Dynamic Agents
Architecture High Level Description

Anson Fan
afan1@jaguarlandrover.com
Magnus Feuer
mfeuer@jaguarlandrover.com

Revisions

Revision	Date	Author	Notes
1	2015-08-11	AF	Created

Table of Contents

Contents

1. References

[1]	Remote Vehicle Interaction: https://github.com/PDXostc/rvi_core
[2]	Remote Vehicle Interaction Backend: https://github.com/PDXostc/rvi_backend
[3]	Remote Vehicle Interaction Software Over the Air: https://github.com/PDXostc/rvi_sota_demo
[4]	MongoDB: https://www.mongodb.org/
[5]	collectd: https://collectd.org/
[6]	Graphite: http://graphite.readthedocs.org/en/latest/

2. Acronyms and Definitions

[1]	RVI	Remote Vehicle Interaction
[2]	SOTA	Software Over The Air
[3]	IVI	In Vehicle Infotainment
[4]	Node	A vehicle, mobile device, or backend server running RVI services
[5]	Service	An external application connected to the RVI system
[6]	Service Name	A global registered name used to identify a specific service
[7]	Agent	A standalone running process that runs silently in the background
[8]	Client	RVI enabled device that the an Agent will be sent to
[9]	Backend Server	A RVI enabled cloud database which will handle deployment of all Agents

3. Introduction and Purpose

This document introduces the Dynamic Agents architecture on a high level, outlining its design, the components, and the core use cases. It is intended for the reader who wants a comprehensive overview of the Dynamic Agents and its inner workings. Once this document has been read, the reader can continue with the DDS(see below) for specific components to understand their implementation requirements.

The purpose of Dynamic Agents architecture is to allow for a smarter connected car in the sense that it isn’t locked down by pre-existing rigidly defined tasks, but has the freedom to run code independently of all other processes. This disconnect from the rest of the IVI or system that RVI is deployed makes it so that users will not have to interact or be affected by these agents at all.

A core feature of these Dynamic Agents is that they will be able to be deployed over the air to any vehicle in a very similar fashion to that of SOTA. The core differences however is that a user will never have to interact with the Agents since they will be remotely deployed and monitored without interruption to any of the user's services.

An example of a simple agent is a geofence agent which constantly checks GPS signal on the local code and if the device’s location goes out of bounds of the perimeter, the agent will initiate full logging of its location back to the RVI server and send out a notification to a predefined endpoint.

Dynamic Agents will be a feature built on the RVI framework, please refer to the RVI architecture documents to understand how communications are handled between RVI nodes.

The Dynamic Agents architecture will describe a method for which any RVI enabled device will be able to receive and run an agent on demand.

4. Requirements and Objectives

The objective of the Dynamic Agents is to provide an end to end solution to deploy agents, any form they may take and run completely separately from any mission critical modules. It is aimed to provide low impact to the entire system while operating as needed.

The following are the objectives in which Dynamic Agents attempts to achieve.

4.1. Agent Deployment

Dynamic Agents must be delivered to the clients in a very similar way that we deliver SOTA packages. The Agents must be delivered to a client in the background and must take into account limited network availability and check that the Agent is not corrupted on the end user client side.

4.2. Independent Virtual Environments (Python)

Dynamic Agents should not interfere with any of the other working parts of whatever it is connected to. Therefore it should in a sense run in a separate virtual machine or environment where all of the Agent’s required dependencies are met and will not interact with any of the other processes running in the system.

4.3. Universal Logging and Alerts

Given that Dynamic Agents may all serve very different purposes and return many different data values, an Agent logging service must be available that can conform to any data that may roll in. This also ensures privacy of sensitive car related information to not be redirected out to where it is not supposed to go.

In addition to just logging the data, we will further process the data on our backend server to raise alarms as needed. If the data being logged into our backend server shows signs that the agent is logging in values that are outside normal operating ranges, it will send an alert to the Agent owner who can then dig deeper into all the sent logs.

