Project Molecule

Design Document v1 Revised 08-Nov-2016

May1739
project.molecule@outlook.com

Dr. Arun Somani – Advisor
arun@iastate.edu

Ryan Wade – Team Leader
ryanwade@iastate.edu

Nathan Volkert – Communications Lead
nvolkert@iastate.edu

Daniel Griffen – Key Concept Holder
dgriffen@iastate.edu

Alex Berns – Webmaster & Scribe
tagger94@iastate.edu

1 Contents

2 Introduction

2.1 Project Statement

Project Molecule is a system for smart home living. It is a series of nodes that link applications across the house for convenience and ease of use. Each node will be a Raspberry Pi with secure and safe communication.

2.2 Purpose

As a group, we are intrigued by the possibilities of a smart home system. Project Molecule serves as an example of how society as a whole might adapt to smart home living. It can allow people to control aspects of their home conveniently, safely, and securely adding to their quality of life.

2.3 Goals

We have a good number of goals to achieve throughout our work on this senior design project. First, we have general learning to accomplish. This includes learning the Rust Programming Language and getting a better understanding of fault tolerance. Next, we hope to achieve network communication between the Raspberry Pi's in a safe and reliable manner. We want to maintain integrity of the system while still retaining functionality. We want to create nodes in such a way that they can have functionality, such as turning on a light, something the users could benefit from. We want to create a UI that is convenient to use to operate the nodes. Another goal is to introduce some automation to the system using something like IFTTT (If This Then That) or web hooks. Stretch goals include installing a system in an actual home or apartment, more complicated application procedures, and getting the rights to potentially expand the project beyond the time period of senior design.

3 Deliverables

Primary Goals

- Network of Rasberry Pis
- Web Application to control network
- Scale Model House for demo
- Source Code on GitLab
- Documentation explaining use

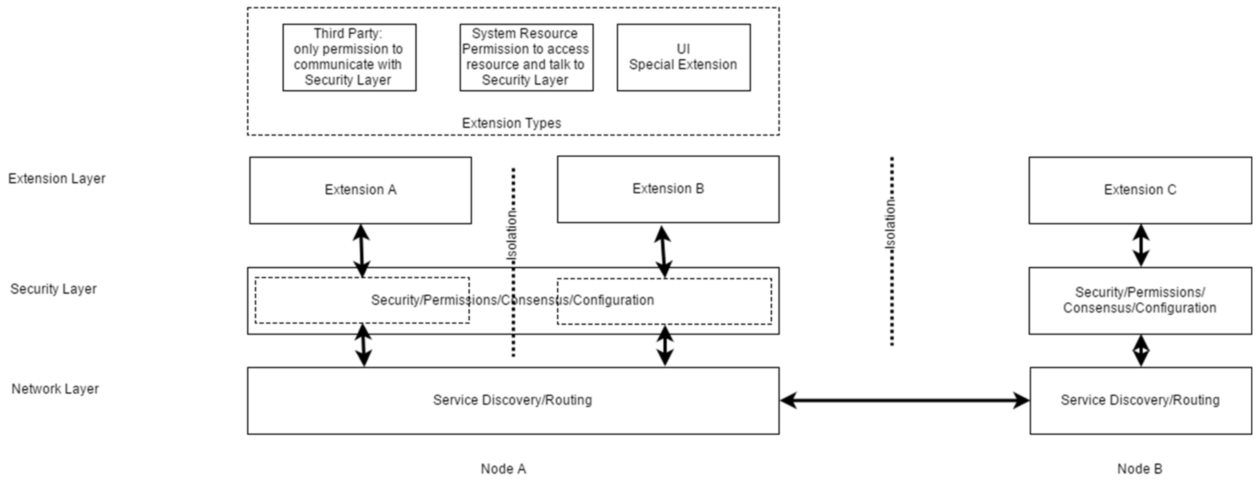
Extra Goals

- Demo video of real world use

4 DESIGN

4.1 SYSTEM SPECIFICATIONS

4.1.1 System Diagram


Third Party: only permission to communicate with Security Layer
System Resource Permission to access resource and talk to Security Layer
UI Special Extension
Extension Types
Extension Layer
Extension A
Extension B
Extension C
Isolation
Isolation
Security Layer
Security/Permissions/Consensus/Configuration
Security/Permissions/ Consensus/Configuration
Network Layer
Service Discovery/Routing
Service Discovery/Routing
Node A
Node B

4.1.2 Definitions

Project Molecule is composed of many components and arranged similarly to a molecule. As such, naming conventions are drawn from that domain.

