Architecture Design

* *IoT anyware* -



Team number 1

June 2015

Document History

|  |  |  |
| --- | --- | --- |
| **Rev.** | **Description** | **Date** |
| 0.0 | Creation | 07/06/2015 |
| 0.1 | Section 1, 2 | 10/06/2015 |
| 0.9 | Refine all sections | 17/06/2015 |
|  |  |  |
|  |  |  |

**TABLE OF CONTENTS**

[1. DOCUMENT DESCRIPTION 3](#_Toc422997096)

[1.1 The Purpose of Document 3](#_Toc422997097)

[1.2 The Document Organization 3](#_Toc422997098)

[1.3 Terminology and Definitions 5](#_Toc422997099)

[1.4 References and Relevant Documents 5](#_Toc422997100)

[2. PROJECT OVERVIEW 5](#_Toc422997101)

[2.1 The Purpose of Project 5](#_Toc422997102)

[2.2 The Scope of Project 6](#_Toc422997103)

[2.3 The Stakeholders 6](#_Toc422997104)

[3. ARCHITECTURAL DRIVERS 6](#_Toc422997105)

[3.1 High Level Functional Requirements 6](#_Toc422997106)

[3.2 Quality Attributes 8](#_Toc422997107)

[3.3 Constraints 10](#_Toc422997108)

[4. SYSTEM CONTEXT 10](#_Toc422997109)

[5. 1st DECOMPOSITION OF IOT SYSTEM 12](#_Toc422997110)

[5.1 Dynamic Perspective 12](#_Toc422997111)

[6. DECOMPOSITION OF IOT SERVER 14](#_Toc422997112)

[6.1 Dynamic Perspective 14](#_Toc422997113)

[6.1.1 1st Decomposition of IoT Server 14](#_Toc422997114)

[6.1.2 2nd Decomposition of IoT Server 16](#_Toc422997115)

[6.1.3 3rd Decomposition of IoT Server 21](#_Toc422997116)

[6.2 Static Perspective 27](#_Toc422997117)

[6.3 Physical Perspective 32](#_Toc422997118)

[7. DECOMPOSITION OF USER APP 33](#_Toc422997119)

[7.1 Dynamic Perspective 33](#_Toc422997120)

[7.2 Static Perspective 36](#_Toc422997121)

[8. DECOMPOSITION OF SA NODE 39](#_Toc422997122)

[8.1 Static Perspective - 1st Decomposition of SA node 39](#_Toc422997123)

[8.2 Dynamic Perspective - 2nd Decomposition of SA Network Manager 42](#_Toc422997124)

[9. SYSTEM DESIGN SUPPLEMENT 45](#_Toc422997125)

[9.1 Sequence diagram 45](#_Toc422997126)

[9.2 User Case Scenario 48](#_Toc422997127)

[9.3 API Specifications 57](#_Toc422997128)

[APPENDIX A – EVENT BUS EXPERIMENTS 58](#_Toc422997129)

1. DOCUMENT DESCRIPTION

1.1 The Purpose of Document

The purpose of this document is to describe the software architecture design of the “Internet of Things” infrastructure. This project is the part of the course work of the LG Architecture Training Program at Carnegie Melon University. This document contains the architectural drivers, and architecture design of the system, and design decisions driven by the architectural drivers. It also provides the interfaces and protocols between system components in detail for the implementers to work in parallel.

1.2 The Document Organization

The order of presentation is chosen to convey the design of the system in a top down approach. First, section 2 introduces the project context to give a general understanding of the system. The summary of the architectural drivers are presented in section 3 to elaborate what are important factors that define the design choices. In section 4, the system context is described; explaining the boundary of the system, and system entities – *IoT Server*, *SA Node*, and *User App* interact with the environment. Each entity is decomposed to the level where significant functional requirements and quality attributes are applied to the design through section 5 to 8. Section 9 provides the supplementary system designs which can help developers better understand how the system entities, described in section 4, interact with each other.

This document is organized as follows.

