Resilience at the Edge in Cyber-Physical Systems

Abhishek Dubey*, Gabor Karsai*, Subhav Pradhan*
*Dept of Electrical Engineering and Computer Science
Vanderbilt University, Nashville,TN 37235, USA

***Abstract*—As the number of low cost computing devices at the edge of communication network increase, there are greater opportunities to enable innovative capabilities, especially in cyber-physical systems. For example, micro-grid power systems can make use of computing capabilities at the edge of a Smart Grid to provide more robust and decentralized control. However, the downside to distributing intelligence to the edge away from the controlled environment of the data centers is the increased risk of failures. The paper introduces a framework for handling these challenges. The contribution of this framework is to support strategies to (a) tolerate the transient faults as they appear due to network fluctuations or node failures, and to (b) systematically reconfigure the application if the faults persist.**

I. Introduction

Cyber-physical systems (CPS), specially related to large societal infrastructures have been mostly self-contained as seen from the perspective of computing resources. For example, the traditional architecture for the Smart Grid is to transfer all SCADA (Supervisory Control and Data Acquisition) data to centralized utility servers [1]. The next evolution in the design of these systems came with cloud computing, when many of the analytics functions were deployed in the cloud [2]. However, even with the availability of on-demand resources in the cloud, the critical CPS often are unable to transfer the time-critical control tasks to the cloud due to communication latencies [3][4]. This centralized SCADA architecture is changing with recent developments like Fog Computing [5][6], which have advertised the use of dual purpose sensing and computation nodes at the edge in the city, closer to the physical phenomenon being observed or analyzed. For example, the SCALE-2 [7] platform provides the capability to run air-quality monitoring sensors, the Paradrop architecture [8] provides the capability to run containerized applications in network routers.

As a direct consequence of the evolution of the computing paradigm from 'central data-centers' to 'shared cloud computing resources' to 'distributed edge computing resources plus shared cloud resources', critical CPS like Smart Grids can distribute the intelligence further down into network, away from the centralized utility servers. For example, this capability provides us the means to build energy management applications of the future that are both distributed and coordinated, with heavy reliance on communication and coordination among local sensing and control algorithms, while also obeying strategic energy management decisions made on a higher level of the control hierarchy. Figure 1 illustrates the concept of a computing platform that our team is currently building.

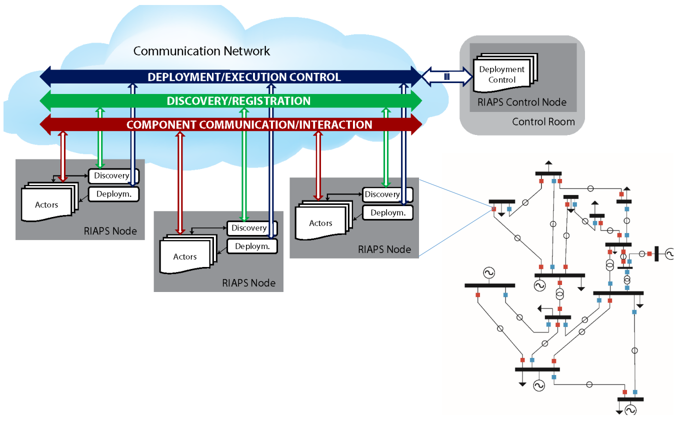

Fig. 1: RIAPS Concept. The figure in the inset on the right shows an IEEE-14 bus concept schematic. Each red and blue node in the figure is a RIAPS node running on am embedded computer. The rest of the figure shows that the RIAPS nodes depend on a number of decentralized, but shared platform services such as deployment execution and control, discovery, and component to component communication. A control room can be optionally used as a central location for application management.

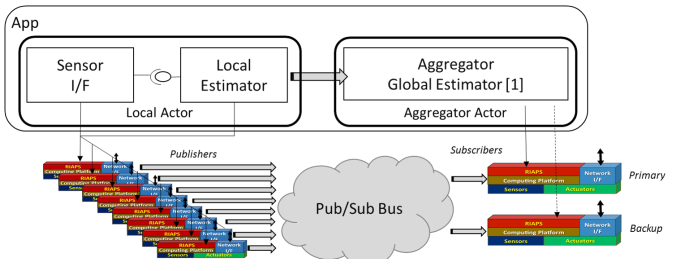

Fig. 2: RIAPS Application Architecture. The example shows two actors. The first actor collects sensor data from a 'sensor' component and then estimates the local system state via the 'local estimator' component. The aggregator actor collects local estimates from a number of local actors via the publish/subscribe communication bus and estimates global state using the global estimator component. The figure also shows that the aggregator actor can be arranged in a primary/backup replica pattern.

This platform called Resilient Information Architecture for Smart Grid (RIAPS) [9][10][11] enables software platform hosted on computing nodes, called 'RIAPS Nodes', on the network that have access to local sensor signals (e.g., they measure voltage and current) and can issue commands to local actuators (e.g., circuit breakers). The nodes can also communicate with each other and a control authority in a central control room via a communication network. Each node is running the RIAPS software that facilitates the execution of various applications.

