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

The 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 this need, we present CHARIOT, which is orchestration middleware capable of autonomously managing IoT systems that comprises 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 in persistent data storage 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 is responsible for parking space management.

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], 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 can host 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. 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 cyber-physical nature of applications and the system hardware.

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 mobile nature of some systems (for example, swarm of UAVs, fractionated satellites). Unlike traditional enterprise architectures whose resource constraints narrowly focus on only CPU, memory, storage and network, IoT systems require a capability 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 the uncertainties and constraints mentioned above, 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

DOI: http://dx.doi.org/10.1145/0000000.0000000

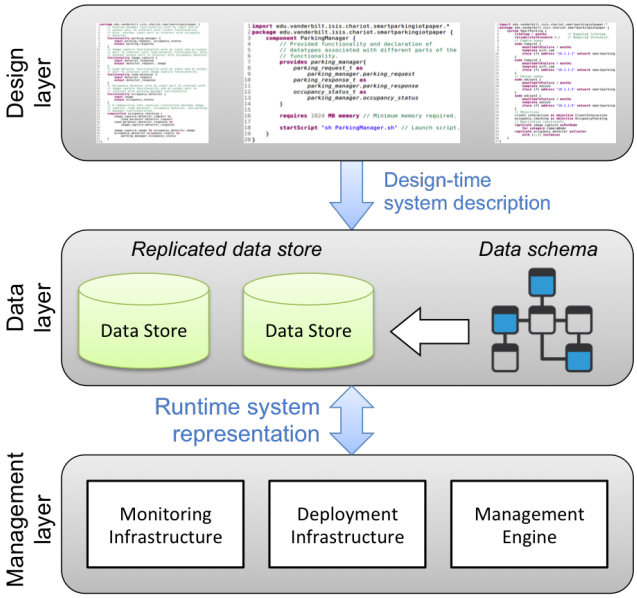


Figure 1: The Layered Architecture of CHARIOT.

software changes over time. In the context of IoT systems, resilience involves more than fault tolerance since a resilient system must not only tolerate failures, but should also adapt seamlessly to planned and unplanned changes. Moreover, since IoT systems comprising edge resources are often remotely deployed, autonomous system resilience mechanisms can help ensure availability and cost-effective management strategies.

Solution approach → Autonomous resilience management mechanisms. To address the challenges described above, IoT systems should be equipped with autonomous mechanisms that enable the analysis and management of (1) the overall system goals describing the required applications that must be available, (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 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.

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 design layer captures system specifications in terms of different kinds of available hardware resources, software applications, and the resource provided/required relationship between them. As shown in Section 4, 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 this middle of CHARIOT's stack is a data layer implemented using a persistent data storage and the corresponding well-defined schema to store 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 present the details of this contribution in Section 5.

Contribution 3: A management engine to facilitate autonomous resilience. The bottom of CHARIOT's stack is a management layer that comprises monitoring and deployment infrastructures, as well as a novel management engine that facilitates application (re)configuration as a mechanism to support autonomous resilience. As described in Section 6, this management engine uses IoT system information stored in the persistent storage 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. This approach extends our prior work [26] by (1) adding the capability to compute exact component instances from available component types, (2) encoding redundancy patterns using SMT constraints, and (3) adding capability to use a finite horizon look-ahead strategy that pre-computes solutions to significantly improve the performance of CHARIOT's management engine.

Contribution 4: Distributed implementation and evaluation of CHARIOT. Section 7 describes an implementation of CHARIOT, which 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 also describe a smart-parking application responsible for parking space management that serves as a use case scenario to present experimental evaluation to show how CHARIOT, as an orchestration middleware, is suitable to manage edge computing for IoT systems.

Paper organization. The remainder of this paper is organized as follows: Section 2 summarizes the problem background and surveys related research to distinguish our work on CHARIOT presented in this paper; Section 3 provides detailed coverage of the research topics addressed by this paper; Sections 4-7 describes in detail our four contributions described above; and Section 8 presents concluding remarks and future work.

2. BACKGROUND AND RELATED WORK

Software plays an essential role in mission-critical systems both as the provider of functionality and as a universal system integrator. The survivability and resilience of an IoT system thus critically depends on software. If any hardware or software components in an IoT system fail the system should be able to recover and survive with the help of capabilities provided by the software. Moreover, the complexity of IoT systems has progressed to the point where zero-defect systems (consisting of both hardware and software) are hard and costly to develop, validate, and sustain. An IoT system must therefore be prepared to handle a myriad of failures. This section describes related research to distinguish it from our work on CHARIOT presented in the remainder of this

paper.

2.1 Redundancy-based Strategies

Fault tolerance in computing has a long history, but resilience [17][1] is beyond the capabilities of conventional fault-tolerant approaches since resilience means "adapting to change." Conventional fault tolerance techniques are based on redundancy together with comparison or acceptance-based testing, especially for mission-critical systems with extremely high availability requirements. 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.

Redundancy-based resilience techniques use comparison (*e.g.*, a voter) or acceptance check (*e.g.*, an acceptance test) schemes to decide if a component is working correctly or not, as well as pass on the 'correct' output to the downstream subsystem. 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.

2.2 Reconfiguration-based Strategies

Reconfiguration-based resilience 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 the computing nodes and applications, watchdogs associated with hardware and software operation, 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) based on a table look-up or by using rule-based or model-based reasoning. 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 form the basis for the diagnosis system applied in this paper.

There are two types of reconfiguration-based techniques: offline strategies using pre-specified reconfiguration rules and dynamic online reconfiguration. Statically-specified reconfiguration techniques require explicit and declarative modeling of how a system should be reconfigured before it is deployed. Conversely, dynamic reconfiguration techniques require implicit and symbolic capturing of system behavior as a mathematical model that is dynamically searched at runtime to find solutions used to repair and restore a system.

2.2.1 *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 [3, 28, 2] presents different policy-based approaches. In [3], 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 [2], 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]. In general, statically-specified reconfiguration techniques 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, however, since these systems are dynamic and thus all possible runtime scenarios cannot be pre-determined at design-time.

2.2.2 *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 of utmost importance 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 systems based on services. At desin-time, the schedulability and complexity of a system is analyzed and fine tuned to bound sources of

[1] Resilience [17] is a system level property—by definition any part of the system can fail, yet the system should be able to keep providing the services it supports.

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 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 ours, 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.

3. PROBLEM FORMULATION AND CHARIOT OVERVIEW

This section describes the problem we address with the work on CHARIOT 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 (both online and offline) due to failures or addition and removal of resources. Figure 2 shows a typical node of this distributed platform created by these nodes. Each node contains a layered software stack consisting of an operating system (OS), communication middleware, and platform services (such as failure monitoring and detection, application management, and network analysis).

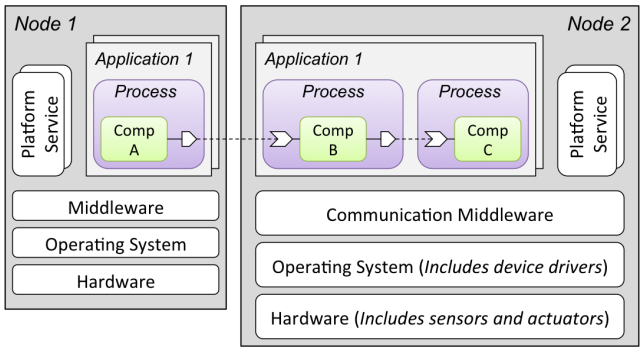


Figure 2: A Component-based IoT Application Model.

