FAULT TOLERANCE DESIGN APPROACHES FOR DISTRIBUTED CYBER PHYSICAL SYSTEMS WITH APPLICATIONS IN ENERGY SYSTEMS

By

Purboday Ghosh

Dissertation
Submitted to the Faculty of the
Graduate School of Vanderbilt University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY
in

Electrical Engineering

August 11, 2023

Nashville, Tennessee

Approved:

Dr. Gabor Karsai, Ph.D.

Dr. Bharat Bhuva, Ph.D.

Dr. Abhishek Dubey, Ph.D.

Dr. Aniruddha Gokhale, Ph.D.

Dr. D. Mitchell Wilkes, Ph.D.

To Ma and Baba, who bought me my first book when I was four and would read *The Three Little Pigs* and *Pinocchio* to me every morning before school.

ACKNOWLEDGMENTS

Writing this section is actually quite a cathartic experience, since it made me look back at all those junctures in my life that have led me to this point. It reminds me of *The Butterfly Effect* and how causality works in the universe where every decision I had taken, either willingly or by happenstance or through the generosity of others have shaped my life in unique ways to produce the exact conditions for me to be here, right at this moment. So, this is me, retracing my steps and highlighting some of those moments and influences (since it is inevitable that I miss some of them, apologies!).

Firstly, I would like to thank my advisor Dr. Gabor Karsai, for selecting me as his PhD student and trusting in my potential all the way back in 2017. He has been a truly supportive mentor throughout my PhD journey who has always treated me with respect, patience and allowed me to make mistakes and learn from them.

I would also like to thank all committee members for their valuable feedback and hope to receive their guidance in improving this dissertation. I would like to thank all my collaborators that I have had the chance to work with namely Dr. Dubey, Tim, Scott, Mary, Gerry, Istvan and Peter at ISIS; Dr. Lukic and Dr. Hao at NCSU; Shashank, Christoph and Ulrich at Siemens and Yashen at NREL. Thank you to Harsh Vardhan as well for being a great colleague and a friend and for helping me out in many tough situations. I got to learn a lot from each of them and hope to carry those lessons forward in my future career path.

I also want to thank the many good friends that I made here in Nashville during my time at Vanderbilt that have helped me acclimatize so easily to a new city in a new country. Our weekly potlucks, conversations, hiking and vacation trips and just the emotional support and comforting presence for each other have helped me tide over many a stressful moment and low point of my graduate student life and find renewed energy. So, thank you to Arnab, Neelanjana, Anabil, Avisek, Deepayan, Gourab, Souhrid, Rudra, Soumita and Tonuka. A big thank you also to Dr. Shyamali Mukherjee for being so welcoming and caring towards all of us incoming students and making us feel like we are in a home away from home.

I want to also mention my professors from Jalpaiguri Government Engineering College Dr. Swapan Kr. Saha for his constant motivation to pursue higher studies; project mentors Mr. Anup Kr. Mondal and Dr. Madhumala Ghosh for providing the opportunity to contribute to an exciting project that motivated my interest in research and their help in the PhD admission process. Thank you also to the MW1 gang Madhubanti and Mahasweta and our conversations, Soham, and Dash who inspired me to dream and believe in myself.

On a personal note, I would never be here without the efforts, sacrifices made by my parents and the values that they inculcated in me. They were the ones who instilled the value of education and its empowering quality which has translated into my interest in research and curiosity about learning new things. My sister who was my first teacher ever and an ever present source of encouragement and who paved new pathways all the time for me to follow. Saying thank you would not be enough and I would just like to use this opportunity to acknowledge their contribution in my life.

All of the works related to RIAPS was funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0000666 and a grant from Siemens, CT. The views and opinions of authors expressed herein do not necessarily state or reflect those of the US Government or any agency thereof or of Siemens, CT.

The work in Chapter 5 was supported in part by Department of Energy's Office of Energy Efficiency and Renewable Energy (EERE) under the Solar Energy Technologies Office (SETO) Award Number DE-EE0008769. This work was authored in part by Siemens Technology and the National Renewable Energy Laboratory, operated by Alliance for Sustainable Energy, LLC, for the U.S. Department of Energy (DOE) under Contract No. DE-AC36-08GO28308. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of funding agency.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

List of Abbreviations

CPS	Cyber Physical Systems
DCPS	Distributed Cyber Physical Systems
NAS	National Academy of Science
PMU	Phasor Measurement Unit
V	volts
kV	kilovolts
DER	Distributed Energy Resources
MGCC	Microgrid Central Controller
RIAPS	Resilient Information Architecture Platform for Smart Grids
QoS	Quality of Service
RL	Reinforcement Learning
PV	photovoltaic
DOE	Department of Energy
PNNL	Pacific Northwest National Laboratory
NIST	National Institute of Technology
DSO	Distribution System Operator
RES	Renewable Energy Sources
F.L.I.R	Fault Location Isolation Restoration
CAP	Consistency, Availability, Partition Tolerance
CBSE	Component Based Software Engineering
AST	Abstract Syntax Tree
TA	Timed Automata
AADL	Architecture Analysis and Design Language
SUT	System Under Test
CTL	Computation Tree Logic
RIAPS	Resilient Information Architecture Platform for Smart Grids
CFG	Control Flow Graph
DSL	Domain Specific Language
CFAST	Control Flow Abstract Syntax Tree
TAG	Timed Automata Generation
CFG	Control Flow Graph
BPD	Bounded Path Dissemination

IoT	Internet of Things
SDN	Software Defined Networking
QoS	Quality of Service
CTL	Computation Tree Logic
PTP	Precision Time Protocol
NTP	Network Time Protocol
GPS	Global Positioning System
RL	Reinforcement Learning
CLI	Command Line Interface
FDIA	False Data Injection Attack
EMS	Energy Management System
OPF	Optimal Power Flow
BPD	Bounded Path Dissemination
DE	Dissemination Efficiency
CBSE	Component Based Software Engineering
tc	traffic controller
DoS	Denial of Service
FNCS	Framework for Network Cosimulation
TESP	Transactive Energy Simulation Platform
LV	Low Voltage
MV	Medium Voltage
DG	Distributed Generator
ANN	Artificial Neural Network
TSN	Time Sensitive Networking

CHAPTER 1

Introduction

Distributed computing architectures are increasingly being adopted in the design of large-scale Cyber Physical Systems (CPS) due to the inherent advantages of being more robust to single point failures [Gupta et al. (2006)], efficient sharing of computational load [Mirchandaney et al. (1990)] and resources [Ali et al. (2004)], and being highly scalable [Skala et al. (2015)] and interoperable [Ouksel and Sheth (1999)]. Their application domains range from connected autonomous vehicles and smart transit systems, edge-cloud-based clinical applications such as image-guided surgery, as well as intelligent coordination and control of energy resources and consumers in Smart Grids.

Safety-critical CPS must adhere to stringent safety, reliability, and security requirements. This has put an increased focus on the incorporation of fault identification and mitigation techniques into the design of any software application for such systems. The realization of fully or partially decentralized algorithms provides more robustness by eliminating single points of failure and allowing for greater information exchange between components. Fault tolerance in such systems involves a close interplay between the physical system: the *plant* to be controlled and the distributed computing layer that controls it.

Such large scale and safety critical systems are required to maintain acceptable performance levels even in case of adversities such as unintended failures or malicious attacks. Some examples of faults include faults in the communication network, faults in the computing node hardware, malfunctions, and disturbances to the physical plant elements such as short circuits or malicious attacks such as security breaches on autonomous vehicles.

The term *acceptable* used here is qualitative and is decided by the design architects based on certain assumptions and requirements. These include assumptions regarding how the system should behave under different input stimuli, what is the maximum range of variation that can be tolerated over the output signals for the plant to continue its operation, what is the maximum amount of computing resources or power consumption that the integrated CPS is allowed to use, what is the longest response delay that downstream systems can tolerate and so on. These are mostly determined by domain experts, architects and various other stakeholders prior to the design phase. All these considerations and the mechanisms used to realize them collectively form what is known as the *fault management* subsystem. There are a wide number of fault tolerance approaches and design patterns that have been extensively studied and analyzed in both academia and industry [Hanmer (2013)] that are applicable to Distributed Cyber Physical Systems (DCPS). The key aspect of a CPS is that the cyber components must always work in compliance with the underlying physics of the plant for the system to be deemed correct. It is imperative that the plant in question continues to operate at an acceptable level of safety that does not cause risk to people and property, even if the cyber parts fail. In DCPS, the added complexity added by the communication and coordination of multiple processes, as well as network bandwidth and resource constraints, make the design of fault tolerant systems especially challenging. Traditional approaches for designing fault tolerant software are not adequate for DCPS since they fail to consider the *cyber physical* aspect of the system as to how the distributed algorithms executed by the cyber layer can react to such faults to keep the physical system safe. It must be noted here that complete fault tolerance is a very strong requirement that may not be achieved by software techniques alone.

1.1 Example of a Cyber Physical System: The modern electric grid

CPS refer to a class of engineered systems that are built upon the integration of computing, networking, and physical processes. They are primarily feedback systems that consist of networked sensory components that monitor one or more physical processes. That information is then passed to a computational subsystem that performs control and supervisory responsibilities affecting the said physical processes such that the system is able to meet the required specifications related to its performance, safety, timeliness, etc. [Ptolemy (2012)]. CPS technologies and methods have found their usage in a range of application domains including agriculture, aeronautics, building design, civil infrastructure, energy, environmental quality, healthcare and personalized medicine, manufacturing, and transportation.

One of the largest adoption of CPS principles and technologies has been in the energy systems field. The infusion of intelligence into the traditional power grid in a way that improves its efficiency and reliability as well as reduce costs has led to the concept of the modern *Smart Grid* [Fang et al. (2011)]. The demands of the modern electric grid involve greater efficiency, which can be enabled by greater integration of Distributed Energy Resources (DER) using renewable energy such as solar, wind, geothermal, etc. The traditional utility production model is also being evolved into a more distributed producer-consumer approach facilitated by local energy sources and two-way communication capabilities between the utility and the customers. This added flexibility is provided by the creation of what is known as *microgrids*. A microgrid is a group of interconnected energy consumption units, called load, and DERs within clearly defined electrical boundaries that acts as a single controllable entity with respect to the grid. A microgrid can connect and disconnect from the grid to allow it to operate in both *grid-connected* or *islanded* mode [Bower et al. (2014)]. Figure 1.1 shows a typical microgrid arrangement.

A microgrid serves as a good example of a CPS as it has a well defined physical layer, namely the electrical network comprising the various devices such as generators, loads, transformers, relays connected by physical links. This physical system is augmented with a cyber or logical layer, comprising computation, net-

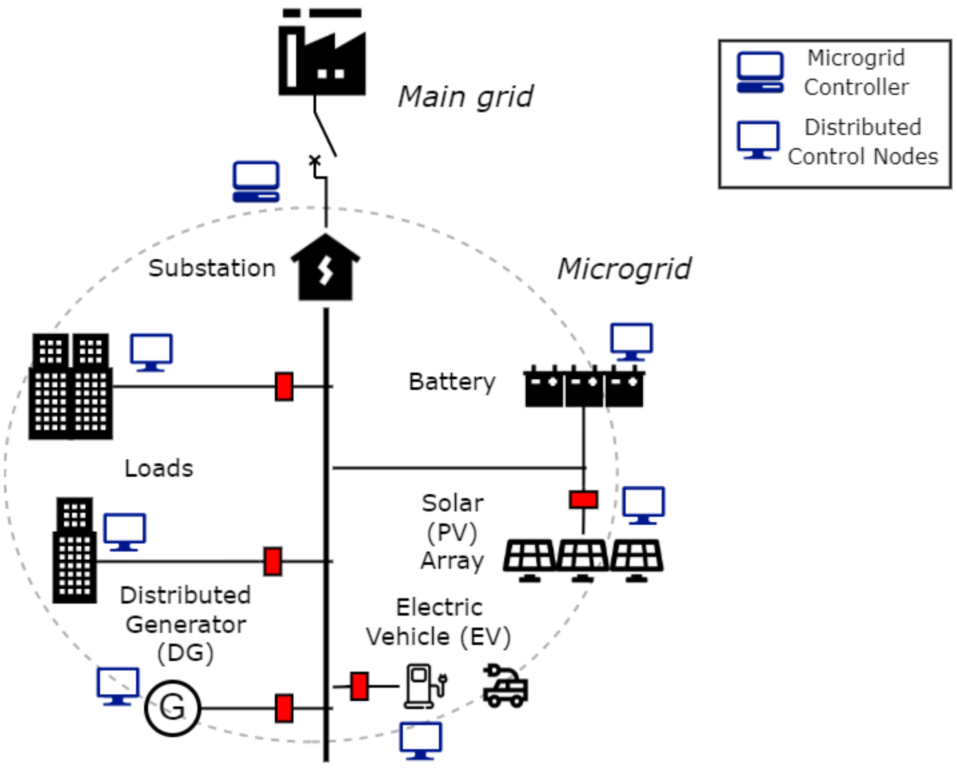


Figure 1.1: Typical microgrid showing both the cyber (in blue) and physical elements

working, information processing, sensing, and actuation capabilities that help monitor and control the physical system. More formally, the National Institute of Technology (NIST)'s Smart Grid and Cyber-Physical Systems Program Office defined six characteristics for CPS in a special publication in 2019 [DeFranco and Serpanos (2021), Greer et al. (2019)]. Using them as guidelines, allows us to analyze microgrids and their consideration as a CPS.

1. *Hybrid architecture*: A CPS instance should consist of components that belong to the physical, logical, transducing and human categories. As discussed previously, in the case of a microgrid, these include the electrical network and devices such as generators, loads, batteries and solar panels (physical), the computational and communication infrastructure (logical), the sensors and actuators (transducers) and the various roles performed by engineers and technicians such as designing, operating and guiding the microgrid system (human).

2. *Hybrid methods*: All elements of a CPS should be integrated that allow the transmission, transformation, and storage of information between the physical and logical elements and input/ output capabilities for the sensing and actuating components. For microgrids, the underlying software provides capabilities to process and algorithms to transform data that can integrate with the physical elements. Data

storage can be done locally at the various computing nodes or on a centralized database.

3. *Control*: A CPS must utilize the link between its computational and physical components to achieve some control objective. In the case of a microgrid, various such control schemes are employed that are each responsible for some specific functions. The algorithms are usually employed in a three-level control scheme. At the lowest level is the primary control responsible for device level voltage/ current regulation. Secondary control, which acts at a higher level, optimizes the operation of the whole microgrid. Its functions include voltage/ frequency restoration for islanding or reconnecting to the main grid, enhanced power sharing among the microgrid elements etc. These functions can be realized using either a centralized architecture where a centralized microgrid controller communicates with the local control nodes, a decentralized method where decisions are taken by the control nodes using local information, or a distributed structure where the control nodes exchange information among themselves to arrive at some sort of consensus. Tertiary control algorithms act at the highest level that perform functions such as power and energy management, economic dispatch [Al Farsi et al. (2015)] etc.

4. *Time*: A CPS should integrate real-world physical time with event-driven computation effectively. Various control algorithms in a microgrid are event driven which are triggered by a physical event occurring in real time, such as a command from the Distribution System Operator (DSO) or a change in the frequency of the alternating current. The communication and computation time in turn coincides with the passage of physical time and with how the physical system might evolve during that time. This interdependence needs to be considered in the implementation of the various control algorithms for a microgrid.

5. *Trustworthiness*: Most CPS applications involve critical physical processes and systems and thus invoke requirements for safety, reliability and security. This is also true for microgrids since they often serve critical sectors such as hospitals, university campuses and defense encampments. Threats to a microgrid include faults as well as malicious cyber-attacks. The former will be the focus of this dissertation.

As is evident, a CPS is often a vast, complex system with various moving parts. Thus, it also comes with its associated challenges. Looking at microgrids as an example, they can be categorized as

- *Maintenance of the technical performance objectives for the system using the integration between the cyber and physical layers.* For a microgrid, the general performance objectives would be to ensure that all loads connected to the grid receive enough power and that there is sufficient power generated, allowing the microgrid to be islanded from the main grid and reconnect with it while maintaining its stability in terms of frequency and voltages, and so on.

- *Ensure the safety of the system while facing unpredictable events such as faults.* A microgrid is a critical CPS that can be affected by phenomena such as natural disasters, component failures, communication outages and so on. The various supervisory and control mechanisms should be designed to protect or compensate for such anomalies wherever possible to ensure that the overall system continues to operate at some accepted performance level. If that is not feasible, then the system should have protocols in place for it to either isolate the failure or shut down the whole system or some parts of it so that the damage does not propagate, and also recover from it quickly. The main objective is to prevent the system from going into an unsafe state that can cause damage to life or property, as was the case in Texas in 2021 [King et al. (2021)].

- *Ancillary objectives* such as economic operation of the microgrid both from the perspective of consumers and the utility or the DSO, minimizing environmental impact, better quality of service, and so on.

For all of the above challenges, there is an electrical engineering aspect that is based on the physics of the underlying electrical network, taking advantage of years of literature and principles behind it. For example, for the electrical network requirements, determining the voltage and current set points for the generators by using power flow equations; or the differential equations that govern how the real and reactive power needs to be updated for each DER to synchronize the voltage and frequency with the main grid prior to islanding or reconnection. For the second challenge, these would be what sort of protection devices should be installed and at what places within the network; the optimal switching sequence to reconfigure the grid following the loss of one or more DERs in one part of the microgrid etc. For the ancillary objectives, it would be the type and amount of storage technologies that should be used or algorithms to determine the inverter set points for the Renewable Energy Sources (RES) connected.

The other aspect is to facilitate the implementation of these techniques and algorithms by providing tools and services as part of the associated logical layer of the system, the so called *cyber* aspect of the CPS. These would involve setting up the appropriate control hierarchy and the communication framework, incorporating fault-tolerant design patterns within the software and implementing advanced algorithms like data analytics [Hu and Vasilakos (2016)] , forecasting [Hong et al. (2020)] and game-theoretic markets [Tushar et al. (2018)].

The final aspect that is somewhat beyond the scope of this dissertation is the non-technical aspect. This involves decisions at the policy level by international governments and organizations to incentivize microgrid development, creating a free and fair energy market, encourage power companies to modernize the grid infrastructure that supports smart grid principles and fund more research into energy technologies.

1.2 Definitions

Before proceeding to discuss the various design and implementation methodologies, it is important to understand some of the key concepts and terminologies related to fault tolerance in computing systems [Nelson (1990)].

Definition 1 (Fault) *A defect in a system. It is the cause of malfunctioning of a component of a system. In the distributed systems domain, faults can be categorized into transient, intermittent, or permanent faults.*

Definition 2 (Transient faults) *Transient faults occur once and then disappear. For example, a network message transmission times out but works fine when attempted a second time*

Definition 3 (Intermittent faults) *These are sporadic faults that can occur and then vanish continually. An example of an intermittent fault can be a loose power connector.*

Definition 4 (Permanent faults) *These are persistent faults that continue to exist until the faulty component is repaired or replaced. Examples of this fault are disk head crashes, software bugs, and damaged hardware.*

Definition 5 (Failure) *A failure is an observable anomaly in the system that is contrary to its specifications. Specifications are some performance or reliability requirements that have been agreed upon as part of the expected service of the system. Failures can be either fail-silent (also known as fail-stop) or Byzantine.*

Definition 6 (Fail-silent / fail-stop failure) *A fail-silent failure is one in which the faulty unit stops working and does not produce an erroneous output. It produces no output or produces an output to indicate failure.*

Definition 7 (Byzantine failure) *A Byzantine failure occurs when the faulty unit continues to run but produces incorrect results that cannot be distinguished from the correct one. These types of failure are the most difficult to detect and mitigate from a system designer's perspective.*

Definition 8 (Error) *An error is the part of the system state that is liable to a failure, i.e., to the delivery of a service not complying with the specifications. The cause of an error is always a fault. Thus, the order of causality is Fault $\rightarrow$ Error $\rightarrow$ Failure [Laprie (1985)].*

Definition 9 (Failure Mode) *Failure mode is the effect by which a failure is observed [Naresky (1970)]. It essentially denotes the ways or modes in which something might fail.*

Definition 10 (Dependability) *Dependability represents 'the collective term used to describe the availability performance and its influencing factors: reliability performance, maintainability performance and maintenance support performance' [ISO / TC 176/SC 1 N 93].*

Definition 11 (Fault tolerance) *Fault tolerance is the property of a system to continue to operate and provide an acceptable level of service as per the specifications in the presence of one or more faults within the system. The level of fault-tolerance provided by the system can vary depending on sporadic or persistent faults, the number of simultaneous faults, and other factors.*

Definition 12 (Resilience) *Resilience is defined as the ability to anticipate and adapt to changing conditions as well as to withstand and recover rapidly from disruptions. According to the National Academy of Science (NAS), the various resilience requirements can be categorized into four goals, namely, plan/prepare, absorb, recover, and adapt [Council et al. (2012)]. Although the terms resilience and fault-tolerance are used somewhat interchangeably in the field of computer science, they are not the same, and resilience must not only consider how to keep the system operational under fault conditions, but it must also consider restoring the system functionality to its original state once the fault is removed.*

1.3 A general methodology for fault tolerant CPS design

Application developers need to determine which specific fault tolerance algorithms/patterns would need to be leveraged for their specific application and robustness requirements. It is a fairly complex problem that requires a deep understanding of the physical domain and its underlying principles and how that translates to variables and constraints that interact with the software [Miller and Tribble (2001)]. In generalizing the basic process flow for the incorporation of fault tolerance, we propose a hierarchical pyramidal design principle. We start from a low-level set of assumptions that govern the physical system which form the base of the *fault management pyramid* [Ghosh and Karsai (2020), ©2020 IEEE]. On top of it, we add more detailed steps that gradually become more tightly interrelated with the application itself until the apex of the pyramid is reached. The corresponding visualization is shown in Figure 1.2. The different layers comprising the fault management pyramid are explained below.

1. *Formulation of baseline assumptions*: The starting point in the design of any fault-tolerant application is to lay down a set of assumptions pertaining to the performance or safety requirements, anticipated properties of the physical environment, as well as what failure types the analysis would cover. These assumptions are not dependent on the actual control task of the application and represents the knowledge of the system and its environment.

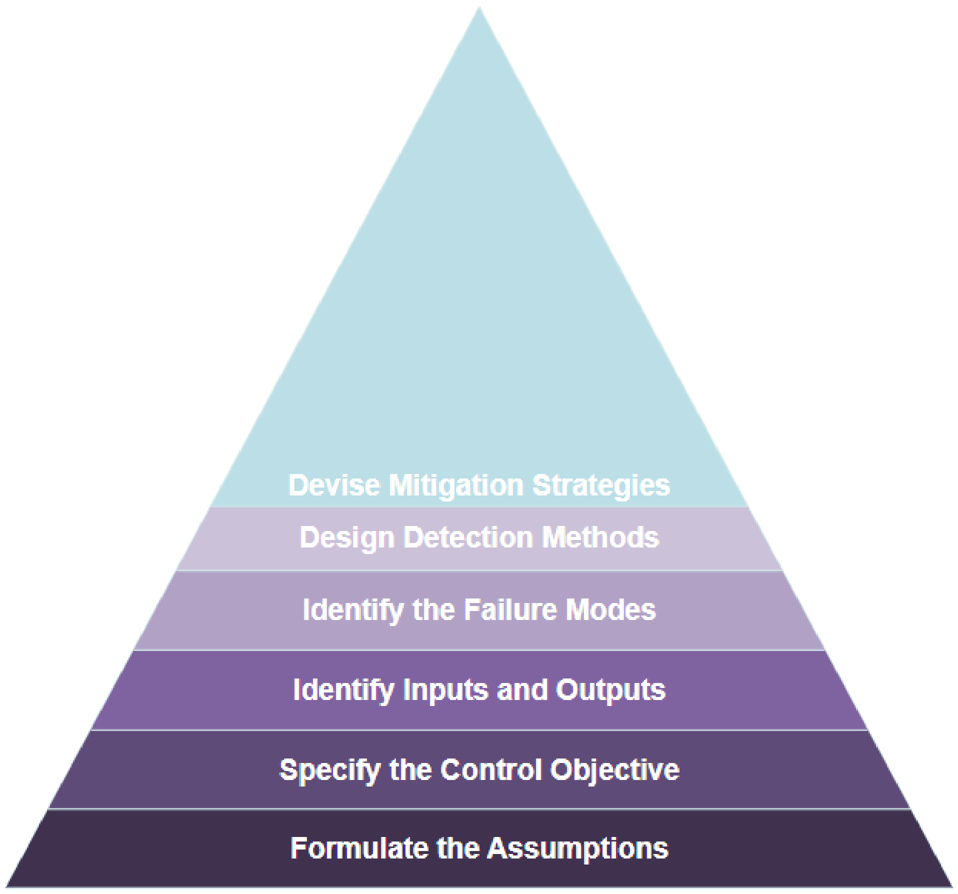

Devise Mitigation Strategies
Design Detection Methods
Identify the Failure Modes
Identify Inputs and Outputs
Specify the Control Objective
Formulate the Assumptions

Figure 1.2: The Fault Management Pyramid ©2020 IEEE

Cyber: For the cyber part, they include architectural assumptions and assumptions regarding the faulty behavior of the software.

Physical: The assumptions for the physical system include identifying the possible states of the plant elements that will be controlled by the application and their possible transitions. The state-space representation of a system implies the set of variables that can mathematically describe it at any given time, with their various possible combinations being states in which the system can exist. Transition between states always brings about a change in the state variables, and based on the system specifications, these states can be categorized into safe, marginal, or unsafe.

2. *Specify the control objective*: The control objective refers to the overall goal that the designed application is supposed to serve. It can be defined by a set of bounds on a desired observable quantity of the physical system, by putting an upper bound on the time taken to produce a response, by specifying the expected state of a component, etc.

3. *Identify the Inputs and Outputs of the components of the system*: The next step is to determine the input and output quantities that the application requires. The inputs are usually parameters of the system that provide information about the current state of the system, while the output is a measurable attribute that can provide information about the control action performed by the application.

   Cyber: For the cyber part, the input/output refers to how the physical signals are interpreted in the software itself. Specifically, how are they encoded and what are the associated data structures used to store them. In case of decentralized components, the input/output quantities also include the type of information that is being shared by the components among themselves.

   Physical: For the physical part, the input quantities are the observed values of some entity within the physical system under consideration, usually recorded by sensors. The outputs are the actuation signals sent out by the control algorithm that are used to bring the system closer to the desired control objective.

4. *Failure mode identification*: In this step, the entire application and the deployment architecture is scrutinized to identify possible points of failure. The critical questions here are *what can fail?*, *how can they fail?* and *how do they interact?*. Typically, this analysis is carried out starting from the smallest units of the architecture model and is gradually traced to higher system-level failures.

   Cyber: For a decentralized implementation, the important areas to consider are the communication infrastructure, the effect of network-induced delays, crashes of the component software or hardware (fail-silent), or a component flooding the network with bad data, often called the babbling idiot problem [Temple (1998)].

Physical: The failure modes of the physical system refer to faults in any of the devices (for example, a generator or an inverter), sensors or actuators, the ordering of events and how they determine the course of the application lifecycle, and how a possible failure in one component can propagate to induce secondary faults in other components.

5. *Detection methods*: This step involves devising features or mechanisms to detect the anomalies (deviations from expected behavior) that were identified in the previous step. The key here is identifying or synthesizing observable anomalies that are caused by the fault. Techniques might include well-known patterns such as *Heartbeats, Acknowledgments, Timeouts* [Hanmer (2013)] and so on.

   Cyber: Detection methods for cyber faults involve using exceptions or implementing a combination of the above-mentioned detection patterns either within the application software (application specific fault tolerance) or by leveraging features of the underlying architecture (architectural fault tolerance). Apart from that, built-in monitors in the software can also be incorporated to continuously monitor and detect anomalies such as usage of the hardware and processing resources of the computing node, network traffic and so on. An important factor to consider here is that in the case of a decentralized implementation, it is not only necessary for a component to detect its own fault, but it also becomes important that other components that depend on information from the faulty component also detect it.

   Physical: In general, we do not detect physical faults directly but detect anomalies using measurement data from dedicated sensors and infer the physical fault from this information. The latter is called *diagnostics*.

   A difficult case is when the detectable anomaly caused by one type of fault is indistinguishable from another. In that case, the system needs to use some other information to decide on its next course of action.

6. *Mitigation strategy*: Once a fault condition is detected, the next step is choosing a mitigation strategy. The application might either proceed with a reduced functionality (graceful degradation) or the application might try to recover from the fault. The appropriate response is dependent on factors such as resource availability, internal state of the application prior to the fault, and the type of failure, i.e., if it is a transient or a permanent failure. The goal of the fault tolerance logic is that even if the physical or the software system fails the control objective can still be satisfied.

   Cyber: Mitigation strategies for faults in the cyber layer would involve techniques such as system restart, provided the application is not dependent on the current state of the faulty component, or the component can recover its state just prior to the fault (*Roll Back*), or advance to a point where the system state is reset (*Roll Forward*). It can also fail over to a backup component or if the fault is irrecoverable (*Failover*), or proceed with the control task with a reduced level of performance.

Physical: Recovery from physical faults is a more difficult problem as it often requires manual inspection and intervention. Thus, it becomes important to determine a *default* or *safe* mode of operation in case of such failures. This ensures that the system does not degrade into a potentially dangerous state and can be recovered if the fault is repaired. Effective recovery strategies for physical faults generally involve *Redundancy* and *Failover* [Hanmer (2013)].

1.4 Fault characterization based on their origin

Faults that can arise at different layers of the system can be classified as follows [Ghosh et al. (2019), ©2019 IEEE]. The term system here includes the cyber components (both hardware and software), the physical devices as well as the network-level components, such as the routers and switches.

1. *Device interface fault:* These faults occur when a physical device does not function properly. For example, if an electrical relay gets stuck in an on/off stage.

2. *Communication fault:* These faults occur in the communication channels and can be caused by faults in the physical links, a fault in the kernel or a fault in the network hardware such as the network interface controller that causes the distributed node to become disconnected from the rest of the nodes.

3. *Hardware level fault:* These faults occur when a component raises a hardware exception, such as a CPU hardware exception or a segmentation fault. For example, if any program running on the component tries to read from or write to a memory location to which it does not have access, it would lead to, for example, a segmentation fault.

4. *Kernel level fault:* These faults are caused by a flaw in the code or resource violation that causes an invalid request to the kernel. For example, running processor-heavy operations such as machine learning tasks or bandwidth-intensive disk transfers may exceed the available CPU or disk space of the cyber node on which it is running.

5. *Process level fault:* These faults occur when one of the computation processes crashes or there are *Resource Management Faults* caused by the computational component asking for more resources than it has been assigned. For example, if a calculator process involves reading data logged to an incrementally growing database, it might eventually require more CPU time to process than allotted by the operating system and crash.

6. *Framework level fault:* A framework refers to an integrated software platform that is responsible for coordination and maintenance of the various distributed nodes of a CPS deployment. They act as a virtual operating system on top of the actual one for running distributed CPS applications, providing many useful services and features. Framework level faults are caused by errors in the framework code. An example of such a fault would be if the network discovery service responsible for identifying all the nodes connected in the network does not use the correct port to receive incoming messages from the other nodes.

7. *Logical faults:* These faults are caused by errors in the business logic of the algorithm code. For example, if a Phasor Measurement Unit (PMU) sensor in a power grid measures the voltage of a transmission line in volts (V), but in the power calculation logic, the unit used is kilovolts (kV).

As mentioned above, all these faults can propagate and lead to secondary faults. For example, a fault in the network interface of a component can prevent it from sending and receiving service queries which might lead to infinite blocking in another component that waits for a response from a service on that node. This makes it important to consider the dependency relations between the constituent entities of a distributed cyber physical system.

1.5 Research Challenges

The classical fault tolerance approaches for a CPS such as the distribution grid has progressed quite disjointedly from advances in that of computing systems. In the power systems domain, the main emphasis has been on protection schemes that use specialized hardware or control algorithms. The main philosophy behind it can be summarized as Fault Location Isolation Restoration (F.L.I.R), which translates into: *only the faulty part of the system should be identified and isolated within a minimum time so that the rest of the system operates normally while the fault is restored.* Protection devices include fuses, relays, circuit breakers, surge protectors, etc. From the perspective of control theory, the main focus is on grid stability in terms of voltage and frequency deviations. Consequently, it has led to the evolution of advanced control algorithms such as *droop control* [Tayab et al. (2017)]. On the other hand, fault tolerance approaches in computing systems, especially distributed systems, have advanced through formal analysis using concepts such as the *Consistency, Availability, Partition Tolerance (CAP)* theorem [Brewer (2012a)], including the use of redundancy in both hardware and software and the formalization of software design patterns such as *checkpointing, rollback* etc. Therefore, it is fundamentally necessary to unify the two domains such that one layer can affect and inform the other layer to become a true CPS.

Taking a more fine-grained look, the principal research challenges in the domain can be explained as follows:

1. Design of DCPS is a complex process since it relies on multiple sub-components working together without a central entity to orchestrate everything. The architecture is mostly event-driven, brought about through different interaction types among the participants leading to the desired outcome. This makes the system trajectory inherently non-linear due to there being multiple possible interleavings of different events at any point of time. There are also different interaction patterns to consider and their handling mechanisms to consider. However to design high confidence systems it becomes important to be able to put forward well-defined and formally checked specifications regarding its performance, such as timing limitations, speed of data ingestion and response etc. Leveraging model-based engineering concepts can become an important research space if adapted to serve distributed CPS. The main advantage of model-based analysis is they have proven and well-known semantics which can be used to derive concrete conclusions regarding them and they have formal verification techniques defined. However, most modeling techniques are applied to standalone systems or pieces of software. Thus, an important research question in this area is: *how to verify distributed CPS software using standard modeling techniques such that it considers the timing and interaction patterns between the individual agents?*

2. Following on from the previous point, once the system design is finalized it becomes important to weigh the pros and cons of the different configurations in which the distributed software entities can be physically located on remote edge devices. This mapping process is known as *deployment*. This is an important variable in the overall fault tolerance property of the system since the underlying deployment architecture introduces additional constraints on the distributed computational layer and the choice of the proper architecture can imbue the system with greater flexibility with respect to its resilience and in terms of resource and infrastructure costs. This introduces the following challenge: *How does the deployment configuration determine the resilience behavior of distributed CPS applications without modifying the control logic and how can it be optimized based on maximum resilience or minimum resource usage?* It is also important to have a way of rapid prototyping and testing the various deployment settings in a preproduction environment to empirically validate their resilience while introducing faulty behavior in the testing environment.

3. A distributed CPS also consists of a networking layer which plays a crucial role in the realization of its operational functions and its decision-making protocols. Thus, any comprehensive fault tolerance scheme will also need to take into account the role of communication infrastructure on the system as a whole and the impact of possible faults affecting it. The straightforward measure in this regard

would be to augment hardware and algorithmic redundancy with network redundancy, by having the computing nodes be connected to multiple network interfaces that can be switched over to depending on circumstances. Even when the communication network is operating normally, it has certain limitations with regard to its properties, such as channel bandwidth and latency, that can impact the performance of distributed algorithms. There is an inherent trade-off that exists from using a fully connected, highly responsive but congested communication graph; to using a sparsely connected, less network intensive communication graph but having a slower information dissemination speed. The relevant research question then becomes: *what role does the communication topology play in the execution of distributed algorithms for critical CPS and how can it be dynamically reconfigured to achieve a trade-off between faster information dissemination and better control algorithm convergence?*

4. Finally, to bring it all together, it is important to take the multiple mechanisms to achieve fault tolerance at both the hardware and software level, and integrate them. The combined layers of fault detection and recovery across the levels of the system must come together to create a highly resilient hierarchical fault management scheme that can deal with fault tolerance at multiple levels. The key idea is to make each layer robust so that faults from the layer below it cannot propagate to the layer above and cause a failure. The layer above can assume certain behavior about the layer below, and if that is violated, the layer below should inform the layer above. An interesting design-time decision is the choice of communication patterns in a distributed or decentralized system. Although most modern computing platforms embrace **CBSE!** (**CBSE!**) [Heineman and Councill (2001)], the choice of communication patterns still remains challenging, especially considering the fault tolerance requirements. The main question that all of these discussions ultimately boil down to is *how can the cyber and physical fault tolerance principles be integrated into a resilient cyber physical application such that it utilizes the different communication patterns and how does it weave both platform level and application fault tolerance mechanisms into it?* Chapter 1 describes the general steps and considerations that must be undertaken by an engineer in regard to developing a distributed CPS that can operate and withstand faults both in its computing infrastructure and the physical subsystem on which it operates, thus providing a philosophy for an integrated fault management approach, but, there are not many accounts that shed light on the integration of the operational and dependability objectives for any generic problem and their translation into actual code showing the usage of software and algorithms to deal with faults arising in both the cyber and physical subsystems.

1.6 Summary of Contributions

This research looks at the integration of fault tolerance through two specific aspects – *proactive* and *reactive*. Proactive fault tolerance entails the policies and procedures that are applied at design time in an anticipatory sense of the potential risks that the application is expected to encounter and endure according to its safety specifications. Reactive fault tolerance adopts a more ad hoc philosophy, wherein the application is equipped with certain detection and mitigation measures that can be leveraged by the application, either through its host infrastructure or the software algorithms themselves to take situation-specific actions during the runtime. The dissertation includes two studies each from the proactive and reactive domains that correspond to the research questions posed earlier.

1. *Model checking of CPS software using Timed Automata*: Chapter 3 introduces a tool chain for automated TA generation for DCPS software applications, called *RIAPS2UPPAAL*. The advantage of this tool chain over current state-of-the-art model generation methods is that it can integrate distributed software component code written in Python for a decentralized framework called Resilient Information Architecture Platform for Smart Grids (RIAPS) [Eisele et al. (2017a)] and assimilate them to produce a composite TA system. Thus, it manages to preserve the interaction dynamics among the constituent components through specialized modeling patterns. It can also be augmented with user-defined specifications about the timing of different operations through a simple annotation-based input method, and the tool chain combines and integrates it into the generated models. The generated TA network can then be read using the popular UPPAAL [Bengtsson et al. (1995)] model checker and verified against formally defined properties. The chapter begins with a brief explanation of the problem followed by a brief background on TA, UPPAAL and RIAPS. Then it explains the different techniques used to model the different aspects of a DCPS application illustrated with an example. The method is then applied on an example distributed application using two different deployment architectures. Finally, it goes through the verification and timing analysis of the two architectures by formulating different categories of specifications.

2. *An Automated Resilient Deployment and Testing Framework for Smart Grid Applications*: Chapter 4 tackles the problem of finding an optimal deployment scheme while balancing the resilience and economic requirements of a DCPS application by combining both decision choices into a *Resilient Deployment Solver* that can automatically produce the optimal mapping of software components to hardware edge devices. It also introduces a run-time testing framework for deploying the resulting solutions on virtual emulated network hosts running locally on a single system using *Mininet* [Lantz et al. (2010)]. The added advantage of this setup is the ability to control the disturbances by using a

behavior model that can schedule the introduction of simulated faults at specific times within the execution cycle. It consists of a formal definition of the different variables and constraints utilized by the solver along with their actual significance, followed by its evaluation using an example of a distributed microgrid energy management application. The evaluation consists of testing the solver's different solution parameters as well as its scalability performance. The test bed is then used to validate how the resilience characteristics imparted to the system as a result of the deployment configuration perform in an actual cyber physical scenario with predefined safety, performance and resource requirements.

3. *Improving Communication Robustness and Performance for Distributed Peer-to-Peer Applications*: Chapter 5 explores the inherent trade-offs that are determined by the type of communication used in distributed peer-to-peer applications. It explains the pros and cons of prioritizing the speed of communication by flooding the network with communication messages on each available communication path against prioritizing bandwidth efficiency and preventing network congestion by only utilizing a part of the communication network. It then proceeds to introduce two contributions to better evaluate and then improve this trade-off over presently available methods. The first is a configurable virtual topology creation and management framework called *TopLinkMgr* that can realize any given communication graph between a group of peer-to-peer computing agents. It does so with the help of a custom Domain Specific Language (DSL) *toplink*, which can be used to define any graph structure. Secondly, it describes and implements a novel, dynamic peer-selection algorithm called Bounded Path Dissemination (BPD) that can ensure that the maximum path length between any two peer-to-per agents in an application is within a specified threshold. The algorithm is also enhanced with resilience protocols that can prevent partitioning of the communication graph if one or more peers drop out along with restorative properties. The chapter provides the technical specifications for *TopLinkMgr* as well as *toplink* and then proceeds to describe the various steps of the BPD algorithm. It then shows its implementation on a distributed averaging problem running on a network of nodes and demonstrates how the convergence accuracy and speed of the algorithm compares with using other peer selection strategies. The resilience performance is then evaluated with respect to interruptions in the communication graph and how BPD can compensate for nodes dropping out.

4. *Fault Tolerant Load shedding using a Decentralized Software Platform*: Chapter 6 provides a road map for combining platform level fault tolerance features with specific behavior handlers and control logic modifications at the application level to implement a robust, fault tolerant CPS application in the energy domain. Specifically, it describes the fault detection and recovery features of the decentralized software platform called RIAPS and discusses the important role that those underlying platform services play,

in conjunction with design choices such as communication protocols, in the fault-tolerant behavior of a DCPS. It describes how the different services react and interact to different fault signals and the way application developers can leverage the detectors and handlers provided by the abstracted services to weave fault tolerance features into the high level control algorithm itself. The selected example application is a distributed load shedding scenario, which is a demand curtailment technique that is used in the field of electric distribution systems to disconnect certain consumer lines during peak usage to prevent grid overload. The experiments were performed using a discrete time cosimulation test bed using the GridlabD [Chassin et al. (2008)] simulation software to model a distribution system with load shedding control running externally on networked embedded devices. The chapter describes the details of the load shedding algorithm without the fault tolerance features first, and then shows how it can be modified based on the anticipated failure modes and the available platform level services. The modified algorithm is then tested using the cosimulation framework with respect to different fault scenarios to analyze their robustness performance.

CHAPTER 2

Related Research

2.1 Fault tolerance Design Approaches

Having a dedicated fault management subsystem allows the construction of resilient cyber-physical systems with weaker assumptions regarding the anticipated failure modes of the system. For example, consider a system composed of *n* distributed agents, each running its own process. At each round, all agents broadcast their local state. Upon receiving the same information from its peers, it performs a processing step to calculate its next state, followed by another broadcast. This is a fairly common controller state flow for a CPS. Let us also assume that the communication channel is unreliable, but fair. A fair channel implies that although there is no guarantee that a transmitted message will always be successfully received, if a message of a specific type is transmitted infinitely often, then eventually it will be received. In that case, in order for each process to reliably broadcast a message, the system must be qualified with an additional assumption that *the majority of distributed processes in the system must be nonfaulty*, or in other words, the total number of crash failures within the system must be $< n/2$. Reliable broadcast would mean that any message broadcast by a process is eventually received by all other non-faulty processes in the system. This can be achieved in software by including the *id* of the sender and a *sequence number* with the actual content of each sent message on the sender side and then have it listen to acknowledgment messages which are sent by the receiver.