As seen in Figure 2, an application in RIAPS is a composition of *software components*, with each component implementing one functionality. This architectural principle is used in many applications today, including the Android platform [12] for smartphones and the AUTOSAR standard [13] for Embedded Control Units (ECUs) for cars. In RIAPS, the component framework provides

- the component scheduler, which implements the component execution semantics,
- the component interaction library, which enables publish/subscribe and remote method invocation on the same node or across the network.
- the lifecycle management support that assists in remotely managing the software components,
- the language run-time libraries (e.g., for the C/C++ language),
- and the resource management support to monitor computing platform resource utilization/availability.

These applications are distributed across the system and placed on different computation nodes depending upon the availability of the resources on the node, which includes access to specific sensors and the objective of the application. The computation nodes in this system are of two types: edge nodes and the central control room computation node. The edge nodes are limited in resources, but are directly integrated into the physical devices that are either being measured or controlled by the distributed application. The central computation nodes are not limited in resources but are often far away from the physical phenomena, which affects the network latencies. This architecture is supported by a suite of *platform managers* that run as independent processes and implement system-level management functions.

Contributions: While this architecture exposes a number of interesting challenges and opportunities, we explore the challenge of managing the applications across this system in this paper. Unlike the data center nodes, the edge nodes are exposed to elements and therefore suffer from a higher likelihood of failure. Therefore, it is very important that we first ensure that the applications can be deployed following a fault-tolerant pattern, reducing the need of any drastic redeployment in case of a transient failure. Then, if there is a drastic failure, the system should be able to reconfigure the affected applications without any central input. In this paper, we first describe the conceptual architecture of RIAPS (section II and II-A) and then describe the operations management layer in the CHARIOT architecture [14] (section III) that our team has developed. We are currently working on the integration of the RIAPS and the CHARIOT architecture.

Outline: The outline of the paper is as follows: Section II describes our system architecture of RIAPS. Section II-A describes the design language used to describe RIAPS applications. Runtime management of the system is described in Section III. In that section, we first describe the fault avoidance mechanisms and then we describe the fault recovery mechanisms. Scalability analysis of our approach is presented in Section IV and Section V. Section VI describes the related research. Finally, Section VII concludes the paper.

II. System Architecture

A RIAPS component is a reusable computational unit that has a set of operations that manipulate the individual component's state and that can interact with other components of the application via communication *ports*, which are owned by the component. As shown in Figure 3, a RIAPS component can have four different kinds of ports: *sender*, *receiver*, *client*, *server*. Sender is a single-purpose port because it is used to only send messages outside of a component. The receiver port is used only to bring messages into the component. Both client and server are special ports in that they can be used to send messages and receive messages. This is required to implement the common pattern of request/response, where the "client" can send a request message via a client port, which is then received by the server port of some other component, which then sends a "response" back to the component that originated the request.

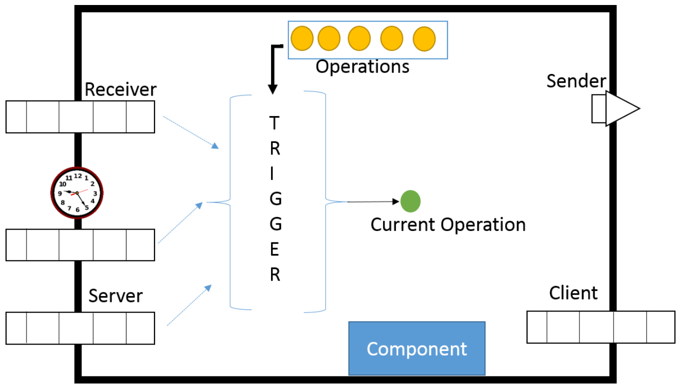

Fig. 3: RIAPS Component Model

Client, server, and receiver ports are individually buffered, i.e. the messages are guaranteed to be held in the port unless the port's buffer is full or a component operation has consumed the message. An additional construct is *timer*: it can be armed and can be used to produce messages that record the time of timer expiry and that triggers a component operation. The computational feature of a component is single threaded execution that is managed by a *trigger* method provided by the component developer. The trigger method can evaluate arbitrary conditions (e.g., the state of the component) and events related to the ports and use the result to select an operation that implements the core business logic of the component. The trigger can be executed (1) when the state of ports of the components change, (2) when a timer expires, or (3) when a component operation is completed.

These components are launched dynamically on the computation nodes based on the deployment rules. The connection between different component ports is managed via a decentralized *discovery service* which keeps track of the different messages that are currently available in the system [11]. When a new component starts, the discovery service informs it of possible matches (for connecting receivers to sender, etc.) and then the component can connect to them. With the help of the discovery service, the RIAPS architecture can support both publish-subscribe and client-server communication patterns.