4.4. Accurate Remote Diagnostics

A main idea of these Agents is to create a method in which we will be able to

perform low cost car diagnostic tests ranging from a single car to a whole fleet consisting of hundred of thousand cars. This also will allow for car owners to experience uninterrupted car usage even while performing diagnostic testing.

4.5. Future – Resource Management

Dynamic Agents must manage its own resource usage given a predefined amount depending on what is available to the whole system. Therefore there must be an upper limit to the hardware resources that the agents can access. The Agents must be optimized and conform to set runtime specifications in order to avoid system-wide complications.

4.6. Future – Language Independent

We want to allow for the maximum flexibility in what the Agent can do, that means that certain programming languages may better accomplish a task than another. Therefore Agents will be allowed to written in any language as long as there is a package to compile it on the available hardware.

5. Architecture Overview

There will be two Agent servers available, Agent_Backend that will be integrated with RVI Backend server systems and also Agent_Handler which will exist on the client side. Both will be permanently running as long as the system is powered on the client. This means that even if the local RVI node is not found Agent_Handler will still be running and creating a local store for all the data to transmit after the node is found again.

The components are described in the following chapters along with a visual representation of the overall system path.

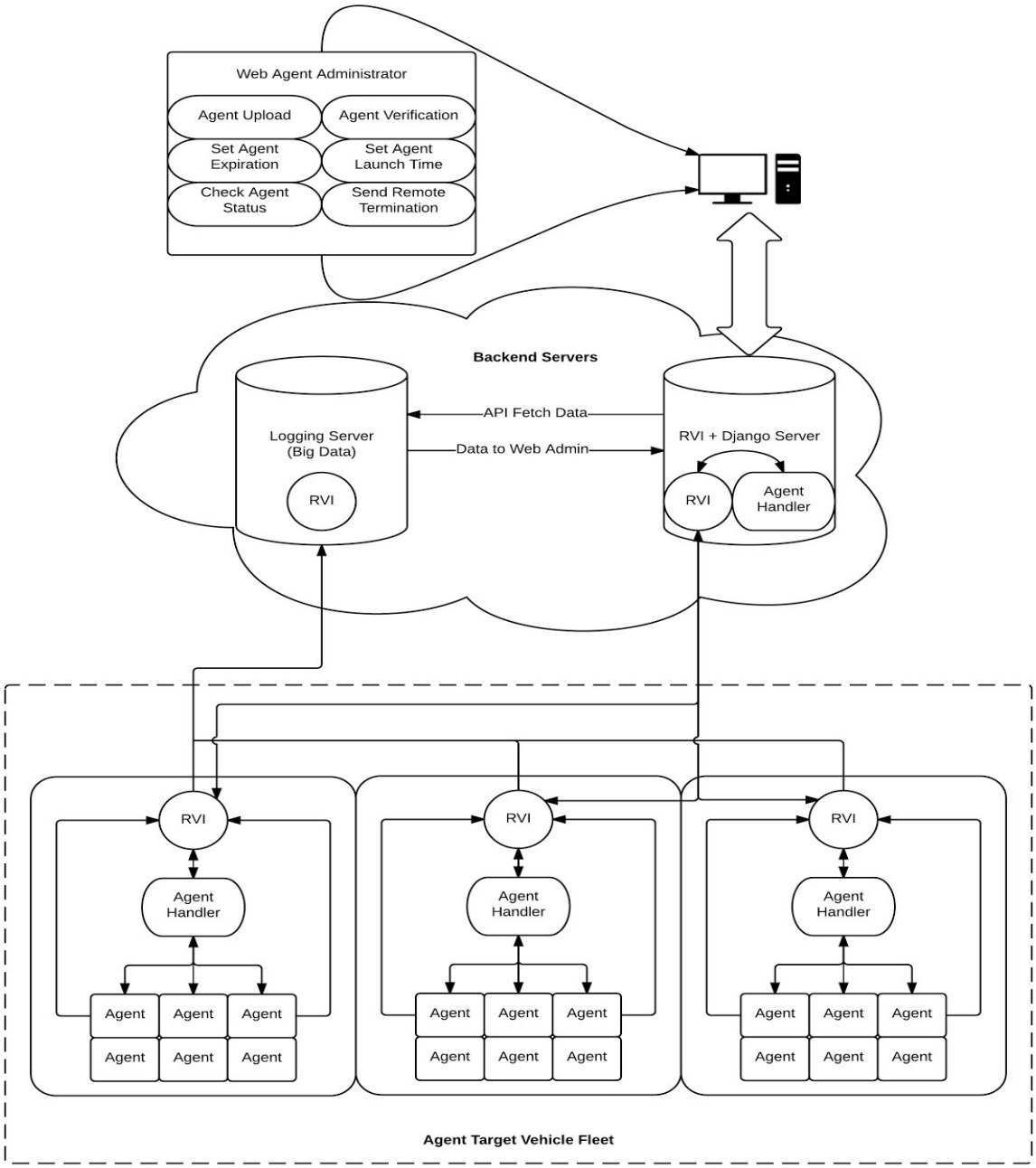


Fig 1.

The diagram above is a typical flow of events that will span a Dynamic Agent's lifetime and how it interacts with RVI.

Event Flow

5.1. Agent Upload

The Agent owner must upload their Agent into a single executable file. For the first few iterations we will assume this to be a self contained Python script. In future iterations there may be docker deployments to allow for any language to be used.

5.2. Set Agent Settings

The Agent owner must select their target vehicles along with the desired run time. This can become more customizable in the future.

5.3. Agent Packaging

The backend server will create an Agent Package which can then be sent to the remote client. This step will also test the script to make sure that it is able to run on the target.

5.4. Agent Deployment

Launching the Agent will create corresponding mapping for all the Agent and devices that it is deployed on. This will be what our main monitoring method to check the status of Agents at any given time.

This will initiate the whole download process of the Agent onto the Client. The Client will then return service calls to the Server until it completes the Agent download.

5.5. Client Receives Agent

After the client has successfully received the package it will attempt to start the Agent. Then it will trigger another service call to the server to update the table mapping with the Agent’s start status.

5.6. Register Clients with Agent Handlers

Register the new Agent on the local Client side mapping. This will help the client detect if the Agent goes down when it isn’t supposed to and terminate the Agent when it hits its expiration date. Also keeps a running tab on all active Agents.

