WSO2 IoTServer

A Case Study

Akash Singh
McGill University
Student ID: 260728046
akash.singh@mail.mcgill.ca

Shabir Abdul Samadh
McGill University
Student ID: 260723366
shabir.abdulsamadh@mail.mcgill.ca

Rohit Verma
McGill University
Student ID: 260710711
rohit.verma@mail.mcgill.ca

ABSTRACT

This study provides a comprehensive architectural overview of the *WSO2 IoTServer*. The WSO2 IoTServer is a *device-management-server* framework that serves the central management and control requirement of the increasing number of network enabled (smart) devices. The study employs the classic *reverse-engineering* process to gain an architectural understanding of the complete picture as to how this purpose is achieved. We begin by analyzing the static components of the project and expand towards the dynamic behavior of the system. We identify the most salient architectural decisions of the system and critically analyze its components. The study provides a detailed piece-wise description of how this project is put together. It also serves the important purpose of producing a complete architectural review of such systems related to the IoT domain. Thus, imparting valuable & important knowledge with respect to (software) systems for IoT, to its greater audience.

Keywords

Internet-of-Things; middleware; device-management, server, framework

1. INTRODUCTION

The recent developments in technology has seen to an increase in devices that are network (connectivity) enabled; thus being in a constant state of plugged-in to the internet. Being connected to the internet enables these devices to communicate with various other sources. This added capacity as a result of connected-ness deems them *smart* and thus reachable & controllable over the internet. Hence, as a result of the advent of increased numbers of such *smart* devices the internet today has evolved beyond just a network of computers. It has developed into a complete grid of network-enabled-devices of all kinds. Thus, was the term **Internet-of-Things** (IoT) coined to denote a more broader view of the internet which includes devices of every nature interconnected to one another. Studies and research pertaining to any problem related to connected-devices (things) are considered as belonging to this new domain known as *IoT*.

As with any new concept, this area too is inherent with numerous application spaces and related problems. As the internet-of-things grew with more and more smart-devices, a huge number of applications were also introduced to enable remote management and control of each of these devices over the internet. This creates a gap for centralized management of all devices owned by a single entity. As the number of smart-devices in the market increases the number of devices owned by a single entity also tends to increase. On the other hand the need to be able to monitor and keep track of every device manufactured/sold becomes crucial for companies. This creates the need for a central server that can be used to manage devices of all kinds (with different technologies) and to leverage their capabilities remotely over the internet.

Thus, the WSO2 IoTServer serves the middleware[1] purpose of the many connected devices that make up the *Internet of Things*. It is essentially the foundation for any server-side IoT Platform that leverages the capabilities of multiple types of devices[10]. It acts as the central platform that allows to manage and control the various registered device-types and specific instances of those types. In addition it provides a framework to define analytics on the data accumulated (from the devices connected to it) and to present this information to device owners & manufacturers in a highly efficient manner.

1.1 Contribution

The key contribution of this project to the IoT domain is its capacity to cater to devices of any-type. The system acts as a core-framework and provides the required templates to plug-in various kinds of devices. Its in-built features addresses some of the most unique use-cases specific to connected-devices and smart environments. The milestone plan of the project includes various improvements to the system in order to ensure its robustness to support different device-types of the future. As an *Open-Source* initiative the WSO2 IoTServer also contributes to the IoT community by enabling external contributions. It has a very crucial impact on the project given that these contributions include various device-type specific implementations. Thus, this enables rapid development of the project (to support a variety of devices and to cover the entire IoT spectrum) over a short period of time. It helps the project to maintain pace with that of the rate at which new smart-devices are introduced to the network. All of its features and intended use-cases make the WSO2 IoTServer an ideal deployment for any entity that owns numerous devices.

1.2 History

The project was initiated by a senior architect of WSO2 in *April 2015* with a team of 3 fresh computer-science grad-

[1]Middleware is an intermediary software that enables integration and communication between various other softwares[6]

uates. It began as a simple proof-of-concept project given as part of the training exercise to the 3 new recruits. The first sample was only a web-app that could just communicate to an *Arduino* and also receive *pin-readings* from it. This was later modified to a server, managing a simple *Sonar-Sensing* device which was used to count the points scored by each team at WSO2's in-house basketball tournament. Since then, it has gone through various iterations, acquired numerous components and grown into a team of over 20 engineers. It is currently set for its 1^{st} release by early 2017.

As of now, the server is a compilation of multiple modules *(written in accordance to the OSGI-based WSO2 Carbon component structure)* to provide Device Management, API Management, Transport Protocol Extensions, Analytics, etc. The final distribution is a server running on top of the WSO2 Carbon Platform[2] with additional carbon modules specific for device-management. The distribution also consists of device-type specific modules *(also implemented as Carbon components)* that serves the purpose of interacting and managing specific types of devices (ex: Arduino, RaspberryPi, Virtual Devices, etc). These device-type specific modules provide the interface for the devices to communicate with the server and also the device-type specific source (agent) that acts as the device's firmware to communicate with the server.

1.3 Outline

In the sections that follow we present a detailed outline of the entire architecture of the WSO2 IoTServer. We first introduce the technology used and the core concepts related to it. Then we present the process employed in understanding the system and breaking it down into architectural components. We also produce results of our findings and discuss the important metrics understood from our static analysis of the system. Then we present an architectural study of the system, explaining the prevalent styles, design-patterns and implementation choices. We then produce a critical analysis of the system based on our discovery of the architectural elements. We also briefly discuss with regards to other alternative design choices that could have been employed. Finally we conclude by summarizing our overall findings from the reverse-engineering process.

2. TECHNOLOGY & OVERVIEW

The entire core of the WSO2 IoT Server is written in *Java*. The key underlying technology on top of which the system is built is **Java OSGI**. The source also consists of implementations specific to device-types that may communicate with the server. These are programs (also known as agents) that will execute in the device hardware. Thus, these parts of the source are written in languages specific to the device-type in consideration. For example the agent specific to Arduino - written in **Arduino C**, RespberryPi - written in **Python** and ESP8266 - written in **Lua Scripts**. Apache's **Velocity Template Engine**[5] is used to inject device-specific variables into the agent-code and the device configuration files.

The UI components of the server is constructed in accordance to WSO2's **U**nified **UI** **F**ramework (UUF). UUF is component based framework to construct complete UI units from templates which are reusable[9]. **JavaScript** is also used in UI specific components. The UI pages are written as **Handlebar** (.hbs) files. 'Handlebars' is a minimization framework that is used to template HTML pages. It enables separation of HTML generation from rest of the JaveScript[2]. The configuration files of the distribution are maintained as **XML** files. However, the UI component specific configurations are written in **JSON**.

The distribution by default comes with an embedded **H2 Database** server for experimentation purposes[8]. However, the documentation recommends the use of an external database server for actual deployments. Thus, the source also includes **SQL** scripts written to cater different database engines *(MySQL, Postgre, Oracle & etc)*. The overall build procedure of the source is based on the **Apache Maven** build automation tool. WSO2 maintains its own public repository of its components which are readily accessible for building any of its products. The server (or distribution) is started by executing a **Shell Script** in unix based systems and a **Batch File** in windows systems.