A. System Description

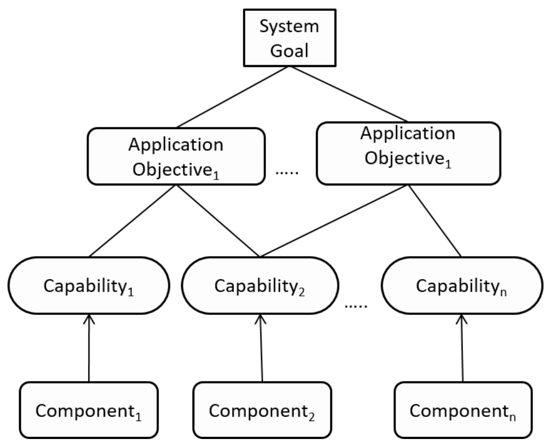

Fig. 4: The goal of a management system is to ensure that all applications are online. Applications require various capabilities, which are provided by different components. A capability can be reused between two applications, and more than one component can provide the same capability.

RIAPS is an example of a platform to support *managed distributed systems* [15]. A managed distributed system can be expressed using a combination of the (1) applications required to be deployed in the system - which describes the set of system objectives, (2) the composition of and the requirements imposed by the applications, and (3) the constraints governing the deployment and (re)configuration of applications. In this approach, an application can be composed of different components providing the same capability - each component only provides one unique capability. Figure 4 illustrates this concept. Dependencies between capabilities and the components are expected. For example, a "state estimator" component can necessitate the presence of a specific "phasor measurement unit". In addition to the functional dependencies the deployment mechanism requires precise specification of computation resources required by a component.

Computation nodes provide the resources in the distributed system. Nodes can be defined using the concept of node templates and node (instances). Node templates describe the resources available in a category of computation nodes, while the nodes are specific instantiation of a node template. We expect that while some nodes will be known a priori, other nodes can be added as the system evolves over time. For example, Figure 5 describes an image processing application composed of two components, one of them identifies a bounding box and the other computes the grid approximation, which can then be used to detect specific objects. Additionally, the specification can provide specific deployment constraints such as distribute and collocate. The distribute specification requires the components mentioned to be always deployed on separate nodes, while the collocate specification requires the components to be always deployed on the same node. Note that we assume that all nodes are part of the same overlay network,

```
package sample {
    struct point
    {
        long x
        long y
    }
    struct quadrilateral{
        sequence<point,4> _vertices
        double area
    }
    struct square{
        sequence<point,4> _vertices
    }
    struct grids{
        sequence<square> _grids
    }
    message<quadrilateral> boundingbox
    message<grids> polyhedra
    component ImageCapture {
        client <boundingbox,polyhedra> port2
        requires regular.camera device
        requires  256 MB memory
        provides capability imaging
    }
    component Approximator {
        server <boundingbox,polyhedra> port2
        requires  256 MB memory
        provides capability processing
    }
    nodeTemplate edge {
        provides 256 MB memory
        device camera {
            artifact cameralibrary located at  '/opt/camera
                .so'
        }
    }
    app riaps_hello {
        components ImageCapture , Approximator
        distribute ImageCapture , Approximator
        ImageCapture => Approximator
    }
    system {
        nodes edge
        node bbb1:edge
        objective riaps_hello
    }}
```

Fig. 5: This listing shows the specification of an image processing application composed of two components, one of them identifies a bounding box and the other computes the grid approximation, which can then be used to detect specific objects. Notice that the application's required capabilities are not modeled explicitly, rather they are derived from the specified components.

i.e. the components running in one node can communicate with components running on the other node.

III. System Management

Given the system description, the management of various applications require handling of: (a) operations management, which includes the system installation and updates, including application updates and computation node updates, (b) transient failures of resources, and (c) long term resource failures. This is implemented by the CHARIOT management architecture, available at [14]. Before we describe the operation management and the mechanism that enables handling of long term resource failures, we will first describe how we can handle transient failures.

A. Handling Transient Failures With Redundancy

While we consider three kinds of failures in our architecture: component failures, node failures and communication failures. Transient failures are typically expected to be communication

```
1
2  package sample {
3      ...
4      system {
5          nodes edge
6          node bbb1:edge
7          objective riaps_hello
8          deploy ImageCapture per edge node
9          deploy Approximator in [5,3] voter
               configuration
10         deploy Approximator in [3,5] replica
               configuration
11     }}
```

Fig. 6: System definition of fig 5 with the redundancy patterns

and node failures that are not long lasting. Redundancy is a typical mechanism to handle such temporary failures. The system can support three kind of redundancy patterns: the (a) voter pattern, (b) active replication pattern, and (c) per-node pattern.

The per-node pattern requires that the associated capability be replicated on a per-node basis (specific to a node of a specific type). Replication of capabilities associated with the other two redundancy patterns is based on their redundancy factor, which can be expressed by either (a) explicitly stating the number of redundant capabilities required or (b) providing a range for the cardinality of the expected resources. The latter requires the associated capability to have a minimum number for redundancy and a maximum number for redundancy, *i.e.*, if the number of component instances providing that capability present at any given time is within the range, the system is still valid and no reconfiguration is required.