Particle Layer: Smallest component of the system are divided into the following types

Electrons: 3rd Party Extensions

Protons: System Resource Wrappers

Neutrons: Device Level Drivers

Atomic Layer: Manages **particle** lifecycle, provides data synchronization, and enforces permissions

Bonding Layer: Networking handles **particle** to **particle** communication and discovery

4.1.3 Functional

Particles

shall be isolated from other layers

shall provide a common package format

shall be unique (identifiable instance)

may share configuration (common service type)

shall implement common **interfaces** for service discovery

- shall advertise implemented **interfaces**
- shall respond to lifecycle events
- may consist of:
  - **Electrons** which
    - Shall not have direct System Access
  - **Neutrons** which
    - Shall encapsulate peripheral devices
  - **Protons** which
    - Shall wrap specific System Resources

Interfaces

- Shall describe features of each **particle** (Switch, Media Player, Audio Player, Outlet Control)

Atoms

- Shall manage **particle** Lifecycle. They:
  - shall ensure messages between **particles** is permitted
  - shall ensure inter-**particle** communication is secured
  - shall ensure only the destination **particle** may read the message
- Shall manage **particle** Configuration:
  - **Atoms** shall synchronize data between other **atoms**
  - **Atoms** shall ensure **particles** have permission to access the configuration
  - Configuration may be partitioned:
    - Each partition shall have independent permissions
- Shall store **particle** permissions:
  - Permissions must be synchronized between all **atoms**

Bonds

- Shall facilitates inter-**particle** communication which includes inter-**atom** and **atom** – **atom** communication
- Shall route multiple instances of the same **particle**.
- Shall have a routing layer which

Must determine the best **particle** instance to use for a given message

Must ensure successive requests for a **particle** by the same service will resolve to the same **particle** instance

Must be aware of all **particle** types and instances on the network

Shall provide an API for **particle** registration and discovery

4.1.4 Non-Functional

The system must have sufficient bandwidth to stream HD (1080p) video

A single node failure must not bring down the entire system

The system must not be down for more than 10 minutes in a given year (0.001% downtime)

A single compromised node should not be able to exploit the entire system

Extension API and lifecycle must be fully documented

Cost of an individual node (Raspberry Pi, storage + power supply) should be no more than $100

4.2 Proposed Design/Method

4.2.1 Bonding Layer

The bonding layer manages the connections between each node. The bonding layer is in charge of:

- Service Discovery
- Node Discovery
- Message Passing
- Node Verification
- End-to-End Encryption

Service discovery is the process where the bonding layer determines which particles are located on which devices. The discovery of particles allows the bonding layer to transparently route messages to the proper particles without the rest of the system knowing where exactly the message is going.

Node discovery is the process that allows the bonding layer to create the device network. The bonding layer will have a known list of good nodes that it will attempt to find on startup. A cryptographically secure public key will identify good nodes to ensure that a malicious node cannot fake its way into the network.

Message passing is the main functionality of the bonding layer. The bonding layer will take messages provided to it by its nodes atomic layer, and forward them to the correct destination device.

