Command and Control: A Programming Model for IoT Device Management

Shreyas Tirumala

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 2

3 Design 4

3.1 The Borg Programming Language 4

3.1.1 Device Groups 4

3.1.2 Reductions and Actions 5

3.1.3 When-Blocks 5

3.1.4 Events and Conditions 6

3.1.5 Variables 6

3.2 Program Structure and Execution Model 6

3.3 Use Case Analysis 8

3.3.1 Case 1: Controlling An Air Conditioning Unit 9

3.3.2 Case 2: Managing an Entertainment System 10

3.4 Language Completeness 11

3.5 Basic System Architecture 12

3.6 Static Program Analysis 12

4 Deliverables 20

5 Discussion 21

6 Future Work 22

7 Acknowledgements 23

8 References 24

A Backus-Naur Definition of Borg 25
1 Abstract

This project was a joint effort with Nishant Jain (TC ’18). This paper was adapted from our joint work, Command and Control: A Scalable System and Programming Model for IoT Device Management. Please refer to the full paper for a complete description of our project. We developed a system known as Command and Control, which encompasses both a programming model and a system architecture for creating scalable, event-driven Internet of Things (IoT) applications. In this paper, we will focus primarily on the programming model, which was my area of focus. Our model revolves around the concept of a device group. We designed a programming language called Borg to allow developers to encode temporal and logical event dependencies between device groups in an efficient and scalable manner. The paper goes into detail about two real-world use cases that are made easier by using the language and illustrate key features as well. Through static program analysis algorithms I go on to describe, I show that Borg programs can be intelligently analyzed to enable partitioning of IoT applications quickly and systematically. This is because Borg programs can be represented via undirected graphs. Vertices in these graphs are made up of the fundamental organizational units of the language, constructs called when-blocks. Edges reflect various types of dependencies between these blocks. Partitioning is thus a matter of parsing these dependency graphs. A formal Backus-Naur Form definition of the language is provided in the Appendix. When combined with the network architecture, Borg, and more generally, the Command and Control framework, provide developers with useful tools to program and deploy software across a multitude of IoT devices simultaneously.

2 Introduction

The Internet of Things (IoT) refers to the interconnected web of Internet-enabled, or smart, devices that use embedded systems to send and receive data. [1] [2] [3]. IoT applications
are set to change the way we think about our homes, our communities, and our workplaces. With more and more smart devices appearing on the market everyday, from the Amazon Echo to the Google Nest, there is a pressing need for scalable infrastructure that can manage many devices at once. It is in response to this challenge that our system, Command and Control, was born.

Unfortunately, there isn’t a clear definition of what an IoT application is, and due to the huge variance in features between devices, coordinating IoT devices is incredibly complex. Programming devices one-at-a-time can be painful enough, but navigating bilateral device interactions is made especially complex by inconsistent formats for maintaining device states. Individual devices also usually lack the resources to have complete knowledge of what other machines are on any given local network, making programming even simple interactions a project in and of itself. To address the complexities of programming individual devices, we created a programming model centered around device groups. Such groups can also be sub-divided into two primary categories: sensor groups, which get data from the environment, and actuator groups, that conduct actions directly. Many devices and device groups fall into both categories. Our programming model is actualized by a programming language called Borg, which we have designed to simplify the process of linking together events and other dependencies between device groups.

One of the major advantages of our model is that we can support dependencies in a logical basis, as well as a temporal basis. Many IoT applications depend on describing actions based on both events that occur in the environment as well as time, but most programming languages focus on encoding only the logical conditions. In the sections to come, I’ll describe the design of Borg in more detail and explain how it can be used to simplify IoT programming.
3 Design

Overview:

The project in its entirety consists of two main components: a system architecture and a programming model for managing IoT devices. To actualize the programming model, we developed a programming language called Borg, which will be discussed in detail in this paper. For more details about the system architecture component, please refer to our joint work.

3.1 The Borg Programming Language

To implement the aforementioned abstractions, we have developed a proof-of-concept programming language called Borg. This section will describe the high-level design of this language. To see a formal Backus-Naur Form definition of the language syntax, please refer to the section Backus-Naur Definition of Borg in Appendix A.

