Project Molecule

Final Document v2 Revised 23-April-2017

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

Contents

Introduction

Project Statement

Project Molecule is a multi-node shared application environment for smart home systems.

Purpose

The creation of Project Molecule was motivated by frustration over incompatible smart devices on the market. It is intended to be used by 3rd party developers and enthusiasts to link such devices and create a connected home. The project forms a backbone for communication between different devices and handles concerns such as networking, availability, and security.

Goals

The primary goal for this project was a general learning experience for the team. Project Molecule was written in Rust and required many advanced programming techniques to work. This class required project management documents to be produced alongside the code. These skills will be useful knowledge for work in the industry.

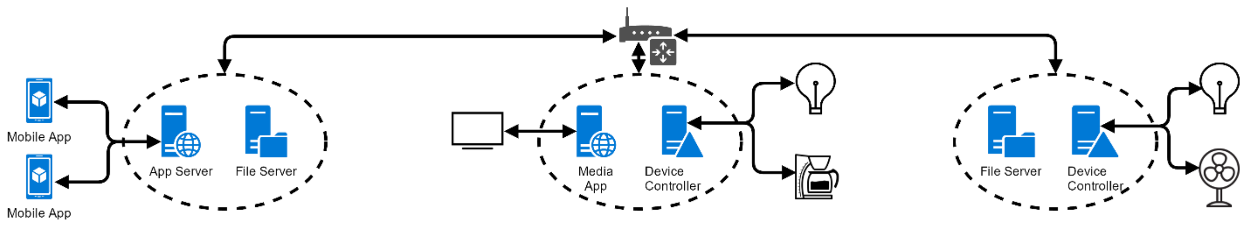
The final project is under the MIT license so any party can continue to work on it. This was determined to be the best license for this project since the audience for it is enthusiasts and businesses.

Fault tolerance and security are also important parts for this project. Not only are they a valuable learning experiences, but also demonstrate good design.

Deliverables

Project Molecule provides a runtime binary for the system and a common API with documentation for 3rd party app developers. Application creation and configuration tools will be provided to setup new projects and configure our system. An operation manual (Appendix I) is also provided to guide the user through setting up their system and creating an example application. All source code will be provided under the MIT license.

Design



System Specifications

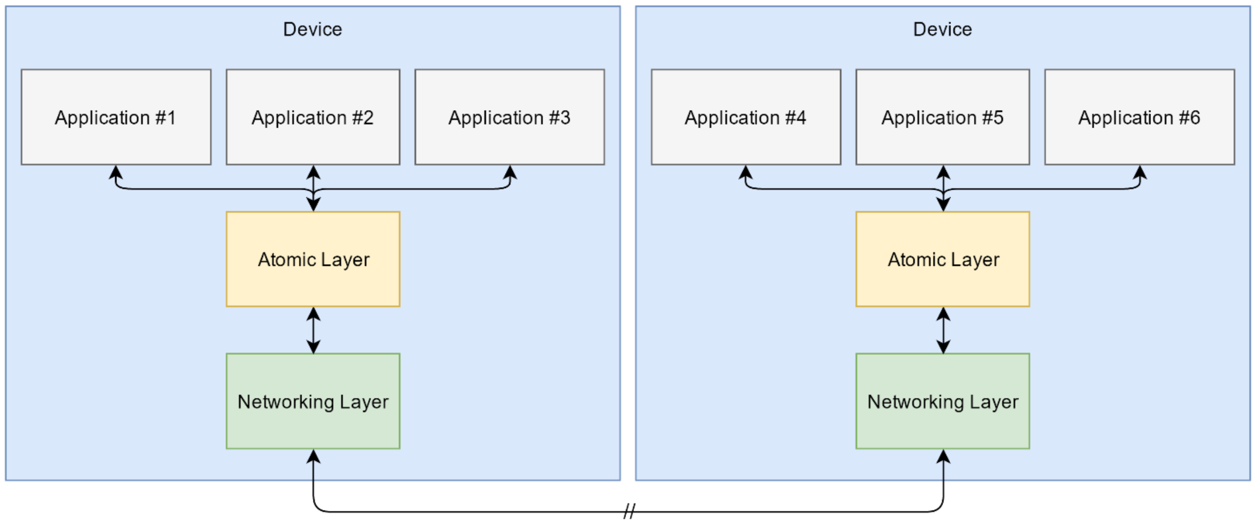
Functional Requirements

- A Device shall be able to host more than one Application
- Applications will be isolated from other processes on the Device
- Applications shall be able to communicate via Messages
- Messages shall contain origin, routing, Action (protocol), data, and stream information
- All Messages shall be routed by the Device
- An Application will be restricted to send/receive/broadcast Actions as defined in a configuration file available at system startup
- Devices shall synchronize configuration and user data with each other
- Messages shall be routed between Devices via a network
- A single Device failure must not bring down a multi-Device setup

Non-Functional Requirements

- The system shall be asynchronous (Non-blocking IO operations)
- Communication shall have minimal latency
- Communication shall have high bandwidth
- Fault Tolerance
    - Device failure shall be handled gracefully
    - Network shall attempt to reroute messages if a destination is unreachable

Proposed Design Method



Networking Layer

A full mesh connection between each atom which abstracts the routing. This allows messages to use the Any, Some, One, or All type for the destination.