2.1 OSGI based architecture

OSGI stands for Open Services Gateway Initiative. It is a modularization framework for java based applications[3]. Applications based on the OSGI technology have its functionality broken into several components, each of them implemented as a collection of one or more OSGI bundles. A bundle is OSGI's modularization unit. An OSGI bundle is a *jar* file which explicitly states which of its packages that it is ready to share with other bundles and which of them are kept private and hidden. All such OSGI bundles are managed by an OSGI run-time environment.

All WSO2 products *(including the WSO2 IoTServer)* uses the **Eclipse Equinox** implementation of the OSGI specification as its underlying run-time environment for managing OSGI bundles. The Eclipse Equinox platform also serves an additional important requirement. It manages the complexity of deploying new components and removing old ones. It orchestrates the component *provisioning* aspect of the platform[7]. Thus, the WSO2 IoTServer's architecture follows a modularized model where all the processes are implemented as separate OSGI components consuming the functionality of each other. The deployment and lifecycle of these components are managed by the Eclipse Equinox P2 provisioning platform.

A key advantage of using OSGI based components is that it supports multiple versions of the same package to exist in the runtime. This is good because the application could operate flawlessly with several components using different versions of the same package. This is not possible in a standard Java application[3]. OSGI's bundle resolver mechanism allows bundles to export/import different versions of the same package. These dependencies are managed by the OSGI environment.

2.2 Carbon Platform & OSGI components

All WSO2 products are built over the WSO2 Carbon Platform. This platform itself is a collection of OSGI based components that provides all core functionality of a server. These functionality includes *User-Management, Service Management, Caching, Registry, Logging, Clustering, Security, Bundle Management & etc.* The distribution of a product

[2]A collection of Java OSGI bundles provided as components that orchestrates server functionality in a unified manner.

that is built over this platform is a collection of these components plus additional ones that provides the specific functionality of the product. All such additional components written to run over the Carbon-Platform and are commonly known as Carbon-Components.

Thus, the WSO2 IoTServer is a collection of OSGI bundles (on top the carbon platform) that operate together to achieve all device-management specific functionality. Whilst the carbon platform specific components provide the core server functionality, the IoTServer specific components yield all device-management specific functionality. As explained in the earlier section, the architecture is composed of several such OSGI based components managed and provisioned by the underlying Eclipse Equinox implementation of the OSGI specification.

2.3 Repository structure

The components of the WSO2 IoT Server are maintained in separate repositories according to their functionality in the final distribution. The core logic specific to the IoT-Server which is responsible for fundamental tasks such as *device enrollment, device communication, device authentication and authorization, API-management and etc* are maintained in the **carbon-device-mgt (CDM)**[3] repository. Components at this level are common to all device-types that may communicate and interact with the server. These components reflect the overall capabilities of this device management server. They use the core server functionality provided by the Carbon Platform.

All components related to *device-type specific* implementations are maintained at the **carbon-device-mgt-plugins (CDM-P)**[4] repository. These components provide all the implementation required for a specific device-type to be a listed device-type in the server. Each component in this repository represents a specific device-type and is called a **device-plugin**. The source of these components utilize the interfaces provided by the previously explained repository to register itself with the server and to channel communication between the underlying server-core & the device. By default this repository holds some sample device-plugin implementations available with a fresh build of the server.

Finally, the **product-iots (P-IoTS)**[5] repository holds all the build rules to combine the components from the previous two repositories and produce the final distribution. It is the source of this repository that is responsible for fetching all the necessary features/components from the Carbon Kernal[6] to build the server and to install the device-management & device-plugin components from the previously explained repositories.

2.4 Packaging source to distribution

The components of the distribution are packaged as features and installed into the final build by means of the **carbon-p2-plugin**. The carbon-p2-plugin is a maven artifact written by WSO2 to automate the carbon-component based build process. The Eclipse Equinox environment is also packaged as part of the distribution in order to support OSGI bundle management and OSGI based component (feature) installation & removal during run-time. During compilation of the product, first the WSO2 Carbon Platform based features are packaged together with the Equinox engine to create the server run-time. Then the device-management specific features are installed into the distribution to enable the core capabilities of the IoTServer. Finally, device-type specific plugins (components/features) are also pushed into the package. Each of these plugins are intended to add capabilities specific to a device-type. The Equinox P2 based carbon-platform enables new device-types to be installed into the server even at run-time. Thus, the final distribution of the WSO2-IoTServer is generated from source.

The server is started by running a script. This script is responsible for setting up the system environment and initiating the bootstrap module. The bootstrap module loads the OSGI run-time and initializes the carbon-platform specific core bundles. Subsequently, device-management and device-plugin specific bundles are loaded onto the run-time.

Table 1: Statistics per repository

Repo	Language	files	blank	comment	LOC
CDM	Java	728	10985	18608	56602
	JavaScript	379	14188	22115	82429
	SQL	14	474	171	4094
	XML	64	496	2927	3139
	JSON	77	0	0	628
CDM-Plugins	Java	591	8533	16135	46105
	JavaScript	114	2969	5783	27730
	Python	5	136	310	347
	ArduinoC	4	96	102	263
	SQL	33	77	186	569
	XML	211	507	3264	5267
	JSON	71	33	0	961
Product-IoTS	Java	312	3693	7648	22053
	JavaScript	202	17118	19534	124803
	SQL	37	1038	166	10132
	XML	426	1580	8504	13988
	JSON	128	44	0	3864
	Python	15	355	704	1473
	Lua	6	32	1	227

2.5 Run-time environment

One key OSGI run-time related concern addressed in the WSO2 carbon platform is the bundle start order of the OSGI environment. It is essential that some bundles be started prior to the others in order for the system to initialize and operate without start-up errors. The server start-up script (explained in the earlier section) executes the *Bootstrap* carbon module. This module in turn calls the *Main* class of the server **org.wso2.carbon.server.Main**. The server module first invokes the server extension tasks *DefaultBundleCreation*, *SystemBundleExtensionCreation*, *Log4j PropertyFileFragmentBundleCreation*, *DropinsBundlesDeployment* and *PatchInstallation*.

Upon completing the above tasks the server module initializes the OSGI framework. The framework loads the OSGI System bundle which in turn loads OSGI Simple Configurator bundle. The configurator bundle reads the **bundle.info** file found in the distribution and loads all OSGI bundles according to the start order. Once all the core OSGI bundles are loaded and registered with the OSGI run-time, the carbon kernal will be initialized. The carbon kernal will execute different artifact deployment tasks and initial-

[3]https://github.com/wso2/carbon-device-mgt
[4]https://github.com/wso2/carbon-device-mgt-plugins
[5]https://github.com/wso2/product-iots
[6]https://github.com/wso2/carbon-kernel

ize transports. Finally the UI framework will initialized and all UI related resources will loaded.