This distributed platform supports the needs of IoT applications, which may span multiple nodes for reasons related to the availability of resources, *e.g.*, some nodes may have sensors, some may have actuators, some may have the computing or storage resources, some need more than the processing power available on one node. These 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[2] 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. 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 this distributed 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 goal must be available as long as possible. This notion of functionality requirement can also be hierarchical, *i.e.*, we can have a notion of a high-level functionality that can 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 just 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, such as (a) availability of memory and storage capacity for components to use, (b) availability of devices and software artifacts (libraries) for components to use, and (c) network links between nodes of a system, which restricts deployment of component instances with inter-dependencies.

3.1 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 make it easier for clients to use 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; it consists of a number of pairs of camera nodes (wireless camera) and processing nodes (Intel Edison module mounted on Arduino board)[3] placed on the ceiling to provide coverage for multiple parking spaces. Each pair of 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 to as they enter the parking lot.

In addition to the hardware devices that comprises the

[2]In this context, functionalities are synonymous to services or capabilities associated with a component.
[3]https://www.arduino.cc/en/ArduinoCertified/IntelEdison

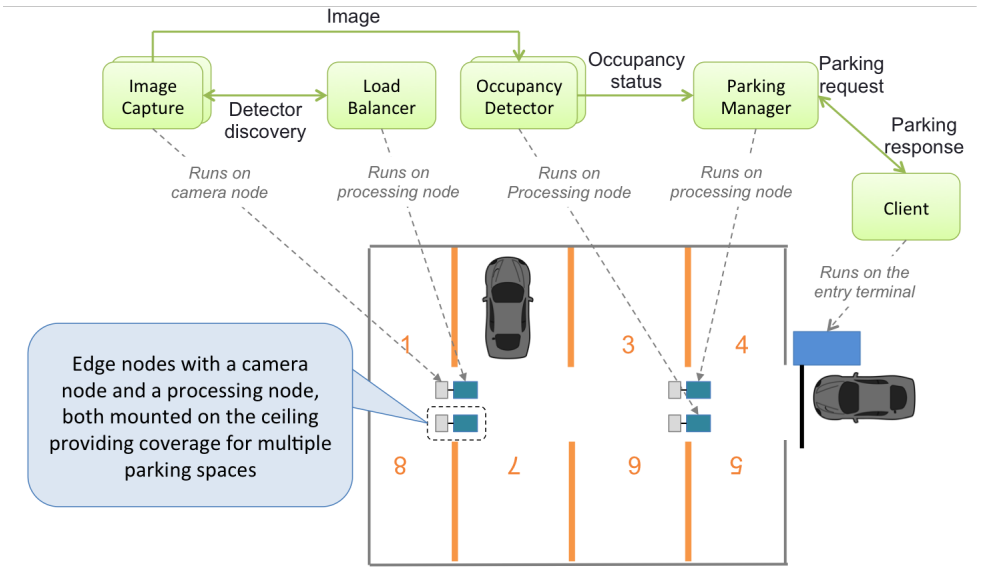


Figure 3: An Overview of the Parking Management System Case Study.

system, Figure 3 also shows a distributed application consisting of five different types of components deployed on the hardware outlined above. An *ImageCapture* component runs on a camera node and periodically captures an image and sends it to an *OccupancyDetector* component that runs on a processing node. An *OccupancyDetector* component detects vehicles in an image and determines occupancy status of parking spaces captured in the image. For an *ImageCapture* component to send images to an *OccupancyDetector* component, it must first find the *OccupancyDetector* by using the *LoadBalancer* component, which also runs on a processing node. The *LoadBalancer* component keeps track of the different *OccupancyDetector* components available. After an *OccupancyDetector* component analyses an image for occupancy status of different parking spaces, it sends the result to the *ParkingManager* component, which keeps track of occupancy status of the entire parking lot. The *ParkingManager* component also runs on a processing node. The fifth and final component comprising the smart parking application is the *Client* component, which runs on the entry terminal and interacts with users to allow them to query, reserve, and use the parking lot.

3.2 Problem Statement

Resilience can be described in terms of ensuring the survivability of the system's high-level mission. Anything can go wrong at any time, including faults in the computing and communication hardware, in the platform, and in the application software. Over time, nodes may be added or removed. Also, the applications can change as well in response to new requirements. This results in dynamic systems; however, the degree of dynamism can vary significantly based on the types of systems. For example, the smart parking example presented in Section 3.1 is an example of a less dynamic system as the physical resources are spatially static and any dynamism is related to system update associated with addition or removal of resources. However, a cluster of UAVs or fractionated satellites are example of highly dynamic systems. Furthermore, many IoT systems may operate continuously over years, so changes must be rolled out during production. Moreover, unanticipated changes in the system (*e.g.*, erroneous updates) or in the environment (*e.g.*, physical obstructions resulting in loss of wireless signal and therefore network partition) must be survivable, *i.e.*, the IoT system should recover automatically without requiring manual intervention.

The case study shown in Section 3.1 motivates the need for orchestration middleware like CHARIOT to manage deployment, execution, and update phases. For example, middleware capable of deploying distributed applications is quite useful in a large multi-level parking garage. Likewise, managing the life-cycle of previously deployed applications during the execution phase is also important. Factors that could trigger execution phase management actions vary from optimization to resilience. 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. We therefore require middleware that can detect failures, determine if a failure affects the *ParkingManager* component, and if it does, then autonomously reconfigure the system so that a *ParkingManager* component is always available. Reconfiguration of an IoT system requires a certain amount of time, which might not be acceptable for safety-critical, real-time systems. In such scenarios, the only viable solution is to have redundant copies of applications.

Addressing the problems described above requires a solution that holistically addresses both (a) the design-time challenges of capturing the system description, and (b) the runtime challenges of implementing the dynamic reconfiguration strategies. In particular, the following key factors must be considered by such a solution:

1. *Failure avoidance*, which is necessary since failures can cause downtime. IoT systems consist of hardware components that degrade over time and hence eventually fail. Likewise, software applications can also fail due to various defects. Since these types of failures cannot

be avoided altogether in an IoT system, one approach to handling failure is to minimize its impact.

2. *Failure management*, which 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*, which 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 resources and software applications.

3.3 Overview of the CHARIOT Ecosystem

An overview of our approach to solving these requirements is presented in Figure 4. As shown in the figure, the design-time aspect includes a modeling language and associated interpreters. We describe the modeling language in Section 4. The runtime aspect includes entities that comprise a self-reconfiguration loop, which implements a *sense-plan-act* closed-loop to (a) detect and diagnose failures, (b) compute reconfiguration, and (c) reconfigure the system. With respect to the layers of CHARIOT previously described in Section 1, we can say that 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 both aspects.