5.7. Agent Runtime

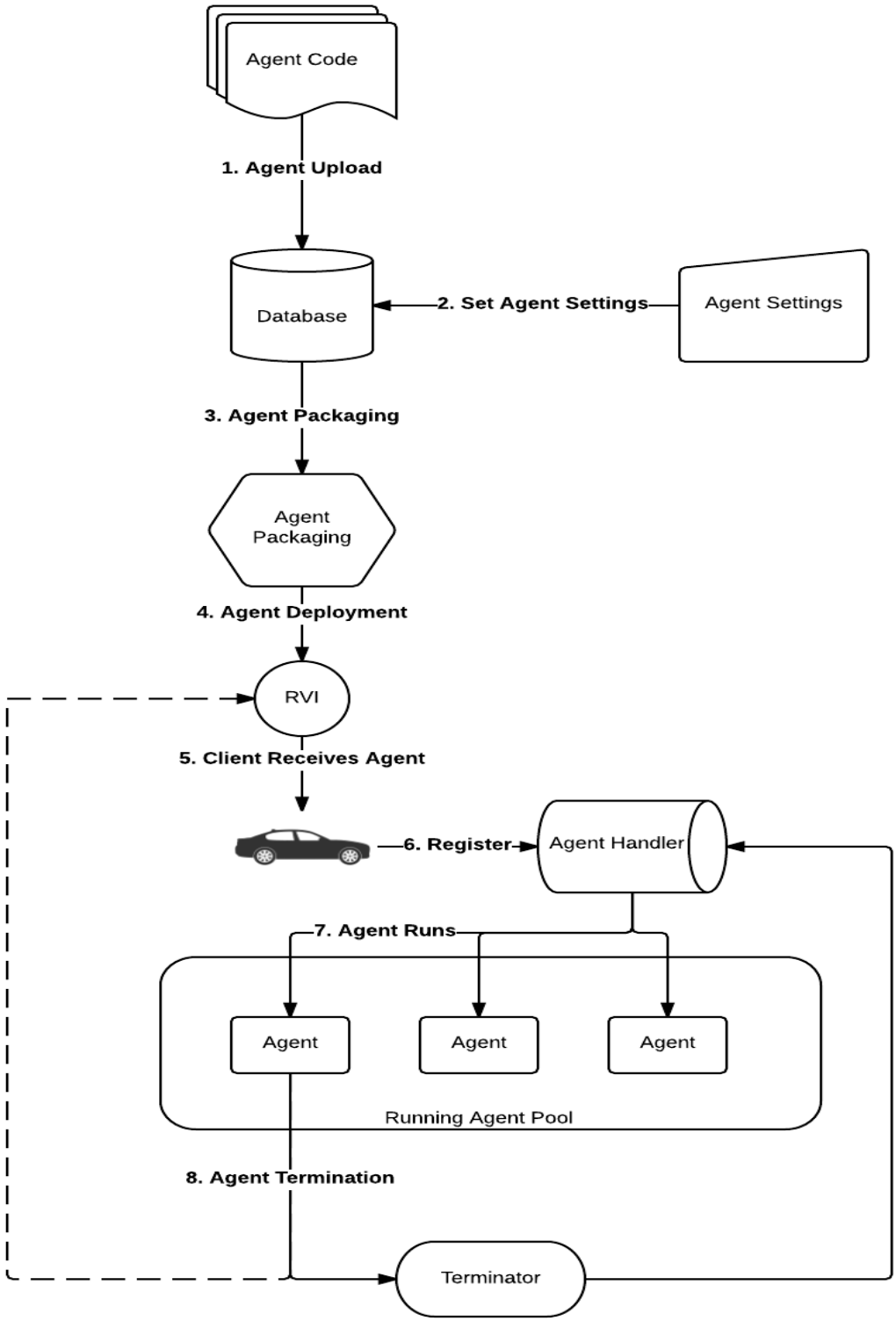
Agent will keep running in the background of the system. If it does lose connection to RVI or RVI loses connection to the server, it will create local logs that will be sent

later when connection is restored. The Agent should always have access to RVI backend/logging and backend/System_Notify.

5.8. Agent Termination

Agent will either be remotely terminated or hit its expiration date. In the event of either it will send the shutdown signal to the process that the Agent is running on, send a signal to the server and delete all relevant information that isn’t needed anymore including its mapping on the client side Agent Mapper.

Agent Event Flow Diagram



6. RVI Integration

To successfully deploy Dynamic Agents, there must be a few rigidly established service calls to RVI to ensure that there is adequate functionality for all of the Dynamic Agent’s features.

This section will give an overview description on what each service call will perform. All service calls will be bundled into what is known as the Agent Handler. There will be an Agent Handler on the server and also on any client that will have an Agent deployed on it.

Since RVI messages are all in JSON we will be packaging all of our communication in JSON strings as well. This means that some messages will have to be converted to base64 to ensure that we do not lose any data. This will also help the JSON parser avoid problems with nested quotations and such.