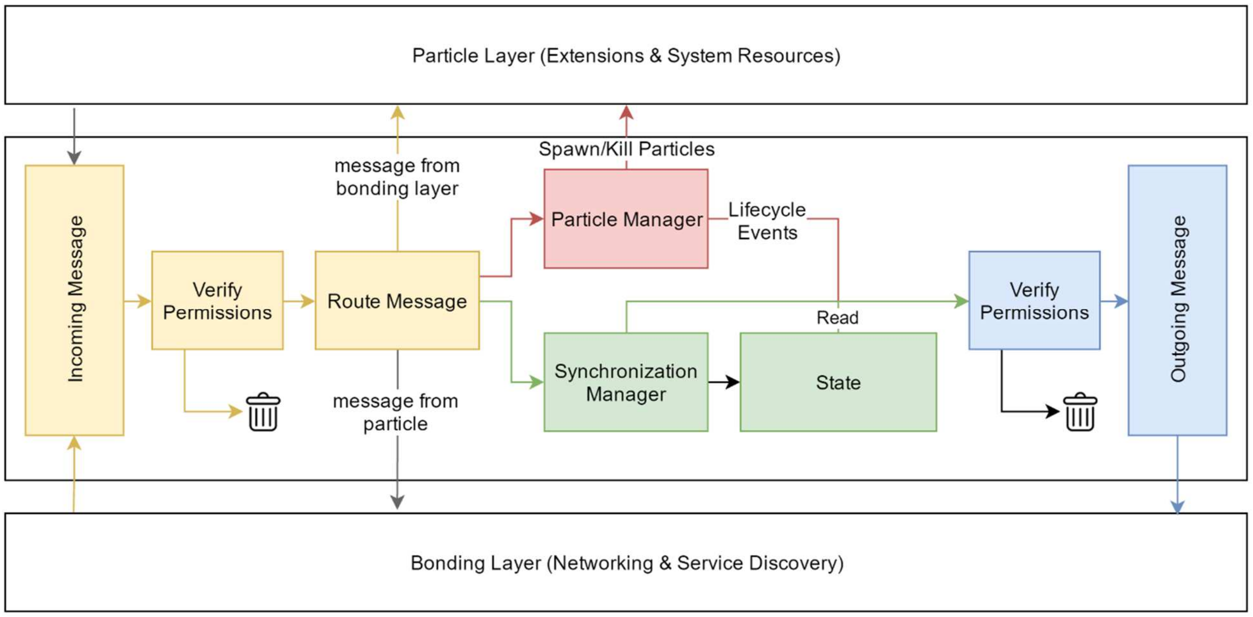
Node verification is how the bonding layer determines that it is not talking to a maliciously added node. When a node is added to the network it will share a static public key with the rest of the network. The bonding layer will save this public key permanently on its own device. Every message sent over the network will be cryptographically signed with the nodes private key, in this way the bonding layer can verify the authenticity of the node by verifying the message signature.

The bonding layer will also encrypt all communication end-to-end using a Diffie-Hellman key exchange system. When communication is established, the sending node will generate an ephemeral key pair. It will then use this key pair with the static public key of the receiving node to derive an ephemeral symmetric key. After this it will pass the ephemeral public key, signature, and encrypted message to the receiving node. The receiving node will verify the identity of the sending node (part of node verification) using the signature, then decrypt the message by combining the ephemeral public key with its static secret key.

4.2.2 Atomic Layer

The atomic layer has the following responsibilities:

- Manage Particle Lifecycle
- Verify Message Permissions
- Synchronize Particle State Data



Message Routing and Permissions (Incoming: Yellow, Outgoing: Blue):

Incoming messages are first checked for adequate permissions. If they are inadequate for the requested action, the message is discarded from the system. The Atomic layer then routes the message to one of the three layers with the following stipulation: All messages originating in the Particle or Atomic layers must be routed through the Bonding layer. This ensures consistent behavior for all messages entering the system. All outgoing messages from the atomic layer must also be passed through permissions verification before being forwarded to the bonding layer.

Particle Management (Red)

Particles are spawned and killed by the Particle Manager. This component also emits lifecycle events in response to other events in the system such as initialization, shutdown, and service requests.

Synchronization and State Data (Green):

Particle State is synchronized between all nodes in the system.

4.2.3 Particle Layer

Particles will interface with the atomic layer which is implemented by the node. There are three types of Particles:

Electrons:

The **Electron** is a third-party extension, and will be sandboxed with as few permissions as possible. These minimal permissions will allow it to communicate with the atomic layer only. In order to gain access to device drivers (**Protons**) or system resources such as the file system (**Neutrons**), the **Electron** will have to make a request through the atomic layer.

