of Heterogeneous Resources,” in *Proceedings of the 56th IEEE Conference on Decision and Control.* IEEE, 2017.

[8] 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,” 2007.

[9] 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.

[10] G. N. Roberts and R. Sutton, *Advances in unmanned marine vehicles.* Iet, 2006, vol. 69.

[11] D. Floreano and R. J. Wood, “Science, technology and the future of small autonomous drones,” *Nature*, vol. 521, no. 7553, p. 460, 2015.

[12] C. Urmson, C. Baker, J. Dolan, P. Rybski, B. Salesky, W. Whittaker, D. Ferguson, and M. Darms, “Autonomous driving in traffic: Boss and the urban challenge,” *AI magazine*, vol. 30, no. 2, p. 17, 2009.

[13] 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.

[14] J. Lee, B. Bagheri, and H.-A. Kao, “A cyber-physical systems architecture for industry 4.0-based manufacturing systems,” *Manufacturing Letters*, vol. 3, pp. 18–23, 2015.

[15] 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.

[16] “Cyber-physical system,” https://en.wikipedia.org/wiki/Cyber-physical_system, accessed: 2018-06-28.

[17] 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.

[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] 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.

[20] A. G. Phadke, “Synchronized phasor measurements in power systems,” *IEEE Computer Applications in power*, vol. 6, no. 2, pp. 10–15, 1993.

[21] G. Lipari, P. Gai, M. Trimarchi, G. Guidi, and P. Ancilotti, “A hierarchical framework for component-based real-time systems,” in *International Symposium on Component-Based Software Engineering*. Springer, 2004, pp. 209–216.

[22] K. Sun, L. Zhang, Y. Xing, and J. M. Guerrero, “A distributed control strategy based on dc bus signaling for modular photovoltaic generation systems with battery energy storage,” *IEEE Transactions on Power Electronics*, vol. 26, no. 10, pp. 3032–3045, 2011.

[23] V. Musolino, P.-J. Alet, L.-E. Perret-Aebi, C. Ballif, and L. Piegari, "Alleviating power quality issues when integrating PV into built areas: Design and control of dc microgrids," in *DC Microgrids (ICDCM), 2015 IEEE First International Conference on.* IEEE, 2015, pp. 102–107.

[24] F.-L. Lian, J. Moyne, and D. Tilbury, "Network design consideration for distributed control systems," *IEEE Transactions on Control Systems Technology*, vol. 10, no. 2, pp. 297–307, 2002.

[25] Y. Xia, M. Fu, and G.-P. Liu, *Analysis and synthesis of networked control systems.* Springer Science & Business Media, 2011, vol. 409.

[26] X. Ge, F. Yang, and Q.-L. Han, "Distributed networked control systems: A brief overview," *Information Sciences*, vol. 380, pp. 117–131, 2017.

[27] N. McKeown, "Software-defined networking," *INFOCOM keynote talk*, vol. 17, no. 2, pp. 30–32, 2009.

[28] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[29] K. Joshi and T. Benson, "Network function virtualization," *IEEE Internet Computing*, vol. 20, no. 6, pp. 7–9, 2016.

[30] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015.

[31] A. Panda, W. Zheng, X. Hu, A. Krishnamurthy, and S. Shenker, "SCL: Simplifying Distributed SDN Control Planes." in *NSDI*, 2017, pp. 329–345.

[32] E. A. Lee, "Cyber physical systems: Design challenges," in *Object oriented real-time distributed computing (isorc), 2008 11th ieee international symposium on.* IEEE, 2008, pp. 363–369.

[33] J. A. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation systems," *Computer*, vol. 21, no. 10, pp. 10–19, 1988.

[34] 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.

[35] 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 *20th Power Systems Computation Conference (PSCC 2018)*, no. CONF, 2018.

[36] H. Farhangi, “The Path of the Smart Grid,” *Power and Energy Magazine, IEEE*, vol. 8, no. 1, pp. 18–28, 2010.

[37] D. M. Stavens, *Learning to drive: Perception for autonomous cars.* Stanford University, 2011.

[38] O. Alsac and B. Stott, “Optimal load flow with steady-state security,” *IEEE transactions on power apparatus and systems*, no. 3, pp. 745–751, 1974.

[39] 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.

[40] J. Nilsson *et al.*, “Real-time control systems with delays,” 1998.

[41] R. E. Kalman, “A New Approach to Linear Filtering and Prediction Problems,” *Journal of basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960.

[42] S. H. Fuller and L. I. Millett, “Computing performance: Game over or next level?” *Computer*, vol. 44, no. 1, pp. 31–38, 2011.

[43] 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.

[44] N. Instruments, “ NI Compactrio,” http://www.ni.com/compactrio/, accessed: 2015-02-15.

[45] Alstom, “ Digital Automation Platform Server,” http://www.alstom.com/grid/products-and-services/Substation-automation-system/dap-substation-server/, accessed: 2015-02-15.

