2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)

On the Design of Fault-Tolerance in a Decentralized Software Platform for Power Systems

Purboday Ghosh Scott Eisele Abhishek Dubey Mary Metelko
Istvan Madari Peter Volgyesi Gabor Karsai
Institute for Software-Integrated Systems, Vanderbilt University Nashville, TN, 37212

***Abstract*—The vision of the 'Smart Grid' assumes a distributed real-time embedded system that implements various monitoring and control functions. As the reliability of the power grid is critical to modern society, the software supporting the grid must support fault tolerance and resilience in the resulting cyber-physical system. This paper describes the fault-tolerance features of a software framework called Resilient Information Architecture Platform for Smart Grid (RIAPS). The framework supports various mechanisms for fault detection and mitigation and works in concert with the applications that implement the grid-specific functions. The paper discusses the design philosophy for and the implementation of the fault tolerance features and presents an application example to show how it can be used to build highly resilient systems.**

***Index Terms*—component, fault tolerance, distributed systems, smart grid**

I. Introduction

Emerging Trends. The modern "Smart Grid" allows bi-directional energy flows (where locally generated energy is dynamically shared with other customers on the same network, contrary to the traditional central generation model), demand-response (where sudden and urgent energy needs are addressed by rapidly reallocating generation resources), and transactive energy (where distributed production and consumption are dynamically balanced through market mechanisms). These properties are possible because of the ubiquity of information that is now available.

The increased complexity and dynamism in smart grids have 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 conforming to industry standards and protocols, such as those being developed by IEEE and IEC [1]. Having standards in place enables innovation and creativity across all aspects of the grid, from enterprise applications to local device control. Interaction protocols facilitate 1) two-way data sharing, 2) measurement and state estimation, 3) leveraging of sensing device information for system availability and stability assessment, 4) some local control/protection activity, and 5) competitive transactions to encourage direct engagement with consumers [2], [3]. But along with the open protocol and its advantages, comes a strong need for security, quality control, reliability, and resilience [3].

In this paper, we discuss the problem of reliability and resilience in decentralized, real-time, embedded systems with respect to fault detection and mitigation. We will describe the design and implementation of the concepts in a software framework called Resilient Information Architecture Platform for Smart Grid (RIAPS) [4]. RIAPS is an *open application platform* that 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, and increasing reliability by reducing dependency on the availability of and access to centralized resources. The platform supports multi-tenancy and the controlled sharing of computational and communication resources. The key concept of RIAPS is to provide a "middleware" to facilitate the interactions among networked computational actors that focus on specific grid issues, such as state estimation, remedial action schemes, and power and energy management, and time-sensitive applications.

The problem of distributed fault tolerance is not new. The past is filled with examples of critical failures [5], [6], [7]. The problem is that even though there are multiple mechanisms to achieve fault tolerance at both the hardware and software level, very few implemented architectures are available for a highly resilient, hierarchical fault management scheme. The key idea is to make each layer robust such 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.

Of particular interest to fault tolerance 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) [8][1], the choice of communication patterns still remains challenging. 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. We discuss the motivation of the choices of communication patterns made available in RIAPS and how it helps in achieving fault tolerance later in the paper.

Contributions: The specific contributions of this paper are as follows:

1) We describe the detailed architectural interaction pattern of RIAPS and use it to illustrate the anticipated failure modes.

[1]The guiding principles of CBSE are interfaces with well-defined execution models [9], compositional semantics [10] and model-driven analysis [11]

978-1-7281-0151-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ISORC.2019.00018

2) We describe the implementation of the fault-tolerance architecture and how it provides the layered protection.
3) We finally present an evaluation of the design using a complex transactive energy application [12].

Outline. We start with a background and related research on fault tolerance. Then we describe the internal architecture and interaction patterns of RIAPS. Then, we discuss the anticipated failure modes and the fault-tolerance features that have been designed to recover from those faults. We finish with a set of case studies.

II. Related Research

There has been significant research on the development of integrated platforms prior to RIAPS. In this section, we discuss some of those technologies.