The assumption works because it implicitly allows processes to determine that at least one nonfaulty process has received the broadcast if they receive $N/2+1$ acknowledgment messages, which in turn guarantees that other nonfaulty processes will eventually receive the message in subsequent broadcasts. However, the implementation of a dedicated fault detection and monitoring architecture can inform the nonfaulty agents if any of the other nodes crash and thus remove the need for this additional assumption. This would allow developers to build distributed algorithms that can also be fault-tolerant.

There exists quite an extensive body of work on fault tolerance studies in the case of computing systems. With the advent of the distributed computing paradigm in the 1980s, there were new failure types that needed to be accounted for and handled in the design of such systems. This led to several architectures and design patterns with their own advantages and disadvantages. In general, fault tolerance approaches can mainly follow one of two philosophies [Raynal (2018)]:

1. *Masking*: Masking fault tolerance approach is based on the philosophy of nullifying the impact of a fault on the whole system. The most common way to address this is to use redundancy in both

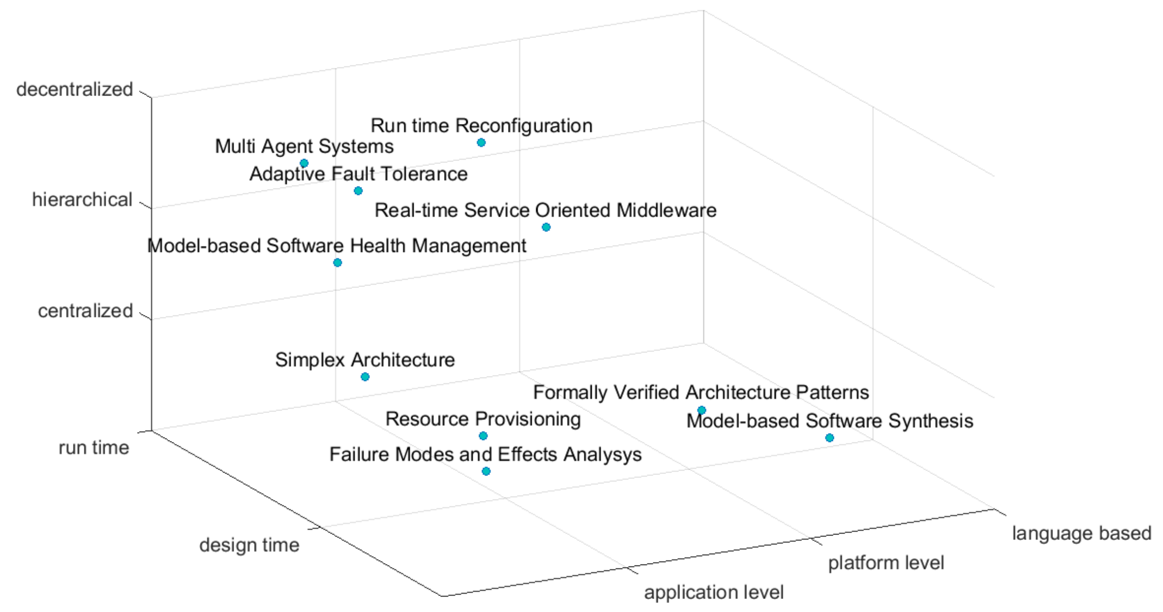


Figure 2.1: The overlapping axes of fault tolerance design in CPS (all examples are not shown due to space limitations)

hardware and software as required. Such an approach results in a highly available system which can *mask* an agreed number of concurrent failures according to the reliability specifications, but this added redundancy may considerably increase the cost of the system.

2. *Non-masking*: Non-masking fault tolerance, on the other hand, does not provide any guarantees of correctness in the presence of faults. It states that upon the cessation of the fault condition within the system, the tasks executed by the system revert to a state that satisfies their said specifications.

As discussed in the previous section, application designers need to carefully consider the general system specifications and assumptions and carefully integrate fault tolerance logic so that it can maintain the system performance and safety specifications in the presence of faults. Of course, it is practically impossible to design a perfectly robust CPS that is resilient against all possible fault conditions. However, a thorough and well-informed analysis of the possible failure modes of the system during design time provides a solid platform for a robust fault management plan which can, for most anticipated failure scenarios, keep the application functional up to a certain prenegotiated level of performance. Another important consideration is the available budget in terms of computational and hardware resources, which puts an upper limit on the comprehensiveness of the fault tolerance design framework.

Upon examining the current and past literature on fault-tolerant CPS, keeping the above considerations in mind, a few distinct directions emerge: with respect to (1) the framework architecture, (2) the design philosophy followed, and (3) the scope of the fault tolerance measures. These directions, or *axes* can be thought of as acting orthogonally to each other so that they constitute a three-dimensional design space. The various approaches found in the literature occupy a point in this space. Thus, there can be copious amounts of overlap between the three axes for multiple instances. An illustration of this is provided in Figure 2.1.

Taking a closer look at the axes, the first consideration takes into account the architecture chosen for the fault tolerance implementation. Depending on the scale of the system, the requirements, and the available resources; application designers need to decide on a structure that best suits the application needs. This process involves a trade-off between making the fault tolerance subsystem powerful enough to account for a large-scale safety critical system and making it too complex such that it itself becomes a potential weakness that adds considerable computational overhead that affects the ability of the system to operate efficiently. The choice of architecture is an important decision that engineers need to make when designing a CPS-based solution for a given system. The term *architecture* in this sense refers to the arrangement and interaction of the various components that constitute the fault tolerance subsystem of the given CPS. Some consist of a centralized entity responsible for monitoring the system and then dispatching fault handling commands when required. Local nodes only relay information to and from this agent and are not responsible for any decision making [Iamnitchi and Foster (2000), Bintoudi et al. (2020)]. A hierarchical architecture usually employs one top-level fault management agent with one or more levels of subordinate agents [Panahi et al. (2012) , Dubey et al. (2011), Januário et al. (2019)]. The flow of control usually flows from bottom up with each successively higher level responsible for the level below it. A fully decentralized architecture employs a group of autonomous entities or agents that work together to achieve the resilience properties of the system [Kumar and Cohen (2000), Becker (1994), Chen et al. (2015), Aidemark et al. (2005), Khan et al. (2016), Sanislav et al. (2016)]. It differs from the hierarchical architecture in the sense that all agents perform their own tasks without being managed by any other agent above it. They can communicate among themselves to determine certain actions, but all their decision-making is independent.

The second axis is based on the applicability of the fault tolerance design approach. As a designer, one can choose, up to a certain extent, how tightly integrated the fault identification and mitigation protocols can be with the actual application logic. Certain features such as hardware redundancy, monitoring, and communication infrastructure are generic reliability requirements independent of the application. They can be built into the system architecture itself and would be applicable for any scenario. Building the system by composing these algorithmic blocks in a certain manner can then imbue the system as a whole with resilience characteristics. The advent of dedicated platforms which can host CPS applications, monitor and manage them are equipped with a number of robustness qualities which are imparted to the hosted applications as specialized services. Thus, fault tolerance of such systems is the property of their underlying platform [Akyol et al. (2012), Debnath et al. (2019), Dubey et al. (2017), Filgueiras et al. (2019) Kim et al. (2020), Kumar and Singh (2019), Mohamed et al. (2017), Pradhan et al. (2015), Xu et al. (2017)]. Middleware frameworks are a good example that have undergone widespread adoption in DCPS [Akyol et al. (2012), Kumar and Singh (2019), Pardo-Castellote (2003), Filgueiras et al. (2019), Debnath et al. (2019), Mohamed et al. (2017), Foundry (2020)]. The original definition of a middleware is *software that mediates between an application program and a network. It manages the interaction between disparate applications across the heterogeneous computing platforms* [Howe (2012)]. Thus, it is an intermediary piece of software that handles the interaction between software agents while abstracting the details at the application level. They are ubiquitous in distributed systems in implementing messaging brokers, remote procedure calls, database interactions, etc. This can be very useful in realizing large scale, complex DCPS applications, as they can then take care of all the communication pipelines leaving the business logic to be free of them. However, there are some fault handling behaviors that are tightly coupled with the application logic itself. This can then be termed as *application level fault tolerance*. For example, determining what should be the *safe* mode of operation of a component once a fault is detected or what specific action should be taken so that the algorithm can proceed in spite of a non-ideal event is specific to the actual task logic and its corresponding safety criteria. As design specifications move from general to more specific considerations, it becomes increasingly more difficult to separate the fault tolerance logic from the application functional logic [Roxana and Eva-Henrietta (2017), Singh and Sprintson (2010), Gunes et al. (2013), Vachtsevanos et al. (2018)]. Fault tolerance properties can also be integrated into the way the programming model of the various algorithms is developed for a CPS application. The term *language-based fault tolerance* can be used to denote this category [Birman et al. (1985), Kiczales et al. (1997), Zheng et al. (2019)].

A CPS by definition involves a tight coupling between its *cyber* layer or the software and computing infrastructure and its *physical* layer or the actual plant hardware devices that are being controlled. A comprehensive fault management strategy must consider fault scenarios originating from both parts and how they influence each other. System designers can then opt to determine fault handling policies at design time by anticipating possible failure modes [Faza et al. (2009), Sha and Meseguer (2008), Tanganelli et al. (2020), Wu et al. (2017) , Xie et al. (2018)] or it can equip the system with a run time analyzer and decision maker that can gauge the system parameters during fault occurrence and then determine the best course of action [Ali et al. (2018), Jagtap et al. (2020), Krishna and Koren (2013), Kong et al. (2018) Ratasich et al. (2017), Wang et al. (2013)]. Analyzing the literature through this lens leads to the third axis, which is the fault tolerance scope, either *design time* or *run-time*.

2.2 Model Checking Approaches for CPS

Analysis of software is a challenging problem due to the absence of strict rules of construction and heterogeneity in coding styles, structures, and patterns. Model checking and model-based abstractions of software have been investigated in the past, particularly for improving code clarity and interpretability. For example, Hovemeyer et al. (2016) develop an extension to Abstract Syntax Tree (AST) called Control Flow Abstract Syntax Tree (CFAST) that focuses on the control structures. They use it to analyze and compare different programming styles by students and try to ascertain if the students are using the same control structures to solve a given problem. Although the purpose of the study is not systems design but more to understand programming behavior, it shows the usefulness of representing software using an AST like intermediate representation.

In the context of automatic model generation from system software, broadly two approaches have been followed. Both have interesting contributions and features that have inspired certain choices in this research. The first is from analyzing the source code itself and using a translation process that can perform the model transformation and generation. These approaches include works like Hamdane et al. (2013), where the authors defined metamodels for the Architecture Analysis and Design Language (AADL) [Feiler et al. (2006)] and TA models and used them to perform the model traversal and model transformation. The AADL model was translated into an intermediate TA specification and then converted to the TA language, which is compatible with the UPPAAL tool. The authors did not consider interaction between distributed components in their work and used a simple water leveling system to demonstrate the transformation process.

Herber et al. (2015) designed a transformation tool that can produce timed automata models from SystemC [Panda (2001)] code. The process involves two steps. First, the AST generated for a given SystemC module is translated into an intermediate representation consisting of a hierarchical class structure corresponding to the SystemC elements. The intermediate model is then transformed into a TA that can then be imported into UPPAAL for further analysis. Apart form the model-to-model conversion, the proposed tool also performs some optimization related to data race conditions and memory usage in the first stage and simplifying the TA model in the second stage. The tool was evaluated using three case studies which were successfully converted into TA. However, the transformation tool does not provide any option for users to determine the level of abstraction while constructing the TA models.

Kulczynski et al. (2021), use LLVM [Lattner and Adve (2004)] code to generate UPPAAL timed automata. The main advantage of LLVM code is that it is a general-purpose metalanguage on which several programming languages have been developed. Thus, it becomes compatible with all languages that can generate an LLVM representation. However, the LLVM library is vast, making implementing the whole language extremely complex.

Liva et al. (2017), Spalazzi et al. (2018) extract timed automata from Java methods by looking for calls to standard library functions that manipulate timing. They defined the semantics for each of the standard functions and used them to generate states and transitions using an algorithm. The models were then loaded into UPPAAL to visualize and verify the properties. They can only analyze a single method at a time and do not consider any interactions between distributed processes. However, it is still an important result that shows the feasibility of TA in performing the timing analysis of software systems.

The second approach is gathering run-time information about the system to be modeled and then using it to construct a correct representation. Tappler et al. (2018) use genetic programming to learn TA models from System Under Test (SUT) implementations. The process involves generating simulation traces of the input system subject to various configurations and using them to train TA. Experiments on different real world as well as artificially generated TA models showed good correspondence between the actual model properties and the learned TA properties. However, the input for their algorithm were TAs themselves, thus it is not clear if the approach can be translated into software. Also, since the algorithm always starts from an initial guess, it involves a lot of random transitions that take up a lot of time to converge. This step can be optimized if the existing knowledge of the developer can be used to guide the model generation process.

Pastore et al. (2017) use a specification mining technique called *Timed-k Tail* to analyze traces of system execution and generate TA that captures both the sequence of events and the timing information. The traces log the execution of the operations and their duration. The algorithm creates a separate branch for each trace of the same program and dedicates a separate clock for each step. Thus, it needs a complex merging algorithm to create the final automata.

Similarly, Cornanguer et al. (2022) describe a novel algorithm Timed Automata Generation (TAG) that can learn TA from collected logs. One of the advantages that it has is that it uses a global clock that keeps track of the total system time. However, the model is geared more towards probabilistic verification queries and it also requires multiple iterations of merging and splitting which can be quite complex.

2.3 Allocation Strategies for Distributed Application Components

Over the years, there has been substantial research on distributed systems deployment and synthesis that has, in turn, shaped this research. However, they are restricted to only hardware reliability [Onishi et al. (2007), Glaß et al. (2007)] , require fine-grained information [Gao and Wu (2007)], or require enforcing constraints in the design itself [Maticu et al. (2016)].

Cloud-fog computing architectures for Internet of Things (IoT) applications also aim at the allocation of distributed processes or services to computing hardware while trying to optimize some criteria. Roy et al. (2011) use a bin-packing heuristic to determine the placement of components that implement web services on machines. However, it only looks at CPU utilization as a criterion for allocation. The algorithms in Xia et al. (2018), Zhao et al. (2018), Herrera et al. (2021) also incorporate other information such as geography, link characteristics, and interference sources along with computational resources to calculate the optimal placement of the fog nodes. No evaluation of fault tolerance performance is performed after the deployment. They mainly aim to reduce latency, response times, and managing node resources. The use of software containerization as a deployment unit is becoming increasingly popular in the distributed systems domain. To manage, execute, and deploy such container-based applications, orchestration platforms such as Docker Swarm [Magableh and Almiani (2019)], and Kubernetes [Zhou et al. (2021), Rossi et al. (2020)] provide services that can automate container provisioning, management, and communication. However, these are all run-time monitoring tools that usually rely on static thresholds to automatically scale the cluster size depending on the computational load. Such automatic scaling approaches are a challenge for CPSs like smart grids, where the heterogeneity of the various components and dependency on actual physical hardware mean that they cannot always be mapped to any available node.

Among other constraints-based works, Chariot [Pradhan et al. (2018)] employs a run-time self-reconfiguration algorithm to deploy IoT applications to edge devices, but it relies on commands from a continuously running manager component in all nodes and cannot be used in design time. Zephyrus2 [Ábrahám et al. (2016)] introduces replication and separation constraints, similar to this work (Chapter 4) and uses a specification language, but the platform is not designed to consider CPS. The system architecture considered by Nuzzo et al. (2020) assumes a fixed number of nodes, while Manolios and Papavasileiou (2010) only considers resource allocation.

2.4 Peer-to-Peer Communication Strategies for Distributed CPS

There is a substantial amount of literature available in the field of distributed cloud and fog computing on the topic of overlay networks. An overlay is a virtual network built on top of the network layer, supported by its infrastructure that is capable of selectively forwarding packets to peers in the network, in an application-specific way [Peterson and Davie (2007)]. Various types of overlay topologies exist in the literature and can be categorized into two categories, namely *structured* and *unstructured*. Structured overlays are static in nature, where connections are predefined and remain unchanged throughout the operation lifecycle [Castro et al. (2002), Dhara et al. (2010), Shukla et al. (2021)]. On the other hand, in unstructured peer-to-peer overlays, connections are determined at run-time and can change at each round [Ganesh et al. (2003) , Genç and Özkasap (2005)], Jin and Chan (2010). Among them, gossip-based protocols have found widespread adoption in industrial distributed systems use cases such as Cassandra [Lakshman and Malik (2010)] and Hyperledger [Androulaki et al. (2018)]. In a gossip style communication, at each iteration a peer randomly selects a subset of peers to send its updates to. They focus mainly on the efficient organization of the nodes to optimize parameters such as latency [Chen et al. (2021)], for file storage and look up [Monga et al. (2019), Shojafar et al. (2017)]. In Voulgaris et al. (2005), a peer selection strategy is introduced which is built by improving the randomized shuffling protocol. Using this protocol, an efficient topology management scheme was developed in Jelasity and Babaoglu (2005) for a distributed file sharing application. Other algorithms seek to minimize proximity to centralized cloud data centers Costa et al. (2020) or dynamic clustering in the case of moving nodes, such as vehicles Rashid et al. (2020). The general principles and solution techniques used in these works focus on factors important for large-scale dynamic distributed systems where better load balancing and faster response times are desired. For a peer-to-peer CPS application like smart grids, the key factor is the distribution of information generated from all peers to all other peers throughout the network efficiently and quickly.

Overlay solutions have recently been implemented for smart grid platforms [Marzal et al. (2017), Tebekaemi and Wijesekera (2019), Orda et al. (2021)]. These approaches mainly look at using some heuristics to optimize the routing efficiency of messages based on information about the power network. However, the experiments performed always assume a network model that has a few hops to any node, which is the problem we are trying to solve. They are mainly dependent on Software Defined Networking (SDN). SDN uses software APIs to simulate network layer devices such as routers and gateways and can be used by network engineers to tweak certain Quality of Service (QoS) connection parameters. These changes are usually permanent and useful for optimizing communication performance with respect to the electrical grid under consideration. Implementing it requires reconfiguration of the entire network infrastructure to make it compatible with specific controllers and protocols, incurring significant costs. Also, SDN usually employ a logically centralized control plane, which essentially becomes a single point of failure and introduces higher security risks.

CHAPTER 3

Model Checking of Distributed Cyber Physical Systems Software using Timed Automata

3.1 Problem Statement

Design of fault tolerant DCPS is not a trivial task considering the heterogeneity of the different players involved and their associated uncertainties. However, due to the safety critical nature of a majority of these systems, it becomes important for both system designers and stakeholders to get some guarantees regarding their construction and performance under different conditions. While it is possible to analyze a single piece of software or isolated system in its entirety using well-known modeling and verification techniques, the process becomes more challenging for a distributed architecture since it involves multiple communicating, interdependent entities working together.

The timing analysis of distributed software systems for CPS is one of the challenges that arises due to the reasons mentioned above. The architecture that we consider here is a network of event-triggered , inter-communicating processes or components that are spread over a physical network and work collaboratively to achieve some control objective pertaining to the physical system, for e.g. the electric grid. Examples of events can be new sets of sensor readings, arrival of messages from upstream processes or the passage of time itself to trigger a new process. These different events and streams of information flowing through different distributed processes, with a defined origin and end but different possible interleavings in between, constitute distributed algorithms that achieve the said control objectives.

Model-based techniques use different modeling abstractions with well-defined semantics to represent CPS and serve as an effective tool for analyzing their behavior. The advantage of using model-based analysis is that it can provide formal guarantees about certain properties of the system. Performing the formal analysis manually can quickly become tedious as the system scales up and requires extensive rounds of testing and validation to find out the exact input and event sequence that can violate a property. Also, debugging is a complicated process for distributed systems which requires collecting logs from each of the target nodes and going through them to identify the concerned anomaly. Most modeling languages such as finite state machines, dataflow models, synchronous reactive models, timed automata, etc. [Lee and Seshia (2015)] allow formulation of logical verification conditions that can be tested on the model. Many of the modern modeling tools can also generate execution traces for cases when the verification condition is violated. This is called counterexample generation. It is the job of the designer to determine the most appropriate model for the given application domain and the most accurate representation rules of the different aspects of that application domain within the modeling paradigm. However, they have mostly been used to model standalone systems or each individual entity of a distributed system in isolation. This work extends the model-based analysis to look at distributed software systems as a composite entity. It introduces a method to validate its design and timing related properties and how the deployment architecture, i.e., the communication configuration and arrangement of the individual components running the various processes, can have an impact on those properties. TA [Alur (1999)] is chosen as the model of computation since it allows for the explicit modeling of time using the concept of logical clocks and can represent time-based conditions on states and transitions required for timing analysis.

However, the benefits of model-based analysis are only significant if the effort required to translate the distributed software artifacts to the modeling language, TA in this case, does not overwhelm the design process. This is a nontrivial task that consists of: First, determining the appropriate level of abstraction that can represent the various participants, the data structures of the system with the required level of fidelity to perform the analysis without leading to the so called state explosion problem Clarke et al. (2012). We introduce one such technique of TA generation whereby starting from a collection of DCPS software components, their interactions, and information on their logical and deployment architecture, it can automatically produce a TA representation of the whole system. Secondly, allowing the model to easily incorporate modifications and property specifications provided by end users. This is achieved through a domain specific language for users to annotate the code itself with specific timing information about their execution and then integration of that information with the generated model. The result is a streamlined and generalized workflow to translate DCPS software into TA that can then be visualized using any standard model checker, such as UPPAAL Behrmann et al. (2004), which is specifically designed for TA. UPPAAL supports a simplified version of Computation Tree Logic (CTL), a formal verification language to formally define the requirements specifications and verify the model Reynolds (2001). We then demonstrate our model checking approach on a distributed application example having a non-trivial architecture to perform timing analysis. For example, check the stimulus-to-response timing interval constraints of the system for different architecture configurations and show how the varying interleaving of different events can affect the timing.

As an example, the distributed software application is developed using a decentralized middleware framework called RIAPS [Eisele et al. (2017b)] since it implements a rich domain specific language model defining how the individual components communicate and are mapped to actual hardware platforms. In RIAPS, an application consists of multiple actors deployed to remote nodes called RIAPS nodes, where each actor can contain multiple instances of components that perform dedicated functions. The components support a wide variety of interaction patterns through end points called ports. RIAPS allows the components to be encoded as Python classes with a dedicated API for component level and framework level functions.

In details, the contributions of this work are the following:

1. Automated TA generation framework RIAPS2UPPAAL that can take a RIAPS application model, deployment model and the Python source code for RIAPS components and produce a timed automata model in XTA language that can be read by UPPAAL.
2. Modeling six different port types that support three different communication patterns for distributed systems and the timer port to generate timed events using TA constructs supported by UPPAAL.
3. Integration of user timing specifications as clock conditions and invariants in the generated model, from annotations added to the source code itself.
4. Performing model-based analysis of a practical example by formulating verification queries in UPPAAL for different deployment configurations and generating counterexamples.

The said contributions are adapted and expanded from the published work Ghosh, P. and Karsai, G. (2023). Distributed cyber physical systems software model checking using timed automata. In *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE ©2023 IEEE.

3.2 Solution approach

3.2.1 Timed Automata

A timed automaton Alur (1999) *TA* is defined as a tuple $\langle L, L_0, \Sigma, C, I, E\rangle$, where L is a set of *locations*, $L_0 \subseteq L$ is a set of *initial locations*, Σ is a finite set of *labels*, C is a finite set of *clocks*, $I : L \to \Phi(C)$ is a set of *invariants* that map each location in L to some clock constraint defined in $\Phi(C)$, which are simple conditions of clocks with other clocks or real numbers, and $E \subseteq L \times \Sigma \times 2^C \times \Phi(C) \times L$ is a set of *edges*. An edge is defined as a tuple $\langle l, a, \phi, \lambda, l'\rangle$ from locations l to l' on symbol $a \in \Sigma$. $\phi \subseteq \Phi(C)$ denotes a clock constraint that defines when that edge is enabled, and $\lambda \subseteq C$ is the set of clocks that are reset when the edge is taken.

The semantics of timed automata is defined by associating a transition system $S_A = \langle S, s_0, \to\rangle$ with it. Here, S is a state of states where each state s is defined as $\langle l, v\rangle$ with l being a location in L and v being a valuation of clocks that satisfies the invariant $I(s)$. $s_0 = \langle l_0, v_0\rangle$ is the initial state and $\to$ is the transition relation. There can be two types of transitions:

1. *Delay transitions:* For a state $s = \langle l, v\rangle$ and a real-valued time interval δ, $\langle l, v\rangle \xrightarrow{\delta} \langle l, v+\delta\rangle$ if $\forall\ 0 \leq \delta' \leq \delta$, $v+\delta'$ satisfies the invariant $I(s)$.
2. *Action transitions:* For a state $s = \langle l, v\rangle$ and an edge $\langle l, a, \phi, \lambda, l'\rangle$ such that v satisfies ϕ, δ, $\langle l, v\rangle \xrightarrow{a} \langle l', v[\lambda := 0]\rangle$.

Thus, the label set for the transition system S_A is $\Sigma \cup \mathbb{R}$.

3.2.2 UPPAAL

UPPAAL is a real-time model checker and verification engine based on timed automata [Bengtsson et al. (1995)]. It was jointly developed by Uppsala University and Aalborg University and is one of the most widely used tools in CPS research.

3.2.2.1 Modeling Language

UPPAAL introduces some additional concepts to timed automata. The ones which are relevant to this work are discussed below [Behrmann et al. (2004)].

- *System:* In UPPAAL, a system is composed of a network of TA defined as processes, which are instances of *templates* with associated arguments and consisting of locally and globally defined variables.
- *Struct Variables and Call-by-Reference:* UPPAAL allows a declaration of complex data types using the `struct` keyword similar to C. They can also be customized using the `typedef` keyword. Also, parameters can be passed by reference using the *ampersand (&)* prefix, similar to *C++*.
- *Template:* A template defines an automaton that consists of locations and edges. It can also have local declarations and parameters that are substituted for a given argument during the process declaration.
- *Committed Locations and Invariants:* Locations defined in UPPAAL can be qualified with invariant conditions in agreement with the TA definition.

  Furthermore, the locations can be denoted as *urgent* or *committed*. Urgent locations freeze time. If a process is in such a location, time is not allowed to pass. A committed location has the same property, with the added restriction that if any process is in a committed location, the next transition must involve an edge from one of the committed locations. They are useful for representing atomic sequences and interleaving between different automata.
- *Synchronization and Channels: Channels* are a special data type that can be used to sync processes in UPPAAL. This can be achieved by using the *synchronization* property of the edges. Two edges in different processes can synchronize, i.e., execute the transition associated with that edge simultaneously, if they have complementary synchronization labels annotated with the same channel, provided that the guards are satisfied. The syntax for the labels are `channel_name?` for the sender and `channel_name!` for the receiver, respectively. Channels can also be declared as *urgent* or *broadcast*. Urgent channels ensure that the said synchronization is taken as soon as it is enabled. Broadcast channels allow $1-to-n$ synchronization and is useful if more than two processes need to be triggered.

3.2.2.2 Verification Queries

UPPAAL can be used to specify symbolic queries using formal syntax that are verified using the semantics of TA. The syntax is similar to CTL [Reynolds (2001)] and can be classified into state formulae and path formulae. The latter can be further categorized into *safety*, *reachability* and *liveness* properties.

- *State Formulae:* State properties can be defined using any expression and referencing the state using `process_name.location_name`. The state property `deadlock` can be used to check for deadlock conditions.
- *Safety Properties:* In UPPAAL, these include the *invariantly* (`A[]`) and the *potentially always* (`E[]`) properties. The property `A[] p` evaluates to true if (and only if) every reachable state satisfies *p*. The property `E[] p` evaluates to true if and only if there exists a maximal path for which *p* is always `true`.
- *Reachability Properties:* This includes the *possibly* (`E<>`) property. `E<> p` evaluates to true if *p* is satisfied by any reachable state.
- *Liveness Properties:* These include the *eventually* (`A<>`) and the *leads to* (`-->`) properties. The property `A<> p` evaluates to true if (and only if) all possible transition sequences eventually reach a state satisfying *p*. `p --> q` can be interpreted as whenever *p* holds true, eventually *q* will hold `true`.

3.2.2.3 RIAPS Component Model

RIAPS [Eisele et al. (2017b)] is an open, decentralized middleware platform for the development and orchestration of distributed software for smart grids. It supports the execution of distributed software applications that run on a network of computing nodes by facilitating application deployment, providing for the remote management of applications on nodes, supporting reliable communication patterns among computation units, enabling distributed data management and resource sharing, providing distributed coordination and high-precision time synchronization services between the nodes, and implementing a robust fault management framework that enables the applications to be resilient and reliable across all architectural layers. Additionally, it provides a domain specific modeling language to define the application deployments, including the target nodes, as well as the application structure description model file. Details about the RIAPS API and modeling language can be found in [RIAPS (2017)].

A RIAPS application is realized through distributed *Components* wrapped in *Actors* (see Figure 3.1). In the RIAPS execution model, a unit of deployment is called an actor. It is analogous to a process in Linux. An actor contains one or more reusable components. The advantage of encapsulating multiple components

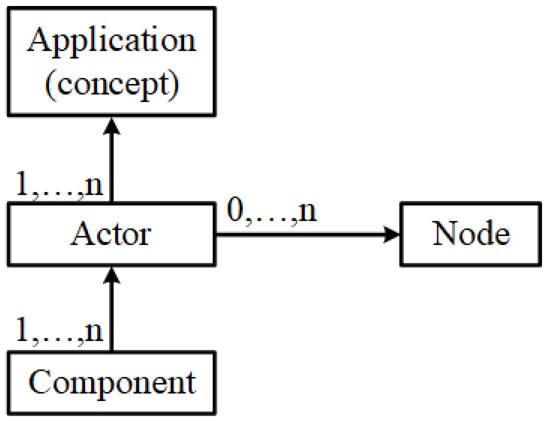


Figure 3.1: RIAPS application/actor/component relationship.

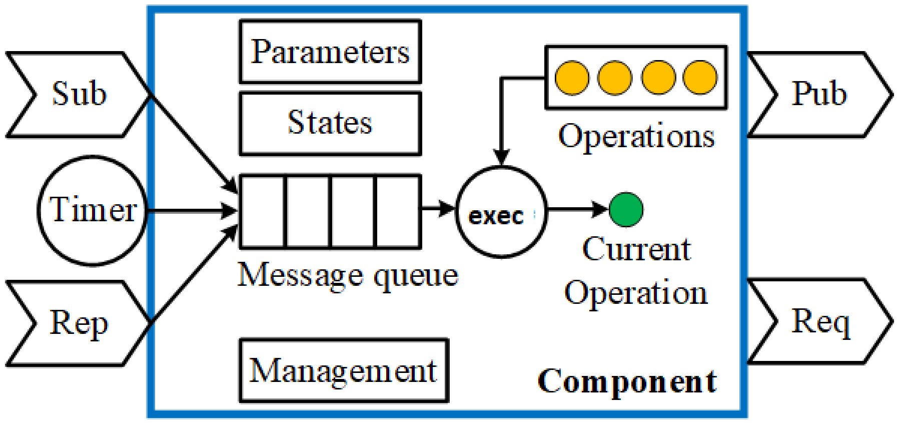


Figure 3.2: RIAPS component model ©2020 IEEE

within an actor is the reduction in the cost of communication between components since they are part of the same process.

Components implement the business logic of the application. RIAPS employs a single-threaded message passing based programming model for the components, which simplifies the timing and synchronization constraints. A RIAPS component can have different kinds of ports: *request*, *reply*, *client*, *server*, *query*, *answer*, *publish*, *subscribe* and a special type called a *timer* port. These ports have specific properties that are applied for suitable communication patterns. All ports have associated handler methods that can be invoked within the component code to perform specific functions when a message arrives, or a timer expires. These handler methods are invoked when there is any incoming messages to the corresponding port that arrives from another component or triggered by a timer thread. Each component also has an associated *scheduler* that is responsible for polling on the different port interfaces for incoming messages and then scheduling the corresponding handler methods based on some policy, such as *round robin*, *fixed priority*.

ZeroMQ [Hintjens (2013)] is used as the underlying messaging layer. Message serialization can be achieved through Cap'nProto [Varda (2013)] because of its fast in-memory marshalling and un-marshalling capabilities. Figure 3.2 illustrates the RIAPS Component model.

3.2.3 Automated Model Generation

The model generation process for the RIAPS2UPPAAL framework consists of a *parsing* stage, a model *generation* stage, and a model *merging* stage.

First, the information about the distributed application model and its deployment configuration is read and used to determine the number and type of global variables, template definitions, and their associated parameters. In the context of RIAPS, this information is included in two files. The first is an `app.riaps` file describing the number of distributed components, their associated ports and message types and how one or more components are grouped into actor processes. The deployment information is specified in an `app.depl` file, which maps the defined actors to remote networked computing platforms that run them.

Next, using the parsed data, TA definitions are generated based on the deployed component instances, the associated port types of the components, and the component implementation code in Python, using specific placeholder templates also generated in Python.

Finally, the process definitions are added along with the necessary arguments using the TA definitions. The generated artifacts are merged together into a single `app.xta` file which can then be ingested by UPPAAL.

In order to translate a given application into a network of timed automata, it is important to determine the level of abstraction to which the constituent elements need to be represented. Since the purpose of this research is to perform application level analysis, especially for timing related properties of the system as a whole, the models capture the application logical architecture, the component port handlers for reacting to different triggering events at the level of the RIAPS framework, and how the RIAPS component model schedules the handling of multiple such port events. The lower level details like how the messaging framework works or how the operating system schedules the actor processes are abstracted.

3.2.3.1 Urgent Transitions

The semantics of TA does not force a transition to be taken as soon as they are enabled. This is an implicit assumption for software instructions. Thus, to model this behavior, the *urgent edge* pattern [Behrmann et al. (2004)] is utilized. It involves an additional automata with a single location and an outgoing edge labeled with a synchronization label. Any edge synchronized on the same label thus becomes an urgent edge. Figure 3.3 shows the implementation using two templates. The template *TransitionHelper* implements the always active outgoing edge using the label *go* which transforms the transition using the corresponding synchronizing edge

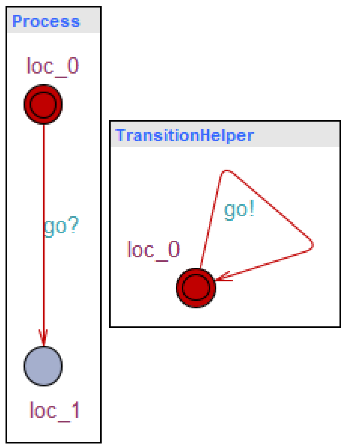


Figure 3.3: Urgent transition implementation

of the template *Process* into an urgent transition.

3.2.3.2 Interaction Types

Distributed systems can employ a number of communication patterns depending on the type of triggering event (periodic or aperiodic), the number of participating agents (one-to-one or one-to-many or many-to-one), the frequency of communication (send and receive in step or asynchronously, fire and forget), etc. As introduced in Section 3.2.2.3, in RIAPS, special interfaces called *ports* are defined to implement the different interaction types. There are eight different port types, namely, *request (req)*, *reply (rep)*, *client (clt)*, *server (srv)*, *publish (pub)*, *subscribe (sub)*, *timer (tim)* and a special type called *inside*. The connections between the ports are defined using *message types* with the ports linked by the same message types communicating by sending and receiving messages. The data structures and model patterns implemented for the different interaction types are described below. These patterns include whole templates as well as fragments of templates that are used as building blocks to compose a composite TA for each component declared in the application model.

- *Port Queues:* Each port is associated with an internal queue into which messages are pushed during a *send* operation and then subsequently read during a *receive* operation. This is achieved by declaring a `struct` data type `intq`, essentially implementing as a queue data structure. There are two operations, `push` and `pop`, defined for the `intq` data structure, for writing into the queue and for reading from the queue, respectively. These are implemented using simple C-like functions supported in UPPAAL.

- *Timer:* This type of interaction involves one sender and one (or more) receivers. The communication proceeds in a lockstep, i.e., a send operation must be followed by a corresponding receive. This is achieved through the *request* and *reply* ports and the *send* and *receive* methods in RIAPS.

The *request* port is not explicitly modeled in UPPAAL, instead, the *send* operation on a *request* port is directly implemented within the *component* automata itself. Figure 3.5a shows the send operation, modeled as a synchronized edge leading to a committed location `post_send`. The transition produces the synchronization signal on the channel `req_port_channel` which is a placeholder for its corresponding channel. One such channel is declared for each port belonging to every component instance.

Each reply port is modeled as a separate automata, as shown in Figure 3.5b. It consists of a single location with an edge synchronized on the same channel as the request port, which is passed as a parameter. It also pushes an integer value of 1 to the corresponding queue. The parameters are: `broadcast chan &msg_type, intq &port_name`.

Figure 3.5c shows how the receive operation called on the port in the component automata is modeled. This pattern is reused for both request port and reply port receive calls. At first, the associated queue is dequeued and put into a variable `status`. From the `pre_recv` location, the model can either transition into the `post_recv` location if $status \geq 0$, implying that the queue was not empty, or go into a `blocking` state. Readers should note that the latter state is not a committed state, unlike the previous two. This indicates that a component can block at this state for any amount of time until a new item is pushed to the queue, indicated by the guard for outgoing edge. This is in line with the behavior implemented in the software. The code does have the option of setting a timeout on the blocking, which will be implemented in a future iteration.

- **Synchronous Request-Response:** This type of interaction involves one sender and one (or more) receivers. The communication proceeds in a lockstep, i.e., a send operation must be followed by a corresponding receive. This is achieved through the *request* and *reply* ports and the *send* and *receive* methods in RIAPS. For example, considering a sender component with a request port `req_port` and a receiver component with a reply port `rep_port`, the sequence should be: `req_port.send()`, `rep_port.receive()`, `req_port.send()`, `rep_port.receive()` and so on.

  The *request* port is not explicitly modeled as a template in UPPAAL. Instead, the *send* operation on a *request* port is directly implemented as a fragment of a template using a location corresponding to the line where the send operation is performed in the component source code and an edge leading to that location. Figure 3.5a shows the send operation, modeled as a synchronized edge leading to a committed location `post_send`. The transition produces the synchronization signal on the channel `req_port_channel` which is a placeholder for its corresponding channel. One such channel is

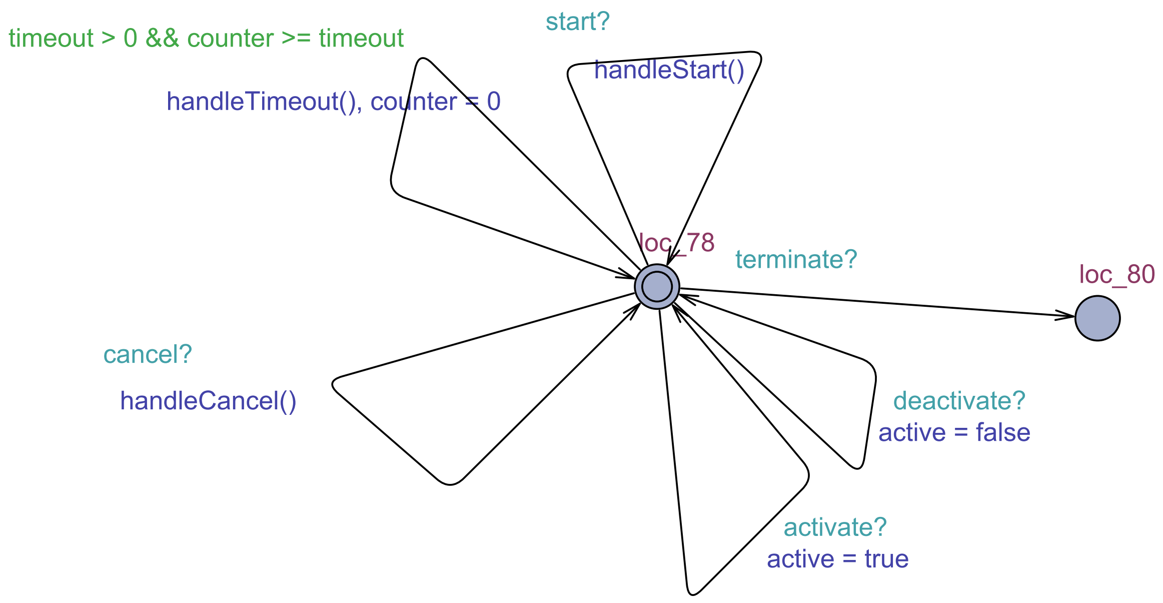


Figure 3.4: Timer port automata

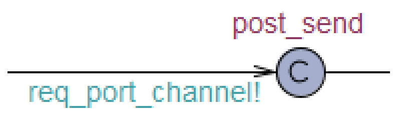
declared for each port belonging to every component instance.

Each reply port is modeled as a separate automata, as shown in Figure 3.5b. It consists of a single location with an edge synchronized on the same channel as the request port, which is passed as a parameter along with the port queue data structure. It also pushes an integer value of 1 to the corresponding queue. The channel should be declared as a broadcast channel to synchronize multiple port TA instances.

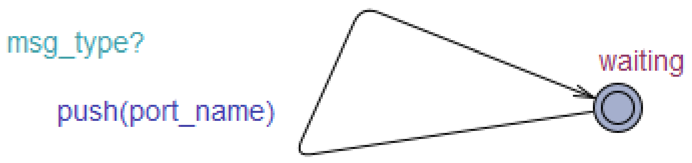
Figure 3.5c shows how the receive operation called on the port is modeled as another template fragment. This pattern is reused for both request port and reply port receive calls. At first, the associated queue is dequeued and put into a variable `status`. From the `pre_recv` location, the model can either transition into the `post_recv` location if $status \geq 0$, implying that the queue was not empty, or go into a `blocking` state. Readers should note that the latter state is not a committed state, unlike the previous two. This indicates that a component can block at this state for any amount of time until a new item is pushed to the queue, indicated by the guard for the outgoing edge. This is in line with the behavior implemented in the software. The code does have the option of setting a timeout on the blocking, which will be implemented in a future iteration.