3.1.1 Device Groups

To implement the abstraction of a device group, Borg defines a variable type known as $DG$. The initial specification of this language requires devices to be specified by IP address and port number, however, it is trivial to expand this specification to encompass any type of identifier, such as Bluetooth ID, MAC Address, etc. Currently, the language specification mandates the explicit definition of devices, but the eventual goal of Borg is to allow the implicit definition of dynamic groups based on shared properties. For instance, a device group could theoretically be described as the group of devices that implements a `getTemperature()` function).
3.1.2 Reductions and Actions

Drawing inspiration from programming models such as MapReduce and Apache Spark, all device groups provide support for the following two functions: `reduce()` and `action()` [5, 6]. The former is primarily intended to support devices with sensory capabilities – that is, devices that can detect environmental inputs and return a float, integer, or string reflecting such inputs. The `reduce()` function takes in two required parameters: the function to be called on each end-device, and a reduction/aggregation function to be applied on all the data gathered from the group. For example, `groupname.reduce("getTemperature", "mean")` would return the mean of the temperature values read from every end-device in the group `groupname`. On the other hand, the `action()` function takes one mandatory parameter: a function to be called on every device in a given group. Therefore, `groupname.action("powerOn")` would call a function called `powerOn` on every end-device in the device group `groupname`. Our proof-of-concept compiler also implements optional parameters for the `action()` function that can be provided to augment the function call.

3.1.3 When-Blocks

The other fundamental construct of Borg is the `when-block`. Many applications of IoT technologies require the user to persistently wait for specific conditions to be met. Such conditions can be temporal (such as a specific time or day of the week) or based on sensor input (such as when a sensor’s values fall within a certain range). A when-block generalizes both temporal and input-based conditions into a single semantic construct. As we will go on to show in the Discussion section, such divisions allow Borg to not only provide a natural-language-like environment for application developers to program in, but also allow a compiler to intelligently partition execution of code across multiple controller nodes. This is further described in the Static Program Analysis section.
3.1.4 Events and Conditions

Multiple when-blocks on the same controller node can be thought of as a series of conditional checks. Notably, our language guarantees sequential execution of code that depends on the same end-devices/device groups. That is, if two when-blocks depend on the same device group, the first one written will execute before the second one. This when-block structure also allows additional optimizations to occur in the compiler to efficiently load-balance code execution across controller nodes. The conditions in when-blocks can be sets of reductions, timer events, or user-defined events. Reduction conditions involve comparing group reductions to numerical or string values. Timer conditions are true when programmer-specified time-based conditions occur. For example, a programmer can specify a timer event that fires every day at 7:00 PM. The third and final type of condition is the user-defined event. These are defined by creating event variables. The programmer must manually raise such events by executing the raise eventname command. Once a user-defined event has been raised, any when-block employing the event as a condition will continue to execute until the program issues the end eventname command. As we discuss in the Use Case Analysis section, user-defined events allow for events that depend on others that may have occurred in the distant past.

3.1.5 Variables

To assist programmers in specifying conditions and assignments, Borg also supports two additional types of variables: numerics and strings. Numerics encompass both integers and floats. Strings are text-based variables. There is no concept of scope in Borg. All variables are considered global variables.

3.2 Program Structure and Execution Model

A typical Borg program can be roughly visualized through the following code sample:
// Declarations stylistically should be at the top of the program
DG group1 = {...}
DG group2 = {...}
string sample_str = "str1"
number x = 1

// Statements that always execute should occur here
// Statements outside of when-blocks should be very rare, however
x = 4
...

// A series of when-blocks follows
when (condition 1 and/or condition 2)
{
    group1.action(...) // Statements are typically in when-blocks
    sample_str = "str2"
}
when (condition 3)
{
}
...

The fundamental unit of work in Borg is a statement. Statements can consist of device
group actions, raising or ending events, declarations, assignments and whens (declarations of
when-blocks). Execution of the program proceeds sequentially as written by the programmer,
looping through sets of conditions as necessary.

On an organizational level, statements are typically executed within the body of a
when-block. Although statements can technically execute outside of when-blocks, this is
discouraged and should be done sparingly. As we will show in the Static Program Analysis
section, this when-block centric organization is what allows Borg to be a powerful tool for
managing distributed systems.

Although real-world deployments of Borg code likely require multithreading and complex
hardware configurations, a developer can visualize a Borg program’s execution model as
follows. Consider a hypothetical single-threaded application with a FIFO queue of events,
in which each event corresponds to a when-block condition.