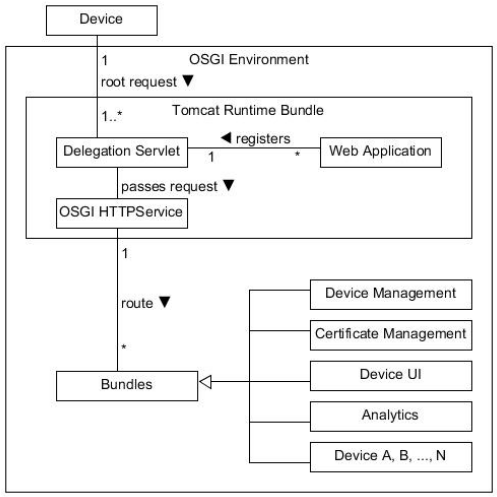

Figure 1: Domain Diagram: Run-time environment

The run-time of the WSO2 IoTServer is primarily the Eclipse Equinox OSGI run-time running on top of the Java Virtual Machine (JVM). All other product related components register with the OSGI run-time as OSGI bundles. One key component of the run-time is the embedded Tomcat bundle. Thus, the run-time environment has an embedded Tomcat instance running as an OSGI bundle which is initialized when the server starts. The root context *('/')* webapp is deployed with tomcat initialization and a servlet called 'Delegation Servlet' is registered to listen to all incoming requests to the root('/*') context[4]. Incoming requests are then dispatched to this delegation servlet which transfers control to the *OSGI HTTPService*[1]. This OSGI service is responsible for routing the requests to the correct context specific OSGI bundle. The OSGI HTTPService is the entry point for all requests into the OSGI environment[4].

2.6 Architectural Overview

2.6.1 Styles & Architectures

The fundamental style in which the project is composed is that of the **Interaction-Oriented Style**. The different tasks of the system are modularized as OSGI bundles and the overall execution of the server is driven by the interaction between these OSGI bundles. Whilst the modularization of processes into bundles can be described as following a **Component based Architecture** the overall logical structure in which these components are interconnected and operate follows the **Layered Architecture**. The system also reflects other architectural styles internal to specific components based on the tasks it handles. The *Data-Analytics* component of the server follows the **Batch-Sequential Architecture** of the **Data-flow Style** to transform the incoming data stream into a server-compatible format and to store it. In addition the *MQTT Transport* module of the server uses an internal broker implementation that follows the **Broker Architecture** of the **Distributed Style**. This enables persistence of all device specific topics and pub-sub based communication between the server and the devices.

2.6.2 Design Patterns

The implementation reflects different design-patterns utilized to achieve specific functionality in the components. We were able to identify the usage of the *Creational Design Pattern:***Factory Pattern** and the *Behavioral Design Pattern:***Observer Pattern** in the core components. We also noticed the usage of the *Structural Patterns*: **Adaptor Pattern** & **Bridge Pattern** in the device-type specific components in addition to the core components. We discuss how these architectures and design patterns are inherent in the system in more detail in the sections that follow.

We have further explained our discovery process, the specific methodologies employed & the related results in the appendices section. We have also provided the scripts used in our analysis.

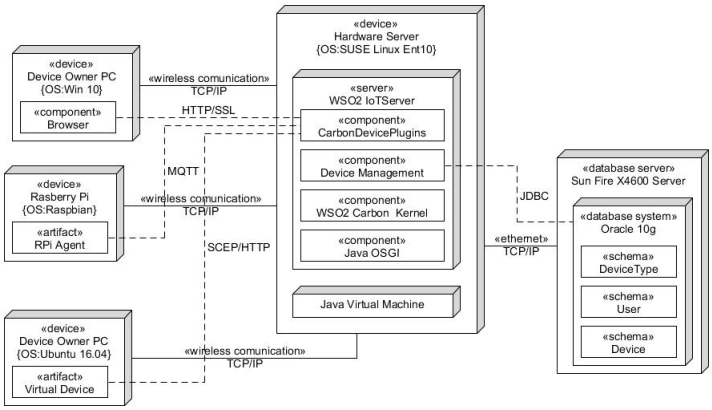

Figure 2: Deployment Diagram of IoTServer

3. DISCOVERY PROCESS

An initial understanding of the architectural structure was gained by reflecting on the need to build three different repositories to obtain the final distribution of the project. We then individually analyzed the order of compilation of these repositories and how code from each repository contributes to the overall system. Whilst gaining background understanding of the different components (from the documentation) we further analyzed the code to identify the execution path. We also realized (by means of external blogs) that the server was compatible for *profiling* with *JProfiler*. We were able to gain a deeper understanding of class dependencies and also component dependencies by studying the profiled metrics available with JProfiler. The user-friendly java development environment *IntelliJ IDEA* was very useful in traversing through interface-based method invocations between components of different repositories. Thus, we were able to carefully sketch the inter-dependencies and construct the layered structure of the components. We were also able to follow the data-flow path with our analysis. Upon establishing an overall architectural understanding we expanded our analysis into specific units of the system.

We identified that several important but extended components of the server were deliverables developed as part of other projects at WSO2. The *Analytics Component* &

the *Message Broker Component* are examples of such components. Analysis of the documentation for these components revealed that they were targeted for specific purposes (Data analysis and storage & message persistence and mediation respectively) following the styles specific to the domain. We were able to establish our understanding of the architecture of these components by means of the server's *debugging-mode* capability. We were able to start the server as debug-enabled and analyze each task. We had breakpoints set up at different method points of communicating classes and were able to follow the data-transformation upon invoking a specific action.

However, we had to rely heavily on documentation to understand how the complete packaging of the distribution is done by means of *maven* plugins. The maven official documentation helped us understand the different build components used in the project. In addition the maven based compilation process was also very-verbose in printing-to-console every task involved in the compilation process. Thus, we were able to understand how each folder of the distribution was created and from where different units of these folders were pulled in. However, we had to read multiple WSO2 related blogs to understand the purpose of each folder and its constituents.

3.1 Documentation Quality

The documentation of WSO2 is rich in content and its scope is rationale to its context; in connection with this, they have a number of characteristics, mainly: clarity, descriptiveness, accuracy, and correctness. Not only does it provides the foundation to the end users but has also considered corporate leadership as its audience. Moreover there are various tutorials & webinars to understand the product in a more interactive way. In addition, regular updates to blogs written for WSO2-IoTServer act as the de facto documentation. Also they play the role of a platform for resolving queries with WSO2-IoTServer specialists. In sum, the documentation is accurate and readable. The qualitative documentation also contains the solution to the problems that can appear while using this product.

However, one notable weakness of the documentation is that there are multiple *work-in-progress* pages for some of the bundles that are not intuitive enough to work with. Also these components depend on each other for execution but the documentation for such weaving is either truant or ambiguous. In some places it was seen that the documentation of the WSO2-IoTServer assumes prior knowledge of the WSO2 Carbon Platform and some of its key components.

4. ARCHITECTURE

The OSGI bundle based deployment of the product follows a **Layered-Architectural Style**. The foundation of this architecture is the *Carbon-Platform* layer over which resides the *Device-Management* (DM) layer. Over the device-management layer resides the *Device-Plugins* (DM-Plugins) layer. The underlying carbon-platform layer consists of components specific for generic server functionality. The middle layer provides the device-management specific components and the top-most layer consists of the device-type specific components. All interaction with the server by an external client or device goes through the components of the top most device-plugins layer. The device-plugins layer invokes components of the device-management layer to get generic device specific processes executed. The middle device management layer utilized the server components of the bottom level carbon-platform layer to receive platform server functionality.

