Command and Control: A Scalable System for IoT Device Management

Nishant Jain

Under the Direction of
Yang Richard Yang and Mahesh Balakrishnan
Department of Computer Science
Yale University

CPSC 490: Special Projects
May 4, 2017

Yale University
Contents

1    Abstract  2

2    Introduction  3

3    Design  4

3.1    System Architecture:  4

3.1.1    End-Devices  5

3.1.2    Controllers  6

3.1.3    Metaserver  6

4    Performance Evaluation  7

4.1    System Deployment and Produced Deliverables  7

4.2    Workload Partitioning Runtime Experiment  8

5    Discussion  9

6    Future Work  11

7    Acknowledgements  12

8    References  13
1 Abstract

Command and Control is a novel programming model and system architecture for creating scalable, event-driven Internet of Things (IoT) applications that center around the abstraction of device groups. This project was a joint-collaboration between myself and Shreyas Tirumala (TC 2018) and contains two components: a programming model and accompanying high-level language described in another paper and a system architecture described here. In this manuscript, I present a scalable system architecture inspired by the Software Defined Networking (SDN) paradigm.

The fundamental concept of the Command and Control system architecture is that groups of IoT devices are assigned to physical controller nodes that can facilitate group network interactions. Each device group is only assigned to one controller node in the system and all operations are expressed as interactions between groups of devices. This helps address the challenges of network dynamism and heterogeneous IoT device coordination, enabling robust IoT applications to be deployed on the system. At the highest level, a logical entity known as a metaserver coordinates the assignment of devices to controllers using static program analysis algorithms to balance workloads. The metaserver manages the pool of controller nodes and is able to perform other logistical tasks such as device health monitoring.

Our evaluation of the system indicates that the metaserver is able to perform dynamic rebalancing of workloads upon network topology changes. The Command and Control system therefore contains the components necessary to design, program, and deploy robust, distributed IoT applications while abstracting away the complexity of bilateral device interactions. For a complete view of the capabilities of the system, please read the joint report prepared by Shreyas and me that describes the full programming model, the Borg programming language, our static program analysis algorithms, and the system architecture.
2 Introduction

The Internet of Things (IoT) refers to the network of Internet-enabled everyday devices that can transmit and receive data via embedded computing systems [1, 2, 3]. IoT applications have the potential to revolutionize fields as diverse as home automation, infrastructure management, and medical monitoring [1, 2, 3]. Indeed, given the promise that IoT has shown, many modern technology companies are attempting to capture market share in these areas, with several prominent examples including Amazon’s Echo and Google’s Nest for smart home automation. In practice, many IoT applications involve disparate devices engaged in activities across dynamically changing environments. These devices have to be able to both gather data from and engage with their surroundings while performing tasks that necessitate complex interactions between sets of devices.

Due to the expected growth in the number of deployed IoT devices over the next few years [4], there is a need for scalable infrastructure that can manage a cloud of interacting IoT devices. In response to this challenge, we have developed a high-level programming model and system architecture for controlling clusters of IoT devices. While the programming model centers around an event-driven programming language known as Borg, this paper focuses on an exposition of the system architecture and its properties for managing groups of IoT devices.

Because of the distributed nature of IoT applications and the heterogeneity of device interfaces, programming coordination between IoT devices can prove to be difficult. Scripting behavior at the individual device level involves peer-to-peer interactions that can be further complicated by inconsistencies in device state as in the case of time-based events. Providing reliable peer-to-peer network connection capabilities between devices in a dynamically changing network is also problematic for maintenance. Furthermore, individual devices may not have complete knowledge of the device landscape, which makes expressing global behavior a challenge [5]. To circumvent the challenges that accompany device management, we propose the following solution: a system centered around device groups that abstracts away
the complexity of programming individual devices. Device groups can be broadly divided into one of two major categories: sensors, that gather data from the environment, and actuators, that affect their surroundings. Our system simplifies managing event chains and dependencies between such groups by only allowing group-on-group interactions facilitated by centralized controller nodes.

Through the development of a group-centric distributed system architecture, we aim to address the challenge of simplifying the expression and deployment of scalable event-driven IoT applications.

3 Design

Overview:

The core of this paper is a scalable, networked controller system that can register, transmit to, and receive data from IoT devices. Together with the programming model and the Borg language, these components create a complete system for commanding and controlling a cloud of IoT devices at scale.

3.1 System Architecture:

At a high level, our system is divided into three sets of interacting components. The first of these categories, which we call *end-devices*, refers to individual IoT sensors and actuators. A second category, known as *controllers*, manages the execution of programming logic on groups of end-devices. Finally, a coordinating master known as a *metaserver* handles tasks such as distribution of labor among controller nodes and end-device registration. This is further expanded upon below.
3.1.1 End-Devices

Fundamentally, IoT devices can be roughly partitioned into two sets of functionalities: those that affect their environment, known as actuators, and those that gather data from the environment, known as sensors. Some devices offer both capabilities. Examples of sensors include thermometers, wind speed monitors, and humidity detectors. Examples of actuators include smart light bulbs and sprinklers. In the context of the Borg language paradigm for end-device management that is meant to be used with this system, reduction functions are meant to act upon the data received from sensors and action functions are meant to be called on actuators.