* Section 2 - Project Overview
* In this section, purpose, scope, and stakeholder of the project are described.
* Section 3 - Architectural Drivers
* In this section, the architectural drivers of the IoT project are described: high level functional requirements, the summary of quality attributes, and constraints.
* Section 4 - System Context
* This section describes the boundary of the system and how the system entities interact with the internal and external entities.
* Section 5 - 1st Decomposition
* This section describes how responsibilities of *IoT System* are assigned to sub components.
* Section 6 - Decomposition of IoT Server
* The detailed decomposition of *IoT Server* is described through the sub sections, and the rationales of selected design choices are presented.
* Section 7 - Decomposition of SA node
* The detailed decomposition of *SA node* is described through the sub sections, and the rationales of selected design choices are presented.
* Section 8 - Decomposition of User App
* The detailed decomposition of *User App* is described through the sub sections, and the rationales of selected design choices are presented.
* Section 9 - System design supplement
* This section provides the supplementary system designs which can help developers better understand how the three major system entities – *IoT Server*, *SA Node*, and *User App* interact with each other in different scenarios.

1.3 Terminology and Definitions

This section defines acronyms and terms that used in this document.

|  |  |
| --- | --- |
| **Terminology** | **Definition** |
| IoT | Internet of things |
| SA | Sensor/actuator |
| SA node | A device consists of sensors and/or actuators, and a network module communicates with the IoT system. |
| QA | Quality attribute |
| Sensor/actuator profile | A characteristic and/or capability description of the sensor or actuator. |
| SA node description | A description of the SA node; including names of sensors and actuators, and their profiles. |
| REST | A REST (REpresentational State Transfer) is an architectural style for distributed hypermedia system. |

1.4 References and Relevant Documents

[1] <http://en.wikipedia.org/wiki/Internet_of_Things>

[2] Team1\_01\_Architecture\_Driver\_Specifications\_v1.0.doc

[3] Team1\_03\_Event Bus Interface Specification\_v1.0.doc

[4] Team1\_04\_REST Web API Specification\_v1.0.doc

2. PROJECT OVERVIEW

2.1 The Purpose of Project

The Internet of things refers to the network of devices equipped with electronics, software, sensors, and connectivity to provide the greater values or services by exchanging data with the manufacturer, service provider, and/or other connected devices [1]. The purpose of this project is to build the IoT infrastructure which meets the following key goals.

* Create an IoT infrastructure to support accessing sensors and actuators installed in the home or business.
* Create an infrastructure to provide an ecosystem to develop cost competitive home or business IoT products for value-added-resellers and other 3rd party hardware and software application developers, service providers, and installers and maintainers.
* Build a basic data centric infrastructure to provide IoT data sets for developers to create future data mining, analytic operations, and services.

2.2 The Scope of Project

The scope of the project is to design and build the IoT infrastructure. The IoT infrastructure is consist of three major parts; the IoT server, the SA node, and the user application. The IoT server provides services - event publication and subscription, and secure access to the connected devices or the data collection for SA nodes and user applications. The SA node can post sensor states or notifications, and/or retrieve actuator control messages or any events from other devices, systems, or users for extended services through the IoT server. In the other hand, the user application can post actuator control events, retrieve sensor states or notifications from the SA node, and/or access the SA node state/user command history data through the IoT server.

2.3 The Stakeholders

The stakeholders of the project are customers, and IoT business interest group such as sensor/actuator producers, home builders, third-party service providers, application developers, etc, and the project management. Functional requirements are extracted from the use cases of customers or the IoT business interest group, and the quality attributes are derived from the requirements. Business and/or technical constrains, the project planning and the priority of the QAs are discussed and confirmed with the project managements.

3. ARCHITECTURAL DRIVERS

The aim of this section is to describe the architectural drivers of the IoT project: high level functional requirements, the summary of quality attributes, and constraints. Use case scenario, presented in section 9.2 breaks down the high level functional requirements into detailed scenarios. Refer to “Team1\_01\_Architecture\_Driver\_Specifications\_v1.0.doc” [2] for more detail on the architectural drivers of the project.

3.1 High Level Functional Requirements

The key functional requirements of the system are as follows.