Neutrons:

The **Neutron** is a system resource wrapper and first party extension. System resources include file system folders, direct GPIO access, and other hardware. **Neutrons** will not be able to communicate with other particles. Rather an interested particle will have to set up message listeners and register them with the system resource. This will require all messages to go through the atomic layer and prevent system resources from injecting messages into the system.

Protons:

The **Proton** is a device level driver. It will be a third-party extension that extends **Neutrons**. Unlike a **Neutron**, it will have permissions to send explicit messages to system resources. It still cannot message an **Electron** directly but must wait for an interested particle to set up message listeners. A **Neutron** could implement protocols such as I2C using a GPIO **Neutron**, and such has the ability to access device level drivers like General Purpose IO (GPIO) pins on a raspberry pi. Once again, it cannot communicate with other particles without the atomic layer.

Each particle will require the ability to interface with the atomic layer.

Permissions:

	Electrons	Neutrons	Protons
Receive Messages	X	X	X
Register Message Listeners	X	X	
Send Message to Listeners		X	X
Send Message Directly	X		

4.2.4 User **Interface**

The User Interface will be implemented as a Neutron which has permission to host a web server and send messages to event listeners. Electrons register UI panes with the UI Neutron. This pane state is synchronized throughout the system. Only one UI Neutron may be running on a node at a time. UI Neutrons running on different nodes may be viewing different panes, but if they display the same pane, their state must be the same.

Below is a flowchart showing how UI messages propagate throughout the system:

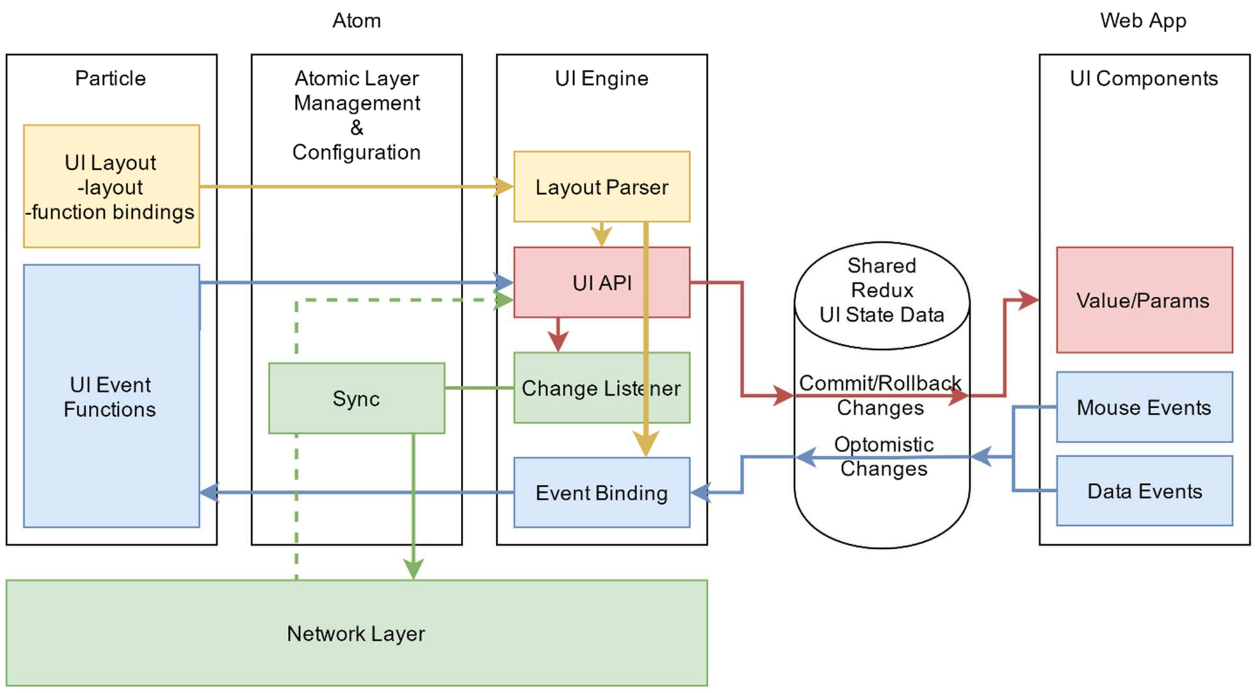


Figure: UI & System Architecture

Yellow: Electrons register Panes and Message Listeners (Event Bindings) with the UI Neutron.