[46] ABB, “MGC600,” https://new.abb.com/docs/default-source/ewea-doc/microgrid-controller-600_en_lr(dic2013).pdf, accessed: 2015-02-15.

[47] B. Automation, “Automation PC 910,” https://www.br-automation.com/en/products/industrial-pcs/automation-pc-910/, accessed: 2015-02-15.

[48] W. Bolton, *Programmable logic controllers.* Newnes, 2015.

[49] M. Barabanov and V. Yodaiken, “Real-time linux,” *Linux journal*, vol. 23, no. 4.2, p. 1, 1996.

[50] I. Tesla Motors, “Git hub Tesla motors,” https://github.com/teslamotors, accessed: 2016-06-06.

[51] K. G. Shin and P. Ramanathan, “Real-time computing: A new discipline of computer science and engineering,” *Proceedings of the IEEE*, vol. 82, no. 1, pp. 6–24, 1994.

[52] K. C. Budka, J. G. Deshpande, and M. Thottan, “Communication Networks for Smart Grids,” in *Computer Communications and Networks.* Springer, 2014.

[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] 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.

[55] F. Consortium *et al.*, “Flexray communications system-protocol specification,” *Version*, vol. 2, no. 1, pp. 198–207, 2005.

[56] H. Charara, J.-L. Scharbarg, J. Ermont, and C. Fraboul, “Methods for bounding end-to-end delays on an AFDX network,” in *Real-Time Systems, 2006. 18th Euromicro Conference on.* IEEE, 2006, pp. 10–pp.

[57] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer, *The time-triggered ethernet (TTE) design.* IEEE, 2005.

[58] 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.

[59] 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.

[60] A. Sheth, S. Nedevschi, R. Patra, S. Surana, E. Brewer, and L. Subramanian, “Packet loss characterization in WiFi-based long distance networks,” in *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE.* IEEE, 2007, pp. 312–320.

[61] 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 *22nd IEEE International Conference on Emerging Technologies And Factory Automation.* IEEE, 2017.

[62] 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.

[63] J. Wan, H. Yan, D. Li, K. Zhou, and L. Zeng, “Cyber-physical systems for optimal energy management scheme of autonomous electric vehicle,” *The Computer Journal*, vol. 56, no. 8, pp. 947–956, 2013.

[64] M. Elattar and J. Jasperneite, “Using LTE as an access network for internet-based cyber-physical systems,” in *Factory Communication Systems (WFCS), 2015 IEEE World Conference on.* IEEE, 2015, pp. 1–7.

[65] “Northeast blackout of 2003,” https://en.wikipedia.org/wiki/Northeast_blackout_of_2003, accessed: 2018-06-28.

[66] G. Andersson, P. Donalek, R. Farmer, N. Hatziargyriou *et al.*, “Causes of the 2003 Major Grid Blackouts in North America and Europe and Recommended Means to Improve System Dynamic Performance ,” *Power Systems, IEEE Transactions on*, vol. 20, no. 4, pp. 1922–1928, Nov 2005.

[67] J. Jacinto, “Automation services reduce downtime for manufacturers,” https://www.totallyintegratedautomation.com/2009/10/automation-services-reduce-downtime-for-manufacturers/, accessed: 2016-06-06.

[68] R. I. Association, “Industrie 4.0 reducing downtime in automotive industry,” https://www.controleng.com/single-article/industrie-40-reducing-downtime-in-automotive-industry/c81e25bc19ea692b235431f4007996d2.html, accessed: 2018-06-06.

[69] S. Kane, E. Liberman, T. DiViesti, and F. Click, “Toyota Sudden Unintended Acceleration has Killed 89,” *Safety Research & Strategies*, 2010.

[70] A. Klein, “Tesla driver dies in first fatal autonomous car crash in US,” *New Scientist*, 2016.

[71] E. Ackerman, “Fatal tesla self-driving car crash reminds us that robots aren’t perfect,” *IEEE-Spectrum*, vol. 1, 2016.

[72] H. Kirrmann, “§2.5 Dependable Automation,” *Collaborative Process Automation Systems*, p. 100, 2010.

[73] “Split-Brain (Computing),” https://en.wikipedia.org/wiki/Split-brain_(computing), accessed: 2018-06-28.

[74] A. Avizienis and J. P. Kelly, “Fault Tolerance by Design Diversity: Concepts and Experiments,” *Computer*, vol. 17, no. 8, pp. 67–80, 1984.

[75] A. Avizienis, J.-C. Laprie, B. Randell *et al.*, *Fundamental Concepts of Dependability.* University of Newcastle upon Tyne, Computing Science, 2001.

[76] 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.

[77] Y. C. Yeh, “Triple-triple redundant 777 primary flight computer,” in *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, vol. 1. IEEE, 1996, pp. 293–307.

[78] T. E. Oliphant, “Python for scientific computing,” *Computing in Science & Engineering*, vol. 9, no. 3, 2007.

[79] M. Castro, B. Liskov *et al.*, “Practical Byzantine Fault Tolerance,” in *OSDI*, vol. 99, 1999, pp. 173–186.