* The system should provide secured IoT services.
* The system enables the user to control actuators, and to determine sensor states.
* The system should collect user command and sensor states, and present them to the rightful customers.

Below are the functional requirements of the system in more detail.

|  |  |
| --- | --- |
| **Access secured services** | FR01 |
| **Description**:  User accesses the system in secured environment. User must login to the system for services. Unauthorized persons are not allowed to control sensors installed in home, register SA nodes, or access any data gathered in the system. | |

|  |  |
| --- | --- |
| **Discover SA nodes** | FR02 |
| **Description**:  User queries home to find out how many nodes are installed and what sensors/actuators are installed on each node. | |

|  |  |
| --- | --- |
| **Determine sensors and control actuators** | FR03 |
| **Description**:  User can determine the temperature/humidity, turn on and off lights, open and close the door, turn on the alarm, and determine if anyone is home. However, user must set the alarm off prior to opening the door. | |

|  |  |
| --- | --- |
| **Log user commands and sensor values** | FR04 |
| **Description**:  User commands and sensor values are stored in IoT infrastructure for some period of time. This data set can be utilized by developers to create future data mining, analytic operations, and services. | |

|  |  |
| --- | --- |
| **Send emergency message** | FR05 |
| **Description**:  An emergency message is sent to the user when door is opened manually or the house is suddenly occupied while alarm is set. | |

|  |  |
| --- | --- |
| **Lock house automatically** | FR06 |
| **Description**:  User is informed upon the vacancy of house and asked to lock the house. If the home is vacancy for 30 seconds and alarm is not set, SA node notifies user. If the user failed to respond to the message within 5 minutes, the door is closed, and the alarm is set automatically. | |

|  |  |
| --- | --- |
| **Turn off light automatically** | FR07 |
| **Description**:  When no one is home for 10 minutes, the light is turned off automatically. | |

|  |  |
| --- | --- |
| **Register SA node** | FR08 |
| **Description**:  Authorized user adds nodes to the system. Equipped sensors and actuators are recognized. | |

|  |  |
| --- | --- |
| **Unregister SA node** | FR09 |
| **Description**:  Authorized user removes nodes from the system. | |

|  |  |
| --- | --- |
| **Grant SA node access permission** | FR10 |
| **Description:**  User who registered a SA node is the owner of the node. He/she grants an access permission of the node to others. However, only owner of the node can unregister the node. | |

|  |  |
| --- | --- |
| **Transfer SA node ownership** | FR11 |
| **Description:**  User who registered a SA node is the owner of the node. He/she can give up the ownership of the node, and transfer the ownership to other user. The user who gives up the ownership still has the access permission of the node. | |

3.2 Quality Attributes

Quality attributes that the system must promote in the architecture design are:

* The system must identify unauthorized accesses, and protect data and services from it.
* The system should not allow users to register SA nodes that are not owned by them.
* The system should identify the failure of SA nodes.
* The system should make it easy to add emerging protocols.
* The system should make it easy for application developers (private persons, VARs, or other 3rd parties) to build custom apps, services, and/or make mashups from existing available services.

One of the most important quality attribute of the IoT system is the security, since the security breach of the system can lead to physical damages of people. The system must provide best efforts to prevent the malicious use of the system. Only authorized users should be able to control or register the SA nodes of which they have permission.

The essential quality attribute of the IoT system is the availability. The failure of the SA node can cripple the quality of entire system services. The result of this drawback can be lost productivity, lost revenue, damaged customer relationships, bad publicity, and lawsuits. If a mission-critical sensor or actuator becomes unavailable, the user could be placed in jeopardy.

The IoT fragmentation will grow until one standard, ecosystem or technical model dominates the market. For this reason, our IoT system has to be flexible enough to adapt emerging technologies or protocols. In addition, the system should provide an easy to use development environment for 3rd party service providers or application developers in order to build the solid IoT ecosystem to become a leader in the industry.