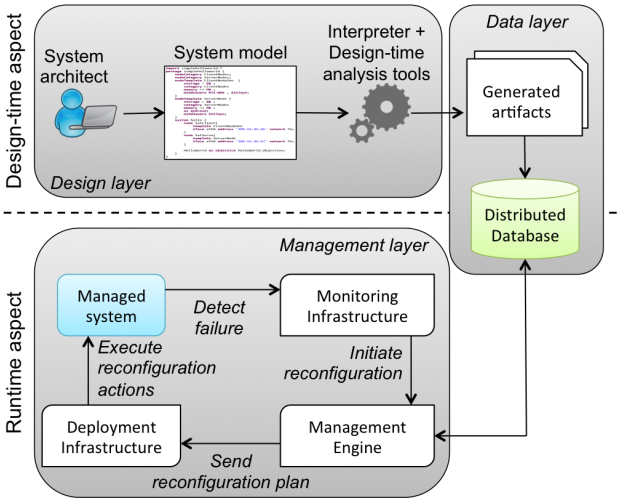


Figure 4: Overview of the self-reconfiguration mechanism.

CHARIOT handles failure avoidance via functionality redundancy and optimized distribution of redundant functionalities. The general idea here is to be able to tolerate more failures by strategically deploying redundant copies of components that provide critical functionalities, such that more failures are avoided/tolerated without having to reconfigure the system; further description of functionality redundancy is presented in Section 4.2.

Failure management is handled using the above-described sense-plan-act loop. The *Monitoring Infrastructure* is responsible for detecting failures; this is the *sensing* phase. After failure detection and diagnosis, it is the responsibility of the *Management Engine* to determine the actions needed to reconfigure the system such that failures are mitigated; this is the *planning* phase and is based on Z3 [8], which is an open source Satisfiability Modulo Theories (SMT) solver. Once reconfiguration actions are computed, the *Deployment Infrastructure* is responsible for taking those actions to reconfigure the system; this is the *acting* phase. Since failure detection and diagnosis have been extensively studied in existing literature, our focus in this paper is strictly on the second (planning) and third (acting) phases of the self-reconfiguration loop; this is presented in Section 6. We use capabilities supported out-of-the-box by ZooKeeper [14] to implement a monitoring infrastructure based on a group membership mechanism (see Setion 7.1.2).

Operations management is required to handle anticipated changes (*i.e.*, planned update or evolution). These changes include both hardware and software changes carried out at runtime. Addition of new nodes and removal of existing nodes are example of hardware changes. Similarly, addition of new applications, removal of existing applications, and modification of existing applications are example of software changes. While failure management is triggered by detection of failures, operations management can be triggered for various reason. A software related change is always instigated from changes made to the appropriate design-time system model. So, for a software related change there is no detection mechanism; the trigger has to be human/manual invocation of the management engine. However, in the case of a hardware related change, since hardware nodes are not modeled explicitly as part of a design-time system model, some external entity is required to detect these changes and invoke the management engine. This detection is also done by the group membership mechanism presented in Section 7.1.2.

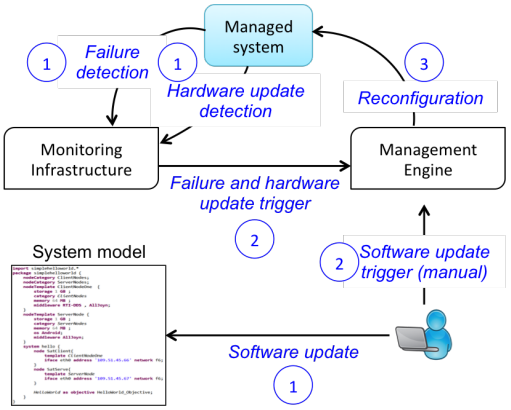


Figure 5: Reconfiguration triggers associated with failure management and operations management.

To summarize detection and reconfiguration trigger mechanisms associated with failure management and operations management, we present Figure 5 as an overview. As shown in the figure, reconfiguration for failure management and hardware update (operations management) is triggered by the monitoring infrastructure. Whereas, reconfiguration for software update (operations management) is manually triggered once the system model is updated.

4. THE CHARIOT DESIGN LAYER

This section describes the CHARIOT design layer, which addresses the requirement of a design-time entity to capture system descriptions. CHARIOT's design layer allows

implicit and flexible system description prior to runtime. In general, an IoT system can be described in terms of required components or it could 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, so that different components can be used to satisfy system requirements, depending on their availability.

A key challenge we 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*). The purpose of this mechanism is to enable CHARIOT to manage failures by effciently searching for alternative solutions at runtime. Another challenge we faced was how to devise abstractions that ensure both correctness *and* flexibility so CHARIOT can easily support operations management.

To meet the challenges described above, CHARIOT's design layer allows application developers to model IoT systems using a generic system description mechanism. We implement this mechanism 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. Since IoT applications are generally mission-specific, their goals should be satisfied during a specified amount of time. CHARIOT defines a *goal* as a collection of *objectives*, where each objective is a collection of *functionalities* that can have inter-dependecies.

CHARIOT's design layer concretizes the functionality tree described in Section 3. It currently 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 elaborate CHARIOT's design layer the remainder of this section presents the system description of the smart parking system initially presented in Section 3.1. Figure 6 shows the corresponding functionality tree, which is used below to describe the different entities comprising the system's description using snippets of models built using CHARIOT-ML, which is our design-time modeling environment. For a detailed description of the modeling language itself, see our prior work [27].

4.1 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. As such, in CHARIOT we only model *node categories* and *node templates*. A node category can be defined as a logical concept used to establish groups of nodes; every node that is part of a IoT system belongs to a certain node category.

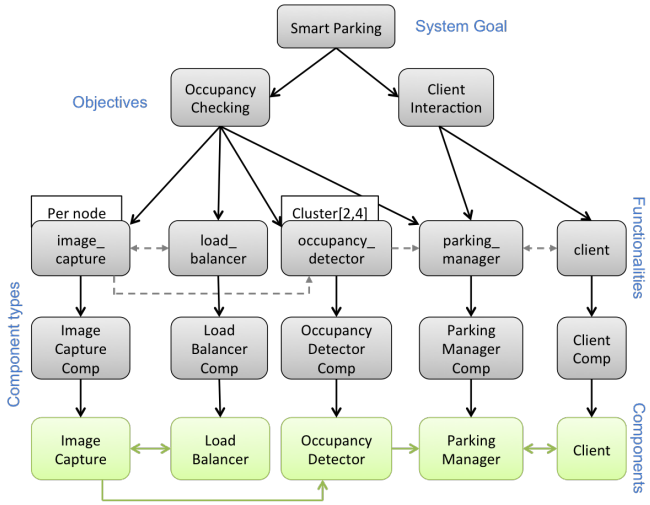


Figure 6: Parking System Description for the example shown in figure 3.

```
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 we do not explicitly model nodes at design-time, we use the concept of node templates to represent the kinds of nodes that can belong to a category. Therefore, a node category is a collection of node templates and a node template is a collection of generic information (specifications such as memory, CPU, devices, etc) that 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. It is important to note that the concept of node categories becomes important when assigning a per-node replication constraint (discussed in Section 4.2), which requires that a functionality be deployed on each node of the given category.

Figure 7 presents the node categories and templates for the smart parking system. As shown in this figure, there

are three categories of nodes: *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 3.1.

4.2 Goal Description

The goal description for the smart parking application is shown in Figure 8 . 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.

```
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 8: Snippet of Smart Parking Goal Description Comprising Objectives and Replication Constraints.