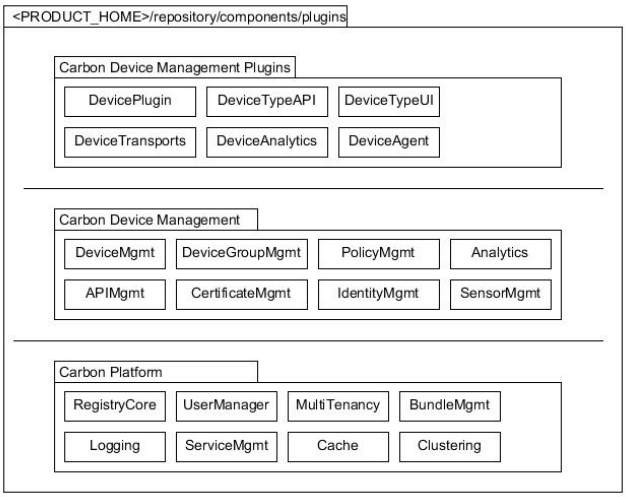

Figure 3: Package Diagram of the Distribution

Figure-3 is a depiction of how the different components are distributed across the three layers. The DM layer in the middle consists of the core components specific to the IoT-Server product. It is to this layer, all device-plugins (device-types) from the top-most layer are registered to. A *Device-Plugin* component of the DM-Plugins layer implements an interface provided by the middle DM layer. It then registers itself against this interface with the OSGI run-time during bundle initiation. Thus, the *DeviceMgmt* component (of the middle-DM layer) is notified of all implementations of this interface and it keeps a list of all such implementations. This way the DM core always executes actions specific to a device-plugin by correctly invoking the interface methods that corresponds to that specific plugin.

Figure-4 shows the basic flow of events for the device registration task. Device registration happens when a device-owner logs into the server and decides to register his hardware with the server by downloading the firmware agent. An agent download request triggers a new device registration event for this specific user. At the end of the activity flow a downloadable 'zip' file containing the agent source together with server specific configurations is returned.

Another key process related to the server is the device-to-server communication. This communication can be secured using various mechanisms. One key capability that the server provides for secure device communication is the *Simple Certificate Enrollment Protocol (SCEP)* based encryption of data. SCEP uses a PKI based certificate mechanism to authenticate an entity. Thus, certificates are created for all devices registered with the server. These SCEP related functionality are provisioned by the *CertificateMgmt* component of the server. A sequence diagram showing the flow of this enrollment and a secure data-transfer is shown in Figure-5.

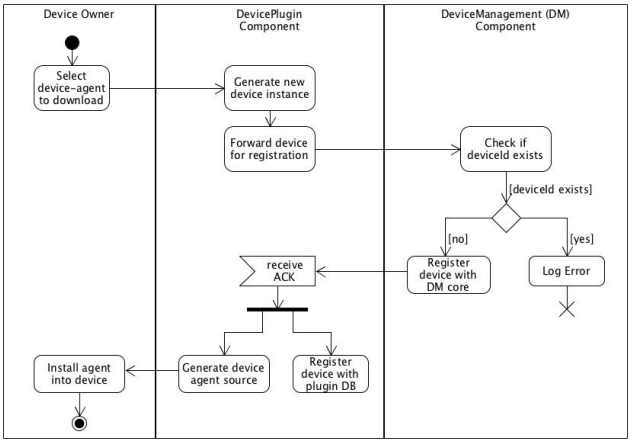

Figure 4: Activity Diagram for Device Registration

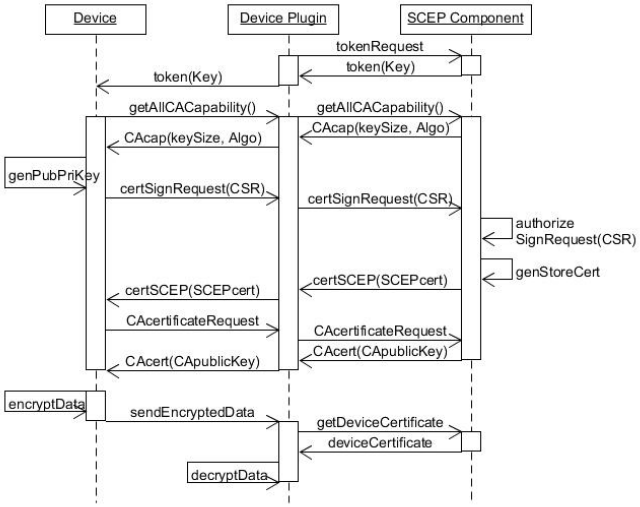

Figure 5: Sequence Diagram for Device Communication using SCEP

We can establish the fact that the system follows a layered architecture from the previous diagrams. It is evident that the components of each layer (as shown in the *Package Diagram* in Figure-3) are clearly separated and communicate with their adjacent layers during the execution flow. It is also noteworthy that different processes of the server are modularized into separate OSGI bundles. It is by registering each of these bundles with the OSGI run-time and leveraging its capabilities via interface mapping the logical layered structure is formed. However, the bundles itself reflect a **Component based Architecture** on top of the OSGI run-time. Hence, this enables bundle installation and removal as and when needed. Also this enables the server to scale its features by borrowing suitable OSGI components from other projects. All components are registered with the OSGI framework at run-time. When a component from one of the logical layers requires a functionality of its adjacent layers it checks for the availability of the specific interface in the OSGI framework. The framework then exposes the interface implementation of the adjacent component.

Figure-6 shows how the package dependency of components in different layers can be mapped to each other. A pointed arrow is a depiction of a package being dependent on the package where the arrow initiates. To further understand the diagram - note that the package abbreviation **o.w.c.d.m** denotes **org.wso2.carbon.device.mgt**. All packages of the project related to the IoTServer starts with this prefix. However, these packages are separated between the two repositories (CDM & CDM-Plugins) by adding an additional **'iot'** namespace for the ones from the CDM-Plugin repository. Thus, all packages form the top-most layer have the abbreviation **o.w.c.d.m.i**.

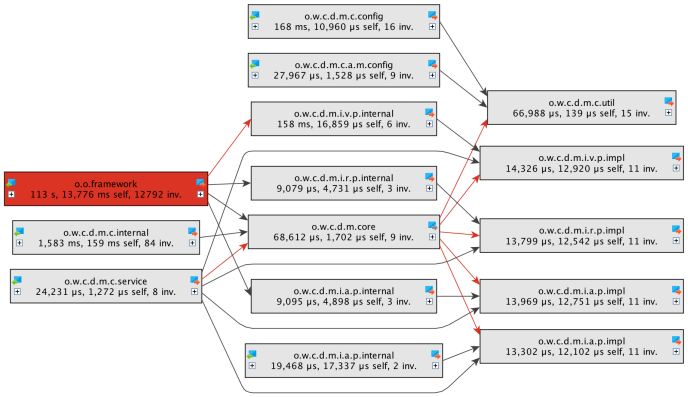

Figure 6: Package Dependency Graph

Also note that the package marked in red denotes the OSGI run-time packages on which the OSGI bundles are dependent on for registering with the OSGI environment. Figure-7 is an example of how class-calls are mapped between the layer-components. If we analyze by the same namespace description provided for Figure-6, we can see that all classes with package names of prefix **o.w.c.d.m.i** are dependent on the ones with prefix **o.w.c.d.m**. The packages with the former prefix are members of the CDM-Plugins layer and the ones with the latter prefix are of the CDM layer. A closer look at the class names will reveal that several *DeviceManagerService* implementations of several device-types *(ex: Arduino, RaspberryPi, etc.)* are prevalent in the plugins layer.