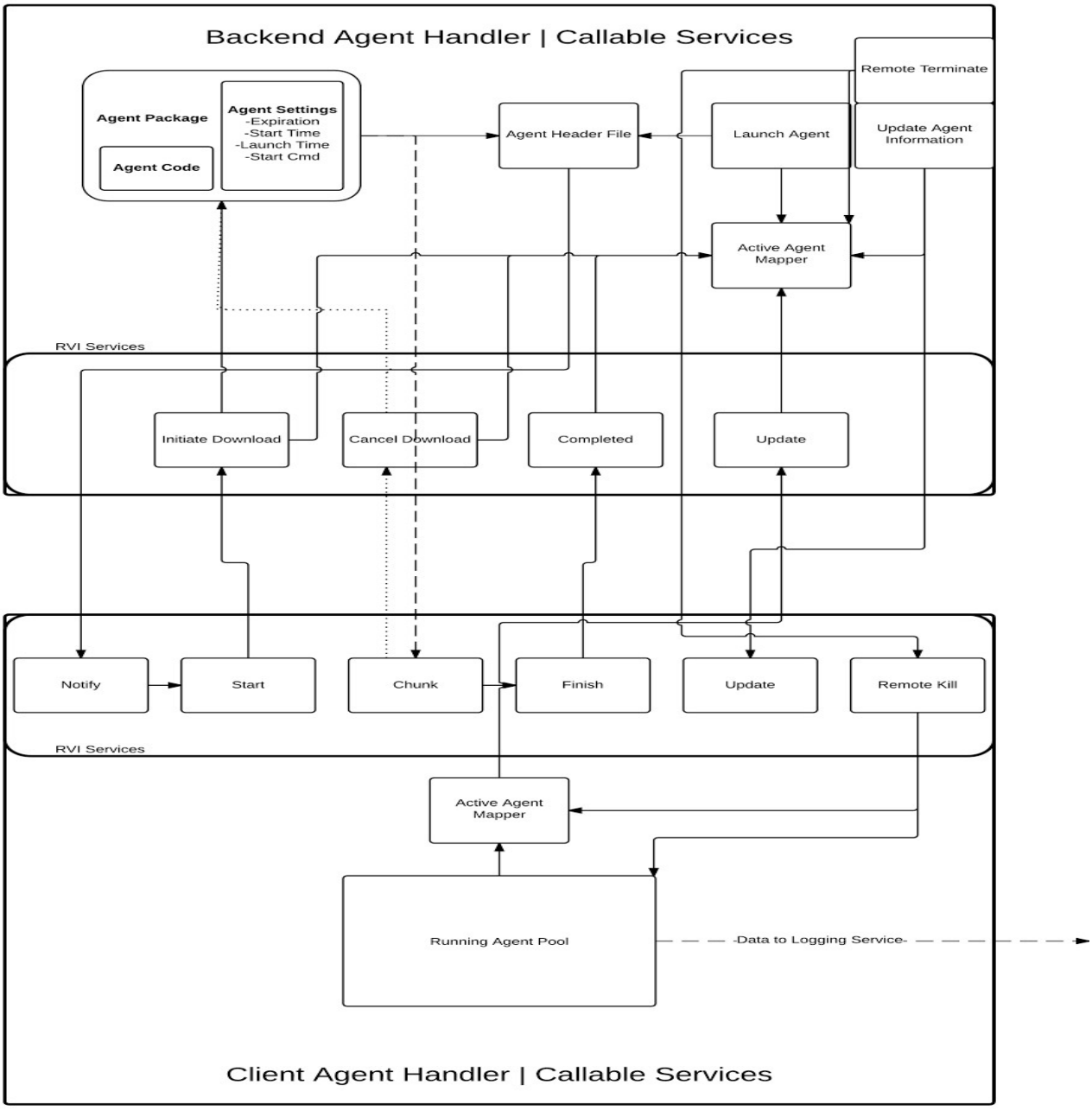


Fig 2.
The following diagram gives a general outline of the service calls and how they will be accessed by each side.

6.1. RVI Services – Server Side (Agent_Server)

6.1.1. Initiate Agent Download

Receive signal that client is ready to receive agent and will open Agent code and start sending chunks.

backend/agentserver/initiate_download

initiate_download(agent_id, retry, destination)

- agent_id (string)= specific agent
- retry (unsigned int)= retry number to install agent
- destination (string)= client id to send agent to

6.1.2. Cancel Agent Download

Will break the specific Agent's process from continuing to send data to the client.

backend/agentserver/cancel_download

cancel_download(agent_id, retry)

- retry (unsigned int)= retry number to cancel
- agent_id (string)= specific agent

6.1.3. Agent Download Complete

Will receive completion message when client has installed and tested code runtime.

backend/agentserver/download_complete

download_complete(status, retry, destination)

- status (bool)= True | False return value of script start
- retry (unsigned int)= retry number to install agent
- destination (string)= client id that agent was sent tohttp://ubuntuforums.org/showthread.php?t=2197601