Volttron [13] is a real-time software platform developed by the Pacific Northwest National Laboratory (PNNL) with support from the US Department of Energy. It is a completely language agnostic, agent-based, distributed framework that provides interfaces for secure, scalable execution of power systems applications. The Volttron architecture is comprised of several modules that provide specific capabilities like secure communication, resource monitoring, authentication and non-volatile data storage to agents, which have specific roles.

Open Field Message Bus (OpenFMB) [14] is another distributed, intelligent platform for electrical power systems. It was adopted by the North American Energy Standards Board (NAESB) in 2016. Its primary goal is to enable interoperable systems on modern power grids. It emphasizes a data-driven approach comprised of peer to peer interconnected nodes. It has a multi-layered architecture that specifies configuration parameters, interaction patterns, and QoS requirements leveraging existing publish-subscribe middleware technologies like DDS [15] and MQTT [16]. It also performs management level tasks like release management and monitoring.

Neither of these platforms provides an extensive, hierarchical, resilient fault management framework to detect, report and recover from faults at multiple levels of the architecture. They also lack support for 1) a wide variety of communication patterns (with the agents/nodes being limited to publish-subscribe messaging), 2) distributed coordination protocols like leader election and consensus, and 3) timing or synchronization related features which are critical in modern real-time systems. Similarly, other topic-based messaging platforms like ROS [17] are not adequate to handle the complex control logic, timing requirements and granular level resilience guarantees for power systems applications. RIAPS addresses these needs by providing a robust fault management architecture along with necessary platform-level services for real-time coordination and control.

III. Overview of RIAPS

RIAPS supports execution of distributed software applications that run on a network of computing nodes by 1) facilitating deployment, 2) providing remote management of applications on nodes, 3) supporting reliable communication

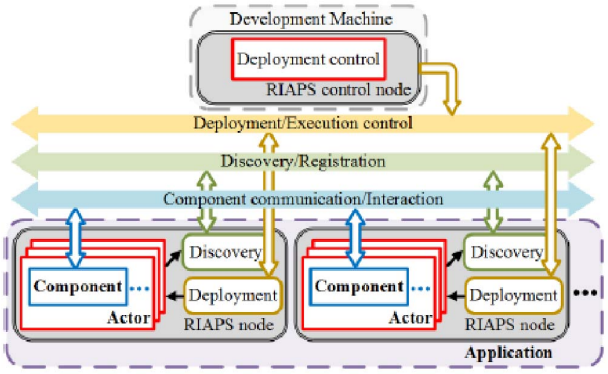

Fig. 1. RIAPS Architecture

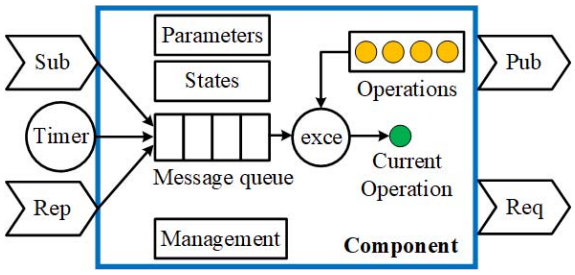

Fig. 2. RIAPS component model.

patterns among computation units, including anonymous publish/subscribe, and synchronous/asynchronous remote method invocation, 4) enabling distributed data management and resource sharing, 5) providing distributed coordination and high precision time synchronization services between the nodes, and 6) implementing a robust fault management framework which 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 the the application structure description model file. Details about the RIAPS API and modeling language can be found in [18].

The RIAPS run-time system is composed of several layers (see figure 1). The top layer contains the decentralized computational units below which reside the framework level services. We discuss them in greater detail below.

1) **Components and Actors.** In the RIAPS modeling semantics, 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 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. Request-reply and client-server is used for synchronous send-receive operations, query-answer is used for asynchronous coupled request and response, publish-subscribe is used for asynchronous decoupled send and receive. Timer ports are used to invoke periodic or sporadic events. 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.

