Frameworks for a General-Purpose Smart Home Operating System

Donald Hoelle
Advisor: Dr. John Seng

June 13, 2014

Table of Contents

Abstract

Smart home technologies are rapidly growing in prevalence. For my senior project, I designed and implemented the beginnings of a general-purpose framework for a unified Smart Home Operating System, capable of controlling all of the diverse aspects of automated homes. This paper outlines the design challenges involved in building generic smart home systems, as well as the project architecture I designed and implemented to attempt to solve this problem.

Introduction

The purpose of this project was to develop the initial framework for a complete ***Smart Home Operating System***: a combination of hardware and software that controls collections of disparate smart devices through a single, unified system, specifically in the context of a home or business environment. This framework is to be used as the basis for a complete ***Smart Home System***, which will automate a significant portion of its users' everyday tasks.

Background

Smart devices like the Nest Learning Thermostat, Phillips Hue smart lightbulb, and Yale smart lock allow users a limited degree of digital control over their physical environment. However, most of the smart devices that exist today use proprietary interfaces which do not interact, which introduces some significant problems including a lack of shared data, frustrating user experiences, and high device manufacturer overheads.

Design goals

The end goal of this project was a framework which is capable of supporting all of the major requirements of a Smart Home system. This system also must be easy for both device manufacturers and smart home software developers to utilize. Given these goals, the framework must support a wide variety of devices, and it must meet the usability and security requirements of the domain.

To ensure that the framework can support a wide variety of devices:

- The framework must be extendible to new types of components, devices, and new uses of existing appliances
- The framework must be compatible with as many accepted industry standards as possible

- The framework must be easily adaptable to changes in network protocol, physical interfaces, and device structure and composition

To ensure that the framework is easy to use:

- The framework must prioritize simplicity in design and function
- The framework must use common protocols and design patterns
- The framework must make itself human-readable, wherever possible

To ensure the framework is secure:

- The framework must minimize the information that is communicated, providing only what is required for a given system
- The framework must prevent code from accessing and manipulating values outside of its scope, whether the code is developed internally or by a third party
- The framework must encrypt information at rest and information in transit for all parts of the system

Motivations & Considerations

Programming language

The following factors were important to consider in choosing the appropriate programming language to use for the smart home framework:

- Portability: the ideal programming language would be quickly portable to different architectures
- Safety: the ideal programming language would help prevent buggy or unsafe code. Features that contribute to that goal include: type-safety, memory-safety, and good constructs for concurrency
- Performance: the ideal programming language would use a minimal amount of resources to complete its task. For the purposes of this project, the ideal programming language would be able to work within the limited resources of a low-power embedded system
- Support: the ideal programming language would be well supported by its authors

Given those considerations, the team decided to use the *Go* programming language. Natively supported *Go* builds exist for the x86, ARMv5-7, and amd64 architectures[1], and many ports exist for specific processors and embedded hardware.[2] *Go* is type-safe, memory-safe, and contains built-in concurrency features which are both simple and sound. In terms of performance, *Go* performs comparably to other natively compiled

[1] http://golang.org/doc/install
[2] http://golang.org/doc/install/gccgo

programs on standard benchmarks, and their developers argue that improvements to standard libraries will increase those metrics over time.[3]

Smart Home Model

A smart home system can be broken down to three essential functionalities:

1. Sensing: the ability to perceive physical phenomena
2. Actuation: the ability to affect physical phenomena
3. Control: the ability to determine a useful course of action based on sensed phenomena, user preferences, schedules, or other inputs. The determinations of a *useful* smart home should have some meaningful effect, such as commanding actuators to manipulate the environment, or notifying the user

While sensing and actuation are theoretically separate, there are very few differences in the way in which they are manipulated. In practice, many devices contain a combination of sensors and actuators within a single unit. Therefore, in this project, the sensing and actuation categories will be merged into a single category: ***Components***.

Control: centralized, decentralized or mixed?

Centralized

In a ***Centralized Smart Home System,*** all meaningful control is granted to a single unit or a single collection of units. In this project, this single unit or single collection of units is referred to as a ***Smart Home Controller***. In this model, the Smart Home Controller interprets readings from sensors across a network to build a picture of an environment. The Smart Home Controller issues commands to actuators in response to those readings. In a fully-centralized smart home system, all control logic is contained within the Smart Home Controller.