[80] H. Khurana, M. Hadley, N. Lu, and D. Frincke, “Smart-grid security issues,” *Security Privacy, IEEE*, vol. 8, no. 1, pp. 81–85, Jan 2010.

[81] H. Weibel, “Tutorial on Parallel Redundancy Protocol (PRP),” 2003.

[82] J. Gray, “Why do Computers Stop and what can be Done about It?” 1986.

[83] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011.

[84] L. Lamport, “The Part-Time Parliament,” *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[85] ——, “Time Clocks and the Ordering of Events in a Distributed System,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[86] 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.

[87] P. Mahajan, L. Alvisi, and M. Dahlin, “Consistency, Availability, and Convergence,” *University of Texas at Austin Tech Report*, vol. 11, 2011.

[88] 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.

[89] W. Vogels, “Eventually consistent,” *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[90] D. Bermbach and S. Tai, “Eventual consistency: How soon is eventual? an evaluation of amazon s3’s consistency behavior,” in *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing.* ACM, 2011, p. 1.

[91] P. Bailis and A. Ghodsi, “Eventual consistency today: Limitations, extensions, and beyond,” *Queue*, vol. 11, no. 3, p. 20, 2013.

[92] 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 *EEE Transactions on Industrial Informatics*, no. EPFL-ARTICLE-234511, 2018.

[93] J. McCauley, "The POX network software platform," last accessed: 2018-06-10. [Online]. Available: https://noxrepo.github.io/pox-doc/html/

[94] Ryu, "Component-based SDN framework," 2018. [Online]. Available: https://osrg.github.io/ryu/index.html

[95] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013. [Online]. Available: http://doi.acm.org/10.1145/2408776.2408794

[96] C. G. Requena, F. G. Villamón, M. E. G. Requena, P. J. L. Rodríguez, and J. D. Marín, "Ruft: Simplifying the fat-tree topology," in *Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE International Conference on*. IEEE, 2008, pp. 153–160.

[97] Mininet, "An Instant Virtual Network on your Laptop (or other PC)," 2018. [Online]. Available: http://mininet.org/

[98] E. D. Sontag, *Mathematical control theory: deterministic finite dimensional systems*. Springer Science & Business Media, 2013, vol. 6.

[99] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The Primary-Backup Approach," *Distributed systems*, vol. 2, pp. 199–216, 1993.

[100] F. B. Schneider, "The state machine approach: A tutorial," in *Fault-tolerant distributed computing*. Springer, 1990, pp. 18–41.

[101] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *Computer*, vol. 30, no. 4, pp. 68–74, 1997.

[102] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[103] 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.

[104] 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.

[105] 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.

[106] 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.

[107] 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.

[108] H. Garcia-Molina, "Elections in a distributed computing system," *IEEE transactions on Computers*, no. 1, pp. 48–59, 1982.

[109] S. D. Stoller, "Leader election in asynchronous distributed systems," *IEEE transactions on computers*, no. 3, pp. 283–284, 2000.

[110] Ö. Babaoğlu, R. Davoli, and A. Montresor, "Group membership and view synchrony in partitionable asynchronous distributed systems: Specifications," *ACM SIGOPS Operating Systems Review*, vol. 31, no. 2, pp. 11–22, 1997.

[111] R. Guerraoui and L. Rodrigues, *Introduction to reliable distributed programming.* Springer Science & Business Media, 2006.

[112] O. Laadan and S. E. Hallyn, "Linux-cr: Transparent application checkpoint-restart in linux," in *Linux Symposium*, vol. 159. Citeseer, 2010.

[113] 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.

[114] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation.* San Francisco, 2008, pp. 161–174.

[115] L. Lamport *et al.*, "Paxos Made Simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[116] E. F. Camacho and C. B. Alba, *Model predictive control.* Springer Science & Business Media, 2013.

[117] A. Schiper and A. Sandoz, "Uniform reliable multicast in a virtually synchronous environment," in *Distributed Computing Systems, 1993., Proceedings the 13th International Conference on.* IEEE, 1993, pp. 561–568.

[118] C. J. Fidge, "Timestamps in Message-Passing Systems that Preserve the Partial Ordering," 1987.

[119] 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.

[120] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, “Efficient byzantine fault-tolerance,” *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, 2013.

[121] 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.

[122] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, “Zyzzyva: Speculative Byzantine Fault Tolerance,” in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 45–58.

[123] 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.

[124] 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.

[125] 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.

[126] D. L. Parnas, “Software aging,” in *Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press, 1994, pp. 279–287.

[127] 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.

[128] M. Grottke, L. Li, K. Vaidyanathan, and K. S. Trivedi, “Analysis of software aging in a web server,” *IEEE Transactions on reliability*, vol. 55, no. 3, pp. 411–420, 2006.

[129] 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.

[130] J. Mostafa, “Software Aging in Real-Time Control Systems,” 2017.

[131] H. Kopetz and G. Bauer, “The Time-Triggered Architecture,” *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.