Atomic Layer

Manages the life times and permissions of the applications on a device. At install of an application, the atomic layer reads the manifest file which contains information about the initial state of the application, including permissions. It stores the initial state and all further updates to it in a Redis database.

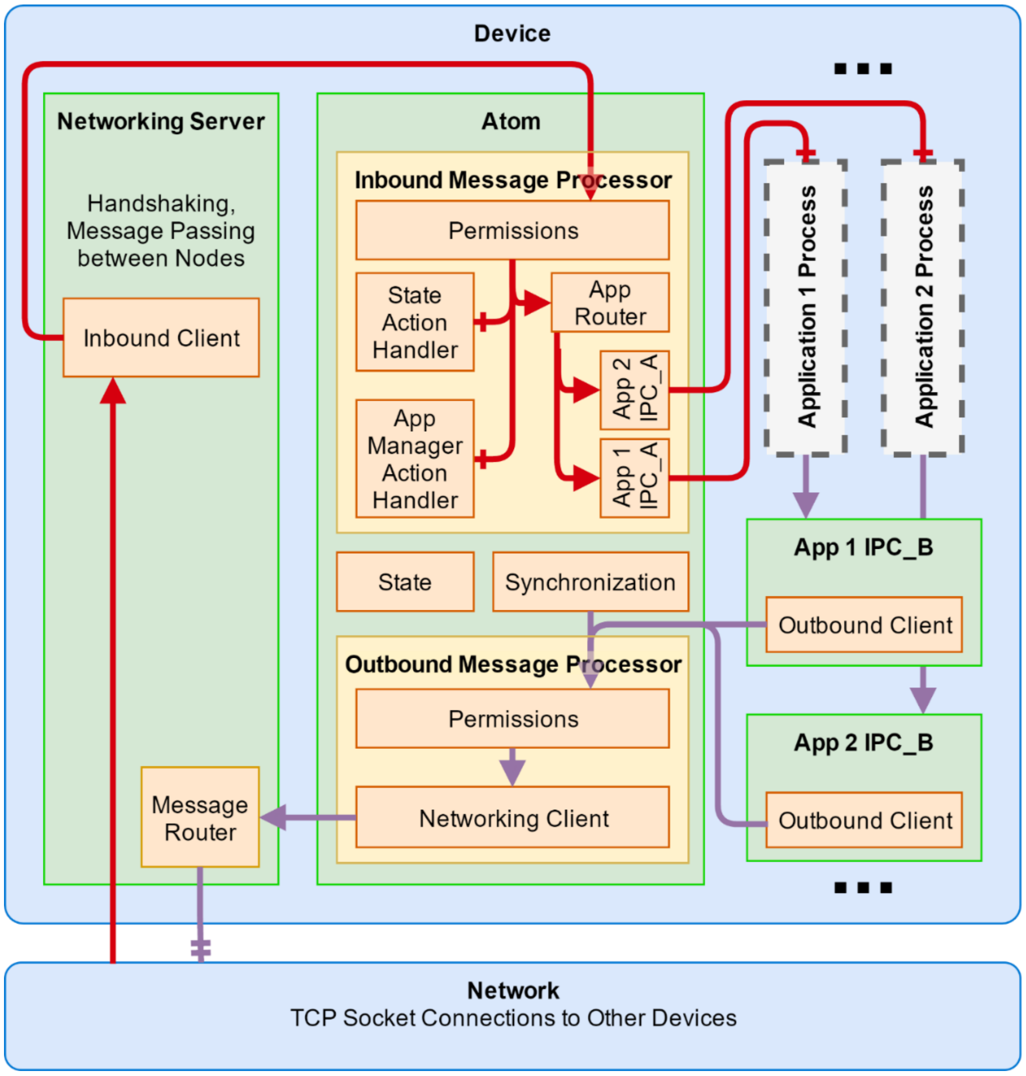
Application Layer

A set of applications to be developed by a 3rd party which can be hosted on a device. These applications provide functionality to the system, including interfacing with external devices. Examples include coffee makers, thermometers, and lightbulbs.

Message

Uses the Request-Response Pattern for internal communication – once a message reaches an endpoint, a response is sent back along the same route it arrived on.