Five out of nine quality attributes are prioritized as high as shown below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Total** | **High Priority** | **Medium Priority** | **Low Priority** |
| 9 | 5 | 4 | 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Priority** | **Quality Attribute** | **Descriptions** |
| QA01 | H | Security | Hackers or malicious people try to break into the system. When unauthorized user attempts to login to the system, the system maintains the audit trail. If the attempt is repeated more than 5 times, the account is locked, and the source of tempering is identified. |
| QA02 | H | Security | Hackers or malicious people try to register the SA node that is not owned by them. When unauthorized user attempts to register the SA node that he/she doesn’t own, the system maintains the audit trail, and cancels the registration in 10 minutes. |
| QA03 | H | Availability | SA node can crash, hang, or be disconnected from the network for various reasons. If SA node is inoperable or out of reach, the system should be aware of such events, and notify user within 2 minutes. |
| QA04 | M | Availability | SA node can be disconnected from the network for various reasons. If SA node is not able to reach the system due to network failure, it should store recent event at least for one day. When the network is restored, SA node should send the logs to the system. |
| QA05 | M | Scalability | The number of SA node user can be more than one. The system should be able to serve 10 user controls to the same SA node. (Concurrent access and control is not considered in this scenario) |
| QA06 | M | Scalability | More than one SA node can be installed at home. The system should be able to support at least 100 nodes concurrently. |
| QA07 | H | Modifiability | The system should make it easy to add emerging protocols (e.g. Bluetooth 802.15, ZigBee 802.15.4) to the system. Average skilled developers should be able to implement it within two months. |
| QA08 | M | Usability | The system should make it easy for users to register or unregister SA nodes. Ordinary user should be able to register or unregister the node within 5 minutes by following the provided manual. |
| QA09 | H | Extensibility | The system should make it easy for application developers (private persons, VARs, or other 3rd parties) to build custom apps, services, and/or make mashups from existing available services. Average skilled developers should be able to build the application in six months. |

3.3 Constraints

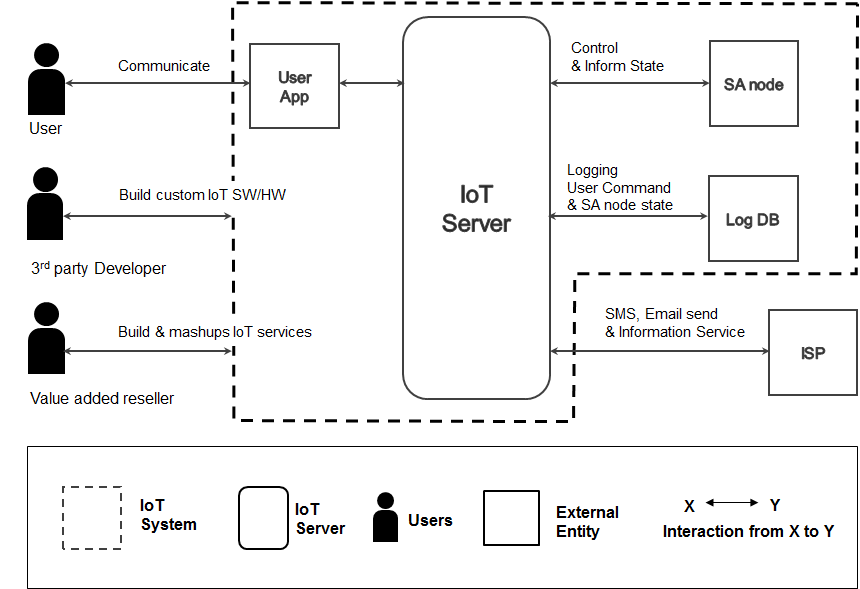
The major risk that the project team confronts is the budget of the project. The team is consisting of five developers; very new to the development environment such as JAVA, Python, and Arduino. However, the project has to be completed in 35 days to outpace competitors.

