CHARIOT: A Holistic, Goal Driven Orchestration Solution for Resilient IoT Applications

Subhav Pradhan†, Abhishek Dubey†, Shweta Khare†, Saideep Nannapaneni⋆, Aniruddha Gokhale†, Sankaran Mahadevan⋆, Douglas C Schmidt†, Martin Lehofer‡

†Dept of Electrical Engineering and Computer Science, Vanderbilt University, Nashville,TN 37235, USA
⋆Dept of Civil and Environmental Engineering, Vanderbilt University, Nashville,TN 37235, USA
‡Siemens Corporate Technology, Princeton, NJ, USA

ABSTRACT

An emerging trend in Internet of Things (IoT) applications is to move the computation (cyber) closer to the source of the data (physical). This paradigm is often referred to as *edge computing*. If edge resources are pooled together they can be used as decentralized shared resources for IoT applications, providing increased capacity to scale up computations and minimize end-to-end latency. Managing applications on these edge resources is hard, however, due to their remote, distributed, and possibly dynamic nature, which necessitates autonomous management mechanisms that facilitate application deployment, failure avoidance, failure management, and incremental updates. To address these needs, we present CHARIOT, which is an orchestration middleware capable of autonomously managing IoT systems consisting of edge resources and applications. CHARIOT implements a three-layer architecture. The topmost layer comprises a system description language; the middle layer comprises a persistent data storage layer and the corresponding schema to store system information; and the bottom layer comprises a management engine, which uses information stored persistently to formulate constraints that encode system properties and requirements thereby enabling the use of Satisfiability Modulo Theories (SMT) solvers to compute optimal system (re)configurations dynamically at runtime. This paper describes the structure and functionality of CHARIOT and evaluates its efficacy as the basis for a smart parking system case study that uses sensors to manage parking spaces.

1. INTRODUCTION

Emerging trends and challenges. Popular IoT ecosystem platforms, such as Beaglebone Blacks, Raspberry Pi, Intel Edison and other related technologies like SCALE [5] and Paradrop[33], provide new capabilities for data collection, analysis, and processing at the *edge* [32] (also referred to as Fog Computing [6]). When pooled together, edge resources can be used as decentralized shared resources that host the data collection, analysis, and actuation loops of IoT applications. Examples of such applications include air quality monitoring, parking space detection, and smart emergency response. In this paper, we refer to the combination of remote edge resources and applications deployed on them as *IoT systems*. These IoT systems provide the capacity to scale up computations as well as minimize end-to-end latency, which makes them well-suited to support novel use cases for smart and connected communities.

While the promise of the IoT paradigm is significant, several challenges must be resolved before they become ubiquitous and effective. Conventional enterprise architectures use centralized servers or clouds with static network layouts and a fixed number of devices without sensors and actuators to interact with their physical environment. In contrast, edge deployment use cases raise key challenges not encountered in cloud computing, including (1) handling the high degree of dynamism arising from computation and communication resource uncertainty, and (2) managing resource constraints imposed due to the cyber-physical nature of applications and system hardware/software components.

Computation resource uncertainty in IoT systems stems from several factors, including increased likelihood of failures, which are in turn caused by increased exposure to natural and human-caused effects, as well as dynamic environments where devices can join and leave a system at any time. Communication resource uncertainty is caused by network equipment failure, interference, or due to the mobile nature of some systems (*e.g.*, swarm of drones or fractionated satellites). Unlike traditional enterprise architectures, whose resource constraints narrowly focus on only CPU, memory, storage and network, IoT systems must be able to express and satisfy more stringent resource constraints due to their cyber-physical nature, such as their deployment on resource-limited sensors and actuators.

Even under aforementioned uncertainties and constraints, IoT systems must be capable of managing their applications to ensure maximum availability, especially since these applications are often mission-critical. Each application deployed for a mission has specific goal(s) that must be satisfied at all times. IoT systems should therefore be equipped with mechanisms that ensure all critical goals are satisfied for as long as possible, *i.e.*, they must be resilient by facilitating failure avoidance, failure management, and operations management to support incremental hardware and software changes over time. Moreover, since IoT systems comprising edge resources are often remotely deployed, resilience mechanism should be autonomous to ensure availability and cost-effective management.

Solution approach → Autonomous resilience management mechanisms. To address the challenges described above, IoT systems need autonomous mechanisms that enable the analysis and management of (1) the overall system goals describing the required applications, (2) the composition and requirements of applications, and (3) the constraints governing the deployment and (re)configuration of applications. This paper describes a holistic solution called *Cyber-pHysical Application aRchItecture with Objective-based reconfiguraTion* (CHARIOT), which is an orchestration middleware that supports the autonomous management of remotely deployed IoT systems. Specifically, CHARIOT uses the analysis and management capabilities outlined above to provide services for initial application deployment, failure avoidance, failure management, and operations management. CHARIOT implements a three-layered architecture stack consisting of a design layer, a data layer, and a management layer, as shown in Figure 1.

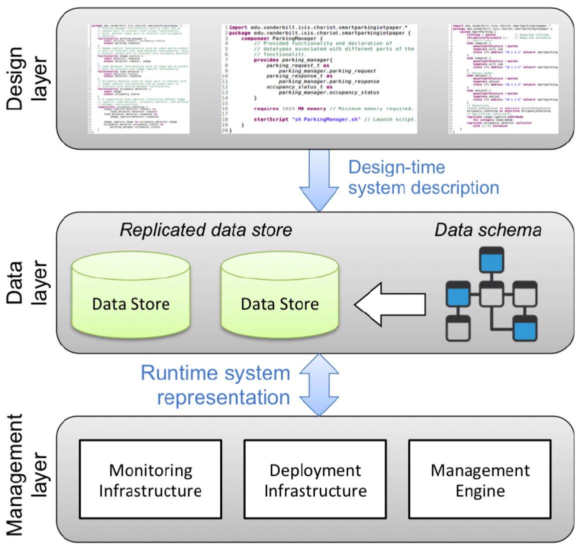


Figure 1: The Layered Architecture of CHARIOT.

Contribution 1: *A generic system description language.* At the top of CHARIOT's stack is a design layer implemented via a generic system description language. This layer captures system specifications in terms of different types of available hardware resources, software applications, and the resource provided/required relationship between them. As shown in Section 4.1, CHARIOT implements this layer using a *domain-specific modeling language* (DSML) called CHARIOT-ML whose goal-based system description approach yields a generic means of describing complex IoT systems. This approach extends our prior work [26, 27] by (1) using the concept of component types (instead of specific implementations) to enhance flexibility and (2) supporting a suite of redundancy patterns.

Contribution 2: *A schema for persistent storage of system information.* In the middle of CHARIOT's stack is a data layer implemented using a persistent data store and the corresponding well-defined schema to characterize system information, which includes a design-time system description and a runtime representation of the system. This layer canonicalizes the format in which information about an IoT system is represented. We describe this contribution further in Section 4.2.

Contribution 3: *A management engine to facilitate autonomous resilience.* The bottom of CHARIOT's stack is a management layer the supports monitoring and deployment mechanisms, as well as a novel management engine that facilitates application (re)configuration as a means to support autonomous resilience. This management engine uses IoT system information stored in CHARIOT's data layer to formulate Satisfiability Modulo Theories (SMT) constraints that encode system properties and requirements, enabling the use of SMT solvers (such as Z3 [8]) to dynamically compute optimal system (re)configuration at runtime. We describe this contribution further in Section 4.3.

Contribution 4: *Distributed implementation and evaluation of CHARIOT.* CHARIOT uses MongoDB [23] as a persistent storage service, ZooKeeper [1] as a coordination service to facilitate group-membership and failure detection, and ZeroMQ [13] as a high-performance communication middleware. We describe this contribution in Section 5, which also describes a smart-parking application that serves as a case study to present experimental evaluation that shows how CHARIOT's orchestration middleware capabilities are suitable to manage edge computing for IoT systems.

Paper organization. The remainder of this paper is organized as follows: Section 2 describes the problem at hand; Section 3 compares our work on CHARIOT with related work; Section 4 presents our first three contributions by describing the CHARIOT solution in detail; Section 5 presents our fourth contribution by describing an implementation of CHARIOT and evaluating this implementation; and Section 6 presents concluding remarks and future work.

2. PROBLEM DESCRIPTION

This section describes the problem we address with our work presented in this paper. We focus on IoT systems comprising clusters of heterogeneous nodes that provide computation and communication resources, as well as a variety of sensors and actuators. Cluster membership can change over time due to failures, or addition and removal of resources.

This distributed platform supports the needs of IoT applications, which may span multiple nodes due to the availability of resources, *e.g.*, some nodes may have sensors, some may have actuators, some may have the computing or storage resources, and some need more than the processing power available on one node. These IoT applications are composed of loosely connected, interacting components [11], running on different processes, as shown in Figure 2.

A component provides a certain functionality and may require one or more functionalities[1] via its input and output ports. The same functionality can be provided by different components. These *provided* and *required* relations between components and functionalities establish dependencies between components. IoT applications can thus be assembled from components that provide specific services. Likewise, components may be used (or reused) by many active applications. Moreover, the cluster of computing nodes can host multiple applications concurrently.

An IoT system running in a CHARIOT-based distributed

[1]In this context, functionalities are synonymous to services or capabilities associated with a component.

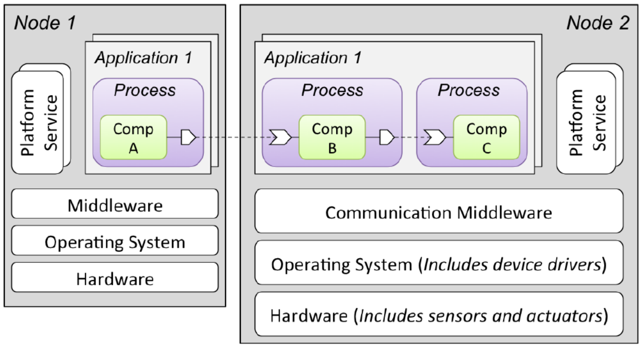


Figure 2: A Component-based IoT Application Model.

platform must manage the resources and applications to ensure that functionalities provided by application components are always available. This capability is important since IoT applications are often mission-critical, so functionalities required to satisfy mission goals must be available as long as possible. This notion of functionality requirement can also be hierarchical, *i.e.*, high-level functionality may be further divided into sub-functionalities.

The possibility of having hierarchical functionalities results in a *functionality tree*, which distinguishes between functionalities that can be divided into sub-functionalities and functionalities that cannot be decomposed further. The latter represents a leaf of the tree and should always map to one or more application components. Although each component provides a single functionality, the same functionality can be provided by multiple components.

The requirement relationship between each parent and its children at every level of this functionality tree can be expressed using a boolean expression [24, 15] that yields an *and-or* tree. Additional resource and implicit dependency constraints between components may arise due to system constraints. Examples of these system constraints include (1) availability of memory and storage capacity for components to use, (2) availability of devices and software artifacts (libraries) for components to use, and (3) network links between nodes of a system, which restricts deployment of component instances with inter-dependencies.

2.1 A Representative IoT System Case Study

Consider an indoor parking management system installed in a garage. This case study focuses on the vacancy detection and notification functionality. This system is designed to simplify clients' use of parking facilities by tracking the availability of spaces in a parking lot and servicing client parking requests by determining available parking spaces and assigning a specific parking space to a client. We use this system as a running example throughout the rest of this paper to explain various aspects of CHARIOT. Figure 3 visually depicts this IoT system, which consists of a number of pairs of camera nodes (wireless camera) and processing nodes (Intel Edison module mounted on Arduino board)[2] placed on the ceiling to provide coverage for multiple parking spaces. Each pair comprising a camera and a processing node is connected via a wired connection. In addition, the parking lot has an entry terminal node that drivers interact with as they enter the parking lot.

[2]https://www.arduino.cc/en/ArduinoCertified/IntelEdison

In addition to the hardware devices that comprise the system, Figure 3 also shows a distributed application consisting of five different types of components deployed on the hardware outlined above and described below:

- *ImageCapture* component, which runs on a camera node and periodically captures an image and sends it to an *OccupancyDetector* component that runs on a processing node.
- *OccupancyDetector* component, which detects vehicles in an image and determines occupancy status of parking spaces captured in the image.
- *LoadBalancer* component, which keeps track of the different *OccupancyDetector* components available. Before an *ImageCapture* component can send images to an *OccupancyDetector* component, it must first find the *OccupancyDetector* by using the *LoadBalancer* component, which also runs on a processing node.
- *ParkingManager* component, which keeps track of occupancy status of the entire parking lot. After an *OccupancyDetector* component analyses an image for occupancy status of different parking spaces, it sends the result to the *ParkingManager* component, which also runs on a processing node.
- *Client* component, which runs on the entry terminal and interacts with users to allow them to use the smart parking application.

2.2 Problem Statement

As mentioned before, IoT systems are dynamic; the degree of dynamism can vary from one system to another. For example, the smart parking example presented in Section 2.1 is an example of a less dynamic system since the physical resources are spatially static and any dynamism is related to system update associated with addition or removal of resources. A cluster of drones or fractionated satellites, however, is an example of highly dynamic systems. Regardless of the degree of dynamism, support for autonomous resilience is of high importance to every IoT system. For example, it is essential to ensure that the *ParkingManager* component is not a single point of failure, *i.e.*, the smart parking system should not fail if the *ParkingManager* component fails.

Addressing the problems described above requires orchestration middleware that holistically addresses both (1) the design-time challenges of capturing the system description and (2) the runtime challenges of designing and implementing a solution that facilitates failure avoidance, failure management, and operations management.