[132] P. Veríssimo and A. Casimiro, "The timely computing base model and architecture," *Computers, IEEE Transactions on*, vol. 51, no. 8, pp. 916–930, 2002.

[133] M. Scharf and S. Kiesel, "Head-of-line blocking in tcp and sctp: Analysis and measurements." in *GLOBECOM*, vol. 6, 2006, pp. 1–5.

[134] M. Scharf and A. Ford, "Multipath tcp (mptcp) application interface considerations," Tech. Rep., 2013.

[135] 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.

[136] J. Iyengar and M. Thomson, "Quic: A udp-based multiplexed and secure transport," *draft-ietf-quic-transport-01 (work in progress)*, 2017.

[137] M. Allman, V. Paxson, and E. Blanton, "Tcp congestion control," Tech. Rep., 2009.

[138] M. García, R. Agüero, and L. Muñoz, "On the unsuitability of tcp rto estimation over bursty error channels," in *IFIP International Conference on Personal Wireless Communications.* Springer, 2004, pp. 343–348.

[139] D. S. Lun, M. Médard, R. Koetter, and M. Effros, "On coding for reliable communication over packet networks," *Physical Communication*, vol. 1, no. 1, pp. 3–20, 2008.

[140] R. Coltun, D. Ferguson, J. Moy, and A. Lindem, "OSPF for IPv6," RFC 5340 (Proposed Standard), Internet Engineering Task Force, Jul. 2008, updated by RFCs 6845, 6860, 7503. [Online]. Available: http://www.ietf.org/rfc/rfc5340.txt

[141] C. Hedrick, "Routing Information Protocol," RFC 1058 (Historic), Internet Engineering Task Force, Jun. 1988, updated by RFCs 1388, 1723. [Online]. Available: http://www.ietf.org/rfc/rfc1058.txt

[142] R. Pallos, J. Farkas, I. Moldovan, and C. Lukovszki, "Performance of rapid spanning tree protocol in access and metro networks," in *Access Networks Workshops, 2007. AccessNets '07. Second International Conference on*, Aug 2007, pp. 1–8.

[143] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. P. Vasseur, and R. Alexander, "Rpl: Ipv6 routing protocol for low-power and lossy networks," Tech. Rep., 2012.

[144] N. Sprecher and A. Farrel, "MPLS Transport Profile (MPLS-TP) Survivability Framework," IETF, RFC 6372 (Informational), Internet Engineering Task Force, Sep. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6372.txt

[145] 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.

[146] 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.

[147] H. Yuasa, T. Satake, M. J. Cardona, H. Fujii, A. Yasuda, K. Yamashita, S. Suzaki, H. Ikezawa, M. Ohno, A. Matsuzaki *et al.*, "Virtual lan system," 2000, uS Patent 6,085,238.

[148] R. Yuan, W. T. Strayer, and T. Strayer, *Virtual private networks: technologies and solutions.* Addison-Wesley, 2001.

[149] A. Medina, M. Allman, and S. Floyd, "Measuring the evolution of transport protocols in the internet," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 2, pp. 37–52, 2005.

[150] M. Rentschler and H. Heine, "The Parallel Redundancy Protocol for Industrial IP Networks," in *Industrial Technology (ICIT), 2013 IEEE International Conference on*, Feb 2013, pp. 1404–1409.

[151] R. Guerraoui, D. Kozhaya, M. Oriol, and Y. A. Pignolet, "Who's on Board?: Probabilistic Membership for Real-Time Distributed Control Systems ," in *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, Sept 2016, pp. 167–176.

[152] W. Zhang, M. S. Branicky, and S. M. Phillips, "Stability of networked control systems," *IEEE Control Systems*, vol. 21, no. 1, pp. 84–99, 2001.

[153] A. K. Singh, R. Singh, and B. C. Pal, "Stability analysis of networked control in smart grids," *IEEE Transactions on Smart Grid*, vol. 6, no. 1, pp. 381–390, 2015.

[154] M. Sanfridson, M. Törngren, and J. Wikander, "The effect of randomly time-varying sampling and computational delay," *IFAC Proceedings Volumes*, vol. 38, no. 1, pp. 209–218, 2005.

[155] P. Tabuada, "Event-triggered real-time scheduling of stabilizing control tasks," *IEEE Transactions on Automatic Control*, vol. 52, no. 9, pp. 1680–1685, 2007.

[156] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao, "Feedback control real-time scheduling: Framework, modeling, and algorithms," *Real-Time Systems*, vol. 23, no. 1-2, pp. 85–126, 2002.

[157] T. C. Yang, “Networked control system: a brief survey,” *IEE Proceedings-Control Theory and Applications*, vol. 153, no. 4, pp. 403–412, 2006.

[158] S. Gatziu and K. R. Dittrich, “Events in an active object-oriented database system,” in *Rules in Database Systems.* Springer, 1994, pp. 23–39.

[159] 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, 2004.

[160] W. Saab, R. Rudnik, L. Reyes-Chamorro, J.-Y. Le Boudec, 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)*, no. CONF, 2018.