The business and technical constrains of the project are organized in the table below.

|  |  |  |
| --- | --- | --- |
| **ID** | **Type** | **Description** |
| TC01 | Technical | JAVA compiler, Arduino 1.0.6 is preferred. |
| TC02 | Technical | Permissible languages for this system (excluding the SA nodes) are JAVA and Python. |
| TC03 | Technical | 802.11 is only supported in the system. |
| TB01 | Business | Development period: 5 weeks (3 hours/day) |
| TB02 | Business | Development team: 5 developers. |

4. SYSTEM CONTEXT

Figure 4.1 System Context shows the boundary of the system and how the system entities interact with the internal and external entities. End users communicate with sensors and actuators installed in the home or business via any mobile device (laptop, phone, and tablet) or desktop system connected to the Internet. The system supports an ecosystem of developers, value-added-resellers (VARs), and other 3rd party hardware and software application developers, service providers, and installers and maintainers. A basic data centric infrastructure supports the collection of IoT data that would enable developers to create future data mining, analytic operations, and services. The *IoT Server* interacts with external information-service-providers (ISP) to send messages, or retrieve valuable information such as weather casts or local traffic information.

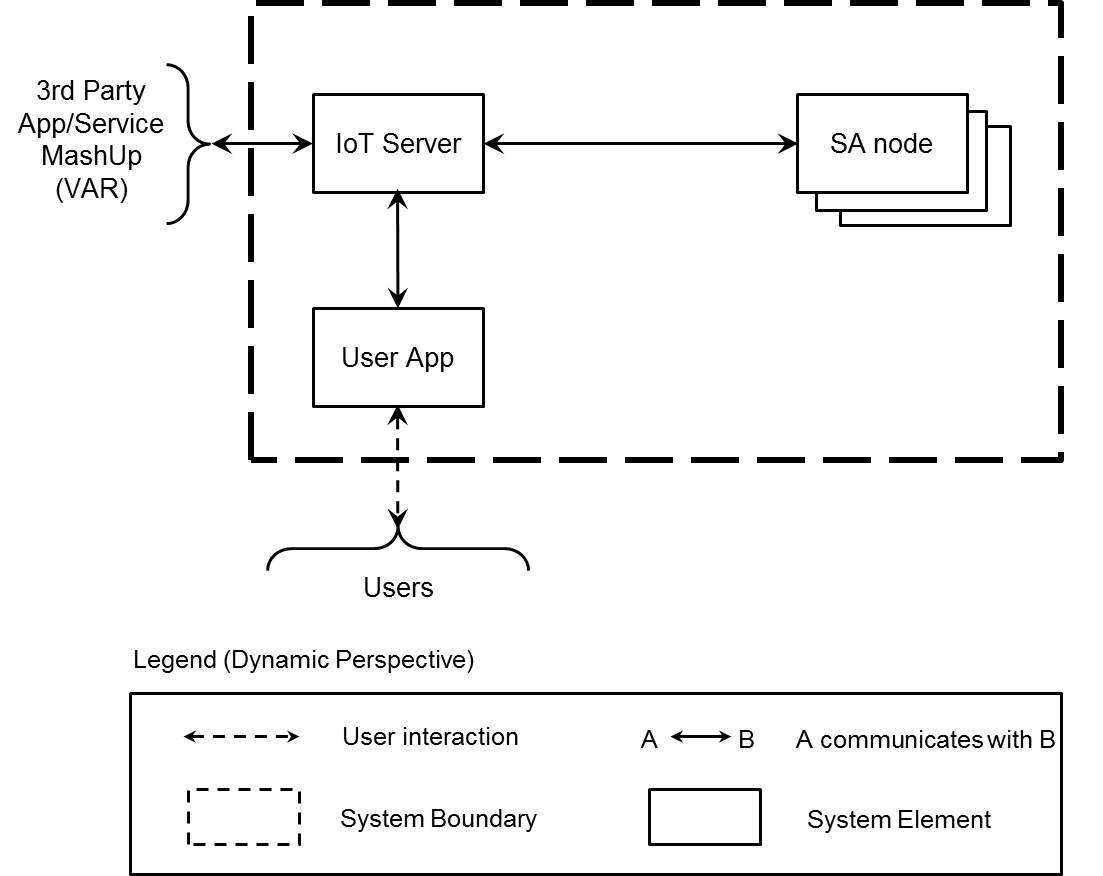


**Figure 4.1 System Context**

5. 1st DECOMPOSITION OF IOT SYSTEM

5.1 Dynamic Perspective

This section describes how responsibilities of *IoT System* presented in the context diagram (Figure 4.1) are assigned to sub components. As Figure 5.1 shows, the *IoT System* is decomposed into three major components; *IoT Server*, *SA node*, and *User App***.** *SA node,* *User App* and 3rd party applications communicate with *IoT Server* to get permission for the services. Users interact with *User App* to control actuators, or to read sensor values through *IoT Server*. *SA node* communicates with *IoT Server* to post sensor values or to receive actuator control commands. The responsibilities and relationship of each element are described in Table 5.1 and 5.2.



**Figure 5.1 First Level of Decomposition - Dynamic perspective of IoT system**

**Table 5.1 Element Responsibility Catalog for First-Level Decomposition**

|  |  |
| --- | --- |
| **Perspective:** Dynamic  **Associated Drawings:** Figure 5.1 | |
| **Element** | **Responsibilities** |
| *IoT Server* | * This element (*IoT Server*) provides IoT services to *SA node*, *User App* and 3rd party applications. * IoT *Server* authenticates and authorizes *User App, SA nodes*, and 3rd party app/service to access services. They communicate with *IoT server* in secured environment. * IoT *Server* enables *SA node* or *User App* to send messages to each other. |