1. *Failure avoidance* is necessary for scenarios where failures must be tolerated as long as possible without having to manage them. This capability is important for systems that cannot withstand reconfiguration downtime. Although failures cannot be avoided altogether, we require mechanisms to avoid failure management.
2. *Failure management* is needed to minimize downtime due to failures that cannot be avoided, including failures caused by unanticipated changes. The desired solution should ensure all application goals are satisfied for as long as possible, even after failures.
3. *Operations management* is needed to minimize the challenges faced when intentionally changing or evolving an existing IoT system, *i.e.*, these are *anticipated* changes. A solution for this should consider changes in hardware

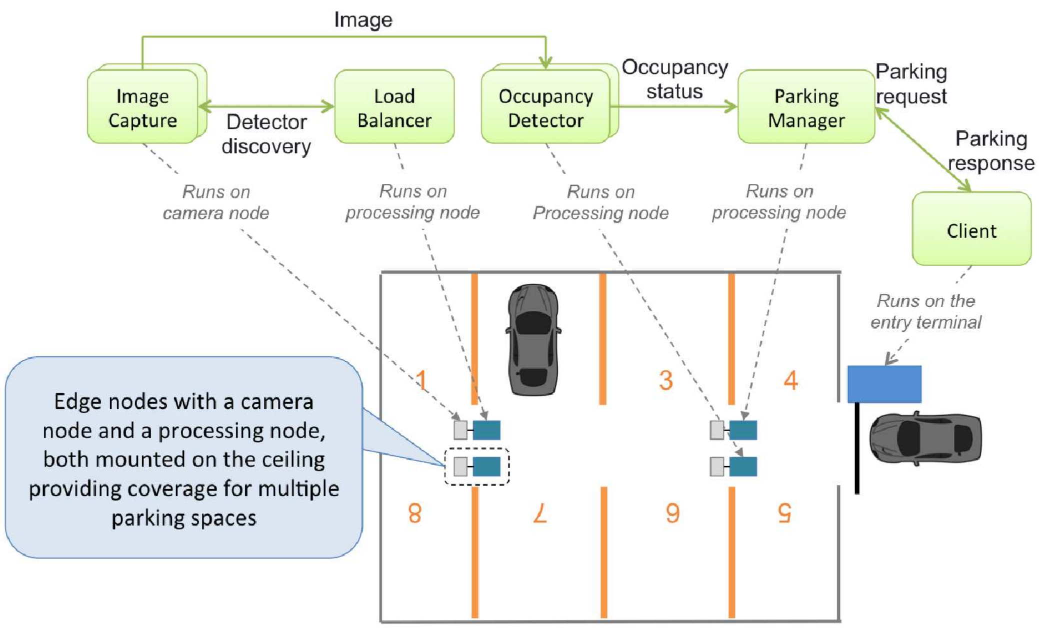


Figure 3: An Overview of the Parking Management System Case Study.

components, as well as software applications and middleware components.

3. RELATED WORK

This section summarizes related work and distinguishes it from our research on CHARIOT presented in the remainder of this paper.

3.1 Redundancy-based Strategies

Fault tolerance in computing has a long history, but resilience [17] is beyond the capabilities of conventional fault-tolerant approaches since resilience means providing the services even if any part of the system fails, which requires adaptation. Conventional fault tolerance techniques are based on redundancy together with *comparison* (*e.g.*, a voter) or *acceptance checking* schemes to decide if a component is functioning correctly. Redundancy-based techniques mask certain classes of persistent and transient faults that may develop in one or more (but not in all) redundant components at the same time, thereby ensuring that faults do not lead to eventual system or subsystem failures. These techniques rely on the assumption that failure of a component is an independent event. Hence, the failure probability of the overall system or subsystem is lower since it is a product of the failure probabilities of the individual components. Other well-known redundancy techniques include recovery blocks and self-check programming [30]. None of these methods are sufficient, however, for IoT systems where both software and hardware topologies can change dynamically.

3.2 Reconfiguration-based Strategies

Reconfiguration-based techniques provide an alternative to the redundancy-based strategies described above. The goal of reconfiguration is to detect anomalous behavior, perform diagnosis to identify the fault cause(s) responsible for the detected anomalies, and apply remedies to restore the functionalities affected by anomalies. These techniques can be configured to account for anomalous behavior and their cascading effects due to faults identified at design time, as well as latent bugs, common mode failures, or other unforeseen events or attacks that disrupt the nominal operation. Moreover, these approaches can be applied to augment system resilience when redundancy-based fault tolerance strategies are already in place.

Anomaly detectors can be based on observing different system aspects, such as heartbeats of nodes and applications, resource utilization of the hosted applications, or unexpected perturbations of application data. These observations are periodically compared against preset values or thresholds, model outputs, or expected behaviors. Diagnosis schemes can use the status of these anomaly monitors to localize and isolate the fault source(s). Anomaly detectors can also employ a hierarchical approach, as well as consensus-based schemes between multiple independent observers. Our prior work [19, 22] on anomaly detection and diagnosis forms the basis for the diagnosis system we use. However, we should point out that diagnosis is not the focus of this paper and therefore is not discussed further.

There are two types of reconfiguration-based techniques: *offline strategies* using pre-specified reconfiguration rules and *online strategies* using dynamic reconfiguration where solutions are computed/searched for at runtime.

3.2.1 *Offline Strategies*

In [21, 10] the authors present two 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 solutions is that [21] does not consider timing constraints, whereas the solution in [10] targets scheduling constraints in cyber-physical systems. Neither of these solutions meet the needs of IoT systems, however, since they do not consider dynamic reconfiguration and focus solely on automatically synthesizing optimal system assemblies at design-time.

The work appearing in [4, 28, 3] presents different policy-based approaches. In [4], 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, as well as how these roles should be reassigned in response to changes or failures. This mission specification explicitly encodes reconfiguration actions, *e.g.*, role reassignments, at design-time. In [28], the authors apply a similar approach using declarative policies to specify adaptation. In [3], the authors present a policy-based approach where each adaptation policy comprises rules, actions, and the rate at which each rule should be evaluated. These approaches differ from our work because they are based on static reconfiguration, whereas CHARIOT is based on dynamic reconfiguration.

Our prior work based on static reconfiguration [20] shows how system-wide mitigation can be performed based on reactive, timed-state machines specified at design-time, using the results of a two-level fault-diagnoser [9]. Statically specified reconfiguration techniques typically result in faster performance since reconfiguration actions are pre-determined so no additional computations are required at runtime. These techniques are generally untenable for IoT systems since these systems are dynamic and thus all possible runtime scenarios cannot be pre-determined *a priori* at design-time.

3.2.2 *Online Strategies*

The CHARIOT solution described in this paper uses online, dynamically computed strategy for reconfiguration. It requires runtime computation to search for a solution. Reducing this search time and ensuring its predictability is essential for IoT systems that host mission-critical, cyber-physical applications. Our prior work [18] on dynamic reconfiguration was based on boolean encoding of a system. This work has some limitations, however, since it was (1) based on a SAT solver and therefore could not accommodate complex constraints over integer variables, (2) not flexible enough to consider runtime modification of a system's encoding, and (3) unable to take timing requirements into account.

In [31], the authors present middleware that supports timely reconfiguration in distributed real-time and embedded systems based on services. At design-time, the schedulability and complexity of a system is analyzed and fine-tuned to bound sources of unpredictability. The resulting *Scheduled Expanded Graph* is used at runtime to determine the *Execution Graph*, which represents the application in execution. Although this approach is flexible and relies on runtime search of the execution graph for viable reconfiguration solutions, the predictability and schedulability analysis is conducted at design-time, so system resources cannot be modified at runtime. In contrast, CHARIOT supports runtime modification required for systems with dynamic resources.

Dynamic Software Product Lines (DSPLs) have also been suggested for dynamic reconfiguration. In [7], the authors present a survey of the state-of-the-art techniques that attempt to address many challenges of runtime variability mechanisms in the context of DSPLs. The authors also provide a potential solution for runtime checking of feature models for variability management, which motivates the concept of *configuration models*. A configuration model acts as a database that stores a feature model along with all possible valid states of the feature model. Although this work is conceptually similar to our CHARIOT middleware, it does not take timing requirements into account.

Ontology-based reconfiguration work has been presented in [12, 29], where the analytical redundancy of computational components is made explicit. On the basis of this ontology, the system can be reconfigured by identifying suitable substitutes for the failed services. Unlike our CHARIOT middleware, these ontology-based reconfiguration solely rely on redundancy.

4. THE CHARIOT SOLUTION

This section presents detailed description of CHARIOT, our solution to the challenges and requirements previously identified in Section 1. A depiction of CHARIOT's approach to addressing these requirements is presented in Figure 4. As shown in this figure, the design-time aspect of our solution includes a modeling language and associated interpreters. The runtime aspect includes entities that comprise a self-reconfiguration loop, which implements a *sense-plan-act* closed-loop to (1) detect and diagnose failures, (2) compute reconfiguration plan(s), and (3) reconfigure the system. With respect to the different layers of CHARIOT (see Section 1), the design layer is part of the design-time aspect, the management layer is part of the runtime aspect, and the data layer cross cuts across both aspects.

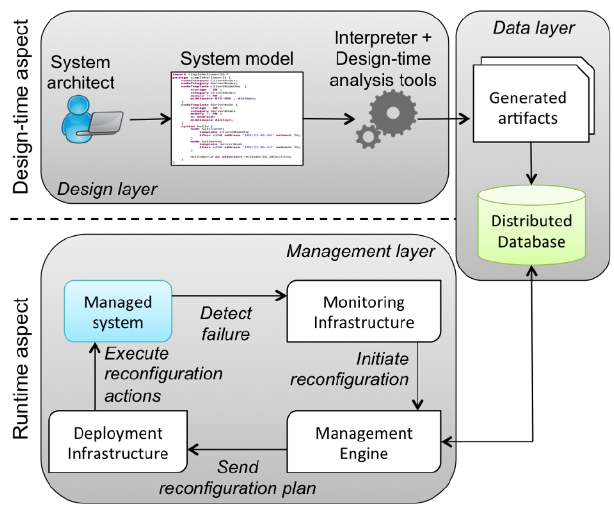


Figure 4: Overview of the CHARIOT Solution.

CHARIOT handles *failure avoidance* via functionality redundancy and optimal distribution of redundant functionalities. It tolerate failures by strategically deploying redundant copies of components that provide critical functionalities, so more failures are avoided/tolerated without having to reconfigure the system. This is described in more detail in Section 4.1.2.

CHARIOT handles failure management via the sense-plan-act loop outlined above. Its *Monitoring Infrastructure* is responsible for detecting failures, which is the *sensing* phase. We use capabilities supported by ZooKeeper [14] to implement a monitoring infrastructure (see Section 5.2). After failure detection and diagnosis, the *Management Engine* determines the actions needed to reconfigure the system so that failures are mitigated, which is the *planning* phase and is based on the Z3 [8] open-source Satisfiability Modulo Theories (SMT) solver (see Section 4.3). Once reconfiguration actions are computed, the *Deployment Infrastructure* uses them to reconfigure the system, which is the *acting* phase.

CHARIOT handles anticipated changes (*i.e.*, planned update or evolution) via operations management. These changes include both hardware changes (*e.g.,* addition of new nodes and removal of existing nodes) and software changes (*e.g.,* addition of new applications, and modification or removal of existing applications) performed at runtime.

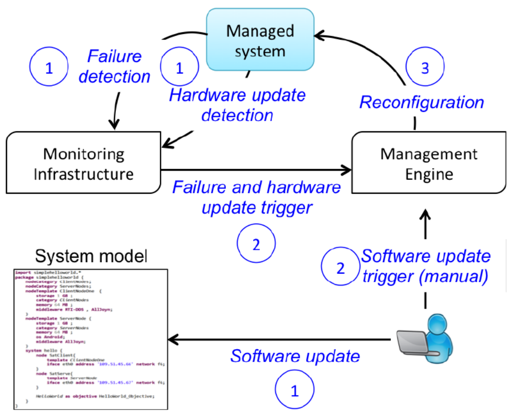


Figure 5: Reconfiguration Triggers Associated with Failure Management and Operations Management.

Figure 5 depicts detection and reconfiguration trigger mechanisms associated with failure management and operations management. As shown in this figure, reconfiguration for failure management and hardware update (operations management) is triggered by the monitoring infrastructure. Conversely, reconfiguration for software update (operations management) is triggered manually after the system model is updated.

4.1 Design Layer

This section describes the CHARIOT design layer, which addresses the requirements of a design-time entity to capture system descriptions. CHARIOT's design layer allows implicit and flexible system description prior to runtime. An IoT system can be described in terms of required components or it can be described in terms of functionalities provided by components. The former approach is inflexible since it tightly couples specific components with the system. CHARIOT therefore supports the latter approach, which is more generic and flexible since it describes the system in terms of required functionalities, where different components can be used to satisfy system requirements, depending on their availability.

A key challenge faced when creating CHARIOT was to devise a design-time environment whose system description mechanism can capture system information (*e.g.,* properties, provisions, requirements, and constraints) *without* explicit management directives (*e.g., if node A fails, move all components to node B*). This mechanism enables CHARIOT to manage failures by efficiently searching for alternative solutions at runtime. Another challenge faced when creating CHARIOT was how to devise abstractions that ensure both correctness *and* flexibility so it can easily support operations management.

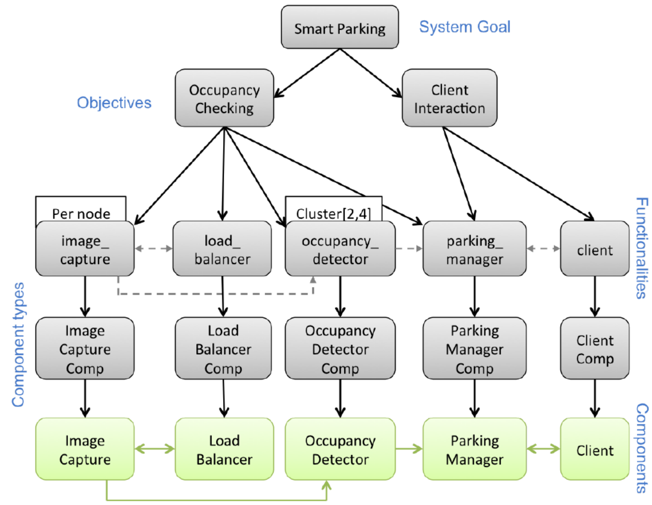


Figure 6: Parking System Description for the Example Shown in Figure 3.

To meet the challenges described above, CHARIOT's design layer allows application developers to model IoT systems using a generic system description mechanism. This mechanism is implemented using a goal-based system description approach. The key entities modeled as part of a system's description are (1) resource categories and templates, (2) different types of components that provide various functionalities, and (3) goal descriptions corresponding to different applications that must be hosted on available resources. CHARIOT defines a *goal* as a collection of *objectives*, where each objective is a collection of *functionalities* that can have inter-dependencies.

CHARIOT's design layer concretizes the functionality tree described in Section 2. It enforces a two-layer functionality hierarchy, where *objectives* are high-level functionalities that satisfy goals and *functionalities* are leaf nodes associated with component types. When these component types are instantiated, each component instance provides associated functionalities. To maximize composability and reusability, a component type can only be associated with a single functionality, though multiple component types can provide the same functionality.

To further explain CHARIOT's design layer the remainder of this section presents the system description of the smart parking system summarized in Section 2.1. Figure 6 shows the corresponding functionality tree, which is used below to describe the different entities comprising the IoT system's description using snippets of models built using CHARIOT-ML, which is our design-time modeling environment. A detailed description of the modeling language appears in [27].