Implementation Details & Design Analysis



Green boxes represent a thread
White boxes represent a process
Red arrows denote messages originating outside the device
Purple arrows denote messages originating inside the device

->	right before a block designates that the message will be processed and routed by that block
-|-	right before a block designates a terminal where a response is generated and sent back
-|-|-	before the network layer denotes that the terminal is in another device
...	denotes that an undefined number of application processes can be running at once

Modules

Permissions

The Permissions module ensures the contents of the message are within the allowed permissions list for the sender and receiver. This includes checking if the sender is allowed to use the specified routing type, checking if the sender is allowed to call the designated action, and if the receiver is allowed to perform the designated action.

State Action Handler:

The State Action Handler manages requests from outside the Atom to create, read, modify, or delete key-value pairs in the state.

Since the actions that modify state are handled in the atom, applications are unable to directly modify each other's state without permissions. Since permission is stored as part of the state, this also prevents applications from granting themselves or other applications permissions.

App Manager Action Handler:

The App Manager Action Handler manages the lifecycle actions and the Action Routing table in the state for an application. These lifecycle actions include install, start, stop, update, and uninstall.

By managing the lifecycle in the atom, applications become unable to modify the lifecycle of other applications. This creates isolation and better security.

Application

By providing a common API for messages and an IPC socket, a 3rd party is able to create applications that can communicate with the atom and thus provide functionality to the system.

State

The State stores information about each instance of an application as a key-value pair. This can only be accessed and modified by the atom. This allows the safe storage of an applications permission in the state.

Synchronization

The synchronization module ensures that each atom has the current state of each application. This allows applications that are running on multiple atoms to have a shared state.

Messages

A Message contains:

- **UUID**: The identification of the sender
- **Routing Type**: Any, Some, One, or All
- **Message Type**: declares whether message is a packet, synchronized, or a stream
- **Action**: the identifier to the destination for what to do with the data
- **Data**: information used to perform the action
- **Stream**: If the message is a stream, then this contains the address to retrieve the stream

Testing & Results

Interface Specifications

Networking Layer

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

This is accomplished by:

- Network to Network communication through TCP sockets
- Atomic to Network communication through pipes

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

Atomic Layer

The atomic layer sends messages for system functionality.
This is accomplished by:

- Atomic to Network communication through pipes
- Atomic to Application communication through pipes

Subcomponent tests check to make sure the correct information and quantity of it is being sent from the Atomic Layer to the other layers correctly. Tests have also been put in place to ensure only approved messages of the systems can be distributed throughout it, and they have proper clearance.

Application Layer

The Application Layer consists of the applications that provide functionality to the system.
These are passed in the following way:

- Atomic to Application communication through pipes

Applications have permissions to request actions from other applications. This is checked and tested in the Atomic Layer. This is on top of the tests needed to ensure data is being sent and received accurately.

Hardware & Software

We executed our software on Linux/Rust capable devices. The software used for testing the Networking Layer, the Atomic Layer, and the Application Layer is the basic Rust testing suite, which is included with the Rust compiler.

Networking Layer

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

Atomic Layer

Stub networking and Application Layers are used for the Atomic Layer to hook into during testing. Tests verify that the Atomic Layer properly passes messages between the Application Layer and Network Layer.

Application API

Since applications are developed by 3rd parties, the application API is what is tested. This was done by sending messages from one Application to another Application in the system. This message was spun around the system to test time outs and for dropped message checks.

Full System

Need to test the entire system integrated together and ensure all previous tests still return the correct values.

Process

Networking Layer

Test Fault Tolerance: Networking Layer tolerates faults and routes to other available devices. By terminating a device early, we can verify that other devices do not crash.
Test Service Discovery: Tested that the Networking Layer can find services and open connections to them
Test Authentication: Verified that devices will not start communication with unregistered devices.
Test Encryption: Test that data sent from a device is encrypted and cannot be read by a 3rd party.

Atomic Layer

Test Configuration: Data is properly stored and read.
Test Communication: Atoms properly communicate with one-another

Message Handles Test Cases:

- Message In, Message Out
- Outgoing Message for Application Routed to network layer
- Incoming Message for Application Send on to Network layer for Processing
- Futures

Test Permissions: Check to ensure only authenticated applications are allowed access to the functionality they request. Send both valid and invalid requests from applications and ensure they are handled correctly.

Application Layer

Test Lifecycle Events: Ensured correct outputs. Tested out of order Life Cycle event behavior.

Test Message Binding: Received Messages are appropriately bound to functions. Tested undefined message type responses.

Results

Network

Fault Tolerance

Network layer was able to properly route data to a backup application when the application it is communicating with fails. This was tested with a system composed of two file servers and a file browser. One of the file servers was turned off and the network layer automatically switched over to using the other.