- **Asynchronous Query-Answer:**

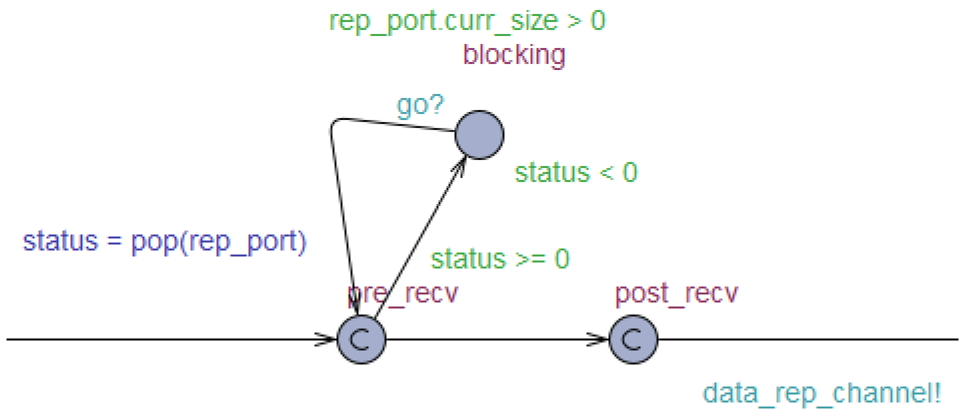
  This is similar to the synchronous type, but is less restrictive with respect to the sequence of the send and receive operations. Here, the sender can send multiple messages and the receiver can receive multiple messages in any order. This is achieved through the *query* and *answer* ports. However, the



(a) Fragment showing the send operation



(b) Reply port automata

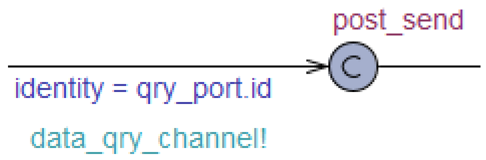


(c) Fragment showing the receive operation

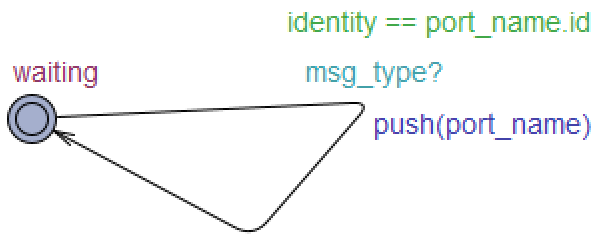
Figure 3.5: Synchronous Request-Reply

receiver must keep track of the source *identity* of each received message so that responses are routed to the appropriate destinations. The *identity* is sent as a part of the message by the query and answer ports. The modeling pattern is similar to the request-reply case with some modifications to handle the identity logic. Since in UPPAAL, it is not possible to pass values between automata processes, a global variable called `identity` is set to its `queue.id` whenever a *query* port sends a message. This value is read and stored in a local variable, `ans_port_identity`, when the *answer* port receives it, in order to set the global variable again for the outgoing response. The models are shown in Figure 3.6.

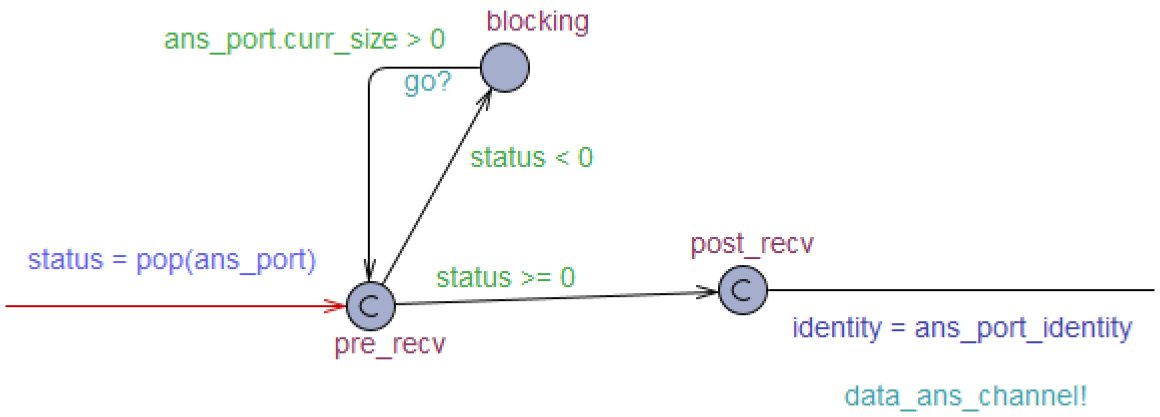
- **Asynchronous Publish-Subscribe:** This type of interaction consists of a publisher that continuously emits a stream if data with one or more subscribers receiving it. Often, subscribers can filter the incoming messages based on some *topics*. The *publish* and *subscribe* ports in RIAPS can be used to



(a) Fragment showing the send operation



(b) Answer port automata



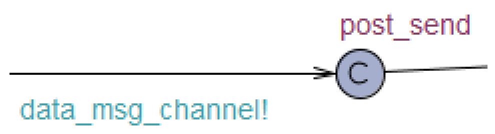
(c) Fragment showing the receive operation

Figure 3.6: Asynchronous Query-Answer

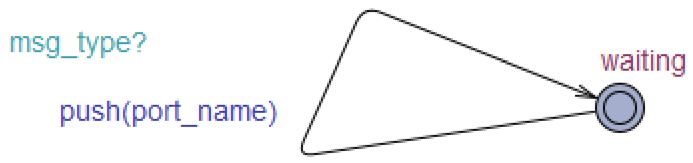
implement this type of communication. The corresponding timed automata representations are shown in Figure 3.7. It is very similar to the ones described previously, except that there is only one channel defined for one *pub-sub* connection.

3.2.3.3 Component Model

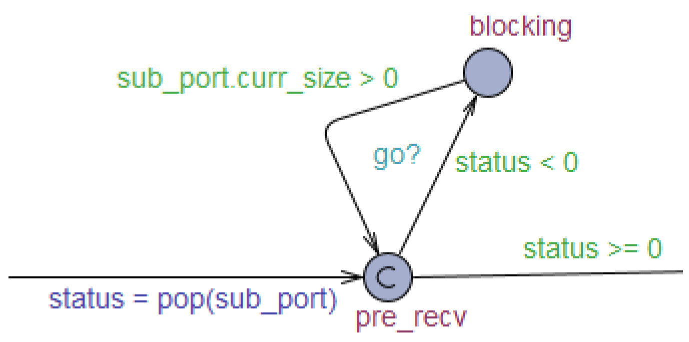
While the component structure is defined in the architecture model, the behavior is implemented as a Python class. It contains specific handler methods dedicated to each port, which are called when there is an event, such as an incoming message or a timer event. The framework also provides methods for receiving and sending messages on the network, start and stop timers, etc. However, it is up to the application developers to write code that implements the control logic. Thus, the TA model for each component is dynamically constructed with respect to its structure, based on the parsed model and the component implementation code.



(a) Fragment showing the send operation



(b) Subscribe port automata



(c) Fragment showing the receive operation

Figure 3.7: Asynchronous Publish-Subscribe

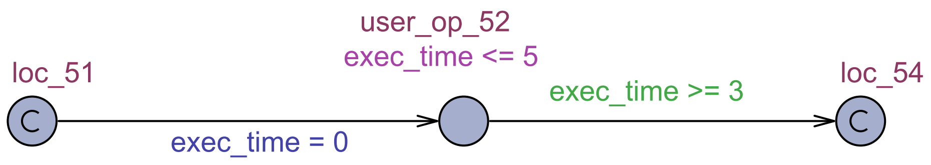
The various template fragments described above are used according to the behavior logic implemented in the source code to build the composite TA of the component. The approach followed is the following:

- *Abstract Syntax Tree (AST) Generation*: An AST [Python (2023)] is a tree-like representation of code written in any language that represents the actual structure of the code. It is abstract because it only preserves the structural constructs used in the code like the sequence of the statements using line numbers or offsets, expressions and their operands, if conditions, loops, function declarations, etc. without any details specific to the programming language itself. They are a fundamental part of the way a compiler parses the text input provided by programmers and generates machine instructions.

  The component code is parsed into its AST representation, using which a Control Flow Graph (CFG) like representation is generated. This also serves as an auxiliary visual representation of the code, which can be used as a design aid. The TA model is generated from the AST. Each defined port handler method is modeled as a new location with an incoming edge. The specific port-level method calls such as *receive*, *send* and *timer* commands follow the interaction type rules described previously.

```
class Comp_name(Component):
    ...
    def port_handler(self):
        ...
        port.receive(msg)    # line 51
        #ta: add time 3 5    # line 52
        control_action()     # line 53
        port.send(result)    # line 54
        ...
```

(a) Annotated code fragment



(b) Extension to component TA

Figure 3.8: Adding timing information to component TA

The location names are a combination of the operation they represent and the line number in the source code. Parameters for each component automata include the port `intq` instances and their associated *channel* variables.

- *Timing Information Integration*: The generated TA is extended with the timing information provided by the users in the form of specific annotations to the code. Timing information refers to a maximum and minimum time interval for executing a fragment of the code, typically for implementing some control algorithm. These annotations are in the form of specially formatted comments that are parsed using an XText [Dejanović et al. (2017)]-based DSL metamodel. This timing information is integrated into the model. An example is shown in Figure 3.8. As can be seen, the user adds the annotation before the code segment for which the timing information needs to be added. Consequently, during the component TA construction, a local clock `exec_time` is declared. A new location `user_op` is added to the position (using the line number) where the annotation is present. The clock variable is used to keep track of the time the automata spend in that location, and it is bounded by an invariant and a guard condition on the outgoing edge. Thus, the automata must stay in that location for at least the minimum time, but it cannot stay past the maximum time. This would translate to an inequality constraint in the symbolic execution and verification engine of UPPAAL.

3.2.3.4 Component Scheduler

The final element that is modeled is a scheduler for the components. This should not be confused with the task scheduler at the operating system level. The role of the scheduler in a message passing-based distributed

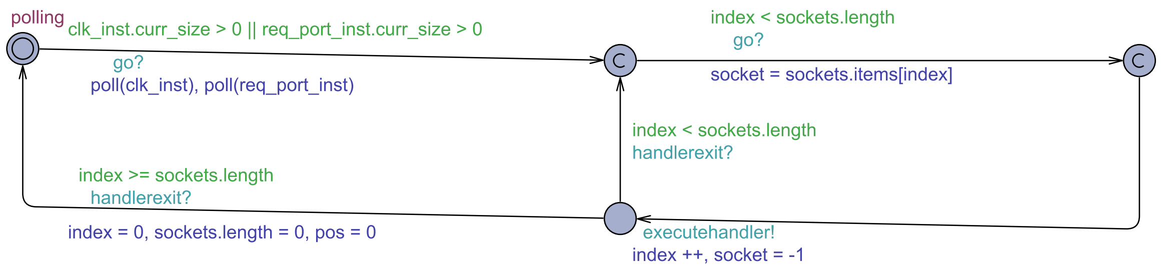

polling
clk_inst.curr_size > 0 || req_port_inst.curr_size > 0
go?
poll(clk_inst), poll(req_port_inst)
index < sockets.length
go?
socket = sockets.items[index]
index < sockets.length
handlerexit?
index >= sockets.length
handlerexit?
index = 0, sockets.length = 0, pos = 0
executehandler!
index ++, socket = -1

Figure 3.9: Component Scheduler Example

system is to regulate the processing of incoming messages. The port sockets are polled continuously, and those with any messages arriving are put into a Python dictionary. Different scheduling policies can be used to determine the order in which the associated port handler methods are invoked. For the *Batch Scheduler*, the dictionary containing the active sockets is scanned and the associated handler is invoked. In case of the *Round Robin Scheduler*, the sockets are scanned in the order they were defined in the component definition and the handlers are invoked in a round-robin fashion. There is also a *Priority Scheduler*, where the scanned sockets are put in a priority queue and read in their priority order. The scheduler automata are dynamically generated in a similar process to the component automata. However, they need to be customized for each component since UPPAAL does not support dynamically sized arrays (to pass the port queues for each component) as template parameters.

The batch scheduler automaton is shown in Figure 3.9. The initial location is `polling`. From there, it checks the port queues for any items added (using an *or* condition) and then calls the `poll` function on those ports. The `poll` function is a local function defined within the scheduler template that dequeues an item from the port queue and pushes an item into another custom data structure `socketlist`, consisting of an integer array called `items` and another integer variable `length` that stores the current size of the array.

The `items` array keeps a record of the scanned ports and the variable `index` is used to iterate over it. The value of each item in the array is the port ID, which is read and stored in a global variable `socket`. This value is used in the component automata to determine which port handler transition needs to be taken, synchronized using the `executehandler` channel. Once the component automaton transitions back to its initial location, the `handlerexit` synchronization channel is invoked, which resets all scheduler variables. The parameters for the model are the corresponding `socketlist` data structure instance as the variable `sockets`, the global integer variable `socket` and the queues of all the ports that are defined for the associated component of the scheduler.

```
app RelayMonitor {
    // message types
  message RelayStatus;
    // component definition
  component RelayManager(interval)
  scheduler priority;
  {
    timer countdown;
    sub RecvRelayStatus : RelayStatus timed;
  }
    ... // other components

    // actor definition
  actor RelayManager(interval){
    {
      manager: RelayManager(interval=interval);
    }
  }
    ... // other actors
}
```

(a) RIAPS Application model fragment describing the RelayManager component

```
app RelayMonitor{
  // actor deployed to host
  on (h2) RelayManager(interval=10) ;
    ... // other actor deployments
}
```

(b) A portion of the RIAPS Deployment Model

```
from riaps.run.comp import Component

class RelayManager(Component):

#constructor
    def __init__(self, interval):
        super(RelayManager, self).__init__()                    # line 13
        self.interval = float(interval)
        ...

# RecvRelayStatus subscribe port handler
    def on_RecvRelayStatus(self):                               # line 22
        value, timestamp = self.RecvRelayStatus.recv_pyobj()    # line 23
        self.countdown.cancel()                                 # line 24
        # reset count down
        self.setupTimer()                                       # line 26

#countdown timer port handler
    def on_countdown(self):                                     # line 45
        now = self.countdown.recv_pyobj()                       # line 46
        self.countdownStop = now
        self.valueHistory=(-1,now)
        # timeout!

    def setupTimer(self):                                       # line 55
        # set waiting period
        self.countdown.setDelay(self.interval)                  # line 57
        self.countdown.launch()                                 # line 60

    def handleActivate(self):                                   # line 62
        self.setupTimer()
```

(c) RelayManager component code fragment

Figure 3.10: RelayManager component information used for model generation

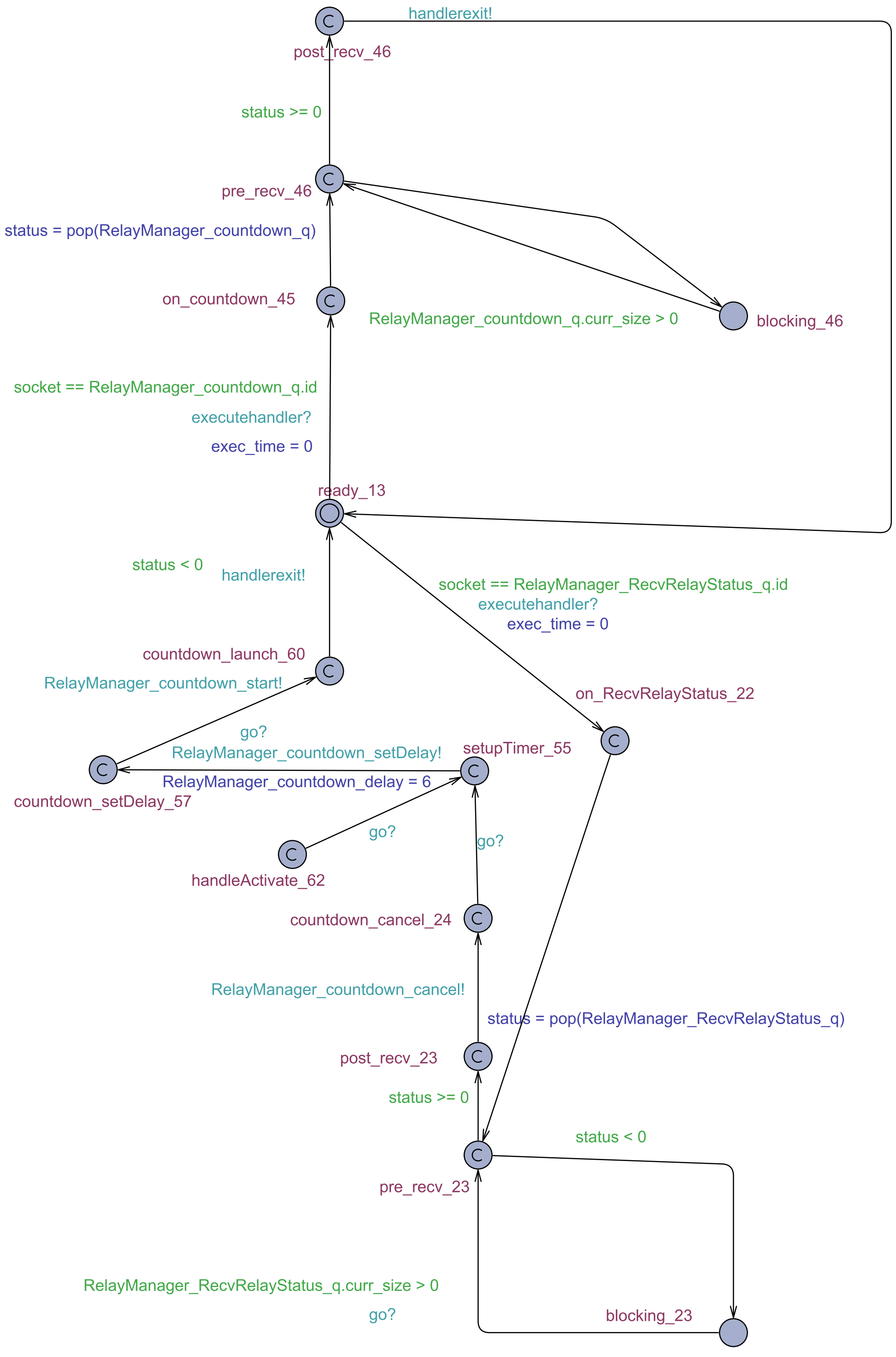

handlerexit!
post_recv_46
status >= 0
pre_recv_46
status = pop(RelayManager_countdown_q)
on_countdown_45
RelayManager_countdown_q.curr_size > 0
blocking_46
socket == RelayManager_countdown_q.id
executehandler?
exec_time = 0
ready_13
status < 0
handlerexit!
socket == RelayManager_RecvRelayStatus_q.id
executehandler?
exec_time = 0
countdown_launch_60
RelayManager_countdown_start!
on_RecvRelayStatus_22
go?
RelayManager_countdown_setDelay!
setupTimer_55
RelayManager_countdown_delay = 6
countdown_setDelay_57
go?
go?
handleActivate_62
countdown_cancel_24
RelayManager_countdown_cancel!
status = pop(RelayManager_RecvRelayStatus_q)
post_recv_23
status >= 0
status < 0
pre_recv_23
RelayManager_RecvRelayStatus_q.curr_size > 0
go?
blocking_23

Figure 3.11: Generated Component Automata

3.2.3.5 Concrete Example

Consider an example component *RelayManager* which is described in a RIAPS application model *RelayMonitor* as shown in Figure 3.10a. The various ports and message types associated with the component, as well as the actor definition that instantiates that component, are also defined in the model file. As shown, it contains a subscribe port *RecvRelayStatus* and a sporadic timer *countdown*. The corresponding deployment configuration is shown in Figure 3.10b. It shows that one instance of the *RelayManager actor* is deployed to *host h1*. The component code implemented in Python is shown in Figure 3.10c. It defines the handler methods for the two ports and the operations that must be performed when those methods are invoked. The basic behavior of the *RelayManager* component is that it receives a message from a publisher, likely the status of a relay that it controls. A relay is a device used in electrical distribution networks that acts as a switch to connect or disconnect a portion of the network. It is mainly used for protection or to reconfigure a part of the grid following a fault. Once the status message arrives, the *RelayManager* launches the timer *countdown* with a fixed delay. If the next message arrives before the delay, then it cancels the timer and restarts it. Otherwise, the timer handler is called at the end of the delay, indicating a timeout.

Using the information from these three sources, RIAPS2UPPAAL generates a TA model for the component as shown in Figure 3.11. It automatically declares the required variables, data structures, and channels from the provided information. As can be seen in Figure 3.11, it has an initial location set to `ready_13` with outgoing edges to two other locations `on_countdown_45` and `on_RecvRelayStatus_22` corresponding to the two port handlers, respectively. The various model patterns described previously are also used to compose parts of the component automata and then incorporated into the intermediate locations and edges. These include the sending and receiving operations at the different ports as well as the commands to launch, cancel, and set delay on the timer *countdown*.

3.3 Evaluation

The model generation was tested on a simple distributed application with a nontrivial architecture. In order to demonstrate the capabilities of the model-based technique for timing analysis, the same application was evaluated with two different deployment configurations. The purpose of doing so was to answer the question: *given a distributed application with defined actors, components, ports, and corresponding component behavior logic, further augmented with a set of timing specifications for each component, can the composite system satisfy a set of requirements related to input and output data stream equilibrium, nonblocking execution, timing, etc.*?

3.3.1 Application Description

The application chosen for the experiments is a distributed data aggregation application, let it be called *Distributed Estimator*. The logical structure of the application of the different components and their associated communication paths was as follows.

- *Sensor* acted as a data source. It periodically published a *SensorReady* message.
- *LocalEstimator* acted as an intermediate filter to collect sensor messages. It subscribed to the *Sensor* component and then retrieved the data by sending a *SensorQuery* request and receiving a *SensorValue* message type as the reply. It also published as *Estimate* message.
- *GlobalEstimator* acted as the final data aggregator that subscribed to *LocalEstimator* and then performed some estimation algorithm using it.

Two actors were defined, namely *Estimator* and *Aggregator*. The components belonging to each actor could be varied depending on the desired deployment configuration, as discussed in the next subsection.

3.3.2 Deployment Configurations

3.3.2.1 Configuration 1

Figure 3.12 shows the first deployment configuration that was investigated. In this setup, each instance of the *Sensor* component (called *sensor*) was colocated with an instance of the *LocalEstimator* component (called *filter*). This implied that each *sensor* was directly communicating with a dedicated *filter* component instance. All of them connected to one centralized *aggr*, which was an instance of the *GlobalEstimator* component.

The generated UPPAAL timed automata models for this configuration are shown in Figure 3.13. In total, 11 template models were generated, 3 for the components, 3 for the schedulers, 4 for the different types of ports present, and 1 for implementing the urgent transition. The final system consisted of 21 processes, which were instances of those 11 templates, based on the deployment configuration.

3.3.2.2 Configuration 2

Figure 3.14 shows an alternative deployment configuration. In this case the *filter* and *aggr* instances were colocated. Two sensor components, namely *Sensor1* and *Sensor2* were defined to differentiate communication between them with the *filter* using separate ports and message types. This implied that the *sensor1* and *sensor2* instances connected to a central *filter* instance which locally collated the data and sent them to the *aggr*.

Figure 3.15 shows the models generated corresponding to configuration 2. It contained 13 template definitions with 19 process instances required to describe the system.

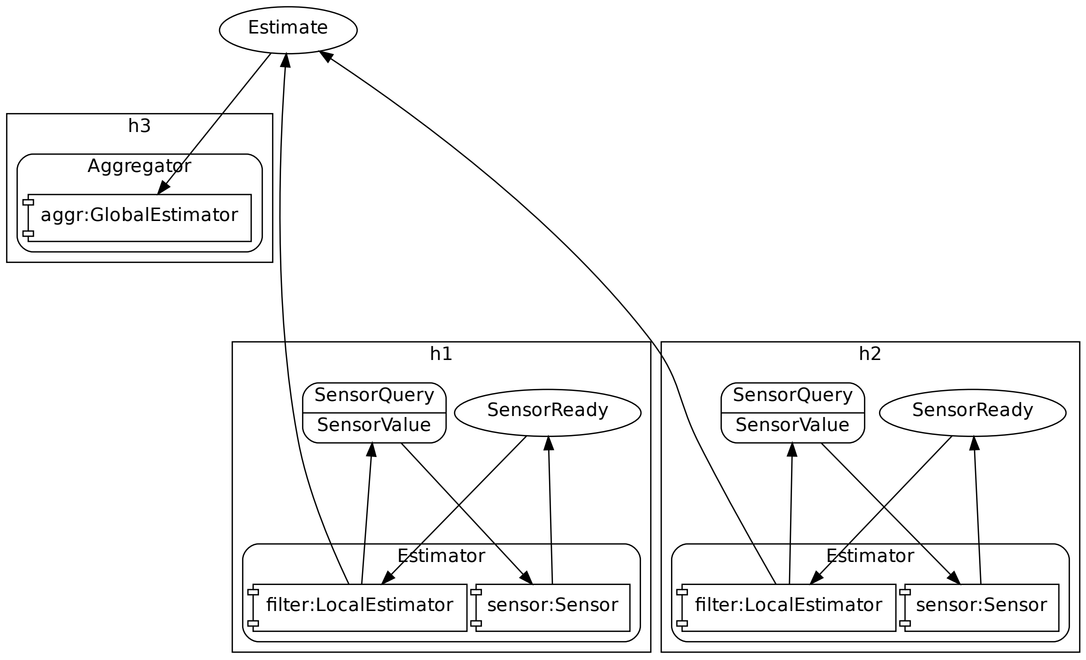


Figure 3.12: Deployment Configuration 1

3.3.3 Verification and Analysis

One of the main advantages of using a model checker such as UPPAAL is that it provides a verification engine that can formally validate specifications constructed by users in the form of verification queries. These specifications indicate the presence or absence of certain properties related to the state of the system or along some execution sequence. It automatically calculates over all possible interleavings of the states and transitions to verify the properties. This is very important for a distributed application since the number of such inter-leavings can become difficult to examine manually for even a fairly simple system.

The goal of the verification phase was to identify some specific patterns in the design and to analyze the temporal constraints imposed by composing several distributed components together. The queries were consistent with the level of abstraction used to generate the models, which in this case was the port operations and the timing information provided by the users. Four types of queries were used:

- *Blocking State*: This type of query was used to check whether the automata system never got stuck in a `blocking` location. The main cause of this error would be if a port tries to complete a *receive* operation when that port does not contain any unread messages. An example query for this category is `h1_Estimator_sensor._17 --> h1_Estimator_sensor.pre_recv_17`. Similar queries were used to check for all modeled processes.

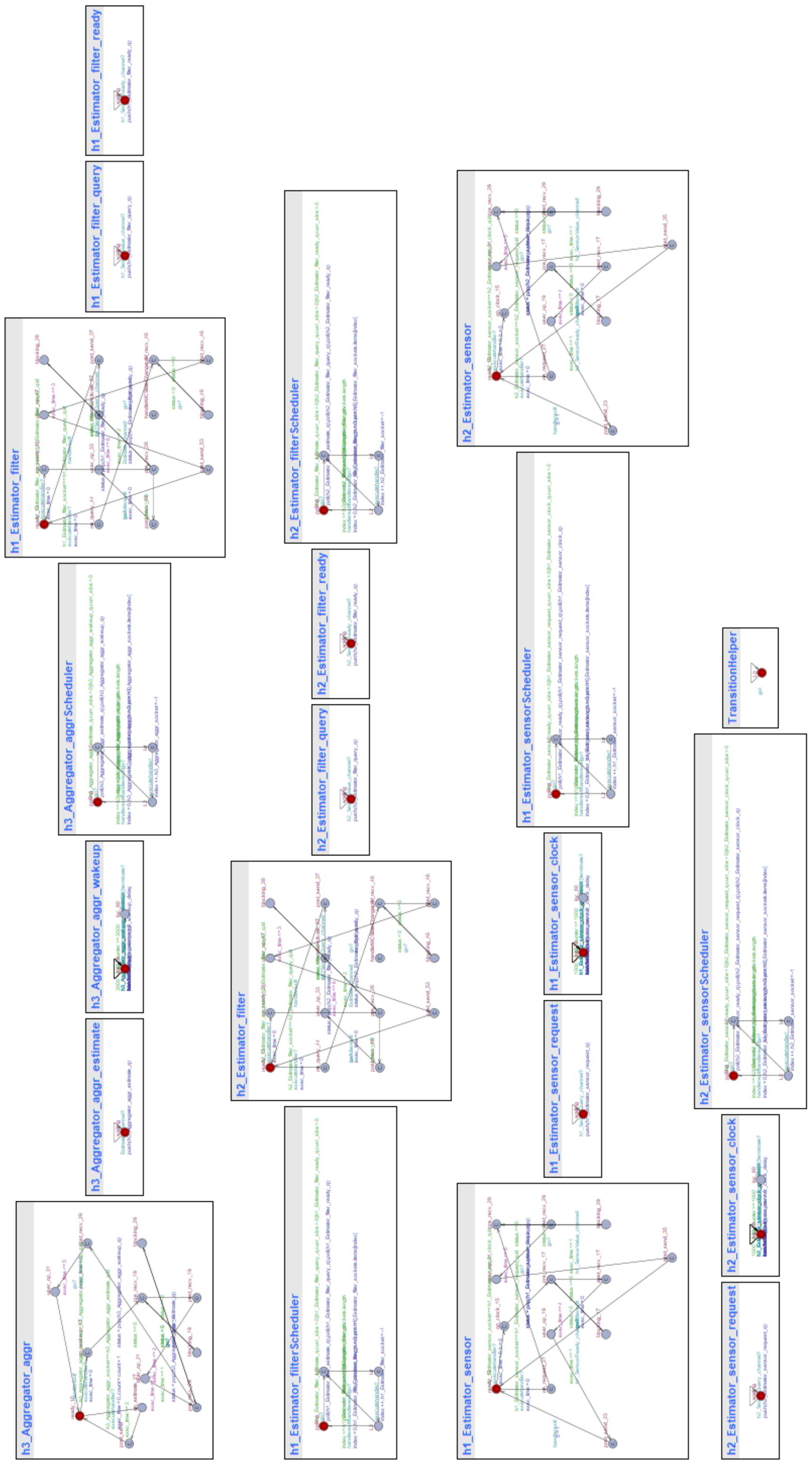


Figure 3.13: Screenshot of UPPAAL showing all TA models for Configuration 1

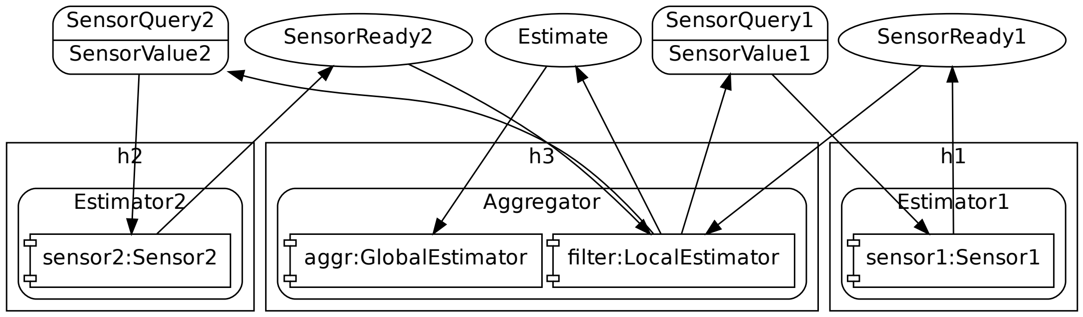


Figure 3.14: Deployment Configuration 2

- *Incoming Data Overflow*: This type of query was used to check whether any of the data streams were putting items into the port queues at a faster rate than they were being read by the receiving component. This error can occur if two components were communicating but at different rates. An example of this category is: `A[] h3_Aggregator_aggr_estimate_q.curr_size < 10`. Similar queries were used to check for other ports.

- *Time Boundedness*: These queries were a sanity check to validate the user timing information produced the correct bounds on the local clock `exec_time` for each of the annotated locations. They were specified as for example: `A[] h3_Aggregator_aggr.pre_recv_29 imply (h3_Aggregator_aggr.exec_time>=1 && h3_Aggregator_aggr.exec_time<= 2)`.

- *Stimulus-Response Delay*: The final category of queries tried to find a lower and upper bound on the global clock `global_time` between the beginning of an interaction cycle (stimulus) to its end (response) across all possible combinations of interleavings of events. A complete stimulus-response cycle was defined in case of this application as the interval between the first invocation of a *sensor* instance and the completion of the *aggr* instance. A local variable `count` was used to keep track of the completion cycle of each component automata. The query was of the form: `A[]`

```
(h3_Aggregator_aggr.on_estimate_27 && h3_Aggregator_aggr.count == 1
&& h1_Estimator_sensor.ready_10 && h1_Estimator_sensor.count == 1
&& h2_Estimator_sensor.ready_10 && h2_Estimator_sensor == 1
&& h1_Estimator_filter.ready_13 && h1_Estimator_filter.count == 1
```

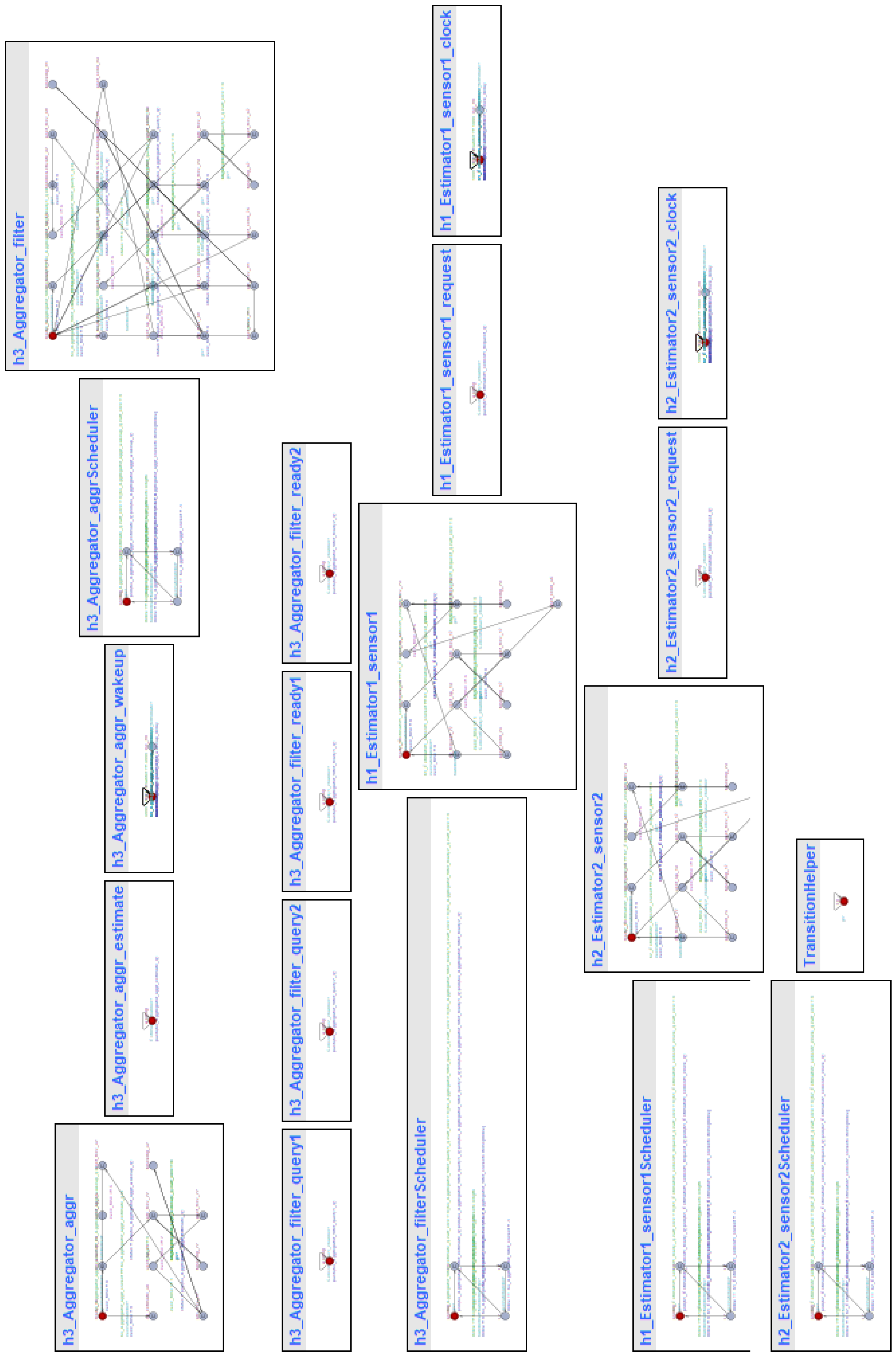

h3_Aggregator_filter
h3_Aggregator_aggrScheduler
h3_Aggregator_aggr_wakeup
h3_Aggregator_aggr_estimate
h3_Aggregator_aggr
h3_Aggregator_filter_ready2
h3_Aggregator_filter_ready1
h3_Aggregator_filter_query2
h3_Aggregator_filter_query1
h1_Estimator1_sensor1_clock
h1_Estimator1_sensor1_request
h1_Estimator1_sensor1
h3_Aggregator_filterScheduler
h2_Estimator2_sensor2_clock
h2_Estimator2_sensor2_request
h2_Estimator2_sensor2
h1_Estimator1_sensor1Scheduler
TransitionHelper
h2_Estimator2_sensor2Scheduler

Figure 3.15: Screenshot of UPPAAL showing all TA models for Configuration 2

Query	**Deployment 1**	**Deployment 2**
Blocking State	Satisfied	Satisfied
Incoming Data Overflow	Satisfied	Satisfied
Time Boundedness	Satisfied	Satisfied
Stimulus-Response Delay	$global\_time \geq 1006$	$global\_time \geq 1010$

Table 3.1: Verification Query Results

```
&& h2_Estimator_filter.ready_13 && h2_Estimator_filter.count == 1
&& global_time > 0 && global_time < 2000)
imply (global_time > 1000 && global_time < 1009)
```

Table 3.1 shows the results of the different verification queries performed in the UPPAAL model checker. The same four categories were tested on both types of deployment to capture a comparative analysis. As seen in the table, UPPPAL was able to verify that both deployments were free from blocking and overflows of the port queues. Thus, it pointed to the conclusion that in terms of these three criteria, both deployment configurations were valid and equivalent. However, when it came to the stimulus-response delay, a difference was observed in the output. The lower bound on the clock value was satisfied on both counts, but the lower limit for deployment configuration 1 was 4 units of time less than that of deployment configuration 2. The time unit scale used for translation to model time was *milliseconds*. Thus, it meant that considering all possible interleavings between the different components for both configurations, configuration 2 would take *at least* 4*ms* more to produce a response if all the intermediate transitions took the minimum time possible. It must be noted here that this was for one completed cycle, which meant if successive cycles were considered the gap would be bigger. Also, the time on the network transit times were not currently modeled within the UPPAAL models, which should also make the differences larger in the real system. UPPAAL was unable to verify an upper bound on the clock value for either system for the current model and the level of abstraction. This is one limitation of the current model that we will investigate in the future. However, for a system design question, considering that all components are running distributed on sufficiently fast hardware, it can be said that deployment 1 has a better stimulus-response delay performance than deployment 2.

3.3.4 Validation

The timing constraints produced by the generated TA models were validated against the actual measurements obtained by running the corresponding distributed application for both deployment configurations. The distributed application was run on beaglebone black embedded boards [Coley (2013)] installed with the RIAPS platform. The local clocks of the edge nodes were synchronized according to Precision Time Protocol (PTP) with the help of a special time synchronization service [Chapter 6] provided by RIAPS. The timestamps for

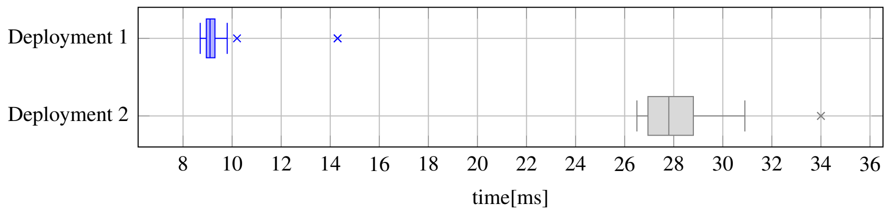


Figure 3.16: Stimulus-Response time measurements for both deployment configurations

	Average (ms)	Worst Case (ms)
Deployment 1	9.309	14.3
Deployment 2	28.162	34
Difference (ms)	18.853	19.7

Table 3.2: Average and worst case stimulus-response delay for deployment configuration1 and 2 respectively and their differences

the various events were recorded within the components and logged in a file. Figure 3.16 shows the distributions for the stimulus-response times for both deployments. As predicted by the models, the measurements showed that the times are worse for deployment 2. Table 3.2 shows the average and worst case values of the measured stimulus-to-response times for the two deployment configurations. It can be seen that both the average and worst case values for deployment configuration 2 were greater than that of deployment configuration 1. This was in agreement with what the models predicted. Individually, the average value for deployment configuration 1 was 9.309 ms, and the worst case was 14.3 ms, which satisfied the TA clock constraint $global\_time > 1006$ after discarding the sensor period of 1000 ms. Similarly, for deployment configuration 2, the average value of 18.853 ms and the worst case value of 34 ms were consistent with the clock constraint $global\_time > 1010$. However, considering the differences between the two types, the average difference of 18.734 ms was much higher than the minimum difference predicted by the model, which was 4 ms. This was because of the larger delays observed for both deployment 1 and especially deployment 2, compared to the lower limit computed by UPPAAL, most likely due to the network transit times and scheduling delays which were not used in the calculations in UPPAAL. This aspect deserves further investigation in the future. Thus, it can be concluded that, in general, the modeling approach was able to provide a sound estimate of the timing performance of the application consistent with its run-time measurements, although the calculations from the TA tended to be more conservative.

3.4 Summary

This chapter introduced a technique for automatically generating a TA representation of distributed CPS software components from the source code and integrating it with a commercially available software package UPPAAL. The approach followed was a combination of intermediate transformation using an augmented control flow graph TA and incorporating run-time information with the help of specialized annotated comments and a custom language parser.

The modeling framework was implemented on the RIAPS software platform to demonstrate the different interaction patterns that can be supported while still preserving a certain level of abstraction. The deployment information supplied in RIAPS was also integrated into the model generation process to create multiple instances of the component and port automata as specified in the deployment file. This was then used to design a simple distributed application and examine two deployment configurations using the UPPAAL query language. Four different query patterns were also demonstrated to analyze the design choices as well as the timing behavior with respect to stimulus-to-response times. The generated model was able to capture the required details of the application source code and was able to successfully run the verification queries. It showed how the deployment configuration can affect the timing performance due to the different ways in which the components are connected. Thus, it can be said that model-based analysis and model checking can serve as a great tool for identifying glaring design issues in distributed CPS in addition to being an effective way of formally validating its properties.

The model generation had some limitations in that it was not able to produce a bounded upper limit for the delay. Thus, future works would investigate ways to improve the model by using some additional constraints or variables to achieve this bounded condition. That would make it even more helpful and applicable for real-time analysis.

CHAPTER 4

Automated Deployment and Testing of Resilient, Distributed Cyber Physical Systems Application Components

4.1 Problem Statement

In a decentralized scenario, the application functions are modularized into distributed computing processes or *actors* which are then assigned to separate networked computing nodes. This process of installing and executing distributed processes on edge devices is termed *deployment*. How these application actors will be deployed on different remote nodes or hosts can greatly enhance its resilience and performance, in compatibility with the various fault tolerant algorithms and design patterns discussed previously. From a fault tolerance perspective, the desirability of a deployment is characterized by high level of redundancy and less overlap between the actors, while from an efficiency or budgetary perspective, it is about how well the available resources are shared and utilized. Thus, there exists a trade-off between these two degrees of freedom which makes it an interesting problem.