As an example: in a centralized smart home system, a collection of temperature sensors may report temperature information back to a Smart Home Controller, indicating that it has become too hot. In response, the Smart Home Controller may instruct a connected air conditioning unit to turn on.

Decentralized

[3] http://golang.org/doc/faq#Performance

In a ***Decentralized Smart Home System***, control over connected actuators and access to connected sensors may be granted to some or all devices within the network. Explicit control roles are not necessarily defined.

As an example: in a decentralized smart home system, a connected toaster may detect the presence of connected lightbulbs on a shared network. It could then flash those connected lightbulbs when it had finished toasting.

Mixed

In a ***Mixed Smart Home System***, actuators and sensors are controllable either by a centralized Smart Home Controller or by other means. This is most commonly seen in the form of devices which are controllable via their own mobile or web applications (for direct control), but who also open their devices up to control via an API. Smart home aggregator products such as the Revolv[4] use device API access to coordinate activities between discrete devices; however they most often do not disable the ability to control the device through other means.

Decision and Justification:

A primary goal of the greater smart home operating system is to be secure from attack. In the decentralized and mixed smart home models, each connected device is capable of determining methods by which it may be controlled. In those models, it is nearly impossible to ensure that each device will maintain the privacy and security standards required to meet the security goal. Therefore, this project, and the greater smart home operating system design on which it is based, is based on the Centralized Smart Home System model.

Instructions: Device-Oriented or Component-Oriented?

Device-Oriented

In a ***Device-Oriented Model***, each separately-controllable unit is independently addressed. In this way, each "thing" is considered to be a full "Device". A single physical unit may contain more than one controllable part, but each of those must have a unique network address. Commands issued in a device-oriented model are flat and easy to parse.

[4] http://revolv.com/

As an example: a single physical unit which contained both a light-emitting part and a light-sensing part would be considered to be two separately addressable Devices; one light-emitting Device and one light-sensing Device..

Component-Oriented

In a ***Component-Oriented Model***, each discrete component within a separately-controlled unit is considered a unique whole. Commands are delivered to the ***device controller***, which is a single CPU tasked with controlling multiple physically connected components. The device controller must unmarshal the instruction and pass it to the correct component.

For example: a single physical unit which contained both a light-emitting component and a light-sensing component would provide one network address. An instruction meant for the light-sensing component would be sent to the Device address, then routed to the Component.

A Component-Oriented Model supports composite instructions natively.

Composite Instructions

One weakness of a Device-Oriented Model is that it does not natively support ***Composite Instructions***. A Composite Instruction is a single message which contains multiple instructions meant for different components on the same physical unit. Composite instructions may require that individual instruction stanzas be executed in a specific order. This allows the system to simply enforce orders-of-execution within physical units. Enforcing execution ordering without composite instructions is difficult

Decision and Justification:

Due to the potential need for composite instructions, this project followed the component-oriented model.

Addendum:

It should be noted that the definition of a component is not precise. A microwave oven can potentially be considered a combination of a light emitting component, a spinning tray component, a door-opening

component, and a magnetron component. However, this level of component control may not be beneficial to a smart home system. This project followed a pragmatic rule when defining components, favoring definitions which make it easy to produce necessary effects. In this project, a single microwave oven component would contain all of the pieces represented above, where instructions would include “turn on” and “pause”.

Components: Typed or Untyped?

In any useful system which interacts with different kinds of devices, the types of those devices must, at some level, be understood. For example: a useful smart home system will make a distinction between a lightbulb and a door lock that it controls because the functionality of those two types of device are different. However, the point at which the type distinction is made within a framework and the method used to make the type distinction can greatly affect the behavior of the system. Most existing systems fall between on a spectrum between two extremes:

Typed

In a ***Typed Component Framework***, each component must be typed according to a ***Component Class***. A Component Class defines a set of devices with shared functionality. This level of abstraction allows a control system to use higher-order logic to control the behaviors of like devices, which in turn reduces the size and complexity of control code.

As an example: consider the class of “light-sensing components”. Although different models of light-sensing components exist, they all perform the same basic function: detecting the physical phenomena of light (e.g. intensity, color, power). Thus a Smart Home Controller may choose to control them similarly.