4.1.1 Node Categories and Templates

Since physical nodes are part of an IoT system, CHARIOT-ML models them using categories and templates. The nodes are not explicitly modeled since the group of nodes comprising a system can change dynamically at runtime. CHARIOT thus only models *node categories* and *node templates*. A node category is a logical concept used to establish groups of nodes; every node that is part of a IoT system belongs to a certain node category.

```
1 import edu.vanderbilt.isis.chariot.smartparkingiotpaper.*
2 package edu.vanderbilt.isis.chariot.smartparkingiotpaper {
3     nodeCategory CameraNode {
4         // Template for Wi-Fi enabled (wireless IP)
5         // camera nodes.
6         nodeTemplate wifi_cam {
7             memory 32 MB
8             storage 1024 MB     // 1 GB external
9         }
10    }
11
12    nodeCategory ProcessingNode {
13        // Template for Edison nodes.
14        nodeTemplate edison {
15            memory 1024 MB      // 1 GB
16            storage 4096 MB     // 4 GB
17        }
18    }
19
20    nodeCategory TerminalNode {
21        // Template for entry termial nodes.
22        nodeTemplate entry_terminal {
23            memory 1024 MB      // 1 GB
24            storage 8192 MB     // 8 GB
25        }
26    }
27 }
```

Figure 7: Snippet of Node Categories and Node Templates Declarations.

Since CHARIOT does not explicitly model nodes at design-time, it uses the concept of node template to represent the types of nodes that can belong to a category. A node category[3] is thus a collection of node templates, where a node template is a collection of generic information, such as specifications of memory, storage, devices, available software artifacts, supported operating system, and available communication middleware. A node template can be associated with any node that is an instance of the node template. When a node joins a cluster at runtime the only information it needs to provide (beyond node-specific network information) is which node template it is an instance of.

Figure 7 presents the node categories and templates for the smart parking system.There are three categories of nodes shown in this figure: *CameraNode* (line 3-10), *ProcessingNode* (line 12-18), and *TerminalNode* (line 20-26). Each category contains one template each. The *CameraNode* category contains a *wifi_cam* template that represents a Wi-Fi enabled wireless IP camera. The *ProcessingNode* category contains an *Edison* template that represents an Edison board. The *TerminalNode* category contains an *entry_terminal* template that represents a parking control station placed at an entrance of a parking space. This scenario is consistent with the smart parking system described in Section 2.1. For simplicity, we only model memory and storage specifications for each node template.

4.1.2 Functionalities, Compositions and Goals

Functionalities in CHARIOT-ML are modeled as entities with one or more input and output ports, whereas compositions are modeled as a collection of functionalities and their inter-dependencies. Figure 8 presents four different functionalities (*parking_manager*, *image_capture*, *load_balancer*, and *occupancy_detector*) and the corresponding composition (*occupancy_checking*) that is associated with the *OccupancyChecking* objective (see line 6 in Figure 9). This figure also shows that composition is a collection of functionalities and their inter-dependencies, which are captured as connections between input and output ports of different functionalities.

[3] The concept of node categories becomes important when assigning a per-node replication constraint (discussed in Section 4.1.2), which requires that a functionality be deployed on each node of the given category.

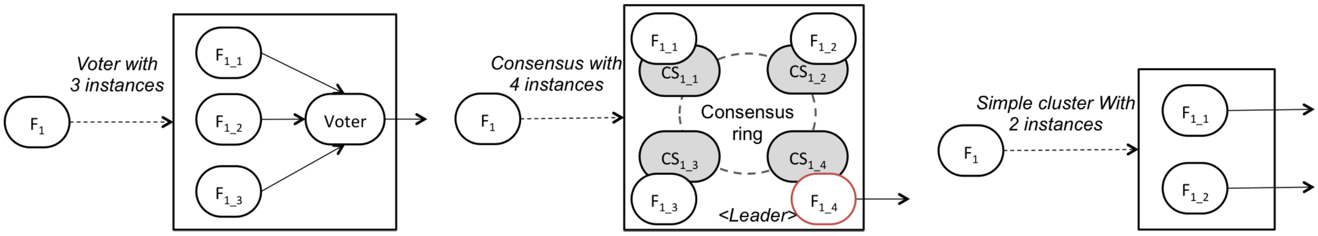
```
package edu.vanderbilt.isis.chariot.smartparkingiotpaper {
    // Parking manager functionality with an input and an
    // output port to interact with client functionality.
    // Also, another input port to interact with occupancy
    // detector.
    functionality parking_manager {
        input parking_request, occupancy_status
        output parking_response
    }
    // Image capture functionality with an input and an output
    // port to interact with load balancer functionality. Also,
    // another output port to interact with occupancy detector.
    functionality image_capture {
        input detector_response
        output detector_request, image
    }
    // Load balancer functionality with an input and an output
    // port to interact with image capture functionality.
    functionality load_balancer {
        input detector_request
        output detector_response
    }
    // Occupancy detector with an input port to interact with
    // image capture functionality and an output port to
    // interact with parking manager functionality.
    functionality occupancy_detector {
        input image
        output occupancy_status
    }
    // A composition that captures interaction between image
    // capture, load balancer, occupancy detector, and parking
    // manager functionalities.
    composition occupancy_checking {
        image_capture.detector_request to
            load_balancer.detector_request
        load_balancer.detector_response to
            image_capture.detector_response

        image_capture.image to occupancy_detector.image
        occupancy_detector.occupancy_status to
            parking_manager.occupancy_status
    }
}
```

Figure 8: Snippet of Functionalities and Corresponding Composition Declaration.

```
1 import edu.vanderbilt.isis.chariot.smartparkingiotpaper.*
2 package edu.vanderbilt.isis.chariot.smartparkingiotpaper {
3     goalDescription SmartParking {
4         // Objectives.
5         client_interaction as objective ClientInteraction
6         occupancy_checking as objective OccupancyChecking
7
8         // Replication constraints.
9         replicate image_capture asPerNode
10            for category CameraNode
11        replicate parking_client asPerNode
12            for category TerminalNode
13        replicate occupancy_detector asCluster
14            with [2,4] instances
15    }
16 }
```

Figure 9: Snippet of Smart Parking Goal Description Comprising Objectives and Replication Constraints.

The goal description for the smart parking application is shown in Figure 9.The goal itself is declared as *SmartParking* (line 3). Following the goal declaration is a list of the objectives required to satisfy the goal (line 5-6). Two objectives are defined in this example: the *ClientInteraction* objective and the *OccupancyChecking* objective. The *ClientInteraction* objective is related to the task of handling client parking requests, whereas the *OccupancyChecking* objective is related to the task of determining the occupancy status of different parking spaces.



(a) Voter pattern with factor = 3. (b) Consensus pattern with factor = 4. (c) Cluster pattern with factor = 2.

Figure 10: Example Redundancy Patterns for Functionality F_1. The CS_{n_m} Entities Represent Consensus Service Providers.

In CHARIOT-ML, objectives are instantiations of compositions. The *ClientInteraction* objective is an instantiation of the *client_ineraction* composition (line 5) and the *OccupancyChecking* objective is an instantiation of the *occupancy_checking* composition (line 6).

Support for Redundant Deployment Patterns: CHARIOT-ML also supports redundant deployment patterns as a result of which functionalities can be associated with replication constraints. For example, Figure 9 shows the association of the *image_capture* functionality with a per-node replication constraint (line 9-10), which means this functionality should be present on each node that is an instantiation of any node template belonging to *CameraNode* category. Similarly, the *parking_client* functionality is also associated with a per-node replication constraint (line 11-12) for *TerminalNode* category. Finally, the *occupancy_detector* functionality is associated with a cluster replication constraint (line 13-14), which means this functionality should be deployed as a cluster of at-least 2 and at-most 4 instances.

CHARIOT-ML supports functionality replication using four different redundancy patterns: the (1) voter pattern, (2) consensus pattern, (3) cluster pattern, and (4) per-node pattern, as shown in Figure 10. The per-node pattern (as described above for the *image_capture* functionality) requires that the associated functionality be replicated on a per-node basis. Replication of functionalities associated with the other three redundancy patterns is based on their redundancy factor, which can be expressed by either (1) explicitly stating the number of redundant functionalities required or (2) providing a range. The latter (as previously described for the *occupancy_detector* functionality) requires the associated functionality to have a minimum number for redundancy and a maximum number for redundancy, *i.e.*, if the number of functionalities present at any given time is within the range, the system is still valid and no reconfiguration is required.

Figure 10 presents a graphical representation of voter, consensus, and cluster redundancy patterns (the case of the consensus pattern, CS represents consensus services). Different redundancy factors are used for each. As shown in the figure, the voter pattern involves a voter in addition to the functionality replicas, the consensus pattern involves a consensus service each for the functionality replicas and these consensus services implement a consensus ring, and the cluster pattern only involves the functionality replicas. Implementing the consensus service is beyond the scope of this paper. In practice, CHARIOT uses existing consensus protocols, such as Raft [25], for this purpose.

4.1.3 Component Types

CHARIOT-ML does not model component instances, but instead models component types. Each component type is associated with a functionality. When a component type is instantiated, the component instance provides the functionality associated with its type. A component instance therefore only provides a single functionality, whereas a functionality can be provided by component instances of different types. Two advantages of modeling component types instead of component instances include the flexibility it provides with respect to (1) the number of possible runtime instances of a component type and (2) the number of possible component types that can provide the same functionality.

```
1 import edu.vanderbilt.isis.chariot.smartparkingiotpaper.*
2 package edu.vanderbilt.isis.chariot.smartparkingiotpaper {
3     component ParkingManager {
4         provides parking_manager  // Provided functionality.
5
6         requires 128 MB memory // Minimum memory required.
7
8         startScript "sh ParkingManager.sh" // Launch script.
9     }
10 }
```

Figure 11: Snippet of Component Type Declaration.

Figure 11 shows how the *ParkingManager* component type is modeled in CHARIOT-ML. As part of the component type declaration, we first model the functionality that is provided by the component (line 4). After the functionality of a component type is modeled, we model various resource requirements (Figure 11 only shows memory requirements in line 6) and the launch script (line 8), which can be used to instantiate an instance of the component by spawning an application process.

CHARIOT supports two different types of component types: hardware components and software components. The component type presented in Figure 11 is an example of a software component. Hardware components are modeled in a similar fashion, though we just model the functionality provided by a hardware component and nothing else since a hardware component is a specific type of component whose lifecycle is tightly coupled to the node with which it is associated. A hardware component is therefore never actively managed (reconfigured) by the CHARIOT orchestration middleware. The only thing that affects the state of a hardware node is the state of its hosting node, *i.e.*, if the node is on and functioning well, the component is active and if it is not, then the component is inactive.

In context of the smart parking system case study, the *ImageCapture* component is a hardware component that is associated with camera nodes. As a result, an instance of the *ImageCapture* component runs on each active camera node. We model this requirement using the per-node redundancy pattern (see line 32-33 in Figure 9). Likewise, the failure of a camera node implies failure of the hosted *ImageCapture* component instance, so this failure cannot be mitigated.

4.1.4 Summary of the Design Layer

CHARIOT-ML is a Domain Specific Modeling Language (DSML) built using the Xtext framework [2] that comprises CHARIOT's design layer. This DSML is a textual modeling language designed using the Xtext framework [2]. Currently, CHARIOT-ML allows modeling of resources such as software artifacts, devices, memory, storage, operating system, and communication middleware. Although this is an extensive list of resource types for most IoT systems, it might not be sufficient for all possible IoT systems. Therefore, it might require modifications and extensions depending on the domain in which it is being used. For example, in order to model self-degrading systems that rely on monitoring of QoS parameters, CHARIOT-ML must facilitate modeling of QoS thresholds at different levels of abstractions.

Furthermore, the language currently only facilitates replication constraints, which are a type of deployment constraint that specifies the number of certain functionality that must be deployed. There are other scenarios, however, where replication constraints are not sufficient and more specific deployment constraints are required, such as deploy-on-same-node, deploy-on-different-node, and deploy-only-on-a-specific-resource-category.

Due to the modular nature of the xtext framework, introducing these changes will not be difficult. However, we must ensure that the new concepts do not violate any existing rules already implemented. Furthermore, the data schema defined in Section 4.2 ensures that the extensions introduced at design layer can be supported by the underlying management layer, assuming that the functionalities are only being added in and not modifying existing concepts.

4.2 Data Description Layer

This section presents the CHARIOT data layer, which defines a schema that forms the basis for persistently storing system information, such as design-time system description and runtime system information. This layer codifies the format in which system information should be represented. A key advantage of this codification is its decoupling of CHARIOT's design layer (top layer) from its management layer (bottom layer), which yields a flexible architecture that can accommodate varying implementations of the design layer, as long as those implementations adhere to the data layer schema described in this section.

Figure 12 presents UML class diagrams as schemas used to store design-time system description and runtime system information. These schemas are designed for document-oriented databases. An instance of a class that is not a child in a composition relationship therefore represents a root document. Below we describe CHARIOT's design-time and runtime schemas in detail.

4.2.1 Design-time System Description Schema

The schema for design-time system description comprises entities to store node categories, component types, and goal descriptions, as shown in Figure 12a. These concepts have been previously described in Section 4.1. Neither node categories nor component types are application-specific since multiple applications can be simultaneously hosted on nodes of an IoT system and a component type can be used by multiple applications. In addition to other attributes, the *ComponentType* class also captures scripts that can be used to start and stop an instance of a component type; this information is used at runtime to instantiate components.

As shown in Figure 12a, a goal description comprises objectives, which are composed of functionalities, and replication constraints. The *ReplicationConstraint* class represents replication constraints and consists of *maxInstances*, *minInstance*, and *numInstances* attributes that are related to the degree of replication. The latter attribute is used if a specific number of replicas are required, whereas the former two attributes are used to describe a range-based replication. The *nodeCategories* attribute is used for per-node replication constraints. The *serviceComponentType* attribute is related to specific component types that provide special replication services, such as a component type that provides a voter service or a consensus service.