Consider an application with m such actors. In this case, the naive strategy would be a $1-to-1$ mapping where each actor is assigned to a node that requires a total of m nodes. This deployment represents the least risk of faults related to the exhaustion of hardware resources on the embedded nodes. It also provides a measure of resilience in case of noncritical node failures since there is no overlap among the actors. However, application developers must also adhere to some other constraints related to the total cost or the actual capabilities of the hardware nodes. So, the other extreme would be to assign m actors to a single node. This is not very interesting because then the system is not really distributed. But still theoretically, if this were done, the system would suffer from very low reliability and might overwhelm the computational resources on that one node, acting as a single point of failure in that case. Thus, a ”good” solution lies somewhere in between, obviously depending on the application hardware and software specifications. Usually, developers need to manually try different deployment configurations and then come up with the most suitable one, which can be quite tedious and time consuming.

Most state-of-the-art deployment algorithms are built for cloud-fog computing based applications that mainly look at achieving good load balancing while reducing latency to maintain CPU, RAM, and disk usage within safe bounds. The ones that consider cyber physical applications and requirements are mostly implemented as a run time monitoring tool [Ábrahám et al. (2016)] that requires its own overhead to be deployed along with the application itself. Alternatively, we propose a resilient deployment solver that takes the best of both worlds in terms of combining the resilience requirements and resource constraints. It can be used to automate the deployment process so that the application has the required amount of redundancy from the start. Thus, it can run with the required level of fault tolerance without needing any further adjustments.

After deciding on the best fault tolerance approach and coming up with the best deployment strategy to support it, application developers need to perform a comprehensive evaluation of their final design in a realistic environment before putting the system into production. Testing distributed applications is considerably more challenging, since in order to create a realistic simulation, the various processes would need to be executed on separate hosts and a separate communication network needs to be set up. For fault tolerance testing, testing should also include inducing different types of faults that affect different parts of the system at different times. This requires additional effort to set up and is a hassle for application developers who would ideally want to have a seamless transition from application development, deployment, and then testing.

We try to bridge the gap between generating a valid deployment and its run-time evaluation by introducing a testbed for prototyping distributed software using a virtual network emulator called *Minimum* [Lantz et al. (2010)]. Mininet allows one to create a network of virtual hosts in a single workstation or virtual machine with separate network interfaces for each host. We augment the network architecture with a behavior model implementation that can inject faults based on user specifications.

We integrate our solution to the deployment and testing problems into a decentralized software platform for distributed software development called *Resilient Information Architecture Platform for Smart Grids (RIAPS)* [Eisele et al. (2017a)], in order to leverage its inherent resilience features and ease of use. This leads to a seamless workflow starting from developing an application in RIAPS, followed by coming up with a deployment scheme for the application actors, and then finally testing the resilience of the given application and the deployment configuration subject to faults customizable by a behavior model.

The main contributions of this work are as follows:

- Development of a resilient deployment solver that takes a RIAPS application model and a set of deployment constraints and generates an allocation of application actors to nodes. We also improve the solver to serve as a design aid for developers by optimizing the solution based on achieving maximum redundancy and minimum cost.

- Creation of a fault tolerance testing environment for RIAPS applications using Mininet to run a virtual network and then using a user-defined behavior model to induce faults in the system.

- Experimental verification of the entire workflow using a practical example of an energy management system application that evaluates how through careful formulation of the deployment constraints and resource limits the resulting deployment can achieve the expected level of fault tolerance and reduce the resource load on the hosting devices.

- As a minor contribution, we also showcase the implementation of two fault tolerance patterns in the application using a RIAPS feature called *groups*. The groups are characterized based on their fault tolerance behavior, and they serve as a good use case for our test bed.

The given contributions are adapted and expanded from the publication Ghosh, P., Tu, H., Krentz, T., Karsai, G., and Lukic, S. (2022b). An automated deployment and testing framework for resilient distributed smart grid applications. In *2022 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*, pages 1–6. IEEE ©2022 IEEE.

4.2 Solution Approach

4.2.1 RIAPS Groups

The guiding principle in the RIAPS fault management architecture is that there is a clear separation between the application (which implements the application-specific functionality) and the framework (which provides and manages the resources needed by the applications) [Ghosh et al. (2019)]. The framework provides several out-of-the-box services for detecting and possibly mitigating anomalies, but it is ultimately the application's responsibility to react to faults and to take a corrective action. The RIAPS modeling library provides several tools at the application level that users can leverage to react to certain events and implement behavior that is appropriate for that particular application context. One such tool is the *groups* feature. In RIAPS, a group defines a virtual publish/subscribe type cluster among components over the actual physical network. RIAPS groups provide several functions:

- *Group communication*: Members of groups can send and receive private messages that are only circulated within that group.

- *Membership management*: Components can dynamically join or leave a group at any time. RIAPS also provides APIs for detecting and handling membership changes.

- *Leader election*: RIAPS groups can also perform leader election using a RAFT-based algorithm [Ongaro and Ousterhout (2014)]. Members can also communicate directly with the group leader using special methods.

- *Voting and consensus*: Group members also have the ability to start a voting process on values or actions and gather the final result based on the votes of all other members.

While the primary application of groups is in creating a hierarchical architecture within a system for multilevel coordination and control, we demonstrate here that some of those features can be leveraged to implement robust fault tolerance design patterns. We utilize two such patterns as described below:

1. *Redundancy groups*: It basically provides an automated failover mechanism. The behavior is the following: Elect a leader, then detect if the leader is out. If detected, then replace with new leader.

2. *Consensus groups*: It is a distributed agreement pattern on top of the leader election. Each member initiates a vote on a value. If the leader's vote is successful, then it uses its own value. Otherwise, it selects a group member with a verified vote to send their value.

4.2.2 Resilient Deployment Solver

The distributed application deployment problem can be defined as follows: *Given a RIAPS model of a distributed application, a set of nodes on which the application actors can be deployed, and a set of specifications based on the application software and resource needs, determine a deployment configuration that satisfies these specifications.* An additional criteria is to translate the computed solution into a RIAPS compatible deployment file that can be readily used to deploy and run the application.

Our developed deployment solver was implemented in Python and consists of three parts: a deployment specification language *dspec* using which users can define specifications that are then parsed using a *model parser*, a constraint optimization solver implemented using Z3 and finally a *deployment generator* that converts the computed solution into a RIAPS deployment file. It takes as input a RIAPS model file `app.riaps`, a deployment specification file `app.dspec`, a hardware configuration file `hardware-spec.conf` for resource limits and produces as output a RIAPS deployment file `app.depl`.

4.2.2.1 Constraint formulation

Before formalizing the constraints used in the solver, some basic definitions are required.

Definition 13 (Actor) *An actor is a tuple consisting of an identifier ID, redundancy D, host dependency H and a set of resource constraints R.* $A_j := \langle ID, D, H, R \rangle$.

Definition 14 (Node) *A node is a tuple consisting of an identifier ID and a set of resource constraints R.* $Node_i := \langle ID, R \rangle$

Definition 15 (Redundancy) *It is an integer that implies the number of copies of an actor that needs to be deployed.* $D(A_j) \in \mathbb{N}$

```
# Hardware Specifications
#Beaglebone Black
[bbb]
# no. of CPU cores
cores = 1
max_cpu = 0.7 # percentage
# Internal memory (MB)
mem = 512
max_mem = 0.7
# disk space (MB)
spc = 4096
max_spc = 0.7
```

Figure 4.1: hardware specification file. The name of the hardware device is used as the key. Thus, the same file can be used for deploying to a network of heterogeneous devices. ©2022 IEEE

Definition 16 (Host dependency) *The host dependency for an actor is a mapping between an actor and a node that the actor must be deployed to.* $H(A_j) = Node_i.ID$*. This captures the scenario of actors tied to a particular host, say, for example edge located devices that must be accessed from that particular location.*

Definition 17 (Colocation condition) *It is a set of actors that must be deployed to the same node.* $C = \{A_j | j \in \mathbb{N}\}$.

Definition 18 (Separation condition) *It is the opposite of colocation. It is a set of actors that can not be placed on the same node.* $S = \{A_j | j \in \mathbb{N} \wedge (A_j, A_k \in S \implies A_j, A_k \notin C. \forall i \neq j)\}$

Definition 19 (Resource limits) *These specify the worst case resource usage metrics defined for the actors or the hardware resource specifications in case of nodes. Four types of resources are considered, CPU utilization (expressed as a percentage over an interval), memory (expressed in MB), disk space (expressed in MB) and network bandwidth(two quantities are used, an average rate and a maximum ceiling, both expressed in kilobits per second (kbps)).* $R() \in \{cpu, interval, mem, spc, rate, ceil\}$.

The basic principle for deployment in terms of these resources is that *the total worst case or maximum resource requirements of the actors deployed to a particular node should not exceed the maximum capacity of that node for that resource (with some buffer).* The buffer is selected as 5%. Such resource requirements must be additive. The various resource limits for the hardware are specified using a configuration file under the name of the hardware device acting as the key. An example is shown in Figure 4.1. The network specifications for the node are taken from the RIAPS configuration file included with the RIAPS installation, which contains the values specified for the network interface card rate (*NIC_RATE*) and ceiling (*NIC_CEIL*). The worst-case resource usage for the various actors are specified in the RIAPS model under the individual actor definitions. An example is shown in Figure 4.2. The solver reads the values from these two locations and then solves the appropriate constraints (Equations 4.7-4.10) to generate the deployment configuration.

```
actor Aggregator(configfile, id, grptype) {
     uses {
      cpu max 35 % over 1;     // cpu limit over 1 second
      mem 200 mb;          // Mem limit
      space 2048 mb;         // File space limit
      net rate 40 kbps ceil 60 kbps; // Net limits
    }
   ...
 }
```

Figure 4.2: Resource limits placed on the actor can be defined in the RIAPS model under the `uses` block. ©2022 IEEE

The final deployment solution space is encoded as a 2D deployment matrix where each element is a binary variable, X defined as Equation 4.1.

$$X_{i,j} = \begin{cases} 1, & \text{if } A_j \text{ assigned to } Node_i. \\ 0, & \text{otherwise.} \end{cases} \tag{4.1}$$

Based on these definitions, the different constraints used by the solver can be formulated as:

Redundancy

$$\forall j(\sum_i X_{i,j} = D(A_j)) \tag{4.2}$$

Host-actor dependency

$$\forall j(Node_i \in H(A_j)) \implies (\forall p \neq i,\, X_{p,j} = 0) \tag{4.3}$$

Colocation constraints

$$\forall A_m, A_n \in C, (X_{i,m} = 1) \implies (X_{i,n} = 1) \tag{4.4}$$

Separation constraints

$$\forall A_m, A_n \in S, (X_{i,m} = 1) \implies (X_{i,n} = 0) \tag{4.5}$$

Separation constraints

$$\forall A_m, A_n \in S, (X_{i,m} = 1) \implies (X_{i,n} = 0) \tag{4.6}$$

CPU

$$\begin{gathered} \forall i \sum_j (X_{i,j} \times cpu(A_j))/100 * interval \\ < max_c pu(Node_i)/100 * cores * interval \end{gathered} \tag{4.7}$$

Memory

$$\forall i \sum_{j} (X_{i,j} \times mem(A_j)) < mem(Node_i) \tag{4.8}$$

Disk space

$$\forall i \sum_{j} (X_{i,j} \times spc(A_j)) < spc(Node_i) \tag{4.9}$$

Network bandwidth

$$\begin{aligned} \forall i \sum_{j} (X_{i,j} \times rate(A_j)) + Max(ceil_j - rate j) \\ < 0.95 * NIC\_RATE(Node_i) \end{aligned} \tag{4.10}$$

4.3 Evaluation

4.3.1 An Example Application

In order to show the effectiveness of our deployment solver and the resilience testing setup, we will be using the example of an intelligent distributed energy management application for a microgrid. The U.S. Department of Energy defines the microgrid as 'a group of interconnected loads and distributed energy resources within clearly defined electrical boundaries that acts as a single controllable entity with respect to the grid. A microgrid can connect and disconnect from the grid to enable it to operate in both grid-connected or islanded mode' [Ton and Smith (2012)]. The decentralized open nature of a microgrid along with the growing popularity of new players within the grid, such as electric vehicles (EVs) and smart buildings, have led to newer challenges in managing and maintaining energy supply to critical loads in both grid connected and islanded modes.

The energy management application using RIAPS (or REMApp) consists of a building load, an EV charging station, and a Battery Energy Storage System (BESS). It was developed in collaboration with North Carolina State University. The microgrid has a centralized *aggregator* which connects to the main utility and regulates the energy supply to the loads according to the demands of each type of load. In RIAPS parlance, these entities can be considered as the actors of the application. The basic functionalities of each actor are described in Table 4.1. The dispatch problem is formulated as an optimization problem described in Equation 4.11. Here x_{ij} denotes the power granted to load i for time interval j, K is the prediction horizon, $Pgrid_j$ denotes the total power available from the grid in time interval j and $P_{req_{ij}}$ denotes the power required for unit i in time interval j. w_i is a weight assigned to each load that signifies its priority. Thus, the *Coordinator* component within the *Aggregator* actor functions as a model predictive controller. Both the *Building* and *Charger* actors consist of a predictive component, a manager component, and an interface device component. The predictive component incorporates a deep neural network-based forecasting algorithm that is trained us-

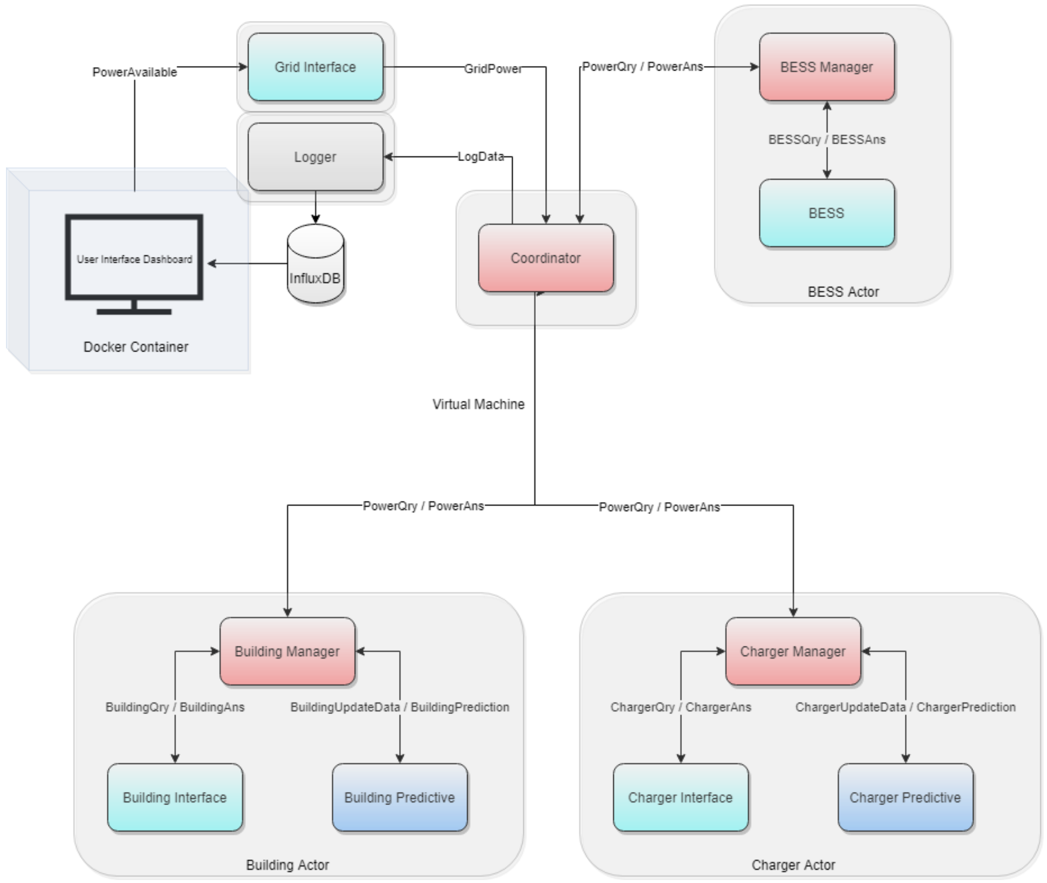


Figure 4.3: RIAPS Energy Management Application. The translucent rectangles represent actors and the solid rectangles represent components. The arrows indicate messages exchanged. This is the logical architecture prior to the application of deployment features.

ing pre-recorded consumption data. We simulate building and charger loads by using actual measurement data from a microgrid. Since we use simulated loads, the aggregator essentially works as an open-loop controller where the set point commands are not actuated. We omit the implementation details of the REMApp application and the control algorithms, as it is not within the scope of this dissertation.

$$\begin{aligned}
\min_{x_{ij}} \sum_{i=1}^{N} \sum_{j=1}^{K} w_i \times 0.5 \times (P_{req_{ij}} - x_{ij})^2 \\
\text{subject to:} \\
\sum_{i=1}^{N} x_{ij} \leq P_{grid_j}\ \forall j \\
x_{ij} \geq 0 \\
x_{ij} \leq P_{req_{ij}}
\end{aligned} \tag{4.11}$$

Actors	**Functions**
Building and Charger actors	**1.** Get power consumption readings from sensors **2.** Estimate power demands for a future time horizon (e.g. a day or 12 hours) **3.** Send power request to the Coordinator, receive setpoint commands from the Coordinator
Grid actor	**1.** In grid-connected mode, send information to the Coordinator about the power available from the main grid for the time horizon
BESS actor	**1.** Act as an energy buffer when the microgrid operates in islanded mode or if the power demand can not be met by the grid **2.** Receive charging/ discharging commands from the Coordinator
Aggregator	**1.** Receive power requests from the load actors **2.** Produce an optimal allocation strategy for each future time step based on the power available **3.** Based on the BESS state of charge, make decisions about discharging the battery to supply the loads or charge it if excess power is available for a particular time step **4.** Provide allocated power to the loads

Table 4.1: Actors and their functions in REMApp ©2022 IEEE

In addition to these actors, we also implement a *Logger* component to log measurement and event-related data and also run a remote dashboard for visualizing the results. The entire application architecture is shown in Figure 4.3.

Considering the REMApp application, its principal performance and safety specifications can be described as:

1. *The system should not dispatch more power than is available at the current time step.*
2. *The power allocated to the loads should be close to the predicted demand, provided the first condition is not violated.*

For the final application design, we consider the following two types of threats for the application:

1. *Node failure or isolation due to network partition*: Both of these scenarios would lead to one or more of the nodes being isolated from the rest of the network, in the process becoming a nonparticipant.
2. A *False Data Injection Attack (FDIA)* occurring at any of the sensors in the building and the EV charger. This could lead to the Coordinator calculating an erroneous solution and allocating too much or too little power.

We applied the redundancy groups pattern to the coordinator component and the consensus groups pattern to the building, EV charger, and BESS manager components.

```
app REMApp {
  Aggregator copies 2;
  colocate (UtilityGrid,DataLogger);
  separate (BESSActor, BuildingActor, ChargerActor);
  deploy (UtilityGrid) on (h1);
  // use limits for bbb on all;
  }
```

Figure 4.4: REMApp.dspec file. The last line is used for resource constraints. ©2022 IEEE

4.3.2 Solver Optimization Modes

We applied the solver to the REMApp model to validate its correctness as well as to evaluate its performance in terms of scalability and timing. An intuitive deployment specification language *dspec*, was used to specify the constraints for the experiments. An example *dspec* file used for the experiments is shown in Figure 4.4. The solver produces a solution that satisfies all the given criteria. Additionally, it also has two optimization modes that use two different objective functions in order to produce the solution that fits the desired objective.

- *Basic Solver*: We use the specifications in Figure 4.4 to run the solver for 4, 8, and 12 nodes, respectively. The solution for 12 nodes is shown in Table 4.2. The respective solution times in seconds were 0.55 s, 0.62 s, and 0.66 s.
- *Maximum Redundancy*: By choosing this optimization mode, it is possible to search for a solution that maximizes the copies of actors given a number of nodes. This mode provides the maximum redundancy for a given node distribution. The solution for the 12 node configuration for maximum redundancy produces the result shown in Table 4.3. It can be clearly seen that this solution shows that up to 12 copies of the aggregator, 3 copies of the BESS and building actors, and 6 copies of the charger actor can be deployed to 12 nodes. Since the logger and grid actors are colocated and tied to one host, there can only be one copy of them.
- **Minimum Cost**: The solver can also be configured to solve for a deployment configuration that uses the minimum number of nodes for the given constraints. This can serve as a useful design aid for system architects looking to plan the resource budget for a given application with set resilience specifications.

This setting was used to generate the deployment for our test run by modifying the specifications of Figure 4.4 as follows:

- Include 3 copies of the building, charger and BESS actors to tolerate a single actor failure due to FDIA attack as per the $2n+1$ redundancy rule [Castro et al. (1999)].

	Aggregator	BESSActor	BuildingActor	ChargerActor	DataLogger	UtilityGrid
h1	0	0	0	0	1	1
h2	0	0	0	0	0	0
h3	0	0	0	0	0	0
h4	0	0	0	0	0	0
h5	1	1	0	0	0	0
h6	0	0	0	0	0	0
h7	0	0	0	1	0	0
h8	1	0	0	0	0	0
h9	0	0	0	0	0	0
h10	0	0	1	0	0	0
h11	0	0	0	0	0	0
h12	0	0	0	0	0	0

Table 4.2: Solver solution for 12 nodes with the basic solver

	Aggregator	BESSActor	BuildingActor	ChargerActor	DataLogger	UtilityGrid
h1	1	0	1	0	1	1
h2	1	0	0	1	0	0
h3	1	0	0	1	0	0
h4	1	0	0	1	0	0
h5	1	0	1	0	0	0
h6	1	0	0	1	0	0
h7	1	1	0	0	0	0
h8	1	1	0	0	0	0
h9	1	1	0	0	0	0
h10	1	0	0	1	0	0
h11	1	0	1	0	0	0
h12	1	0	0	1	0	0

Table 4.3: Solver solution for 12 nodes with maximum redundancy

- Separate the *Charger* actor and the *Aggregator* actor. This was done purely for experimental purposes to demonstrate the effects of the faults on the Aggregator and the Charger independently.

The solution indicates a minimum of 9 nodes to deploy the application, as shown in Table 4.4. An example RIAPS deployment file produced after running the solver is shown in Figure 4.5.

4.3.3 Scalability Experiments

The solver was tested on a number of problems corresponding to a different number of candidate nodes for deployment, as well as increasing complexity in terms of the number of constraints added to the solver. The REMApp model was used here as well to calculate the deployments. The basic solver, the maximum redundancy solver and the minimum cost solver were used to compute the solution for all scenarios, and the total computation time was measured.

Figure 4.6 shows the variation of the computation times with the number of nodes for 10, 20, 30, 50 and

	Aggregator	BESSActor	BuildingActor	ChargerActor	DataLogger	UtilityGrid
h1	0	0	1	0	1	1
h2	1	0	1	0	0	0
h3	0	1	0	0	0	0
h4	0	1	0	0	0	0
h5	0	0	0	1	0	0
h6	0	1	0	0	0	0
h7	0	0	0	1	0	0
h8	1	0	1	0	0	0
h9	0	0	0	1	0	0

Table 4.4: Solver solution for the fault tolerance testing configuration ©2022 IEEE

```
app REMApp {
    on (h1) BuildingActor() ;
    on (h1) DataLogger() ;
    on (h1) UtilityGrid() ;
    on (h2) Aggregator() ;
    on (h2) BuildingActor() ;
    on (h3) BESSActor() ;
    on (h4) BESSActor() ;
    on (h5) ChargerActor() ;
    on (h6) BESSActor() ;
    on (h7) ChargerActor() ;
    on (h8) Aggregator) ;
    on (h8) BuildingActor() ;
    on (h9) ChargerActor() ;
    }
```

Figure 4.5: REMApp.depl deployment file automatically generated by running the solver. This can be used readily with RIAPS to deploy the application. ©2022 IEEE

100 nodes, respectively. It can be seen that the computation time remains below 5 seconds for almost all observations. The few outliers (shown as black diamonds) that are observed occurred when the maximum redundancy solver was used. This was because solving for maximum redundancy required the evaluation of the node-actor assignments for every node regardless of whether the numbers specified in the constraints were already satisfied or not. In spite of that, the timings are still acceptable considering a design-time operation.

Figure 4.7 shows how the solver computation time varies with the number of solver assertions. From the figure, it can be ascertained that the relationship between the two is close to linear. It can also be seen that the average solution time for up to 1600 constraints stays below 9 seconds.

Comparing these values with state-of-the-art approaches, among other work using constraint solvers, Pradhan et al. (2018) calculates the solution time with the number of failure events and reports an average of 1.74 seconds and a maximum of 48 seconds. This is comparable to the results obtained by our solver. The Reinforcement Learning (RL) based optimal placement algorithm in Rossi et al. (2020) reports the fastest average computation time of 59*ms*, with a worst-case time of 1.4*s*. However, their solution only considers minimizing latency and CPU utilization as their objectives and does not mention the time taken to train the RL

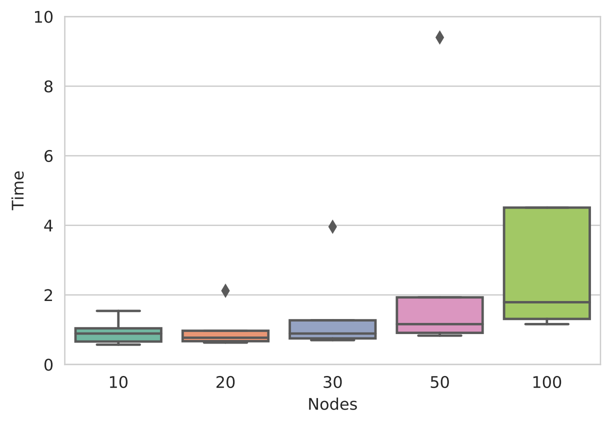


Figure 4.6: Plot of the solution time vs the number of nodes

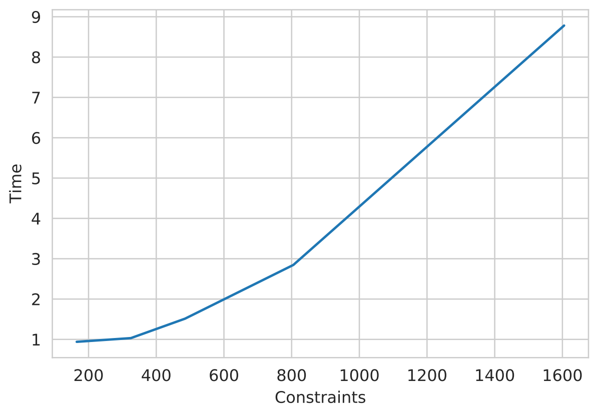


Figure 4.7: Plot of the solution time vs the number of constraints

model. Still, this points to the potential of exploring learning-based solution techniques for CPS deployment use cases as well.

4.3.4 Fault Tolerance Testing

We implemented a RIAPS compatible testing framework design for users to easily test their application using two ingredients: a virtual network emulator Mininet [Lantz et al. (2010)] and a behavior model specification to introduce faults into the network. Mininet allows for rapid prototyping of large networks on a single computer. Due to the use of OS level virtualization features, including processes and network namespaces, Mininet hosts appear the same as actual remote nodes to RIAPS and thus can be used without modifying the code. Mininet also provides a Command Line Interface (CLI) that can be used to control link parameters, bring links up or down, and send commands to hosts.

```
py 'starting test'
py time.sleep(1200)
py 'link failure'
py time.time()
link s1 h2 down
py time.sleep(1200)
py 'starting attack'
h7 xterm -T h7 -e python3 Attacker.py
py time.sleep(300)
py 'end of test'
py 'end'
```

Figure 4.8: Behavior model script showing the mininet cli commands to take down/ bring up links as well as initiate the attacker Python client process.

In order to simulate faults in the system, we created a behavior model script. We also implemented an attacker RIAPS device component that can be triggered remotely through a Python client using a host and port address. This is an example of a malicious component that has been compromised. This attacker can be imported into any RIAPS model by simply including it in the model file and can be interacted with using its ports. The attacker sends garbage messages to one of the application actors once it is triggered. The client uses ZeroMQ [Hintjens (2013)] to communicate with the attacker. The behavior model script uses the Mininet CLI commands and the Python client to introduce faults at specific intervals. Therefore, the testing process only involves launching Mininet, deploying the RIAPS application, and running the behavior model script. The general process flow is as follows: First, the deployment configuration is obtained by running the solver. Second, a behavior model script is written consisting of Mininet CLI commands to introduce faults at specific times. Third, a mininet virtual network of the required size is created on the PC, and RIAPS is started. Fourth, the application is deployed and launched using the RIAPS GUI. Finally, the behavior model script is sourced from the Mininet CLI. An example behavior model script, used for the experiments in Section 4.3.4.1, is shown in Figure 4.8.

4.3.4.1 Experimental Results

Given the two types of faults introduced in Section 4.3.1 and the 9 node deployment configuration obtained in Section 4.2.2, we designed a 60 minute resilience test. The behavior model used was the following:

1. At t_0, start the test.
2. At $t_0 + 20$, wake up, bring down host h2.
3. At $t_0 + 40$, launch the Python client to start the FDIA attack on host h7.

4. At $t_0 + 55$ stop the attack (configurable parameter within the client).

5. At $t_0 + 60$, end of test.

The link was brought down to test the redundancy groups feature, which was implemented in the Coordinator component, while the attack was initiated to target the EV charger on which the consensus group pattern was implemented. Three 60 minute runs of the experiment were performed and the data were recorded.

The important observations from the experiments were:

1. For two of the three runs, the first fault targeted a leader node (leader election is dynamic). This triggered a new leader election. The average time interval between a link failure and the election of a new leader was about 1 s.

2. The average time interval between the beginning of a round of voting and the obtaining of its result was 9ms before and 19ms after the FDIA.

3. Both of these times are negligible compared to the controller time step of 15 min real time that was used in the application.

Figure 4.9 shows how the system performed during run 3 of the experiment. The white vertical line indicates the current simulation time. i.e. the simulation time when the test was ended. The right-hand side of that line indicates the future time horizon. Validating the plots in light of the performance criteria discussed in section 4.3.1, it can be observed that:

1. *Condition 1* was satisfied. This can be seen from the left half of the graphs, which indicate the prefault period when both the power dispatched to the EV charger and the building was lesser than the predicted or requested power. This was because the available power was not adequate, even after discharging the battery, as seen in the state-of-charge (SoC) plot. This condition was also satisfied during the latter stages or post-fault period where the building dispatched power was capped to 1000 KW, while the chargers encountered almost total load curtailment. This was due to the fact that the battery SoC was down to its minimum specification of 0.2 and therefore unable to discharge.

2. *Condition 2* was also satisfied as observed in the graphs that the allocated power closely followed the predicted power and only deviated in the event that the available power was not sufficient.

Thus, it can be said that the application functionality was resilient to both faults.

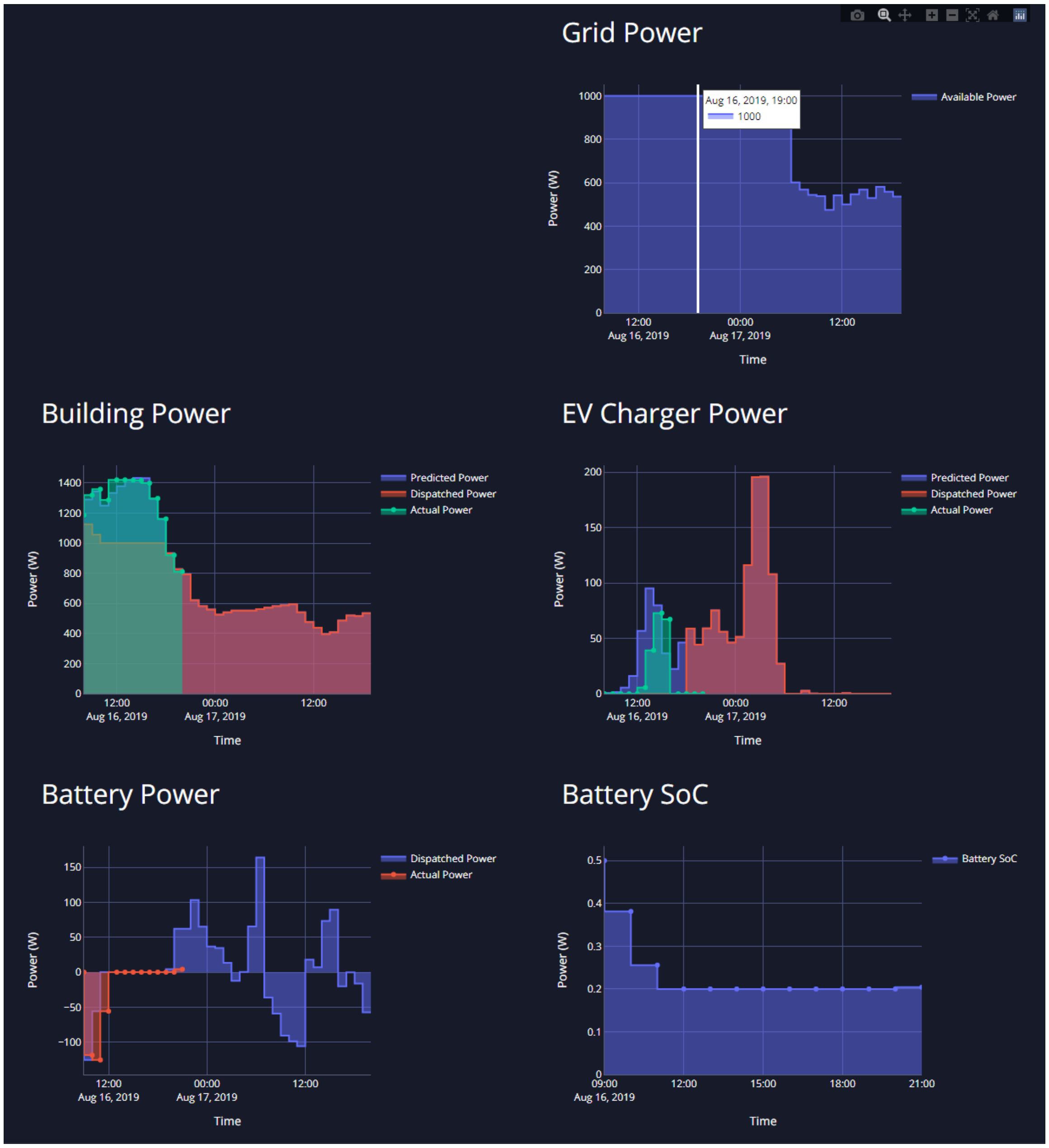


Figure 4.9: Snapshot of run 3 showing the energy profiles of the different loads and the power from the grid. For the loads, The blue curve represents the predicted power, the red curve represents the power dispatched by the coordinator. The green line is the simulated value obtained from the captured data. It is used as a seed for the prediction and can be ignored for our analysis. ©2022 IEEE

4.3.4.2 Deployment with Network Bandwidth Constraints

This section demonstrates how the solver evaluates the deployment configuration taking into account the resource-limiting constraints for the hardware devices and resource requirements for the distributed application actors. Namely, the network bandwidth constraints are used to set up the solver and the REMApp RIAPS

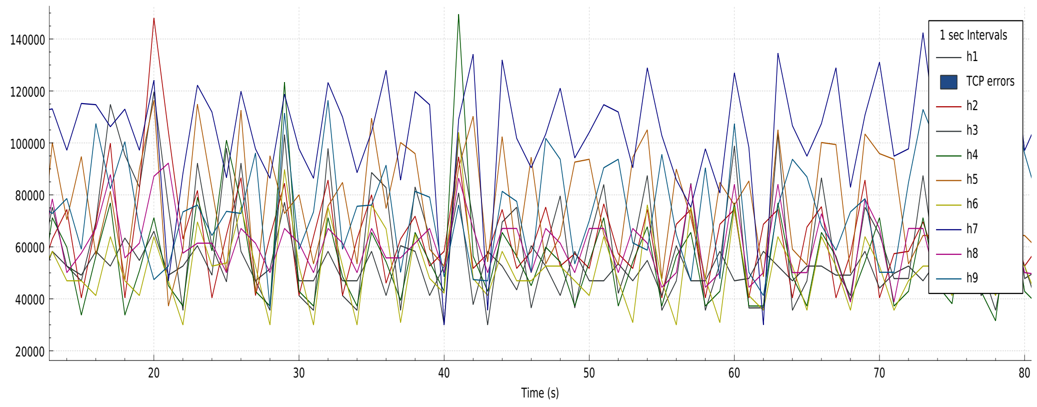
	Aggregator	BESSActor	BuildingActor	ChargerActor	DataLogger	UtilityGrid
h1	0	0	0	0	1	1
h2	1	0	0	0	0	0
h3	0	0	0	1	0	0
h4	0	0	1	0	0	0
h5	0	0	0	1	0	0
h6	1	0	0	0	0	0
h7	0	1	0	1	0	0
h8	0	1	0	0	0	0
h9	0	0	1	0	0	0
h10	0	0	0	1	0	0
h11	0	1	0	0	0	0
h12	0	0	1	0	0	0

Table 4.5: Solver solution for 12 nodes with network limits ©2022 IEEE

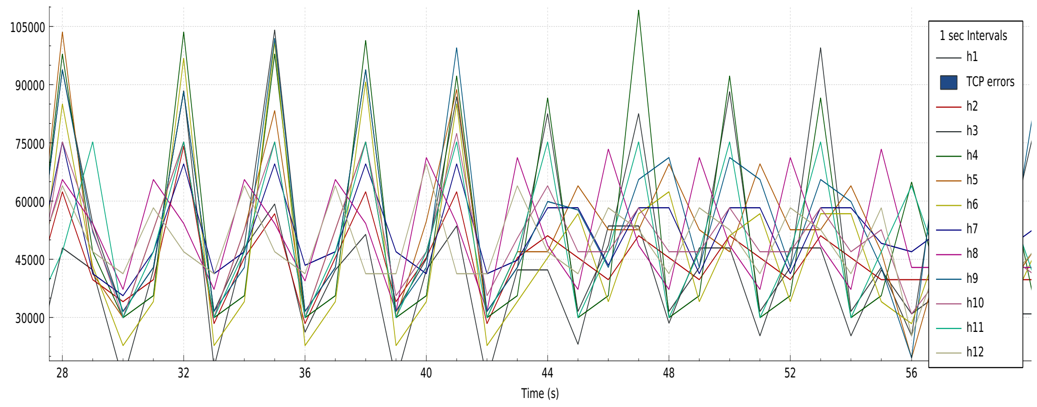
model and conduct the experiments. The reason for choosing them is due to the fact that the network traffic can be easily captured and isolated for each virtual network interface tied to each virtual host while running the application on our mininet testbed.

The hardware specifications chosen to run the experiment were *NIC RATE* as 118 *kbps* and *NIC CEIL* as 131 *kbps*. The network limits which represent the worst-case requirements for the actors of the REMApp RIAPS application were specified as $rate = 40kbps$ and $ceil = 60kbps$ for each actor. These values were chosen in this experiment mainly for demonstration purposes. In a production deployment, typically, each actor will have different values which need to be calculated by prior bench-marking or building faithful models and simulation. The solver was run using the same *dspec* file described in Figure 4.4 but with the last line uncommented, signaling the solver to calculate resource constraints in this case. The solution obtained is shown in Table 4.5. Compared to the solution obtained earlier in Table 4.4 without using the limits, it can be seen that using the resource limits resulted in the usage of 3 additional nodes to create a deployment configuration of 12 nodes.

Figure 4.10 shows the *tcp* network traffic for each host (node) while running the RIAPS Energy Management Application on the testbed captured over 1 second intervals. Figure 4.10a corresponds to the deployment configuration without resource limits (Table 4.4) while Figure 4.10b corresponds to the configuration including resource limits (Table 4.5). The tcp packets include the application messages sent via RIAPS along with some overhead. From the graphs, it can be seen that the average bandwidth consumed for the hosts is $84k$ bits per second in Figure 4.10a and $71k$ bits per second in Figure 4.10b, which points to a reduction of 15%. Looking at the maximum bandwidth consumed, it can be seen that for the deployment configuration considering network limits, it stayed almost entirely within $100k$ bits for all hosts with a few overshoots. It was still within the NIC_RATE of $118kbps$ set for the calculations. On the other hand, the bandwidth consumed for



(a) Network traffic for deployment without using resource constraints



(b) Network traffic for deployment after using resource constraints

Figure 4.10: Fault-tolerance protocol for the scenario when a member of the recv_grp foes offline ©2022 IEEE

host *h1* regularly exceeded 120$kbps$ in the deployment configuration without using the network limits.

4.4 Summary

This work introduced two design time tools for prototyping and deploying resilient distributed applications for CPS such as the Smart Grid. The first is a resilient deployment solver that maps distributed application entities or actors to remote nodes while adhering to certain design and resilience specifications, thus going beyond traditional solutions in the distributed cloud/fog systems literature. The solver can also be used to help system architects optimize either the overall system redundancy or the setup cost by setting the mode of the solver appropriately. The second is a virtual testbed to run and evaluate distributed applications using the Mininet network simulator. The testbed was enhanced by enabling fault injection using link state update commands through Mininet, a reusable attacker component, and a behavior model script to schedule them.

All of these individual components were integrated with the resilient distributed middleware platform for distributed applications RIAPS. We also showcased the application of one of the features provided by RIAPS, *groups*; to implement two fault tolerance patterns called *redundancy* and *consensus* groups to deal with different failure modes. Using a real-world application of a microgrid energy management system designed using RIAPS, it was shown how a seamless workflow can be created using the tools introduced to significantly ease the progression of distributed software development from design to validation. The experiments successfully demonstrated how the deployment decision can affect the fault tolerance behavior of a distributed application and can also help maintain the resource limitations imposed by the hosting platform. Combining it with fault-tolerant algorithm design and platform-level features can provide more robust functional guarantees for applications deployed in production environments.

Future work would include looking at incorporating more types of faults into a reusable library for testing, as well as exploring ways of automating the weaving of fault tolerance patterns into any RIAPS application.

CHAPTER 5

Resilient Communication Connectivity Trade Offs in Distributed Algorithms for Cyber Physical Energy Systems

5.1 Problem Statement

Distributed, Peer-to-peer applications for Smart Grids include algorithms for a host of energy management functions such as voltage control [Almasalma et al. (2018)], energy trading [Abdella and Shuaib (2018)], economic dispatch [Rana et al. (2018)], islanding and resynchronization of microgrids [Du et al. (2018)], reconfiguration and restoration [Liu and Srikantha (2019)] etc. In these peer-to-peer architectures, computational components with communication capabilities are associated with the various *physical* elements of the grid, such as distributed energy resources (DERs), solar photovoltaic (PV), sensors and relays that are regulated and managed by an Energy Management System (EMS). A peer-to-peer EMS comprises a set of algorithms that achieve a common goal of the system or perform some action. It achieves this through the exchange of information with every other participant without going through a centralized controller. Thus, each component combines its local measurements with the information received from its peers at each iteration to produce an output. The output is usually a type of optimization cost or consensus value (which may be used as a control signal). This output signal is improved iteratively as more and more information is exchanged.