Red: UI Neutron receives messages and updates the Web Apps state and user interface. Any changes to the state also fire the synchronization Change Listener

Green: When the UI Neutrons state changes (red), the data is synchronized between all of the nodes. This will cause UI Neutrons on other nodes to receive similar UI Updates (green dotted line to red)

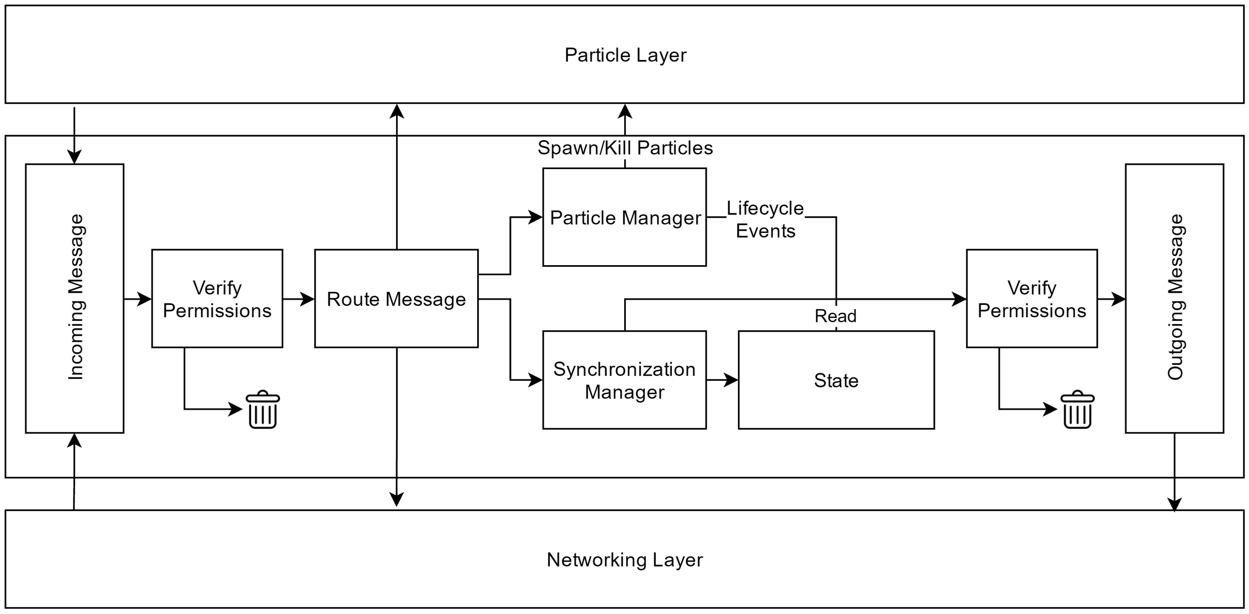
Blue: Web App UI action occurs (onClick, onDataChange) and optimistically changes the user interface. The UI Neutron sends a message to the listeners (Event bindings) registered from the Electrons (yellow). The Electron receiving this event processes the changes and then commits them to the UI Neutron's UI API (red)

4.3 Formal Analysis

Each component of our project will have a separate state machine. When developing the unit tests, as discusses later, the state machine will help us determine what we need to look for that could cause an undesired state.

We currently have not gotten far enough into developing each component to start making the finite state machines but a fair amount of discussion with our advisor has pointed us to this as our primary method of developing tests. The core concept of this project is fault tolerance and a state machine can also help as we start to code each section. Identifying what is necessary for each component to run and making sure that that part has backups is one of the more difficult parts of this project.

The first step in developing the state machine is looking at the fl



ow charts we have created so far. The UI, Particle, and Atomic layer all have flow charts documenting either their life cycle or the movement of actions will help create the finite state machine that will eventually be used to describe the programs they correspond with.