Figure 7: Class Call Hierarchy

The server also includes multiple other flows which are - device communication, device control via the server, viewing device analytics *(real-time & batch)*, device grouping, etc. All such operations are first triggered via the *DeviceAPI* component of the device-plugins layer. Subsequently, com-

ponents of the lower-layer are invoked to achieve the intended tasks.

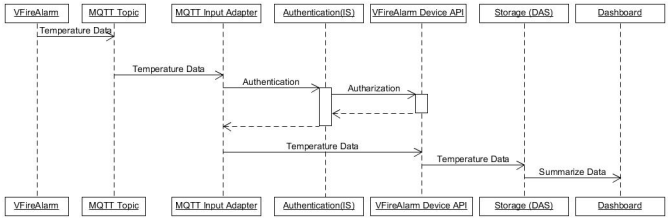

Figure 8: Sequence Diagram:VirtualFireAlarm Data Publishing Flow

Figure-8 is a sequence diagram showing the data publishing flow of the sample-virtual-device provided by default with the server. The flow depicted above includes the use of the *MQTT Transport* and the *Data Analytics* components of the server. The MQTT Transport component leverages the functionality of the *Message Broker* component of the core. This component is implemented in such a way to support broker based communications such as the MQTT protocol. Hence, the internal Message Broker component follows the **Broker Architectural Style** to enable data queuing, fire-and-forget and pub-sub based messaging.

The Data Analytics component enables several manipulations on the incoming/outcoming data and subsequent storage/retrieval to and from the data-store. We learned from our analysis that the units of the Data-Analytics components follow a **Batch Sequential Architecture** where the incoming data is processed at several steps and is dumped to the store and also is re-processed when retrieved to view analyzed information. Thus, these different architectural structures and practices were inherent in some of the components whose purpose is to provide functionality specific to a feature.

4.1 Design Patterns

Usage of different design patterns in different contexts were identified in the reverse-engineering process. The project has utilized these patterns extensively to achieve some common and domain specific implementation of the intended features.

Creational Design Pattern

- **Factory Pattern:** This pattern is used in the database access module of the device-management core component. All database connection logic is handled by a central *DatabaseConnectionFactory* object that is responsible for initializing the connection, reusing existing connections and also releasing all unused resources related to the connection. Moreover, the factory pattern is utilized in cases where a central class takes responsibility of initializing and closing any IO related operations.

Behavioral Design Pattern

- **Observer Pattern:** The *device-mgt core* component registers with the OSGI run-time at startup and listens for any implementations of its *DeviceManagementService* interface. Thus, when a device-plugin registers an implementation of this interface with the OSGI runtime, the device-mgt core is notified of this registration. Subsequently, it adds this to its known list of registered device-types in the server.

Structural Design Pattern

- **Adaptor Pattern:** This pattern is used to transform the incoming data-stream from the devices to the device-type plugin specific *stream-definition*. For all incoming data from devices, the server uses different data-transformation adapters to convert the data into device-type specific stream-definition. The server houses separate device-plugin specific input-adapters & output-adapters to handle incoming device-data and to communicate outgoing data.

- **Bridge Pattern:** This pattern is used in the implementation of the device-type specific communication-protocols/transports. For every communication protocol supported by the server *(MQTT, XMPP etc)* an abstract implementation is found encapsulating the core logic and mechanisms of the protocol. However, the bridge pattern is used to enable device-type specific implementation of how these abstract protocol components are utilized at the device-plugin level for communication.

5. CRITICAL ANALYSIS

Based on the Java OSGI technology, the overall architecture is highly-cohesive with less-coupling. Concise thought towards interface-implementation separation is evident through out the code in-light of the ever expanding list of new device-types.

5.1 Strengths

- **Extensibility**: Given that the architecture is based on the OSGI technology, all features of the server are developed as modularized OSGI components. Thus, this enables easy addition & removal of components as and when needed. Moreover, the server could be scaled to include new device-types (as they come into the market) by implementing the OSGI bundle for the new device-plugin.

- **Clustering & Multi-Tenancy**: The Carbon Platform has components to allow clustering and multi-tenency built into the kernel. Hence, any product build based on the carbon platform supports clustering and multi-tenancy by default. This enables the WSO2IoTServer to be deployed in a distributed fashion with several jvm instances. The core is able to manage the cluster and serve requests duly. Also the multi-tenancy capability allows an organization to run multiple sub-domains of the server and enables cloud deployment.

- **Scalability**: As a result of the server's clustering capability, it is possible to separately deploy the core components *api-manager, analytics-server, message-broker* of the server in a clustered fashion. Thus, we can add more instances of the nodes as required to scale the capacity of the server.

- **Dynamic Adaptation**: One key strength of OSGI is that it allows the existence of multiple versions of different bundles in the same run-time. The framework resolves the version dependency of these bundles and delivers the requirement to the components. Hence, the server is capable of having components that consume different versions of the same libraries.
- **Heterogeneous Device Support**: The server is capable of handling different types of devices. The component based structure allows each device-type to have its plugin written customized to its requirement. Different devices can also implement their own transports to use for communication.

5.2 Weaknesses

- **Heavyweight**: The server has a quiet high footprint in terms of the distribution size *($\approx 1GB$)*. Also the run-time demands high memory capacity to serve features like data-analytics. gh
- **High Latency**: Given the layered style, the server has a significant latency in its responses since all flows navigate through multiple layers.
- **Difficult Error/Exception Handling**: The source of errors are not straight-forward since they are passed over multiple layers before being handled or exposed.
- **Tight coupling with OSGI**: The overall architecture is heavily dependent on OSGI. Thus, component compliance to OSGI is essential and an important drawback is to time and manage the exact bundle deployment order in the framework.
- **Overhead**: As a result of separation into multiple layers there is an overhead of communications introduced between components. Also component-wise system-resource consumption increases overall overhead of the server.

5.3 Alternate Architecture

Whilst the existing architecture is most suited for the dynamic nature of Internet of Things, the Service Oriented Architecture (SOA) will be a prominent alternative for this system. Each bundle/component in WSO2 IoTServer can be implemented as a service in the SOA environment. The service functionality can be exposed as *REST* or *RPC* interfaces. Device-plugins and transports specific to different device-types can be implemented as separate services. This allows for reuse of common services between multiple devices of similar classes. The current architecture encapsulates device-type specific functionality within its own device-plugin component. This restricts the freedom at which similar device-type specific implementations can be shared between multiple plugins. However, SOA can overcome this limitation. A device can wield a service using a service directory and then access it using a request-response model. Coarse-grained service model can be used to make the components/services work in coordination. Therefore a service composition model is mandatory and a containment can be to used to wrap all related services. Also, since a new service can be easily added in a SOA based system, the core benefits of WSO2 IoTServer like scalability and extensibility will be preserved with the alternate SOA approach.