The traffic pattern for the peer-to-peer EMS agents is always *any-to-any*. In such cases, a *fully connected* peer-to-peer configuration ensures the highest amount of information dissemination throughout the network, but also generates the highest network traffic at each round. To be precise, in a network of n nodes, fully connected communication generates $n \times (n-1)$ messages per round (Figure 5.1). This becomes an issue once the size of the system becomes larger, as it might cause messages to be delayed or dropped by the network. One way to overcome this bottleneck is to establish a virtual topology on top of the communication network that creates a sparsely connected communication graph. A virtual topology can be defined as an application-layer overlay that determines how each peer shares information with the rest of the network. Using such virtual topologies, peers can choose a subset of neighbors with whom to share information instead of flooding the network. However, in such a setup, since fewer information is exchanged at each iteration, the convergence performance of distributed algorithms could be affected both in terms of accuracy and speed (Figure 5.1). Thus, there exists a trade-off between convergence efficiency and the amount of network traffic generated that must be considered by system designers for peer-to-peer EMS.

This presents the following challenge: *How can power system engineers and algorithm developers decide*

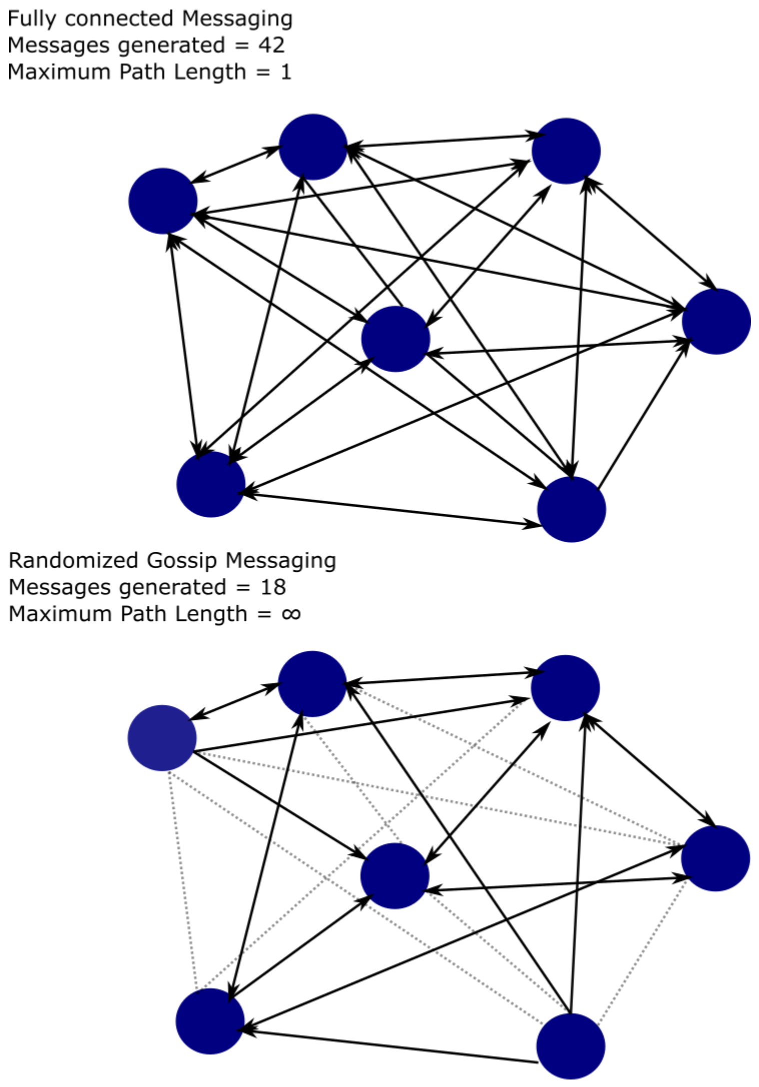


Figure 5.1: Difference between fully connected and sparsely connected (gossip style) messaging

on the connection topology that is best suited to a particular application? The choice of the best virtual topology for a particular network depends on the operational goals and the characteristics of the underlying electrical network. Theoretically, validating how topologies impact peer-to-peer control strategies requires individually tracking all node updates through multiple rounds and manually looking at how each node is connected to others, which becomes infeasible when the number of nodes increases to a larger number. To address this need, we introduce an integrated, configurable framework, *Topology Manager for Peer-to-Peer Links, or TopLinkMgr*, to implement and deploy various topology configurations on remote edge devices for evaluation and production deployments. It allows users to specify any custom connectivity graph using a novel text-based domain specific modeling language, *TopLink*.

While expert users can use TopLinkMgr to come up with their own topology, manually altering a given

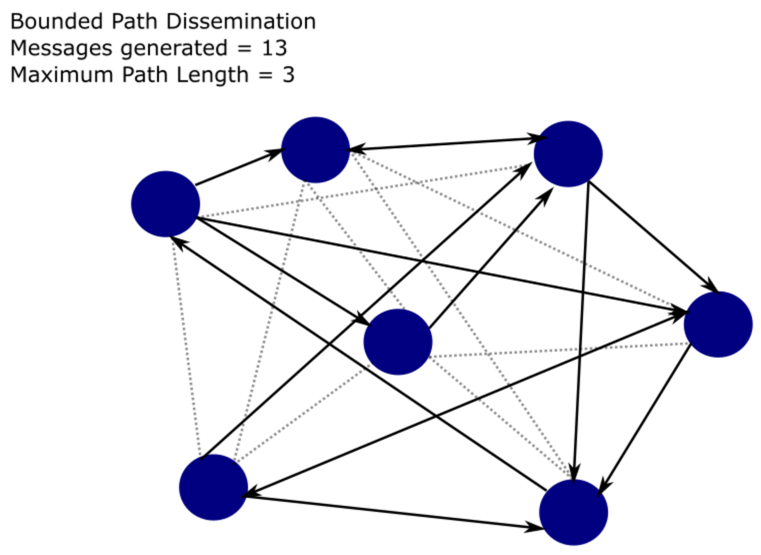


Figure 5.2: Communication graph for Bounded Path Dissemination

topology each time through trial and error is a tedious task for nonexpert users. The initial topology that is deployed can often be improved by a few tweaks to peer-to-peer connectivity without going through a complete redesign. As discussed, the key criteria for a peer-to-peer smart grid application is the effective dissemination of information to all parts of the network. Thus, the final chosen topology should be able to improve on that characteristic.

This brings about a second challenge -*How to devise a strategy that can automatically adapt and improve the dissemination of information to all peers in the case of a sparsely connected peer-to-peer communication topology, especially for nonexpert users?* In terms of graph theory, the problem can be formally defined as, given a set of peers and V, determining a set of virtual connections $E : V \times V$, such that there exists a virtual path $P_{ij}\{e : e \in E\}$ between any two nodes v_i and v_j and the minimum path length satisfies $1 \leq min(n(p_{ij})) \leq th$. Here, $n()$ denotes the cardinal number of a set and th denotes a maximum threshold.

We introduce a novel group-based topology management and peer selection algorithm *BPD*. It dynamically adapts a given virtual communication topology to limit the minimum path length within an adjustable threshold (Figure 5.2). For a safety-critical field such as Smart Grids, fault tolerance also becomes an important factor. The system must still be able to maintain its operational goal under multiple node failures. Thus, the proposed algorithm has dedicated fault tolerance protocols that can prevent partitioning of the virtual network if one or more nodes drop out, thus ensuring information dissemination.

There has been considerable work on the topic of peer-to-peer network overlays for distributed applications such as large scale file sharing and look up, load balancing and so on [Lua et al. (2005)]. They can be categorized into two categories, namely *structured* and *unstructured*. Structured overlays are static in nature where the memberships are predefined and remain unaltered throughout the operation lifecycle [Castro et al. (2002), Dhara et al. (2010), Shukla et al. (2021)]. On the other hand, in unstructured overlays, memberships are determined at run-time and can change at each round [Ganesh et al. (2003) , Genç and Özkasap (2005), Jin and Chan (2010)]. An interesting area of research for unstructured overlay networks is the peer selection and membership maintenance strategy. Careful consideration must be given in this aspect with respect to the application requirements. Class-based application-conscious membership strategies can efficiently distribute information to peers that need them the most with reduced latency [Miguel et al. (2021)]. On the other hand, randomized membership functions are much simpler to implement and require less time to set up, and are also extremely fault tolerant [Voulgaris et al. (2005)].

For a time-bounded critical application such as an EMS, it becomes extremely important to ensure that the system converges quickly so that the control algorithm can be actuated at a higher frequency. The highest rate can obviously be achieved by fully connected networks, but that has its own limitations, as discussed previously. Thus, a potential research question in this case is to explore if there is any optimal way of selecting members in a peer-to-peer overlay that can improve the message convergence time over the state of the art. Fault tolerance is also an important criterion because of which a simpler, easy to maintain algorithm would be better suited. Therefore, the aim of this work is to develop a lightweight virtual topology creation algorithm that ensures the reliable delivery of application messages for an *any-to-any* traffic pattern.

In summary, the main contributions are as follows.

- *TopLink*, an intuitive, text based specification language which can be used by users to describe any chosen topology graph. The specification can then be parsed and translated to deployment configurations.
- *TopLinkMgr*, a configurable framework to implement and deploy various topology configurations on the RIAPS platform that allows the deployment of different peer-to-peer connectivity configurations.
- *Bounded Path Dissemination (BPD)*, a topology management algorithm that improves the speed of information dissemination without flooding the network and is resistant to node failures.

The said contributions are adapted and expanded from the published work Ghosh, P., Shekhar, S., Lin, Y., Muenz, U., and Karsai, G. (2022a). Peer-to-peer communication trade-offs for smart grid applications. In *2022 International Conference on Computer Communications and Networks (ICCCN)*, pages 1–10. IEEE ©2022 IEEE.

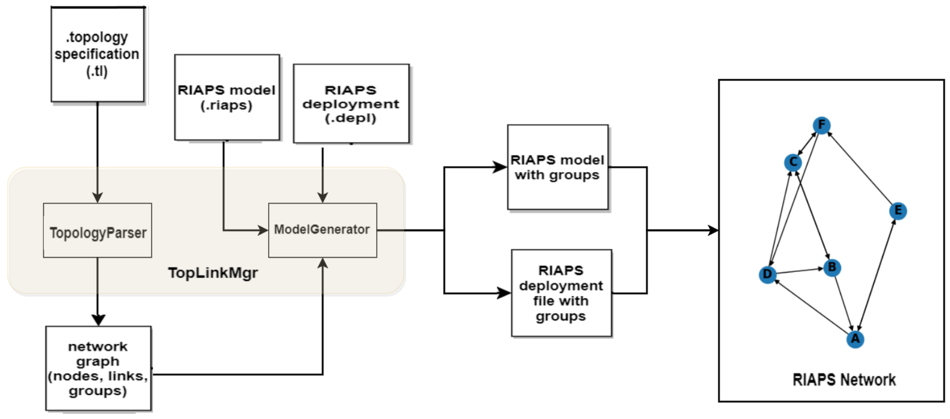


Figure 5.3: TopLinkMgr Architecture ©2022 IEEE

5.2 Solution approach

The overall design approach for the proposed framework can be divided into two stages. The first stage comprises the development of an automated topology manager for peer-to-peer links or *TopLinkMgr*. The second stage is the implementation of a peer selection strategy, the *Bounded Path Dissemination* algorithm that improves the gossip style message dissemination protocol for faster network coverage. We discuss both in more detail in the following sections.

5.2.1 Topology Manager for Peer-to-Peer Links

This section describes our configurable virtual communication topology framework, *Topology Manager for Peer-to-Peer Links (TopLinkMgr)*. It acts agnostic to the overlying application logic and can be leveraged by application developers readily without worrying about the implementation details. The main goal of TopLinkMgr is to allow application users to deploy distributed peer-to-peer applications using different communication structures to evaluate and determine the best option depending on the application needs. In order to facilitate that, the manager also comes with a library of topologies, which are commonly employed, along with providing flexibility to users to design a completely custom topology from scratch. All these options can be provided through the use of a simple text-based user specification language that has been developed for the utility, called *TopLink*. Using *TopLink*, users can specify the names of participating nodes, the type of communication topology, and define the individual links for a custom topology in a `.tl` file, which is then fed to a *TopologyParser* tool that uses the information to translate the user specifications into the appropriate network graph.

To develop and deploy the Smart Grid applications for this article, we also used the integrated decentralized software framework RIAPS. The TopLinkMgr framework utilizes the groups feature, which enables the user-specified virtual topology to be realized into a working RIAPS application.

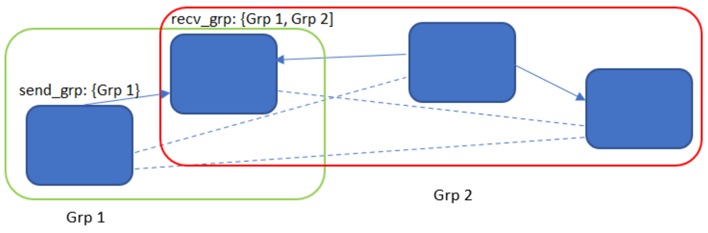


Figure 5.4: Group formation. The rectangular outlines represent the groups. The send and receive groups as perceived by the two left-most nodes respectively are illustrated. ©2022 IEEE

5.2.1.1 Process Flow

A virtual topology can be constructed by assigning peers to groups based on how one peer is logically connected to another. This is done by the module *TopologyParser*, which takes as input a RIAPS model file that contains the actor, component, and message definitions, and a topology specification file written using *TopLink*. For each set of communication links that connect a source peer to any number of destination peers, a group needs to be created. In this context, a group can be defined as a means of establishing the information flow relation between nodes. In TopLinkMgr, there are two categories of groups, a *send group* for outward communication and a *receive group* for inward communication. The number of groups is determined by the defined virtual topology, and then each node is assigned to be part of these groups based on the virtual connections. The group categories are relative to each node.

For example, consider the graph of Figure 5.4 with four nodes, represented as blue rectangles that are logically connected according to the solid arrows. The direction of the arrows specifies whether the node can send messages to the other node or receive messages from it. The dotted lines represent unused communication links that are not part of the virtual topology. To implement this configuration, it would require the formation of two groups, *Grp 1* comprising the left two nodes and *Grp 2* comprising the right three nodes. Since the left most node acts as the sender, TopLinkMgr will assign *Grp 1* as its *send group*. Similarly, the second node from the left receives from the other nodes of both *Grp 1* and *Grp 2*. Thus, ToplInkMgr will assign both groups as its *receive group*.

Finally, once the group assignments are calculated, the RIAPS model deployment file is generated by the *ModelGenerator* utility which also weaves the group configurations and necessary commands into the model file. If a deployment file already exists, the *ModelGenerator* modifies it by adding group arguments, or else it generates a new file. It also automatically generates the required component code to join the appropriate groups. Thus, the output of the entire process flow is a complete working application that implements the specified communication topology among the specified peers. The complete architecture and the process flow

```
// RIAPS Application Model
app GroupTestApp {
    // message types
    message SensorData;
    message NodeData;
   ...
  // Sensor component
    component Sensor(value=1.0) {
      timer clock 10000;
      pub sensorReady : SensorData ; //
           Publish port
    }
    // Averager component
component Averager(Ts=1.0){
      sub sensorReady : SensorData ;  //
           Subscriber port
      pub thisReady : NodeData; // Publish
           port
      sub nodeReady : NodeData; // Subscriber
           port
      ...
    }
...
    // Averager actor
actor Averager(value=0.0,Ts=100.0){
       local SensorData;          // Local
            message types
       {
          sensor : Sensor(value=value);
      averager : Averager(Ts=Ts);
       }
    }
...
}
}
```

(a) RIAPS application model

```
// topology configuration for RIAPS app
// Actor.Component
for Averager.Averager {
create topology custom; // topology name
over (bbb1, bbb2, bbb3, bbb4, bbb5, bbb6); //
     nodes
using (bbb1, bbb2), (bbb2, bbb3), (bbb2, bbb5)
     , (bbb2, bbb6), (bbb3, bbb4), (bbb3, bbb6
     ), (bbb4, bbb5), (bbb5, bbb3), (bbb6,
     bbb1), (bbb6, bbb5) ; // links
inter-group communication on; // turn on
     communication between group leaders
}
```

(b) TopLink input file (all features not shown)

Figure 5.5: TopLinkMgr input files showing an example RIAPS application model and a TopLink specification

```
app GroupTestApp {

on (192.168.57.2) Averager(value = 0.0,Ts = 100.0,send_grp = "grp1",recv_grp = "grp0") ;

on (192.168.57.1) Averager((value = 0.0,Ts = 100.0,send_grp = "grp0",recv_grp = "grp5") ;

on (192.168.57.3) Averager(value = 0.0,Ts = 100.0,send_grp = "grp2",recv_grp = "grp1,grp4") ;

on (192.168.57.5) Averager(value = 0.0,Ts = 100.0,send_grp = "grp4",recv_grp = "grp1,grp3,grp5") ;

on (192.168.57.6) Averager((value = 0.0,Ts = 100.0,send_grp = "grp5",recv_grp = "grp1,grp2") ;

on (192.168.57.4) Averager((value = 0.0,Ts = 100.0,send_grp = "grp3",recv_grp = "grp2") ;

...

}
```

Figure 5.6: TopLinkMgr output deployment file for the input of Figure 5.5

are depicted in Figure 5.3.

5.2.1.2 TopLink Specification Language

TopLink is a text-based domain specific modeling language that can be used by application developers to describe their chosen topology, either a pre-configured or a completely custom one. It provides special keywords through which users can define different aspects of the topology. Properties that can be defined using the file include:

- The RIAPS application actor and component for which the connectivity is defined.
- The type of topology: either *preset* or *custom*. Currently, the preset topologies that are supported are a ring type and a fixed fan-out randomly generated graph, but in the future we plan to add more.
- The node identifiers for the network (IP addresses and host names).
- Individual communication links between nodes (if the topology is custom).
- Weights for individual links if the network links are not uniform.
- If communication is desired among the leaders of the formed group, it can also be enabled.

Figure 5.5 shows a RIAPS application model for a peer-to-peer algorithm that defines the various components, their communication ports and port message types, the actor that encapsulates them, and a corresponding topology specification file that describes a custom topology. Running TopLinkMgr produces a modified model file, a RIAPS deployment file, and the various component starter code as output. The modified model file contains the initialization of the TopLink group and the associated group message type. The deployment file (Figure 5.6) contains the details of how the application actors will be deployed to the various remote nodes (denoted by their IP addresses or hostnames) and the specific arguments for each deployed actor, if necessary. The TopLinkMgr generated deployment sets the send and receive group names (which are instances of the TopLink group) as arguments for the corresponding actors based on the topology. An example is shown in Figure 5.6. Although some features of the language described are based on the RIAPS modeling architecture, the language itself is not tied to RIAPS and can be made to work with other platforms as well using dedicated parsers.

5.2.2 Bounded Path Dissemination Algorithm for Topology Management

Let us consider a fairly common peer-to-peer use case for Smart Grids, the distributed Optimal Power Flow (OPF) problem [Jabr et al. (2002)] for a network of microgrids [Bower et al. (2014)]. OPF is an optimization problem that minimizes the total generation dispatch cost while satisfying physical and technical constraints on the network [Jabr et al. (2002)]. Using the primal-dual algorithm to solve the given problem leads to equations 5.1 (primal update) and 5.2 (dual update). In both equations, k stands for the k-th node and t for time instant t. x and $\hat{\lambda}_t$ are the primal and dual variables, respectively. α, β, γ act as weights. In practice, each microgrid control node keeps a local copy of the dual variable. At each iteration, this dual variable is updated both by local measurements and by communication received from other controllers. This is reflected in the dual update equation 5.2, where the first term is updated from the measurement data and the second term is updated from the communication data. Of course, the implementation of such an algorithm requires that the nodes be synchronized (time- or event-based) to drive the iterations. The performance of the algorithm compared to a fully centralized system for sparse communication connectivity becomes worse than all-to-all connectivity.

$$x_{k,t+1} = Proj_{\chi}[x_{k,t} + \alpha_{k,t}(\nabla f_k(x_{k,t}) + A_k \hat{\lambda_{k,t}})] \tag{5.1}$$

$$\hat{\lambda_{t+1}} = Proj_{\mathbb{R}}[\hat{\lambda}_t + \beta_t M(y-b)] - \gamma_t L \hat{\lambda}_t \tag{5.2}$$

As discussed previously, the convergence of peer-to-peer algorithms such as the one discussed above, both in terms of how close the final solution is to an optimal one and how many iterations it takes to reach it, depends on the proper dissemination of information. Thus, any virtual topology deployed must achieve the balance between proper distribution of messages and limiting network traffic. Another important consideration for a safety critical system, such as smart grids, is its resilience to unexpected faults that cause crashes of individual nodes and lead to partitioning of the virtual topology.

In order to address these requirements, we introduce the *BPD* algorithm. This algorithm improves the dissemination of information to all peers in the network by reducing the hop length of the path from one node to all other nodes within a threshold, which can be specified as an input parameter. The threshold of 1 implies a fully connected graph. BPD limits the minimum number of iterations required for the message generated by a particular node in the network to reach all its peers. This results in speeding up of the information coverage throughout the network, which effectively improves the performance of a peer-to-peer application algorithms over the unmodified scenario. Additionally, BPD also has dedicated self-healing capabilities that can ensure that peers remain connected even if one or more drop out due to faults.

The following notations and definitions will be used throughout the remainder of this section.

- $node_i = (id, send\_grp_i, recv\_grp_i)$ denotes a participating node in the network. It acts as an endpoint for peer-to-peer communication. It consists of an identifier id, a sending group object $send\_grp_i$ and a receiving group object $recv\_grp_i$ that realizes the topology.
- id is a unique identifier for a node. In the case of RIAPS, it is a generated universally unique identifier string (uuid).
- $grp_m = (id, size, send(), recv())$ denotes a group object. It consists of an identifier for the group, the

attribute *size* that stores the number of members currently present in the group and two methods $send()$ and $recv()$ for sending and receiving messages to and from that group.

- $send\_grp_i = \{grp_m.id, weight, send()\}$ refers to the sending group object of $node_i$. It contains the id of the group that $node_i$ has joined as a send_grp. The weight attribute represents the cost incurred for the outgoing links of $node_i$. The default value is 1. The send and receive operation here refers to invoking them on all groups with ids present in send_grp.
- $recv\_grp_i = \{grp_m.id, recv()\}$ refers to the receiving group object of $node_i$. It contains the ids of the groups that $node_i$ has joined as a recv_grp. The send and receive operation here refers to invoking them on all groups with ids present in send_grp.

The algorithm comprises two stages, namely *Discover Peers* and *Group Update*. The first stage dynamically gathers information about the various node-to-node paths within the topology and their lengths, while the second stage then uses that information to alter some of the paths to ensure that they all lie within the defined threshold. For our setup, an external agent called *TopologyManager* was responsible for sending commands to each peer to initiate the algorithm stages. However, the same can be triggered using an internal clock. For smart grids, the number of nodes is determined by the elements of the grid. The network is closed and private for security. Thus, there is no node churn (unlike cloud distributed systems [IBM (2021)]). Therefore, the algorithm starts with an initial configuration of the nodes generated by the topology framework. However, the discovery phase still ensures generalizability since the path lengths are calculated dynamically without any knowledge of the initial configuration.

5.2.2.1 Stage 1: Discover Peers

At this stage, all peers share information about themselves that is propagated through the topology. This information is used by each node to determine and store the minimum paths to all other peers in the network in an internal table. It begins when TopologyManager sends a *discoverPeers* message to all peers on the network. On receiving the message, each node creates a new message containing its own id, a *depth* variable initialized to 0, and sends it to each of the groups in its recv_grp. It also puts its recv_grp id into the message.

On receiving the message, the recipient checks if the group id matches its send_grp id. If the depth field is lower than the previously recorded depth of the same node, then it updates the table entry. Next, it increases the depth by the weight associated with that send_grp and forwards it to its recv_grp. At the end of this round, each node will have a complete table containing the node ids and the depth corresponding to that node, indicating the path cost. The steps are shown in Algorithm 1.

Algorithm 1 BPD Stage 1: DiscoverPeers

```
Input: Initial $send\_grp_i$ and $recv\_grp_i$ for $node_i$
Output: $path_i.node_j \leftarrow (\{node_j : depth_{ij}\} \forall node_j)$ for $node_i$, $i \neq j$
on event $DiscoverPeers$ do
$msg_i \leftarrow \{$ id : $node_i.id$, depth: 0, ‘grp’ : $recv\_grp_i$, ‘order’ : 0, ‘weight’: $recv\_grp_i.weight$ $\}$
$recv\_grp_i.send(msg_i)$
on event $send\_grp_i.recv(msg_j)$ do
if $msg_j.grp = send\_grp_i.id$ then
    $msg_j.depth \leftarrow msg_j.depth + msg_j.weight$
    if $msg_j.depth < path_i.node_j.depth$ then
        $path_i.node_j \leftarrow msg_j$
        $recv\_grp_i.send(msg_j)$
    end if
end if
```

5.2.2.2 Stage 2: Group Update

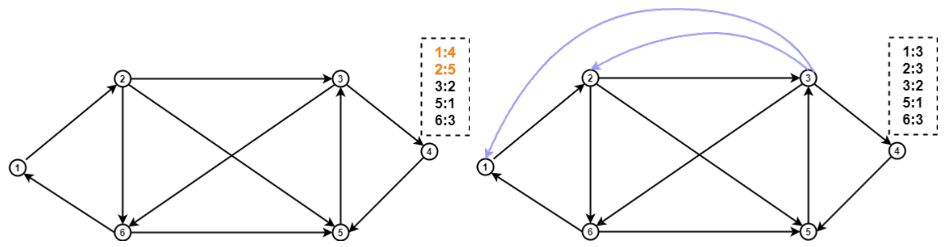
This stage uses the fully fleshed out table entries from Stage 1 to limit the minimum path cost from all nodes of the network to every other node based on a threshold. This threshold is a parameter that can be decided by the user using the connectivity and the desired rate of information dissemination. For the experiments carried out in this work, a threshold was selected using the formula $thresh = (N-1)/2$, where N is the number of nodes in the network, which is half the number of edges required to form a spanning tree, the minimal fully connected graph comprising N nodes. Application designers can select a threshold based on how much they want to compromise convergence performance while trying to reduce bandwidth consumption. It begins when TopologyManager sends a *groupUpdate* message to all peers on the network. On receiving the message, all nodes look at the paths that they have stored locally. If for any path $depth > threshold$, then that node sends a message containing its id, new depth = 0 and weight to its send_grp.

On receiving the message, the recipient node increases the depth by the weight of its send_grp and forwards the message to it. When the message reaches a node for which $depth = thresh - weight$, it appends its send_grp id to the message and forwards it. When the message reaches the target node, the node reads the group id from the field and joins that group. This effectively means that a new link is added to the network graph linking the target node to an intermediate node between it and the source such that the total path depth from the source does not exceed *thresh*. It terminates when there are no new messages to forward. The steps are shown in Algorithm 2.

Figures 5.7a and 5.7b illustrate the operation of the two stages of the algorithm for an example of a 6-node topology. The selected threshold is 3. In Figure 5.7a, it can be seen that after running the *DiscoverPeers* protocol to determine the relative path lengths for all nodes, for node 4, nodes 1 and 2 are further than the threshold. Thus, when the *Group Update* protocol is triggered, nodes 1 and 2 join the send_grp of node 3,

Algorithm 2 BPD Stage 2: GroupUpdate

```
Input: $path_i$ from Stage 1 for $node_i$, $thresh$
Output: $path_i.node_j \leftarrow (\{node_j : depth_{ij} \mid depth_{ij} \leq thresh\} \forall node_j)$ for $node_i$, $i \neq j$
 1: on event $GroupUpdate$ do
 2: for $node_j, depth_j$ in $path_i$ do
 3:     if $depth_j > thresh$ then
 4:         $msg_i \leftarrow \{$'req': $node_i.id$, 'id' : $node_j.id$, 'length' : 0, 'grp': "", 'send_grp' : $send\_grp_i.id$ $\}$
 5:
 6:     end if
 7: end for
 8: on event $recv_grp_i.recv(msg_j)$ do
 9: if $msg_j.req! = node_i.id$ then
10:     if $msg_j.depth <= thresh - weight_k$ and $msg_j.grp ==$ "" then
11:         $msg_j.depth \leftarrow msg_j.length + weight_k$
12:         if $msg_j.depth == thresh - weight_k$ then
13:             $msg_j.grp \leftarrow send\_grp_i.id$
14:         end if
15:     end if
16:     $send\_grp_i.send(msg_j)$
17: else
18:     $joinGroup(msg_j.grp)$
19:     $send_grp_i.id \leftarrow msg_j.grp$
20: end if
```



(a) Node 4 after Stage 1 (b) Node 4 after Stage 2

Figure 5.7: Bounded Path Dissemination Algorithm on a 6 node topology ©2022 IEEE

which establishes a path from node 4 to nodes 1 and 2 via node 3. It can be seen that after the first round is complete, the new (shortest) path of nodes 1 and 2 from node 4 becomes 2, which is within the set threshold. Thus, BPD ensures that any message sent from node 4 would reach nodes 1 and 2 within 3 iterations, provided that node 3 is intact. However, that might not always be the case, as unexpected faults might cause nodes to drop out of the network. How BPD can handle such scenarios is discussed in the next subsection. It must be noted that there can be multiple solution topologies that satisfy the requirement. BPD currently does not differentiate between them, since in terms of minimum path length, they are all equivalent. However, in future works, the algorithm can be optimized, say, for e.g. the solution time, or minimum number of edges etc. The algorithm will converge, provided that the starting topology is connected. The speed of convergence depends on both the number of nodes and the threshold selected.

5.2.2.3 Fault Tolerance

The objective of the fault tolerance protocol is to ensure that the information originating from healthy nodes is distributed to all other peers when one or more nodes drop out due to failure. When such faults happen, it can lead to two things, either some of the path lengths can be altered and exceed the threshold since an intermediate route is removed, or one or more nodes can become isolated or partitioned from the rest of the network. In the first case, the algorithm can self-repair by running the two stages periodically. The period used for the experiments was 2 min. For the second case, there can be two scenarios. The fault tolerance protocol utilizes the leadership feature of groups.

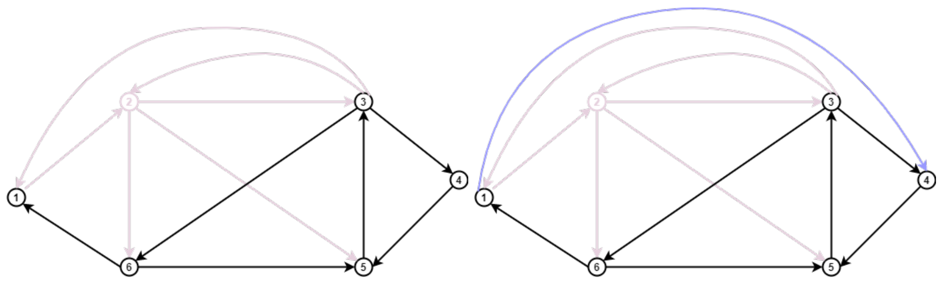
1. *When a member of the send_grp goes offline* Since for each group, the source node joins it as a send_grp itself and all other nodes that receive from it join that group as a recv_grp, if a group has more than one member, it implies that there exists a path from one peer to the other. If a peer is the only member of its send_grp, this implies that that node is partitioned from the rest of the network. The algorithm then proceeds to connect that node to the rest of the network by joining a new group. If a peer is the lone member in a group, then sends a *join_req* message containing its own id and *grp_type* as send_grp to the group leaders. The leaders respond with a *join_rep* message by adding to *join_req* the group id and the group size of the groups in their recv_grp. On receiving the message, the requesting peer chooses the entry with the minimum size and joins that group as specified in the *grp_type* field, in this case as a send_grp. The steps are described in Algorithm 3

   In Figure 5.8, when node 2 goes offline, node 1 becomes the lone member of its *send_grp*, it then initiates the protocol and joins the recv_grp of node 4, thus reconnecting it.

2. *When a member of the recv_grp goes offline* When a node detects a member of the recv_grp has left,

Algorithm 3 BPD Fault-tolerance: If send_grp member leaves

```
1:  on event $MemberLeft(group)$ for $node_i$ do
2:  if $group.id$ in $send\_grp_i.id$ then
3:      if $group.size < 2$ then
4:          $join\_req_i \leftarrow$ { 'req' : $node_i.id$, 'grp_type' : 'send_grp'}
5:          $leader\_grp.send(join\_req_i)$
6:      end if
7:  end if
8:  if $Memberof(leader\_grp)$ then
9:      on event $leader\_grp.recv(join\_req_j)$do
10:     $mingrp \leftarrow minSize(recv\_grp_i)$
11:     $join\_rep_j \leftarrow$ { 'req' : $node_i.id$, 'grp_type' : 'recv_grp', 'grp': $mingrp.id$, 'size' $mingrp.size$ }
12: end if
13: on event $leader\_grp.recv(join\_rep_i)$
14: if $count(join\_rep_i) = leader\_grp.size - 1$ then
15:     $mingrp \leftarrow minSize(join\_rep_i.grp)$
16:     $joinGroup(mingrp)$
17:     $send_grp_i.id \leftarrow mingrp.id$
18: end if
```



(a) Node 2 goes offline (b) Node 1 is connected to Node 4

Figure 5.8: Fault-tolerance protocol for the scenario when a member of the send_grp foes offline ©2022 IEEE

it can imply two possibilities, one, that one of the other receivers in the group has crashed, or two, that the sender node for that group has crashed. For possibility one, the node does not need to act because it still remains connected to its sender, but for possibility two, the node must join a new group. Once a peer detects it, it sends a *grp_qry* message on that particular group containing the group id. If a recipient has the same group as its send_grp, it responds with its own id. Once the node receives all the responses, if none of the responses contains the sender's id, it implies that the sender is offline. It then proceeds to send a *join_req* as described previously but with 'grp_type' as 'recv_grp'. The steps are described in Algorithm 4.

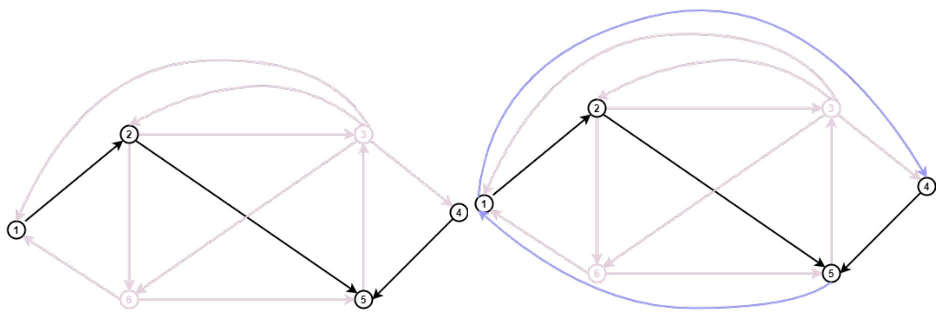
In Fig. 5.9, when both node 3 and node 6 go offline, node 4 and node 1 become isolated from the rest of the topology, it then initiates the protocol, and node 4 joins the send_grp of node 1 while node 1 joins the send_grp of node 5. As a result, all the nonfaulty nodes 1,2,4 and 5 become connected again with both incoming and outgoing routes to each other. In this case, none of the new paths exceeds the threshold distance of 3. However, there might be certain scenarios where that might be the case. In such cases, the next round of the *DiscoverPeers* and the *GroupUpdate* stages can again restructure the connectivity graph to ensure a bounded path.

Algorithm 4 BPD Fault-tolerance: If recv_grp member leaves

```
1:  on event $MemberLeft(group)$ for $node_i$ do
2:  if $group.id$ in $recv\_grp_i.id$ then
3:      $grp\_qry_i \leftarrow$ { 'req' : $node_i.id$, 'grp_type' : 'recv_grp', 'grp' : $group.id$ }
4:      $recv\_grp.send(grp\_qry_i)$
5:  end if
6:  on event $recv\_grp_i.recv(grp\_qry_j)$do
7:  if $grp\_qry_j.grp = send\_grp_i.id$ then
8:      $grp\_ans_i \leftarrow$ { 'req' : $node_i.id$, 'grp_type' : 'recv_grp', 'grp' : $group.id$, 'rep' : $node_i.id$ }
9:  else
10:     $grp\_ans_i \leftarrow$ { 'req' : $node_i.id$, 'grp_type' : 'recv_grp', 'grp' : $group.id$, 'rep' : '' }
11: end if
12: on event $recv\_grp_i.recv(grp\_ans_i)$
13: if $count(grp\_ans_i) = recv\_grp_i.size - 1$ then
14:     if $grp\_ans_i.rep == '' \forall grp\_ans_i$ then
15:         do Algorithm 3 with grp_type = 'recv_grp'
16:     end if
17: end if
```

5.3 Evaluation

The experiments performed mainly looked at evaluating two aspects: the performance of a peer-to-peer algorithm in terms of its convergence accuracy and speed as a result of using the algorithm, as well as the resilience properties of the algorithm under fault conditions, which were the objectives for which the algorithm was designed.



(a) Node 3 and Node 6 go offline (b) Node 4 and Node 5 are connected to Node 1

Figure 5.9: Fault-tolerance protocol for the scenario when a member of the recv_grp foes offline ©2022 IEEE

5.3.1 Experimental setup

The experiments were carried out using a network of six embedded computing nodes running RIAPS. Communication was carried out using Ethernet. A virtual machine running Linux Ubuntu 18.04 was used as the control node from which the applications were deployed to the target nodes using the RIAPS control graphical user interface.

The algorithm was compared with the base topology (of Figure 5.7a) without any modifications, a fully peer-to-peer approach with all-to-all communication, and a randomized gossip style *pull* pattern that is used in most state-of-the-art schemes. The peer-to-peer algorithm used to carry out the experiment was a simple distributed consensus algorithm. For a linear system of N nodes with the local state of the i-th node denoted as x_i, the distributed consensus algorithm updates the state according to the equation 5.3. For discrete time, it can be modified to the form of equation 5.4, with the derivative replaced by the difference operator for the discrete case.

$$\dot{x}_i(t) = \sum_{j \in N \setminus i} a_{ij}(x_j(t) - x_i(t)) \tag{5.3}$$

$$x_i[k+1] = x_i[k] + \sum_{j \in N \setminus i} a_{ij}(x_j[k] - x_i[k]) \tag{5.4}$$

Here, a_{ij} represents the connectivity between node i and node j. It can be shown that the algorithm theoretically converges to the average of the initial states $x_i[0]$ of the system.

The above algorithm was coded into the Averager RIAPS component logic for the application model described in Figure 5.5a in Python. A logger component was also added to the model to collect data for the experiments. The application was allowed to run for 10 minutes for each test to ensure sufficient time for the averaging algorithm to converge. The metrics evaluated for algorithm convergence were the percentage

Method	**Deviation from optimal value**	**Min. number of iterations to reach** $\pm 5\%$	**Messages per iteration**	**Time taken (s)**
All-to-All	4.55%	445	30	3
Gossip	10.76%	172	18	1.8
Unmodified	11.32%	320	10	1
BPD	6.28%	60	13	1.3

Table 5.1: Convergence Performance Comparison ©2022 IEEE

deviation from the optimal value (true average) and the number of iterations taken to reach within a 5% tolerance band of that value. The number of messages generated per iteration was also recorded. As expected, all-to-all had the highest with 30 (6×5), followed by Gossip which had 18 (3×6), the graph of Figure 5.7 initially had 10 links and BPD added 3 more to make the paths bounded. The initial values for the different runs of the peer-to-peer averaging algorithm were changed so that there was a different optimal value for each run of the experiment to eliminate any bias in the results.

5.3.2 Algorithm Convergence Results

Table 5.1 lists the convergence performance of the different topology algorithms. In terms of peer selection strategy, as expected, a fully connected network produces the most accurate results with a deviation within 5% of the theoretical optimal value. However, in doing so, it also consumes the most messages per round (iteration), further emphasizing the trade-off that was discussed previously. BPD improves upon the other approaches on both convergence accuracy and speed. It reaches steady state the fastest, but the value is not optimal. In our experiments, we also observed sharper fluctuations in the values as the algorithm progressed through successive rounds compared to the all-to-all configuration. This is because the new information received caused a larger correction in some rounds than in others. Gossip performance lies somewhere in between. Due to its inherently random nature, some nodes perform well, since they can receive all the information, but other nodes might take longer to receive the same. However, Gossip will perform better when a large number of nodes are incoming and outgoing, since the algorithm is automatically scalable.

5.3.3 Fault tolerance Results

Since the operational efficiency of a peer-to-peer Smart Grid application depends on the effective dissemination of information to all participating nodes in the network, we measure the success of our fault tolerance logic on the basis of its ability to maintain network coverage in the presence of faults. Gossip reliability is a measure that has traditionally been used to evaluate broadcast algorithms. It is defined as the percentage of active nodes that can transmit a gossip broadcast, with 100% denoting a perfectly reliable broadcast [Leitao

Method / Failure	Gossip	Unmodified	BPD
17 % (1/6)	0.82	0.54	0.83
33 % (2/6)	0.66	0.17	0.66

Table 5.2: Dissemination Performance Comparison under Faults ©2022 IEEE

et al. (2007)]. However, it only considers messages originating from one source and forwarded by others until they reach all other nodes. It does not capture the information that shows whether messages originating from all nodes reach all other nodes, which is the case for a peer-to-peer application. Thus, we slightly modify the definition of reliability to be the fraction of messages originating from distinct source nodes that were received by other peers in a peer-to-peer network. We can call this altered reliability *Dissemination Efficiency (DE)*. Its value ranges from $0-1$, where 1 implies that all peers received information generated by every other active member, while 0 implies that no messages were received.

Figure 5.10 shows the average DE for the various techniques under 17% and 33% node failure scenarios. It can be seen that the unmodified topology suffers heavily due to the network being partitioned. Gossip is inherently robust since it randomizes recipients and ensures that faulty nodes are eventually replaced in the sending list. BPD performs similarly to Gossip, since it can reconnect the graph, thus ensuring that the remaining nodes are able to receive information effectively from each other.