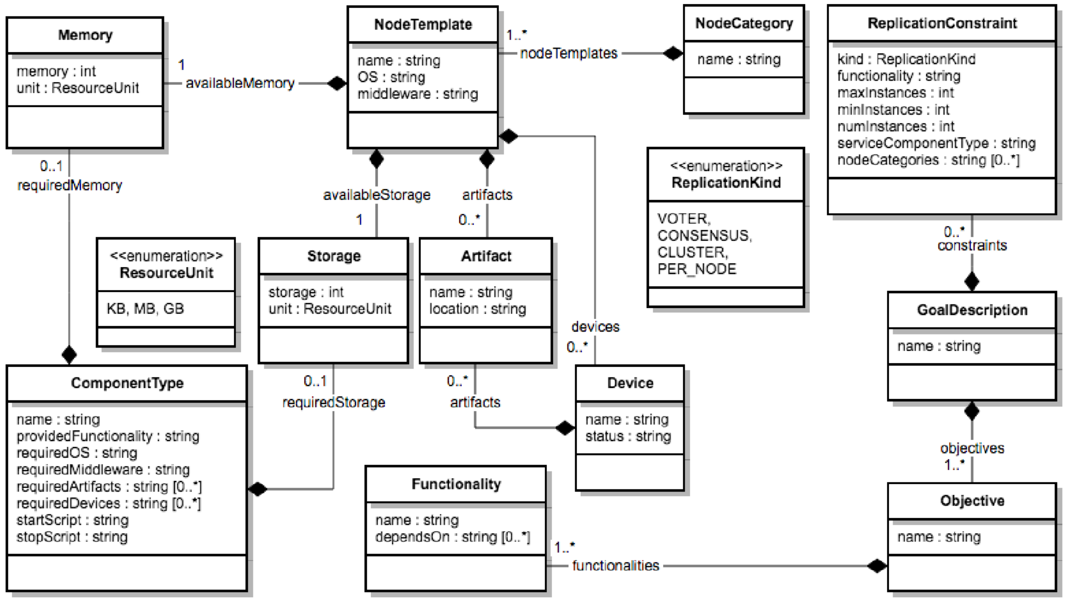
4.2.2 Runtime Information Schema

The schema for runtime system information comprises entities to store functionality instances, nodes, deployment actions, reconfiguration events, and look-ahead information, as shown in Figure 12b. Since functionalities can be replicated, the *FunctionalityInstance* class is used to store information about functionality instances. The *ComponentType* attribute is only relevant for voter and consensus service providing functionality instances as they are not associated with functionalities that are part of a goal description. Furthermore, the *alwaysDeployOnNode* attribute ties a functionality instance to a specific node and is only relevant for functionality instances related to per-node replication groups. Finally, the *mustDeploy* boolean attribute indicates whether a functionality instance should always be deployed.

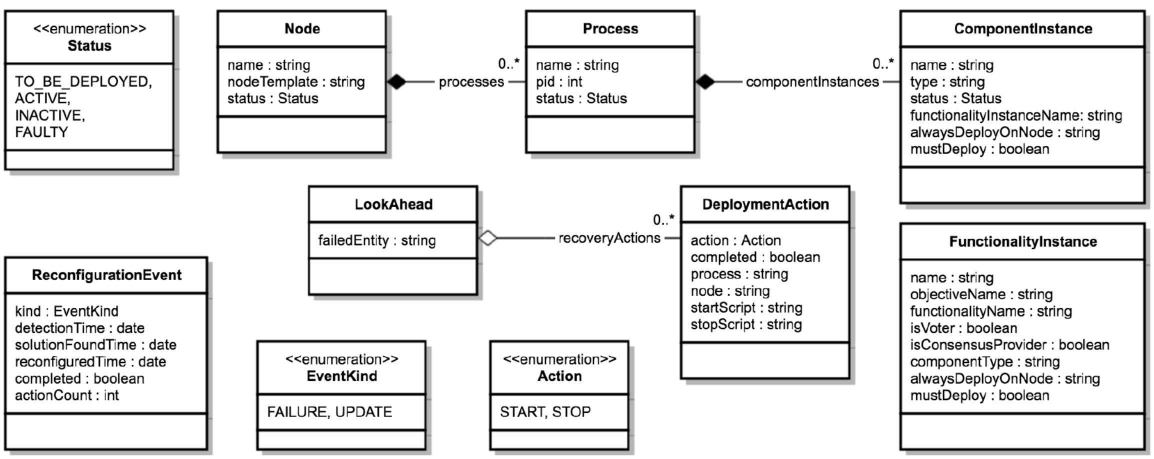
The *Node* class represents compute nodes, the *Process* class represents processes running on nodes, and the *ComponentInstance* class represents component instances hosted on processes. As shown in Figure 12b, these three classes have containment relationship. The *functionalityInstanceName* attribute in *ComponentInstance* class represents the name of the corresponding functionality instance as a component instance is always associated with a functionality instance (see Section 4.3.3).

The *DeploymentAction* class represents runtime deployment actions that are computed by the CHARIOT management engine to (re)configure a system. The *DeploymentAction* class consists of an action, a *completed* boolean flag to indicate if an action has been taken, process affected by the action, node on which the action should be performed, and scripts to perform the action. CHARIOT supports two kinds of actions: start actions and stop actions. The *LookAhead* class represents precomputed solutions (see Section 4.3.6). It consists of attributes that represent a failed entity, and a set of recovery actions (deployment actions) that must be performed to recover from the failure.

The *ReconfigurationEvent* class represents runtime reconfiguration events. It is used to keep track of failure and update events that trigger system reconfiguration. It consists of *detectionTime*, *solutionFoundTime*, and *reconfiguredTime* to keep track of when a failure or update was detected, when



(a) Schema to Store Design-time System Descriptions.



(b) Schema to Store Runtime System Representations.

Figure 12: UML Class Diagrams for Schemas Used to Store System Information.

a solution was computed, and when the computed solution was deployed. It also consists of a *completed* attribute to indicate whether a reconfiguration event is complete or not and an *actionCount* attribute to keep track of number of actions required to complete a reconfiguration event.

4.2.3 Summary of the Data Description Layer

Although not a novel contribution by itself, the data layer described in this section is critical to the overall CHARIOT ecosystem. This layer addresses the challenge of providing a generic and uniform system state that can be queried by the rest of the system at runtime. Having a well-defined model for system information not only helps the CHARIOT ecosystem remain flexible by decoupling the design and management layers, it also aids in future extensions when required.

For example, as previously discussed in Section 4.1.4, let us assume that a set of deployment constraints needs to be added. This requires us to extend the current design-time system description (shown in Figure 12a) in such a way that the deployment constraints modeled at design-time can be easily stored in the data layer and retrieved by the management layer without affecting any existing code. This could be achieved by adding a *DeploymentConstraint* class similar to the *ReplicationConstraint* class and have it be part of the *GoalDescription* class.

4.3 Runtime Management Layer

The CHARIOT runtime management layer comprises a monitoring and deployment infrastructure, as well as a management engine, as previously outlined in Figure 4. The monitoring and deployment of distributed applications is covered in prior work [26]; CHARIOT implements these capabilities using existing technologies described in Section 5. This section focuses on CHARIOT's management engine that facilitates self-reconfiguration of IoT systems by (1) adding the capability to compute exact component instances from available component types, (2) encoding redundancy patterns using SMT constraints, and (3) using a finite horizon look-ahead strategy that pre-computes solutions to significantly improve the performance of CHARIOT's management engine.

4.3.1 *Configuration Space and Points*

The general idea behind CHARIOT's self-reconfiguration approach relies on the concepts of *configuration space* and *configuration points*. If a system's state is represented by a configuration point in a configuration space, then reconfiguration of that system entails moving from one configuration point to another in the same configuration space. A configuration space includes (1) goal descriptions of different applications, (2) replication constraints corresponding to redundancy patterns associated with different applications, (3) component types that can be used to instantiate different component instances and therefore applications, and (4) available resources, which includes different nodes and their corresponding resources, such as memory, storage, and computing elements.

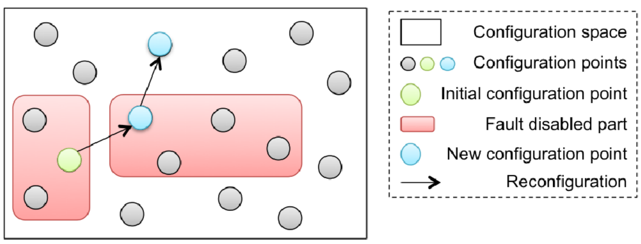


Figure 13: A Configuration Space with different Configuration Points. This figure depicts two faults that disable parts of the system resulting in two reconfigurations.

At any given time a configuration space of an IoT system can represent multiple applications associated with the system. A configuration space can therefore contain multiple configuration points, as shown in Figure 13. These configuration points represent valid configurations of all applications that are part of the IoT system represented by the configuration space.

A valid configuration point represents component-instance-to-node mappings (*i.e.*, a deployment) for all component instances needed to realize different functionalities essential for the objectives required to satisfy goals of one or more applications. The initial configuration point represents the initial (baseline) deployment, whereas, current configuration point represents the current deployment.

4.3.2 *Computing the Configuration Point*

Given above definition of configuration space and points, a valid reconfiguration mechanism entails moving from one configuration point to another in the same configuration space (see Figure 13). When a failure occurs, the current configuration point is rendered faulty. Moreover, parts of configuration space may also be rendered faulty, depending on the failure. For example, consider a scenario where multiple configuration points map one or more components to a node. If this node fails then all aforementioned configuration points are rendered faulty. In addition to failure, hardware and software updates can also result in reconfiguration, as discussed earlier.

Specifically, reconfiguration in CHARIOT happens by identifying a new valid configuration point and determining the set of actions required to transition from current (faulty) configuration point to the new (desired) configuration point. Configuration points and their transitions thus form the core of CHARIOT's reconfiguration mechanism. For any reconfiguration several valid configuration points might be available. From the available configuration points, an optimal configuration point that satisfies the system requirements can be obtained based on several criteria, such as transition cost, reliability, operation cost, and/or utility. The *Configuration Point Computation* (CPC) algorithm serves this purpose and thus defines the core of CHARIOT's self-reconfiguration mechanism. The CPC algorithm can be decomposed into three phases: the (1) instance computation phase, (2) constraint encoding phase, and (3) solution computation phase, as described next.

4.3.3 *Phase 1: Instance Computation*

The first phase of a CPC computes required instances of different functionalities and subsequently components, based on the system description provided at design-time. Each functionality can have multiple instances if it is associated with a replication constraint. Each functionality instance should have a corresponding component instance that provides the functionality associated with the functionality instance. Depending upon the number of component types that provide a given functionality, a functionality instance can have multiple component instances. Only one of the component instances will be deployed at runtime, however, so there is always a one-to-one mapping between a functionality instance and a deployed component instance.

The CPC algorithm first computes different functionality instances using Algorithm 1, which is invoked for each objective. Every functionality is initially checked for replication constraints (line 3). If a functionality does not have a replication constraint, a single functionality instance is created (line 32). For every functionality that has one or more replication constraints associated with it, each constraint is handled depending on the type of the constraint. A per-node replication constraint is handled by generating a functionality instance and an *assign* constraint each for applicable nodes (line 6-11). An application node is a node that is alive and belongs to the node category associated with the per-node replication constraint.

Unlike a per-node replication constraint, the voter, consensus, and cluster replication constraints depend on an exact replication value or a replication range to determine the number of replicas (line 13-19). In the case of a range-based replication, CHARIOT tries to maximize the number of replicas by using maximum of the range, which ensures that maximum number of failures is tolerated without having to reconfigure the system. After the number of replicas is determined, CHARIOT computes the replica functionality instances (line 21), as well as special functionality instances that support different types of replication constraint.

For example, for each replica functionality instance in a consensus replication constraint, CHARIOT generates a consensus service functionality instance (line 23) (a consensus service functionality is provided by a component that implements consensus logic using existing algorithms, such as Paxos [16] and Raft [25]). For a voter replication constraint, in contrast, CHARIOT generates a single voter functional-

Algorithm 1 Functionality Instances Computation.

Input: objective (*obj*), nodes (*nodes_list*), computed functionalities (*computed_functionalities*)
Output: functionality instances for *obj* (*ret_list*)

```
 1: for func in obj.functionalities do
 2:     if func not in computed_functionalities then            ▷ Make sure a functionality is processed only once.
 3:         if func has associated replication constraints then
 4:             constraints = all replication constraints associated with func
 5:             for c in constraints do
 6:                 if c.kind == PER_NODE then                                   ▷ Handle per node replication.
 7:                     for node_category in c.nodeCategories do
 8:                         nodes = nodes in nodes_list that are alive and belong to category node_category
 9:                         for n in nodes do
10:                             create functionality instance and add it to ret_list
11:                             add assign (functionality instance, n) constraint
12:                 else
13:                     replica_num = 0      ▷ Initial number of replicas, which will be set to max value if range given.
14:                     range_based = False          ▷ Flag to indicate if a replication constraints is range based.
15:                     if c.numInstances ≠ 0 then
16:                         replica_num = c.numInstances
17:                     else
18:                         range_based = True
19:                         replica_num = c.maxInstances
20:                     for i = 0 to replica_num do                    ▷ Create replica functionality instances.
21:                         create replica functionality instance and add it to ret_list
22:                         if c.kind == CONSENSUS then                        ▷ Handle consensus replication.
23:                             create consensus service functionality instance and add it to ret_list
24:                             add implies (replica functionality instance, consensus service functionality instance) constraint
25:                             add collocate (replica functionality instance, consensus service functionality instance) constraint
26:                     if c.kind == VOTER then                                    ▷ Handle voter replication.
27:                         create voter functionality instance and add it to ret_list
28:                     if range_based == True then        ▷ If replication range is given, add atleast constraints.
29:                         add atleast (c.rangeMinValue, replica functionality instances) constraint
30:                     add distribute (replica functionality instances) constraint
31:         else
32:             create functionality instance and add it to ret_list
33:         add func to computed_functionalities
```

ity instance for the entire replication group (line 27). In the case of a cluster replication constraint, no special functionality instance is generated as a cluster replication comprises independent functionality instances that do not require any synchronization (see Section 4.1.2).

In order to ensure proper management of instances related to functionalities with voter, consensus, or cluster replication constraints, CHARIOT uses four different constraints: (1) implies, (2) collocate, (3) atleast, and (4) distribute. The *implies* constraint ensures all replica functionality instances associated with a consensus pattern require their corresponding consensus service functionality instances (line 24). Similarly, the *collocate* constraint ensures each replica functionality instance and its corresponding consensus service functionality instance are always collocated on the same node (line 25). The *atleast* constraint ensures the minimum number of replicas are always present in scenarios where a replication range is provided (line 28-29). Finally, the *distribute* constraint ensures that the replica functionalities are distributed across different nodes (line 30). CHARIOT's ability to support multiple instances of functionalities and distribute them across different nodes is the basis of the failure avoidance mechanism.

After functionality instances are created, CHARIOT next creates the component instances corresponding to each functionality instance. In general, it identifies a component type that provides the functionality associated with each functionality instance and instantiates that component type. As explained in Section 4.1.3, component types are modeled as part of the system description. Different component types can provide the same functionality, in which case multiple component types are instantiated, but a constraint is added to ensure only one of those instances is deployed and running at any given time. In addition, all constraints previously created in terms of functionality instances are ultimately applied in terms of corresponding component instances. We describe the constraints next.

4.3.4 Phase 2: Constraint Encoding

The second phase of the CPC algorithm is responsible for constraint encoding and optimization. These constraints are summarized below:

1. Since reconfiguration involves transitioning from one configuration point to another, constraints that represent a configuration point are of utmost importance.

2. Constraints to ensure component instances that must be deployed are always deployed.
3. Constraints to ensure component instances that communicate with each other are either deployed on the same node or on nodes that have network links between them.
4. Constraints to ensure resources' provided-required relationships are valid.
5. Constraints encoded in the first phase of the CPC algorithm for proper management of component instances associated with replication constraints.
6. Constraints to represent failures, such as node failure or device failures.