ZeroMQ [19] is used as the underlying communication middleware. Message serialization can be achieved through Cap'n Proto [20] because of its fast in-memory marshaling and unmarshalling capabilities. Fig 2 illustrates the RIAPS Component model.

2) **Deployment Control.** The Deployment Control is an interactive application for downloading the applications to the RIAPS nodes. It provides a GUI for visualizing the status of the RIAPS nodes and allowing selection, and deployment of Actors to specific nodes, aiding in running RIAPS applications.
3) **Deployment Service.** The Deployment Service is responsible for installing and managing applications on a RIAPS node and for monitoring their operational status. When started, the Deployment Service connects to a dedicated RIAPS Deployment Control node. This connection is facilitated by the secure RPyC [21] distributed computing service.
4) **Discovery Service.** The main role of the Discovery Service [4] (which runs on all nodes) is to enable the dynamic configuration of information flows among RIAPS application components. Each component that either publishes data with a certain topic or provides a specific service is registered with the service. Each component that subscribes to data with a certain topic or requires a specific service locates these with the help of the service. The service maintains a snapshot of the Actors on the network that are currently active, enabling peers to dynamically join and leave the network, while the system still remains operational. It uses a distributed, fault-tolerant hash table for topic publisher and service provider registrations.
5) **Time Synchronization Service.** Accurate time synchronization is critical for designing time-sensitive systems. Thus, RIAPS provides a high precision time synchronization service to maintain synchronization of the clocks among the nodes. The Timesync Service [22] can operate in two modes, a master-slave mode and a standalone mode. In the master-slave mode, one node serves as the synchronization master, while the slave nodes synchronize to the master using PTP. The Timesync Service also has the capabilities to generate precise timing signals for external hardware devices which can be used to synchronize them. It has the capability of providing sub-millisecond level clock accuracy across different nodes.
6) **Other Platform services.** RIAPS also runs several auxiliary platform services that operate below the application layer. They are Distributed Coordination Services, Resource Management Services, Security and Authentication Services, Logging Services, and Persistent Data Storage Services.
7) **Device Interfacing Services.** Device Interface Components (hosted in Device Interface Actors), provide an interface for RIAPS Components to interact with physical systems [23]. This is important to consider because RIAPS is designed to sense and control the physical environment. For example, in a power system, the controller would need to connect to physical devices via GPIO pins, serial ports, or network interfaces. By using device interface Components, we abstract the implementation level details of these physical connections like the communication protocols. Unlike normal Components, device interface Components support multi-threaded operations and concurrent code.

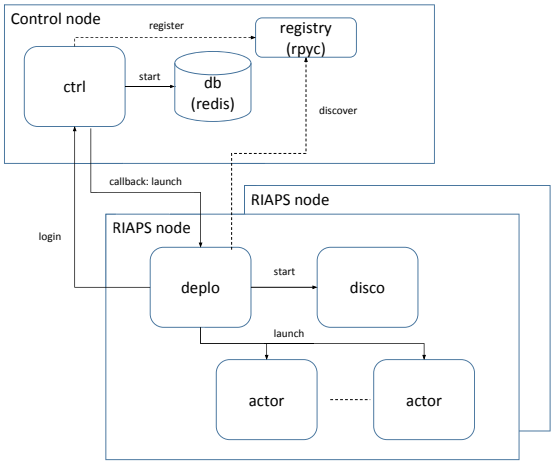

Fig. 3. The interactions of all RIAPS platform services.

A. Service Interaction

To describe the fault management architecture (figure 3) it is important to describe the relationship between all the services and Actors. We describe these interactions in this section.

The control node acts as a control center from which applications can be remotely managed. When starting the controller service (`ctrl` in figure), it registers to an RPyC registry server. RPyC provides a transparent Python library for distributed computing services such as remote procedure calls and service registration. When a RIAPS node logs in, it initiates a callback to request deployment of Actors to the Deployment Service running on the participating RIAPS nodes.