| *User App* | * This element is an application enables end users to access *SA nodes* via *IoT server*, and/or provide IoT services to users. * User *App* can be installed on various devices (mobile, laptop, PC) * User *App* may have GUI interface to interact with user. |
| *SA node* | * This element reads sensors, or controls actuators. * Send sensor values to *User App* via *IoT Server*. * Receives actuator control messages. |

**Table 5.2 Relationship Responsibility Catalog for First-Level Decomposition**

|  |  |
| --- | --- |
| **Perspective:** Dynamic  **Associated Drawings:** Figure 5.1 | |
| **Relationship** | **Responsibilities** |
| B  A | * This relationship is used to connect *SA node*, *User App* and 3rd party App/Service to the *IoT Server*. This symbol indicates that an element A communicates with an element B over a network connection. |
|  | * This relationship indicates interaction between end user and *User App*. |

**Table 5.3 First Level of Server Decomposition Rationale**

|  |
| --- |
| **Perspective:** Dynamic  **Associated Drawings:** Figure 5.1  **Associated Responsibilities:** Tables 5.1 and 5.2 |
| The decomposition focuses on the responsibilities of *IoT System*. *IoT System* is composed of three major parts; *IoT server, SA node, and User App*. *IoT Server* provides the secured communication platform for other elements, *SA node* controls sensors and actuators, and *User App* provides services to users to control *SA node* remotely. |

6. DECOMPOSITION OF IOT SERVER

The detailed decomposition of *IoT Server* is described through the sub sections, and the rationales of selected design choices are presented as well. *IoT Server* is the one of the major component of *IoT System* as shown in Figure 5.1.

6.1 Dynamic Perspective

6.1.1 1st Decomposition of IoT Server

*IoT Server* consists of two elements; *IoT Server Runtime* and *Event Bus*. *Event Bus* provides data channels between *IoT Server Runtime*, *User App* and *SA Node* for event exchange. However, an event bus implementation generally omits authentication features to spread transaction loads. In order to reinforce the security of *Event Bus*, the *IoT Server Runtime* provides advanced authorization and authentication. *IoT Server Runtime* manages user accounts with access control policies, and generates time-limited session which secures the *Event Bus* access.

*IoT Server Runtime* holds responsibilities for other functionalities as well.

* Log sensor status/user transaction history, and provide the log to authorized users.
* Provide intellectual services by analyzing sensor data and the information from the external services (i.e. weather information).
* Send push notification via SMS/Email.

These functionalities are covered in following sections in more detail.