[161] L. Lamport, “Fast Paxos,” *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[162] W. Saab, M. Mohiuddin, S. Bliudze, and J.-Y. Le Boudec, “Quarts: Quick Agreement for Real-Time Control Systems,” in *22nd IEEE International Conference on Emerging Technologies And Factory Automation.* IEEE, 2017.

[163] A. Casimiro and P. Veríssimo, “Timing Failure Detection with a Timely Computing Base,” 1999.

[164] “Ieee standard for a precision clock synchronization protocol for networked measurement and control systems,” *IEEE Std 1588-2002*, pp. i–144, 2002.

[165] D. L. Mills, “Internet time synchronization: the network time protocol,” *Communications, IEEE Transactions on*, vol. 39, no. 10, pp. 1482–1493, 1991.

[166] F. J. Torres-Rojas and M. Ahamad, “Plausible clocks: constant size logical clocks for distributed systems,” *Distributed Computing*, vol. 12, no. 4, pp. 179–195, 1999.

[167] T. Landes, “Dynamic vector clocks for consistent ordering of events in dynamic distributed applications.” in *PDPTA*, 2006, pp. 31–37.

[168] P. S. Almeida, C. Baquero, and V. Fonte, “Interval tree clocks,” in *International Conference On Principles Of Distributed Systems.* Springer, 2008, pp. 259–274.

[169] 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.

[170] S. Kamboj, W. Kempton, and K. S. Decker, “Deploying Power Grid-Integrated Electric Vehicles as a Multi-Agent System,” in *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 1.* International Foundation for Autonomous Agents and Multiagent Systems, 2011, pp. 13–20.

[171] S. Gilbert and N. A. Lynch, "Perspectives on the CAP Theorem." Institute of Electrical and Electronics Engineers, 2012.

[172] A. Schiper, "Early consensus in an asynchronous system with a weak failure detector," *Distributed Computing*, vol. 10, no. 3, pp. 149–157, 1997.

[173] 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.

[174] 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.

[175] R. Rudnik, L. E. 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 *PowerTech 2017*, no. EPFL-CONF-226196, 2017.

[176] S. Sarri, L. Zanni, M. Popovic, J.-Y. Le Boudec, and M. Paolone, "Performance Assessment of Linear State Estimators using Synchrophasor Measurements," *IEEE Transactions on Instrumentation and Measurement*, vol. 65, no. 3, pp. 535–548, 2016.

[177] 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.

[178] J.-Y. Le Boudec, *Performance Evaluation of Computer and Communication Systems.* Epfl Press, 2010.

[179] D. W. Allan, M. A. Weiss *et al.*, *Accurate time and frequency transfer during common-view of a GPS satellite.*

[180] 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.

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

[182] H. Kopetz, "Fault containment and error detection in the time-triggered architecture," in *Autonomous Decentralized Systems, 2003. ISADS 2003. The Sixth International Symposium on.* IEEE, 2003, pp. 139–146.

[183] ——, "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.

[184] S. Poledna, “Replica Determinism in Distributed Real-Time Systems: A Brief Survey,” *Real-Time Systems*, vol. 6, no. 3, pp. 289–316, 1994.

[185] 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.

[186] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, “Microreboot–a technique for cheap recovery,” *arXiv preprint cs/0406005*, 2004.

[187] 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.* ACM, 2018, pp. 303–313.

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

[189] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, “Onix: A Distributed Control Platform for Large-Scale Production Networks.” in *OSDI*, vol. 10, 2010, pp. 1–6.

[190] N. Katta, H. Zhang, M. Freedman, and J. Rexford, “Ravana: Controller Fault-Tolerance in Software-Defined Networking,” in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research.* ACM, 2015, p. 4.

[191] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O’Connor, P. Radoslavov, W. Snow *et al.*, “ONOS: Towards an Open, Distributed SDN OS,” in *Proceedings of the third workshop on Hot topics in software defined networking.* ACM, 2014, pp. 1–6.

[192] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker, “Cap for networks,” in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN ’13. New York, NY, USA: ACM, 2013, pp. 91–96. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491186

[193] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, “Consistent updates for software-defined networks: Change you can believe in!” in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, ser. HotNets-X, 2011, pp. 7:1–7:6.

[194] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “Openflow: enabling innovation in campus networks,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[195] J. N. Gray, "Notes on data base operating systems," in *Operating Systems*. Springer, 1978, pp. 393–481.

[196] N. Spring, R. Mahajan, and D. Wetherall, "Measuring isp topologies with rocketfuel," *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 4, pp. 133–145, 2002.

[197] OpenFlow, "Openflow," 2018. [Online]. Available: https://en.wikipedia.org/wiki/OpenFlow

[198] OVS, "Open vSwitch," 2018. [Online]. Available: http://www.openvswitch.org/

[199] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically Centralized?: State Distribution Trade-offs in Software Defined Networks," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, 2012, pp. 1–6.