In CHARIOT-ML, objectives are instantiations of compositions (see Section 4.3). The *ClientInteraction* objective is an instantiation of the *client_interaction* composition (line 5) and the *OccupancyChecking* objective is an instantiation of the *occupancy_checking* composition (line 6). A description of how we model these compositions in CHARIOT-ML is presented in Section 4.3. After the objectives are modeled as part of a system, CHARIOT-ML allows the association of those objectives's functionalities with replication constraints. For example, Figure 8 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 (a) voter pattern, (b) consensus pattern, (c) cluster pattern, and (d) per-node pattern, as shown in Figure 9. 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 (a) explicitly stating the number of redundant functionalities required or (b) 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 9 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 this paper. We envision using existing consensus protocols, such as Raft [25], for this purpose.

4.3 Functionalities and Compositions

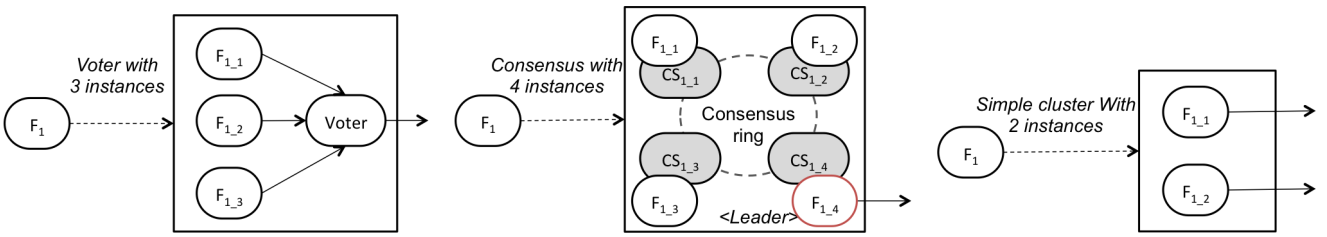
Functionalities in CHARIOT-ML are modeled as entities with one or more input and output ports, whereas compositions are modeled as a collection of functionalities and their inter-dependencies. Figure 10 presents four different functionalities (*parking_manager*, *image_capture*, *load_balancer*, and *occupancy_detector*) and the corresponding composition (*occupancy_checking*) that is associated with the *OccupancyChecking* objective (see line 6 in Figure 8). 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.

4.4 Component Types

CHARIOT-ML does not model component instances, but instead models component types. As discussed earlier in Section 4, 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.

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 com-



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

Figure 9: Example Redundancy Patterns for Functionality F_1. The CS_{n_m} entities represent consensus service providers.

```
1 package edu.vanderbilt.isis.chariot.smartparkingiotpaper {
2     // Parking manager functionality with an input and an
3     // output port to interact with client functionality.
4     // Also, another input port to interact with occupancy
5     // detector.
6     functionality parking_manager {
7         input parking_request, occupancy_status
8         output parking_response
9     }
10    // Image capture functionality with an input and an output
11    // port to interact with load balancer functionality. Also,
12    // another output port to interact with occupancy detector.
13    functionality image_capture {
14        input detector_response
15        output detector_request, image
16    }
17    // Load balancer functionality with an input and an output
18    // port to interact with image capture functionality.
19    functionality load_balancer {
20        input detector_request
21        output detector_response
22    }
23    // Occupancy detector with an input port to interact with
24    // image capture functionality and an output port to
25    // interact with parking manager functionality.
26    functionality occupancy_detector {
27        input image
28        output occupancy_status
29    }
30    // A composition that captures interaction between image
31    // capture, load balancer, occupancy detector, and parking
32    // manager functionalities.
33    composition occupancy_checking {
34        image_capture.detector_request to
35            load_balancer.detector_request
36        load_balancer.detector_response to
37            image_capture.detector_response
38
39        image_capture.image to occupancy_detector.image
40        occupancy_detector.occupancy_status to
41            parking_manager.occupancy_status
42    }
43 }
```