6. CONCLUSION

The WSO2 IoTServer fills a key gap in the current IoT domain. The component based layered architecture of the product caters to the most important attribute of supporting multiple device-types. It also allows for the implementation of device-type specific communication methodology and data-transfer mechanisms. The OSGI framework based run-time enables the server to be extended with numerous features as required. It also allows for sub-sampling of the server with a lesser number of components if need be. Moreover, dynamic OSGI feature installation and removal is enabled through the feature provisioning Eclipse Equinox framework. Following an OSGI based layered architecture suits the dynamic nature of the *Internet-of-Things* domain. It allows for customized expansions and tuning according to the scenario in question.

Whilst the code is of high quality following proper practices and suitable design patterns, the implementation reflects thorough thinking of the domain. The existing documentation is very precise and accurate. However, there seems to be a lack of continuous improvement of the documentation since documentation for some components are still in-progress. The project also has multiple other explanatory channels such as developer blogs and marketing webinars. The richness of the documentation needs to be increased in the upcoming versions to communicate the real-world use-cases of the server. The current bug count of the project is ≈ 110 whilst 23 of them are listed as *'high'* status. An open-source project of this kind in an evolving field such as the IoT is valuable in that, it has the capacity to grow rapidly to support the IoT picture with external contributions.

7. REFERENCES

[1] O. Alliance. Interface httpservice, 2016. [Online; accessed 1-December-2016].

[2] Handlebars. Handlebars, 2016. [Online; accessed 12-November-2016].

[3] S. Jayasoma. How to build osgi bundles using maven bundle plugin - part 1, 2016. [Online; accessed 12-November-2016].

[4] K. Thangarajah. Wso2 carbon architecture: A high-level understanding and how it works, 2016. [Online; accessed 12-November-2016].

[5] Wikipedia. Apache velocity — wikipedia, the free encyclopedia, 2016. [Online; accessed 5-September-2016].

[6] Wikipedia. Middleware — wikipedia, the free encyclopedia, 2016. [Online; accessed 26-October-2016].

[7] WSO2. Provisioning wso2 carbon based soa products with equinox p2, 2016. [Online; accessed 12-November-2016].

[8] WSO2. Setting up embedded h2, 2016. [Online; accessed 12-November-2016].

[9] WSO2. Unified ui framework (uuf), 2016. [Online; accessed 12-November-2016].

[10] WSO2. Wso2 iot server, 2016. [Online; accessed 12-November-2016].

APPENDIX

A. PERFORMANCE ANALYSIS

We used the JProfiler tool for performance & run-time analysis of the WSO2 IoTServer. Various metrics with regards to memory, speed, execution-time and object counts were collected as part of the analyzing process.

A.1 JProfiler setup

1. Application and JProfiler should run in the same JVM
2. Setup the agent path in the wso2server.bat file as: `/[sourcePath]/jprofilerti.dll`.
3. Add Jprofiler agent in application boot path.
4. Specify the Jprofiler port number, default is 8849.
5. Then attach to profiled JVM and start the session.
6. After connecting the Jprofiler to the profiling menu, record memory.

A.2 Actions Performed

1. Detecting the 'bottlenecks' in the source code.
2. Capturing memory and time expensive function, classes and packages.
3. Identification of memory leaks and unnecessary memory allocation.
4. Compute the package, class and Object count.
5. Ascertain the Size of packages, classes and objects.

A summarized metric was created to bolster the reverse engineering process of WSO2 IoT server. The following figures depict various important metrics related to the runtime analysis of the server. We had run the server for approximately $45 mins$ and collected the following information. During this period some basic events like device registration and communication was triggered.