**Figure 6.1 First Level of Decomposition - Decomposition of IoT Server**

**Table 6.1 Element Responsibility Catalog for First-Level Decomposition**

|  |  |
| --- | --- |
| **Perspective:** Dynamic  **Associated Drawings:** Figure 6.1 | |
| **Element** | **Responsibilities** |
| *IoT Server Runtime* | * This element is a process or collection of processes. * Authorize and authenticate to access *Event Bus*. * Log sensor status/user transaction history, and provide the log to authorized users. * Provide intellect services by analyzing sensor data and external service information (i.e. weather information). * Send push notification via SMS/Email. |
| *Event bus* | * This element is a process or collection of processes. * Coordinate message passing between *IoT Server Runtime*, *SA node* and *User App*. |

**Table 6.2 Relationship Responsibility Catalog for First-Level Decomposition**

|  |  |
| --- | --- |
| **Perspective:** Dynamic  **Associated Drawings:** Figure 6.1 | |
| **Relationship** | **Responsibilities** |
| B  A | * This relationship indicates that element A connects to element B over a network connection; TCP/IP, socket- oriented protocol for authentication and authorization. Refer to section 6.1.2 and 6.1.3 for more detail on the network connection protocol. |
| B  A | * This relationship indicates that element A communicates with B by event messages at runtime. |

**Table 6.3 First Level of Server Decomposition Rationale**

|  |
| --- |
| **Perspective:** Dynamic  **Associated Drawings:** Figure 6.1  **Associated Responsibilities:** Tables 6.1 and 6.2 |
| *IoT System* is decomposed into *IoT Server* Runtime and *Event Bus*. This partitioning was chosen to maintain the separation of concerns between authorization/authentication functionality and an event message passing functionality. *User App* or *SA node* must go through the authorization process before accessing *Event Bus*. (*QA01 Security*)  Event bus system promotes the extensibility of the system. Publishers are loosely coupled to subscribers, and they don’t need to know their existence. Since the event producer and the consumer are not connected directly, they are allowed to remain ignorant of system topology. Each can continue to operate normally regardless of the other status. As a result, this structure makes it easy to add or change SA nodes and User app at runtime without affecting each other. (*QA09 Extensibility*) |

6.1.2 2nd Decomposition of IoT Server

Figure 6.2 describes how *IoT Server Runtime* is decomposed further. *IoT Server Runtime* includes two components; *Web Server* and *Event Manager*. *Web Server* is responsible for the authorization and authentication of the user and *SA node*. Moreover, it handles *SA node* management operations associated with the user account; registration, un-registration, sharing access permission (FR10), and transferring ownership (FR11).

The communication channel to *Web Server* is secured by HTTPS. The user account and associated *SA node* information is stored in the RDBMS database, and *Event Bus* refers to this data for the authorization of *Event Bus* access.

To reduce the burden applied to the *Event Bus*, the event logging is processed by *Event Manager*. *Event Manager* monitors and logs the events. In order to maintain scalability, event logs are stored in NoSQL database. The logs are accessible through the *Web Server*.

*Event Manager* is responsible for below functionalities as well.

* Provide intellectual services by analyzing sensor data and the information from the external services.
* Send push notification via SMS/Email.

**Figure 6.2 Second Level of Decomposition - Decomposition of IoT Server Runtime.**

**Table 6.4 Element Responsibility Catalog for Second-Level Decomposition**

|  |  |
| --- | --- |
| **Perspective:** Dynamic  **Associated Drawings: Figure 6.2** | |
| **Element** | **Responsibilities** |
| *Web*  *Server* | * This element is a single process. * Handling authorization and authentication of system access. * Issue a valid session to client. The id of session is used for accessing Event bus. * Provide event logs. |
| *Event Manager* | * This element is a process or a collection of processes. * Log sensor status/user transaction history. * Provide intellect services by analyzing sensor data and external service information (i.e. weather information). * Send push notification via SMS/Email. |
| *Event*  *Bus* | * This element is a process or collection of processes. * Provide message communications between connected clients. (pub/sub) * Client which has valid session id can access *Event Bus*. The session is issued by the *Web Server* and valid only for limited time period. * Monitoring all pub/sub topics, in order to prevent malicious user from creating illegal pub/sub topics. |
| *Sensor Data Store* | * The data store is a file or a collection of files. * The data store is a repository for saving sensor values and user commands. |
| *User Data Store* | * The data store is a file or a collection of files. * The data store is a repository for saving user account information and associated *SA node* information. |