Figure 10: Snippet of Functionalities and Corresponding Composition Declaration.

```
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.

ponent 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 presented in this paper, 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 8). Likewise, the failure of a camera node implies failure of the hosted *ImageCapture* component instance, so this failure cannot be mitigated.

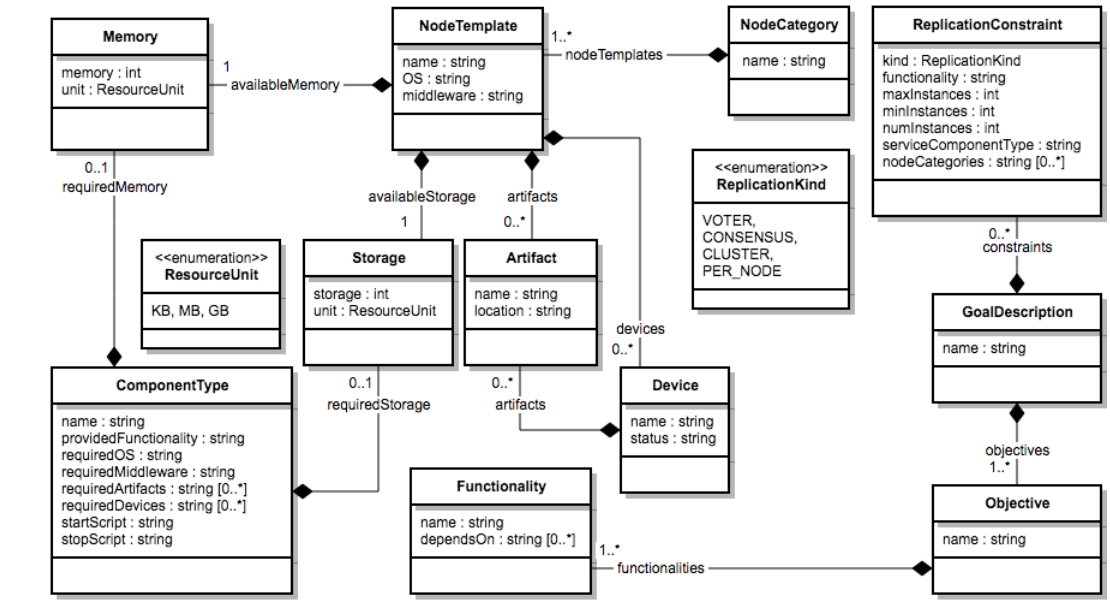
5. THE CHARIOT DATA LAYER

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.

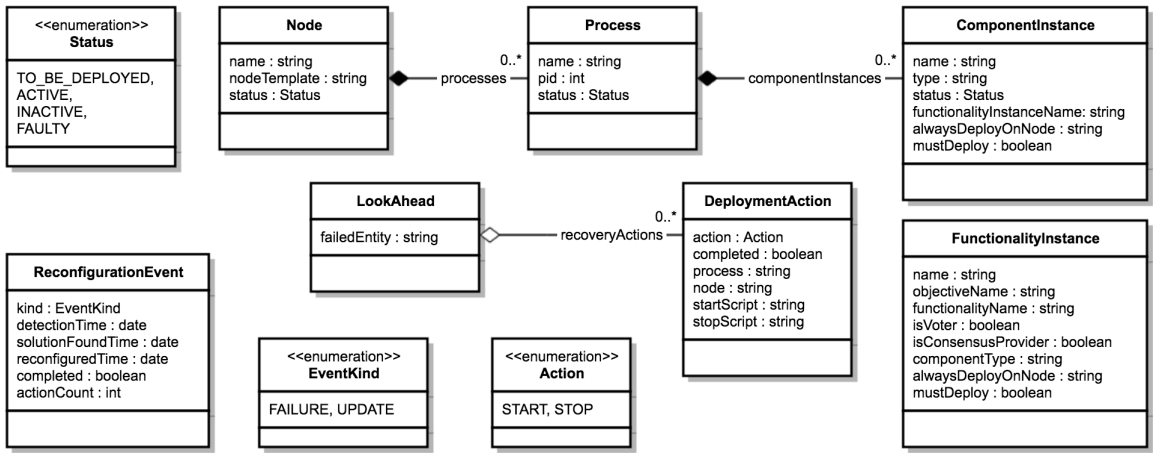
Figure 12 presents UML class diagrams as schemas used to store design-time system description and runtime system information. These schemas are designed for document-oriented databases. An instance of a class that is not a child in a composition relationship therefore represents a root document. Below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

5.1 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. As discussed in Section 4.1, a node category is a concept used to establish logical groups of nodes that form part of an IoT system. It contains a collection of node templates, which provide a generic spec-



(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.

ification of a type of node. The *NodeCategory* class therefore consists of a unique name and a list of node templates. Likewise, the *NodeTemplate* class consists of a unique name and a set of specification attributes, such as available operating system, middleware, memory, storage, software artifacts, and devices.

In addition to node categories, a design-time system description schema also captures component types available for IoT applications. 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.The *ComponentType* class consists of a unique name and a set of other attributes such as (1) name of the functionality provided, (2) required operating system, middleware, memory, storage, software artifacts, and required devices, and (3) scripts that can be used to start and stop an instance of the component type, as shown in Figure 12a.

A design-time system description schema also consists of goal descriptions. As discussed in Section 4, a goal description is application-specific and it describes the goal of an application in terms of objectives and functionalities required to satisfy the goal. The *GoalDescription* class consists of a unique name and a set of objectives and constraints, as shown in Figure 12a. Moreover, objectives are represented by the *Objective* class,w hich consists of a unique name and a set of functionalities.

In addition to objectives and functionalities, a goal description can also contain replication constraints. The *ReplicationConstraint* class represents replication constraints, as shown in Figure 12a. As described in Section 4.2, a replication constraint has a kind, which can either be a voter, consensus, cluster, or per-node. A replication constraint should always be associated with a functionality. The *maxInstances*, *minInstance*, and *numInstances* attributes are related to the degree of replication. The latter attribute is used

if a specific number of replica is 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.

5.2 Runtime System 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. The *FunctionalityInstance* class consists of a unique name, name of the associated functionality and objective, boolean flags to indicate whether a functionality instance corresponds to the voter of a voter replication group (*isVoter* attribute) or a consensus service provider of a consensus replication group (*isConsensusProvider*). The *FunctionalityInstance* class also consists of a *ComponentType* attribute to store exact component type of a functionality instance; this attribute is only relevant for voter and consensus service providing functionality instances as they are not associated with a separate functionality that is part of a goal description. Furthermore, the *FunctionalityInstance* class also consists of an *alwaysDeployOnNode* attribute, which ties a functionality instance to a specific node and is only relevant for functionality instances related to per-node replication groups. Finally, a *mustDeploy* boolean attribute of the *FunctionalityInstance* class indicates whether a functionality instance should always be deployed.

The *Node* class consists of a unique name, associated node template, node status, and a list of hosted processes. For the latter, the *Process* class is used and it consists of a unique name, process ID, status, and a list of hosted component instances. Similarly, the *ComponentInstance* class represents component instances and it consists of a unique name, name of the component type it implements, status, name of the corresponding functionality instance as a component instance is always associated with a functionality instance (see Section 6.2), node name (*alwaysDeployOnNode*) if a component instance needs to be always deployed on that node as part of a per-node replication constraint, and a *mustDeploy* boolean attribute to determine if a component instance must always be deployed.

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 or not, 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 related to CHARIOT's finite-horizon look-ahead strategy described in Section 6.3. It consists of attributes that represents a failed entity, and a set of recovery actions (deployment actions) that must be performed to recover from the failure.

Finally, the *ReconfigurationEvent* class represents runtime reconfiguration events. It is used to keep track of different failure and update events that triggers system reconfiguration. It consists of *detectionTime*, *solutionFoundTime*, and *reconfiguredTime* to keep track of when a failure or update was detection, when a solution was computed, and when the computed solution was deployed. It also consists of a *completed* boolean 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.

6. THE CHARIOT MANAGEMENT LAYER

The CHARIOT management layer comprises a monitoring infrastructure, deployment infrastructure, and a management engine, as shown in Figure 4. The monitoring, deployment, and configuration of distributed applications are well studied, so CHARIOT implements these capabilities using existing technologies, as described in Section 7.1. This section therefore focuses on CHARIOT's management engine, which is a novel contribution that facilitates self-reconfiguration of IoT systems managed via CHARIOT.

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. The remainder of this section describes these concepts and presents CHARIOT's core reconfiguration mechanism and configurable look-ahead strategy.

6.1 Configuration Space and Points

A configuration space includes (1) goal descriptions of different application, (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. 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, which represent valid configurations of all applications that are part of the IoT system represented by the configuration space.

A valid configuration of an IoT system 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.

An IoT system's state is represented by a configuration point in a configuration space, as defined above. System reconfiguration thus entails moving from one configuration point to another in the same configuration space. 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 in Section 3.3.

Given these definitions of configuration space and configuration points, 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 therefore 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.

6.2 Computing the Configuration Point

Given the description of configuration space and point above, a valid reconfiguration mechanism should be based on transitions between configuration points. The *Configuration Point Computation* (CPC) algorithm serves this purpose and thus defines the core of CHARIOT's self-reconfiguration mechanism. Once CHARIOT determines that a system has reached an undesired state (configuration point), the CPC algorithm must compute a new configuration point and a set of actions to transition to the new configuration point to restore normal operation. 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 below.

6.2.1 Instance Computation Phase

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 be 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, we handle each constraint 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 exact replication value or 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 are 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 kinds 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], Raft [25]). For a voter replication constraint, in contrast, CHARIOT generates a single voter functionality 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 4.2).

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 nodes enables failure avoidance.

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 4.4, 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. Section 6.2.2 provides detailed description of how these constraints are encoded.

6.2.2 Constraint Encoding and Optimization Phase

The second phase of the CPC algorithm is responsible for constraint encoding and optimization. The goal is to represent the configuration space and current configuration point using a set of constraints, which allows CHARIOT to use solvers to compute a new configuration point by solving these constraints. The CHARIOT management engine uses *Satisfiability Modulo Theories* (SMT) [4] for constraint encoding and optimization; its underlying solver is Z3 [8]. To present a generic solution, CHARIOT first needs to identify a set of constraints and optimization that are required to model a configuration space and a configuration point, as described 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.

Algorithm 1 Functionality Instances Computation.

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

for *func* in *obj.functionalities* **do**
  if *func* not in *computed_functionalities* **then** ▷ Make sure a functionality is processed only once.
    if *func* has associated replication constraints **then**
      constraints = all replication constraints associated with *func*
      for *c* in *constraints* **do**
        if *c.kind* == *PER_NODE* **then** ▷ Handle per node replication.
          for *node_category* in *c.nodeCategories* **do**
            nodes = nodes in *nodes_list* that are alive and belong to category *node_category*
            for *n* in *nodes* **do**
              create functionality instance and add it to *ret_list*
              add *assign (functionality instance, n)* constraint
        else
          replica_num = 0 ▷ Initial number of replicas, which will be set to max value if range given.
          range_based = *False* ▷ Flag to indicate if a replication constraints is range based.
          if *c.numInstances* ! = 0 **then**
            replica_num = *c.numInstances*
          else
            range_based = *True*
            replica_num = *c.maxInstances*
          for *i* = 0 to *replica_num* **do** ▷ Create replica functionality instances.
            create replica functionality instance and add it to *ret_list*
            if *c.kind* == *CONSENSUS* **then** ▷ Handle consensus replication.
              create consensus service functionality instance and add it to *ret_list*
              add *implies (replica functionality instance, consensus service functionality instance)* constraint