Figure 5.11 shows how the algorithm affects individual nodes in real time under fault conditions for the starting topology of Figure 5.7a. The plot shows the change in the DE for each node in response to node faults as the number of rounds of message exchange progresses. For the experiments conducted, the period chosen for each round was 10 ms. It shows that the DE stabilizes to the expected values of around $0.83(5/6)$ and $0.667(4/6)$ after the crash of node 3 and node 6, respectively. It also shows that the algorithm is able to restore the connectivity once a faulty node is repaired and it rejoins the network. The average time taken by BPD to restore the topology connectivity after a fault was 16 ms. There are some cases where it shows the DE to jump up to 1 right after a fault. This happens because to calculate the metric, each node stores a history of the values that it receives and its source, which is refreshed periodically. Thus, it takes some time for the residual entries to be removed, and then the true value is reflected. We will investigate more efficient methods for recording metrics in the future, possibly by adding tags to all outgoing messages or using sequence numbers to differentiate between current and older entries. However, this is only related to the way the data show up and has no effect on the algorithm performance. The sharp downward spikes are due to the time it took for the BPD protocol to be completed after detecting that a group member left. In all our experiments, this delay was within the range of 10 ms, which is quite low compared to the usual sampling

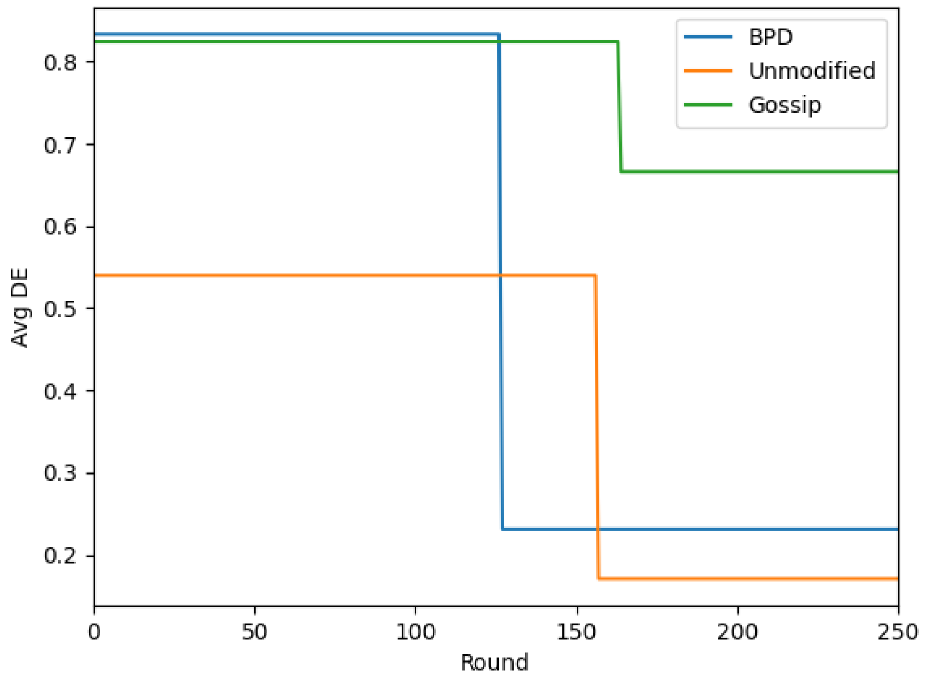


Figure 5.10: Average Dissemination Efficiency for Different Algorithms for 17% and 33% faults

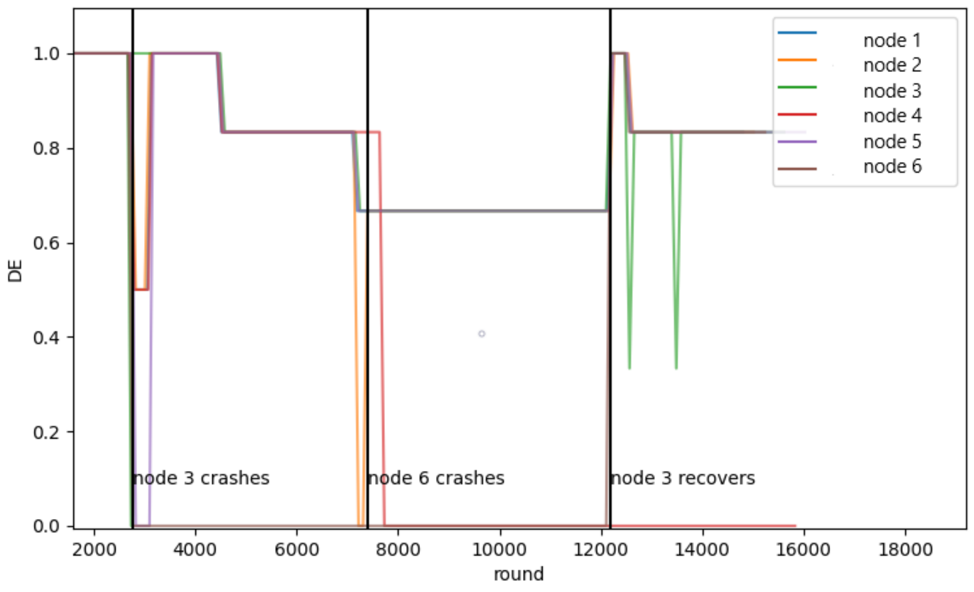


Figure 5.11: Node-wise Dissemination Efficiency for BPD showing faults and recovery ©2022 IEEE

time period of peer-to-peer EMS algorithms of > 100 ms.

5.3.4 Network Measurements

Table 5.3 shows the average bandwidth consumed per node and the latency data collected for the different methods. As seen from the data, BPD uses about 80% less network bandwidth compared to the fully connected configuration, with the performance of the other two lying in between. This is expected, since those two do not take any steps to optimize the virtual topology. However, the methods that employ virtual topologies (BPD modified and unmodified) produced a higher latency than the other two. This is due to the fact that using a virtual topology implies that each message needs to go through an additional layer of routing on top of the physical network routing, while in the other two cases it only needs to go through the physical network routing. Although the extra delay is still negligible for it to affect the experiments performed, we plan to study it in more detail for larger networks in the future.

Method	**Bandwidth consumed (kB/s)**	**Latency (ms)**
All-to-All	33	1.2
Gossip	21.5	0.88
Unmodified	12	3.6
BPD	6.8	2.4

Table 5.3: Bandwidth consumed and latency data for the different methods ©2022 IEEE

5.4 Summary

Peer-to-peer communication plays an important role in the implementation of several of the core functionalities for EMS. An important question that designers must address is how to achieve a balance between the amount of network traffic generated as the network scales and the preservation of performance goals with respect to the various algorithms that are deployed at each functional layer of the grid. The choice of communication topology is an important design decision that power system engineers must take into account with regard to that trade-off. We introduced an integrated framework for Smart Grid applications that allows users to deploy and prototype different virtual communication topologies for peer-to-peer applications and empirically evaluate their performance. Existing peer selection methods do not consider the interrelationship between communication sparsity and how it can reduce the convergence performance of peer-to-peer EMS algorithms employing them.

We also propose a new algorithm, Bounded Path Dissemination, which ensures the dissemination of information throughout all participating peers in the network within a specified threshold. It also has dedicated fault tolerance features to prevent partitioning of the virtual topology if one or more nodes drop out. Experimental evaluations that compare the performance of BPD with other state-of-the-art approaches show improved convergence accuracy and speed for a peer-to-peer application. Similar studies under faulty conditions also show that the algorithm is capable of maintaining network communication connectivity in the presence of such faults and the Dissemination Efficiency factor is on par with a robust randomized gossip technology. With the integration of more ad hoc participants in a grid such as hybrid electric vehicles, a self-adaptive virtual topology management algorithm such as BPD could potentially allow these ad hoc players to join or drop out dynamically. Thus, it would be interesting to study how the algorithm performs in the presence of such entities.

This work was mainly used to introduce the algorithm concepts and show them in a working example. In the future, large scale experiments on more complex networks need to be performed. Comparison between the proposed scheme and some other node linking choices, such as connecting two intermediate nodes, should also be considered. Techniques to optimize the topology based on the number of communication links and desired convergence, as well as a time-synchronized implementation of the averaging algorithm, are also interesting directions to explore.

CHAPTER 6

Fault Tolerant Load Shedding using a Decentralized Software Framework

6.1 Problem Statement

The previous chapters introduced techniques for imparting resilience characteristics in DCPS that could be generally applied to any application provided the platform and communication infrastructures supported the associated assumptions and features described. However, to design a high confidence system that can adapt to different operating conditions seamlessly and in ways that specifically optimize its operation with respect to its current control objectives, the associated control algorithms must also be integrated with the application-agnostic protection layers. This is important because the end user or customer is ultimately concerned with receiving the services they signed up for and the final benefit that the CPS serves them. For example, even if the underlying computing architecture within a microgrid is extremely robust to faults and able to maintain connectivity among the different agents, to the customers, it should still be able to provide the energy at a reasonable price under fault conditions for it to be beneficial.

The increased complexity and dynamism in Smart Grids are difficult to handle with traditional monitoring and control systems, motivating the industry and academia to investigate decentralized control and computing solutions for the grid. To ensure portability, most of these decentralized, real-time, embedded computing solutions should be based on open software application platforms that comply with industry standards and protocols, such as those being developed by IEEE and IEC [EPRI (2014)]. Having standards in place enables innovation and creativity across all aspects of the grid, from enterprise applications to local device control. Interaction protocols facilitate two-way data sharing, measurement and state estimation, leveraging of sensing device information for system availability and stability assessment, some local control/protection activity, and competitive transactions to encourage direct engagement with consumers [Monti et al. (2010), EPRI (2011)]. But, along with the open protocol and its advantages, comes a strong need for security, quality control, reliability, and resilience [EPRI (2011)].

In this chapter, we will describe the design and implementation of the concepts in RIAPS and how application developers can use them to integrate resilience into their applications. RIAPS distributes monitoring and control functions to computing nodes on the edge of a network, reducing total network traffic, improving reaction times by avoiding network latency compared to a centralized architecture, and increasing reliability by reducing dependency on availability and access to centralized resources. The platform supports multi-tenancy and the controlled sharing of computational and communication resources. The key goal of RIAPS is to provide a *middleware* to facilitate interactions between networked computational actors that focus on specific grid issues, such as state estimation, remedial action schemes, energy and power management, and time-sensitive applications.

An interesting design-time decision is the choice of the communication patterns in a distributed or decentralized system. Even though most of the modern computation platforms embrace Component Based Software Engineering (CBSE) [Heineman and Councill (2001)] and model driven analysis [Kumar et al. (2014)], the choice of communication patterns still remains challenging. Software designers have to evaluate their applications, decide which interaction patterns to use (e.g. asynchronous publish/subscribe or synchronous request/reply, etc.) and then implement the low level mechanisms needed, while considering fault tolerance requirements. Apart from this, designers of distributed systems need to have provisions in place for the well known CAP trade offs [Brewer (2012b)]. The system needs to be able to maintain an acceptable level of performance even in the event of communication loss or node failures. We discuss the motivation for the choices of communication patterns made available in RIAPS and how it helps to achieve fault tolerance, while also exploring possible ways in which the reliability of a distributed algorithm can be improved.

The specific contributions are as follows:

1. We describe the architectural interaction patterns of RIAPS and use these to illustrate the anticipated failure modes.
2. We describe the implementation of the fault tolerance architecture and how it provides layered protection.
3. We describe the choice of communication protocols and how they affect fault tolerance.
4. We demonstrate the design of a distributed load shedding application and show how it can be made fault-tolerant using RIAPS building blocks and patterns.

The contributions were adapted from the published work Ghosh, P., Eisele, S., Dubey, A., Metelko, M., Madari, I., Volgyesi, P., and Karsai, G. (2020). Designing a decentralized fault-tolerant software framework for smart grids and its applications. *Journal of Systems Architecture*, 109:101759.

6.2 Solution approach

6.2.1 Framework Level Fault Management Architecture

The guiding principle in the RIAPS Fault Management Architecture is that there is a clear separation between the application (which implements the application specific functionality) and the framework (which provides and manages the resources needed by the applications). While faults can occur anywhere in the system, there is no single comprehensive fault management solution. The reason is that the ultimate goal of supporting power grid operations necessitates an intricate collaboration between the framework and the application(s) running on it.

In designing the fault management framework, we followed a simple principle: the framework detects anomalies and possibly activates some mitigation functions that are applicable, but it is ultimately the application's responsibility to react to faults and to take a corrective action, as it is the application that *understands* what an anomaly means and how to react to the underlying failure mode. Following this principle, we built several detection mechanisms that detect faults in the system (at least the ones that can be detected by the framework itself). These detection mechanisms are occasionally coupled to default mitigation actions and are always connected to the application itself: the application is always informed about the detected anomalies. Then the application *business logic* can decide the specific mitigation action that needs to be taken and execute it.

The fault management capabilities in an edge computing network for the Smart Grid built with RIAPS nodes are required across three layers: the physical and device level, the platform services level and the application level. On the physical level, we expect that the power grid itself is designed with the $N-1$ power system criterion: Any one physical component (e.g., a breaker, a transformer, a transmission line, etc.) can fail, yet the power system remains operational. We also assume that this principle carries over to the sensors and actuators, i.e. there is sufficient redundancy in the system using the approaches described earlier. Therefore, we focus on the fault management capabilities required at the level of the software platform services and the application. Any physical device failure, including networks and compute nodes, is expected to be *fail stop* and can be successfully detected using the *Watchdog* fault tolerance pattern [Hanmer (2007)]. Although there may be some scenarios that arise in the physical system that make verification of this assumption difficult, the various configurable mechanisms provided by RIAPS can be used by application developers to implement fine-tuned detection and recovery policies to any set of cases. Since the correct handling of such faults is highly application specific, we trust the developer to make the correct decisions.

6.2.1.1 Service Interactions

To describe the fault management architecture it is important to describe the relationship between all the services and actors. The various interactions are shown in Figure 6.1.

The RIAPS Deployment Control node acts as a control center from which applications can be remotely managed. When starting the controller service (`ctrl` in the figure), it is registered with an RPyC registry server. RPyC provides a transparent Python library for simple distributed computing services, such as remote procedure calls to registered services, and is used by RIAPS to facilitate the connection between RIAPS ap-

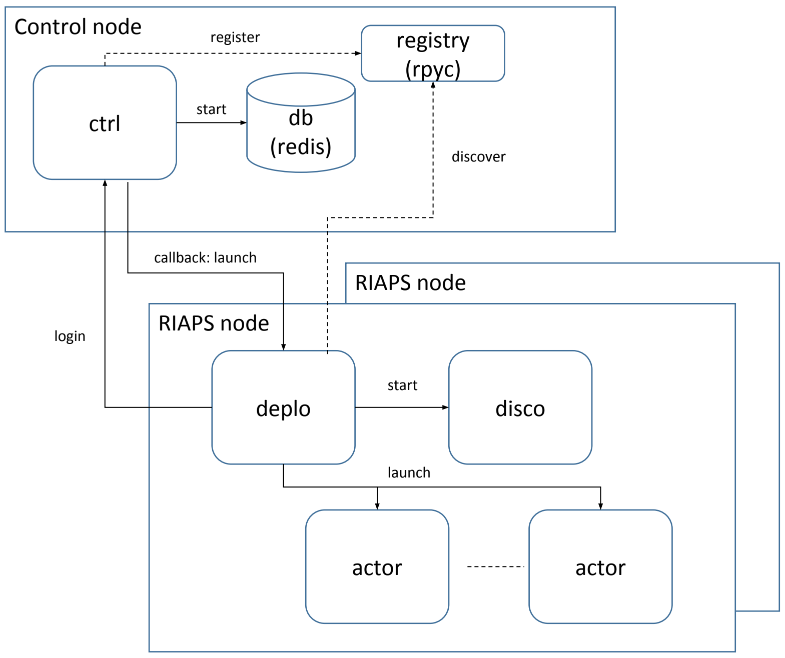


Figure 6.1: The interactions of all RIAPS platform services.

plication nodes and the control node only. Note that RPyC is not sufficient for the requirements of the RIAPS Discovery Service. Hence, this was implemented using ZeroMQ. The Discovery Service is responsible for maintaining the dynamic state of all active peers in a RIAPS cluster. The controller also starts up a local database using Redis [Carlson (2013)], a persistent, in-memory database. When a RIAPS node logs in, it initiates a callback to request the deployment of actors to the deployment service running on the participating RIAPS nodes. When a RIAPS node connects to the control node, it becomes available for the deployment of RIAPS applications. The control node itself does not take part in the application that is running only on the target nodes, autonomously and independently from the control node. Therefore, loss of the control node does not cause loss of application functions.

Each RIAPS node has the Deployment Service (`deplo` in the figure) running. `deplo` uses the RPyC registry to find and connect to the Deployment Control service (`ctrl`). The `deplo` also starts the Discovery Service (`disco` in the figure) on each node. These platform-level services are configured as Linux `systemd` services. Systemd is a Linux service manager that can be configured to start or stop daemons on startup and provide status monitoring and logging capabilities for them. The details regarding `systemd` can be found in [Schroder (2014)].

Once the callback for an application launch is invoked on a RIAPS node, the `deplo` starts the actors for that node. Client actors register with the Discovery Service using a *client/server* socket pair. The service will then create a dedicated socket for the specific client that is used as a private communication channel between a specific client actor and the Discovery Service. Using this channel, each application level service: message publisher and service provider ports are registered, and all application level clients: message subscriber and service receiver ports are connected to their publishers and service providers, respectively. Once this process has completed, the actor starts the associated components in each thread. This sequence of steps is followed in reverse order in the case of termination of the application. Note that the service registration and lookup can happen in any order: a latecomer service (on one node) may trigger the completion of a connection from a client (on another node) after the application has been started. RIAPS provides mechanisms for detecting timeouts on messaging, so applications can be prepared for such contingencies.

6.2.1.2 Platform Level Services

Fault location	Error	Detection	Recovery	Mitigation
RIAPS Services	internal actor exception	framework catches all exceptions	terminate with error/ warm restart	call term handler
	disco stop / exception	deplo detects	deplo (warm) restarts disco	if services OK, upon restart restore local service registrations
	deplo stop	systemd detects	restart deplo	(cold) restart disco / restart local apps
	deplo loses ctrl contact	deplo detects	NIC down ->wait for NIC up; keep trying	
System (OS)	service stop	systemd detects	systemd restarts	clean (cold) state
	kernel panic	kernel watchdog	reboot/restart	deplo restarts last active actors
External I/O	I/O freeze	device actor detects	reset/start HW; device - specific	inform client component
	I/O fault	device actor detects	reset/start HW; device - specific	log, inform client component
HW	CPU HW fault	OS crash	reset/reboot	systemd ->deplo
	Mem fault	OS crash	reboot	systemd ->deplo
	SSD fault	filesystem error	reboot/fsck	systemd ->deplo
Network	NIC disconnect	NIC down		notify actors/call handler
	RIAPS disconnect	framework detects RIAPS p2p loss	keep trying to reconnect	notify actors/call handler ; recv ops should err with timeout, to be handled by app
	DDoS	deplo monitors p2p network performance		notify actors/call handler

Table 6.1: System-level Fault Management Implementation for Platform Services and Hardware

The fault management subsystem at the platform services level is responsible for providing detection and recovery capabilities for the Discovery, Deployment, and Time Synchronization Services [Ghosh et al. (2019)]. Table 6.1 summarizes the various detection and mitigation schemes that RIAPS has incorporated for

Error	Detection	Recovery	Mitigation
Actor termination	deplo detects	(warm) restart of actor	Call handler/ notify peers
Unhandled exception	framework catches all exceptions	if repeated (warm) restart	notify peers about restart
Resource violation	framework detects		Call app resource handler
CPU	soft: cgroups CPU		tune scheduler
	hard: process monitor	if repeated, restart & notify actor/ call handler	
Memory	soft: cgroups memory (low)		notify actor/ call handler
	hard: cgroups memory (critical)	terminate, restart & call termination handler	
Disk	hard: Quota system for files	terminate, restart	call termination handler
Network	hard: Network manager (‘tc’)	if repeated, (warm) restart	notify actor/ call handler
Deadline violation	soft: Component scheduler	if repeated, restart	notify component/ call handler
app freeze	check for thread stopped	terminate, restart actor	notify component/ call cleanup handler/ notify peers restart
app runaway	check for method non-terminating	terminate, restart actor	notify component/ call cleanup handler/ notify peers

Table 6.2: System-level Fault Management Implementation for Applications

platform-service faults. The basic design philosophy of detecting faults in multiple layers and then reacting to them can be clearly seen here. The Discovery and Deployment Services along with OS kernel, work in tandem to prevent faults from propagating from one layer to another. The Discovery and Deployment Services, along with OS kernel, work in synergy to provide mitigation and recovery strategies for these faults, preventing fault propagation from one layer to another.

Applications depend on these services, and therefore it is very important that these services provide a concise and consistent view of the status of the computing resources to all applications running on the platform. For example, the fault management mechanism in RIAPS uses `systemd` services to detect crashes of the Discovery Service. Systemd is also responsible for restarting the service upon failure. The internal protocol used by the Discovery Service, openDHT [Rhea et al. (2005)], ensures the consistency of distributed information.

The fault management capability at the level of applications is present in two places: outside the actor and inside the actor. The fault detection capability outside the actor is responsible for monitoring and recovery of the actor itself, while that inside the actor refers to software exceptions. We describe the specific elements of the fault management architecture below.

- *Resource Constraints*: Table 6.2 provides an overview of the various resource monitoring and resource violation detection and response schemes implemented in RIAPS. It detects violation of memory usage, CPU usage, network usage, and disk usage at the actor level and timing violations at the component level.

  RIAPS uses features provided by Linux. `cgroups` is a kernel feature that allows the setting of restrictions on the memory, CPU and network usage in a collection of processes. These processes make up a control group or `cgroup`. The Linux *traffic controller (tc)* is another useful tool that allows packet-level resource control of TCP and UDP applications. Disk quotas allow system administrators to specify an upper threshold on the maximum size that an application may occupy on the disk. The RIAPS API provides specific handler methods for each of these conditions, which can be overridden in the component code to perform customized operations.

  The resource limits are specified in the model file within an actor block, as shown in Figure 6.2. The actor *LimitActor* has restrictions imposed with respect to CPU usage, Memory footprint, and Disk space using the `uses` block. The first line sets a (hard) CPU utilization limit as a percentage of CPU time over a time interval (1 sec in the example). Violating this hard limit results in an error message being sent to the component. Omitting the `max` keyword sets a soft limit whose violation results in temporarily decreasing the component thread's priority. Details can be found in the `cgroups` documentation [Enterprise (2011)]. The next two lines define the memory footprint and disk space limits, respectively. The last line defines limits on network traffic. Here, `rate` denotes a nominal bandwidth, while `ceiling` defines a hard limit on it. The maximum allowed burst (i.e., the maximum number of *burst* bytes that can be accumulated during idle periods, see [Almesberger et al. (1999)]) is specified by the `burst` keyword. In this example, four separate components are defined in the actor (after the *uses* block) which implements the associated handler for a particular fault.

  RIAPS is also able to enforce time restrictions on specific component port operations and detect these deadline violations. The deadline limit is specified within a component definition in the model file using the `within` keyword, as shown in Figure 6.3. Thus, the timer operation has a completion deadline of 1 millisecond for the component *Sensor*. The specific handler method needs to be implemented by the Sensor component code to define what action to take in case of a violation.

- *Resilient Time Base*: The RIAPS Time Synchronization Service is responsible for maintaining precise timing across the different nodes. It uses a combination of Global Positioning System (GPS), Network

```
actor LimitActor {
 uses {
  cpu max 10 % over 1 sec; // Hard limit, w/o 'max' soft
  mem 200 mb;          // Mem limit
  space 10 mb;         // File space limit
  net rate 10 kbps ceil 12 kbps burst 1.2 k; // Net limit
  }
 {
  cpuLimited: CPULimit;  // Component with CPU usage limit
  memLimited : MemLimit; // Component with memory limit
  spcLimited : SpcLimit; // Component with disk space limit
  netLimited : NetProducerLimit;  // Component with network usage limit
 }
}
```

Figure 6.2: An example of an Actor specification showing the resource monitoring options within the 'uses' block.

```
component Sensor {
    timer clock 1 sec within 1 msec; // Periodic timer to trigger sensor every 1 sec
    pub ready : SensorReady ;
    // Publish port for SensorReady messages
    rep request : ( SensorQuery , SensorValue ) ;
    // Reply port to query the sensor and retrieve its value
    ...
    }
```

Figure 6.3: An example of a component model showing the deadline constraint on a timer port. SensorReady, SensorQuery, and SensorValue are message types. 'clock' is a timer that will fire every second, 'pub' and 'rep' are publisher and reply ports, respectively.

Time Protocol (NTP) and PTP [Mills (1991)] to achieve synchronization. In the clock master node if both GPS and PTP are available, it will dynamically choose the reference with the least variance. If GPS is not available, then the master will use NTP to synchronize its clock. If there is no time reference available, due to GPS failure on the master or network failure on the slaves, the node will continue to use its own clock using the most recent frequency/drift compensation. If the master node is unable to synchronize with its reference due to any failure, the entire LAN will drift from the global time. However, the nodes will remain synchronized.

- *Application Level User identifier*: RIAPS also provides an additional layer of security by providing access control functions through the use of an application level user identifier (unique to the node). The Deployment Service creates a unique, node specific user id before deploying actors on the designated nodes. All actors of an application have the same user id. This identification protocol restricts external agents' access to read and write on the node file system.

- *Watchdogs*: The hardware watchdog monitors on each computing node guarantee that a computing node will be restarted if it suffers from a kernel panic, a kernel crash, or a scheduling fault (i.e. the processes are deadlocked).

- *Process Watchdog*: Within a node, we use *systemd*, a standard service already available on the Linux operating system to manage the health of the most critical RIAPS service on a node, the Deployment Service. It also monitors the states of all other RIAPS platform services. If any service crashes, it will be restarted.

- *Resilient Information Base*: The Discovery Service is responsible for maintaining information regarding all the peers in a cluster, it implements a heartbeat mechanism to maintain current membership. Periodically, the Discovery Service publishes its address and listens for incoming messages. Whenever a message from a new node arrives, it adds the address to its list of known nodes. If it does not receive a message from a known peer for two consecutive time periods, then that node is considered dead, and its corresponding entry is removed. It also maintains the message topic details and services that the actors register with it in a Distributed Hash Table (using OpenDHT [Rhea et al. (2005)]). These values need to be re-registered periodically. If an actor with a registered service crashes, the Discovery Service lets the registered values expire.

- *Time-Sensitive Messaging*: RIAPS leverages the accurate time synchronization service on all nodes to provide precise time stamping to record the times when messages are sent. The time stamp data are added to the payload of the message, and when it is received, the receiving node can use the time difference to calculate the time of flight. Time stamping can be enabled for specified ports from within the model file. This feature can help monitor network latency and potentially identify faults such as bottlenecks, overload, or security attacks such as Denial of Service (DoS).

- *Transaction based Recovery*: The Deployment Service is responsible for maintaining the operational status of individual actors. Thus, in a way, the Deployment Service acts as a (Linux) *systemd* for RIAPS actors. When RIAPS is started, the controller also starts a local database using LMDB [Chu (2011)], a persistent, in-memory database. When an application actor is deployed, the Deployment Service keeps the log of the executed deployment operations in the local database. This information is also replicated on the control node. If an actor fails, then the Deployment Service can reapply the operations from the database. The database is persistent and transactional, which means that it can retain its values even after restarting, and it supports operations like rollback. The application developer can control this default behavior using an application-level configuration policy. If the Deployment Service crashes, it can still retrieve the list of applications currently running from the local database. If the RIAPS node crashes, then the control node can decide to reinstall the actors at the same node or at a different node. The various interactions are shown in Figure 6.1.

- *Application Level Distributed Coordination Services*: RIAPS provides APIs at the application level that support distributed coordination services like group membership, leader election, and consensus among the Components. Groups provide encapsulation, restricting communication so that members only communicate and interact with each other. *Time synchronized coordinated action* is another service provided that coordinates the agreement between distributed nodes about when a coordinated action should be performed.

- *Application level Fault Handlers*: Table 6.2 lists the various fault management schemes used at the application level. The Deployment Service and the Discovery Service work in tandem to monitor the state of the actors during the lifecycle of the application and take corrective steps. For catastrophic failure of an actor, the `deplo` is able to detect it (by monitoring the actor process thread) and perform a restart by reapplying the operations stored in its database. Application developers can customize recovery options using configuration files. This message is sent through a dedicated fault messaging channel. Application developers can define message handlers in their application for such messages.

For internal fault management, the actor is responsible for catching exceptions (indicating anomalies) generated by the component code. Thus, in this case, the responsibility is shared between the application code developer and the framework. The developer will express how the various exceptions are to be handled in the model of the component and the actor.

6.2.1.3 Choice of Communication Patterns

RIAPS supports a wide variety of communication patterns: from anonymous publish-subscribe to peer-to-peer request-reply. All the connection establishment and management are done in a background thread which automatically re-establishes connection upon communication failures. Thus, we leverage most of the communication fault tolerance properties from the underlying framework: ZeroMQ [Hintjens (2013)].

Application developers can use any combination of these patterns in designing dependable distributed applications by using appropriate ports. For example, for applications which require a highly scalable, unidirectional, many-to-many messaging architecture such as sensor readings, a publish-subscribe pattern is more suitable. On the other hand, when strict synchronization is required among agents, such as in a client-server architecture, then a request-reply pattern is more suitable.

The RIAPS framework also utilizes message passing between its different architectural layers to monitor the health of the system, detect and identify possible fault conditions. For example, when a RIAPS node starts, its Deployment Service joins a peer-to-peer network of other RIAPS nodes. When an application is deployed and an application actor starts, its peers within the RIAPS network are informed using the publish-

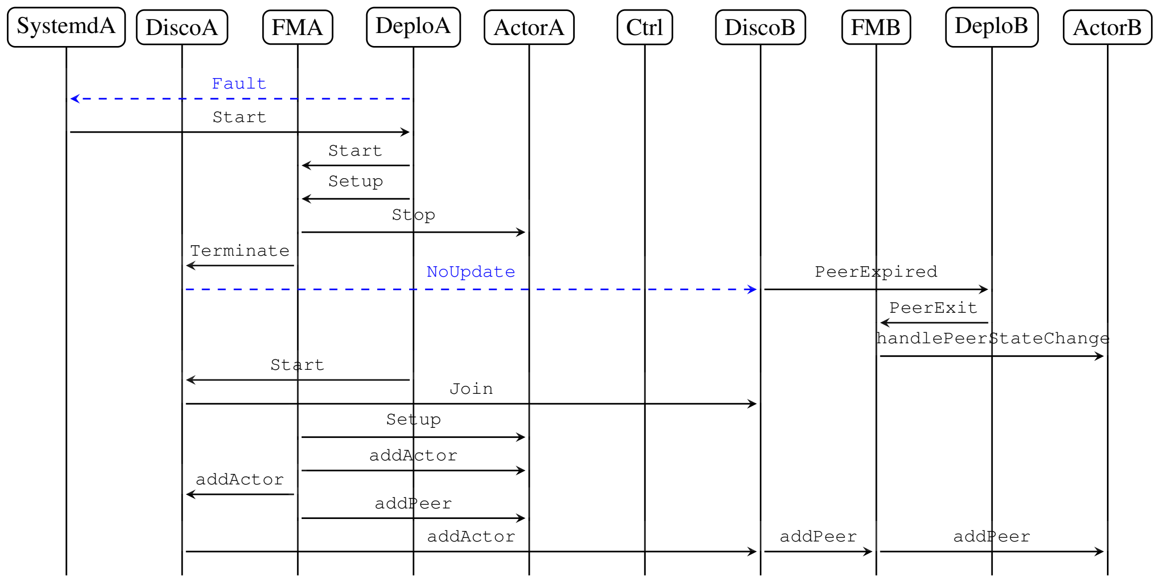


Figure 6.4: Sequence diagram showing the actions that occur when a Deployment Service fails (not all failures are shown). The dashed lines are events that are detected, the first is systemd detecting a process failure, the second is a timeout in the Discovery Service.

subscribe mechanism. If an application actor crashes and/or is restarted, its peer application actors are also informed about the loss and reappearance of the actor.

Beyond these, it is the responsibility of the developer to employ appropriate fault handling schemes in the component code itself by catching exceptions, reacting to timeouts, and other anomalous events, etc. As an example of the usage of communication patterns in recovery mechanisms, we illustrate the messages and events that transpire when a RIAPS node's Deployment Manager fails in Figure 6.4.

The example shows two nodes A and B, and the various agents running in each of them. For each node, the following agents and their communication protocols are depicted:

- *Systemd*: It is the instance of the Linux service that is used to restart any service.
- *Disco*: It is an instance of the RIAPS Discovery Service.
- *FM*: It is an instance of the RIAPS Fault Manager Platform Service that runs on each node.
- *Deplo*: It is an instance of the RIAPS Deployment Manager Service.
- *Actor*: It is the actor process that contains the applications' component instances deployed to each node.
- *Ctrl*: It refers to the RIAPS `ctrl` service that runs on the Control node (this node is separate from the application nodes A and B).

The failure is detected by systemd, and the Deployment Manager is restarted. When restarting, the Deployment Manager restarts the Fault Manager service. The Fault Manager detects processes lingering from the previous instance of the Deployment Manager and cleans them up by stopping any actors and terminating the Discovery Service. Without a Discovery Service, the node is no longer connected to the other RIAPS nodes, and this is detected when an instance of the Discovery Service on a peer does not receive a heartbeat ping during a timeout period. This message is propagated from the peer's Discovery Service to the Deployment Manager, then the Fault Manager, and finally to the actors. After cleaning up the Fault Manager on the failed node, it restarts the Deployment Services, which registers with the other Deployment Service by broadcasting a message indicating its intent to join. This is again propagated through the peer node (not shown). After reinitializing necessary services the Fault Manager performs necessary setup to be able to run the RIAPS application, it then adds the relevant actors, and once they are running, it registers the actor with the Discovery Service. It also notifies the local actor of the presence of the other actors in the network. Finally, the Discovery Service broadcasts that a new actor has joined the system to the other Discovery Service instances, which propagate that information down.

6.2.2 Building a Distributed Fault tolerant Loadshedding Application using RIAPS

The resilience capabilities provided by RIAPS enable developers to design distributed applications with a good degree of reliability and dependability. By taking advantage of some of the features described in Section 6.2.1.2 along with the careful selection of the appropriate communication patterns suited to the particular task, a reliable implementation of an algorithm can be achieved that can recover from faults that the application is specified to withstand.

In this section, we walk through the detailed design decisions and methodology of a distributed load shedding application for power systems that was implemented using RIAPS and simulated in Gridlab-D. Load shedding is a demand control technique employed by utility companies to relieve the grid during peak load conditions, when the total generation is unable to meet the total demand. It involves intentionally shutting down a portion of the distribution network for a period of time until it is safe to reconnect them. It is an important peak shaving maneuver that can prevent overloading of the grid, leading to potential blackouts and complete grid failure. An interesting control problem in this scenario is to determine which loads should be disconnected and in what order to preserve power to critical loads while not jeopardizing grid stability. This will be explored in this experiment through the formulation of a *distributed priority load shedding algorithm*. Gridlab-D [Chassin et al. (2008)] is a simulation engine for power distribution system modeling and analysis developed by the U.S. Department of Energy (DOE) at Pacific Northwest National Laboratory (PNNL). RIAPS acted as the control and communication layer, while Gridlab-D was the simulation testbed. The

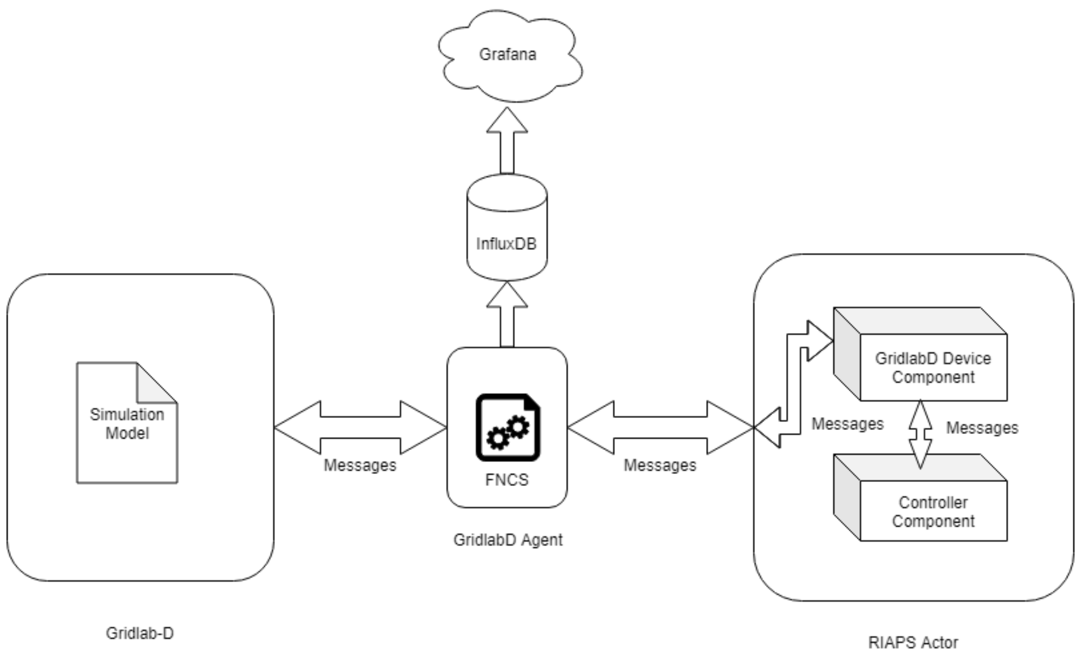


Figure 6.5: Load Shedding Application Interface Design

reasons for choosing Gridlab-D as the power system simulator were its vast library of distribution end-use models, high degree of customization and interoperability with external agents, which will be elaborated later. The overall design architecture is summarized in Figure 6.5.

6.2.2.1 Design

The entire application design can be broken down into four parts:

1. *RIAPS Gridlab-D Interface*: For RIAPS to communicate with the running simulation, the Framework for Network Cosimulation (FNCS) [Ciraci et al. (2014)] package was used. FNCS is a federated cosimulation platform that performs synchronization and coordination between multiple simulations, running as FNCS agents, through a broker process. Gridlab-D can be configured with FNCS using which any system attribute can be subscribed to or modified by other agents in the form of messages. It can also subscribe to values from other agents (RIAPS in our case) that can be used to set or modify network parameters. We implemented a *gridlab-d-agent* program in Python that can run a broker process and launch a Gridlab-D simulation given a model *basename* as an input argument.

2. *Configuration options*: The gridlab-d-agent is also responsible for maintaining the messages to be published/subscribed and the logging of data, using three different types of configuration files. The files have the following naming convention `(basename + {.glm,.gll,.yaml,})`, which are

specific to the particular application. The `.glm` file is the Gridlab-D model definition, the `.gll` file specifies which of the values that are subscribed will be logged and the `.yaml` file specifies configuration options such as the simulation time step and what values are published/subscribed. There is also a file called `gla.yaml` for global settings.

3. *InfluxDB and Grafana Integration*: The gridlab-d-agent can save relevant attribute values in InfluxDB [InfluxData (2017)], a lightweight time series database. These data values can then be visualized using the Grafana [Labs (2014)] tool, which can connect to the InfluxDB instance. The values that will be sent to InfluxDB can be specified in the `.gll` file.

4. *RIAPS Application*: The control commands to the simulation objects would be provided by RIAPS through message passing.

6.2.2.2 Distribution Network

The electrical distribution system on which the application is implemented is based on the IEEE 13-bus feeder network. It is a radial network consisting of a main distribution substation servicing 32 ZIP loads, which can be connected or disconnected from the main grid using 32 switches. Each load also has a meter attached to it to measure its electrical parameters, such as voltage. It is an extension of the setup used in the *loadshed* example of the Transactive Energy Simulation Platform (TESP) project, also developed by PNNL [PNNL (2017)]. But the *Loadshed* example provided by TESP had a much simpler model with a single switch controlling a single load.

The graphical representation of the network model is shown in Figure 6.6. The 32 switches can be controlled to turn the loads on or off from the corresponding external controllers based on some distributed control logic depending on the overall load of the network.

6.2.2.3 RIAPS Model

The RIAPS application model for the *Loadshed* application consists of a *Controller* component and a *GridlabD* device component. The Controller implements the control logic for the switches, while the device component is a specialized component to interface with Gridlab-D through the gridlab-d-agent. It relays published messages to the agent and listens to subscribed topics to receive them. An instance of the Controller and the GridlabD component is deployed within a *ControllerActor*.

The description of the model for the application is given in Figure 6.7. The actor requires three input arguments on launch namely *sensor*, *actuator* and *priority*. The sensor and actuator refer to the network parameter the Controller component subscribes to and the control signal it publishes, respectively. Upon

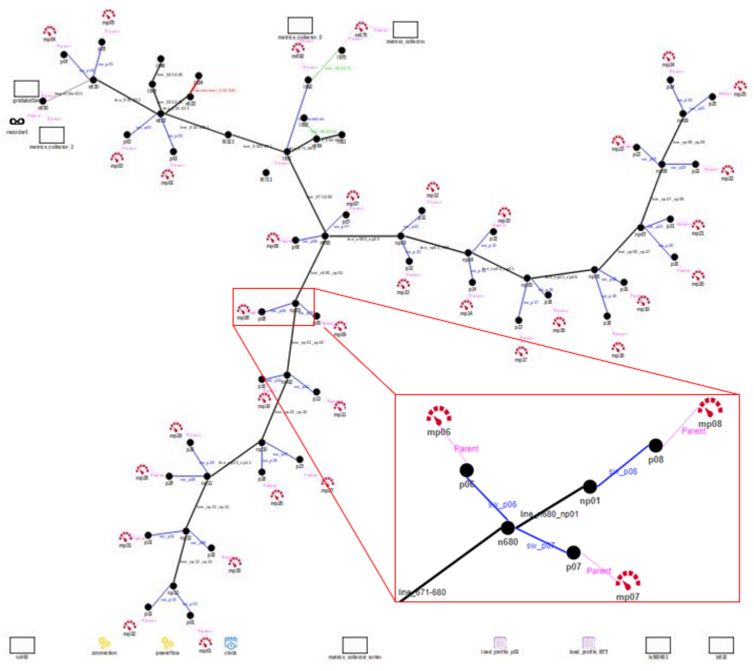


Figure 6.6: The Distribution Network model in Gridlab-D. The black dots indicate loads (names starting with ”l” and ”p”) and the blue lines indicates switches (names starting with ”sw”) that can be controlled using external commands. switch names are matched according to the loads they control. The objects with names ”mp” are meters. Black lines indicate overhead lines.

starting, the Controller sends a subscription request to the device component specifying the *sensor* value. This prepares the GridlabD device component to relay all messages received for the given topic to the requesting component. Similarly, when a Controller component publishes a value, it sends a publish request, specifying the *actuator* name and the value. The *priority* value is used in the control algorithm described in Section 6.2.2.4.

6.2.2.4 Control Algorithm

The principal objective of the load shedding algorithm is to generate a sequence of switch commands to control the loads depending on the total distribution load, as time progresses. Each Controller senses the distribution load periodically and then determines if its controlling switch should be turned on or off. The algorithm is implemented in a distributed manner, where the controllers are not aware of the overall network state, but communicate with each other to compute a control decision.