Figure 6 illustrates the specification for these patterns. For example, line number 8 states that an imaging capability is local to a computation node and no action is required if that node becomes unavailable. A component interacting with a per-node component can choose to communicate with any of the available instances. Line number 9 implies that the approximator is to deployed in a pattern of $5X3$. That is, 5 instances of approximator will be initialized along with three instances of voters. All voters receive all outputs from the approximator instances, do a majority voting, including a secondary voting among all the voters. Any component that needs information from the approximator must instead receive information from one of the voters. This interaction restriction has to be managed by the discovery service. Given the property of the voter configuration, no action is required until at least 3 component replicas and 1 voter is available in the system. The active replica pattern leaves the redundancy interaction management in the hand of application developer. However, the system specification ensures that a reconfiguration becomes essential if the number of available instances become smaller than the minimum number specified in the configuration.

B. Operations Management

Operations management requires three platform services on each node: the discovery service that we discussed in section II, a deployment service, and a planner service. Additionally, a fault-tolerant database configuration is expected. This database is required to store the current configuration of the system. In addition to maintaining the list of various publishers and subscribers, the discovery service is also responsible for managing a list of available nodes in the system and their node types. This information is persisted in the fault-tolerant database. The deployment service running on each node is responsible for starting up components on the node as instructed by the planner service. The planner service is required to compute a valid system deployment configuration given all the previously active applications in the system, the available resources and all the known, existing failures. It is important to emphasize that the planner service is stateless. It always loads in the current configuration from the database and then communicates the deployment changes to each deployment manager, which upon success, (a) confirms the changes to the planner service, and (b) updates the configuration database. In case of a partial deployment failure, the planner service can be invoked again providing a different solution.

C. Planner Service

The planner service, at a high level, can be considered as a constraint solver whose computed solutions are the feasible configurations. It loads the expected configuration of the system from the database, and then it can compute all the possible instances of all the available components that can be deployed in the system. It should be noted that in the live system that number will be much lower. For example, for the Figure 6, the maximal set of component instances, assuming two instances of edge node, *bbb*1 and *bbb*2 (discovered at runtime and not initially specified in Figure 6) are:

- ImageCapture_bbb1 and ImageCapture_bbb2, which are calculated by computing 1 instance per known node, see line 8 in Figure 6.
- Approximator_Voter1, Approximator_Voter2, Approximator_Voter3, Approximator_Voter4, Approximator_Voter5, are deployed because the specification asks for $[5, 3]$ voter configuration, see line 9 of Figure 6.
- Voter_Approximator1, Voter_Approximator2, and Voter_Approximator3 are instantiated to satisfy the 3 voter requirement as specified by line 9 of Figure 6.
- Approximator_replica1, Approximator_replica2, Approximator_replica3, Approximator_replica4, and Approximator_replica5 are 5 active replicas that were instantiated because of line 10 in Figure 6. The rule is to instantiate maximum number of replicas at initialization. The number 3 in line 10's range specifies that no reconfiguration is needed if atleast 3 replicas remain active.

The next phase for the planner involves encoding deployment constraints. For this purpose, the planner uses a Z3 based constraint system described in [16]. The constraints capture the following:

1) Constraints to ensure that component instances that must be deployed are always deployed.
2) Constraints to ensure that component instances that communicate with each other are either deployed on the same node or on nodes that have network links between them.

3) Constraints to ensure that the resource's provided-required relationships are valid.
4) Constraints to ensure that we do not seek reconfiguration if the minimum number of component instances as expressed by the semantics of the voter replication or the lower bound of the active replica are still active.
5) Constraints to represent failures, such as node failure or device failures. This is important to ensure we do not use the nodes or devices that have failed.
6) Constraints to handle deployment restrictions referring to distribution and collocation.

The planner encodes these constraints and the knowledge of component instances and the available nodes as a matrix of decision variables. A C2N (component to node) matrix comprises rows that represent component instances and columns that represent nodes; the size of this matrix is $\alpha \times \beta$, where α is the number of component instances and β is the number of available nodes. Each element of the matrix is encoded as an integer variable whose value can either be 0 or 1. A value of 0 for an element means that the corresponding component instance (row) is not deployed on the corresponding node (column). Conversely, a value of 1 for an element indicates deployment of the corresponding component instance on the corresponding node. Similarly other matrices are used to encode the resource availability of nodes (updated dynamically using the monitors). Communication resource requirements are encoded using a square matrix $\beta \times \beta$. Thereafter, the placement constraints are written in terms of resources required and net resources available. Additional constraints for redundancy and collocation are also added. Each feasible solution of this constraint satisfaction problem is a valid system placement for all applications. The system is marked unstable if no configuration is found. If no answer is found, then depending upon criticality, the variables for applications can be pruned from the problem set until a valid solution is found.