              add *collocate (replica functionality instance, consensus service functionality instance)* constraint
          if *c.kind* == *VOTER* **then** ▷ Handle voter replication.
            create voter functionality instance and add it to *ret_list*
          if *range_based* == *True* **then** ▷ If replication range is given, add atleast constraints.
            add *atleast (c.rangeMinValue, replica functionality instances)* constraint
          add *distribute (replica functionality instances)* constraint
    else
      create functionality instance and add it to *ret_list*
    add *func* to *computed_functionalities*

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. These constraints are generic constraints that apply to IoT systems in different domains, though more constraints can be added for special needs of specific domains. For example, given the availability of period and deadline of all component instances, an IoT system with stringent real-time deadlines might require a specific resource constraint that ensures periodic scheduling of applications, *e.g.*, using Rate Monotonic Scheduling or another real-time scheduling algorithm.

As mentioned in Section 6.1, a configuration point represents a valid deployed of all component instances. A configuration point in CHARIOT is therefore presented using a component-instance-to-node (C2N) matrix, as shown below.

Definition 1 (C2N matrix). *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 (Equation 3).*

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

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

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

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

Now that we have constraints to represent a configuration point (*i.e.*, a valid component-instance-to-node mapping) we need a constraint 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 4.2, which 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 constraint as follows:

Definition 2 (Must deploy assignment). *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 we need 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:

Definition 3 (Dependency Constraint). *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 two matrices: a node-to-node (N2N) matrix and a component-instance-to-component-instance (C2C) matrix. The N2N matrix represents network links between nodes and therefore comprises rows and columns that represent different nodes (Equation 5). Each element of the N2N matrix is either 0 or 1, where 0 means there exists no link and 1 means there a link exists between the corresponding nodes. The C2C matrix is the same except it comprises rows and columns that both represent component instances (Equation 6). The constraint itself is presented in Equation 7.*

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

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

$$C2C = \begin{bmatrix} c2c_{00} & c2c_{01} & c2c_{02} & \dots & c2c_{0\alpha} \\ c2c_{10} & c2c_{11} & c2c_{12} & \dots & c2c_{1\alpha} \\ \dots & \dots & \dots & \dots & \dots \\ c2c_{\alpha 0} & n2n_{\alpha 1} & n2n_{\alpha 2} & \dots & n2n_{\alpha\alpha} \end{bmatrix} =$$

$$c2c_{c_1 c_2} : (c_1, c_2) \in \{0 \dots \alpha\}, \alpha \in \mathbb{Z}^+ \tag{6}$$

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

$$\forall n_1, \forall n_2 : ((c2n_{c_s n_1} \times c2n_{c_d n_2}) \wedge (n_1 \neq n_2)) \implies (n2n_{n_1 n_2} == c2c_{c_s c_d}) \tag{7}$$

The fourth set of constraints CHARIOT needs ensure the validity of resources provided-required relationships, such that essential component instances of one or more applications can be provisioned. CHARIOT encodes these constraints 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:

Definition 4 (Cumulative resource constraint). *The "cumulative resource" constraint is encoded using a provided resource-to-node (CuR2N) matrix and a required resource-to-component-instance (CuR2C) matrix. The CuR2N matrix 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 8). 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 9). Each element of these matrices are integers. The constraint itself (Equation 10) 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.*

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

$$r2n_{rn} : r \in \{0 \dots \gamma\}, n \in \{0 \dots \beta\}, (\gamma, \beta) \in \mathbb{Z}^+ \tag{8}$$

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

$$r2c_{rc} : r \in \{0 \dots \gamma\}, c \in \{0 \dots \alpha\}, (\gamma, \alpha) \in \mathbb{Z}^+ \tag{9}$$

$$\forall r, \forall n : \left(\sum_{c=0}^{\alpha} c2n_{cn} \times r2c_{rc} \right) \leq r2n_{rn} \tag{10}$$

Definition 5 (Comparative resource constraint). *The "comparative resource" constraint is encoded using a provided resource-to-node (CoR2N) matrix and a required resource-to-component-instance (CoR2C) matrix. The CoR2N matrix 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 11). 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 12). Each element of these matrices are either 0 or 1; 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), whereas, 1 means the opposite. The constraint itself (Equation 13) 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.*

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

$$r2n_{rn} : r \in \{0 \dots \phi\}, n \in \{0 \dots \beta\}, (\phi, \beta) \in \mathbb{Z}^+ \tag{11}$$

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

$$r2c_{rc} : r \in \{0 \dots \phi\}, c \in \{0 \dots \alpha\}, (\phi, \alpha) \in \mathbb{Z}^+ \tag{12}$$

$$\forall r, \forall n, \forall c : Assigned(c, n) \implies (r2n_{rn} == r2c_{rc}) \tag{13}$$

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

The fifth set of constraints are needed for management of component instances associated with replication constraints. As mentioned in Section 6.2.1, *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:

Definition 6 (Assign constraint). *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 14.*

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

$$Enabled(c) \implies (c2n_{cn} == 1) \tag{14}$$

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

Definition 7 (Implies constraint). *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.*

Definition 8 (Collocate constraint). *A "collocate" constraint is used to ensure that two collocated component instances are always deployed on the same node. In CHARIOT, as shown in Equation 15, this constraint is encoded by ensuring the assignment of the two component instances is same for all nodes.*

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

$$\begin{aligned} (Enabled(c_1) \wedge Enabled(c_2)) \implies \\ (\forall n : c2n_{c_1 n} == c2n_{c_2 n}) \end{aligned} \tag{15}$$

Definition 9 (Atleast constraint). *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 16. This of course assumes availability of enough resources. A better solution to this would be to use the maximize optimization, as shown in Equation17. 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 18.*

Let $S = \{c_1, c_2 \dots c_{\alpha'}\}$ 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.

$$\sum_{c \in S} \sum_{n=0}^{\beta} c2n_{cn} == max\_value \tag{16}$$

$$maximize(\sum_{c \in S} \sum_{n=0}^{\beta} c2n_{cn}) \tag{17}$$

$$\sum_{c \in S} \sum_{n=0}^{\beta} c2n_{cn} >= min\_value \tag{18}$$

Definition 10 (Distribute constraint). *A "distribute" constraint is used to ensure that a set of components are 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 19.*

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

$$\forall n : \sum_{c \in S} c2n_{cn} \leq 1 \tag{19}$$

The final step (step 8) of the second phase of the CPC algorithm encodes and adds failure constraints. Depending on the kind of failure, there can be different types of failure constraints. We describe how CHARIOT encodes node failures and component failures below.

Finally, the sixth set of constraints handles failure representation. Constraints related to different failures are encoded in CHARIOT as shown below:

Definition 11 (Node failure constraint). *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 20.*

Let n_f be a failed node.

$$\sum_{c=0}^{\alpha} c2n_{cn_f} == 0 \tag{20}$$

Definition 12 (Component failure constraint). *A component can fail for various reasons, so there can be 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 21). If a component keeps failing in multiple different nodes, however, then CHARIOT may need to consider another constraint that ensures the component is not redeployed on any node (Equation 22).*

Let us assume component c_1 failed on node n_1.

$$c2n_{c_1 n_1 == 0} \tag{21}$$

$$\sum_{n=0}^{\beta} c2n_{c_1 n} == 0 \tag{22}$$

6.2.3 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 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 Definition 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:

Definition 13 (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 23; 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. In order to ensure that we obtain least distance configuration point, an ideal solution would be to use minimize optimization (Equation 24) which is supported by SMT solvers like Z3. However, like the maximize optimization, the minimize optimization implementation in Z3 is experimental and does not scale well. As such, in CHARIOT we currently implement this constraint using a recursive logic, which upon every successful solution computation adds the distance constraint (Equation 23) before invoking the solver again to find a solution that is at a lesser distance compared to the previous solution. This recursion stops when no solution can be found, in which case the previous solution is used as the optimum (least distance away) solution.*

$$config\_distance = \sum_{n=0}^{\beta} |c2n\_A_{cn} - c2n\_B_{cn}| \tag{23}$$

$$minimize(config\_distance) \tag{24}$$

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.

6.3 The Look-ahead Reconfiguration Approach

By default, the CPC algorithm presented in Section 6.1 yields a reactive self-reconfiguration approach since the algorithm executes once a failure is detected. Runtime reconfiguration will therefore incur the time taken to compute a new configuration point and determine deployment actions required to transition to a new configuration. This approach might be acceptable for IoT systems consisting of non-real-time applications that can incur considerable downtime. For IoT systems that host 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 presented in Section 6.1 is no different, as shown by experimental results in our prior work [26].

To address this issue, therefore, we 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 Re-Configuration (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. Once a system is initially deployed, we pre-compute solutions to handle failure events. It is important to note that pre-computed solutions cannot be used for update events as update events change the system is such a way that the previously pre-computed solutions are rendered invalid. So, once we have a set of pre-computed solutions, failures are handled by finding the appropriate pre-computed solution, applying the found solution, and pre-computing solutions to handle future failure events. Whereas, for update events, the default CPC algorithm is invoked again (same as during initial deployment) to compute a solution. Once a solution for an update event is computed, we 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
```