5 TESTING

5.1 INTERFACE SPECIFICATIONS

5.1.1 Bonding Layer

The bonding layer sends messages to the other parts of the system.

This is accomplished by:

Network to Network through TCP sockets

Atomic to Bonding through pipes

Subcomponents are included to test and their Interface Specification. These will include a way to verify messages are being sent and received correctly, that the information is correct and that the information is being passed between the correct components.

5.1.2 Atomic Layer

The atomic layer also sends messages for system functionality

This is accomplished by:

Atomic to Bonding through pipes

Atomic to Particle through pipes

Again subcomponent tests will check to make sure the correct information and quantity of it is being sent from the atomic to the other layer correctly. Tests must also be in place to ensure only approved messages of the systems can be distributed throughout it, that they have proper clearance.

5.1.3 Particle Layer

The particle layer is a bit more involved with both system messages as well as system resources in the neutrons.

These are passed in the following ways:

Atomic to Particle through pipes

Particle to System Resource through POSIX API's, User Interface

The user interface must have tests in place to ensure only authorized commands are sent through the system. This is on top of the tests needed to ensure correct data being sent and received accurately.

5.2 Hardware/Software

Requires a raspberry pi with a Linux distribution which can execute our software. The software used for testing the bonding layer, the atomic layer, and most particle layers will be the basic Rust testing suite, which is included with the Rust compiler.

5.2.1 Bonding Layer

Multiple instances can be set up in a fake environment by spawning multiple processes each listening on a different local port. Tests will be carried out by setting up a fake network of nodes and verifying that the nodes behave according to the specifications.

5.2.2 Atomic Layer

Stub bonding and particle layers will be used for the atomic layer to hook into during testing. Tests should verify that the atomic layer properly passes messages between the particle layer and network layer.

5.2.3 Particle Layer

Need test system to sandbox and instantiate particles. Must be able to inject messages and listen to response messages.

5.2.4 UI (Particle Layer)

Requires alternative testing software due to being written in JavaScript. Human testing and mocha, an automated testing software for JavaScript, will comprise the methods for testing the UI components.

5.3 Test Cases/Process

5.3.1 Bonding Layer

Test **Fault Tolerance:** Bonding layer should tolerate faults and route to other available nodes. Terminate a node early to make sure other nodes do not crash.

Test **Service Discovery:** Test that the bonding layer can find services and open connections to them

Test **Authentication:** Verify that nodes will not start communication with unregistered nodes.

Test **Encryption:** Test that data sent from a node is encrypted and cannot be read by a third party.

5.3.2 Atomic Layer

Test **Configuration:** Data should be properly stored and fire update event on changes.

Test **Synchronization**: Test that if data does not change, it should not broadcast changes. Test that when a change broadcast is received, it does not rebroadcast the change. Test change conflicts

Test **Particle Management** (Section 8.1) – Test handling invalid Lifecycle events. Test out of order life cycle events. Test that particle processes are spawned and killed.

Test **Permissions** – Such as neutrons/protons trying to directly message other extensions. Permissions to access particle types.

Message Handling Test Cases:

Message In	**Message Out**
State Update	Send Synchronization messages if change
Outgoing Message for Particle	Routed to network layer
Incoming Message for Particle	Routed to particle
Invalid permissions	Rejects message

5.3.3 Particle Layer

Test **Lifecycle Events**: Ensure correct outputs. Test out of order Lifecycle event behavior.

Test **Message Binding**: Received Messages are appropriately bound to functions. Test undefined message type response and other errors

Test **Electron/Proton Isolation**: Test Sandbox Settings. Test code that tries to access system resources.

Test **Neutron Isolation**: Test Sandbox Settings. Verify permission to access resources. Test code that tries to use system resources it does not have permission to access.

5.3.4 UI (Particle Layer)

Test **UI API**: Test reading config file and transforming into a UI with panes. Ensure panes look like intended.

Test **Event Binding**: Ensure events are bound and listen to Web App.

Test **Syncing**: Test that a change in UI API is carried over to other open UIs.