End-devices can consist of sensors and actuators that can be logically organized into non-overlapping device groups. For the sake of maintaining consistency of network state in the face of end-device or node failures, end-devices are unaware of the presence of other
end-devices and strictly interface with the controller node that they are assigned to by the metaserver. The behavior of end-devices is abstracted into sets of function calls either to provide information in the case of sensors or to perform tasks in the case of actuators. Blackboxing the functionalities of end-devices enables controllers to have complete agency over individual end-devices in device groups, thereby facilitating the implementation of inter-group interactions. The design choice to blackbox individual devices is important because it helps with managing end-device heterogeneity and keeping the system state consistent in the case of device failures.

3.1.2 Controllers

Drawing inspiration from the idea of a centralized controller in the field of Software Defined Networking (SDN), we decided to implement a pool of controller nodes as part of our system architecture. Controller nodes are physical servers that manage end-devices. By sharding the space of end-devices among controller nodes, we are able to create applications that scale well to large numbers of end-devices. As mentioned previously, end-devices are unaware of the existence of other end-devices. All group-based dynamics are implemented at the controller level. Each controller node is assigned a set of groups of end-devices by the metaserver once the high-level program has been compiled and the division of labor is determined via static program analysis. Controller nodes thereby serve as a unit of distribution for application scalability. Upon being assigned a set of groups of end-devices by the metaserver, the controller implements group interactions specified by the application programmer in the high level programming language.

3.1.3 Metaserver

At the highest level of the system architecture that we have defined is a logical abstraction known as a metaserver. The metaserver serves as a master coordinator for controller nodes
and end-devices that are present in the system. Upon system genesis, the metaserver listens for registration of end-devices and controllers. Registration entails obtaining identifying data for end-devices, such as IP addresses and port numbers, for example. Our proof-of-concept system also uses the registration process as a chance to register end-device APIs as well by set up remote procedure calls for every available function. Once a pool of controllers has been registered with the metaserver and end-devices have also been identified by the metaserver, the application programmer is asked to provide a high-level program in Borg to the metaserver. Through static program analysis algorithms, the high-level program is partitioned into logical units that can be executed on different controller nodes, thereby sharding the workload. Controllers are assigned end-device groups specified in the high-level program that share event-based dependencies.

The metaserver itself can also perform other logistical tasks like device health monitoring for downed controller nodes and re-partitioning of workloads among the pool of controller nodes as needed. While in our sample implementation, the metaserver is a single machine, a real world deployment could involve a cluster of metaserver nodes that coordinate through a consensus protocol such as Paxos or Raft [6, 7, 8]. The metaserver should store network and device topology state durably in a replicated database to address the potential of metaserver machine failure. This would allow the system to recover from crashes of master metaserver nodes, addressing the concern of the metaserver being a single point of failure.

4 Performance Evaluation

4.1 System Deployment and Produced Deliverables

The process of setting up the system for deployment in our sample implementation of Command and Control involves the following procedure. First, the metaserver establishes an HTTP server to begin a phase of listening for controller and end-device registrations.
Each controller node then transmits its identifier and RPC server IP address/port to the metaserver for storage. This results in a pool of controller nodes that the metaserver can employ for the division of labor. During this registration phase, the metaserver also listens for end-devices to transmit their identifying information and records this. After registration is completed, the metaserver prompts the application programmer for the high-level program written in Borg. Static program analysis via the Naive Load Balancing algorithm is applied to a Borg program in order to partition the workload of device groups between the controller nodes. The Borg compiler then translates the high-level program into Python scripts that can be deployed on each of the controller nodes to execute the logic of the IoT application. We have created a sample implementation of all parts of the preceding workflow in Python, including code for the metaserver, controllers, end-devices, the Borg Compiler, and the Naive Load Balancing static program analysis algorithm. All basic language features have been implemented in the sample compiler, save for the timer object.

4.2 Workload Partitioning Runtime Experiment

As part of an initial experiment into the capabilities of the system, we investigated the ability of the metaserver to partition the workload. Our first experiment focused on the runtime of a load balancing algorithm. Our test metaserver hardware was a single device with a 2.8 GHz Intel Core i7 processor and 16GB of RAM. As part of the analysis, we used the cProfile utility within Python to measure the time taken by the metaserver to partition successively larger high-level Borg programs. We tested 10 different Borg programs that ranged from trivially small (15 event dependencies), all the way to massive in scale (1,920 event dependencies). The runtime measured by the cProfile tool is presented in Table 1. These times represent how quickly the metaserver could partition the Borg applications between controller nodes.