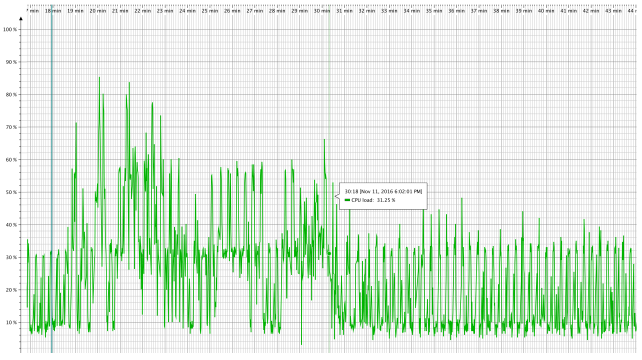

Figure 9: CPU load over time

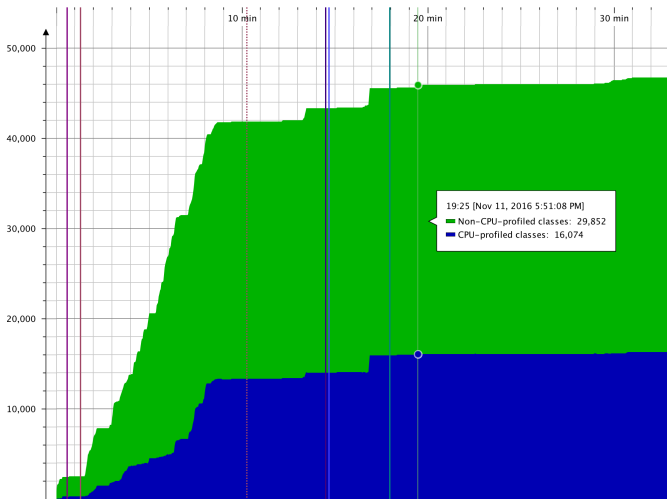

Figure 10: Class count over time

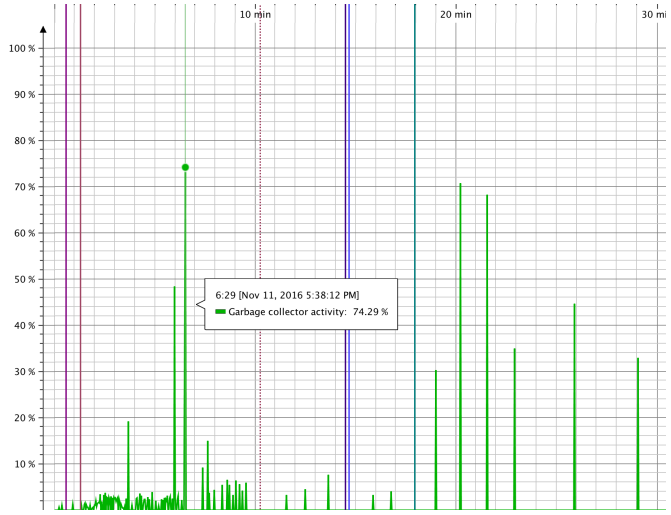

Figure 11: Garbage-Collection over time

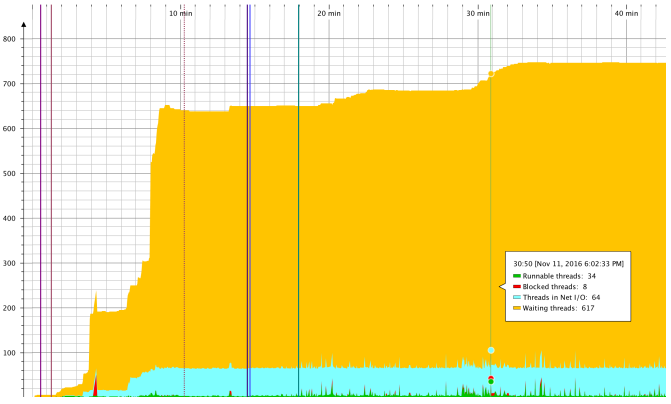

Figure 12: Thread distribution over time

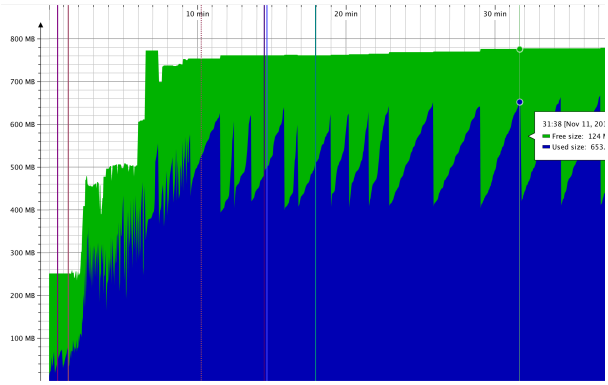

Figure 13: Memory allocation over time

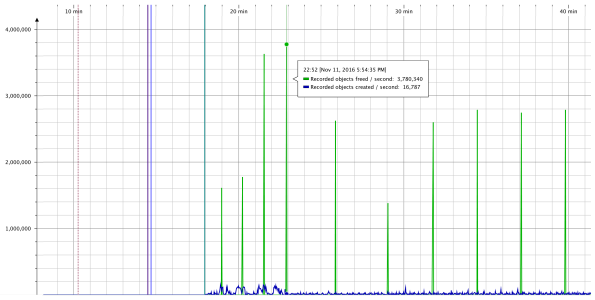

Figure 14: Throughput over time

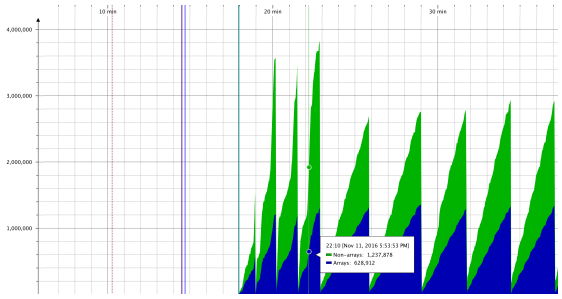

Figure 15: Recorded objects over time

B. SOURCE SPECIFIC METRICS

The linux command line tool **'cloc'**[7] was used to analyze the repositories related to the project and to generate the metrics with regards to the source. A simple shell script was written to collect the necessary data from the repos using this tool. The *github* links of the repositories were passed as argument for this script.

```
#!/usr/bin/env bash
git clone --depth 1 "$1" temp-repo &&
printf "(temp-repo 'deleted)\n\n\n" &&
cloc temp-repo &&
rm -rf temp-repo
```

The output of this script for the 3 repositories of the project can be found in: Table-2, Table-3 & Table-4. The table lists the blanks, comments and code by line count.

[7]http://cloc.sourceforge.net/

B.1 Compile-Time calculation

The project uses the *maven* build tool to automate its build task, put the components together and to deliver the final distribution as a zip-file. Given that maven builds can be executed via the linux terminal, we utilized the linux *SECONDS* environment variable which keeps track of the execution time of shell scripts. We were able to calculate the total compile-time of the project by reading the value of the $*SECONDS* variable upon executing the following script. This gave us an average time of 8 minutes & 2 seconds.

```
#!/bin/sh
SECONDS=0
cd <PATH-TO-REPO-1>/carbon-device-mgt
mvn clean install
cd <PATH-TO-REPO-2/carbon-device-mgt-plugins
mvn clean install
cd <PATH-TO-REPO-3>/product-iots
mvn clean install
duration=$SECONDS
echo '$(($duration/60)) minutes'
echo '$(($duration%60)) seconds elapsed'
# 8 minutes and 2 seconds elapsed.
```

We can learn from the tables 2, 3 & 4 that most of the source **files** are that of Java. However, we see a huge number JavaScript code found in the CDM and Product-IoTS repository. This is primarily because these repositories hold the UI framework related code specific to the Carbon-Platform and the IoTServer respectively. This does not necessarily mean that there is a higher mass of JavaScript, since JS snippets are very short and expand over multiple lines of code. This can be verified from the fact that the total Java related files in the project are 1631 in comparison to only 695 JavaScript files.

Table 2: CDM Repository

Language	files	blank	comment	code
JavaScript	379	14188	22115	82429
Java	728	10985	18608	56602
CSS	55	2469	943	29318
Maven	72	494	1270	8165
Handlebars	103	534	1384	4962
SQL	14	474	171	4094
XML	64	496	2927	3139
LESS	4	346	120	1962
JSON	77	0	0	628
INI	1	5	0	253
SASS	1	25	10	114
XSD	4	13	68	98
Velocity Language	1	3	0	31
HTML	1	1	17	8
Markdown	2	4	0	6
SUM:	1506	30037	47633	191809

Table 3: CDM-Plugins Repository

Language	files	blank	comment	code
Java	591	8533	16135	46105
Handlebars	103	1465	2567	30745
JavaScript	114	2969	5783	27730
Maven	78	623	1444	8180
XML	211	507	3264	5267
CSS	13	378	211	2241
JSON	71	33	0	961
SQL	33	77	186	569
Python	5	136	310	347
Velocity Tem Lang	4	32	0	274
Arduino Sketch	4	96	102	263
Bourne Shell	5	49	437	163
Bourne Again Shell	1	20	21	123
Ant	5	34	123	121
Groovy	3	7	4	75
DOS Batch	1	24	2	64
XSD	1	5	17	47
LESS	1	8	3	37
C/C++ Header	1	17	27	33
Markdown	2	20	0	32
SUM	1247	15033	30636	123377

Table 4: Product-IoTS Repository

Language	files	blank	comment	code
JavaScript	202	17118	19534	124803
Java	312	3693	7648	22053
Maven	106	674	2321	16770
XML	426	1580	8504	13988
SQL	37	1038	166	10132
CSS	36	821	473	7223
Handlebars	40	355	97	4952
JSON	128	44	0	3864
Python	15	355	704	1473
XSD	2	6	17	1153
Bourne Shell	8	67	314	401
JSP	4	40	123	369
INI	1	5	0	253
Lua	6	32	1	227
HTML	3	20	34	213
Ant	9	32	153	208
Arduino Sketch	3	66	66	197
DOS Batch	1	42	44	91
Markdown	3	34	0	73
C/C++ Header	1	18	20	31
SUM:	1343	26040	40219	208474

The following figures provide a graphical summarization of the tables listed earlier. They give a distributed view of source code division over all three repositories.