The remainder of this section describes how CHARIOT implements the constraints listed above as SMT constraints.

Representing the configuration points: A configuration point in CHARIOT is therefore presented using a component-instance-to-node (C2N) matrix, as shown below. A C2N 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 (Equation 1). Each element of the matrix is encoded as a Z3 integer variable whose value can either be 0 or 1 (Equation 2). 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. For a valid C2N matrix, a component instance must not be deployed more than once, as shown in Equation 3.

$$C2N = \begin{bmatrix} c2n_{00} & c2n_{01} & c2n_{02} & \dots & c2n_{0\beta} \\ c2n_{10} & c2n_{11} & c2n_{12} & \dots & c2n_{1\beta} \\ c2n_{20} & c2n_{21} & c2n_{22} & \dots & c2n_{2\beta} \\ \dots & \dots & \dots & \dots & \dots \\ c2n_{\alpha 0} & c2n_{\alpha 1} & c2n_{\alpha 2} & \dots & c2n_{\alpha\beta} \end{bmatrix}$$

$$c2n_{cn} : c \in \{0 \dots \alpha\}, n \in \{0 \dots \beta\}, (\alpha, \beta) \in \mathbb{Z}^+ \tag{1}$$

$$\forall c2n_{cn} \in C2N : c2n_{cn} \in \{0, 1\} \tag{2}$$

$$\forall c : \sum_{n=0}^{\beta} c2n_{cn} \leq 1 \tag{3}$$

Now that we have constraints defined to represent a configuration point (*i.e.*, a valid component-instance-to-node mapping), a constraint is needed to ensure component instances that should be deployed *are* always deployed. At this point it is important to recall range-based replication described in Section 4.1.2. This approach results in a set of instances where a certain number (at least the minimum) should always be deployed, but the remaining (difference between maximum and minimum) are not always required, even though all of them are deployed initially. At any given time, therefore, a configuration point can comprise of some component instances that must be deployed and others that are not always required be deployed. In CHARIOT we encode the "must deploy assignment" constraint as follows:

Capturing the Must Deploy Constraint: The "must deploy assignment" constraint is used to ensure all component instances that should be deployed are in fact deployed. This constraint therefore uses the C2N matrix (Equation 1) and a set of component instances that must be deployed, as shown in Equation 4.

Let M be a set of all component instances that must be deployed.

$$\forall m \in M : \sum_{n=0}^{\beta} c2n_{mn} == 1 \tag{4}$$

The third set of constraints ensure that component instances with inter-dependencies (*i.e.*, that communicate with each other) are either deployed on the same node or on nodes that have network links between them. CHARIOT encodes this constraint as follows:

Capturing the dependencies between components: This constraint ensures that interacting component instances are always deployed on resources with appropriate network links to support communication. This constraint is encoded in terms of a node-to-node (N2N) matrix, which is a square matrix that represents existence of network links between nodes. This N2N matrix thus comprises rows and columns that represents different nodes (Equation 5). Each element of the N2N matrix is either 0 or 1, where 0 means there does not exist a link between the two corresponding nodes and 1 means there exists a link between the two corresponding nodes. The constraint is presented in Equation 6.

$$N2N = \begin{bmatrix} n2n_{00} & n2n_{01} & n2n_{02} & \dots & n2n_{0\beta} \\ n2n_{10} & n2n_{11} & n2n_{12} & \dots & n2n_{1\beta} \\ \dots & \dots & \dots & \dots & \dots \\ n2n_{\beta 0} & n2n_{\beta 1} & n2n_{\beta 2} & \dots & n2n_{\beta\beta} \end{bmatrix}$$

$$n2n_{n_1 n_2} : (n_1, n_2) \in \{0 \dots \beta\}, \beta \in \mathbb{Z}^+ \tag{5}$$

Let c_s and c_d be two component instances that are dependent on each other.

$$\forall n_1, \forall n_2 : ((c2n_{c_s n_1} \times c2n_{c_d n_2} \neq 0) \wedge (n_1 \neq n_2)) \implies (n2n_{n_1 n_2} == 1) \tag{6}$$

Capturing the Resource Constraints: The fourth set of constraints ensure the validity of resources' provided-required relationships, such that essential component instances of one or more applications can be provisioned. In CHARIOT these constraints are encoded in terms of resources provided by nodes and required by component instances. Moreover, resources are classified into two categories: (1) cumulative resources and (2) comparative resources. Cumulative resources have a numerical value that increases or decreases depending on whether a resource is used or freed. Examples of cumulative resources include primary memory and secondary storage. Comparative resources have a boolean value, *i.e.*, they are either available or not available and their value does not change depending on whether a resource is used or freed. Examples of comparative resources include devices and software artifacts. These two constraints can be encoded as follows:

The "cumulative resource" constraint is encoded using a provided resource-to-node (CuR2N) matrix and a required resource-to-component-instance (CuR2C) matrix. The matrix CuR2N comprises rows that represent different cumulative resources and columns that represent nodes; the size of this matrix is $\gamma \times \beta$, where γ is the number of cumulative resources and β is the number of available nodes (Equation 7). The CuR2C matrix comprises rows that represent different cumulative resources and columns that represent component instances; the size of this matrix is $\gamma \times \alpha$, where γ is the number of cumulative resources and α is number of component instances (Equation 8). Each element of these matrices are integers. The constraint itself ensures that for each available cumulative resource and node, the sum of the amount of the resource required by the component instances deployed on the node is less than or equal to the amount of the resource available on the node, as shown in Equation 9.

$$CuR2N = \begin{bmatrix} r2n_{00} & r2n_{01} & r2n_{02} & \dots & r2n_{0\beta} \\ r2n_{10} & r2n_{11} & r2n_{12} & \dots & r2n_{1\beta} \\ \dots & \dots & \dots & \dots & \dots \\ r2n_{\gamma 0} & r2n_{\gamma 1} & r2n_{\gamma 2} & \dots & r2n_{\gamma\beta} \end{bmatrix}$$

$$r2n_{rn} : r \in \{0 \dots \gamma\}, n \in \{0 \dots \beta\}, (\gamma, \beta) \in \mathbb{Z}^+ \tag{7}$$

$$CuR2C = \begin{bmatrix} r2c_{00} & r2c_{01} & r2c_{02} & \dots & r2c_{0\alpha} \\ r2c_{10} & r2c_{11} & r2c_{12} & \dots & r2c_{1\alpha} \\ \dots & \dots & \dots & \dots & \dots \\ r2c_{\gamma 0} & r2c_{\gamma 1} & r2c_{\gamma 2} & \dots & r2c_{\gamma\alpha} \end{bmatrix}$$

$$r2c_{rc} : r \in \{0 \dots \gamma\}, c \in \{0 \dots \alpha\}, (\gamma, \alpha) \in \mathbb{Z}^+ \tag{8}$$

$$\forall r, \forall n : \left(\sum_{c=0}^{\alpha} c2n_{cn} \times r2c_{rc} \right) \leq r2n_{rn} \tag{9}$$

The "comparative resource" constraint is encoded using a provided resource-to-node (CoR2N) matrix and a required resource-to-component-instance (CoR2C) matrix. The matrix CoR2N comprises rows that represent different comparative resources and columns that represents nodes; the size of this matrix is $\phi \times \beta$, where ϕ is the number of comparative resources and β is the number of available nodes (Equation 10). Similarly, the CoR2C matrix comprises rows that represent different comparative resources and columns that represent component instances; the size of this matrix is $\phi \times \alpha$, where ϕ is the number of comparative resources and α is number of component instances (Equation 11). Each element of these matrices is either 0 or 1, where 0 means the corresponding resource is not provided by the corresponding node (for CoR2N matrix) or not required by the corresponding component instance (for CoR2C matrix) and 1 means the opposite. The constraint itself (Equation 12) ensures that for each available comparative resource, node, and component instance, if the component instance is deployed on the node and requires the resource, then the resource must also be provided by the node.

$$CoR2N = \begin{bmatrix} r2n_{00} & r2n_{01} & r2n_{02} & \dots & r2n_{0\beta} \\ r2n_{10} & r2n_{11} & r2n_{12} & \dots & r2n_{1\beta} \\ \dots & \dots & \dots & \dots & \dots \\ r2n_{\phi 0} & r2n_{\phi 1} & r2n_{\phi 2} & \dots & r2n_{\phi\beta} \end{bmatrix}$$

$$r2n_{rn} : r \in \{0 \dots \phi\}, n \in \{0 \dots \beta\}, (\phi, \beta) \in \mathbb{Z}^+ \tag{10}$$

$$CoR2C = \begin{bmatrix} r2c_{00} & r2c_{01} & r2c_{02} & \dots & r2c_{0\alpha} \\ r2c_{10} & r2c_{11} & r2c_{12} & \dots & r2c_{1\alpha} \\ \dots & \dots & \dots & \dots & \dots \\ r2c_{\phi 0} & r2c_{\phi 1} & r2c_{\phi 2} & \dots & r2c_{\phi\alpha} \end{bmatrix}$$

$$r2c_{rc} : r \in \{0 \dots \phi\}, c \in \{0 \dots \alpha\}, (\phi, \alpha) \in \mathbb{Z}^+ \tag{11}$$

$$\forall r, \forall n, \forall c : Assigned(c, n) \implies (r2n_{rn} == r2c_{rc}) \tag{12}$$

Assigned (c, n) function returns true if component c is deployed on node n, *i.e.*, it returns true if $c2n_{cn} == 1$.

Handling the replication constraints: The fifth set of constraints ensures management of component instances associated with replication constraints. As mentioned in Section 4.3.3, *assign*, *implies*, *collocate*, *atleast*, and *distribute* are the five different kinds of constraints that must be encoded. Each of these constraints is encoded as follows:

The "assign constraint" is used for component instances corresponding to functionalities associated with per-node replication constraint. It ensures that a component instance is only ever deployed on a given node. In CHARIOT, an assign constraint is encoded, as shown in Equation 13.

Let c be a component instance that should be assigned to a node n.

$$Enabled(c) \implies (c2n_{cn} == 1) \tag{13}$$

Enabled(c) function returns true if component instance c is assigned to any node, *i.e*, it checks if $\sum_{n=0}^{\beta} c2n_{cn} == 1$.

The "implies" constraint is used to ensure that if a component depends upon other components then its dependencies are satisfied. It is encoded using the implies construct provided by an SMT solver like Z3.

A "collocate" constraint is used to ensure that two collocated component instances are always deployed on the same node. In CHARIOT this constraint is encoded by ensuring the assignment of the two component instances is same for all nodes, as shown in Equation 14.

Let c_1 and c_2 be two component instances that need to be collocated.

$$\begin{aligned} (Enabled(c_1) \land Enabled(c_2)) \implies \\ (\forall n : c2n_{c_1 n} == c2n_{c_2 n}) \end{aligned} \tag{14}$$

An "atleast" constraint is used to encode a M out of N semantics to ensure that given a set of components (*i.e.* N), a specified number of those components (*i.e.* M) is always deployed. CHARIOT only uses this constraint for range-based replication constraints and its implementation is two fold. First, during the initial deployment CHARIOT tries to maximize M and deploy as many component instances as possible. Current implementation of CHARIOT uses the maximum value associated with a range and initially deploys N component instances, as shown in Equation 15. This of course assumes availability of enough resources. A better solution to this would be to use the maximize optimization, as shown in Equation16. However, in Z3 solver, which is the SMT solver used by CHARIOT, this optimization is experimental and does not scale well. Second, for subsequent non-initial deployment CHARIOT relies on the fact that maximum possible deployment was achieved during initial deployment, so it ensures the minimum number required is always met, as shown in Equation 17.

Let $S = \{c_1, c_2 \ldots c_{\alpha'}\}$ be a set of replica component instances associated with an atleast constraint; N is the size of this set. Also, let *min_value* be the minimum number of component instances required; this is synonymous to M.

$$\sum_{c \in S} \sum_{n=0}^{\beta} c2n_{cn} == max\_value \tag{15}$$

$$maximize(\sum_{c \in S} \sum_{n=0}^{\beta} c2n_{cn}) \tag{16}$$

$$\sum_{c \in S} \sum_{n=0}^{\beta} c2n_{cn} \geq min\_value \tag{17}$$

A "distribute" constraint is used to ensure that a set of components is deployed on different nodes. In CHARIOT this constraint is encoded by ensuring at most only one component instance out of the set is deployed on a single node, as shown in Equation 18.

Let $S = \{c_1, c_2 \ldots c_{\alpha'}\}$ be a set of components that needs to be distributed.

$$\forall n : \sum_{c \in S} c2n_{cn} \leq 1 \tag{18}$$

Capturing failures as constraints: The final step (step 8) of the second phase of the CPC algorithm encodes and adds failure constraints. Depending on the type of failure, there can be different types of failure constraints. This sixth set of constraints handles failure representation, which are encoded in CHARIOT as shown below:

A "node failure" constraint is used to ensure that no components are deployed on a failed node. CHARIOT encodes this constraint as shown in Equation 19.

Let n_f be a failed node.

$$\sum_{c=0}^{\alpha} c2n_{cn_f} == 0 \tag{19}$$

Since component can fail for various reasons, there are different ways to resolve a component failure. One approach is to ensure that a component is redeployed on any node other than the node in which it failed (Equation 20). If a component keeps failing in multiple different nodes, then CHARIOT may need to consider another constraint to ensure the component is not redeployed on any node (Equation 21).

Let us assume component c_1 failed on node n_1.

$$c2n_{c_1 n_1 == 0} \tag{20}$$

$$\sum_{n=0}^{\beta} c2n_{c_1 n} == 0 \tag{21}$$

4.3.5 *Solution Computation Phase*

The third and final phase of the CPC algorithm involves computing a "least distance" configuration point, *i.e.*, a configuration point that is the least distance away from current configuration point. This computation ensures that a system always undergoes the least possible number of changes during reconfiguration. The distance is computed as the number of changes required to transition to the new configuration point. Since a configuration point is a component-instance-to-node mapping represented as C2N matrix (see Equation 1), the distance between two configuration points is the distance between their corresponding C2N matrices. In CHARIOT, the least distance constraint is encoded as shown below:

Least Distance Constraint The "least distance" constraint is used to ensure that we find a valid configuration point that is closest to the current configuration point. The distance between two configuration points is the distance between their corresponding C2N matrices. This distance is computed as shown in Equation 22. The distance between two valid configuration points A and B is the sum of the absolute difference between each element of the C2N matrices corresponding to the two configuration points.