Each RIAPS node has the Deployment Service (`deplo` in figure) running. `deplo` calls the RPyC discover method to find and connect to the RPyC registry server. Using this mechanism, all the client nodes defined in the deployment file of the application can login to `ctrl`. `deplo` also starts the Discovery Service (`disco` in figure) on each node. As explained in Section III, the Discovery Service is responsible for maintaining the dynamic state of all active peers in a cluster. These platform level services are configured as Linux `systemd` services. Once the callback for an application

launch is invoked, the `deplo` starts the Actor processes for that node. The client Actors register with the Discovery Service using a 'client/server' (REQ/REP) socket pair. The service will then create a dedicated socket for the specific client. This socket is used as a private communication channel between a specific client Actor and the service. Once the services are registered with the service, then the Actor starts the associated Components in each thread. This sequence of steps is followed in the reverse order in case of termination of the application.

B. Failure Modes

Faults that can arise at different layers of the system can be categorized as follows:

1) **Device interface fault:** These faults occur when a physical device does not function properly.
2) **Communication fault:** These faults occur in the communication channels and they can be caused by faults in the physical links, a fault in the kernel or if a RIAPS node goes down or becomes unreachable.
3) **Hardware level fault:** These faults occur when a Component raises a hardware exception, such as a CPU hardware exception or a segmentation fault which results in a signal generated by the kernel and sent to the Actor to which the Component belongs.
4) **Kernel level fault:** The faults are caused by a flaw in the Component code or resource violation that causes an invalid request to the kernel.
5) **Actor level fault:** These faults occur when the Actor process crashes or an exception is raised by the Component code which is detected at the Actor level. A subcategory of Actor Faults are **Resource Management Faults** caused by the component asking for more resources than it has been assigned.
6) **Framework level fault:** These faults are caused by errors in the RIAPS framework code.
7) **Logical faults:** These faults are caused by errors in the business logic of the Component code.

As mentioned previously, all these faults can propagate and lead to secondary faults. For example, a fault in the network interface will inhibit the Actors from sending and receiving service queries which will cause the Discovery Service to operate with obsolete information. A hardware fault like a node power failure can lead to infinite blocking in a Component that waits for a response from a service on that failed node. A fault in one Component may also trigger a fault in another Component. Therefore, the goal of the RIAPS fault management architecture is to identify faults close to the location they happen and then to provide mechanisms for mitigation actions at the same time ensuring that every party in an interaction sequence is aware of the fault.

IV. Fault Management Architecture

Principles. 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 Smart Grid built with RIAPS nodes are required across three layers: physical and device level, 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. Therefore, in this paper, we focus on the fault management capabilities required at the level of software platform services and the application. It is expected that any failure of the physical device, including networks and computation nodes, is $Fail-Stop$ and can be successfully detected using the $Watchdog$ fault tolerance pattern [24].

The fault management sub-system at the platform services level is responsible for providing detection and recovery capabilities for the Discovery, Deployment and Time Synchronization Services. Table II summarizes the various detection and mitigation schemes that RIAPS has incorporated for 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 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 I 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 extends the functions provided by Linux. `cgroups` is a kernel feature which allows setting of restrictions on the memory, CPU, and network usage on a collection of processes. These processes make up a control group or `cgroup`. The Linux 'traffic controller' (tc) is another useful utility which allows packet level 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 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 4. The Actor "LimitActor" has restrictions imposed with respect to CPU, Memory and Disk space using the "uses" block. The limits are hard limits by default but can be soft limits by indicating a tolerance range. In this example, three separate Components are defined in the Actor which implements the associated handler for a particular fault.

```
actor LimitActor {
 uses {
  cpu max 10 % over 1;//Hard limit, no 'max' is softlimit
  mem 200 mb;          // Mem limit
  space 10 mb;         // File space limit
  net rate 10 kbps ceil 12 kbps burst 1.2 k; // Netlimits
  }
  ...
}
```

Fig. 4. An example of an Actor specification showing the resource monitoring options

RIAPS is also capable of enforcing timing constraints 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 5. Thus, the timer operation has a completion deadline of 1 millisecond for the Component "Sensor". The specific handler method needs to be used in 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 GPS, NTP and PTP