The runtime scales roughly quadratically with respect to the number of when-blocks
<table>
<thead>
<tr>
<th># When-Blocks</th>
<th>Runtime</th>
<th># Function Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>0.003</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>0.004</td>
</tr>
<tr>
<td>3</td>
<td>60</td>
<td>0.010</td>
</tr>
<tr>
<td>4</td>
<td>120</td>
<td>0.019</td>
</tr>
<tr>
<td>5</td>
<td>240</td>
<td>0.042</td>
</tr>
<tr>
<td>6</td>
<td>480</td>
<td>0.085</td>
</tr>
<tr>
<td>7</td>
<td>960</td>
<td>0.223</td>
</tr>
<tr>
<td>8</td>
<td>1920</td>
<td>0.693</td>
</tr>
</tbody>
</table>

Table 1: Runtime Performance Analysis of Naive Load Balancing Algorithm on the Metaserver.

(Figure 2). This matches expectations given that setting up the dependency graph and running the Naive Load Balancing algorithm together have a time complexity of $O(n^2) + O(n) = O(n^2)$. These algorithms are described in the paper detailing the programming model and Borg language. A key insight from the data is that even with vast Borg programs that have many event dependencies, our implementation of the Borg compiler on the metaserver is able to swiftly partition workloads and distribute them among controller nodes. This indicates that in real time, the metaserver could easily rebalance workloads on the system as topologies change, such as in the case of controller node failure or new nodes joining the system. This is an important property that will be useful in future versions of the Command and Control architecture that implement dynamic workload rebalancing.

5 Discussion

This project aims to solve several major challenges in the field of distributed IoT application programming by providing a simple programming model and robust system architecture for developing distributed event-driven applications centered around groups of end-devices. The Command and Control system can be divided into two components:

- the Borg Language which provides an intuitive schema for encoding logical and temporal dependencies between events in an IoT application, as well as support for static
Figure 2: The runtime of the load balancing algorithm and compiler for distributed applications in Borg scales roughly quadratically with program complexity as measured by number of when-blocks.

- An accompanying scalable, networked controller and device system which facilitates the management of device groups in a distributed IoT application

The Command and Control system presents several constructs that make it suitable for IoT application programming and deployment. At the heart of the system is the abstraction of a device group. We believe that powerful IoT applications can be intuitively expressed as interactions between groups of devices, thereby circumventing the challenges of individual device management. The group abstraction also addresses the large problem of IoT device heterogeneity by blackboxing individual end-devices into sets of APIs that can be called from the controller nodes.

The system architecture also presents several advantages that make it suitable for distributed IoT applications. Fault tolerance is built into several levels of the infrastructure. At the controller level, nodes can simply be reassigned workloads by the metaserver upon controller node failure. Metaserver failure can also be accounted for by durably storing network
topology in a replicated database. As the metaserver primarily handles system initialization and health checks, a new metaserver can easily be substituted if there is metaserver machine failure, as long as the network topology is available. Additionally, since each end-device only communicates with a controller that it is assigned to, there is no need for every end-device, or controller node, to maintain an accurate view of the entire system. This design choice will aid in situations where new controllers or end-devices are added to the deployment. Keeping the full system state updated across all devices would be a computationally difficult task, especially since many IoT devices do not possess much storage capacity. In our design, only the metaserver needs to know the topology of the system for initial assignment and reassignment of workloads upon topology changes.

With the additional improvements suggested in our Future Work section, Command and Control should present a robust infrastructure model for scalable IoT applications.

6 Future Work

In this section, we suggest further modifications for Command and Control to improve its ability to serve as a useful system for creating distributed IoT applications.

In terms of implementation at the system level, future versions of the Command and Control infrastructure should implement a distributed metaserver and durable datastore for storing network topology state in the metaserver. This would aid greatly with the limitation of the metaserver being a single point of failure. Furthermore, implementing dynamic rebalancing of workloads seems like a logical next step given the performance data collected in the preceding experiment.

We also recognize the need for further evaluation of the system. hope to run analyses on how system performance scales with large numbers of end-devices interacting across groups.
7 Acknowledgements

Working on Command and Control with Shreyas Tirumala, my roommate and friend, has been an incredible pleasure. Words cannot describe the experiences we have shared, from freshman year all the way through the end of our senior thesis project. Thank you for being my partner through it all.

Furthermore, I would like to extend my sincerest gratitude to my advisors, Professors Yang Richard Yang and Mahesh Balakrishnan, for guiding us through the process of developing this system and providing lots of helpful feedback along the way. I would also to thank my parents, Jayant and Usha Jain, and the rest of my family, for all of their support in raising me and teaching me the value of love and hard work. Finally, special thanks to Professor James Aspnes, the Director of Undergraduate Studies in the Yale Computer Science Department, for allowing us to collaborate on this joint thesis project.
References