The control scheme used is a distributed priority-based algorithm. Each load switch, and consequently each Controller component, is assigned a static priority during launch. In a real-world scenario, the priority could reflect the type or criticality of each load. Each Controller maintains a `global_status` list of `{priority: (switch_status, id, UUID of actor)}` which it receives from other controllers

```
app Loadshed {
    message CommandReq;
    message CommandRep;
    message Measurement;
    message Status;
    message NodeState;

    device GridlabD {
     rep command : (CommandReq, CommandRep);
     pub data : Measurement;
     inside relay;
    }

    component Controller (sensor,actuator,priority) {
      timer trigger;
      timer check 5000;
      req command : (CommandReq, CommandRep);
      sub data : Measurement;
      pub updatestatus : Status timed;
      sub receivestatus : Status timed;
      pub sendnodeinfo : NodeState;
      sub recvnodeinfo : NodeState;
    }

  actor ControllerActor(sensor,actuator,priority) {
    local Measurement, CommandReq, CommandRep;
    {
      gridlabd : GridlabD;
      controller : Controller(sensor=sensor,actuator=actuator,priority=priority);
    }
  }
```

Figure 6.7: RIAPS model for the Loadshed app

as published messages. The priority is an integer value; the higher the number, the lower is the priority.

During the simulation, the Controller checks the distribution load, and if it exceeds a specified threshold (which is configurable), then it looks into the `global_status` list to determine if it is the lowest priority load that is currently on. If it is, then it sends an actuation signal to turn off the switch. There is also a lower threshold value and if the distribution load drops below that, then the controller does the opposite search to determine if it is the highest priority switch that is still off, and turns it on. Thus, the order of switching off is from lower to higher priority, while the order of switching on is from higher to lower priority.

The basic steps are summarized in Algorithm 5.

6.3 Evaluation

6.3.1 Fault Tolerance Features

Some of the design considerations and RIAPS features that were incorporated to improve the robustness of the applications are described as follows.

1. *Use of Request-Reply/ timers*: For the communication between the Controller and the device component, request-reply ports were chosen. This is because it is more reliable than simple-publish subscribe for one-to-one messaging, as it provides a means of acknowledgment when the GridlabD device receives a subscription request or a publish message.

Algorithm 5 Controller Switch Status

```
Input: $wait = False$, $switch\_state = 1$
Output: Switch Status (0 or 1)
 1: if Distribution load message received then
 2:     $wait = False$
 3:     if $load > upperthreshold$ and $switch\_status = 1$ then
 4:         for $entry$ in $global\_status$ do
 5:             if $entry.priority > self.priority$ and $entry.switch\_status == 1$ then
 6:                 $wait = True$
 7:                 break
 8:             end if
 9:             if $wait = true$ then
10:                 switch_status = previous status
11:             else$switch\_status = 0$
12:             end if
13:         end for
14:     else
15:         if $load < lowerthreshold$ and $switch\_status = 0$ then
16:             for $entry$ in $global\_status$ do
17:                 if $entry.priority < self.priority$ and $entry.switch\_status == 0$ then
18:                     $wait = True$
19:                     break
20:                 end if
21:             end for
22:             if $wait = true$ then
23:                 $switch\_status = previousstatus$
24:             else
25:                 $switch\_status = 1$
26:             end if
27:         end if
28:     end if
29: end if
```

```
def on_trigger(self):
    ...
    ''' Send subscription request with the sensor object, attribute name and its unit (parsed from the
        sensor argument)'''

    msg = ['sub', (self.sObj,self.sAttr,self.sUnit)]
    ...
    try:
        self.command.send_pyobj(msg)
    except PortError as e:
        self.logger.info("send␣exception␣:␣error␣code␣%s" % e.errno)
        self.trigger.setDelay(30.0)
        self.trigger.launch()
    else:
        ...
        self.trigger.halt()
        ...
```

Figure 6.8: Timer handler method showing how subscription requests are sent

It was vital for the progression of the algorithm that the initial subscription request was delivered once the application started. Thus, a sporadic timer (`trigger`) was launched at start up and the request message was sent to the handler. However, a possible failure mode of the application is if the send operation fails. To account for that, the RIAPS exception library was leveraged in a Python *try-except* block, as shown in Figure 6.8. If it did, the timer was halted. On an exception, the timer was set again.

2. *Monitor node state changes*: In a distributed system, it is important to consider that nodes may drop out or become partitioned due to a network failure. In such a situation, the state of the lost node becomes unpredictable.

   For the Loadshed application, each Controller also maintained a list of nodes and their states. This referred to the internal state of the node itself and was distinct from the status of the switch that the particular node was controlling. The record keeping was achieved through broadcasting the actor *UUID*, the actuator name (quantity it was controlling), and the initial state of the node (on) by all the Controller components. Thus, each controller maintained two different state lists. One was the `node_state` list which consisted of peer node UUIDs, their last known states (on, off, unknown), and the timestamp of the last message received from that node. The other was the `global_status` list which consisted of the corresponding switch statuses (0 or 1) of the neighboring controller nodes, the switch priorities (also an integer) and the UUIDs of the nodes.

   Once any node became unavailable, it was detected by the RIAPS framework, and a corresponding notification was generated which was captured by the Controller to update the node state. This was done using the `handlePeerStateChange()` message handler, invoked by RIAPS.

   In the main switch command algorithm, while iterating through the `global_status` list of priorities,

```
def on_command(self):
    ...
    # send initial node state to all other Controllers
    self.sendnodeinfo.send_pyobj({self.uuid : (self.aObj, 'on')})


def handlePeerStateChange(self, state, uuid):
    # check if notification is generated by itself
    if not uuid == self.uuid:
        self.logger.info("peer␣%s␣state␣changed␣to␣%s" % (uuid, state))
        # check if node already exists in node_state list
        if uuid in self.node_state:
            if not self.node_state[uuid][1] == state:
            # if node state has changed then update the entry
                self.logger.info("Controller␣node␣for␣%s␣is␣%s" % (self.node_state[uuid][0], state))
                self.node_state[uuid] = (self.node_state[uuid][0], state)
        else:
            # If new node is discovered add the entry
            self.logger.info("Discovered␣new␣node␣%s" % uuid)
            self.node_state[uuid] = ("unknown", state)
```

Figure 6.9: Code showing the sending of the node state message and the peer state change handler

the corresponding UUID was looked up in the `node_state` list and checked if that node was on. If it was off, then the current status of that switch was undetermined, and it was ignored in the decision making, until the node state changed to on again. The corresponding code snippet is shown in Figure 6.9.

3. *Use of timed ports*: As shown in the model in Figure 6.7, the publish and subscribe ports `updatestatus` and `receivestatus` respectively were specified as *timed* ports. These ports were used by the Controller to update the `global_status` information of switch states and priorities.

   By using timed ports, it was possible to keep a record of the send time stamps of these messages, which were also stored alongside the entries. Periodically, a timer (`check` in Figure 6.7) was used to search the entries and look for expired entries that exceeded a timeout threshold. These entries indicated a potential network bottleneck or lost packets that made the current state of the node undetermined. As a result, the corresponding switch entries were discarded in the decision making process.

4. *Accounting for actuation*: There is inevitably a short time delay between the time a load is connected/disconnected by sending the appropriate command and when that change is actually reflected in the distribution load reading. To take this into account, the algorithm incorporated a wait for some successive readings before sending out a new switch command. This was achieved by keeping a count of the number of readings using a variable called `retrycount`. This helped bring the system into an equilibrium state each time before the control logic was applied. It also prevented an oscillating condition where switches continually alternate between on and off states, which could lead to instability in the grid.

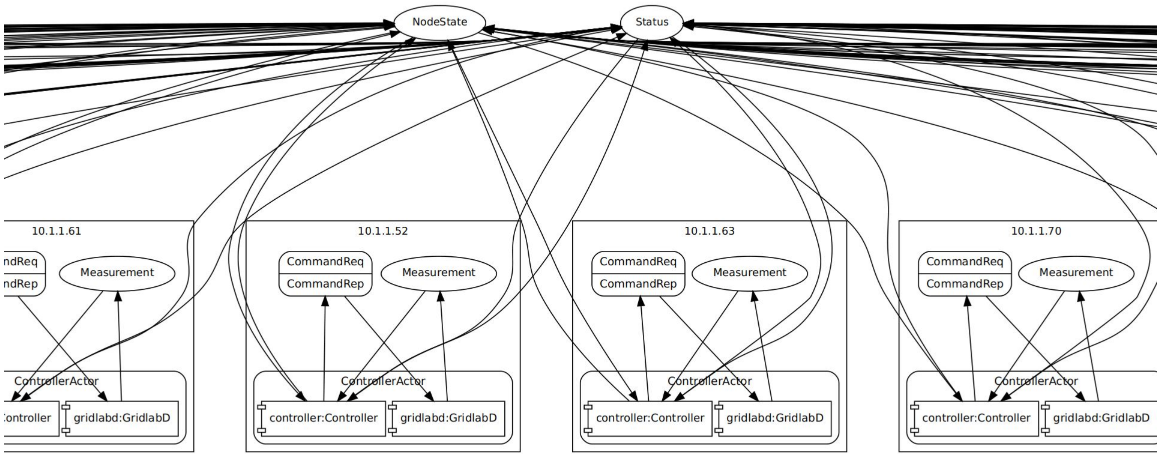


Figure 6.10: Deployment on 32 nodes (shown partially). Each large square represents an actor and the smaller rectangles represent components. Local message types ares shown within the individual actor blocks while global message types are shown outside. Both are represented using ovals (for publish-subscribe) and rounded rectangles (for request-reply).

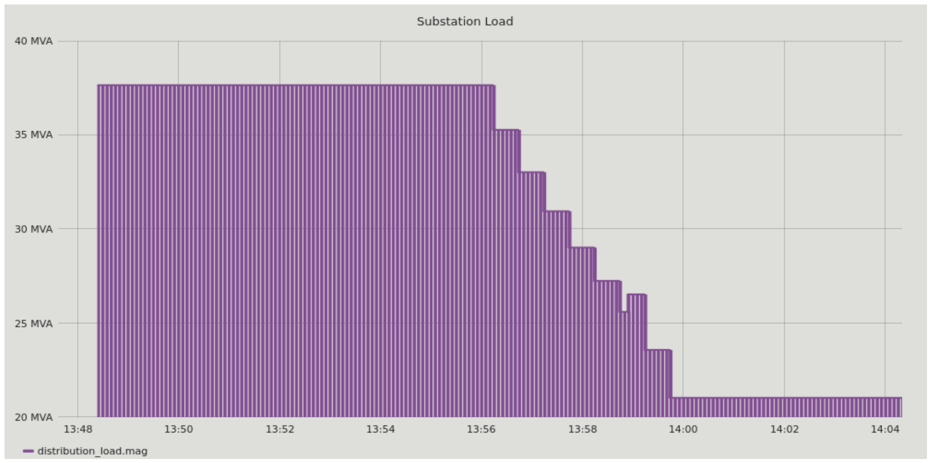
6.3.2 Results

The application was run on a 32 node hardware device cluster with the power grid simulation running on Gridlab-D on a 64-bit Linux Ubuntu 18.04 virtual machine. Each node was running an instance of the actor ControllerActor, as defined in Figure 6.7. The various actor-port-message relationships can be visualized as shown in Figure 6.10.

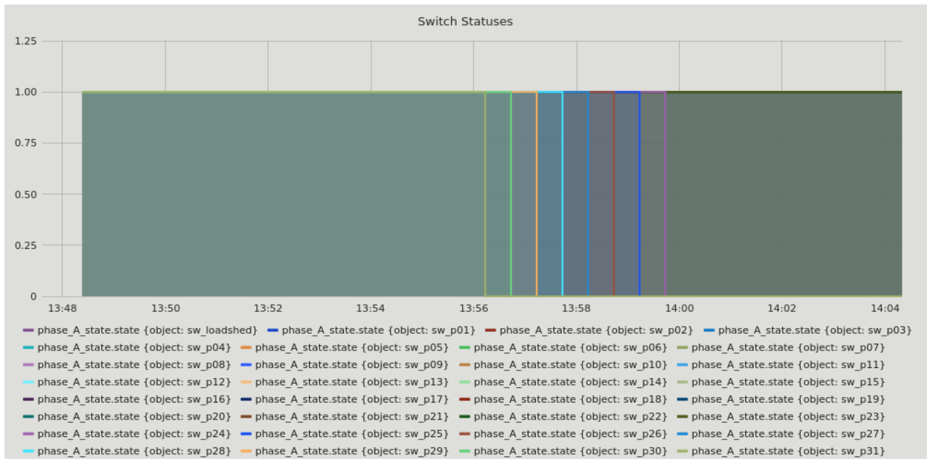
The switch names are given in the order of their priorities. Thus, switch `p_31` has the lowest priority and should be turned off first. The switch statuses are represented by 1 for *on* and 0 for *off*. During the course of the experiment, faults were introduced to some of the nodes to test the resilience properties of the application. The results are shown in Figures 6.11, 6.12 and 6.14.

6.3.2.1 Simulation 1: Timeout Exception

For the first run, the effect of a port timeout on the request port was investigated. The upper and lower threshold values were set to around 22 and 20 MVA respectively. The results are shown in Figure 6.11. When the application was launched, all the Controller components for the switches sent a subscription request to the device component. However, for the lowest priority switch `sw_p31`, the send operation failed with a timeout exception. This was captured through the RIAPS exception library, which showed the error code corresponding to a port timeout. The log entry is shown in 6.11 (c). According to the application logic, the timer was set again to retry the send operation. Meanwhile, since all other switches had higher priority, they waited for `sw_p31` to turn off first. As a result, no loads were disconnected until about *13:56*.



(a) Plot of Distribution Load with time



(b) Plot of Switch Status with time

```
ControllerActor.controller:on_trigger()
ControllerActor.controller:send exception :
    error code 11
```

(c) Send Exception

```
ControllerActor.controller:on_trigger()
GridlabD.GridlabD:on_command(): ['sub', ('n650
    ', 'distribution_load', 'VA')]
GridlabD.GridlabD.GLAClient:run: relay recv =
    ['sub', ('n650', 'distribution_load', 'VA
    ')]
ControllerActor.controller:on_trigger: msg=['
    sub', ('n650', 'distribution_load', 'VA')]
ControllerActor.controller:on_command(): resp
    = ok
```

(d) Retry Successful

Figure 6.11: Simulation 1

During this time, the timer `trigger` for `sw_p31` was fired again and the request message was sent again. This time the operation was successful, as shown by the reply from the device component, which was captured in the log entry shown in Figure 6.11(d).

At *13:56*, switch `sw_p31` was turned off, correspondingly there was a drop in the distribution load value. This was followed by switches `sw_p30`, `sw_p29`, `sw_p28`, `sw_p27`, `sw_p26`, `sw_p25`, `sw_p24` turning off in the order of their priorities. By *14:00*, the distribution load value was within acceptable limits and no further loads were shed.

There was a surge in the load around *13:58*. This was caused by a load schedule specified in the Gridlab-D simulation model, which bumped up all the loads based on the simulation time elapsed.

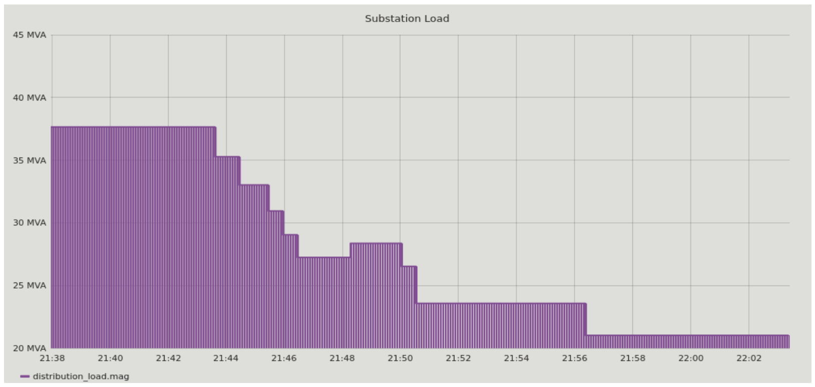
6.3.2.2 Simulation 2: Network Fault

In the second run, the resilience of the application to a fault in the network layer was examined. The threshold values were again set at 22 and 20 MVA, respectively. The results are shown in Figure 6.12.

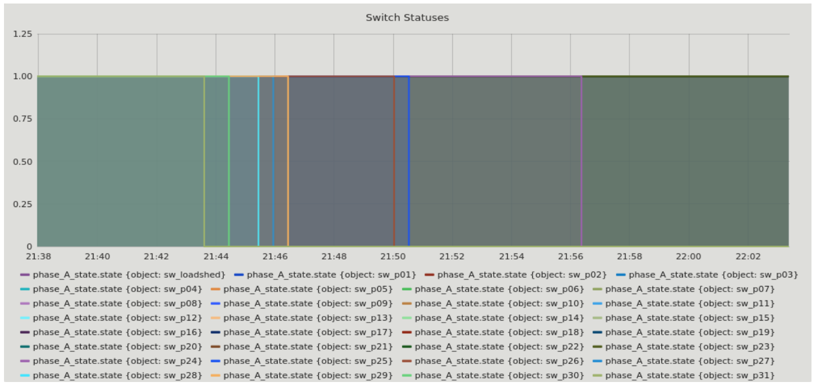
The simulation started with a total load of about 38 MVA, which exceeded the threshold. The first load was disconnected at *21:43*, which was the lowest priority switch `s_p31`. At that time, the network cable for the node corresponding to switch `sw_p29` was unplugged. This was detected by the other Controller components through the peer state change handler method. The log entry for the same is shown in Figure 6.12(c). The next lowest priority switch, `sw_p30` was turned off next, at *21:44*. However, since `sw_p29` was not available, according to the algorithm, it was ignored in the priority sequence. As a result, `sw_p28` was disconnected next, followed by `sw_p27`. During this time, the time difference between the send time stamp recorded for the status message for `sw_p29` (through the use of timed ports) exceeded the time out limit (which is set at ten seconds). Therefore, the entry was considered stale and a corresponding log entry was recorded, as shown in Figure 6.12(d).

At *21:45*, the network cable was reconnected so that the Controller node for `sw_p29` was back online. This was also detected by the other Controllers as shown in Figure 6.12(e). Since then it became the lowest priority switch that was on, `sw_p29` was turned off next at *21:46*. It seemed that there were some messages lost for the nodes for `sw_p25` and `sw_p24` due to which a larger interval was observed between their disconnection times, as they were waiting to receive status messages from all their peers. Finally, around *21:56*, the distribution load was reduced to the required level and the network stabilized. The sequence of events is shown in Figure 6.13.

The increase in load observed around *21:48* was due to the load schedule kicking in.



(a) Plot of Distribution Load with time



(b) Plot of Switch Status with time

```
ControllerActor.controller:peer
    B11433D7D9D02F31DC3AB51B30163769 state
    changed to off
ControllerActor.controller:Controller node for
     sw_p29 is off
```

(c) Peer state change handler: off

```
ControllerActor.controller:checking for stale
    values
ControllerActor.controller:entry 30:('1', '
    B11433D7D9D02F31DC3AB51B30163769',
    1564798045.844169) is stale
```

(d) Status message timeout

```
ControllerActor.controller:peer
    B11433D7D9D02F31DC3AB51B30163769 state
    changed to on
ControllerActor.controller:Controller node for
     sw_p29 is on
```

(e) Peer state change handler: on

Figure 6.12: Simulation 2

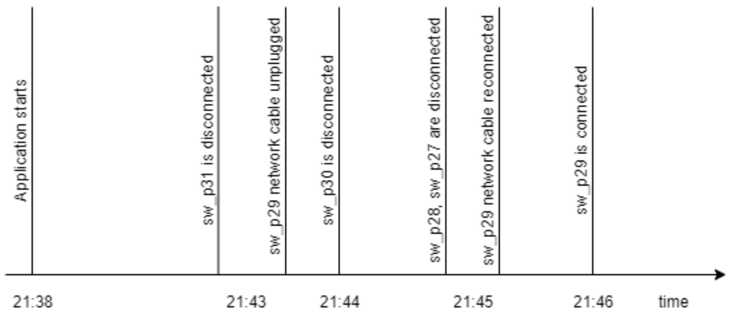


Figure 6.13: Sequence Diagram showing the important events for Simulation 2

6.3.2.3 Simulation 3: Hardware Fault

For the final run, the ability of the algorithm to recover from a hardware fault such as loss of power was tested. Along with it, the reconnection phase for disconnected loads was also demonstrated. At the beginning, the upper threshold value of the magnitude of the substation load was set at around 34 MVA and the lower threshold at around 32 MVA. At the beginning of the test, the load exceeded the threshold, and hence the load shedding logic was initiated. However, the power was unplugged for the node controlling `p_31`. This was detected by the peer state change handler method. The corresponding log entry is shown in Figure 6.14(c).

As a result, it was not able to participate in the message sharing process and the next lowest priority switch `p_30` was turned off first. Subsequently, the switch `p_29` was also turned off, as expected.

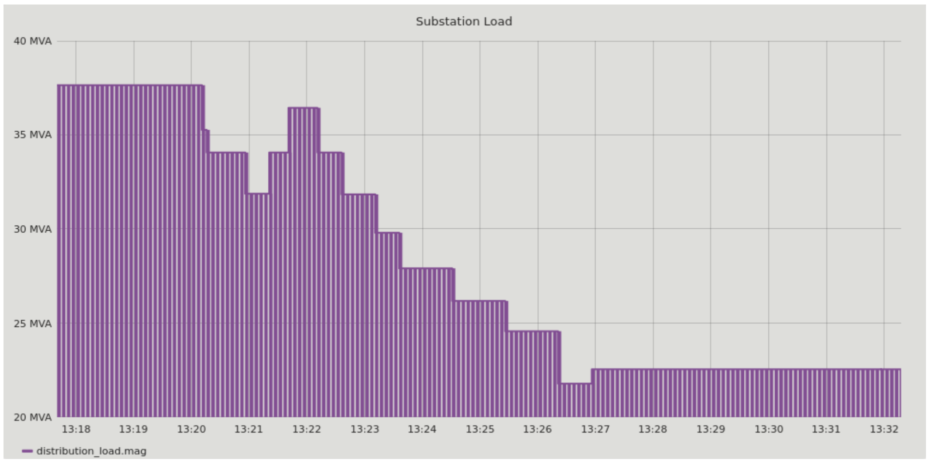
Once the total load fell below the lower threshold, the loads were progressively reconnected. Thus, switches `p_29` and `p_30` were turned on in order. At *13:21*, the node corresponding to `p_31` was powered on. This was also detected by the other Controller components as shown by the application log message in Figure 6.14(d).

The threshold values were then changed to around 22 and 20 MVA respectively. This was done to show the recovery of the application from the previous fault. This again began the load shedding cycle. However, this time the application was able to disconnect loads in their proper order since all nodes were in a healthy state. Thus switches `p_31`,`p_30`, `p_29`, `p_28`, `p_27` and `p_26` were turned off in sequence until the total load was within the specified thresholds. The sequence of events is shown in Figure 6.15.

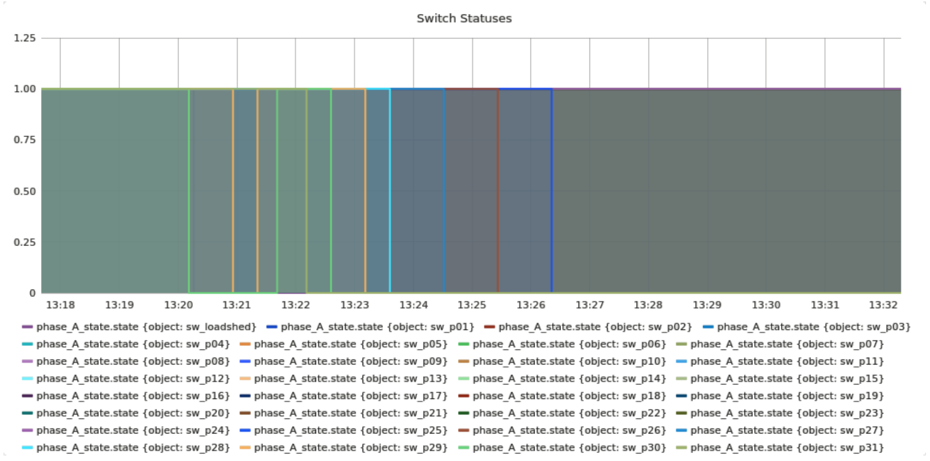
Thus, in all three cases, the application control logic, coupled with RIAPS features, was able to perform acceptably in the event of a fault scenario and return to its original state once the fault was mitigated.

6.4 Summary

In this chapter, we introduce the fault management subsystem of the RIAPS framework and demonstrate its use in a distributed energy application. In the design of the fault management subsystem, a systematic



(a) Plot of Distribution Load with time



(b) Plot of Switch Status with time

```
ControllerActor.controller:peer
    AC5975D70922DF498FEC3587F0F796E9 state
    changed to off
ControllerActor.controller:Controller node for
     sw_p31 is off
```

(c) Peer state change handler: off

```
ControllerActor.controller:peer
    AC5975D70922DF498FEC3587F0F796E9 state
    changed to on
ControllerActor.controller:Controller node for
     sw_p31 is on
```

(d) Peer state change handler: on

Figure 6.14: Simulation 3

approach was followed in which the possible failure modes of the framework were identified by considering the interaction patterns across the RIAPS architectural layers. The choice of specific services and their communication protocols was examined in terms of their role in improving the resilience properties of the

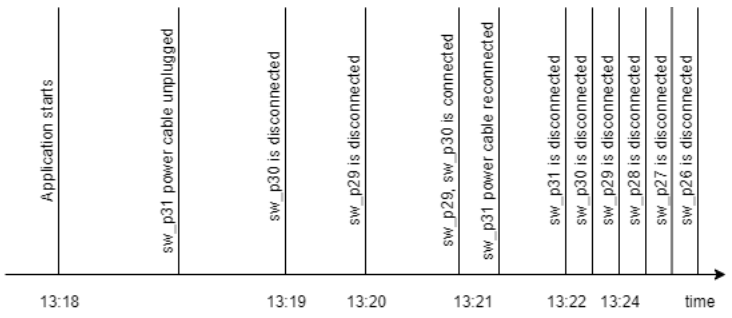


Figure 6.15: Sequence Diagram showing the important events for Simulation 3

system and implemented accordingly.

Finally, a walkthrough of the design process of a RIAPS application was provided with an emphasis on making it fault-tolerant. With the proper usage of the RIAPS library features such as timed ports and peer state change handlers, which accompany the main distributed control logic, the application was shown to function in the event of various types of faults. It provides a useful overview of the various pitfalls that must be kept in mind while developing complex distributed applications that are catered to business critical systems such as the power grid.

CHAPTER 7

Conclusions and Future Work

This dissertation covered the topic of fault tolerance design approaches in the context of distributed systems that cover cyber physical requirements. It tried to look at fault tolerance not just as a passive corollary to building CPS but as an essential part of the system design process. The synergy between the underlying physical processes that govern a system and the computational layer is important in designing fault-tolerant CPSs. There are two principally complementary philosophies for looking at fault tolerance integration into DCPS, *proactive* and *reactive*. Certain decisions can be taken at design time to proactively prepare the system against anticipated anomalies in the future. However, as we go higher up the fault management pyramid, certain provisions require adapting the system configuration at runtime that depend on the application domain. With the transition from decentralized to distributed architectures, it must be emphasized that the communication system also plays an important role in DCPS, and its robustness must be considered essential to system design.

The principal contributions have been in four different aspects of CPS design that might be encountered during the design life cycle itself. Two of them can be considered proactive while the other two can be considered reactive. Chapter 3 introduces a way to automatically generate timed automata models from DCPS source code in Python. It combines information about the application model, its deployment architecture, and custom model parameters given by users to produce a suitable abstract representation of the application components and interaction patterns that can then be used to perform various model-based analysis and validation in UPPAAL. Thus, this can be a good first step in analyzing the robustness of the system being developed as to how it stands against some safety and performance specifications.

Once the system deployment architecture is finalized, a designer can look to optimize the deployment portfolio in terms of its resilience, resource requirements, and costs and then test the resulting application against generated faults. Chapter 4 introduces a resilient deployment solver that can calculate an optimal deployment setup taking into account all those factors and a runtime testbed for evaluating the solver deployment configurations. it requires no change to the internal application, and users can customize requirements through specialized deployment specifications and hardware specifications DSL. The solver solutions can then be put through their paces using the virtual network test bed with the help of *Mininet*. The state of the virtual nodes and the network links can be controlled by using a behavior model that can introduce failure conditions at specific time intervals to evaluate how the deployment can compensate and keep the system operational.

The next aspect that could be looked at is how the distributed entities will communicate with each other to realize their designated function. The choice could be to try to choose a fully connected communication network with fast convergence, but this might not be supported by the available network hardware. On the other hand, going for a gossip-like sparse communication strategy might pay off in the long run, but it might cause the control decisions to be out of time. This trade-off is addressed in Chapter 5 by introducing a configurable framework that can realize any communication topology given a graph description in a simple language, and also a fault-resistant dynamic peer selection algorithm, which can improve the convergence efficiency and speed of distributed algorithms.

Finally, once all the application design and the deployment and communication architecture is finalized, the final step is to incorporate those fault tolerance features that are application-specific. This is the apex of the fault management pyramid (Figure 1.2), and involves carefully considering the services provided by the underlying platforms as well as the physical requirements of the domain in question. The final solution is then a holistically engineered combination of a robust system-level architecture as well as integrated fault-tolerant application logic that work seamlessly in tandem to realize the ultimate goal: prevent the loss of life and property even if everything is not going according to plan.

Fault tolerance in DCPS is a constantly evolving field with interesting research possibilities, and this dissertation is a sample of that. There are many open possibilities and scopes of improvement for the work discussed as part of this dissertation. Those include, scalability experiments and feasibility studies on large-scale systems, improving the TA models to capture time more effectively, optimizing the virtual topology to always have the minimum number of edges, adding explainability features in the deployment solver to allow users to understand which constraints failed, etc.

7.1 Generalized Properties of Fault Tolerant Distributed Cyber Physical Systems

The goal of this dissertation was to highlight the properties and design principles for incorporating fault tolerance properties in DCPS from a systems perspective, while using Smart Grids as an example. The idea here was that even though the applications and case studies were related to power grids, the general principles are applicable to any class of DCPS from any application domain. Of course, these will be subject to some assumptions about the underlying systems, and the research attempts to bring them out and generalize them over the course of addressing the specific research challenges explored in the earlier chapters.

1. *Redundancy in Hardware and Software:* Redundancy is one of the fundamental pillars of ensuring good fault tolerance performance in large-scale systems and minimizing downtimes. Therefore, for any good CPS design, it is important to overprovision resources where possible. $N+1$ redundancy is a system design principle which states that there should be an additional component available for any

part of a system that can be replaced in case of a failure. The 1 in $N+1$ can be replaced by any number to make the system more reliable and capable of handling more concurrent failures, but it would come at a significantly higher installation and maintenance cost. However, that might still be a requirement for highly critical systems, such as aircrafts, where a $N+2$ redundancy scheme can often be employed [Downer (2009)].

DCPS would require redundancy across three layers: *physical*, *computational* and *communication*. At the physical and device level, it would mean having copies of sensor components such as accelerometers, voltage or current meters placed at different points within the system, backups for the controllable entities within the system such as actuators or circuit breakers, as well as redundant interconnection pathways for e.g., data centers being connected to the mains and a backup generator through independent electrical lines. These features are dependent on the domain of the system, as well as the control objective. Redundancy in the computational layer can refer to multiple copies of a software process running on different computers, or to redundancy being built into a single computer. The former would utilize distributed nodes to run multiple copies of software such that one can replace the other in the event of a failure. The latter would utilize different data channels, CPU cores or software from different manufacturers within the same computer. Redundancy in communication can be achieved by establishing alternate network paths by having distributed nodes connected using primary and backup network interfaces. Communication redundancy can also be achieved by choosing appropriate network resources with adequate buffer in terms of bandwidth available or a suitable network topology that keeps the message traversal times low.

2. *Separation between Application and Infrastructure:* To ensure effective fault management, it is important to have a distinct division between the application responsible for carrying out its specific functions and the framework that provides and oversees the resources required by applications. This provides business users with a level of abstraction that allows them to come up with the application specific fault tolerance features without having to worry about the complexity of the underlying layers. Therefore, a power system engineer can design the power sharing strategy without trying to also solve the problem of a network link disconnection, which can be left to network engineers. By separating the application and framework layers, it becomes easier to identify and resolve faults or errors in either layer without affecting the other layer. This can help reduce downtime and ensure better system reliability. This allows applications to be more generalizable by not tying them to a specific infrastructure, and for the control algorithms to be easily ported to different systems without having to reimplement the platform specific fault management features from scratch. Scalability is another important factor in

fault-tolerant **DCPS!** (**DCPS!**) design. Separation between the application and infrastructure makes it easier to scale either layer independently, depending on the specific needs of the system. It can simplify maintenance tasks, making it easier to update and manage the system over time.

3. *Modularity:* Modular design is a direct consequence of the previous point. However, it is listed here as a separate consideration since modular design must permeate both the application algorithm and the platform architecture for a robust DCPS. An effective fault-tolerant system has to ensure that faults from the layer above cannot propagate to a fault in the layer below. And the layer above can assume certain behavior about the layer below, and if that fails, the layer below should inform the layer above. This is the principle of *fault containment*, which means that if a fault occurs in one module, it can be isolated and contained, without affecting the rest of the system. The system can continue to operate normally while the faulty module is repaired or replaced. CPS that use a modular design can recover more quickly from failures or errors, as individual modules can be designed to automatically recover.

   In terms of algorithm design, model-based approaches can be used to divide the algorithm functions into interdependent, communicating modules distributed across the network. A good assumption for the developed software modules in that case is that they should be based on the *Single Responsibility* principle of object oriented software design [Martin (2003)]. It states that a class or module should only have one primary function. In DCPS design, this principle can be followed by identifying and implementing the various functions of an application as distributed components. This would lead to a greater number of components in the application architecture model, but each individual component will be simpler and smaller in scope. This would in turn simplify their implementation, while also making it easier to identify the source of observed anomalies and also to debug.

4. *Services over Products:*

   In order to run safety critical DCPS applications in real time or close to real time, previously it was necessary to run compact code optimized to run on particular embedded platforms. This led to a proliferation of specialized *products* built by the manufacturers themselves that combined the two aspects. With the advent of better and more capable electronics such as processors, high speed memory and faster network communication, it has become possible for the design of hardware and software to be separated, thus leading to a more *service-oriented* approach. A key difference between a service-based approach over a product-based one is that the former allows the owner/user to add or update its functionality, without requiring assistance or permission from the manufacturer. In a service-oriented architecture, specialized features or functions are made available to end users in the form of services, which can then be invoked at the application level depending on the particular use case. Services can be *synchronous* or *asynchronous* and provide specific interfaces that applications can use to take advantage of their features.

Industry and academia have explored the development of open application platforms that provide the flexibility and interoperability of service-oriented architectures. The increasingly popular trend of moving data access and processing to the edge of the network has increased the adoption of such service-based open application platforms. Allowing local applications to run on edge devices while utilizing the different services provided by the underlying platform leads to a more dependable design by reducing network communication times, improving response times leading to faster actuation, and improving reliability by reducing dependencies [EPRI (2014)]. Such platforms can employ a hierarchical model in which the platform services are organized in layers with clearly defined responsibilities and information boundaries. These services can be of two categories depending on their responsibility. *Monitor* services keep track of different parts of the system or other services and dynamically discover changes to the system configuration. They can either notify the application in case of a fault in that part or perform some mitigating action based on a preconfigured policy. *Detector* services are triggered by events that violate certain conditions within the system or faults and provide *handlers* to which the application can bind to generate the appropriate response. Services themselves also need to be fault tolerant, and this can be achieved by taking advantage of the features of the target platform operating system and combining them with well-known fault-tolerant software design patterns [Hanmer (2013)].

Although services offer numerous benefits, there are certain scenarios where implementing system functionalities as services may not be the most suitable approach. This is especially true for tightly coupled and performance-critical systems that require optimized execution. In such cases, the additional overhead and communication layers introduced by services may hinder performance, and a more tightly integrated or low-latency architecture may be preferred.

5. *Diagnostics and Maintainability:*

   One of the challenges in large scale DCPS is monitoring and diagnosing the health of the system and, subsequently, identifying the source of any faults that occur. Although the application architecture is distributed, it is desirable to have a centralized management hub that can allow users to deploy the application processes on the target edge nodes, monitor their health in terms of metrics such as resource usage, status of the services running on them, as well as remotely stop or restart the services. It is also desirable for system administrators to have diagnostic services in terms of system log collection and retrieval from a central console. From an application developer's perspective, it is important for the different modules within the system to be coded in a manner that it can provide propagate information through the layers above regarding the cause of the failure through multilevel log messages and appropriate exception classes.

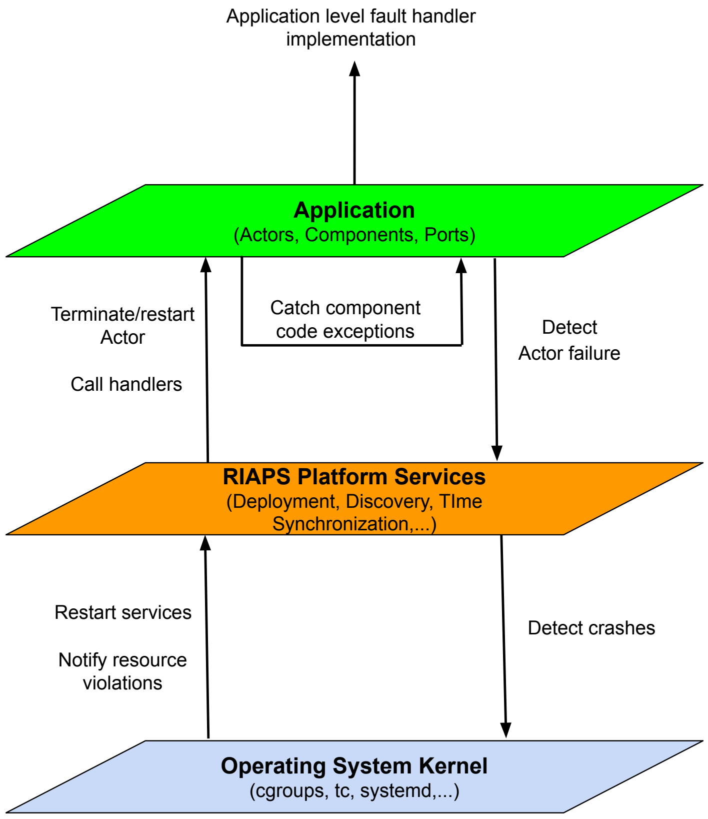

Application level fault handler implementation
Application
(Actors, Components, Ports)
Terminate/restart Actor
Call handlers
Catch component code exceptions
Detect Actor failure
RIAPS Platform Services
(Deployment, Discovery, TIme Synchronization,...)
Restart services
Notify resource violations
Detect crashes
Operating System Kernel
(cgroups, tc, systemd,...)

Figure 7.1: Fault Management Layers in RIAPS

Looking through the four research contributions covered in this dissertation, it is evident that building CPS architectures on top of an open, decentralized middleware platform such as RIAPS equips the system with most of the general properties that are essential for fault tolerance. A RIAPS application is realized through distributed components wrapped in actors that allow the application functionality to be modularized among them. The RIAPS fault management architecture is comprised of several layers that take advantage of several classic fault detection and recovery patterns to design fault-tolerant systems and integrate it together, building on a notion of fault-tolerant synchronized time base and strong resource constraints. These services work in synergy with the operating system kernel, communication middleware, and the application-level actor processes to provide mitigation and recovery strategies and prevent fault propagation from one layer to another. This is different from an assume-guarantee reasoning [Henzinger et al. (2001)], wherein a component is verified (correctness guaranteed) assuming the correctness of its constituent components. In this case, the lower level assumptions are not used to infer anything about the upper layer, but rather it acts in a supervisory role, detecting and reporting faults in the lower layers using well defined interfaces, as shown in Figure 7.1. RIAPS also provides application-level fault handlers that can be used to define the application-specific behavior. RIAPS application deployments can be controlled from a control node through a dedicated GUI with the help of the Deployment and Discovery Services. It also consists of a customizable logging framework that can create log entries dedicated to framework, actor, or component-level logging and write them on to different targets such as files or a dedicated log server.

However, the general methods and techniques described in the different contributions are generic and do not relate to any specific implementation. Thus, in theory, any system can employ the principles of fault management used in any of the case studies and apply them to a suitable application context, provided the underlying properties are met. Of course, developing everything from the ground up would require significant effort and in some cases might be akin to *reinventing the wheel*. Ultimately, it is up to system architects and application engineers to determine the optimal design template and subset of features that are best suited for their specifications.

7.2 Future Research Topics

As energy demands for the future transition away from conventional sources of energy to predominantly renewable energy and from traditional distribution grids to more microgrids, it will inevitably lead to more interesting research questions and perspectives that need to be explored.

Significant developments have already been made in the installation of 100% renewable microgrids that rely entirely on local generation and storage [RCAM (2022)]. To serve a larger consumer base, such efforts must be scaled up and accelerated. This can only be possible if individual microgrids pool their resources together and interact to form *multi-microgrids* [Tao and Schwaegerl (2009)]. Several Low Voltage (LV) distribution microgrids and Distributed Generator (DG) units are connected through Medium Voltage (MV) feeders at the MV level to form a multi-microgrid. Therefore, at the MV level, the individual microgrids act as energy sources or sinks that can be controlled and regulated. A simple one line diagram representation of a multi-microgrid is shown in Figure 7.2.In order to fully understand and then exploit the possibilities of multi-microgrids, new techniques need to be developed to study, manage, and control them. This would also involve adapting the concepts described in this dissertation to be applicable to networks of microgrids instead of a single entity. The associated research challenges can be broadly categorized into three main domain areas.