[200] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 136–141, February 2013.

[201] A. Tootoonchian and Y. Ganjali, "Hyperflow: A Distributed Control Plane for Openflow," in *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, 2010, pp. 3–3.

[202] H. Howard, M. Schwarzkopf, A. Madhavapeddy, and J. Crowcroft, "Raft refloated: Do we have consensus?" *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, pp. 12–21, Jan. 2015.

[203] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12, 2012, pp. 323–334.

[204] T. Mizrahi, E. Saat, and Y. Moses, "Timed consistent network updates," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15, 2015, pp. 21:1–21:14.

[205] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, ser. HotNets-XII, 2013, pp. 20:1–20:7.

[206] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13, 2013, pp. 49–54.

[207] M. Pignati and al., "Real-Time State Estimation of the EPFL-Campus Medium-Voltage Grid by Using PMUs," in *Innovative Smart Grid Technologies Conference (ISGT), 2015 IEEE PES*, 2015.

[208] J. Gao, Y. Xiao, J. Liu, W. Liang, and C. P. Chen, "A Survey of Communication/Networking in Smart Grids," *Future Gener. Comput. Syst.*, vol. 28, no. 2, pp. 391–404, Feb. 2012.

[209] M. Kuzlu, M. Pipattanasomporn, and S. Rahman, "Communication Network Requirements for Major Smart Grid Applications in HAN, NAN and WAN," *Computer Networks*, vol. 67, pp. 74–88, 2014.

[210] T. Kropp, "Assessment of Wireless Technologies in Substation Functions- Part II: Substation Monitoring and Management Technologies," Technical Report, Electrical Power Research Institute, Tech. Rep., 2006.

[211] F. Cleveland, "Use of Wirelless Data Communiicatiions in Power System Operations," in *Power Systems Conference and Exposition, 2006. PSCE'06. 2006 IEEE PES*. IEEE, 2006, pp. 631–640.

[212] G. W. Irwin, J. Colandairaj, and W. G. Scanlon, "An overview of wireless networks in control and monitoring," in *International Conference on Intelligent Computing*. Springer, 2006, pp. 1061–1072.

[213] M. Rentschler, O. A. Mady, M. T. Kassis, H. H. Halawa, T. K. Refaat, R. M. Daoud, H. H. Amer, and H. M. ElSayed, "Simulation of Parallel Redundant WLAN with OPNET," in *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on*. IEEE, 2013, pp. 1–8.

[214] M. T. Kassis, O. A. Mady, H. H. Halawa, M. Rentschler, R. M. Daoud, H. H. Amer, and H. M. ElSayed, "Analysis of Parallel Redundant WLAN with Timing Diversity," in *Computer and Information Technology (WCCIT), 2013 World Congress on*. IEEE, 2013, pp. 1–6.

[215] M. Hendawy, M. ElMansoury, K. N. Tawfik, M. M. ElShenawy, A. H. Nagui, A. T. Elsayed, H. H. Halawa, R. M. Daoud, H. H. Amer, M. Rentschler *et al.*, "Application of Parallel Redundancy in a Wi-Fi-based WNCS using OPNET," in *Electrical and Computer Engineering (CCECE), 2014 IEEE 27th Canadian Conference on*. IEEE, 2014, pp. 1–6.

[216] M. Rentschler and P. Laukemann, "Towards a Reliable Parallel Redundant WLAN Black Channel," in *Factory Communication Systems (WFCS), 2012 9th IEEE International Workshop on*. IEEE, 2012, pp. 255–264.

[217] ——, "Performance Analysis of Parallel Redundant Wlan," in *Emerging Technologies & Factory Automation (ETFA), 2012 IEEE 17th Conference on*. IEEE, 2012, pp. 1–8.

[218] A. Frommgen, T. Erbshäußer, A. Buchmann, T. Zimmermann, and K. Wehrle, "Remp tcp: Low latency multipath tcp," in *Communications (ICC), 2016 IEEE International Conference on.* IEEE, 2016, pp. 1–7.

[219] D. Dzung, R. Guerraoui, D. Kozhaya, and Y.-A. Pignolet, "To transmit now or not to transmit now," in *Reliable Distributed Systems (SRDS), 2015 IEEE 34th Symposium on.* IEEE, 2015, pp. 246–255.

[220] J. Korhonen and Y. Wang, "Effect of Packet Size on Loss Rate and Delay in Wireless Links," in *Wireless Communications and Networking Conference, 2005 IEEE*, vol. 3. IEEE, 2005, pp. 1608–1613.

[221] A. Willig, M. Kubisch, C. Hoene, and A. Wolisz, "Measurements of a Wireless Link in an Industrial Environment using an IEEE 802.11-Compliant Physical Layer," *Industrial Electronics, IEEE Transactions on*, vol. 49, no. 6, pp. 1265–1282, 2002.

[222] G. Cena, S. Scanzio, and A. Valenzano, "Experimental characterization of redundant channels in industrial wi-fi networks," in *2016 IEEE World Conference on Factory Communication Systems (WFCS).* IEEE, 2016, pp. 1–4.