However, when using the elements of abstraction in a Typed Component Framework, a control system may not be able to take advantage of functionality which is unique to a specific model of a component. The total available functionality for any Class of Component will be limited to the functionality built into the known types of the framework.

As a hypothetical example: assume a manufacturer builds a new light-intensity-sensing component capable of determining the “fun” property of light. In a Typed Component Framework, the results of the

"fun" sensor could not be integrated into the behavior of the system until the concept of "fun" was integrated into the class of light sensing devices.

Untyped

In an ***Untyped Component Framework***, each Component may be controlled by "raw" component-specific logic. At some level of the framework, an engineer or user must be able to provide both driver code and control code for each device/component. In this sense, the control of a device is relegated to it's specific control program, and the framework itself has limited control. However, a custom-tailored device control software could be made to use all of the functionality of a novel component.

In reality, no unified system is fully untyped. However, many systems may allow a device programmer to explicitly script the behavior of an individual device, outside of the framework's understanding of type.

Continuing the previous example: imagine the Light-and-Fun-Sensing Component mentioned above were connected to a system with an ***Untyped Component Framework***. Even though the system has no methods for controlling Light-and-Fun-Sensing Components, the device's manufacturer could supply a driver which explicitly scripts it's behavior and use (for example, by requesting that music be played whenever the "fun" levels become too low). This allows for immediate use of new types of components, however, in those instances component programmers would be in charge of determining control behaviors for the system.

Decision and Justification:

The goal of this project is to create a consistent, unified smart home system. It would be difficult to preserve consistency while allowing component programmers to inject their own control routines into the system. Therefore, this project implemented a Typed Component Framework.

Devices: Stateful or Stateless?

Stateful Device Model

In a ***Stateful Device Model***, each device can retain a concept of state, such as whether connected components are "on" or "off". A controller

which wants to control an actuator will generally issue an instruction for the device to switch between states. Different applications of the Stateful Device Model may choose to use different ***state retention policies***, such as “hold state until otherwise notified” or “hold state for a set period of time, after which switch to a default state”. Choice of state retention policy directly affects the experience of the system, especially in the event of control system failure.

Stateless Device Model

In a ***Stateless Device Model***, each device contains no concept of state. When a device controller receives an instruction, it may act upon the instruction and return a response, but no state information is persisted. This model is reasonable for on-demand sensor devices. However, most actuators require state to be useful. In order for actuators to work within a Stateless Device Model, each actuator must implement mechanical mechanisms to retain state after the instruction has been processed.

Decision and Justification:

Although preserving state requires a bit of additional overhead on the part of the Device Controller, it reduces design constraints for the connected physical components and is much easier to use and understand. Therefore this project adopted the Stateful Device Model.

Summary

The framework for this project uses a ***Centralized Control Model***, where all endpoint devices are controlled by a single control entity.

Devices are considered to be ***Component-Oriented*** and messages sent to each device must specify which specific component of the device should be controlled.

Components themselves are ***Typed*** by the framework, and the framework must support all of the types of Components that it wishes to use.

Target devices are ***Stateful***, meaning they preserve concepts of the state of their connected components.

Project Architecture

Below is a model of the target final smart home system:

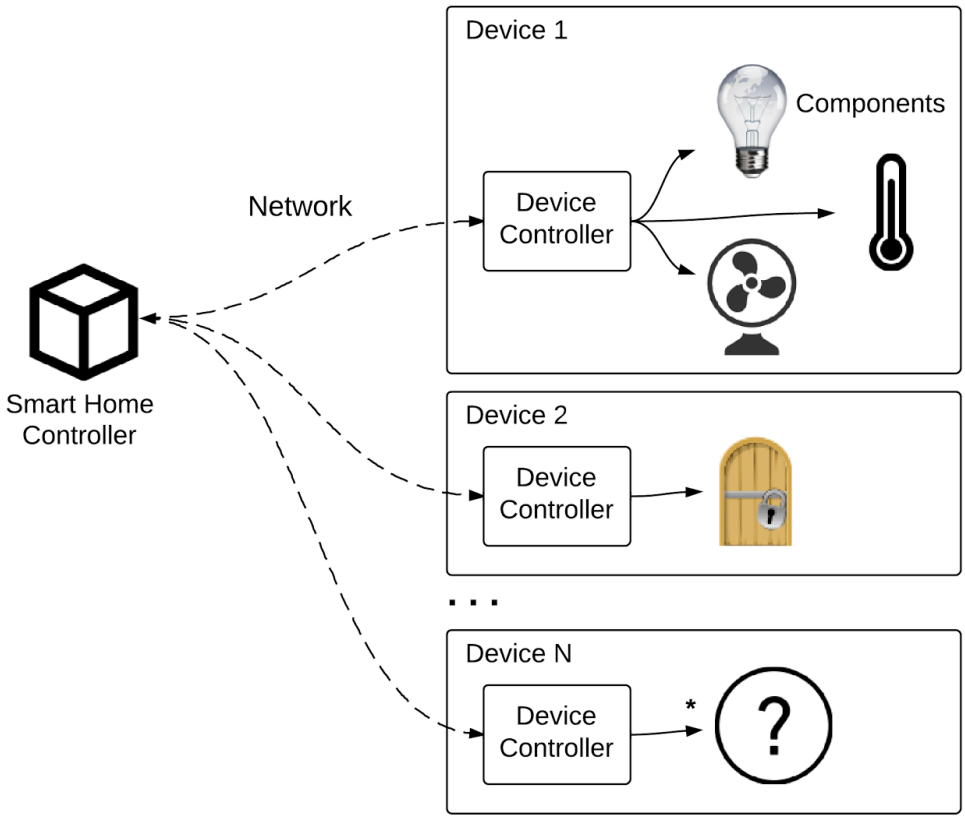

Figure 1: Model of a Smart Home System

Smart Device Communication Protocol

The ***Smart Device Communication Protocol*** defines how Smart Home Controllers and Device Controllers discover each other, how they securely couple, and how they communicate in order to provide the smart home system.

Authentication and Encryption

Considerations

In this project, each device is controlled by one-and-only-one Smart Home Controller unit. Due to the sensitive nature of data that is collected and

transferred by devices in a smart home, it is critical that all communications be securely authenticated and encrypted.

The association process for Smart Devices within this system is pictured below:

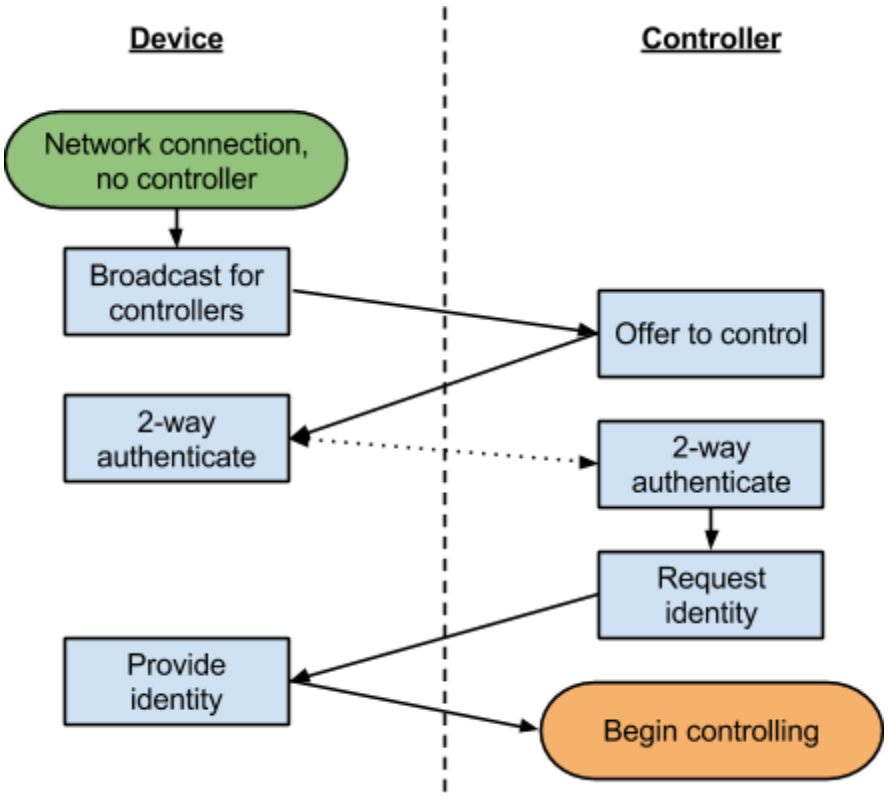

Figure 2: The association process between Smart Devices and a Home Controller

The precise methods for discovery, secure authentication, and encryption do not impact the behavior of the system and therefore are left to the implementation.