TABLE I
SYSTEM-LEVEL FAULT MANAGEMENT IMPLEMENTATION FOR APPLICATIONS

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 II
SYSTEM-LEVEL FAULT MANAGEMENT IMPLEMENTATION FOR PLATFORM SERVICES AND HARDWARE

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

```
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
      ...
    }
```

Fig. 5. An example of a Component model showing the deadline constraint on a timer port. The SensorReady, SensorQuery and SensorValue are messages. 'clock' is a timer that will fire every second, 'pub' and 'rep' are publisher and reply ports.

[25] to achieve synchronization. In the clock master node if both GPS and PTP are available, it will dynamically pick 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 added layer of security by providing access control functions through the use of an application level user id (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 access by external agents to read and write on the file system of the node.

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 Actors register with it in a Distributed Hash Table (using OpenDHT [26]). 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 is 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 `systemd` for RIAPS Actors. When an application Actor is deployed, the Deployment Service keeps the log of the executed deployment operations in a 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, then it can still retrieve the list of currently running applications from the local database. If the RIAPS node crashes, then the control node can decide to re-install the Actors on the same or on a different node.

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. *Choosing a leader* is a process where a single Component becomes designated as an organizer of tasks (or decision maker) among several distributed Components. To select a leader in a group, the RIAPS platform provides the API to an implementation of the *Raft* [27] election algorithm. *Consensus* is the mechanism by which group members form agreement on a set of data values. *Time-synchronized coordinated action* is another service provided which coordinates agreement among distributed nodes regarding when a time-synchronized action should be performed.

Application level Fault Handlers. Table I lists the various fault management schemes employed 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 and perform a restart by reapplying the operations stored in its database. Application developers can customize the 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's 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.

A. Fault Management Interactions

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.

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 instance, 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/subscribe mechanism. If an application Actor crashes and/or 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.

Interaction Example. 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 Fig. 6. The failure is detected by `systemd`, and the Deployment Manager is restarted. Upon 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 liveness 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 JOIN message. This is again propagated through the peer node (not shown). After re-initializing 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.

V. An Example: Transactive Energy Application

In a prior work [12] we used RIAPS to implement a platform for decentralized energy trading in transactive microgrids. The Actors and high-level dataflow of this platform can be seen in figure 7. The typical workflow is for producers and consumers of power to 1) post offers to a distributed ledger for a time interval in the future, in [12] we used Ethereum. Solvers are monitoring the ledger and when offers are posted they 2) receive a notification, and match buyers to sellers. This match is 3) posted to the ledger. The solution for an interval may be updated until it is 4) finalized by the distribution system operator (DSO). At this point the producer and consumer are notified and will 6) exchange the amount of power they were matched for when the interval arrives. RIAPS was used to provide inter-Actor communication, management services, and time-synchronization for the Actors to begin the transfer of power at the right time. In this paper, we again use this example to test some of the fault management properties of RIAPS to determine if it meets the requirements for this application.

A. Experiments

We chose to experiment using a Beaglebone ARM cluster since many IoT devices are ARM machines. All nodes run Ubuntu 18.04. We ran a single instance of the Ethereum Geth client on the cluster master (an Intel machine running Linux). We simulated three failure scenarios. The goal was to see how quickly a failure was detected and propagated as well as the time required for recovery of the Actors.

Experiment 1: Network Failure. Figure 8 shows the results of experiment one, where we disconnected the Ethernet cable from one of the nodes. We see that the time for a peer to be notified of the disconnect is about 3 seconds on average, reconnect is about 1 second, and the time for the Actors on that node to be notified that they have been reconnected is 5 seconds on average.

Experiment 2: Platform failure. Figure 9 shows the results of experiment two, where we kill the Deployment Service on one of the nodes. The events timed here correspond to some of the events shown in figure 6 as that shows what occurs when a Deployment Service fails. Times (a)-(d) are all phases of recovery on the failed node, and times (e) and (f) are the time to notify a peer of the node exiting and joining. This is important because that determines how much time is used before mitigation actions can be taken. In this experiment, it takes 5 seconds on average for the exit message to arrive.