Initially, the event queue is empty as no events have been encountered in the environment
space of the system. The application’s execution thread can be thought of as sleeping until events populate the event queue. As events occur, they are serialized into the queue and processed in order of arrival. Events are blocking and must be processed before the program can continue. Though events have the ability to raise other events, a serial order of the event queue is maintained. If the queue is nonempty, the event at the top of the queue is popped off and processed, with its corresponding logic executed.

In practice, this sequential event processing can be achieved without a literal FIFO queue being implemented. A simpler translation of this execution model into a traditional C-like language could involve code analogous to the following program:

```c
while (true)
{
    if (when-block 1 condition)
    {
        ... // Actions
    }
    if (when-block 2 condition)
    {
        ... // Actions
    }
}
```

Note that the code above is far from the most efficient actualization of the Borg execution model; for instance, an infinite while loop is a very power-inefficient means of forcing the program to sleep until a when-block condition is met. So long as events (when-block conditions) are processed in a sequential manner akin to a FIFO queue and events are not skipped when they should trigger, many implementations are possible.

### 3.3 Use Case Analysis

In this section, we will review two potential use cases for our system and analyze sample Borg programs that could be written for them.

Please note the following:

- Although Borg does not formally support any comment functionality, we have used
C-style comments in the following programs for clarity

- We use IP Addresses and Ports to specify devices here and in our proof-of-concept compiler, but technically, the language could support any generic identifier

3.3.1 Case 1: Controlling An Air Conditioning Unit

**Problem Description**: A developer wants to manage air-conditioning units in a large building. When the average temperature of the building hits 80 degrees, the developer wants to turn the air-conditioning units on. Let us assume that the building also has a number of identical temperature sensors located in various rooms that support a `getTemperature()` function. We’ll only refer to two sensors and two AC units in our actual code for brevity. In addition, we will assume that the air-conditioning units are identical and all support an `enable()` function that enables them if they are not on already.

**Case 1: Sample Borg Solution**

```c
DG temp_sensors = {"192.168.2.7:8080","192.164.2.9:80"}
DG air_conds = {"127.0.0.1:900","127.0.1.2:266"}

when(temp_sensors.reduce("getTemperature", "mean") > 80)
{
    air_conds.action("enable")
}
```

**Discussion**: This somewhat trivial example showcases the basics of Borg. We defined two sensor groups: one for temperature sensors and the other for AC units. We also showcased the `reduce()` and `action()` functions. If run outside of our system, a developer would have to manually write code to facilitate networking between the temperature sensors, the AC units, and potentially some central computing system. He or she would also have to write multiple for loops to obtain data from each temperature sensor, calculate the mean, and enable every AC unit. After registering these devices with our system, we accomplish the same thing in four lines of code.
3.3.2 Case 2: Managing an Entertainment System

Problem Description: A developer wants to configure her entertainment system at home at 7 PM everyday. However, she does not want the television and sound system to turn on when nobody is home. Let us assume that she has a group of motion sensors in her doorway to detect whether someone has walked into the house. Each motion sensor implements a function \texttt{walkedIn()} that returns 1 if it detects movement towards the house. Most motion sensors do not save state, so we will assume that the motion sensors only return 1 at the time that they detect motion.

---

Case 2: Sample Borg Solution

```c
// Two Motion Sensors
DG motion_sensors = {"192.168.2.7:8080","192.164.2.9:80"}

// TV, Sound System
DG entertainment_sys = {"127.0.0.1:900","127.0.1.2:266"}

event entered_home
timer daily_timer = 1900

// By default, timer events are true daily at a time
// This creates a timer that is true at 7 PM

// Check whether both sensors trigger
when(motion_sensors.reduce("walkedIn", "sum") == 2)
{
    // Potentially other actions here
    raise entered_home
}
when (entered_home and daily_timer)
{
    entertainment_sys.action("on")
    end entered_home
}
```

Discussion: This program highlights several other important features of Borg. In this case, the entertainment system device group is made up of heterogeneous devices. The action function in Borg does not differentiate between types of devices; it merely checks for
whether the parameter function, in this case the "on()" function, has been implemented. This program also showcases timer events. Most importantly, however, it highlights the power of user-defined events. The entered_home event in this case is necessary because we want to ensure that both motion sensors *always* trigger before entering the second when-block. User-defined events allow developers to enforce strict temporal orderings of code execution. In other words, events can act as prerequisites for other events—even when one occurs long before another. The Borg user-defined event system therefore provides granular control over program state.