D. Implementation Architecture

Figure 7 describes the implementation architecture, which consists of a three-layered architecture stack consisting of a design layer (top), a data layer (middle), and a management layer (bottom). The design layer comprises the modeling language used to describe the system as shown in figures 5 and 6. The data layer is implemented using a persistent data storage tool (MongoDB [17]) and the corresponding schema to store system information, which includes the JSON representation of the system design models and a run-time representation of the system. This layer provides a canonical format to represent information about the system under management and therefore it decouples the design layer at the top from the management layer at the bottom. The management layer comprises monitoring and deployment infrastructures, as well as the planner described in Section III-C.

IV. Results

To demonstrate and evaluate the planner we performed experiments of different scales in different setup.

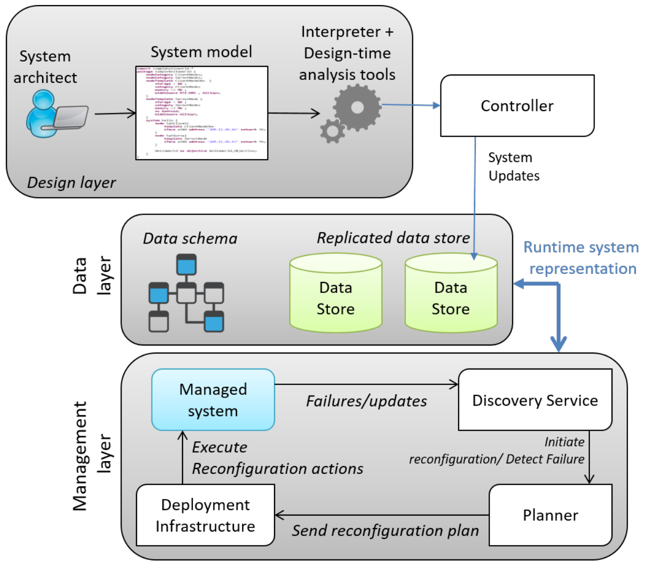

Fig. 7: The operations management architecture. The controller is responsible for adding new applications or updating the existing application configuration.

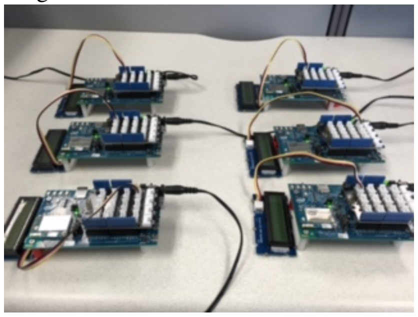

Fig. 8: The test bed comprising six Intel Edison compute modules, each mounted on an Arduino breakout board. Each board also has a Grove LCD screen attached to it.

A. Small-scale Experiment on IoT Devices

First, we performed a small scale test on six Intel Edison compute modules, with each module attached to an Arduino breakout board (see Figure 8). Each Edison module runs Yocto Linux operating system on a 500 MHz dual-core, dual-threaded Intel Atom CPU with 1GB RAM and 4GB Flash storage. Each board communicates with each other over a wireless network established using a single router. We ran a planner on one of the Edison nodes. However, since the planner is stateless it can be dynamically started on any of the nodes.

For this small scale experiment, we used two examples. The first example is a *parking management system* that is capable of tracking the availability of parking spaces and handling client requests. To achieve the goal, the system must support two applications: (1) an *OccupancyChecking*, and (2) a *ClientInfo*. The *OccupancyChecking* application is comprised of two capabilities: imaging and detection. The imaging capability is implemented by an occupancy sensor component, which is installed using the per-node replication. The detection capability is provided by the parking manager,

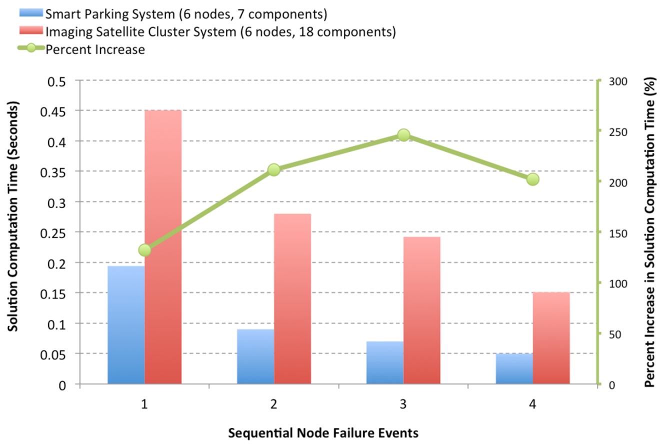

Fig. 9: Time taken by the planner to compute solution for four sequential node failure events in the context of the two examples.

which is not replicated. Deploying this example in the testbed of six Intel Edison nodes results in a total of seven component instances; six instances of the *OccupancySensorComp* component as it is replicated per node, and a single instance of the *ParkingManagerServer* component. This means that the size of the C2N matrix for this example is 7×6 as we have seven component instances and six nodes. Since the instances for *OccupancySensorComp* component are strictly associated with the nodes they are deployed on, they do not need to be migrated when failures occur.