The key observation here is that the peers are notified of failure within about 5 seconds. In the context of the transactive

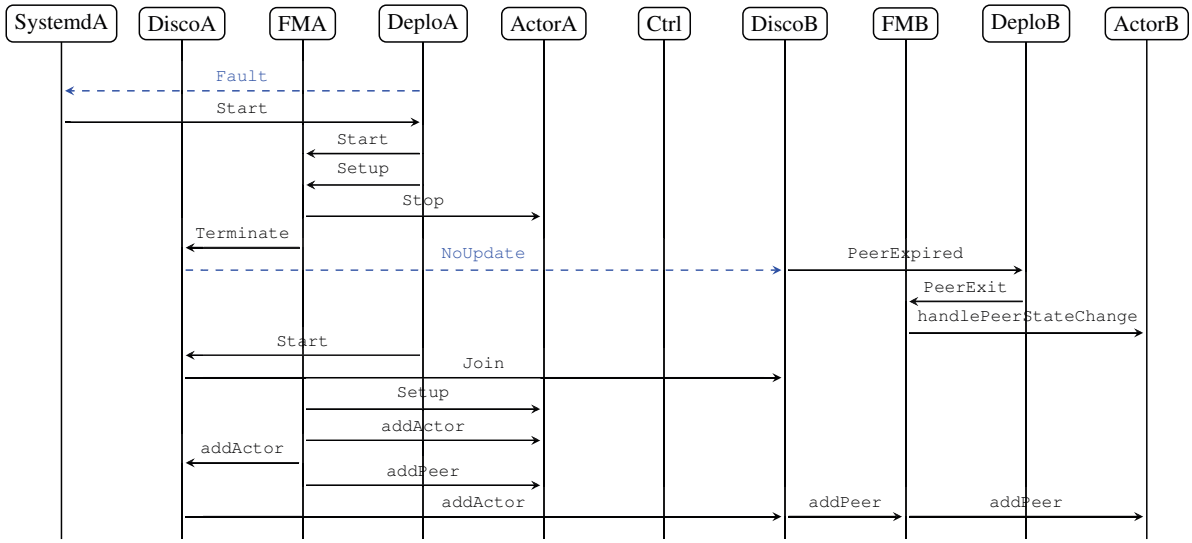

Fig. 6. Sequence diagram showing the actions that occur when a Deployment Service fails (not all failures are shown). The dashed lines are events that are detected, the first is systemd detecting a process failure, the second is a timeout in the Discovery Service.

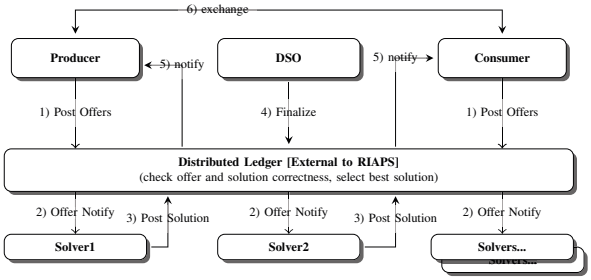

Fig. 7. Data flow between actors of in Transactive Energy application.

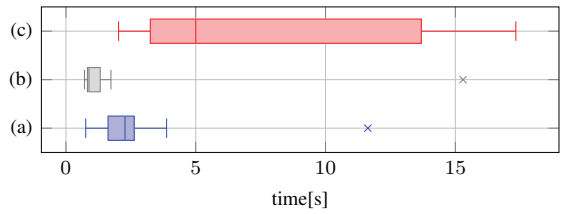

Fig. 8. Results from Experiment 1. The horizontal axis is the elapsed time in seconds. The values shown from bottom to top are: (a) time to notify peer of disconnect, (b) time to notify peer of reconnect, (c) time for actors on a disconnected node to be notified.

energy platform, the potential for a problem is in the 5 seconds prior to beginning an exchange of power. This delay may be reduced if the Actors are run with real-time priority, as they were not for this experiment. Additionally, we could include additional messaging during critical time periods.