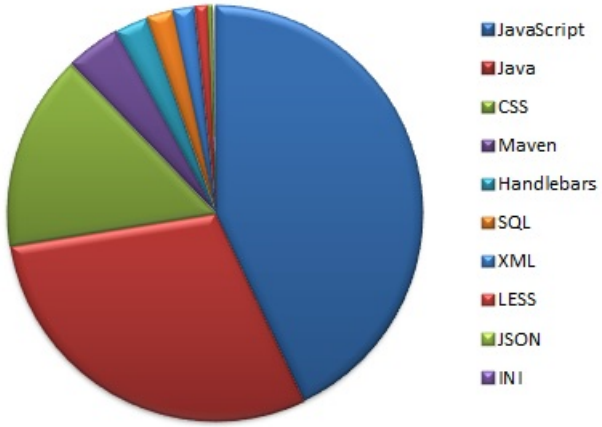

Figure 16: Source Distribution - CDM

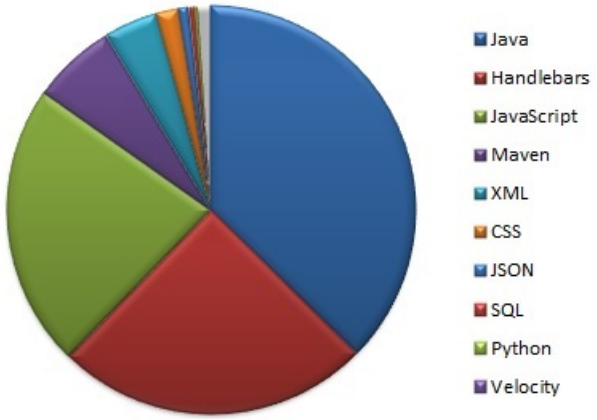

Figure 17: Source Distribution - CDM Plugins

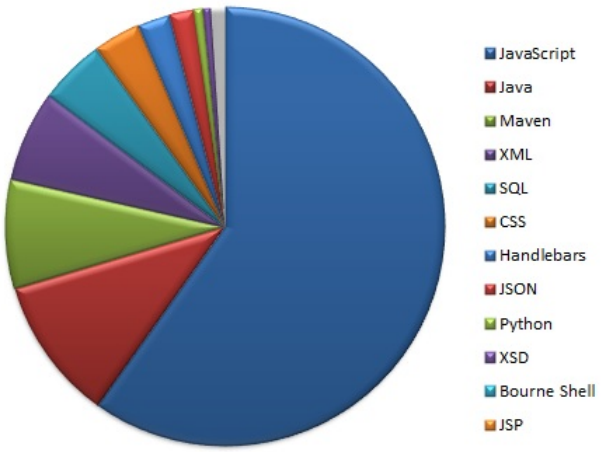

Figure 18: Source Distribution - Product IoTS

C. ADDITIONAL METRICS

Documentation Mass: The project contains approximately ≈ 50659 lines of documentation which includes blank lines. The documentation is available online to view and download. It also states that 'work in progress'.

Library-to-Core Ratio: All the packages utilized by the distribution can be found at the path: $< PRODUCT - HOME > /repository/components/plugins$ inside the final distribution.

- **Total Packages** - 751
- **Project related Packages** - 468
- **Ratio** - 62.3%

D. DIRECTORY STRUCTURE

```
wso2iots-1.0.0-alpha
├── bin
│   ├── wso2server.sh
│
├── dbscripts
│   ├── ...
│
├── device-deployer.xml
├── lib
│   ├── ...
│
├── modules
├── repository
│   ├── components
│   │   ├── features
│   │   ├── p2
│   │   ├── patches
│   │   ├── plugins
│   │
│   ├── conf
│   │   ├── ...
│   │
│   ├── database
│   │   ├── ...
│   │
│   ├── deployment
│   │   ├── client
│   │   └── server
│   ├── log4j.properties
│   ├── logs
│   │   ├── ...
│   │
│   ├── resources
│   │   ├── ...
│   │
│   └── tenants
├── samples
│   └── ...
```

Figure 19: Directory Structure of the Product

E. HOTSPOTS ANALYSIS

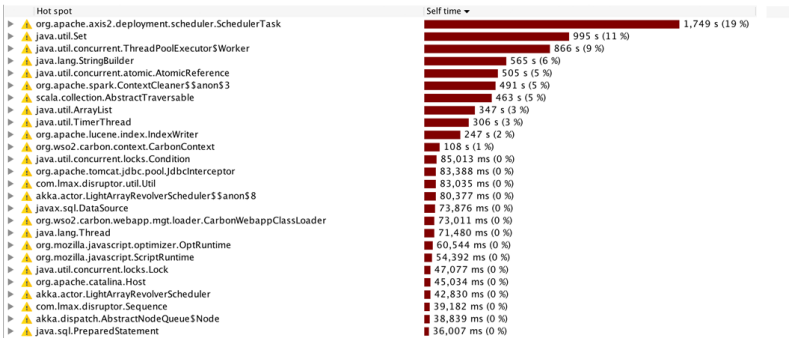

Figure 20: Speed Measurement on Hotspots

F. UI INTERFACE

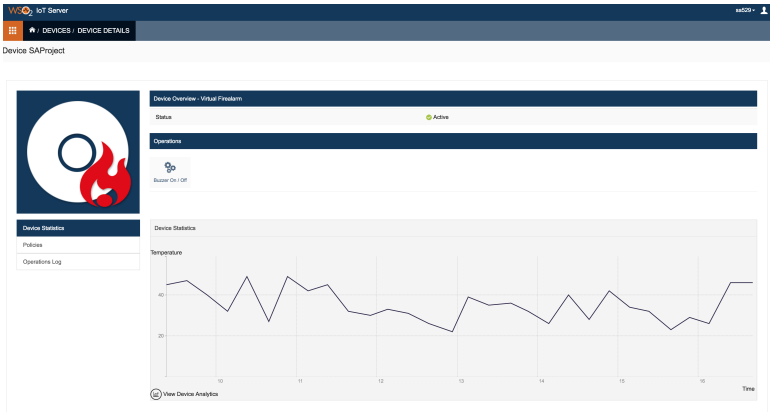

Figure 21: Sample UI Interface for device-view

G. DISTRIBUTION SIZE ANALYSIS

```
wso2iots-1.0.0-alpha    -   2.6GB

1000K   ./dbscripts
784K    ./samples
384K    ./modules
232K    ./bin
 26M    ./lib
2.5G    ./repository

1.9G    ./repository/data
 22M    ./repository/resources
 69M    ./repository/database
234M    ./repository/deployment
347M    ./repository/components
580K    ./repository/logs
896K    ./repository/conf

1.8M    ./repository/components/p2
320M    ./repository/components/plugins
```

Figure 22: Size footprint of main directories