### 3.4 Language Completeness

Though further research is necessary to determine whether Borg is truly Turing Complete, we argue that there is a strong likelihood that it is. A formal proof of Turing completeness can be shown by demonstrating that the $\mu$-recursive functions can be expressed in our language, or by demonstrating its ability to replicate the ability of a Turing machine, lambda calculus, or other Turing complete system [7, 8]. However, researchers, such as Scott Schneider of IBM Research, postulate that the minimal requirements for Turing completeness are that a language can support arbitrarily large amounts of storage, support conditionals, and allow for both bounded and unbounded loops [9]. Borg can indeed support arbitrary numbers of variables and thus can theoretically store massive amounts of state data. A when-block intrinsically mirrors features of both while loops and conditionals. By setting variable values within a when-block as follows, we can emulate conditionals more directly.

```
Replicating a conditional in Borg

```

```
number switch = 1
when(switch != 0 and arbitrary-condition)
{
    // Execute action
    switch = 0 // Ensures this block will only run once (max)
}
```

By incrementing or decrementing variable values instead of merely setting them as we
did above, Borg can also emulate bounded loops. Any when-block using conditions based on device groups is inherently unbounded because when-blocks check for their conditions ad-infinitum—similar to a while(true) construct in other languages. Notably, these are the same conditions used to prove the Turing Completeness of cognitive scientist Douglas Hofstadter’s Floop programming language introduced in *Gödel, Escher, Bach* (1979) [10]. For these reasons, we strongly suspect that Borg is Turing Complete, but more work is needed to confirm this hypothesis.

### 3.5 Basic System Architecture

Our network system is made up of three main parts: *end-devices*, which are individual IoT sensors and actuators; *controllers*, which are physical servers to manage end-devices; and a *metaserver* which distributes labor between the controllers and registers end-devices. Developers write their programs on the metaserver and the system handles the rest. For more information, please consult our joint work: *Command and Control: A Scalable System and Programming Model for IoT Device Management*.

### 3.6 Static Program Analysis

As mentioned previously, one of the big advantages of Borg is that its design lends itself naturally to partitioning applications. Programs are written on a metaserver, which then generates code for individual controller nodes to use to logically manage end-devices. To load-balance the division of device groups and logical dependencies across these controllers, the metaserver conducts static program analysis. In this section, we provide a few potential methods for doing so.

Before we begin our analysis, note that the most basic execution unit for our program is the when-block. A complete program can be specified by just one when-block. Though technically, programs can exist without using a single when-block, such degenerate cases are trivial to load balance and will not be discussed here. We will make the following three
simplifying assumptions:

- **Assumption 1**: Every when-block will fully execute on a single controller
- **Assumption 2**: Every when-block demands an equal amount of resources to execute (e.g. CPU cycles, memory, etc.)
- **Assumption 3**: Every end-device is in, at most, one device group

Further research could expand on this work by relaxing these assumptions.

**Dependency Graph**

Consider the following graph $G$ which can be generated by parsing a Borg program. We will use this graph to decide how to partition our code and organize which device groups are managed by which controller nodes.

Let us define graph $G = (V, E)$ where $V$ is a set of vertices and $E$ is a set of edges. We generate $G$ by creating a vertex $v_i$ for every when-block and an undirected edge $e = (v_i, v_{i+1})$ between when-blocks whose conditions are dependent on other when-blocks. Computing these dependencies requires pairwise comparisons of every when-block and therefore, setting up the graph will take $O(n^2)$ where $n$ is the number of when-blocks/vertices. Hereafter, we will refer to assigning vertices or components of the graph to various controller nodes. Because of the one-to-one relationship between vertices and when-blocks, assigning a vertex $v_i$ to controller $C_i$ equivalently means that the when-block corresponding to $v_i$ should execute on $C_i$. Dependencies between when-blocks occur in two primary situations: when the same device group is used in multiple when-blocks (meaning either the condition or the body of the block), or when when-block conditions are based on user-defined events raised in other when-blocks. We will differentiate between these two types of edges by categorizing the former as the set of edges $F$ and the latter as the set of edges $I$. Note that it is possible for an edge to be in both sets simultaneously, and that the graph need not be connected.
### Load Balancing Methods