Service Discovery and Authentication

Service discovery and authentication were tested at the same time as the file server, since the network needed all of these features in order for the server to work.

Encryption

Encryption was tested automatically with unit tests. The primitives used by the networking layer had automated tests run during every commit to the repository.

Additional Network Notes

During testing is was discovered that one of the underlying libraries had a bug that would cause the program to crash. This bug was due to a capacity overflow on an underlying storage object. The crash caused many problems that prevented the network layer from meeting its functional requirements. Working around the problem took quite a lot of engineering effort. After analyzing the code for the underlying library a fix was made and submitted to the original authors.

Another issue we encountered was that the Windows Subsystem for Linux had a bug with the filesystem that would cause the program to hang for no reason.

Atomic Layer

Atomic layer tests were entirely manual. Tests would be carried out by utilizing applications across the system and analyzing logs whenever an error occurred. A hello world application was made to test all the necessary routing functions, All-All, All-One, Any-One, and One-One.

Configuration storage was tested by analyzing the log outputs of the atomic layer, during this process it was discovered that occasionally the initial configuration would not be completely setup. The problem was addressed by making sure configuration setup commands happened in sequence instead of in parallel. This had the possibility of slowing down the initial load of the atomic layer, but since that only happens once it wasn't determined to be too critical.

Appendix I: Operational Manual

Project Molecule forms a backbone for communication between different devices and handles concerns such as networking, availability, and security. It is intended to be used by 3rd party developers and enthusiasts to link smart devices and create a connected home. This repository provides the tools to create, configure, and deploy 3rd party `Applications` and `Atoms` on the `System`.

Table of Contents

System Description

The `System` is comprised of several nodes called `Atoms`. `Atoms` manage all communication in the `System` and enforce permissions. `Atoms` also manage and run 3rd party `Applications`. These `Applications` run in separate processes and can communicate by sending `Messages` through the `System` using a provided API.

Messages

Communication uses a Request Response architecture meaning that a Sender will receive a response `Message` for each request `Message` sent. `Messages` contain routing and protocol data, a sized data payload, and an optional data stream. The optional data stream can be used to send large files or other types of data.

Actions & Payload

`Messages` implement User or `System` defined protocols called `Actions`. The `Action` type is stored in each `Message` and allows an `Application` The type of an `Action` is part of the header in a `Message` stored as a String in the `Message` to reason about the data that will be included in the payload and data streams. For example, when an `Application` receives a **"FILE"** `ACTION`, the message payload of a request will be the following enum:

```
pub enum FileAction {
    LIST(String),   //List all files in the directory name stored in the contained string
    GET(String),    //Get the file with the name stored in the contained string
}
```

The response payload will be dependent on the request. i.e:

- On a List request, the Response Message will return a Vector of Strings representing the file name in a given directory.
- On a Get request, the Response Message will contain the status of the request and a Stream containing the file data.

Routing

`Message` routing information is only processed on a request. This processing sets up a path for the data to return. Once a request has been received, the response follows this path back to the requestor. If an error occurs, the requesting `Application` is notified of the failure.

An `Atom` routes a request by analyzing the following `Message` parameters:

1. The `Action` protocol
2. An `Application` identifier filter
3. An `Atom` identifier filter

The Identifier filter is one of the following:

```
Any  - Resolves to one identifier
All  - Resolves to all identifiers
One  - Explicit identifier
Some - Explicit group of identifiers
```

Here are several examples of how a message would be for the given parameters:

Action	Atom	App	Description
T	ALL	ALL	Broadcasts message to all applications on all atoms which can receive `Action` **T**
T	ANY	ONE(id)	Routes a `Message` to an unspecified instance of the `Application`
T	ALL	ONE(id)	Routes a `Message` to all instances of the `Application`
T	ANY	SOME(ids)	Routes a `Message` to an unspecified instance of each `Application`

Applications

`Applications` are 3rd Party Binaries that use our `Messaging` API and are managed by an `Atom`. They have a manifest file which specifies:

- A globally unique identifier
- A unique human readable name
- Supported `Actions`

Permissions

A developer must declare the `Actions` an `Application` can send, receive, or broadcast in order for the `Atom` to route `Messages` and enforce permissions. If no `Actions` are specified, the `Application` will be effectively cut off from the `System`.

Extensibility

Applications allow the system to be extensible and connect many different devices. To connect a Smart Thermostat, for example, an `Application` could implement a **"THERMOSTAT"** `Action` which wraps calls to a 3rd party API.