6.1.4. Logging – Agent Report

Will receive all data if any from clients and redirect to the proper database channels if needed.

backend/logging/agent_report

agent_report(vin, agent_id, timestamp, payload)

- vin (string)= same as destination, vin of client or device_id
- agent_id (string)= specific agent
- timestamp (unsigned int)= unix epoch of local client time, preferably in UTC
- payload (string)= Array of JSON formatted such as : [{name:*** , counter:***, gauge:***}, {name:*** , counter:***, gauge:***}, {name:*** , counter:***, gauge:***}, ...]. Will need some b64 encoding to make less painful. **These JSON strings however can contain anything you want as long as they are valid JSON format. This is due to the fact we will be using a NoSQL database.**

6.1.5. Update Agent Status

Will receive alerts from the client that will send out some other form of notification to a systems administrator/agent owner.

backend/agentserver/status_update

status_update(agent_id, status, vin)

- agent_id (string)= specific agent
- status (unsigned int)= 0 Clean Exit | 1 Running | 2 ....
- vin (string)= same as destination, vin of client or device_id

6.2. RVI Services – Client Side (Agent_Handler)

6.2.1. Notify

Will receive a message to prepare the client to be ready to accept a new Agent and make sure there is enough space, will either call Initiate or Cancel.

vin/agenthandler/notify

notify(agent_id, expiration_date, launch_cmd)

- agent_id (string)= specific agent
- expiration_date (unsigned int)= unix epoch in UTC to save to memory
- launch_cmd (string)= string command to launch the script

6.2.2. Start Agent Download

Will be the first target of the Server side pushing data over to the client which will create the file in the specified directory and keep it open for the Chunks.

vin/agenthandler/start

start(agent_id, chunk_size, total_size)

- agent_id (string)= specific agent
- chunk_size (unsigned int)= size of chunks being sent to calculate where to add in code to our file
- total_size (unsigned int)= total size of the whole agent

6.2.3. Send Agent Chunk

Will receive chunks of the Agent until it is complete.

vin/agenthandler/chunk

chunk(agent_id, index, msg)

- agent_id (string)= specific agent
- index (unsigned int)= the index number of the block of code being received to properly calculate the offset to determine where the message chunk will go into the file
- msg (string)= the actual code that will be added into the agent’s file on the local store. Will need to be set to b64 encoding most likely.
-

6.2.4. Finish Agent Download

When the final chunk of the Agent is received will run the Agent and make sure that no errors are thrown. We will then send the success or failure of the launch code to the Complete_Agent_Download.

vin/agenthandler/finish

finish(agent_id)

- agent_id (string)= specific agent

6.2.5. Terminate

Will receive a message to terminate a specific Agent before its expiration date. This will free up all of its resources tied to the Agent.

vin/agenthandler/terminate

terminate(agent_id)

- agent_id (string)= specific agent

7. Agent Handler – Server Side

The server side Agent Handler will be in charge of packaging and deploying Agents. It will also be in charge of keeping track of all Agents, their expiration date, and any status changes.

7.1. Agent Packaging

The Agent Packaging on the server side will consist of a few different components in an attempt to set a global standard that will allow any Agent to be ran on any RVI enabled device.

The Packaged Agent will be represented in a header such as this JSON string and all values will then be stored in the client's memory:

{

"AgentName": * ,

"MD5/AgentSize": * ,

"ExpirationDate": * ,

"RequirementsFile": * ,

"LaunchCommand": * ,

"SuccessCode": *

}

This package will then be sent to our client as a form of a Header file before sending over the actual Agent.

7.2. Agent Mapper

Agent Mapping will be a simple SQL accessible database with the following columns to keep track of the Agents that are distributed to all of the available clients.

[1]	**AgentName**	The name of the Agent and any corresponding version number
[2]	**AgentOwner**	The Owner of the Agent to determine who to notifty when alarms are triggered
[3]	**ClientVIN**	The VIN of the device that our Agent is deployed on or device ID
[4]	**ServerLaunchTime**	The date and time that the Agent was deployed from our server
[5]	**ClientLaunchTime**	The date and time that the Client started running the Agent
[6]	**ClientStatus**	The signal code that was last sent from the Agent to
[7]	**LastMessageReceived**	The last timestamp received through the backend logger
[8]	**Alarm**	Will be triggered to true if any monitoring system is in place parsing logger data for the agent.