To ensure we obtain least distance configuration point, an ideal solution would be to use minimize optimization (Equation 23), which is supported by SMT solvers like Z3. Like the Z3 maximize optimization, however, the Z3 minimize optimization implementation is experimental and does not scale well. In CHARIOT we therefore implement this constraint using an iterative logic, which upon every successful solution computation adds the distance constraint (Equation 22) before invoking the solver again to find a solution that is at a lesser distance compared to the previous solution. This iteration stops when no solution can be found, in which case the previous solution is used as the optimum (least distance away) solution.

$$config\_distance = \sum_{n=0}^{\beta} |c2n\_A_{cn} - c2n\_B_{cn}| \tag{22}$$

$$minimize(config\_distance) \tag{23}$$

At this point in the CPC algorithm, CHARIOT invokes the Z3 solver to check for a solution. If all constraints are satisfied and a solution is found, the CPC algorithm computes a set of deployment actions. CHARIOT computes deployment actions by comparing each element of the C2N matrix that represents the current configuration point with the corresponding element of the C2N matrix associated with computed solution, *i.e.*, the target configuration point. If the value of an element in the former is 0 and later is 1, CHARIOT adds a *START* action for the corresponding component instance on the corresponding node. Conversely, if the value of an element in the former is 1 and the later is 0, CHARIOT adds a *STOP* action. Applying this operation to each element of the matrix results in a complete set of deployment actions required for successful system transition.

4.3.6 *The Look-ahead Reconfiguration*

The CPC algorithm presented above yields a reactive self-reconfiguration approach since the algorithm executes after a failure is detected. As such, runtime reconfiguration incurs the time taken to compute a new configuration point and determine deployment actions required to transition to a new configuration. This approach may be acceptable for IoT systems consisting of non-real-time applications that can incur considerable downtime. For IoT systems involving real-time mission-critical applications, however, predictable and timely reconfiguration is essential. Since all dynamic reconfiguration mechanisms rely on runtime computation to calculate a reconfiguration solution, the time to compute a solution increases with the scale of the IoT system. The CPC algorithm is no different, as shown by experimental results in our prior work [26].

To address this issue, we therefore extend the CPC algorithm by adding a configurable capability to use a finite horizon look-ahead strategy that pre-computes solution and thus significantly improves the performance of the management engine. We call this capability the Look-ahead ReConfiguration (LaRC). The general goal of the LaRC approach is to pre-compute and store solutions, so it just finds the appropriate solution and applies it when required. When the CPC algorithm is configured to execute in the "look-ahead" mode, solutions are pre-computed every time the system state (*i.e.*, the current configuration point) changes.

The first pre-computation happens once the system is initially deployed using the default CPC algorithm. After a system is initially deployed, CHARIOT pre-computes solutions to handle failure events. These pre-computed solutions cannot be used for update events since these types of events change the system is such a way that the previously pre-computed solutions are rendered invalid. Once CHARIOT has a set of pre-computed solutions, therefore, failures are handled by finding the appropriate pre-computed solution, applying the found solution, and pre-computing solutions to handle future failure events. For update events, in contrast, the default CPC algorithm is invoked again (same as during initial deployment) to compute a solution. After a solution for an update event is computed, CHARIOT again pre-compute solutions to handle failure events.

Algorithm 2 Solution Pre-computation.

Input: nodes (*nodes_list*)

1: remove existing look-ahead information from the configuration space
2: **for** *node* in *node_list* **do**
3:     **if** *node* is alive **then**
4:         tmp_config_space = get configuration space
5:         mark *node* as failed in tmp_config_space
6:         *actions* = CPC algorithm on tmp_config_space
7:         **if** *actions* != *null* **then**
8:             *l_ahead* = new LookAhead instance
9:             $l\_ahead.failedEntity = node.name$
10:            $l\_ahead.failureKind$ = NODE
11:            $l\_ahead.deploymentActions = actions$
12:            store *l_ahead* in the configuration space

To pre-compute solutions, CHARIOT currently uses Algorithm 2. Since this paper focuses on node failures, Algorithm 2 only pre-computes solutions for node failures. Assuming that a system is in a stable state, this algorithm first removes any existing look-ahead solutions (line 1) since it is either invalid (update event) or already used (failure event). After this the algorithm iterates through each available node (line 2-3) and for each node, the algorithm creates a temporary copy of the configuration space (line 4), which includes the current (stable) configuration point. All subsequent actions are taken with respect to the temporary configuration space copy, so the original copy is not corrupted during the pre-computation computation process.

After a copy of the configuration space is made, the particular node is marked as failed (line 5) and the CPC algorithm is invoked (line 6). This pre-computation algorithm thus essentially injects a failure and asks the CPC algorithm for a solution. If a solution is found, the injected failure information and the solution is stored as an instance of the *LookAhead* class presented in Section 4.2.2 (line 7-12).

4.3.7 Summary of management layer

The description of the LaRC approach in Section 4.3.6 yields interesting observations with regards to the solution pre-computation algorithm. First, the current version of the solution pre-computation algorithm only considers node failures. We will alleviate this limitation in future work by adding system-wide capabilities to monitor, detect, and handle failures involving application processes, components, and network elements.

Second, the pre-computation algorithm specifically pre-computes solutions only for the next step,*i.e.*, the algorithm only looks one step ahead. We believe that *the number of steps to look-ahead* should be a configurable parameter as different classes of system might benefit from different setting of this parameter. For example, consider highly dynamic IoT systems that are subject to frequent failures resulting in bursts of failure events. For such systems, it is important to look-ahead more than one step at a time, otherwise multiple failures that happen in short timespan cannot be handled. However, for IoT systems that are comparatively more static, such as the smart parking system presented in Section 2.1, a higher Mean Time To Failure (MTTF) is expected, so pre-computed solutions need not look ahead more than one step at a time.

There is clearly a trade-off between time, space, and number of failures tolerated when considering the number of pre-computation steps. Multi-step pre-computation takes more time and space to store large number of solutions based on various permutation and combination of possible failures, but can handle bursts of failures. Conversely, a single-step pre-computation will be much faster and occupy less space, but it will be harder to handle bursts of failures.

An ideal solution would involve a dynamic solution pre-computation algorithm. The dynamism is with respect to the configuration of the pre-computation steps parameter. For any given system, however, we assume that there is an initial value that can change at runtime depending on the system behavior. Further investigating and implementing such a solution is part of our future work.

5. IMPLEMENTATION AND EVALUATION

This section describes and empirically evaluates the CHARIOT runtime implementation using the Smart Parking System use-case scenario presented in Section 2.1. Figure 14 depicts CHARIOT's implementation architecture, which consists of compute nodes comprising the layered stack shown in figure 2.

Each CHARIOT-enabled compute node hosts two platform services: a Node Monitor and a Deployment Manager. The Node Manager assesses the liveliness of its specific node, whereas the Deployment Manager manages the lifecycle of applications deployed on a node. In addition to compute nodes, CHARIOT's runtime also comprises one or more in-

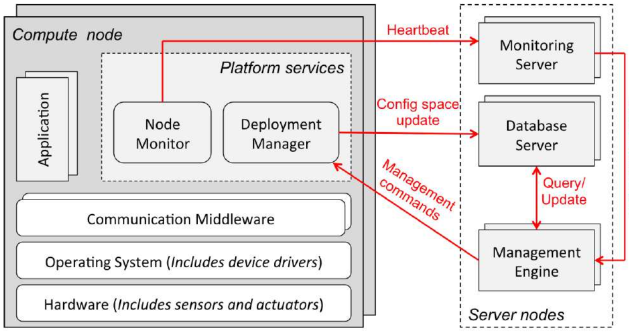


Figure 14: The Implementation Design of the CHARIOT Runtime.

stances of three different types of server nodes: (1) Database Servers that store system information, (2) Management Engines that facilitate failure avoidance, failure management, and operation management, and (3) Monitoring Servers that monitor for failures.[4]

CHARIOT's Node Manager is implemented as a ZooKeeper [14] client that registers itself with a Monitoring Server. In turn, the Monitoring Server is implemented as a ZooKeeper server and uses ZooKeeper's group membership functionality to detect member (node) additions and removals (*i.e.*, failure detection). This design supports dynamic resources, *i.e.*, nodes that can join or leave a cluster at any time. A group of Node Monitors (each residing on a node of a cluster) and one or more instances of Monitoring Servers define the monitoring infrastructure described in Section 4.

The Deployment Manager is implemented as a ZeroMQ [13] subscriber that receives management commands from a Management Engine, which is in turn implemented as a ZeroMQ publisher. The Management Engine computes the initial configuration point for application deployment, as well as subsequent configuration points for the system to recover from failures. After a Deployment Manager receives management commands from the Management Engine, it executes those commands locally to control the lifecycle of application components. Application components managed by CHARIOT can be in one of two states: active or inactive. A group of Deployment Managers, each residing on a node of a cluster, represents the deployment infrastructure described in Section 4.

A Database Server is an instance of a MongoDB server. For the experiments presented in Section 5.4, we only consider compute node failures, so deploying single instances of Monitoring Servers, Database Servers, and Management Engines fulfills our need. To avoid single points of failure, however, CHARIOT can deploy each of these servers in a replicated scenario. In the case of Monitoring Servers and Database Servers, replication is supported by existing ZooKeeper and MongoDB mechanisms. Likewise, replication is trivial for Management Engines since they are stateless. A Management Engine executes the CPC algorithm (see Section 4.3.2), with or without the LaRC configuration (see Section 4.3.6), using relevant information from a Database Server. CHARIOT can therefore have multiple replicas of Management Engines running, but only one performs reconfiguration algorithms. This constraint is achieved by implementing a rank-based leader election among different Management Engines. Since a Management Engine implements a ZeroMQ server and since ZeroMQ does not provide a service discovery capability by default, CHARIOT needs some mechanism to handle publisher discovery when a Management Engine fails. This capability is achieved by using ZooKeeper as a coordination service for ZeroMQ publishers and subscribers.

5.1 Application Deployment Mechanism

For initial application deployment, CHARIOT ML (see Section 4.1) is used to model the corresponding system that comprises the application, as well as resources on which the application will be deployed. This design-time model is then interpreted to generate a configuration space (see Section 4.3.1) and store it in the Database Server, after which point a Management Engine is invoked to initiate the deployment. When the Management Engine is requested to perform initial deployment, it retrieves the configuration space from the Database Server and computes a set of deployment commands. These commands are then stored in the Database Server and sent to relevant Deployment Managers, which take local actions to achieve a distributed application deployment. After a Deployment Manager executes an action, it updates the configuration space accordingly.

5.2 Failure and Update Detection

CHARIOT leverages capabilities provided by ZooKeeper to implement a node failure detection mechanism, which performs the following steps: (1) each computing node runs a Node Manager after it boots up to ensure that each node registers itself with a Monitoring Server, (2) when a node registers with a Monitoring Server, the latter creates a corresponding ephemeral node.[5], and (3) since node membership information is stored as ephemeral nodes in the Monitoring Server, it can detect failures of these nodes.

5.3 Reconfiguration Mechanism

After a failure is detected a Monitoring Server notifies the Management Engine, as shown in Figure 14. This figure also shows that the Management Engine then queries the Database Server to obtain the configuration space and reconfigure the system using relevant information from the configuration space and the detected failure.

5.4 Experimental Evaluation

Although we have previously used CHARIOT to deploy and manage applications on an embedded system comprising Intel Edison nodes (see chariot.isis.vanderbilt.edu/tutorial.html), this paper uses a cloud-based setup to evaluate CHARIOT at a larger scale. Below we first describe our experiment test-bed and then describe the application and set of events used for our evaluation. We next present evaluation of the default CPC algorithm and evaluate the CPC algorithm with the LaRC algorithm. Finally, we present CHARIOT resource consumption metrics.

5.4.1 Hardware and Software Testbed

[4]Since failure detection and diagnosis is not the primary focus of this paper, our current implementation focuses on resolving node failures, though CHARIOT can be easily extended to support mechanism to detect component, process, and network failures.

[5]ZooKeeper stores information in a tree like structure comprising simple nodes, sequential nodes, or ephemeral nodes.

Table 1: Sequence of Events used for evaluation of CPC algorithm.

Events	Description
1	Initial deployment over 21 nodes (10 processing nodes, 10 camera nodes, and 1 terminal node) resulting in 23 component instances; 10 different component instances related to the *occupancy_detector* functionality due to its corresponding cluster replication constraint, 10 different component instances related to the *image_capture* functionality due to its corresponding per-node replication constraint associated with camera nodes (we have 10 camera nodes), a component instance related to the *client* functionality due to its corresponding per-node replication constraint associated with terminal nodes (we have 1 terminal node), and a component instance each related to the *load_balancer*, and *parking_manager* functionalities.
2	Failure of a camera node. No reconfiguration is required for this failure as a camera node hosts only a node-specific component that provides the *image_capture* functionality.
3	Failure of the processing node that hosts a component instance each related to the *load_balancer* and *parking_manager* functionalities. This results in reconfiguration of the aforementioned two component instances. Furthermore, since the processing node hosts an instance of the *occupancy_detection* functionality, the number of component instances related to this functionality decreases from 10 to 9. Since 9 is still within the provided redundancy range (min = 7, max = 10), however, this component instance does not get reconfigured.
4	Failure of the processing node on which the component instance related to the *parking_manager* functionality was reconfigured to as the result of the previous event. This event results in the *parking_manager* functionality related component instance to again be reconfigured to a different node. Moreover, the number of component instances related to the *occupancy_detector* functionality decreases to 8, which is still within the provided redundancy range; as such, reconfiguration of that component instance is not required.
5	Failure of the processing node on which the component instance related to the *load_balancer* functionality was reconfigured to as result of event 3. This event results in the component instance being reconfigured again to a different node. Also, the number of component instances related to the *occupancy_detector* functionality decreases to 7, which is still within the provided redundancy range so no reconfiguration is required.
6	Failure of another processing node. This node only hosts a component instance related to the *occupancy_detector* functionality. As a result of this failure event, therefore, the provided redundancy range associated with the *occupancy_detector* functionality is violated since the number of corresponding component instances decreases to 6. This component instance is then reconfigured to a different node to maintain at least 7 instances of the *occupancy_detector* functionality.
7	Failure of the single available terminal node on which the component instance related to the *client* functionality was deployed as part of the initial deployment (event 1). This event results in an invalid system state since there are no other terminal nodes and thus no instances of *client* functionality are available.
8-31	Hardware updates associated with addition of 2 terminal nodes, 11 processing nodes, and 11 camera nodes. Due to associated per-node replication constraints, addition of a terminal node results in deployment of a component instance associated with the *client* functionality. Similarly, adding a camera node results in deployment of a component instance associated with the *image_capture* functionality. Adding processing node does not result in any new deployment, however, since it is not associated with a per-node replication constraint.
32	Failure of a processing node that hosts a component instance related to the *occupancy_detector* functionality. This results in reconfiguration of the component instance to a different node.
33	Failure of another processing node, which hosts no applications. Therefore, no reconfiguration is required.
34	Failure of a camera node. Again, no reconfiguration is required (see event 2 above).