Setup & Requirements

To use Project Molecule, you will need to setup a development computer and several host devices running Linux. The development computer must be able to compile binaries for Linux. You will also need to have redis installed on all of the host devices. For the purposes of this tutorial, we will assume the development computer is running Linux and that you have a basic understanding of the command line. The development computer will be used to create the `System Files` to be deployed on the host devices (`System`).

Development Computer

To generate the `System Files,` you will need to have rust and the molecule-runtime binaries installed on the development computer. The following steps guide you through compiling the molecule-runtime binaries for use in other tutorials.

1. Install Rust & Redis

www.rust-lang.org
www.redis.io

2. Install the Molecule-Runtime

1. Download the most recent release here.
2. Unzip the release.
3. compile the runtime.

Note: This overwrites the `./bin` folder.

```
./release.sh # This Script compiles molecule-runtime and outputs it to the ./bin folder
```

Tools Overview

This repository contains a command line utility called `generate` to create new `Application` projects, configure `Atoms`, and generate the necessary `System Files`. It will create a `./dat` folder to store development and other files necessary to produce `System Files`.

```
./generate.sh # This script runs the generator binary located in the ./bin folder
```

Usage

cmd	description
app	Create a project-molecule `Application` project
atom	Configure a new `Atom` by generating its `Atom Manifest`, `Device Manifest`, and its `Redis Manifest`
system	determine which `Atoms` will be included in a `System`
bundle	Create `System Files` to be deployed on host devices

Directory Structure

Applications are stored in two places. Projects are stored in the `dat/dev` folder while precompiled binaries are stored in `dat/sys/app`. The bundle command will look for apps in both of these folders when compiling an `Atom's` `System Files`. The `dat/boot` folder contains the `System Files` for each atom. This includes: the application binaries specified in the `Atom Manifest`, the `Device Manifest` for other host devices running in the `System`, and other configuration files.

```
bin/
    resources/                  # These are the default files used to generate applications an
d atoms
        .gitignore              #
        app.rs                  # hello world example
        log.yaml                # configures logging
        redis.conf              # configures redis-server [see documentation](http://download.
redis.io/redis-stable/redis.conf)
        release.sh              # Script to compile App
        setup.sh                # Add cargo to path
        db                      # starts database on host
        run                     # starts atom on host
    generate                    # system generator
```

```
dat/
    boot/
        [atom]/
            app/
                [app]/          # mirrors app/release folder
                    data/
                    .app
                    app.bin
                ...
            atm/
                [atom]          # Device Manifest
                ...
            .atm                # Atom Manifest
            .redis              # Redis Manifest
            .gitignore          # ~ from bin/resources/
            db                  # ~ from bin/resources/
            log.yaml            # ~ from bin/resources/
            redis.conf          # ~ from bin/resources/
            run                 # ~ from bin/resources/
    dev/
        molecule_app__[app]/    # Put App Projects Here
            release/
                data/           # Any app files/resources
                .app            # App Manifest
                app.bin         # Runtime compiled using release.sh
            src/                # App Source folder
                app.rs          # ~ from bin/resources/
            release.sh          # ~ from bin/resources/
            setup.sh            # ~ from bin/resources/
            ...
    sys/
        app/
            [app]/             # Put Precompiled Apps here
                data/
                .app            # App Manifest
                app.bin
            ...
        atm/
            [atom]/
                .atom           # Atom Manifest
                .device         # Device Manifest
                .redis          # Redis Manifest
            ...
        [system].sys            # Defines the atoms that are part of a system
```

Manifest Files

The generate utility can only be used to create new Manifest Files. To make updates, you will need to modify the manifest files directly. Make sure you do not make your changes in the boot directory as these files will be deleted and overwritten when you run the bundle command. All Manifest files are in the JSON format and can be found. A description of each manifest file can be found below as well as how the sys and dev files are mapped to the boot directory.

Application Manifest

(dev/molecule_app_[app]/release/.app, sys/app/[app]/.app -> boot/[atom]/app/[app/.app])

```
{
  "id": "",               // Uuid
  "name": "",             // Application Name
  "permissions": {
    "send": [""],         // Actions this Application can Send
    "receive": [""],      // Actions this Application can Receive
    "broadcast": [""]     // Actions this Application can Broadcast
  }
}
```

Atom Manifest

(sys/atm/[atom]/.atom -> boot/[atom]/.atom)

```
{
  "atom": {
    "name": "",           // Atom Name
    "network_id": ""      // Internal Uuid
  },
  "device": {},           // Device Manifest
  "applications": [""], // Applications which will run on this system
  "system": "system"      // Name of the System which this Atom is a part of
}
```

Device Manifest

(sys/atm/[atom]/.device -> boot/[atom]/atm/[other_atom])

```
{
    "id": "",             // Atom Id
    "address": "",        // IP Address:Port
    "sign_key": {},       // 128 bit signing key
    "cipher_key": {},     // 128 bit cryptographic key
}
```

Redis Manifest

(sys/atm/[atom]/.redis -> boot/[atom]/.redis)

```
{
  "config": "",          // Relative or Absolute path to redis.conf file
  "socket": "",          // Absolute path to redis socket as specified in redis.conf
  "db": 0                // Redis db number
}
```

System

(sys/[system].sys)

```
[""]                     // Names of Atoms in a System
```

Examples

1. Hello World
2. Client Server

The following examples assume you are running Linux and have the required tools and dependencies installed. In this case we will be using our development machine as a host device. In a real system, the boot files would be copied to the host devices and then run.