Experiment 3: Resource Violations. Figure 10 shows the speed of response of the RIAPS platform for three different scenarios, namely disk space limit, memory limit and CPU usage limit violations. The time recorded is the interval between the instant the fault manager detects a violation to when the associated handler method is called. The observed data shows that for all scenarios, the system was able to respond within 85 milliseconds.

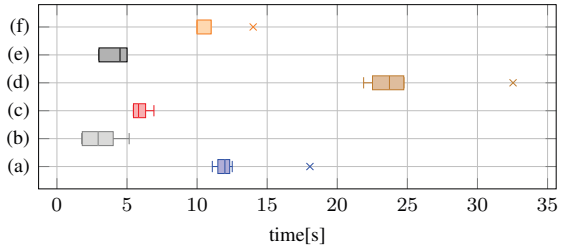

Fig. 9. Results from Experiment 2. The horizontal axis is the elapsed time in seconds. From bottom to top they are: (a) time to notify local actors, (b) time to terminate actors owned by previous deplo, (c) time to clean up other actors owned by previous deplo, (d) time until actors are fully recovered. (e) is the time until the peers know the node has left, and (f) is when the nodes know the peer has rejoined.

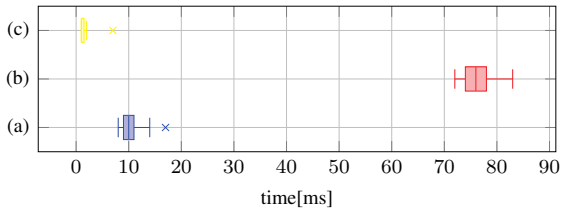

Fig. 10. Results from Experiment 3. The horizontal axis is the elapsed time in milliseconds. The values indicate the time interval between detection and invocation of the associated handler for (a) Disk Usage limit, (b) Memory Usage limit and (c) CPU limit

In all of the of experiments, the results show a highly reactive system that is able to sense an anomaly and react to it on the order of a few seconds. This is very important for a critical application like a power system, where the state of the system might change abruptly due to arbitrary device faults or environmental factors.

System Overhead. Overhead is principally due to the Deployment, Discovery and Time Synchronization Services. The average CPU and Memory utilization metrics for these processes were observed to be around 1.5% and 1.9%. Thus,

RIAPS did not add any significant memory or processing load to the nodes. However, since each RIAPS Actor runs in a single thread, running large-scale applications with numerous Actors may lead to some performance bottlenecks. In those cases, the resource throttling features discussed previously can be employed.

VI. Conclusions

In this paper, we introduced the fault management subsystem of the RIAPS framework and demonstrated its use in a transactive energy application. In the design of the fault management subsystem, a systematic approach was followed where 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 were examined in terms of their role in improving the resilience properties of the system and implemented accordingly. The experiments performed using the transactive energy application show that the response and recovery times of the system for various fault conditions and resource limit violation events were within reasonable bounds. Adopting techniques such as real-time scheduling can possibly lead to further improvements in the overall performance. In future work, we plan to conduct more experiments to cover a wider gamut of dependability and resilience metrics.

Acknowledgment

This work 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.

References

[1] EPRI, "Transforming smart grid devices into open application platforms," Electric Power Research Institute Report 3002002859, July 2014. [Online]. Available: http://www.epri.com/abstracts/Pages/ProductAbstract.aspx?productid=000000003002002859

[2] A. Monti, F. Ponci, A. Benigni, and J. Liu, "Distributed intelligence for smart grid control," in *2010 International School on Non-sinusoidal Currents and Compensation*, June 2010, pp. 46–58.

[3] E. P. R. Institute, "Needed: A grid operating system to facilitate grid transformation," PDF, June 2011. [Online]. Available: https://www.smartgrid.gov/files/Needed_Grid_Operating_System_to_Facilitate_Grid_Transformati_201108.pdf

[4] S. Eisele, I. Mardari, A. Dubey, and G. Karsai, "Riaps: resilient information architecture platform for decentralized smart systems," in *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2017, pp. 125–132.