[223] W. Fenner, "Internet group management protocol, version 2," Tech. Rep., 1997.

[224] S. Deering, W. Fenner, and B. Haberman, "Multicast listener discovery (mld) for ipv6," Tech. Rep., 1999.

[225] J. Nonnenmacher and E. Biersack, "Optimal Multicast Feedback," in *INFOCOM '98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3, Mar 1998, pp. 964–971 vol.3.

[226] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," no. 6347, 2012.

[227] E. N. Gilbert, "Capacity of a Burst-Noise Channel," *Bell system technical journal*, vol. 39, no. 5, pp. 1253–1265, 1960.

[228] C. Z. Mooney, R. D. Duval, and R. Duval, *Bootstrapping: A Nonparametric Approach to Statistical Inference.* Sage, 1993, no. 94-95.

[229] O. Stack, "Welcome to neutron's documentation!" https://docs.openstack.org/neutron/pike/, accessed: 2018-06-06.

[230] M. Handley, S. Floyd, J. Padhye, and J. Widmer, "Tcp friendly rate control (tfrc): Protocol specification," Tech. Rep., 2002.

List of Publications

Following is the list of all my publications written as a PhD student at EPFL.

Accepted

1. 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.
2. M. Popovic, M. Mohiuddin, D.-C. Tomozei, and J.-Y. Le Boudec, "iPRP: Parallel Redundancy Protocol for IP Networks," in *Factory Communication Systems (WFCS), 2015 IEEE World Conference on.* IEEE, 2015, pp. 1–4.
3. M. Popovic, M. Mohiuddin, D.-C. Tomozei, and J.-Y. Le Boudec„ "iPRP—the Parallel Redundancy Protocol for IP Networks: Protocol Design and Operation," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 5, pp. 1842–1854, 2016.
4. 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.
5. M. Mohiuddin, M. Popovic, A. Giannakopoulos, and J.-Y. Le Boudec, "Experimental Validation of the Usability of Wi-Fi over Redundant Paths for Streaming Phasor Data," in *Smart Grid Communications (SmartGridComm), 2016 IEEE International Conference on.* IEEE, 2016, pp. 533–538.
6. 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.
7. 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.

8. 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*, 2017.
9. 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.
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.* ACM, 2018, pp. 303–313.
11. E. Mohammadpour, E. Stai, M. Mohiuddin, and J.-Y. Le Boudec, "End-to-End Latency and Backlog Bounds in Time-Sensitive Networking with Credit Based Shapers and Asynchronous Traffic Shaping," in *International Workshop on Network Calculus and Applications (NetCal 2018).* IEEE, 2018.

Under Review

1. M. Mohiuddin, W. Saab, and J.-Y. Le Boudec, "Consistent Replication of Controllers in a Cyber-Physical System with Asynchronous Sensors," *IEEE Transactions on Industrial Informatics*, 2018.
2. M. Mohiuddin, M. Primorac, E. Stai, and J.-Y. Le Boudec, "The Quick Coordination Layer for Consistent Controller-Replication in SDN," 2018.
3. 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," 2018.

Maaz Mohiuddin

Curriculum Vitae

Route Cantonale 39B
1025, St. Sulpice
☎ *+41 78 805 50 78*
✉ *maaz.mohiuddin@epfl.ch*

Research Interests

Cyber-Physical Systems, Distributed Systems, Fault-Tolerance, Communication Networks

Education

2014–2018 **PhD in Computer Science**, *Swiss Federal Institute of Technology (EPFL)*.

Advisor	Prof. Jean-Yves Le Boudec
Thesis	Reliability Mechanisms for Controllers in Real-Time Cyber-Physical Systems

2009–2013 **Bachelor in Technology**, *Indian Institute of Technology Hyderabad (IITH)*.

Major	Electrical Engineering with Honors
Minor	Computer Science and Engineering
Grade	9.01/10
Advisor	Dr. Kiran Kuchi
Thesis	Performance Limits of a Cloud Radio

Research Experience

Current
2014–Present **Research Assistant**, *Laboratory for Computer Communications and Applications (LCA2)*, EPFL, Switzerland.

- Research on real-time cyber-physical systems (e.g., smart grids, autonomous cars, SDN)
- Designed reliability mechanisms (Axo, Quarts) for software-based real-time control
  - Fault-tolerance for delays and crashes of controllers under stringent real-time constraints
  - Improved availability and latency by several orders of magnitude
- Involved in deployment of campus-wide communication test-beds
  - Conducted a two-month long measurement campaign to study the fail-independence of directional Wi-Fi links

2013–2014 **Research Intern**, *Laboratory for Computer Communications and Applications (LCA2)*, EPFL, Switzerland.

- Successfully completed an evaluation to be hired as a PhD student at LCA2
- Designed and implemented UDP-based applications over fail-independent Wi-Fi paths
  - Worked with ALIX system boards and OpenWRT operating system
- Implemented iPRP, an IP-layer protocol for transparent-packet replication
  - Worked with Linux kernel modules and iptables
  - Identified and repaired performance bottle-necks to achieve low dataplane latency

2012 **Research Intern**, *University of Bamberg*, Germany.