- **Method 1: Naive Load Balancing**

![Diagram of graph with vertices v1, v2, v3, v4, v5](image)

*Figure 1: In this example, v₁, v₂, and v₃ should be assigned to one controller, and v₄ and v₅ should be assigned to another.*

### Algorithm Specification

In the most naive case, when-blocks corresponding to the vertices in every connected component can be assigned to a different controller as seen in Figure 1. If more components exist than controllers, then the excess components can be split among the controllers such that the fewest overlaps occur. Sample pseudocode for this algorithm is as follows:
Finding connected components can be done in linear time using breadth-first or depth-first search, so this naive strategy runs in $O(n)$ where $n$ is the number of edges and vertices in $G$ summed together. Because edges in the graph represent dependencies between when-blocks, such a load-balancing strategy ensures that individual controller nodes can handle all event dependencies. It has the added benefit of guaranteeing that all end-devices in the system only interact with one controller. These benefits not only make this load-balancing method easy to program, but also allow every controller node to be fully independent from one another. As such, replacing a controller node that fails is simple and no overhead needs to occur with coordinating communication between controllers or between controllers and metaservers. However, such a strategy is not scalable. Programs with many dependencies between when-blocks will not be distributed evenly between controller nodes. For example, if $G$ is connected, then all code will execute on one controller node.

- **Method 2: Group Priority Balancing**

Because we have a metaserver to coordinate communication between controllers, it is
possible to introduce further load balancing by allowing the metaserver to signal to controllers that user-defined events occurred. We call this group priority balancing as we prioritize keeping device groups on a single controller, but allow for metaserver coordination of user-defined events. The algorithm for this type of load balancing is as follows.