Implementation

Unfortunately, complete forms of discovery, authentication and encryption were outside of the scope of this project. In this implementation, discovery was performed manually, and secure authentication and encryption were stubbed.

Message Types

Considerations

Modes of Communication

In this project, a single Smart Home Controller unit communicates with any number of Device Controllers, where each Device Controller contains Components which are either sensors or actuators. The communication protocol for this model must support messages of the following types:

- Instructions sent from the Smart Home Controller to Device Controllers, with the intent of either:
  - affecting a specific actuator
  - requesting a reading from a specific sensor
- Responses to Instructions sent from the Smart Home Controller to the Device Controller.
- Notifications (e.g. sensor readings or heartbeat pings) sent from Device Controllers to a specific Smart Home Controller.

The ideal communication system should support varied transport mechanisms, so long as those transport mechanisms can guarantee message integrity. As mentioned in the previous section, all communications must be authenticated and encrypted.

Representation of Communication

There are many potential paradigms to choose from when representing communication across networks. As performance is a major issue for smart home devices, particularly as it relates to energy usage, it is important to choose a paradigm that minimizes raw data transfer.

Implementation

This project used the JSON-RPC protocol. JSON-RPC is a simple and well-supported protocol, and the JSON data format is relatively lightweight while remaining human-readable. In production implementations, it may be worth exploring more compressed marshalled message formats, such as MessagePack, or binary message formats.

Example messages within the communication protocol are provided below:

```
{
  "method": "ToDeviceService.Send",
  "params": [{ "Component": "lighting_component-1",
               "Method": "set_power",
               "args": [75.0] }],
  "id": 5542
}
```

Example Call 1: JSON-RPC message directed at a Device, instructing it to set the output power level of a connected Component, lighting_component-1, to 75.0%

```
{"result": [true], "id": 5542}
```

Example Response 1: JSON content of the response returned by a Device that received Example Call 1 and was able to successfully complete the instruction.

```
{
  "method": "ToDeviceService.Send",
  "params": [{ "Component": "light_sensing_component-1",
               "Method": "get_reading",
               "args": [] },
             { "Component": "door_lock_component-1",
               "Method": "unlock",
               "args": [] }],

  "id": 671
}
```

Example Call 2: This composite message instructs a Device to take a reading of the light levels, and to unlock a door

```
{"result": [1452.88, false], "id": 671}
```

Example Response 2: This composite response indicates that the light level was 1452.88 lumens, and that door_lock_component-1 could not complete the unlocking procedure

Device Controller Framework

The Device Controller Framework provides easy control for Devices with an arbitrary composition of Components. The following example illustrates a Device Controller that has been configured to control a *Light Emitting Component*, an *Air Moving Component*, and a *Temperature Sensing Component*.