AGENT MAPPER SQL COLUMNS

7.3. Agent logger

Agent logger will be a running Python server that will feed through all of our data into a noSQL MongoDB database. This will centralize all data to ensure data privacy and security which can then be sent to the appropriate channels for processing.

The messages that come in must come in as a very generic form, but since our infrastructure is set up around JSON messages it is very simple to create added layers of data in just one simple rigid form. The most basic form that our Logger will expect will be as shown below:

{

“Agent”: *,

“VIN”: *,

“TimeStamp”: *,

“Payload”: [ ] or another JSON object for added layering / data complexity

}

8. Agent Handler – Client Side

The client side Agent Handler will be in charge of making sure that the Agents have adequate channels to RVI logging and also that the Agents get restarted if they are not expired yet.

8.1. Agent Receiver

The Agent Receiver will be a separate permanently running process that will wait for a message to wake it up. This will initiate the whole Agent download process and make sure that the Agent was sent correctly and runs without throwing any exceptions on the target. Will add the newly formed Agent to the local client side Agent Mapper upon success of launch.

8.2. Agent Removal

After an Agent reaches its expiration date or the client is sent a remote termination signal. The Agent Removal system must find all logs and processes that may have been spawned by the Agent and delete them (May want to push logs to logging if not done before if applicable). Will delete all relevant entries of the Agent out of the Agent Mapping and also send a signal to the backend server that it has completed deletion of all of its data and it’s reason for termination.

8.3. Agent Mapper and Tracker

Agent Mapping + Tracking will be very similar to that of the resource manager, however this will run in memory to make sure that our Agents are up and running. The main purpose of this is to make sure that the processes that the Agents are running up are alive and in good status, if not it will relaunch with a new process. This mapping will also keep an eye on execution of the Agent and its expiration date to send more signals to the system to remove the Agent if it is expired.

8.4. Agent Recoverer

A critical function of the Agent Handler is to make sure that the Agents are all running while they haven't reached their expiration date. Therefore it will ping/monitor the Agent's process and if it goes down for any reason before it expires it will relaunch the Agent as needed.

9. RVI Backend Server Logging

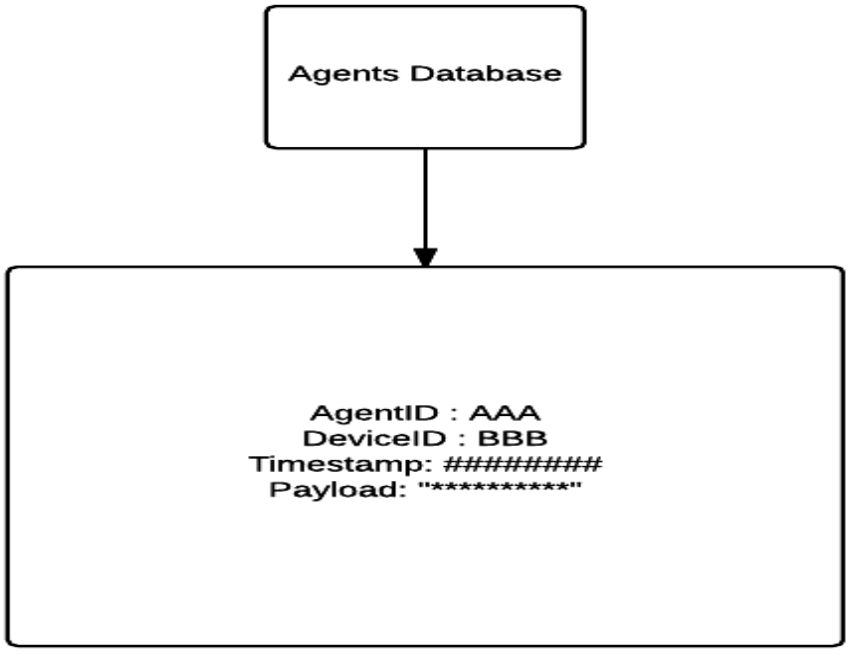
The logging service for the Dynamic Agents will be built on a MongoDB noSQL database. This is due to the fact that the data that the Agents send back may not always be in any exact format. Therefore we need a database system that can conform to any of the data while using the same function calls. MongoDB is chosen for its straightforward low level to entry but in theory the only requirement for the logging database is the ability to accept JSON as a blob type.