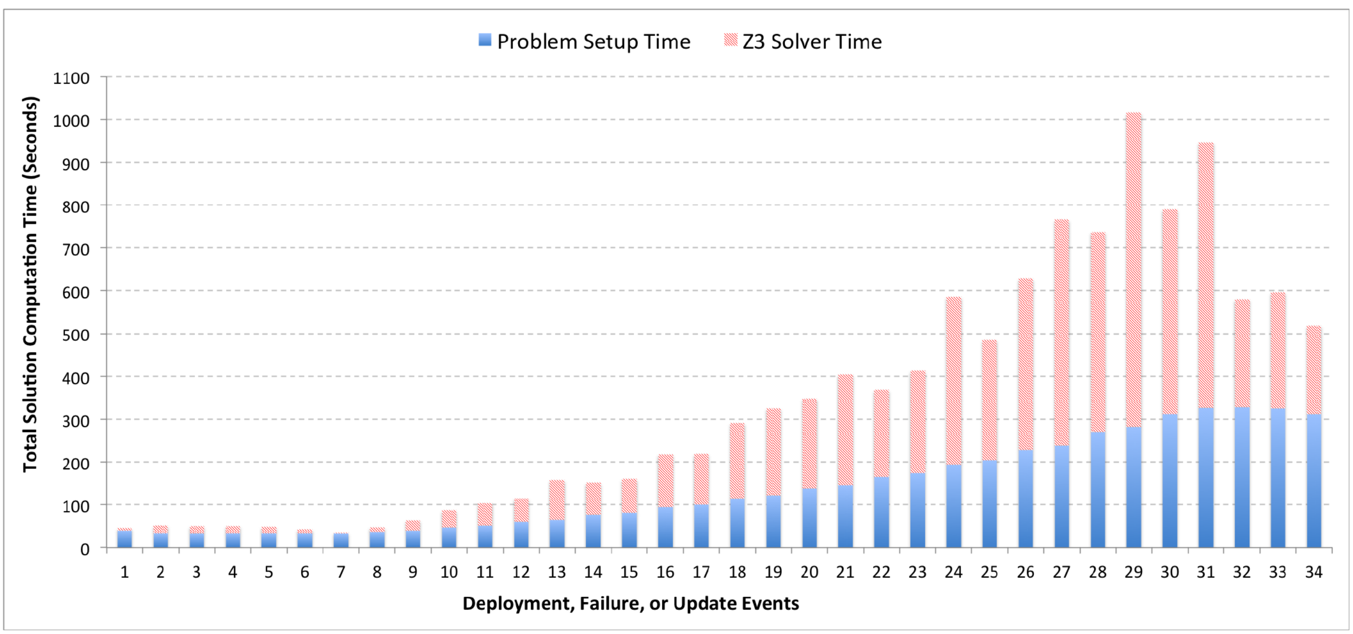


Figure 15: Default CPC Algorithm Performance. (Please refer to Table 1 for details about each event shown in this graph.)

Our testbed comprises 45 virtual machines (VMs) each with 1GB RAM, 1VCPU, and 10GB disk in our private OpenStack cloud. We treat these 45 VMs as embedded compute nodes. In addition to these 45 VMs, 3 additional VMs with 2 VCPUs, 4 GB memory, and 40GB disk is used as server nodes to host Monitoring Server, Database Server, and Management Engine (see Figure 14). All these VMs ran Ubuntu 14.04 and were placed in the same virtual LAN.

5.4.2 Application and Event Sequence

To evaluate CHARIOT, we use the smart parking system described in Section 2.1. We divide the 45 compute nodes into 21 processing nodes (corresponding to the *edison* node template in Figure 7), 21 camera nodes (corresponding to the *wifi_cam* node template in Figure 7), and 3 terminal nodes (corresponding to the *entry_terminal* node template in Figure 7). The goal description we used is the same shown in Figure 9, except we increase the replication range of the *occupancy_detector* functionality to minimum 7 and maximum 10.

To evaluate the default CPC algorithm we use 34 different events presented in Table 1. As shown in the table, the first event is the initial deployment of the smart parking system over 21 nodes (10 processing nodes, 10 camera nodes, and 1 terminal node). This initial deployment results in a total of 23 component instances. After initial deployment, we introduce 6 different node failure events, one at a time. We then update the system by adding 2 terminal nodes, 11 processing nodes, and 11 camera nodes. These nodes are added one at a time, resulting in a total of 45 nodes (including the 6 failed nodes). These updates are examples of intended updates and show CHARIOT's operations management capabilities. After updating the system, we introduce three more node failures.

5.4.3 Evaluation of the Default CPC Algorithm

Figure 15 presents evaluation of the default CPC algorithm using application and event sequence described above. To evaluate the default CPC algorithm we use the total solution computation time, which is measured in seconds. The total solution computation time can be decomposed into two parts: (1) problem setup time and (2) Z3 solver time. The problem setup time corresponds to the first two phases of the CPC algorithm (see Section 4.3.3 and Section 4.3.4), whereas the Z3 solver time corresponds to the third phase of the CPC algorithm (see Section 4.3.5).

Figure 15 shows that for initial deployment and the first 5 failure events, the total solution computation time is similar (average = 48 seconds) because the size of the C2N matrix and associated constraints created during the problem setup time are roughly the same. The 6th failure (7th event in Figure 15), is associated with the one and only terminal node in the system. The Z3 solver therefore quickly determines there is no solution, so the Z3 solver time for the 7th event is the minimal 1.74 seconds.

Events 8 through 31 are associated with a system update via the addition of a single node per event. These events show that for most cases the total solution computation time increases with each addition of node. The problem setup time increases consistently with increase in the number of nodes because the size of the C2N matrix, as well as the number of constraints, increases with an increase in the number of nodes. The Z3 solver time also increases with increase in number of nodes in the system, however, it does not increase as consistently as the problem setup time due to the least distance configuration computation presented in Section 4.3.5. The number of iterations, and therefore the total time, taken by the Z3 solver to find a solution with least distance is non-deterministic. If a good solution (with respect to distance) is found in the first iteration, it takes less number of iterations to find the optimal solution. We demonstrate this non-deterministic behavior using experimental results in Section 5.4.6.

Finally, events 32 through 34 are associated with more node failures. The total solution computation time therefore decreases due to the decrease in number of nodes and component instances, which results in a smaller C2N matrix and a fewer number of constraints.

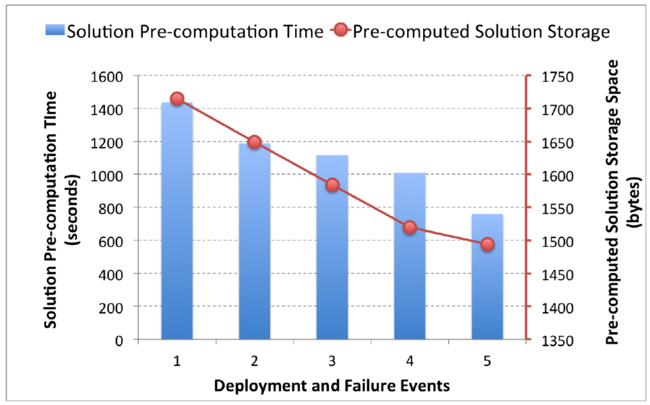


Figure 16: Solution Pre-computation Time for CPC with LaRC.

5.4.4 Evaluation of the CPC algorithm with LaRC

For the purpose of this evaluation we use the first 5 events since this is enough to showcase the trade-off between the default CPC algorithm and the CPC algorithm with LaRC. In this approach, the total solution computation time (apart from the initial deployment) is the time taken to query the database for pre-computed solution. This time is significantly lower (average = 0.0085 seconds) than that for the default CPC algorithm (average = 48 seconds).

To demonstrate the trade-off between the two versions of the CPC algorithm, Figure 16 presents the time taken for solution pre-computation and the space required to store pre-computed solution in (the solution for failure event i+1 is computed when the reconfiguration action for the failure event i is being applied). As shown in this figure, the time taken to pre-compute solution after initial deployment is 1,400 seconds, which is the time needed to pre-compute solution for 21 node failures (initial configuration). To store this pre-computed solution 1,715 bytes of storage space is used. Events 2 through 5 represent node failures and the solution pre-computation time and storage used to store the pre-computed solution decreases with each failure because failures result in less number of scenarios for which we need to pre-compute a solution.

5.4.5 Resource Consumption

To demonstrate the usability of CHARIOT in IoT systems, we present various resource consumption of CHARIOT entities (Deployment Manager and Node Monitor, as shown in Figure 14) that run on each compute node. The resource consumption numbers only consider the chariot management entities and not the actual application being managed. Moreover, for the purpose of this evaluation we categorize the compute nodes based on their lifetime (short, medium, long) and randomly pick 4-5 nodes form each category. Nodes A, B, C, D, and E are nodes with short lifetime (less than 15 minutes); nodes F, G, H, and I are nodes with medium lifetime (between 110 and 154 minutes); nodes J, K, L, and M are nodes with long lifetime (between 200 and 235 minutes).

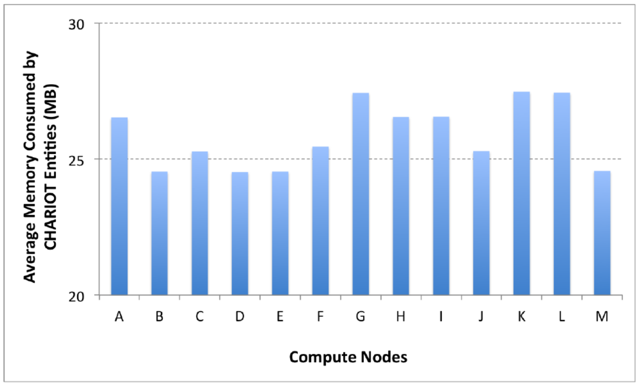


Figure 17: Average Memory Consumption.

Figure 17 presents the average memory consumed by CHARIOT entities running on 13 nodes mentioned above throughout their lifetime. This figure shows that the average memory consumption is close to slightly above or below 25 MB in each node. Similarly, Figure 18 presents the average network bandwidth consumed by CHARIOT entities running on the aforementioned 13 nodes throughout their lifetime. This figure shows that the network bandwidth used to send and receive information is minimal and predictable. We do not show the CPU utilization since it was between 0 - 0.5%.

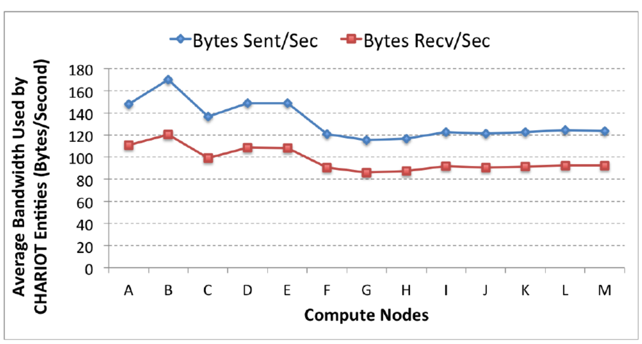


Figure 18: Average Network Bandwidth Consumption.

The results presented above show that the CHARIOT infrastructure is not resource intensive and therefore can be used for resource-constrained IoT devices. CHARIOT is currently written using Python[6], though we intend to convert our code to C++/Golang to further improve performance.

[6] github.com/visor-vu/chariot

5.4.6 *Analyzing Performance of the CPC algorithm*

To further analyze the CPC algorithm's performance, the experiment presented in Section 5.4.3 was replicated in a single machine simulation environment. This new analysis was run on a 64 bit Windows 7machine with 8 GB memory and 8 cores resulting in 4 GB of additional memory and 6 additional cores compared to the distributed testbed used for experiment presented in Section 5.4.3. Figure 19 presents the overall performance of the CPC algorithm using application and event sequence described in Section 5.4.2. Figure 20 compares the performance of CPC algorithms in simulated and non-simulated environment. The results in this figure show the performance improvement facilitated by the more resourceful hardware used in the simulated environment.

Figure 21 analyzes the Z3 solver time jitter by comparing the Z3 solver time with the corresponding Z3 problem complexity. Here, the Z3 problem complexity is a metric defined as the product of (1) total number of solver assertions, which indicates the size of the problem being solved by the Z3 solver, and (2) total number of least-distance iterations, which indicates the number of times a problem is solved by the Z3 solver. As shown in the figure, barring few anomalies, the Z3 solver time depends on the Z3 problem complexity.

Finally, to determine possible performance bottlenecks, the default CPC algorithm was further analyzed using different initial deployment scenarios (based on varying scale) of the application presented in Section 5.4.2. Figure 22 presents the total solution computation time, divided into three different phases of the CPC algorithm, for six different initial deployment scenarios. The first deployment scenario comprises 11 nodes and 10 components; the second deployment scenario comprises 22 nodes and 18 components; the third deployment scenario comprises 33 nodes and 26 components; the fourth deployment scenario comprises 44 nodes and 34 components; the fifth deployment scenario comprises 55 nodes and 42 components; and the sixth deployment scenario comprises 66 nodes and 50 components.

Figure 22 shows that the second phase of the CPC algorithm, which corresponds to the constraint encoding phase, contributes to majority of the total solution computation time. Further analysis of the constraint encoding phase of the CPC algorithm (shown in Figure 23) shows that the dependency constraint encoding (see Equation 6) is the main bottleneck as it accounts for more than 90% of the constraint encoding time. This result occurs 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.