1. *Enhanced modeling and simulation capabilities*: A principal challenge in scaling modeling and simulation approaches to multi-microgirds is the increased dimension and complexity. Given that the size

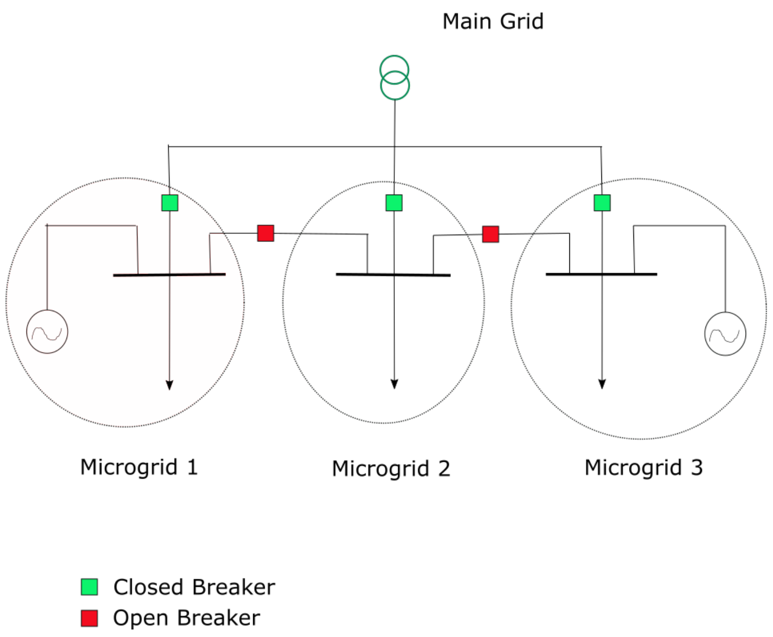


Figure 7.2: Multi-microgrid arrangement (all elements not shown)

of such systems can include several hundred to a thousand buses, it is impractical to model and simulate each element within each of the participant microgrids with high fidelity. Thus, new modeling and simulation approaches need to be investigated that will allow the full model representation of such systems that can operate at close to real time scale and be allowed to interface with real-time control algorithms. This necessitates further research into integrated software frameworks that can support such large-scale simulations. An approach would be to distribute the simulation burden among multiple simulation agents or *federates* which are responsible for a single microgrid and then use some kind of cosimulation orchestrator to coordinate and control the simulation, a concept called *federated simulation* [Vélez-Rivera et al. (2018), Palmintier et al. (2017)]. The challenges that the software must be able to solve would then involve managing and synchronizing the individual simulation federates such that the simulation logical time steps with the physical time that an external controller would be expected to follow. The approach can be thought of as an extension of the TESP based simulation framework discussed in Chapter 6 where the agent is responsible for multiple Gridlab-D simulations. However, with more heterogeneity introduced into the system with multiple simulation models running at different granularities, this becomes an interesting problem. Figuring out the communication between the federates is also important considering that at each step, external information from both controllers and other simulation models will feed into each other to advance the composite system. This would involve designing suitable communication interfaces either using ZeroMQ like middleware or coming up with customized ones, deciding on the data structures and modes of message transfers and how the simulations can tolerate faults in the messaging layer such as loss of messages or delays. The second direction would be to adopt a decoupled modeling style where some of the elements are modeled in detail, while for others some approximate model is used. Model-based system design tools like SystemC AMS [Chen et al. (2020)] have been claimed to significantly speed up simulation times. It can also be possible to emulate an entire microgrid with the help of Artificial Neural Network (ANN)s that can replicate the steady-state behavior of the LV network, while incurring significantly less computational overhead as well as time [Lopez-Garcia et al. (2020)]. The principal challenge would then be to determine a suitable data-driven model that can emulate the behavior of a microgrid system, what are the major input variables that it needs to consider, and how to train such models such that they resemble the behavior of an actual system with the desired level of accuracy.

2. *Sophisticated control algorithms*: Existing control architectures, as well as EMS also need rethinking to be applicable to multi-microgrid systems. To effectively manage such a system, a hierarchical control architecture, with the addition of another control layer at the MV level that can control the participant Microgrid Central Controller (MGCC)s might be required [Kaur et al. (2016)]. The flow of control in this case should be from the top to bottom with the higher level controllers determining rules and actions that would then be delegated to the lower level controllers. Similarly to a traditional microgrid, multi-microgrids would also operate in two modes - a normal *grid-connected* mode and an emergency or *islanded* mode. All existing control algorithms applicable for local microgrids related to energy balance, voltage and frequency control, protection, and transition from one operating mode to another would still be applicable at these levels. The key advantage of using software-based control in such a scenario would be in its modularity and any research on software pipelines for implementing multi-microgrid control algorithms must consider how it can be achieved. Modularity implies that the same control logic or algorithm can be applied to different physical microgrid models with minimal changes to the supporting software, in a true *plug and play* style architecture. The associated research challenges are to determine the interface layer that can act between the application layer and the physical electrical system layer. Instead of delving straight into concrete software development, a better approach would be to look at developing *meta models* [Maróti et al. (2014)] for energy systems software that defines the different layers and interfaces, how they are related to each other, and what are the rules for derived concrete implementations of software systems from the meta model depending on the multi-microgrid configurations [Fkaier et al. (2020)].

   However, an added complexity needs to be accounted for regarding the inter-microgrid interactions.

Building fault-tolerant control algorithms for multi-microgrids would also need to consider the operating mode and the level of involvement of the top level controller. An interesting problem in this case is microgrid *reconfiguration* for energy demand or load sharing [Thakar et al. (2019)]. Following a fault or an increased energy consumption demand in islanded mode, it is imperative that a microgrid is able to provide at least the critical loads with uninterrupted power. A multi-microgrid allows greater flexibility in that case in sharing the increased demand by altering the network topology by opening and closing relevant breakers and switches, thereby allowing those loads to be served by adjacent microgrids. This process is known as *reconfiguration*. Hierarchical algorithms are essential in this regard to redirect power flow to the remaining unaffected loads on the grid and to determine the power sharing arrangement between the microgrids. Taking reconfiguration as an optimization problem, the objective is to maximize the supplied load power or minimize the load curtailment [Shariatzadeh et al. (2014), Shariatzadeh et al. (2011)]. Recently, **RL!** has also been adopted to solve the problem of network optimization effectively. RL is a machine learning technique that uses incentives and rewards to learn the optimal control algorithm, and scales very well with added complexity. It also eliminates the need to rerun the optimization algorithm for every new event once the controller has been sufficiently trained [Bui and Su (2022), Rahman et al. (2022)]. Thus, it would involve contributions from both electrical engineers and machine learning experts. The main challenges with respect to algorithm design would include appropriate modeling of the multi-microgrid network graphs that would allow the optimization problem to be formulated as a graph-forming problem. Careful consideration should also be given to methods of detecting an unplanned islanding or fault event such that the reconfiguration process can be triggered. In the case of the learning-based approaches the main challenge is data generation that sufficiently covers all possible scenarios that are most likely to be encountered in a real world setting. This might be difficult to obtain from actual microgrid installations since fault events are quite rare and therefore may involve the use of high-fidelity simulation techniques as described above or the need for research on data-efficient system design [Renganathan et al. (2021)].

3. *Real time communication*: The deployment of an effective communication infrastructure that can facilitate complete *observability* and management of the multi-microgrid system is also essential. The increased adoption of decentralized, interoperable, and resilient frameworks such as RIAPS and leveraging the platform services they provide can help design a comprehensive and robust CPS architecture allowing deeper levels of coordination and management. The increased complexity of multi-microgrids would necessitate additional networking capabilities in order to aid data collection from sensors and metering devices, control decisions and commands between upper and lower level controllers and dis-

patch of control output commands to the local generation and consumption device components. To implement real-time control and monitoring, specialized hardware and software must be incorporated that support real-time communication as well. Having tightly bound end-to-end temporal delays in network communication enables the platform layer to detect network-level failures or performance drops more readily and equip the application layer to react and recover from them. On the hardware side, specialized standards and protocols such as Time Sensitive Networking (TSN) support real time application development by providing guaranteed upper bounds on end-to-end latency, extremely low packet loss due to hardware failure and congestion [Finn (2018)]. These are achieved using specialized routers, bridges, and network switches that are time-synchronized using PTP [IEEE (2008)] and prenegotiated contracts between the transmitter and the network regarding the transmission of duplicate messages over multiple network paths between the source and destination. Building support for TSN in distributed applications in platforms like RIAPS would enable real-time communication and control in multi-microgrid systems resulting in faster stimulus-response times and more consistent, scalable performance. On the software side, the extension of novel messaging techniques such as the one described in Chapter 5 and the employing real-time messaging services on top of the transport layer protocols [Kim et al. (2011)] into the middleware architectures for multi-microgrids could be an interesting case study. Another important consideration could be incorporating these time-bound network models into the model-based analysis of DCPS such as in Chapter 3 and how their incorporation can improve the verification of the generated TA models.

CHAPTER 8

Publications

8.1 First Author Publications

- Ghosh, P., Eisele, S., Dubey, A., Metelko, M., Madari, I., Volgyesi, P., and Karsai, G. (2019). On the design of fault-tolerance in a decentralized software platform for power systems. In *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 52–60. IEEE

- Ghosh, P. and Karsai, G. (2020). An integrated cyber-physical fault management approach. In *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 148–149. IEEE

- Ghosh, P., Eisele, S., Dubey, A., Metelko, M., Madari, I., Volgyesi, P., and Karsai, G. (2020). Designing a decentralized fault-tolerant software framework for smart grids and its applications. *Journal of Systems Architecture*, 109:101759

- Ghosh, P., Shekhar, S., Lin, Y., Muenz, U., and Karsai, G. (2022a). Peer-to-peer communication trade-offs for smart grid applications. In *2022 International Conference on Computer Communications and Networks (ICCCN)*, pages 1–10. IEEE

- Ghosh, P., Tu, H., Krentz, T., Karsai, G., and Lukic, S. (2022b). An automated deployment and testing framework for resilient distributed smart grid applications. In *2022 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*, pages 1–6. IEEE

- Ghosh, P. and Karsai, G. (2023). Distributed cyber physical systems software model checking using timed automata. In *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE

8.2 Collaborations

- Eisele, S., Ghosh, P., Campanelli, K., Dubey, A., and Karsai, G. (2019). Transactive energy application with riaps. In *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 85–86. IEEE

- Pal, N., Ghosh, P., and Karsai, G. (2019). Deepeco: applying deep learning for occupancy detection from energy consumption data. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, pages 1938–1943. IEEE

References

Abdella, J. and Shuaib, K. (2018). Peer to peer distributed energy trading in smart grids: A survey. *Energies*, 11(6).

Ábrahám, E., Corzilius, F., Johnsen, E. B., Kremer, G., and Mauro, J. (2016). Zephyrus2: On the fly deployment optimization using smt and cp technologies. In *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, pages 229–245. Springer.

Aidemark, J., Folkesson, P., and Karlsson, J. (2005). A framework for node-level fault tolerance in distributed real-time systems. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 656–665. IEEE.

Akyol, B., Haack, J., Carpenter, B., Ciraci, S., Vlachopoulou, M., and Tews, C. (2012). Volttron: An agent execution platform for the electric power system. In *Third international workshop on agent technologies for energy systems valencia, spain*.

Al Farsi, F., Albadi, M., Hosseinzadeh, N., and Al Badi, A. (2015). Economic dispatch in power systems. In *2015 IEEE 8th GCC Conference & Exhibition*, pages 1–6. IEEE.

Ali, A. A., Krayem, S., Chramcov, B., and Kadi, M. F. (2018). Self-stabilizing fault tolerance distributed cyber physical systems. *Annals of DAAAM & Proceedings*, 29.

Ali, S., Siegel, H. J., and Maciejewski, A. A. (2004). The robustness of resource allocation in parallel and distributed computing systems. In *Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, pages 2–10. IEEE.

Almasalma, H., Claeys, S., Mikhaylov, K., Haapola, J., Pouttu, A., and Deconinck, G. (2018). Experimental validation of peer-to-peer distributed voltage control system. *Energies*, 11(5):1304.

Almesberger, W. et al. (1999). Linux network traffic control—implementation overview.

Alur, R. (1999). Timed automata. In *Computer Aided Verification*, pages 8–22. Springer Berlin Heidelberg.

Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., et al. (2018). Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15.

Becker, T. (1994). Application-transparent fault tolerance in distributed systems. In *Proceedings of 2nd International Workshop on Configurable Distributed Systems*, pages 36–45. IEEE.

Behrmann, G., David, A., and Larsen, K. G. (2004). A tutorial on uppaal. In *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg.

Bengtsson, J., Larsen, K. G., Larsson, F., Pettersson, P., and Yi, W. (1995). UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag.

Bintoudi, A. D., Zyglakis, L., Tsolakis, A. C., Ioannidis, D., Hadjidemetriou, L., Zacharia, L., Al-Mutlaq, N., Al-Hashem, M., Al-Agtash, S., Kyriakides, E., et al. (2020). Hybrid multi-agent-based adaptive control scheme for ac microgrids with increased fault-tolerance needs. *IET Renewable Power Generation*, 14(1):13–26.

Birman, K. P., Joseph, T. A., Raeuchle, T., and El Abbadi, A. (1985). Implementing fault-tolerant distributed objects. *IEEE Transactions on software Engineering*, (6):502–508.

Bower, W. I., Ton, D. T., Guttromson, R., Glover, S. F., Stamp, J. E., Bhatnagar, D., and Reilly, J. (2014). The advanced microgrid. integration and interoperability. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

Brewer, E. (2012a). Cap twelve years later: How the" rules" have changed. *Computer*, 45(2):23–29.

Brewer, E. (2012b). Cap twelve years later: How the "rules" have changed. *Computer*.

Bui, V.-H. and Su, W. (2022). Real-time operation of distribution network: A deep reinforcement learning-based reconfiguration approach. *Sustainable energy technologies and assessments*, 50:101841.

Carlson, J. L. (2013). *Redis in action*. Manning Publications Co.

Castro, M., Druschel, P., Ganesh, A., Rowstron, A., and Wallach, D. S. (2002). Secure routing for structured peer-to-peer overlay networks. *ACM SIGOPS Operating Systems Review*, 36(SI):299–314.

Castro, M., Liskov, B., et al. (1999). Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186.

Chassin, D. P., Schneider, K., and Gerkensmeyer, C. (2008). Gridlab-d: An open-source power systems modeling and simulation environment. In *2008 IEEE/PES Transmission and Distribution Conference and Exposition*, pages 1–5. IEEE.

Chen, A., Xiao, H., Haeberlen, A., and Phan, L. T. X. (2015). Fault tolerance and the five-second rule. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*.

Chen, C., Vitenberg, R., and Jacobsen, H.-A. (2021). Building fault-tolerant overlays with low node degrees for topic-based publish/subscribe. *IEEE Transactions on Dependable and Secure Computing*.

Chen, Y., Vinco, S., Pagliari, D. J., Montuschi, P., Macii, E., and Poncino, M. (2020). Modeling and simulation of cyber-physical electrical energy systems with systemc-ams. *IEEE Transactions on Sustainable Computing*, 5(4):552–567.

Chu, H. (2011). Lightning Memory-Mapped Database from Symas. https://github.com/LMDB.

Ciraci, S., Daily, J., Fuller, J., Fisher, A., Marinovici, L., and Agarwal, K. (2014). Fncs: a framework for power system and communication networks co-simulation. In *Proceedings of the symposium on theory of modeling & simulation-DEVS integrative*, page 36. Society for Computer Simulation International.

Clarke, E. M., Klieber, W., Nováček, M., and Zuliani, P. (2012). Model checking and the state explosion problem. In *Lecture Notes in Computer Science*, pages 1–30. Springer Berlin Heidelberg.

Coley, G. (2013). Beaglebone black system reference manual. *Texas Instruments, Dallas*, 5:2013.

Cornanguer, L., Largouët, C., Rozé, L., and Termier, A. (2022). Tag: Learning timed automata from logs. In *36th AAAI Conference on Artificial Intelligence (AAAI'22)*.

Costa, P. A., Fouto, P., and Leitão, J. (2020). Overlay networks for edge management. In *2020 IEEE 19th International Symposium on Network Computing and Applications (NCA)*, pages 1–10. IEEE.

Council, N. R. et al. (2012). Disaster resilience: A national imperative.

Debnath, H., Khan, M. A., Paiker, N. R., Ding, X., Gehani, N., Curtmola, R., and Borcea, C. (2019). The moitree middleware for distributed mobile-cloud computing. *Journal of Systems and Software*, 157:110387.

DeFranco, J. F. and Serpanos, D. (2021). The 12 flavors of cyberphysical systems. *Computer*, 54(12):104–108.

Dejanović, I., Vaderna, R., Milosavljević, G., and Vuković, Ž. (2017). Textx: a python tool for domain-specific languages implementation. *Knowledge-based systems*, 115:1–4.

Dhara, K., Guo, Y., Kolberg, M., and Wu, X. (2010). Overview of structured peer-to-peer overlay algorithms. In *Handbook of Peer-to-Peer Networking*, pages 223–256. Springer.

Downer, J. (2009). *When failure is an option: Redundancy, reliability and regulation in complex technical systems*. Number DP 53. Centre for Analysis of Risk and Regulation, London School of Economics and . . . .

Du, Y., Tu, H., Lukic, S., Dubey, A., and Karsai, G. (2018). Distributed microgrid synchronization strategy using a novel information architecture platform. In *2018 IEEE Energy Conversion Congress and Exposition (ECCE)*, pages 2060–2066. IEEE.

Dubey, A., Karsai, G., and Mahadevan, N. (2011). Model-based software health management for real-time systems. In *2011 Aerospace Conference*, pages 1–18. IEEE.

Dubey, A., Karsai, G., and Pradhan, S. (2017). Resilience at the edge in cyber-physical systems. In *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 139–146. IEEE.

Eisele, S., Ghosh, P., Campanelli, K., Dubey, A., and Karsai, G. (2019). Transactive energy application with riaps. In *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 85–86. IEEE.

Eisele, S., Mardari, I., Dubey, A., and Karsai, G. (2017a). Riaps: Resilient information architecture platform for decentralized smart systems. In *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 125–132. IEEE.

Eisele, S., Mardari, I., Dubey, A., and Karsai, G. (2017b). RIAPS: Resilient information architecture platform for decentralized smart systems. In *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE.

Enterprise, R. H. (2011). 3.2. cpu red hat enterprise linux 6 — red hat customer portal. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-cpu. (Accessed on 12/25/2019).

EPRI (2014). Transforming smart grid devices into open application platforms. Electric Power Research Institute Report 3002002859.

EPRI, E. P. R. I. (2011). Needed: A grid operating system to facilitate grid transformation. PDF.

Fang, X., Misra, S., Xue, G., and Yang, D. (2011). Smart grid—the new and improved power grid: A survey. *IEEE communications surveys & tutorials*, 14(4):944–980.

Faza, A. Z., Sedigh, S., and McMillin, B. M. (2009). Reliability analysis for the advanced electric power grid: From cyber control and communication to physical manifestations of failure. In *International Conference on Computer Safety, Reliability, and Security*, pages 257–269. Springer.

Feiler, P. H., Gluch, D. P., and Hudak, J. J. (2006). The architecture analysis & design language (aadl): An introduction. Technical report, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst.

Filgueiras, T. P., Rodrigues, L. M., de Oliveira Rech, L., de Souza, L. M. S., and Netto, H. V. (2019). Rt-jade: A preemptive real-time scheduling middleware for mobile agents. *Concurrency and Computation: Practice and Experience*, 31(13):e5061.

Finn, N. (2018). Introduction to time-sensitive networking. *IEEE Communications Standards Magazine*, 2(2):22–28.

Fkaier, S., Khalgui, M., and Frey, G. (2020). Meta-model for control applications of microgrids. In *2020 6th IEEE International Energy Conference (ENERGYCon)*, pages 945–950.

Foundry, E. (2020). Introduction - edgex foundry documentation. https://docs.edgexfoundry.org/2.1/. (Accessed on 11/18/2021).

Ganesh, A. J., Kermarrec, A.-M., and Massoulié, L. (2003). Peer-to-peer membership management for gossip-based protocols. *IEEE transactions on computers*, 52(2):139–149.

Gao, Z. and Wu, Z. (2007). Component assignment for large distributed embedded software development. In *International Conference on Grid and Pervasive Computing*, pages 642–654. Springer.

Genç, Z. and Özkasap, Ö. (2005). Peer-to-peer epidemic algorithms for reliable multicasting in ad hoc networks. *World Academy of Science, Engineering and Technology*, 3.

Ghosh, P., Eisele, S., Dubey, A., Metelko, M., Madari, I., Volgyesi, P., and Karsai, G. (2019). On the design of fault-tolerance in a decentralized software platform for power systems. In *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 52–60. IEEE.

Ghosh, P., Eisele, S., Dubey, A., Metelko, M., Madari, I., Volgyesi, P., and Karsai, G. (2020). Designing a decentralized fault-tolerant software framework for smart grids and its applications. *Journal of Systems Architecture*, 109:101759.

Ghosh, P. and Karsai, G. (2020). An integrated cyber-physical fault management approach. In *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 148–149. IEEE.

Ghosh, P. and Karsai, G. (2023). Distributed cyber physical systems software model checking using timed automata. In *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE.

Ghosh, P., Shekhar, S., Lin, Y., Muenz, U., and Karsai, G. (2022a). Peer-to-peer communication trade-offs for smart grid applications. In *2022 International Conference on Computer Communications and Networks (ICCCN)*, pages 1–10. IEEE.

Ghosh, P., Tu, H., Krentz, T., Karsai, G., and Lukic, S. (2022b). An automated deployment and testing framework for resilient distributed smart grid applications. In *2022 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*, pages 1–6. IEEE.

Glaß, M., Lukasiewycz, M., Streichert, T., Haubelt, C., and Teich, J. (2007). Reliability-aware system synthesis. In *2007 Design, Automation & Test in Europe Conference & Exhibition*, pages 1–6. IEEE.

Greer, C., Burns, M., Wollman, D., Griffor, E., et al. (2019). Cyber-physical systems and internet of things.

Gunes, V., Peter, S., and Givargis, T. (2013). Modeling and mitigation of faults in cyber-physical systems with binary sensors. In *2013 IEEE 16th International Conference on Computational Science and Engineering*, pages 515–522. IEEE.

Gupta, V., Langbort, C., and Murray, R. M. (2006). On the robustness of distributed algorithms. In *Proceedings of the 45th IEEE Conference on Decision and Control*, pages 3473–3478. IEEE.

Hamdane, M. E., Chaoui, A., and Strecker, M. (2013). From aadl to timed automaton-a verification approach. *International Journal of Software Engineering & Its Applications*, 7(4).

Hanmer, R. (2007). *Patterns for Fault Tolerant Software*. Wiley Publishing.

Hanmer, R. S. (2013). *Patterns for fault tolerant software*. John Wiley & Sons.

Heineman, G. T. and Councill, W. T. (2001). Component-based software engineering. *Putting the pieces together, addison-westley*, 5.

Henzinger, T. A., Minea, M., and Prabhu, V. (2001). Assume-guarantee reasoning for hierarchical hybrid systems. In Di Benedetto, M. D. and Sangiovanni-Vincentelli, A., editors, *Hybrid Systems: Computation and Control*, pages 275–290, Berlin, Heidelberg. Springer Berlin Heidelberg.

Herber, P., Pockrandt, M., and Glesner, S. (2015). STATE – a SystemC to timed automata transformation engine. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. IEEE.

Herrera, J. L., Galan-Jimenez, J., Bellavista, P., Foschini, L., Garcia-Alonso, J., Murillo, J. M., and Berrocal, J. (2021). Optimal deployment of fog nodes, microservices and SDN controllers in time-sensitive IoT scenarios. In *2021 IEEE Global Communications Conference (GLOBECOM)*. IEEE.

Hintjens, P. (2013). *ZeroMQ: messaging for many applications.* " O'Reilly Media, Inc.".

Hong, T., Pinson, P., Wang, Y., Weron, R., Yang, D., and Zareipour, H. (2020). Energy forecasting: A review and outlook. *IEEE Open Access Journal of Power and Energy*, 7:376–388.

Hovemeyer, D., Hellas, A., Petersen, A., and Spacco, J. (2016). Control-flow-only abstract syntax trees for analyzing students' programming progress. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. ACM.

Howe, D. (2012). Foldoc - computing dictionary. https://web.archive.org/web/20120717095437/http://foldoc.org/. (Accessed on 11/14/2021).

Hu, J. and Vasilakos, A. V. (2016). Energy big data analytics and security: challenges and opportunities. *IEEE Transactions on Smart Grid*, 7(5):2423–2436.

Iamnitchi, A. and Foster, I. (2000). A problem-specific fault-tolerance mechanism for asynchronous, distributed systems. In *Proceedings 2000 International Conference on Parallel Processing*, pages 4–13. IEEE.

IBM (2021). What is distributed cloud — ibm. https://www.ibm.com/topics/distributed-cloud. (Accessed on 12/19/2021).

IEEE (2008). Ieee sa - ieee 1588-2008. https://standards.ieee.org/ieee/1588/4355/. (Accessed on 02/13/2023).

InfluxData (2017). InfluxDB: storing time series data with high availability and high performance requirements. . https://influxdata.com/time-series-platform/influxdb/. [Online; accessed 28-July-2019].

Jabr, R., Coonick, A., and Cory, B. (2002). A primal-dual interior point method for optimal power flow dispatching. *IEEE Transactions on Power Systems*, 17(3):654–662.

Jagtap, P., Abdi, F., Rungger, M., Zamani, M., and Caccamo, M. (2020). Software fault tolerance for cyber-physical systems via full system restart. *ACM Transactions on Cyber-Physical Systems*, 4(4):1–20.

Januário, F., Cardoso, A., and Gil, P. (2019). A distributed multi-agent framework for resilience enhancement in cyber-physical systems. *IEEE Access*, 7:31342–31357.

Jelasity, M. and Babaoglu, O. (2005). T-man: Gossip-based overlay topology management. In *International Workshop on Engineering Self-Organising Applications*, pages 1–15. Springer.

Jin, X. and Chan, S.-H. G. (2010). Unstructured peer-to-peer network architectures. In *Handbook of peer-to-peer networking*, pages 117–142. Springer.

Kaur, A., Kaushal, J., and Basak, P. (2016). A review on microgrid central controller. *Renewable and Sustainable Energy Reviews*, 55:338–345.

Khan, M. R. B., Jidin, R., and Pasupuleti, J. (2016). Multi-agent based distributed control architecture for microgrid energy management and optimization. *Energy Conversion and Management*, 112:288–307.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer.

Kim, H.-W., Yi, G., Park, J. H., and Jeong, Y.-S. (2020). Adaptive resource management using many-core processing for fault tolerance based on cyber–physical cloud systems. *Future Generation Computer Systems*, 105:884–893.

Kim, K.-H., Qian, J., Zhang, Z., Zhou, Q., Moon, K.-D., Park, J.-H., Park, K.-R., and Kim, D.-H. (2011). A scheme for reliable real-time messaging with bounded delays. *Software: Practice and Experience*, 41(12):1387–1407.

King, C., Rhodes, J. D., Zarnikau, J., Lin, N., Kutanoglu, E., Leibowicz, B., Niyogi, D., Rai, V., Santoso, S., Spence, D., et al. (2021). The timeline and events of the february 2021 texas electric grid blackouts. *The University of Texas Energy Institute*.

Kong, F., Xu, M., Weimer, J., Sokolsky, O., and Lee, I. (2018). Cyber-physical system checkpointing and recovery. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 22–31.

Krishna, C. M. and Koren, I. (2013). Adaptive fault-tolerance fault-tolerance for cyber-physical systems. In *2013 International Conference on Computing, Networking and Communications (ICNC)*, pages 310–314. IEEE.

Kulczynski, M., Legay, A., Nowotka, D., and Poulsen, D. B. (2021). Analysis of source code using uppaal.

Kumar, M. and Singh, A. K. (2019). Fdds: An integrated conceptual fdds framework for dds based middleware. In *2019 International Conference on Communication and Electronics Systems (ICCES)*, pages 1952–1956. IEEE.

Kumar, P. S., Dubey, A., and Karsai, G. (2014). Colored petri net-based modeling and formal analysis of component-based applications. In *MoDeVVa@ MoDELS*, pages 79–88. Citeseer.

Kumar, S. and Cohen, P. R. (2000). Towards a fault-tolerant multi-agent system architecture. In *Proceedings of the fourth international conference on Autonomous agents*, pages 459–466.

Labs, G. (2014). Grafana: time series analytics. https://grafana.com/. [Online; accessed 28-July-2019].

Lakshman, A. and Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40.

Lantz, B., Heller, B., and McKeown, N. (2010). A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6.

Laprie, J.-C. (1985). Dependable computing and fault-tolerance. *Digest of Papers FTCS-15*, 10(2):124.

Lattner, C. and Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE.

Lee, E. A. and Seshia, S. A. (2015). Introduction to embedded systems - a cyber-physical systems approach.

Leitao, J., Pereira, J., and Rodrigues, L. (2007). Epidemic broadcast trees. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 301–310. IEEE.

Liu, J. and Srikantha, P. (2019). Decentralized topology reconfiguration in multiphase distribution networks. *IEEE Transactions on Signal and Information Processing over Networks*, 5(3):598–610.

Liva, G., Khan, M. T., and Pinzger, M. (2017). Extracting timed automata from java methods. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE.

Lopez-Garcia, T. B., Coronado-Mendoza, A., and Domínguez-Navarro, J. A. (2020). Artificial neural networks in microgrids: A review. *Engineering Applications of Artificial Intelligence*, 95:103894.

Lua, E. K., Crowcroft, J., Pias, M., Sharma, R., and Lim, S. (2005). A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials*, 7(2):72–93.

Magableh, B. and Almiani, M. (2019). A self healing microservices architecture: A case study in docker swarm cluster. In *Advanced Information Networking and Applications*, pages 846–858. Springer International Publishing.

Manolios, P. and Papavasileiou, V. (2010). Virtual integration of cyber-physical systems by verification. In *Proc. AVICPS*, page 65. Citeseer.

Maróti, M., Kecskés, T., Kereskényi, R., Broll, B., Völgyesi, P., Jurácz, L., Levendovszky, T., and Lédeczi, Á. (2014). Next generation (meta) modeling: web-and cloud-based collaborative tool infrastructure. *MPM@ MoDELS*, 1237:41–60.

Martin, R. C. (2003). *Agile software development: principles, patterns, and practices*. Prentice Hall PTR.

Marzal, S., González-Medina, R., Salas-Puente, R., Figueres, E., and Garcerá, G. (2017). A novel locality algorithm and peer-to-peer communication infrastructure for optimizing network performance in smart microgrids. *Energies*, 10(9):1275.

Maticu, F., Pop, P., Axbrink, C., and Islam, M. (2016). Automatic functionality assignment to autosar multicore distributed architectures. Technical report, SAE Technical Paper.

Miguel, E. C., Silva, C. M., Coelho, F. C., Cunha, I. F., and Campos, S. V. (2021). Construction and maintenance of p2p overlays for live streaming. *Multimedia Tools and Applications*, 80(13):20255–20282.

Miller, S. and Tribble, A. (2001). Extending the four-variable model to bridge the system-software gap. In *20th DASC. 20th Digital Avionics Systems Conference (Cat. No.01CH37219)*, volume 1, pages 4E5/1–4E5/11 vol.1.

Mills, D. L. (1991). Internet time synchronization: the network time protocol. *IEEE Transactions on communications*, 39(10):1482–1493.

Mirchandaney, R., Towsley, D., and Stankovic, J. A. (1990). Adaptive load sharing in heterogeneous distributed systems. *Journal of parallel and distributed computing*, 9(4):331–346.

Mohamed, N., Al-Jaroodi, J., Jawhar, I., Lazarova-Molnar, S., and Mahmoud, S. (2017). Smartcityware: A service-oriented middleware for cloud and fog enabled smart city services. *Ieee Access*, 5:17576–17588.

Monga, S. K., Ramachandra, S. K., and Simmhan, Y. (2019). Elfstore: A resilient data storage service for federated edge and fog resources. In *2019 IEEE International Conference on Web Services (ICWS)*, pages 336–345. IEEE.

Monti, A., Ponci, F., Benigni, A., and Liu, J. (2010). Distributed intelligence for smart grid control. In *2010 International School on Non-sinusoidal Currents and Compensation*, pages 46–58.

Naresky, J. J. (1970). Reliability definitions. *IEEE Transactions on Reliability*, R-19(4):198–200.

Nelson, V. P. (1990). Fault-tolerant computing: Fundamental concepts. *Computer*, 23(7):19–25.

Nuzzo, P., Bajaj, N., Masin, M., Kirov, D., Passerone, R., and Sangiovanni-Vincentelli, A. L. (2020). Optimized selection of reliable and cost-effective safety-critical system architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2109–2123.

Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA. USENIX Association.

Onishi, J., Kimura, S., James, R. J., and Nakagawa, Y. (2007). Solving the redundancy allocation problem with a mix of components using the improved surrogate constraint method. *IEEE transactions on reliability*, 56(1):94–101.

Orda, L., Gehrke, O., and Bindner, H. (2021). Applying overlay networks to the smart grid and energy collectives. *Electric Power Systems Research*, 192:106702.

Ouksel, A. M. and Sheth, A. (1999). Semantic interoperability in global information systems. *ACM Sigmod Record*, 28(1):5–12.

Pal, N., Ghosh, P., and Karsai, G. (2019). Deepeco: applying deep learning for occupancy detection from energy consumption data. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, pages 1938–1943. IEEE.

Palmintier, B., Krishnamurthy, D., Top, P., Smith, S., Daily, J., and Fuller, J. (2017). Design of the helics high-performance transmission-distribution-communication-market co-simulation framework. In *2017 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*, pages 1–6. IEEE.

Panahi, M., Nie, W., and Lin, K.-J. (2012). Rt-llama: Providing middleware support for real-time soa. In *Theoretical and Analytical Service-Focused Systems Design and Development*, pages 328–345. IGI Global.

Panda, P. R. (2001). Systemc: a modeling platform supporting multiple design abstractions. In *Proceedings of the 14th international symposium on Systems synthesis*, pages 75–80.

Pardo-Castellote, G. (2003). Omg data-distribution service: Architectural overview. In *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, pages 200–206. IEEE.

Pastore, F., Micucci, D., and Mariani, L. (2017). Timed k-tail: Automatic inference of timed automata. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.

Peterson, L. L. and Davie, B. S. (2007). *Computer networks: a systems approach*. Elsevier.

PNNL (2017). PNNL, Transactive energy simulation platform (TESP); 2017. https://tesp.readthedocs.io/en/latest/.

Pradhan, S., Dubey, A., Khare, S., Nannapaneni, S., Gokhale, A., Mahadevan, S., Schmidt, D. C., and Lehofer, M. (2018). CHARIOT. *ACM Transactions on Cyber-Physical Systems*, 2(3):1–37.

Pradhan, S. M., Dubey, A., Gokhale, A., and Lehofer, M. (2015). Chariot: A domain specific language for extensible cyber-physical systems. In *Proceedings of the workshop on domain-specific modeling*, pages 9–16.

Ptolemy (2012). Cyber-physical systems - a concept map. https://ptolemy.berkeley.edu/projects/cps/. (Accessed on 02/02/2022).

Python (2023). ast — abstract syntax trees — python 3.11.1 documentation. https://docs.python.org/3/library/ast.html. (Accessed on 02/06/2023).

Rahman, J., Jacob, R. A., Paul, S., Chowdhury, S., and Zhang, J. (2022). Reinforcement learning enabled microgrid network reconfiguration under disruptive events. In *2022 IEEE Kansas Power and Energy Conference (KPEC)*, pages 1–6. IEEE.

Rana, R., Bhattacharjee, S., and Mishra, S. (2018). A consensus based solution to the classic economic dispatch problem on a multi-agent system framework. In *2018 IEEMA Engineer Infinite Conference (eTechNxT)*, pages 1–6. IEEE.

Rashid, S. A., Audah, L., Hamdi, M. M., Abood, M. S., and Alani, S. (2020). Reliable and efficient data dissemination scheme in vanet: a review. *International Journal of Electrical and Computer Engineering (IJECE)*, 10(6):6423–6434.

Ratasich, D., Höftberger, O., Isakovic, H., Shafique, M., and Grosu, R. (2017). A self-healing framework for building resilient cyber-physical systems. In *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 133–140. IEEE.

Raynal, M. (2018). *Fault-tolerant message-passing distributed systems: an algorithmic approach*. Springer.

RCAM (2022). First 100% renewable multi-customer microgrid online in california. https://www.energy-storage.news/first-100-renewable-multi-customer-microgrid-online-in-california/. (Accessed on 02/12/2023).

Renganathan, S. A., Maulik, R., and Ahuja, J. (2021). Enhanced data efficiency using deep neural networks and gaussian processes for aerodynamic design optimization. *Aerospace Science and Technology*, 111:106522.

Reynolds, M. (2001). An axiomatization of full computation tree logic. *Journal of Symbolic Logic*, 66(3):1011–1057.

Rhea, S., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I., and Yu, H. (2005). Opendht: a public dht service and its uses. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 73–84.

RIAPS (2017). RIAPS Tutorials. https://riaps.github.io/tutorials.html.

Rossi, F., Cardellini, V., Presti, F. L., and Nardelli, M. (2020). Geo-distributed efficient deployment of containers with kubernetes. *Computer Communications*, 159:161–174.

Roxana, R.-B. and Eva-Henrietta, D. (2017). Fault-tolerant control of a cyber-physical system. In *IOP Conference Series: Materials Science and Engineering*, volume 261, page 012003. IOP Publishing.

Roy, N., Dubey, A., Gokhale, A., and Dowdy, L. (2011). A capacity planning process for performance assurance of component-based distributed systems. In *Proceeding of the second joint WOSP/SIPEW international conference on Performance engineering - ICPE '11*. ACM Press.

Sanislav, T., Mois, G., and Miclea, L. (2016). An approach to model dependability of cyber-physical systems. *Microprocessors and Microsystems*, 41:67–76.

Schroder, C. (2014). Understanding and using systemd - linux.com. https://www.linux.com/tutorials/understanding-and-using-systemd/. (Accessed on 12/28/2019).

Sha, L. and Meseguer, J. (2008). Design of complex cyber physical systems with formalized architectural patterns. In *Software-Intensive Systems and New Computing Paradigms*, pages 92–100. Springer.

Shariatzadeh, F., Vellaithurai, C. B., Biswas, S. S., Zamora, R., and Srivastava, A. K. (2014). Real-time implementation of intelligent reconfiguration algorithm for microgrid. *IEEE Transactions on Sustainable Energy*, 5(2):598–607.

Shariatzadeh, F., Zamora, R., and Srivastava, A. (2011). Real time implementation of microgrid reconfiguration. In *2011 North American Power Symposium*, pages 1–6.

Shojafar, M., Pooranian, Z., Naranjo, P. G. V., and Baccarelli, E. (2017). Flaps: bandwidth and delay-efficient distributed data searching in fog-supported p2p content delivery networks. *The journal of supercomputing*, 73(12):5239–5260.

Shukla, N., Datta, D., Pandey, M., and Srivastava, S. (2021). Towards software defined low maintenance structured peer-to-peer overlays. *Peer-to-Peer Networking and Applications*, 14(3):1242–1260.

Singh, C. and Sprintson, A. (2010). Reliability assurance of cyber-physical power systems. In *IEEE PES General Meeting*, pages 1–6. IEEE.

Skala, K., Davidovic, D., Afgan, E., Sovic, I., and Sojat, Z. (2015). Scalable distributed computing hierarchy: Cloud, fog and dew computing. *Open Journal of Cloud Computing (OJCC)*, 2(1):16–24.

Spalazzi, L., Spegni, F., Liva, G., and Pinzger, M. (2018). Towards model checking security of real time java software. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE.

Tanganelli, G., Cassano, L., Miele, A., and Vallati, C. (2020). A methodology for the design and deployment of distributed cyber–physical systems for smart environments. *Future Generation Computer Systems*, 109:420–430.

Tao, L. and Schwaegerl, C. (2009). Advanced architectures and control concepts for more microgrids. *EC Project, Tech. Rep. SES6–019864, Tech. Rep.*

Tappler, M., Aichernig, B. K., Larsen, K. G., and Lorber, F. (2018). Learning timed automata via genetic programming.

Tayab, U. B., Roslan, M. A. B., Hwai, L. J., and Kashif, M. (2017). A review of droop control techniques for microgrid. *Renewable and Sustainable Energy Reviews*, 76:717–727.

Tebekaemi, E. and Wijesekera, D. (2019). Secure overlay communication and control model for decentralized autonomous control of smart micro-grids. *Sustainable Energy, Grids and Networks*, 18:100222.

Temple, C. (1998). Avoiding the babbling-idiot failure in a time-triggered communication system. In *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 98CB36224)*, pages 218–227. IEEE.

Thakar, S., Vijay, A., and Doolla, S. (2019). System reconfiguration in microgrids. *Sustainable energy, grids and networks*, 17:100191.

Ton, D. T. and Smith, M. A. (2012). The us department of energy's microgrid initiative. *The Electricity Journal*, 25(8):84–94.

Tushar, W., Yuen, C., Mohsenian-Rad, H., Saha, T., Poor, H. V., and Wood, K. L. (2018). Transforming energy networks via peer-to-peer energy trading: The potential of game-theoretic approaches. *IEEE Signal Processing Magazine*, 35(4):90–111.

Vachtsevanos, G., Lee, B., Oh, S., and Balchanos, M. (2018). Resilient design and operation of cyber physical systems with emphasis on unmanned autonomous systems. *Journal of Intelligent & Robotic Systems*, 91(1):59–83.

Varda, K. (2013). Cap'n proto: Introduction. *Sandstorm,[Online]. Available: https://capnproto. org/index. html.[Använd 31 mars 2017].*

Vélez-Rivera, C. J., Andrade, F., Arzuaga-Cruz, E., and Irizarry-Rivera, A. (2018). Gorilla: An open interface for smart agents and real-time power microgrid system simulations. *Inventions*, 3(3):58.

Voulgaris, S., Gavidia, D., and Van Steen, M. (2005). Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and systems Management*, 13(2):197–217.

Wang, X., Hovakimyan, N., and Sha, L. (2013). L1simplex: Fault-tolerant control of cyber-physical systems. In *2013 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, pages 41–50. IEEE.

Wu, M., Zeng, H., Wang, C., and Yu, H. (2017). Safety guard: Runtime enforcement for safety-critical cyber-physical systems. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE.

Xia, Y., Etchevers, X., Letondeur, L., Coupaye, T., and Desprez, F. (2018). Combining hardware nodes and software components ordering-based heuristics for optimizing the placement of distributed IoT applications in the fog. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. ACM.

Xie, G., Zeng, G., An, J., Li, R., and Li, K. (2018). Resource-cost-aware fault-tolerant design methodology for end-to-end functional safety computation on automotive cyber-physical systems. *ACM Transactions on Cyber-Physical Systems*, 3(1):1–27.

Xu, Y., Koren, I., and Krishna, C. M. (2017). Adaft: A framework for adaptive fault tolerance for cyber-physical systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(3):1–25.

Zhao, Z., Min, G., Gao, W., Wu, Y., Duan, H., and Ni, Q. (2018). Deploying edge computing nodes for large-scale IoT: A diversity aware approach. *IEEE Internet of Things Journal*, 5(5):3606–3614.

Zheng, B., Liang, H., Wang, Z., and Zhu, Q. (2019). Model-based software synthesis for safety-critical cyber-physical systems. In *Safe, Autonomous and Intelligent Vehicles*, pages 163–186. Springer.

Zhou, N., Georgiou, Y., Pospieszny, M., Zhong, L., Zhou, H., Niethammer, C., Pejak, B., Marko, O., and Hoppe, D. (2021). Container orchestration on HPC systems through kubernetes. *Journal of Cloud Computing*, 10(1).