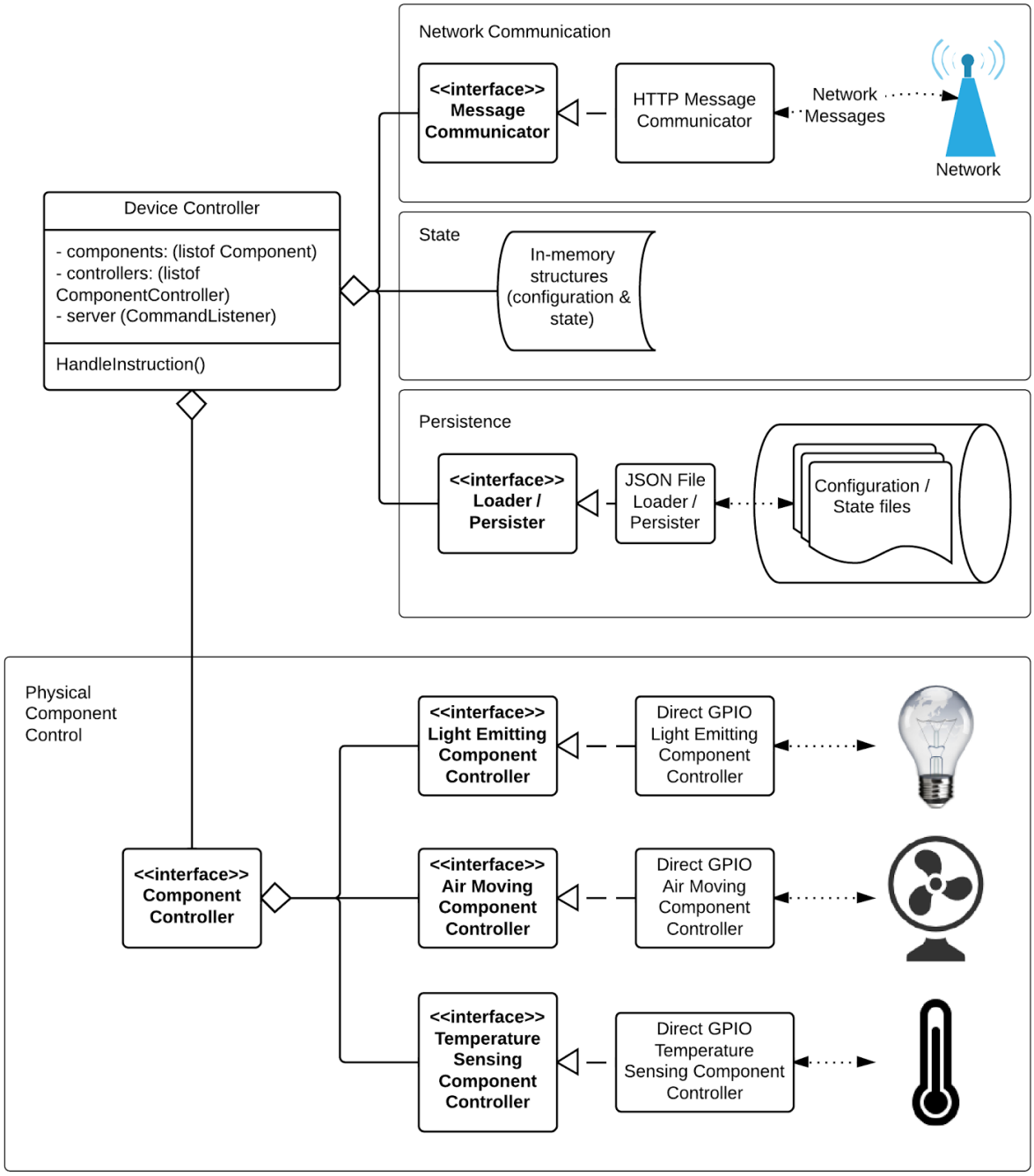

Figure 3: Software Diagram for the Device Controller Framework

Network Communication

The **Network Communication** system sends and receives Messages with a Home Controller across a network interface. The details of the specific network structure and message transfer protocols are abstracted from the primary Device Controller class via the **Message Communicator** interface, which provides methods to set an incoming message handler (callback), and to send messages

(including responses) to the connected Smart Home Controller. The Message Communicator also performs any message processing required by the system, including message queuing, dispatching, and encryption.

The Handle function of the Device Controller is expected to interpret and act on Messages passed by the Message Communicator. It is also expected to format outgoing messages appropriately before handing them to the Message Communicator to be sent.

The implementation for this project used an HTTP Message Communicator. JSON-RPC messages were expected to be sent to the Device Controller as HTTP Content embedded within an HTTP Post. Response strings were embedded into the content of the server's HTTP Response.

State & Persistence

A reasonable smart home model assumes that devices can fail or lose power at any time. For this reason, it is important to constantly preserve the state of the machine to media which persists through power failure. A **Loader/Persister** interface provides a level of abstraction for the Device Controller so that the exact mechanics of persistence can be adjusted easily.