The second example is an *imaging satellite cluster* system, first described in [16]. Unlike the parking system, which only has seven component instances, this system has a total of eighteen component instances. In general, this system comprises of three applications: (1) imaging , which is responsible for using cameras with different resolutions to capture and GPU-s to process captured images, (2) cluster flight planning application, which is responsible for receiving encoded ground commands, processing each command and calculating flight plan (new target location) for each satellite in the cluster, and (3) satellite flight application which is responsible for receiving aforementioned target locations and controlling satellite thrusters to move towards that position.

Figure 9 shows the time taken to compute solution in response to four different but sequential node failure events in the two examples. Even though this experiment is small-scale (six nodes) and the increase in system size is not significant between the two use cases (the second use case, has eleven more component instances than the first use case), as shown in Figure 9, the increase in solution computation time for same node failure events ranges from 132% for *event 1* to 245% for *event 2*. It is important to note that the increase in solution computation time cannot solely be attributed to the system size as it also depends on the number of constraints being encoded.

B. Performance Analysis

To determine any possible performance bottleneck, the planner was further analyzed using initial deployment scenarios

TABLE I: Different initial deployment scenarios used for planner performance analysis

Scenario	Description
1	11 nodes, 10 components
2	22 nodes, 18 components
3	33 nodes, 26 components
4	44 nodes, 34 components
5	55 nodes, 42 components
6	66 nodes, 50 components

TABLE II: Total solution computation time for initial deployment of scenarios in Table I broken down into different phases. Numbers presented are average over three executions.

Scenario	Phase-1 time (s)	Phase-2 time (s)	Phase-3 time (s)	No. of SMT assertions
1	0.0053	1.2797	0.158	3065
2	0.0087	13.5017	1.0243	33401
3	0.0117	58.1427	4.1473	144741
4	0.015	168.04	16.389	419849
5	0.0177	390.2207	42.16	970529
6	0.021	777.7423	60.973	1937625

with varying scale. Table I describes these initial deployment scenarios. As shown in the table, the complexity of initial deployment scenarios used were increased linearly.

For our analysis presented in this section, we divided the planner algorithm into three phases. The *problem setup time* is divided into two distinct phases: (1) Phase 1, which is the instance computation phase where the planner determines the component instances that need to be deployed based on dependency and redundancy requirements, and (2) Phase 2, which is the constraint encoding phase that is executed by the planner every time it is invoked since the planner is stateless. The *Z3 solver time* is the third and final phase.

For each initial deployment scenarios presented in Table I, Table II presents a breakdown of the total solution computation time with respect to the three phases described above. For each scenario, the total number of SMT assertions used by the planner is also presented.

From Table II it is clear that the majority of the total solution computation time is taken by Phase 2, which is the constraint encoding phase. To further investigate this, we analyzed the time taken to encode different constraints. Since our initial tests pointed to the dependency constraint[1] as being a major bottleneck, we divided Phase 2 into two distinct phases: (1) dependency constraint encoding phase, and (2) other constraints encoding phase. Results of this is presented in Table III. These results clearly show that the dependency constraint encoding mechanism is a major bottleneck as it accounts for more than 90% of the constraint encoding time. This is because for every dependency, the current encoding mechanism incurs $O(n^2)$ time complexity. Any improvement to the way in which this constraint is encoded will result in significant reduction of the total solution computation time. One possible solution could be storing the encoded results in the database rather than always encoding the constraints as discussed earlier. We are

[1]Dependency constraint is the constraint used to ensure that component instances that communicate with each other are either deployed on the same node or on nodes that have network links between them

TABLE III: Breaking down constraint encoding phase (Phase 2 in Table II) into dependency constraint encoding phase and other constraints encoding phase.

Scenario	Dependency constraint encoding phase (s)	Other constraints encoding phase (s)
1	1.159	0.1207
2	13.1673	0.3343
3	57.486	0.6567
4	166.9337	1.1063
5	388.525	1.6957
6	775.6133	2.129

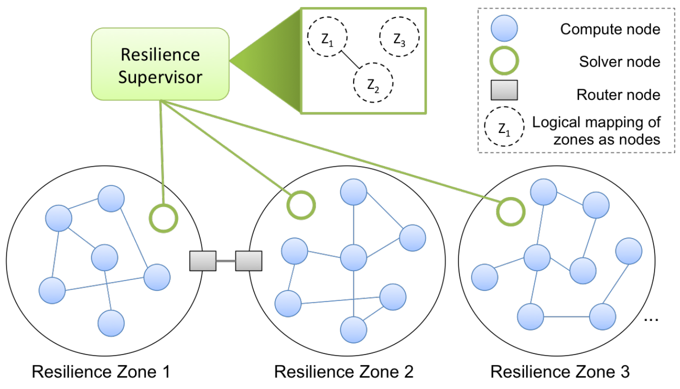

Fig. 10: Overview of the Multi-zone Resilience Mechanism comprising a two-layer architecture. Each resilience zone comprises (1) multiple compute nodes on which applications are deployed, and (2) a solver node, which hosts the planner service described in Section III-C. Nodes of a resilience group can have varying degree of interconnection, which are taken into account by the solver when computing (re)configuration solutions. Furthermore, multiple resilience zones themselves can have interconnection via designated router nodes.

currently improving the implementation to take care of this issue.