**Algorithm Specification**

Please see the following page for the algorithm specification.

```
Data: Graph $G = (V, E)$, Set of Controllers $C$

1. $W = \{e \in E \mid e \in I \setminus F\}$;
2. Remove edges to create Graph $G' = (V, E - W) = (V, E')$;
3. Use BFS/DFS to find the set of all connected components;
4. while number of components $> |C|$ do
   5. Let component $k_1$ be the smallest component;
   6. Let component $k_2$ be the second-smallest component;
   7. Let edge $w = (u, v) \in W$ s.t. $u \in k_1$ and $v \in k_2$;
   8. $E' = E' \cup \{w\}$;
   9. $G' = (V, E')$;
5. end

**Algorithm 2: Group Priority Balancing Algorithm**

First, remove all $e \in E$ such that $e \in I \setminus F$. If the connected components that remain contain roughly the same number of vertices and the number of components is equal to the number of controller nodes, then the algorithm can halt. If more connected components exist than controller nodes available in the system, then re-add edges $e \in I \setminus F$ one at a time, prioritizing edges that connect the smallest components. As we will show later, this greedy prioritization does not guarantee optimality, but seems to often yield sufficient results for many use cases. After each edge addition, check
the number of connected components again. Halt when the number of components is less than or equal to the number of controller nodes.

![Initial State of graph G](image)

*Figure 2: Initial State of graph G*

An example run of this algorithm can be illustrated by the Figures 2-4. Let us assume, for example, that our system has two controller nodes available and that the edges marked in red are edges in $I \setminus F$, that is edges that represent user-event based dependencies, but not device group dependencies between when-blocks. At the beginning of the program, the graph appears as it does in Figure 2.

![Remove all edges e ∈ I \ F](image)

*Figure 3: Remove all edges $e \in I \setminus F$*

The first step of the algorithm is to remove edges that represent only user-event based dependencies. As we can see in Figure 3, the graph now has 3 connected components.
(\(v_5\) is trivially connected). However, we only have two controller nodes available, so we move on to the next step.

![Graph](image)

Figure 4: Add edges back one at a time

We now add back an edge that we removed, prioritizing edges between the smallest connected components. In this case, we add back the edge \((v_5, v_6)\). We now have two connected components and two controllers, so our algorithm halts. Once the algorithm halts, simply assign the vertices of each connected component to a different controller per component.

**Algorithm Discussion**

**Time Complexity**

Given that connected components of a graph can be found in linear time using breadth or depth first search, this algorithm also can be optimized to run in \(O(n^2)\) where \(n\) is the number of vertices and edges in \(G\) summed together. Only one BFS/DFS needs to run to calculate the initial number of connected components, which takes linear time. Thereafter, for every edge added, if the vertices are in different components, then the total number of components decreases by one. This calculation can be done in constant time for each edge added back, and there are at most \(|V|(|V| - 1)\) edges in a graph, meaning the while loop can take, at worst, \(O(n^2)\) operations. Thus, the algorithm in
total runs in, at maximum, quadratic time, although the graph will be fully connected and the algorithm will halt much sooner.

**Optimality**

Notably, improvements still exist for this algorithm. Because the parsing of this algorithm is greedy, it will not necessarily generate an optimal solution—that is, it does not necessarily guarantee a solution that makes the number of vertices in each component as close to equal as possible. For example, consider 3 controllers and a completely connected graph with five components containing 5, 4, 3, 2, and 1 vertex/vertices in them respectively. The optimal solution would be to merge the 4-vertex component and the 1-vertex component and the 2 and 3-vertex components, creating three 5-vertex components. This algorithm would instead merge the 1, 2, and 3-vertex components, creating 4, 5, and 6-vertex components instead. However, cursory analysis seems to indicate that our algorithm is sufficient for many use cases. Despite the algorithm not being the most optimal solution, its ease of implementation makes it a strong candidate for further research.

**Setup Costs and Overhead**

It is also useful to compare this method to the naive method in terms of resource usage and system overhead. Let us define the dependency graph after this algorithm halts as $G' = (V, E')$. By construction, every $e = (v_i, v_{i+1}) \in E \setminus E'$ represents a dependency between when-blocks that is based on user-defined events alone (not device group dependencies). Let us assume when-block $v_{i+1}$ is dependent on an event $ev$ raised in when-block $v_i$. To make this algorithm useful, a metaserver must configure the controller for $v_i$ to send it a signal that $ev$ occurred, and then the metaserver must send a signal to the controller for $v_{i+1}$ raising the event. Therefore, this algorithm adds additional system overhead in comparison to the naive method. However, this algorithm balances workload across controllers more effectively than the naive method.
For example, a fully connected graph $G$ will always be assigned to one controller in the naive method, but may not always be assigned to one controller using this one.

**Static Program Analysis: Discussion**

Both of the methods described above share an important property: every device group, and by our assumptions, every end-device, in the system only interacts with, at most, one controller. This allows our system to avoid issues with conflicting, concurrent requests to a device. However, relaxing our initial assumptions could allow for more granularity in partitioning. For example, loosening Assumption 2, that every when-block requires an equal amount of resources, could be easily accommodated by simply assigning a weight to connected components in $G$ based on the number of end-devices in the component in total. Then, balancing can be done on both the number of components and the number of devices per controller. We expect that future research will involve analyzing the performance of these algorithms and similar application partitioning methods that account for other system attributes, such as device group sizes.