In order to pre-compute solutions, CHARIOT currently uses Algorithm 2. Since our work presented in this paper focuses on node failures, this algorithm pre-computes solutions for node failures only. 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). In essence, this pre-computation algorithm 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 5.2 (line 7-12).

6.3.1 Design Discussion and Rationale

The description of the LaRC approach in Section 6.3 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, and the more interesting observation is related to the fact that the solution pre-computation algorithm specifically pre-computes solution 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 systems that are highly dynamic and therefore subject to frequent failures resulting in bursts of failure events. For such systems, it would be important to look-ahead more than one step at a time otherwise we won't be able to handle multiple failures happening in short timespan. However, if we consider systems that are comparatively more static, like the smart parking system presented earlier in this paper (Section 3.1), we expect a higher Mean Time To Failure (MTTF) and therefore do not require to pre-compute solutions by looking ahead more than one step at a time.

Overall, 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 as well as space to store large number of solutions based on various permutation and combination of possible failures, but can handle bursts of failures. Whereas, a single-step pre-computation will be much faster and occupy less space but it will be non-trivial to handle bursts of failures.

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

7. IMPLEMENTATION AND EVALUATION

This section presents a detailed description of CHARIOT's implementation and empirically evaluates its implementation using the smart parking system use-case scenario previously described in Section 3.1.

7.1 CHARIOT Runtime Implementation

This section presents an overview of the CHARIOT runtime implementation and evaluates its performance. Figure 13 depicts CHARIOT's implementation architecture, which consists of a compute node comprising the layered stack described in figure 1.

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 instances of three different types of server nodes: (1) Database

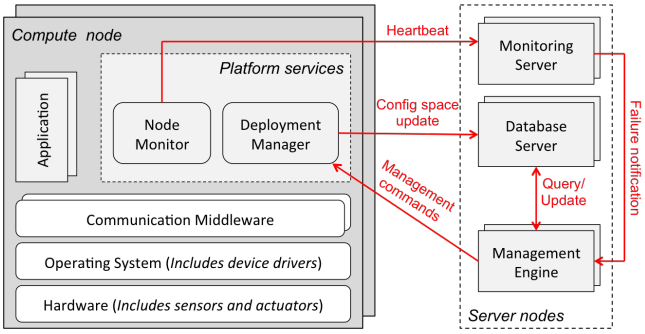


Figure 13: The Implementation Design of the CHARIOT Runtime.

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, which is in turn 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 3.3.

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 3.3.

A Database Server is an instance of a MongoDB server. For the experiments presented in Section 7.2, 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 6.2), with or without the LaRC configuration (see Section 6.3), 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.

[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.

7.1.1 Application Deployment Mechanism

For initial application deployment, CHARIOT ML (see Section 4) 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 6.1) and store it in the Database Server, after which point a Managment 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 compute 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.

7.1.2 Group Membership Mechanism for 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.

7.1.3 Reconfiguration Mechanism

After a failure is detected a Monitoring Server notifies the Management Engine, as shown in Figure 13. 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.

7.2 Experimental Evaluation

Although we have previously used CHARIOT to deploy and manage applications on an embedded system comprising Intel Edison nodes (see http://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. We then describe the application and the 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.

7.2.1 Test-bed

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

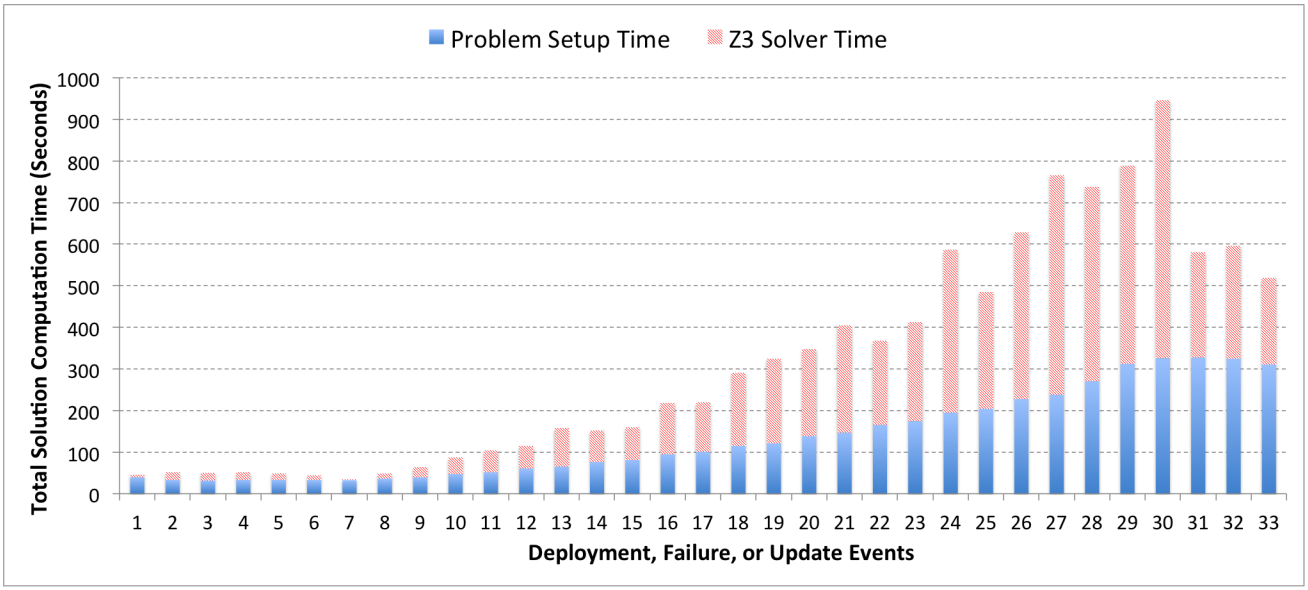


Figure 14: Default CPC Algorithm Performance. (Please refer to Table 1 for details about each event shown in this graph.)