Hello World

To start out we will create the default Hello World `Application`. This creates a server which watches for ***HELLO*** `Actions` and responds with an arbitrary `Message`. It also sends out a ***HELLO*** `Action` when it starts up so we can see an output. First open a terminal and follow the steps below:

Create an Application

1. Run the generate script and select the `app` command.
2. Give it the name ***hello***.
3. When it asks for `Action` permissions add ***HELLO*** to the send, receive, and broadcast action lists.
4. When it finishes a new folder in the dev directory (`dev/molecule_app__hello`) will contain a default Hello World application.

Note: that Actions is case sensitive

Configure an Atom

1. Run the generate script and select the `atom` command.
2. You can use the defaults except for the redis socket. You will need to specify the following absolute path (/path/to/dat/boot/atom/redis.conf). At this point /atom/redis.conf will not exist as it is generated with the bundle command.
3. When you get to the apps section add the ***hello*** application to the list.

Configure a System

1. Run the generate script and select the `system` command.
2. Give it the default name and add ***atom`*** to the list of atoms in the system.

Bundle

1. Run the generate script and select the `bundle` command.
2. Bundle the system named system

Note: bundling a system prepares a boot directory for all of its atoms. If you would like to only bundle a particular atom, you can also specify just the atom's name.

Code

Before we run the example let's take a look at the code. It is located at `dev/molecule_app__hello/src/app.rs` This example uses the `futures` and the `tokio_core` crates to handle asynchronous communication. We also use several modules from molecule_common to communicate with the host `Atom`. Namely we use the ***Messaging***, ***Reqes*** (**Req** uest / R ***es*** ponse), ***Error***, and ***Application*** modules. The ***Messaging*** Module provides utilities to serialize and deserialize message data and create responses. The ***Reqes*** module is used to send and receive messages from the host `Atom`. The ***Error*** Module defines a custom error and result type. Finally, the ***Application*** Module sets up communication with host `Atom`.

```
extern crate tokio_core;
extern crate molecule_common;
extern crate uuid;
extern crate futures;

use futures::{BoxFuture, Future, Stream, future};
use molecule_common::app;
use molecule_common::err::*;
use molecule_common::message::*;
use molecule_common::reqes::*;
use tokio_core::reactor::Core;
use uuid::Uuid;
```

The following code segment starts an event loop core and sets up communication with the host `Atom`.

```
fn main() {
    let mut core = Core::new().unwrap();
    let reqes: Reqes<Message> = app::start(core.handle()).unwrap();

    ...
```

Here we setup a Server to process incoming `Messages` and generate responses. We are only looking for ***HELLO*** actions and expect the payload to be a string. After setting everything up, we spawn it in the event loop.

```
...

 let handle = core.handle();

//Listen For Messages
let in_server = reqes
    .server
    .receive()
    .and_then(move |mut req| -> BoxFuture<(), Error> {
        //Get Message
        let msg: Message = req.take_data().unwrap();
        //Determine the Action
        let action: String = msg.get_action().into();
        //Process the Actions
        let result: Box<Future<Item = Message, Error = Error>> = match action.as_ref() {
            "HELLO" => {
                //Get Message Data
                let data: Result<String> = msg.get_data();
                let data: Result<&str> = match data {
                    Ok(ref s) => Ok(s.as_ref()),
                    Err(e) => Err(e),
                };
                //Determine Response
                match data {
                    Ok("hello") => {
                        let response = msg.response()
                            .data(Ok("hello to you too!"))
                            .trace("HELLO_WORLD", Uuid::nil())
                            .build_packet();
                        Box::new(future::ok(response))
                    }
                    _ => {
                        let response = msg.response()
                            .data::<()>(Err(Error::new(ErrKind::Other,
                                                       "I only understand one word")))
                            .trace("HELLO_WORLD", Uuid::nil())
                            .build_packet();
                        Box::new(future::ok(response))
                    }
                }
            }
            // If you support more actions, implement them here
            _ => unimplemented!(),
        };
        //Send Response
        req.reply(&handle, result);
        future::ok(()).boxed()
    })
    .map_err(|_| ())
    .for_each(|_| -> BoxFuture<(), ()> { future::ok(()).boxed() });

// Run Server in Event Loop
core.handle().spawn(in_server);

...
```