V. Extending to very large deployments

In this section we describe our ongoing work on implementing a hierarchical approach for handling very large deployment scenarios (≥ 100 nodes). This is required due to the increase in computation time taken to formulate a solution in larger deployments. Our approach depends on creating a two-layered architecture comprising a *resilience supervisor* in the first layer and multiple resilience zones in the second layer. A planner (Section III-C) manages application on the computation nodes within that zone.

The resilience supervisor, as shown in Figure 10, is connected to solver nodes associated with each available resilience zone. When this connection is first established, the resilience supervisor acknowledges the existence of corresponding resilience zones. Furthermore, the resilience supervisor maintains a mapping of different zones and their interconnection as logical nodes with edges. This allows the resilience supervisor to use the same (re)configuration logic as that in the single-zone mechanism. The only and critical difference is the level of abstraction associated with the concept of nodes with respect to the C2N matrix described in Section III-C. In a single-zone scenario, a node refers to a physical node present in that zone, whereas, a node in the resilience supervisor refers to a resilience zone.

The above-described multi-zone architecture will work in the following way. Initially, when applications are deployed, they are deployed to their target resilience zones; the resilience supervisor has no role to play here. However, when failures or anomalies occur and the corresponding management engine cannot find a solution (using the mechanisms described in Section III-C), it notifies the resilience supervisor. Upon this notification, the resilience supervisor runs the same mechanism to determine if the failed applications can be moved to a different resilience zone. The solution computed by the resilience supervisor is then sent to management engines in the corresponding zones. For this approach to work, the resilience supervisor needs to keep track of resources available in different resilience zones. Another important point to note here is that the solution provided by the resilience supervisor is not always valid or satisfiable as it is a high-level solution computed in terms of resilience zones (recall that nodes in context of the resilience supervisor are resilience zone). If a solution given by the resilience supervisor is not valid, the corresponding management engine that received the solution invokes the supervisor again after adding a constraint to ensure that same solution will not be computed again.

VI. Related Work

In [18], [19], the authors present solutions for synthesizing an optimal assembly for component-based systems, given a set of constraints. Both solutions perform automatic static assembly at design-time. The key difference between these two solutions is that [18] does not consider timing constraints as it does not target real-time systems, however, the solution presented in [19] targets cyber-physical architectures and therefore considers timing/scheduling constraints. Pseudo-Boolean Satisfiability and optimization (PBSAT) solver is used in the former, while the latter uses Integer linear programming Modulo Theories (IMT) solver. These solutions do not meet our needs, as they do not consider dynamic reconfiguration and only focus on automatically synthesizing optimal system assemblies at design-time.

Significant amount of prior work has been done in order to achieve dynamically reconfiguring/self-adapting systems. Work appearing in [20], [21] present different policy-based approaches to achieving dynamic reconfiguration. In [20], the authors present a policy-based framework that requires mission specification, which describes how specific roles are assigned to different nodes based on their credentials and capabilities, and how these roles should be re-assigned in response to changes or failures. As such, this mission specification explicitly encodes reconfiguration actions, i.e. role re-assignments, during design-time. In [21], the authors also follow similar approach where declarative policies are used to specify adaptation. These approaches are different from ours, as we do not explicitly encode reconfiguration actions at design-time; it is impossible to cover all possible combinations of failure scenarios at design-time.

In [22], the authors present a middleware that supports timely reconfiguration in distributed real-time systems based on services. *Application Graph*, which contains information about what services are required and how they depend on each other, and *Expanded Graph*, which contains information about different service implementations, are studied a priori at design-time in order to analyze the schedulability and complexity of these graphs and to perform fine tuning to bound the sources of unpredictability, if required. The resulting *Scheduled Expanded Graph* is used at run-time to determine the *Execution Graph*, which represents the application in execution. Although this solution supports predictability, schedulability analysis is done at design-time, which means system resources cannot be modified at run-time. This is important for dynamic systems where resources can added or removed at run-time.

VII. Conclusions

Distributed embedded systems that are implementing the fog computing paradigm are expected be managed, yet also exhibit autonomous capabilities for fault tolerance and resilience. In this paper, we have introduced an application platform for component-based applications and a management framework for (a) deployment and (b) fault management. Arguably, both of these are critical for the success of these novel systems due to the sheer size and complexity, which cannot be addressed by brute force approaches. These concepts have been and are being prototyped in the CHARIOT and RIAPS platforms, respectively. Our application domains clearly necessitate these techniques. There are two main concerns that need to be addressed in future research. One is scaling; i.e. how does the approach scale to hundreds or thousands of nodes. Another one is real-time guarantees: in critical systems, fault management often has to be performed under strict timing constraints. The challenge is to model and analyze the behavior of systems, and build technology that addresses these concerns.