Test **Commit / Rollback**:

Test **Layout Parser**:

Test **Layout Parser**:

Test **Layout Parser**:

6 Testing Results

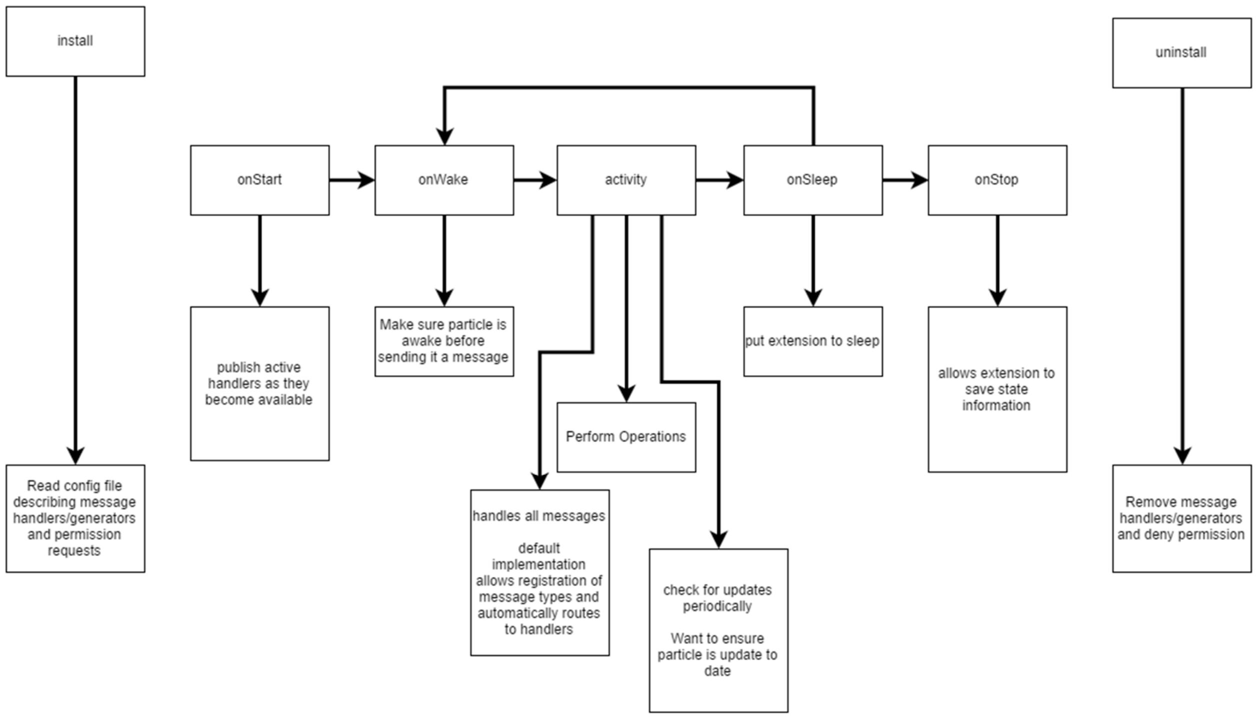
Our initial tests show correct functionality of our basic API and networking, however, we need further implementation of our project completed before more interesting testing results will be available.

7 CONCLUSIONS

So far we have planned out how we want to design our project. We have constructed many diagrams explaining structures and processes. We have thought about many possibilities and methods for doing things, always trying to put extra emphasis on fault tolerant design. Our goal is to create a fault tolerant, safe, and secure platform for smart home computing. We plan to create a network layer, UI, particles, and the atomic layer that links them all together. We decided on this path as it was the easiest way to keep things modular as well as ensure no node can gain too much power or do unauthorized things in the system. We also decided to add voting to the node to counter act any unwanted changes keeping our design fault tolerant.

8 APPENDIX

8.1 PARTICLE LIFECYCLE


install
Read config file describing message handlers/generators and permission requests
onStart
publish active handlers as they become available
onWake
Make sure particle is awake before sending it a message
activity
Perform Operations
handles all messages
default implementation allows registration of message types and automatically routes to handlers
check for updates periodically
Want to ensure particle is update to date
onSleep
put extension to sleep
onStop
allows extension to save state information
uninstall
Remove message handlers/generators and deny permission