### 4 Deliverables

We created a sample Python implementation of Command and Control, including code for the network system, the Borg Compiler, and the Naive Load Balancing static program analysis algorithm. Nishant primarily worked on the system, and I primarily worked on the compiler and static program analysis. All basic language features have been implemented in the sample compiler, save for the timer object. For performance analyses and other benchmarks, please refer to our joint work.
5 Discussion

As mentioned previously, our project aims to address several issues in distributed IoT development by providing an event-driven programming model. The project as a whole was divided into two parts

- 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 program analysis (discussed in this paper)
- A network system architecture to manage IoT devices in an intelligent manner (refer to our joint work)

The Command and Control system presents the fundamental abstraction of the device group, which allows the developer to sidestep the many logistical challenges associated with programming individual devices. It is upon the basis of this abstraction that the Borg language was born.

The ability of the Borg language to express temporal and logical events through the abstractions of when-blocks and device group reductions, lends itself naturally to IoT applications, which tend to be highly event-driven. Borg was specifically defined with the intent of allowing dependencies between groups of devices to be encoded intuitively. This means that complex event chains in both a logical and temporal sense can be represented by linking syntactic constructs such as when-blocks. Device groups in Borg support two primary operations, an *action()* function for making actuators engage with their environment and a *reduce()* function for gathering and performing computations on data from sensors. The composition of reduction functions and chained when-blocks enable powerful batch processing data flow applications to be implemented in an IoT context using Command and Control. The semantics of the language are also close to written English, making Borg simple to program in. As an additional advantage, the division of Borg programs into discrete when-blocks
allows for static program analysis graph algorithms to easily partition workload logic among controller nodes, thereby aiding application scalability.

With the additional improvements suggested in our Future Work section, Command and Control should present a useful programming model for intuitively defining scalable IoT applications.

6 Future Work

One of the key concepts that should be implemented in future iterations of the Command and Control architecture is the idea of logical end-device groups. Currently, physical device groups are explicitly defined as sets of device identifiers in the Borg language. A powerful addition to the system would be the ability for groups to be implicitly defined based on shared attributes or conditions. This is an idea that we are experimenting with that would greatly improve the ability of the system to adapt to changing network topologies as new end-devices register with the metaserver.

Another potential addition would be the implementation of more advanced notions of state in the Borg language. Currently, Borg does not support arrays or dictionaries. Providing an abstraction similar to a key-value store or array could be useful for IoT applications. However, this presents new challenges for accomplishing the division of labor via static program analysis. Shared state between controllers would also complicate the sharding of the Command and Control infrastructure, resulting in new hurdles related to maintaining inter-controller consistency.

Finally, we recognize the need for further evaluation of the system. We plan to investigate non-greedy static program analysis algorithms and their abilities to load-balance workloads in Command and Control. We also hope to run analyses on how system performance scales with large numbers of end-devices interacting across groups.
First and foremost, I'm incredibly thankful to have worked with Nishant Jain, my friend and colleague for many years. I am eternally grateful to our advisors, Professors Yang Richard Yang and Mahesh Balakrishnan, for guiding us throughout this semester. I would also like to thank my parents, K.T. and Vidya Srinivas, for their constant encouragement and advice. Finally, I'd like to acknowledge and thank Professor James Aspnes, the Yale Computer Science Director of Undergraduate Studies, for allowing Nishant and me to collaborate on this joint thesis project.
References


[9] “No, it is well known that the c preprocessor is not turing complete. looking in... — hacker news.” Accessed: 2017-05-03 at [https://news.ycombinator.com/item?id=4796221](https://news.ycombinator.com/item?id=4796221)

A Backus-Naur Definition of Borg

<when> ::= <condition> "{" {<statement>} "}" 
<condition> ::= <value-expr> <relational-opr> <value-expr> | ["not"] 
             "(" <condition> ")" | [logical-opr] <condition> | <event-name> | 
             <timer-name> 
<value-expr> ::= <reduction> [<math-opr> <value-expr>] 
               | <number> [<math-opr> <number>] 
               | <string> [<string-math-opr> <string>] 
<math-opr> ::= + | - | * | / 
<string-math-opr> ::= + | * 
<relational-opr> ::= > | < | >= | <= | == 
<logical-opr> ::= and | or 
<reduction> ::= <group-name> ".reduce(" <function-name> "," 
               <reduction-function> ")" 
<function-name> ::= <string> 
<reduction-function ::= <string> 
<number ::= Just a general number in floating point or decimal form 
<string ::= Just a general string as defined normally 
<statement ::= <declaration> | <var-assignment> | <group-assignment> 
| <timer-assignment> | <action> | <raise-end> 
<declaration ::= <type-specifier> <var-assignment> | "DG" 
| "Timer" <timer-assignment> 
<type-specifier ::= "number" | "string" | "event" 
<var-assignment ::= <var-name> = "<value-expr> 
<group-assignment ::= <group-name> = "{" <devices> "}" 
<timer-assignment ::= <timer-name> = "<number> 
<timer-name ::= <string> 
<var-name ::= <string> 
<group-name ::= <string> 
<devices ::= <identifier> {, <identifier>} 
<identifier ::= IP address:Port 
(Note: Identifier definition can be any generic form of identification) 
<action ::= <group-name> " .action(" <function-name> ["params=" <param-list>]" 
<param-list ::= <key>"="<value> {, <key>"="<value>} 
?key ::= <string> 
<value ::= <string> 
<raise-end ::= "raise" <event-name> | "end" <event-name> 
<event-name ::= <string>