Our test-bed comprises 44 virtual machines (VMs) each with 1GB RAM, 1VCPU and 10GB disk in our private OpenStack cloud. We treat these 44 VMs as embedded compute nodes. In addition to these 44 VMs, three 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 13). All these VMs ran Ubuntu 14.04 and were placed in the same virtual LAN.

7.2.2 Application and Event Sequence

To evaluate CHARIOT, we use the smart parking system described in Section 3.1. We divide the 44 compute nodes into 20 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 8, 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 33 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, 10 processing nodes, and 11 camera nodes. These nodes are added one at a time, resulting in a total of 44 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.

7.2.3 Evaluation of the Default CPC Algorithm

Figure 14 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 6.2.1 and Section 6.2.2), whereas the Z3 solver time corresponds to the third phase of the CPC algorithm (see Section 6.2.3).

Figure 14 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 14), 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 30 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 6.2.3. The amount of iterations (and therefore time) it takes 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.

Finally, events 31 through 33 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.

Table 1: D&C Stages

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 funcitonality decreases from 10 to 9. However since 9 is still within the provided redundancy range (min = 7, max = 10), 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. Furthermore, 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. Therefore, as a result of this failure event, the provided redundancy range associated with the *occupancy_detector* functionality is violated as the number of corresponding component instances decreases to 6. So, this component instance is reconfigured to a different node in order 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 as there are no other terminal nodes and therefore instances of *client* functionality available.
8-30	Hardware updates associated with addition of 2 terminal nodes, 10 processing nodes, and 11 camera nodes. These nodes are added one at a time. 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, addition of a camera node results in deployment of a component instance associated with the *image_capture* functionality. However, addition of a processing node does not result in any new deployment as it is not associated with a per-node replication constraint.
31	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.
32	Failure of another processing node, which hosts no applications. Therefore, no reconfiguration is required.
33	Failure of a camera node. Again, no reconfiguration is required (see event 2 above).

7.2.4 *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 tradeoff 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 tradeoff between the two versions of the CPC algorithm, we present the time taken for solution pre-computation and space required to store pre-computed solution in Figure 15. As shown in the 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 as we can clearly see, 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.

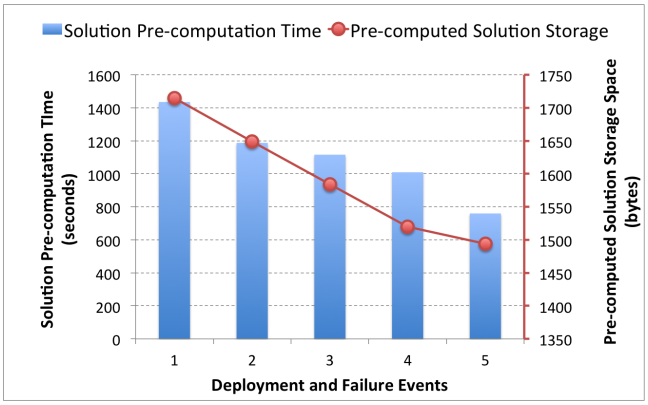


Figure 15: Solution Pre-computation Time for CPC with LaRC. (The solution for failure event i+1 is computed when the reconfiguration action for the failure event i is being applied.)

7.2.5 *Resource Consumption*

To demonstrate the usability of CHARIOT in IoT systems, we present various resource consumption of CHARIOT entities (Deployment Manager and Node Monitor, see Figure 13) that run on each compute node. The resource consumption number 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).

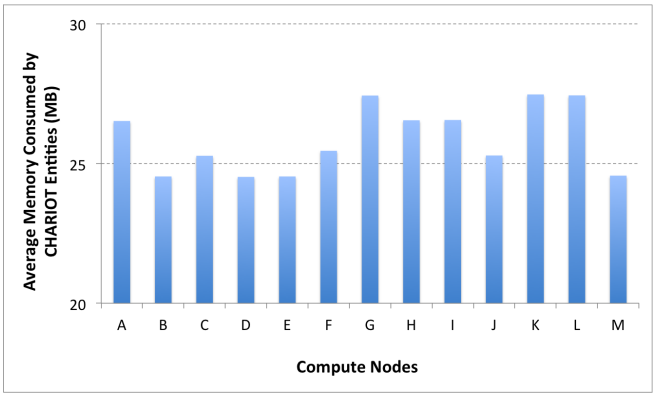


Figure 16: Average Memory Consumption.

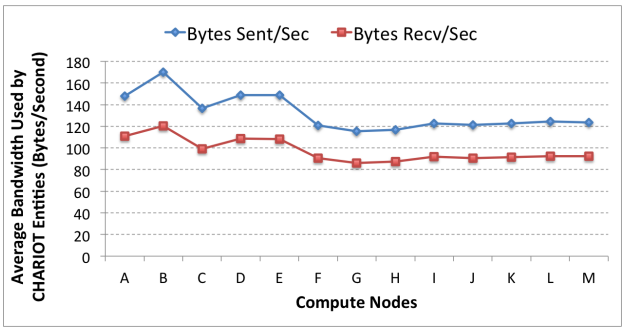


Figure 17: Average Network Bandwidth Consumption.

Figure 16 presents the average memory consumed by CHARIOT entities running on 13 nodes above mentioned 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 17 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 mostly 0%, sometimes rising to less than 0.5%.

From above results presented above we conclude 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 most of our code to C++ to further improve CHARIOT's performance.

8. CONCLUDING REMARKS

This paper described the structure and functionality of CHARIOT, which is an orchestration middleware designed to meet the resilience requirements of IoT systems. The following is a summary of our lessons learned from developing CHARIOT and applying it in the context of a smart parking system case study:

- **Lesson 1: Design-time system description should be generic.** If the objectives of an application and the different functionality 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 association. It is important, however, to extend this concept to support the idea of graceful degradation. As part of future work, we are modeling quality of service functions that provide mechanisms for evaluating the performance of a component's functionality based on available resources. This mechanism can help in cases where we need 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 real-time IoT systems. For those types of systems, it is important to include redundancy in the deployment logic. The CHARIOT modeling language and reconfiguration logic provides support for such redundancy concepts.
- **Lesson 4: Failure reconfiguration approach can be 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,

[6] https://github.com/dcpssc/chariot

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.

9. REFERENCES

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

[2] 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.

[3] 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.

[4] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

[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.