[5] A. Cockroft, C. Hicks, and G. Orzell, "Lessons Netflix learned from the AWS outage," *Netflix Techblog*, 2011.

[6] AWS, "Summary of the Amazon DynamoDB Service Disruption and Related Impacts in the US-East Region," 2016, [Online; accessed 24-January-2019]. [Online]. Available: https://aws.amazon.com/message/5467D2/

[7] GitLab, "Postmortem of database outage of January 31," 2017, [Online; accessed 24-January-2019]. [Online]. Available: https://about.gitlab.com/2017/02/10/postmortem-of-database-outage-of-january-31/

[8] G. T. Heineman and W. T. Councill, "Component-based software engineering," *Putting the pieces together, addison-westley*, p. 5, 2001.

[9] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, "Rigorous component-based system design using the BIP framework," *IEEE software*, vol. 28, no. 3, pp. 41–48, 2011.

[10] H. Schmidt, "Trustworthy components—-compositionality and prediction," *Journal of Systems and Software*, vol. 65, no. 3, pp. 215–225, 2003.

[11] P. S. Kumar, A. Dubey, and G. Karsai, "Colored petri net-based modeling and formal analysis of component-based applications." in *MoDeVVa@ MoDELS*. Citeseer, 2014, pp. 79–88.

[12] A. Laszka, S. Eisele, A. Dubey, G. Karsai, and K. Kvaternik, "Transax: A blockchain-based decentralized forward-trading energy exchange for transactive microgrids," in *Proceedings of the 24th IEEE International Conference on Parallel and Distributed Systems (ICPADS)(December 2018)*, 2018.

[13] R. Lutes, S. Katipamula, B. Akyol, J. Haack, K. Monson, and B. Carpenter, "VOLTTRON 3.0: User guide," Pacific Northwest National Laboratory, U.S. Department of Energy, Tech. Rep. PNNL-24907, November 2015, under Contract DE-AC05-76RL01830. [Online]. Available: http://www.pnnl.gov/main/publications/external/technical_reports/PNNL-24907.pdf

[14] UCA International Users Group, "Open field message bus (openfmb)," 2019, [Online; accessed 24-January-2019]. [Online]. Available: https://openfmb.ucaiug.org

[15] OMG, "The Data Distribution Service specification, v1.2," http://www.omg.org/spec/DDS/1.2, 2007.

[16] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "Mqtt-s—a publish/subscribe protocol for wireless sensor networks," in *Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on*. IEEE, 2008, pp. 791–798.

[17] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[18] "RIAPS Tutorials," https://riaps.github.io/tutorials.html.

[19] P. Hintjens, *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 2013.

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

[21] Open Source, "RPyC - Transparent, Symmetric Distributed Computing," 2019, [Online; accessed 24-January-2019]. [Online]. Available: https://rpyc.readthedocs.io/

[22] P. Volgyesi, A. Dubey, T. Krentz, I. Madari, M. Metelko, and G. Karsai, "Time synchronization services for low-cost fog computing applications," in *Proceedings of the 28th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype*. ACM, 2017, pp. 57–63.

[23] A. Dubey, G. Karsai, P. Volgyesi, M. Metelko, I. Madari, H. Tu, Y. Du, and S. Lukic, "Device access abstractions for resilient information architecture platform for smart grid," *IEEE Embedded Systems Letters*, pp. 1–1, 2018.

[24] R. Hanmer, *Patterns for Fault Tolerant Software*. Wiley Publishing, 2007.

[25] D. L. Mills, "Internet time synchronization: the network time protocol," *IEEE Transactions on communications*, vol. 39, no. 10, pp. 1482–1493, 1991.

[26] "A C++11 Distributed Hash Table implementation," https://github.com/savoirfairelinux/opendht/wiki.

[27] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm." in *USENIX Annual Technical Conference*, 2014, pp. 305–319.