The implementation for this project used a JSON File Loader/Persister. Serializable objects were serialized to JSON using the methods in the Go standard library.

The nature of this system involves arbitrary numbers and types of Components connected to each devices. Therefore it requires heavy use of polymorphism. As Go is a statically-typed language, marshalling and unmarshalling polymorphic structures is non-trivial. Utilizing a generic unmarshalling library was especially helpful here.[5] In addition, Go does not support nillable values for primitives, so supporting JSON marshalling with partially-initialized structures required replacing all primitives with pointers to primitives.[6]

Physical Component Control

There are many methods available for physically controlling components with a microprocessor. At the highest level, components may be controlled directly via GPIO or through bus communication standards like I^2C.

[5] https://github.com/hailiang/snippets/blob/master/persist_polymorphic_objects.go
[6] https://github.com/google/go-github/issues/19

This Device Controller framework supports multiple frameworks through the use of **ComponentController** interfaces. Each type of Component will have a matching ComponentController, which will control all Components of that type connected to the Device through the same method/bus. For example: a Device with multiple LightEmittingComponents connected over an I^2C bus will instantiate one I2CLightEmittingComponentController.

All implementations of the ComponentController interface must implement the concept of **slots**: distinct pathways for communicating to unique components. In a Direct GPIO Component Controller, this may map to physically separate slots. In a bus-based Component Controller, each slot will likely be defined by the unique bus address. This abstraction allows the framework to select from multiple components without having to understand the specific details of the connection.

The implementation for this project included an additional kind of ComponentController, an **Emulated Component Controller**. Emulated Component Controllers programmatically mimic the interactions of real physical components. This allows for the system to be quickly and efficiently tested, and for larger interactions to be fully simulated.

Smart Home Controller Framework

The Smart Home Controller Framework shares a majority of it's design with the Device Controller Framework. The primary difference is that the Smart Home Controller does not implement Physical Component Control, and instead implements a **Decision Engine**.