- Implemented a discrete-event simulator for peer-to-peer live video streaming over Wi-Fi and Cellular networks
- Analyzed quality of video-streaming with stationary and mobile nodes

Teaching Experience

2014-2017 **Teaching Assistant**, *TCP/IP Networking*, EPFL.

Responsible for labs with hands-on exercises on socket programming, TCP congestion control, IPv4/IPv6 interworking, tunneling, routing, and network security. Designed research exercises for advanced concepts in TCP/IP Networking.

2016-2018 **Teaching Assistant**, *Smart Grid Technologies*, EPFL.

Designed the labs on traffic engineering and security attacks in smart-grids. In 2017 and 2018, I taught the course on introduction to TCP/IP networking to a class of 20 students.

2017 **Teaching Assistant**, *Performance Evaluation of Computer Systems*, EPFL.

Responsible for the lab problems that involved performance patterns (bottlenecks, congestion collapse), model fitting and forecasting, discrete-event simulation and queuing theory.

2013 **Teaching Assistant**, *Wireless Sensor Networks*, IITH.

Conducted hand-ons course on wireless networking with CrossBow TelosB sensor motes.

Other Professional Activities

2017-2018 **CEO and Co-Founder**, *RaaSS*.

Reliability software for real-time cyber-physical systems such as smart grids, autonomous cars and datacenter controllers. Conducted interviews for customer discovery and product market-fit. Raised capital through the EPFL Enable grant.

2017-2018 **Deployment of a Campus-Wide Communication Network**, *EPFL*.

Involved in deploying two parallel redundant communication networks for streaming real-time sensor data. Enabled IP multicast routing to support source-specific multicast. Deployed iPRP on the end-hosts for transparent and 0-ms repair of packet losses.

2015-2017 **Project Demos**, *EPFL*.

Demoed the fault-tolerance mechanisms developed as a PhD student at EPFL. Demonstrated real-time packet repair using iPRP at the annual conference of Swiss Competence Center for Energy Research in 2015 and 2016. Demonstrated stability of an inverted pendulum with a delay-faulty controller using Axo at the IC Research Day in 2017.

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 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

Security Infrastructure for the EPFL-Campus Microgrid
Semester Project, Weiyu Zhang, 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 Sanae, Visiting Master Student

Performance Evaluation of the Redundant MPTCP Scheduler
Semester Project, Ines Bahej, EPFL Master Student

Importance Sampling for Fault-Tolerance Systems
Bachelor Thesis, Basile Thullen, EPFL Bachelor Student

2016 **Proactive Fault-Recovery For Real-Time Mission-Critical Systems**
Master Thesis, Aswin Suresh, EPFL Master Student

2015 **Demonstration of Software Aging in the COMMELEC controller**
Semester Project, Shruti Patil, EPFL Master Student

Implementation of an IPv4 version of iPRP for Linux
Research Assistant, Loïc Ottet, EPFL Master Student

2014 **Feasibility of Using Wi-Fi for Real-Time Mission-Critical Communication**
Summer Internship, Thanos Gainnaokopolous, Visiting Master Student

Peer Reviews

Reviewer	Elsevier Journal on Sustainable Energy, Grids and Networks, since 2017
Reviewer	IEEE Transactions on Industrial Informatics, since 2016
External Reviewer	USENIX ATC, since 2015

Skills

Programming	Python, C, C++, Matlab
Tools	Tcpdump, Wireshark, iptables, LaTex, Svn, Git, Mathematica
Environments	Mininet, Quagga, GNS3, Click Modular Router
OS	Linux, MAC OS, Windows, RT-Linux, TinyOS
Languages	English (fluent), French (basic), Hindi (native)

Honors and Awards

2018 **Best Pitch Award (3^{rd} Place)**, *EPFL IC Research Day.*

2017 **EPFL Teaching Award**, *In recognition of excellence in teaching for the graduate course TCP/IP Networking.*

Best Statup Pitch Award, *Innosuisse Business Concept Course, Lausanne.*

2016 **Best Teaching Award**, *In recognition of excellence in teaching for the graduate course Smart Grid Technologies.*

Best Project Award, *Applied Data Analysis Course, EPFL.*

Best Presentation Award, *IEEE IECON 2016*.

2015 **Best Paper Award**, *IEEE World Conference on Factory Communications*.

2012 **Research Fellowship**, *Awarded by the Deutscher Akademischer Austausch Dienst (DAAD) for a summer internship in Germany*.

2011 **TODAI Fellowship**, *Awarded by the University of Tokyo for excellence in undergraduate academics at IITH*.

2010 **IITH Academic Excellence Award**, *For the highest GPA in the academic year*.

TODAI Fellowship, *Awarded by the University of Tokyo for excellence in undergraduate academics at IITH*.

References

References available upon request