Acknowledgment: The work presented herein was funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0000666 and in part by 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] H. L. Storey, "Implementing an integrated centralized model-based distribution management system," in *2011 IEEE Power and Energy Society General Meeting*, July 2011, pp. 1–2.

[2] Y. Simmhan, S. Aman, A. Kumbhare, R. Liu, S. Stevens, Q. Zhou, and V. Prasanna, "Cloud-based software platform for big data analytics in smart grids," *Computing in Science Engineering*, vol. 15, no. 4, pp. 38–47, July 2013.

[3] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, ser. MCC '12. New York, NY, USA: ACM, 2012, pp. 13–16. [Online]. Available: http://doi.acm.org/10.1145/2342509.2342513

[4] A. Botta, W. De Donato, V. Persico, and A. Pescapé, "On the integration of cloud computing and internet of things," in *Future Internet of Things and Cloud (FiCloud), 2014 International Conference on*. IEEE, 2014, pp. 23–30.

[5] S. Yi, C. Li, and Q. Li, "A survey of fog computing: Concepts, applications and issues," in *Proceedings of the 2015 Workshop on Mobile Big Data*, ser. Mobidata '15. New York, NY, USA: ACM, 2015, pp. 37–42. [Online]. Available: http://doi.acm.org/10.1145/2757384.2757397

[6] O. C. A. W. Group *et al.*, "Openfog architecture overview," *White Paper, February*, 2016.

[7] K. Benson, C. Fracchia, G. Wang, Q. Zhu, S. Almomen, J. Cohn, L. D'arcy, D. Hoffman, M. Makai, J. Stamatakis, and N. Venkatasubramanian, "Scale: Safe community awareness and alerting leveraging the internet of things," *IEEE Communications Magazine*, vol. 53, no. 12, pp. 27–34, Dec 2015.

[8] D. Willis, A. Dasgupta, and S. Banerjee, "ParaDrop: A Multi-tenant Platform to Dynamically Install Third Party Services on Wireless Gateways," in *Proceedings of the 9th ACM workshop on Mobility in the evolving internet architecture*. ACM, 2014, pp. 43–48.

[9] G. Karsai, A. Dubey, I. Madri, M. Metelko, P. Volgyesi, J. Sallai, S. Lukic, D. Lubkeman, A. Srivastava, C.-C. Liu, J. Xie, and P. Banerjee, "Analysis Results and Initial Design of the Resilient Information Architecture Platform for the Smart Grid (RIAPS)," Jul. 2016.

[10] "Resilient Information Architecture for Smartgrid," https://riaps.isis.vanderbilt.edu/redmine/projects/riaps.

[11] S. Eisele, I. Madari, A. Dubey, and G. Karsai, "RIAPS:Resilient Information Architecture Platform for Decentralized Smart Systems," in *20th IEEE INTERNATIONAL SYMPOSIUM ON REAL-TIME COMPUTING*, IEEE. Toronto, Canada: IEEE, 05/2017 2017.

[12] A. Developers, "What is android," 2011.

[13] Autosar GbR, "AUTomotive Open System ARchitecture," http://www.autosar.org/. [Online]. Available: http://www.autosar.org/

[14] CHARIOT. [Online]. Available: https://github.com/visor-vu/chariot

[15] G. Karsai, D. Balasubramanian, A. Dubey, and W. Otte, "Distributed and managed: Research challenges and opportunities of the next generation cyber-physical systems," in *17th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2014, Reno, NV, USA, June 10-12, 2014*, 2014, pp. 1–8.

[16] S. Pradhan, A. Dubey, T. Levendovszky, P. S. Kumar, W. A. Emfinger, D. Balasubramanian, W. Otte, and G. Karsai, "Achieving resilience in distributed software systems via self-reconfiguration," *Journal of Systems and Software*, vol. 122, pp. 344 – 363, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121216300590

[17] MongoDB Incorporated, "MongoDB," http://www.mongodb.org, 2009.

[18] P. Manolios, D. Vroon, and G. Subramanian, "Automating component-based system assembly," in *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 2007, pp. 61–72.

[19] C. Hang, P. Manolios, and V. Papavasileiou, "Synthesizing cyber-physical architectural models with real-time constraints," in *Computer Aided Verification*. Springer, 2011, pp. 441–456.

[20] E. Asmare, A. Gopalan, M. Sloman, N. Dulay, and E. Lupu, "Self-management framework for mobile autonomous systems," *Journal of Network and Systems Management*, vol. 20, no. 2, pp. 244–275, 2012.

[21] A. Schaeffer-Filho, E. Lupu, and M. Sloman, "Federating policy-driven autonomous systems: Interaction specification and management patterns," *Journal of Network and Systems Management*, pp. 1–41, 2014.

[22] M. G. Valls, I. R. López, and L. F. Villar, "iland: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems," *Industrial Informatics, IEEE Transactions on*, vol. 9, no. 1, pp. 228–236, 2013.