Here we setup a Client to send a ***HELLO*** `Message` when the `Application` starts. It then waits for a response from the server. In a future example, we will run the client and server in separate applications. After setting everything up, we spawn it in the event loop.

```
    ...

    //Send a Message Somewhere
    let request = reqes
        .client
        .send(MessageBuilder::new("HELLO")
                  .source(Signature::broadcast())
                  .destination(Signature::broadcast())
                  .data("hello")
                  .unwrap()
                  .trace("HELLO_WORLD", Uuid::nil())
                  .build_packet())
        .map(|msg: Message| {
                 println!("I received a message {:?}", msg);
             })
        .map_err(|_| ());

    // Run Client in Event Loop
    core.handle().spawn(request);

    ...
```

If we wanted to setup another future, we could do so here before we start the event loop. Once everything is set up we start the event loop which blocks the thread. For more information on tokio and futures see the documentation here.

```
    ...

    // Start Event Loop
    loop {
        core.turn(None);
    }
}
```

Deploy

1. In a terminal, change directories into the dat/boot/atom folder
2. run `./db`
3. In a second terminal, also change directories into the dat/boot/atom folder
4. run `./run`

If everything went well, the `hello` app should output something like the following:

```
I received a message Message {
        id: f02d55b3-4253-4620-9784-052d7c38962d
        routing: None
        msg_type: Packet
        action: HELLO
        data: Ok([{"Ok":{"id":"f02d55b3-4253-4620-9784-052d7c38962d","routing":null,"msg_type"
:"Packet","action":"HELLO","data":{"Ok":"\"hello to you too!\""},"trace":{"trace":[["HELLO_WOR
LD","00000000-0000-0000-0000-000000000000"],["Atom::run()","22945895-7a17-4266-8dbc-3af1650306
6b"],["Atom::incoming","22945895-7a17-4266-8dbc-3af16503066b"],["HELLO_WORLD","00000000-0000-0
000-0000-000000000000"]]}}}])
        trace: [ (HELLO_WORLD   00000000-0000-0000-0000-000000000000) (Atom::run() 22945895-7a
17-4266-8dbc-3af16503066b) (   00000000-0000-0000-0000-000000000000) ]
}
```

Hello World Client Server Example

To make this example more interesting let's create a new `System` with two applications named `hello_client` and `hello_server`.

Create the Applications

Follow the same steps as before but use `hello_client` and `hello_server` as the application names. Once you have made both apps navigate to the source code. In `hello_client`, remove the Serve code. In `hello_server`, remove the client code. If you need clarification on what code to delete, review the previous section describing how the code works.

Configure an Atom

Next create two new atoms named `atom_client` and `atom_server`. Follow the same steps as before with the following modifications:

1. You will need to choose two different ports for the network socket.
2. The absolute path to the redis.conf file will be (`/path/to/boot/[atom_name]/redis.conf`).
3. In the apps section add `hello_client` to `atom_client` and `hello_server` to `atom_server`.
4. Set the `System` name to `two_atoms`

Configure the System

Next rerun the system command. Name the system `two_atoms` and add both `atom_client` and `atom_server`

Bundle

Finally Run the bundle command on the `two_atoms` `System`. Your two atoms are ready to deploy.

Deploy

To Run the atom's, we will need to open several terminals:

1. In the first terminal change directory into dat/boot/atom_server/ and run the `./db` binary.
2. In another terminal go to the same directory and run the `./run` binaries.
3. Do the same for the atom_client

Note: if you run the client before the server, the example will not work. The client does not wait for the server to be running in the current implementation.

If everything went correctly, you should see the same message as before in the Atom Client.

Recap & Further Exploration

This example demonstrates how a message can be broadcast and how apps on different devices can respond to a message. For further exploration, try adding a third `Atom` running the `hello_server` app. The `hello_client` should see two responses to its hello world message.

Appendix II: Alternative Designs

Storage and Handling – Permission

Initially, permissions were handled on the fly in the networking layer by accessing the manifest files. We realized that this was cumbersome on the system. The solution was to store the permissions in the state. This allowed the system to access permission in the Atomic Layer before the messages were forwarded and reduce load on the system.