We could also use the big data platform utilizing a Kafka Message Queue with Apache HBase. For our test purposes we will not be generating that heavy of a load of data, therefore mongoDB should suit our needs well enough.

We will want to index out our timestamps for faster queries.

Work In Progress:

Will start with a complete data dump and only index timestamp for a little bit better performance. In future we may pre-parse the payload and layer the documents more.



10. Agent Formatting and APIs

The Agent Handler on the client side will be defined as a Python script that will continuously run called the AgentHandler. For Agents to be written easier, we will provide some simple APIs for the Agent to include when being written so it does not have to handle RVI calls.

Doing this means that the Agent must have some basic requirements when writing an Agent.

10.1. Agent Structure/Formatting

- The Agent must import functions from the AgentHandlerAPI if it wishes to use the RVI services.
- The Agent must be executable file must be named as the AgentID for easier book keeping.
- There must be a main execution loop that returns a value on successful launch of the Agent.

For the time being the AgentHandlerAPI will be the file known as AgentHandlerAPI.py and will act as a wrapper for RVI messages. These API calls will send the data to the AgentHandler and from there package the message to be sent to the backend.

General format of the code should implement regular status updates to the AgentHandler using the given APIs.

10.2. AgentHandlerAPI (More TBA)

10.2.1. AgentHandlerAPI.report(msg)

msg: will be a JSON string, an array or whatever you wish to send to the logging service. This message will be appended to the payload that is sent off with the RVI message. The AgentHandler will fill in the rest of the information to be sent to the backend logging.

10.2.2. AgentHandlerAPI.status(bool)

bool: will be boolean value representing whether the script is running or not.

11. collectd and Graphite Integration (Optional)

A main use of Dynamic Agents will be taking advantage that these Agents can be used as a system monitoring plugin for collectd. We have purposely chosen a NoSQL database so that we are able to tail the data that is coming into the database in order to forward that data into collectd and ultimately Graphite.

11.1. MongoDB Tailable Cursor

MongoDB has a built in tail cursor that will allow for the last data entry in our database to be given to us. This will allow for more reliable data and information to be broadcast to our system monitoring applications.

11.2. Custom collectd Plugin

The first step of this integration will be creating a custom plugin that will consume the tail data from MongoDB. From MongoDB due to our simplistic nature of dumping most of our payloaded message into the database, we will first have to parse out all corresponding fields. This means that this custom plugin will have to be written to handle dynamic data fields to some extent i.e. checking for fields that may or may not be consistently reported. This means that your payload array may sometimes be bigger or smaller depending on what information was available to be pushed from the Agent. You can also hard code in the values you expect to get from the Agent, but you must make sure in your Agent design that everything corresponds.

The main point of our database setup is that we are able to give you a reliable stream of data that is coming in and that you can then look back at. This ties into the theme that you can technically use any database that you wish to create the Dynamic Agent logging system but the ease of a tailable data structure will be able to give additional features such as connecting the system to collectd or any other system monitoring tool.

An more detailed explanation on how to write custom plugins in python for collectd can be found in this link.

https://collectd.org/documentation/manpages/collectd-python.5.shtml#functions

11.3. Hooking up collectd to Graphite

From collectd you can then choose to visualize your data in any format that you wish. A very common practice is to use collectd in conjunction with Graphite to give you a visual graphical representation of how your data is changing. This will prove very useful for fleet management Agents whether it be geofencing or conflicting sensor readings. When an Agent begins logging all of this logged data will be displayed and can be compared with its normal baseline values to determine if there actually is a problem or not.