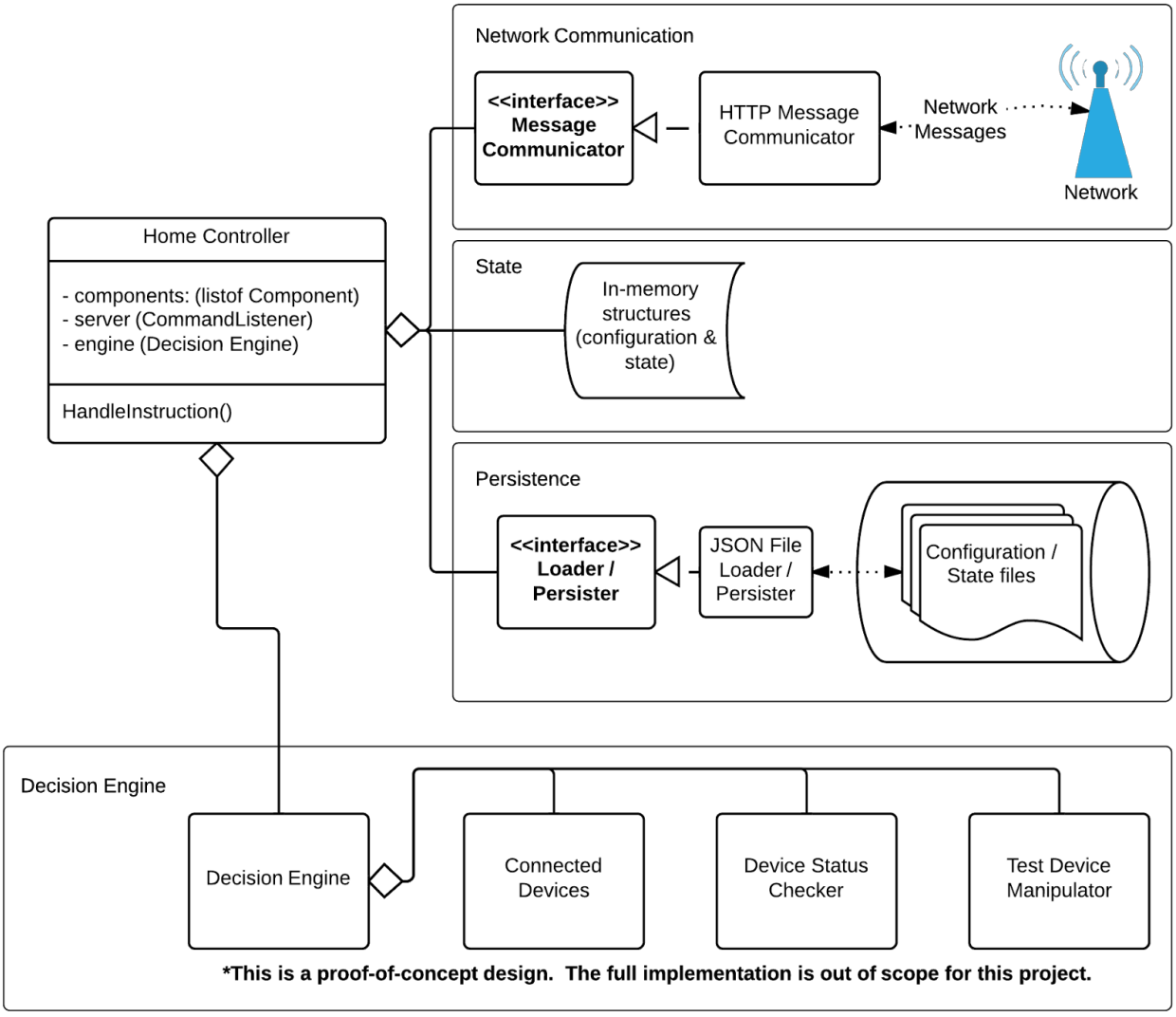

Figure 4: Software Diagram for the Smart Home Controller Framework

Decision Engine

The primary goal of the Smart Home Controller Decision Engine is to make intelligent decisions that benefit the users of the Smart Home System. In theory,

this involves:

1. Assessing the needs of users across affectable domains (through interactive interfaces or other means)
2. Then combining sensor readings with physical models to determine conditions within the building
3. Then determining courses of action which result in conditions which more closely match the needs of users

The decisions of the Decision Engine are then actualized by the Smart Home Controller (via remote control of Devices) in order to produce a beneficial effect.

While the Decision Engine is an exciting and essential component of a full smart home, it is unfortunately outside of the scope of this project. The Decision Engine representation which was used instead consisted of a small skeleton designed to mimic very basic functionality, as well as some testing methods to prove that a larger Decision Engine could be built upon this skeleton.

The **Connected Devices** Class stores a representation of those Devices which have authenticated and connected with the Smart Home Controller. It contains the unique identifiers, authentication information, and the Component structure of each Device. The **Device Status Checker** is a simple Class which periodically determines whether or not a Device is still available (by sending heartbeat messages). The **Test Device Manipulator** contains arbitrary Component-Class-dependant algorithms, running indefinitely, which continually affect the status of connected Components. While these patterns are not useful, they do serve to test the ability of connected Devices to be rapidly manipulated.

Conclusion

The implementation the Device Controller Framework and Communication Protocol sections closely matched the theoretical models described above. A heavy use of interfaces and polymorphism allowed for this framework to be sufficiently general and abstract, while retaining the necessary functionality for smart home control. While a complete Smart Home Controller implementation was outside of the scope of this project, the stub classes that were implemented served as a reasonable proof of concept.

Go served as a useful programming language for this project. It provided strong type safety, and dodged many of the potentially-dangerous manual memory management bugs from C-like systems programming languages, all without an appreciable impact to performance. Certain tasks, like manipulating polymorphic collections and persisting

objects, were difficult to complete with Go, but, eventually, elegant solutions for all of those roadblocks were discovered.

The Test-Driven Development workflow was useful during the implementation phase of this project. It allowed for confirmation that each individual piece of the larger software was complete before moving on. It seemed especially helpful due to the large, layered nature of this project's architecture.

Future Work

With the implementation of a Smart Home Controller Decision Engine, this model will become reasonably complete and pragmatic. The vision for this project is to implement the remaining pieces, and to expand and refine the design of the existing framework, as part of a commercial venture which will sell all-in-one smart homes directly to consumers.

Other areas which require implementation include user interfaces, security, and (optionally) cloud interaction.