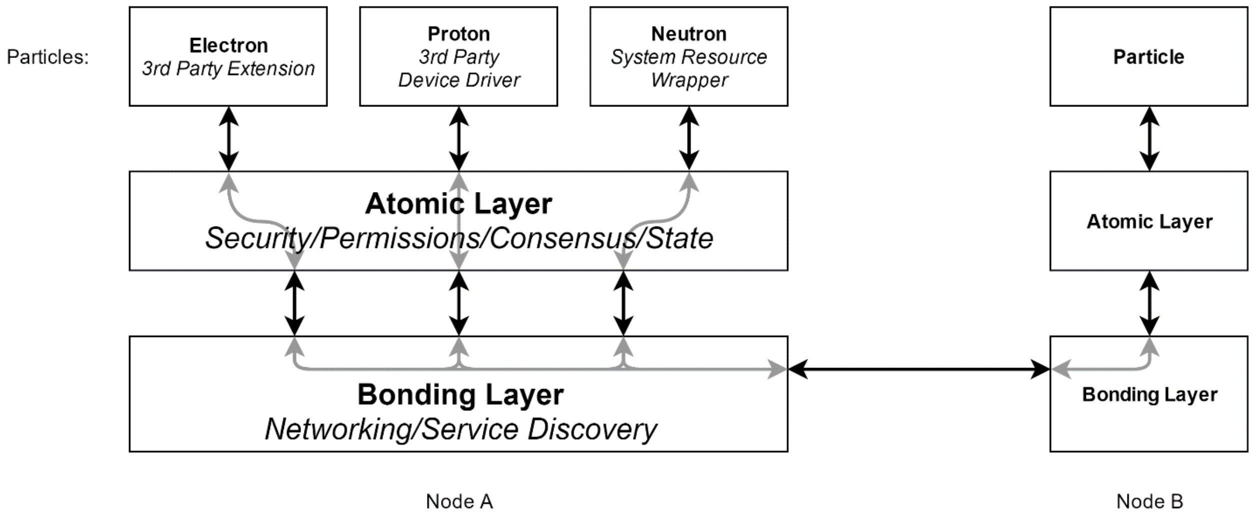
Change of definitions – Bonding and Particle

Originally the Application Layer was called Particle Layer and the Networking Layer was called the Bonding Layer. While these names served to create a theme, they created confusion in explanation. As a result, only the Atomic Layer retains the Molecule theming.

Reassessment of Features – UI Particle

Originally in addition to our project, we wanted to develop a UI Markup language that would allow for developers of particles to conveniently create UIs. But during the development of the project, we realized it was not in the original scope. Since we started to run into delays, we decided to reprioritize it. This freed up development time to complete main parts of the project.

Particles Types – Proton, Neutron, and Electron



The original design for this project had three different types of particles – Electron, Proton, and Neutron. Ultimately, we removed this distinction because each particle's implementation did not have different enough requirements for the API.

Appendix III: Other Considerations

Workload

During the development of this project, we had a few issues with distribution of workload. Bottlenecks would be created from the task distribution and pieces of work would become obsolete after a reassessment of the solution. Also, estimations on the work required would be wrong and create down time for some members and over time for others.

A well-defined agenda before starting a meeting would have helped pace the meetings to be able to give all team members time to ask questions as well all pressing matters are discussed in full.

To solve this during the project, we started to take better notes on weekly goals and spent more time on discussing the design before assigning work. This helped to work through the backlog of work.

Scope

Over the course of the project, the scope was not always clearly defined for specific parts. This lack of consistency in definition caused issues when developing these parts because different interpretation would arise. This led to confusion and prolonged discussions over what was a goal.

One such example of this is the UI Markup Language. Originally, we thought it would be convenient to have when we started to develop demo applications. It was cut after we reassessed the required components for the final deliverable. In future projects, we would build out from the center of the project instead of starting development on parts not directly contained within the scope.

Next Steps

There are a few pieces of the project that we were not able to finish by the end of the project period. The two components we were unable to complete in time were the dynamic UI application and state synchronization. While neither of these components saw the light of day, their requirements heavily influenced the design of the system, meaning that it should not be difficult to add them at a later date.

Not having these two components meant that we had to implement some workarounds in order to get a working demo for our project. We created a stateless demo application to avoid the problem of sharing state. In order to have a UI for the demo, we implemented a far simpler static UI that was specific to our demo application instead of a generic UI application that could be used by any application.

Were this project to continue, the next component to add would be state synchronization. Having state synchronization written into the core system would allow applications to be fulfill state requirements without having to write their own synchronizing code.

The UI application is not necessary for the core functionality of the system but would provide a more cohesive user experience. After the core system had stabilized to a point that applications would no longer experience breaking API changes, we would begin work on the dynamic UI.

Appendix IV: Code

Do to the complexity of our code base, it is not included in this document. Instead, you can find it in the following repositories:
Core Library
https://gitlab.com/may1739-molecule/molecule-common
Configuration and Runtime Utilities
https://gitlab.com/may1739-molecule/molecule-runtime