6. CONCLUDING REMARKS

This paper described the structure and functionality of CHARIOT, which is an orchestration middleware we developed to meet the resilience requirements of IoT systems. The following is a summary of our lessons learned from developing and using CHARIOT:

Lesson 1: Design-time system description should be generic. If the objectives of an application and the functionalities that it requires can be specified in a generic manner, CHARIOT can create an online mechanism that maps the system objectives to required resources based on functionality decomposition and functionality-component asso-

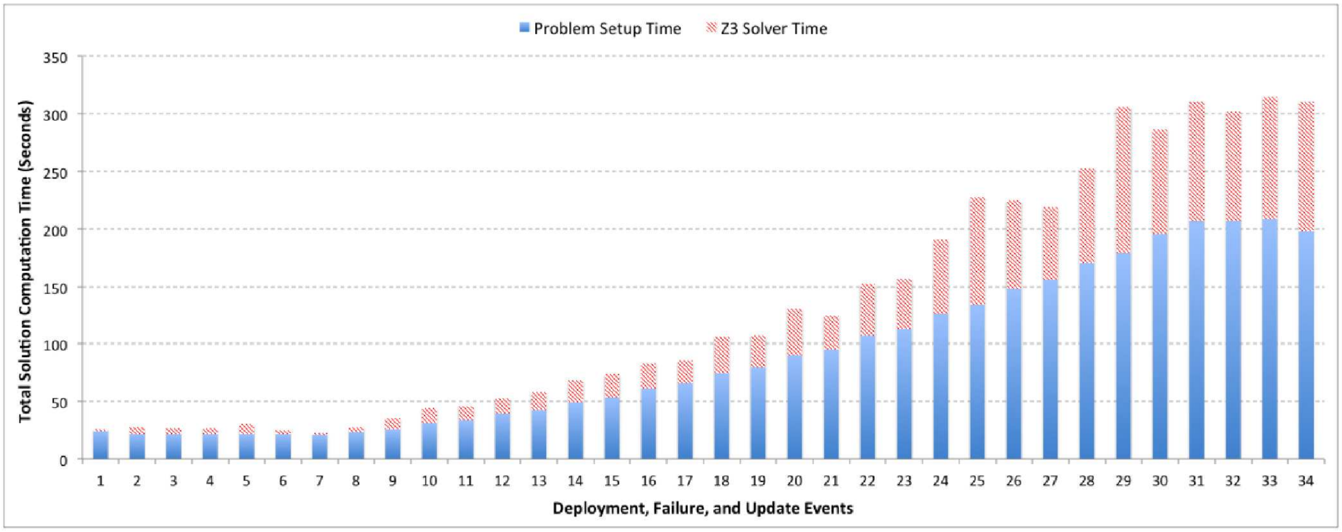


Figure 19: Default CPC Algorithm Performance in Simulated Environment. (Please refer to Table 1 for details about each event shown in this graph.)

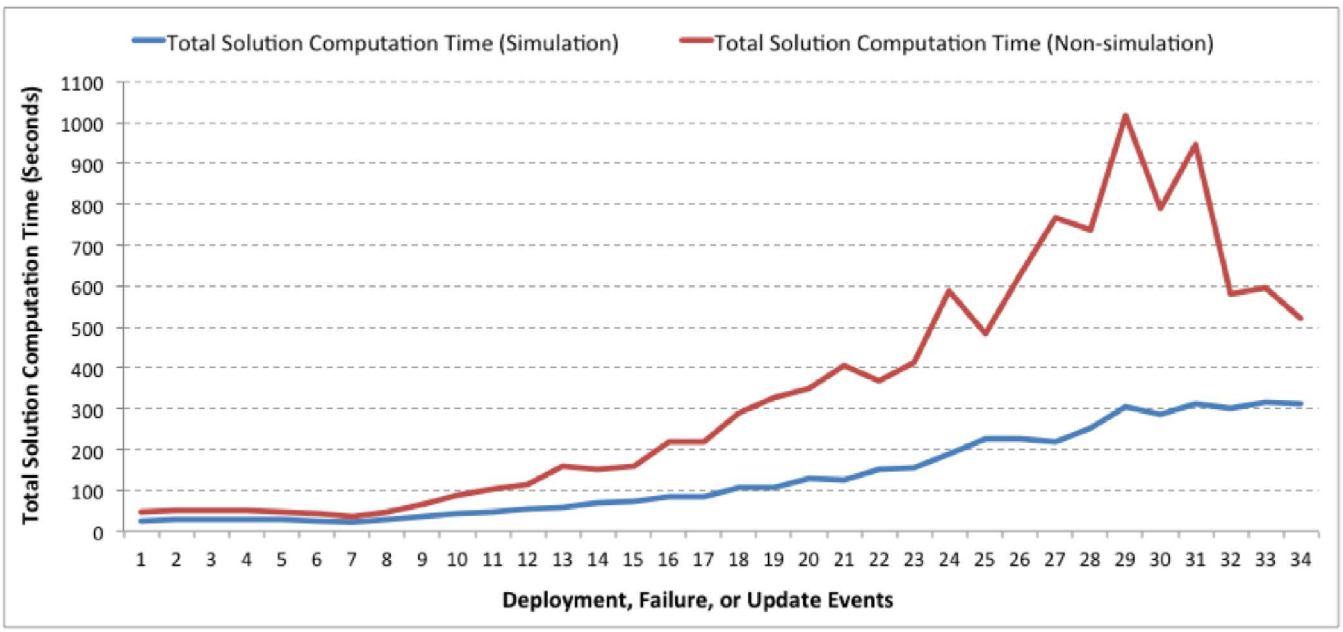


Figure 20: Default CPC Algorithm Performance Comparison between Non-simulated and Simulated Environments. (Please refer to Table 1 for details about each event shown in this graph.)

ciation. It is important, however, to extend this concept to support the idea of graceful degradation. In future work, we are modeling quality of service (QoS) functions that provide mechanisms for evaluating the performance of a component's functionality based on available resources. These QoS mechanisms can be helpful when CHARIOT needs to arbitrate between different system objectives.

Lesson 2: Design-time and runtime system information can be used to encode constraints at runtime. Using design-time system description and runtime system representation, constraints can be dynamically encoded to represent various system requirements. These constraints can aid online reconfiguration via the use of state-of-the-art solvers such as Z3, which is a SMT solver. To minimize downtime, however, efficient pre-computation of reconfiguration steps is necessary. CHARIOT's Look-ahead approach described in this paper is a step in this direction.

Lesson 3: Dynamic online reconfiguration is time consuming. Online reconfiguration is time consuming and is thus not suitable for low latency IoT systems with stringent real-time constraints. For those types of systems, therefore, it is important to include redundancy in the deployment logic. The CHARIOT modeling language and reconfiguration logic supports these redundancy concepts.

Lesson 4: Failure reconfiguration approach can be

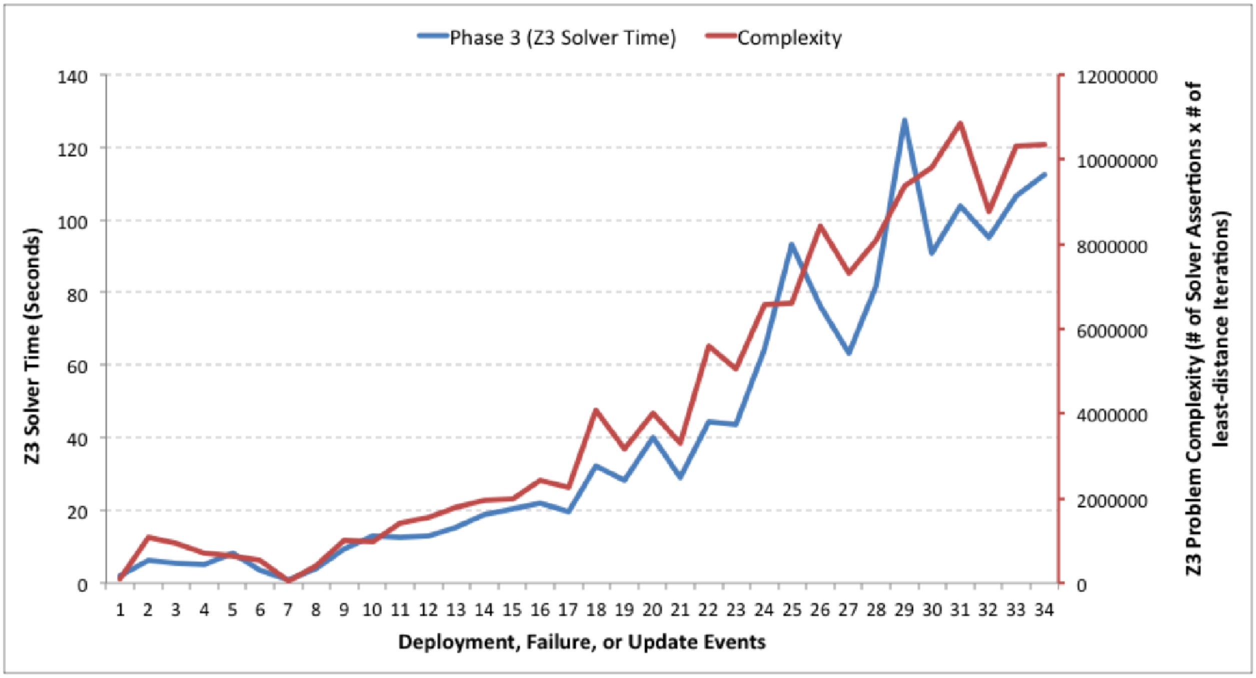


Figure 21: The Z3 Solver Time Jitter versus the corresponding Problem Complexity. (Please refer to Table 1 for details about each event shown in this graph.)

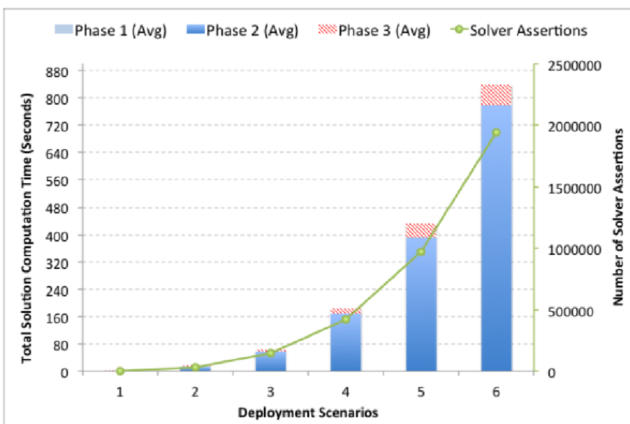


Figure 22: Solution Computation Time for Different Initial Deployment Scenarios.

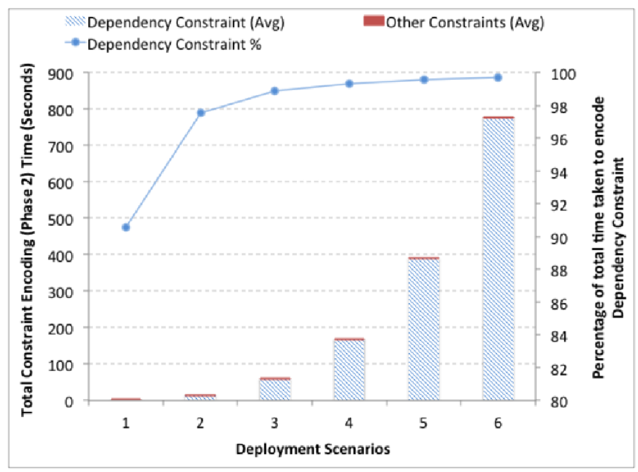


Figure 23: Breakdown of Total Constraint Encoding Time into Different Constraints.

extended to support system updates as well. CHARIOT's reconfiguration framework can be extended to address IoT system evolution, which corresponds to the addition of computational capabilities or new software applications. By generalizing and automating reconfiguration steps CHARIOT can be adopted to IoT apps in many domains.

Our future work on CHARIOT will analyze the time complexity of the reconfiguration analysis and develop strategies to minimize downtime to facilitate its use in safety- and time-critical IoT application domains.

Acknowledgments

This work is sponsored in part by Siemens Corporate Technology and in part by a NSF grant 1528799. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Siemens Corporate Technology or NSF.

7. REFERENCES

[1] Apache Zookeeper. https://zookeeper.apache.org/.

[2] Xtext. https://eclipse.org/Xtext/.

[3] S. S. Andrade and R. J. de Araújo Macêdo. A non-intrusive component-based approach for deploying unanticipated self-management behaviour. In *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS'09. ICSE Workshop on*, pages 152–161. IEEE, 2009.

[4] E. Asmare, A. Gopalan, M. Sloman, N. Dulay, and E. Lupu. Self-management framework for mobile autonomous systems. *Journal of Network and Systems Management*, 20(2):244–275, 2012.

[5] K. Benson, C. Fracchia, G. Wang, Q. Zhu, S. Almomen, J. Cohn, L. D'arcy, D. Hoffman, M. Makai, J. Stamatakis, et al. Scale: Safe community awareness and alerting leveraging the internet of things. *IEEE Communications Magazine*, 53(12):27–34, 2015.

[6] 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*, pages 13–16. ACM, 2012.

[7] R. Capilla, J. Bosch, P. Trinidad, A. Ruiz-Cortés, and M. Hinchey. An overview of dynamic software product line architectures and techniques: Observations from research and industry. *Journal of Systems and Software*, 91:3–23, 2014.

[8] L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

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

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

[11] G. T. Heineman and W. T. Councill, editors. *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[12] O. H??ftberger and R. Obermaisser. Runtime evaluation of ontology-based reconfiguration of distributed embedded real-time systems. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*, 2014.

[13] P. Hintjens. *ZeroMQ: Messaging for Many Applications*. " O'Reilly Media, Inc.", 2013.

[14] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[15] T. Kurtoglu, I. Y. Tumer, and D. C. Jensen. A functional failure reasoning methodology for evaluation of conceptual system architectures. *Research in Engineering Design*, 21(4):209–234, 2010.

[16] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[17] J.-c. Laprie. From dependability to resilience. In *In 38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks*. Citeseer, 2008.

[18] N. Mahadevan, A. Dubey, D. Balasubramanian, and G. Karsai. Deliberative, search-based mitigation strategies for model-based software health management. *Innovations in Systems and Software Engineering*, 9(4):293–318, 2013.

[19] N. Mahadevan, A. Dubey, and G. Karsai. Application of software health management techniques. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 1–10, New York, NY, USA, 2011. ACM.

[20] N. Mahadevan, A. Dubey, and G. Karsai. Application of software health management techniques. In *SEAMS*, pages 1–10, 2011.

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

[22] R. Mehrotra, A. Dubey, S. Abdelwahed, and R. Krisa. Rfdmon: A real-time and fault-tolerant distributed system monitoring approach. In *The Eighth International Conference on Autonomic and Autonomous Systems*, pages 57–63, 2012.

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

[24] S. Nannapaneni, A. Dubey, S. Abdelwahed, S. Mahadevan, S. Neema, and T. Bapty. Mission-based reliability prediction in component-based systems. *International Journal of Prognostics and Health Management*, 7(001), 2016.

[25] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, pages 305–320, 2014.

[26] 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*, 2016.

[27] S. M. Pradhan, A. Dubey, A. Gokhale, and M. Lehofer. Chariot: a domain specific language for extensible cyber-physical systems. In *Proceedings of the Workshop on Domain-Specific Modeling*, pages 9–16. ACM, 2015.

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

[29] A. Shaukat, G. Burroughes, and Y. Gao. Self-reconfigurable robotics architecture utilising fuzzy and deliberative reasoning. In *SAI Intelligent Systems Conference (IntelliSys), 2015*, 2015.

[30] W. Torres-Pomales. Software fault tolerance: A tutorial. 2000.

[31] 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*, 9(1):228–236, 2013.

[32] L. M. Vaquero and L. Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *SIGCOMM Comput. Commun. Rev.*, 44(5):27–32, Oct. 2014.

[33] 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*, pages 43–48. ACM, 2014.