**Table 6.5 Relationship Responsibility Catalog for Second-Level Decomposition**

|  |  |
| --- | --- |
| **Perspective:** Dynamic  **Associated Drawings:** Figure 6.2 | |
| **Relationship** | **Responsibilities** |
| B  A | * This relationship indicates that element A connects to element B over HTTPS. |
| B  A | * This relationship indicates that element A communicates with B by event messages at runtime. |
| B  A | * This relationship indicates that element A CURD (Create / Update / Read / Delete) B. The connector is based on a database query. |

**Table 6.6 Second Level of IoT Server Runtime Decomposition Rationale**

|  |
| --- |
| **Perspective**: Dynamic  **Associated Drawings**: Figure 6.2  **Associated Responsibilities**: Tables 6.4 and 6.5 |
| The second level decomposition shows how the responsibilities of *IoT Server Runtime* is divided, and assigned to two separate elements. *Web Server* is responsible for authorization and authentication of system access. It issues a session id to client, and the client access the *Event Bus* with the session id. This mechanism ensures the secured communications among *User App*, *SA nod*e and 3rd party applications. (*QA01 Security*)  *Web Server* also provides the logs of sensor status and user commands. However, *Event Manager* is responsible for collecting the logs to reduce the burden of *Event Bus*. Sending push notification and the intellectual service management are the roles of *Event Manager* as well.  The user account, *SA node* information associated with the account, and session information are stored in a RDBMS data store. The sensor status and user commands are stored in a NoSQL data store. *Event Manager* is the only writer of this data store, and *Web Server* and *Event Manager* read the data from it, thus concurrent data access is not the concern. |

Figure 6.3 shows the second level decomposition of *Event Bus*. *The Event Bus* includes two components; *MQTT Broker* and *MQTT Proxy*.

*MQTT Broker* is responsible for basic functionality of event bus; event pub-sub. MQTT Broker uses subject-based message filtering. In other world, the message contains the subject called topic, and it is delivered only to clients subscribing the topic. The format of topics used for the system services is discussed in section 9.3.

*MQTT Proxy* is responsible for authorization and authentication of *Event Bus* access. It validates the session id which is included in the *Event Bus* connection request*.* Furthermore, *MQTT Proxy* provides the access control of publish/subscribe request. With session id, *MQTT Proxy* identifies the topics that are allowed to the client to publish or subscribe. *MQTT Proxy* verifies the topic of the message, and transmits only valid message to *MQTT Broker*. This promotes the security of pub-sub communications among the system entities. In addition, the communication channel to *MQTT Proxy* is secured by SSL.

**Figure 6.3 Second Level of Decomposition - Decomposition of Event Bus.**

**Table 6.7 Element Responsibility Catalog for Second-Level Decomposition**

|  |  |
| --- | --- |
| **Perspective:** Dynamic  **Associated Drawings: Figure 6.3** | |
| **Element** | **Responsibilities** |
| *MQTT  Proxy* | * This element is a process * Provide authorization for *Event Bus* connection request using session which is provided by Web Server. * Provide access control of publish/subscribe request. *MQTT Proxy* verifies the topic of the message, and transmits only valid message to *MQTT Broker.* |
| *MQTT Broker* | * This element is a process. * Provide message communications between clients. (publish/subscribe) |

**Table 6.8 Relationship Responsibility Catalog for Second-Level Decomposition**

|  |  |
| --- | --- |
| **Perspective:** Dynamic  **Associated Drawings:** Figure 6.3 | |
| **Relationship** | **Responsibilities** |
| B  A | * This relationship indicates that element A connects to element B over HTTPS. |
| B  A | * This relationship indicates that element A communicates with B by event messages at runtime. Secured by session id and SSL. |
|  | * This relationship indicates that element A connects to element B over TCP/IP. |

**Table 6.9 Second Level of Event Bus Decomposition Rationale**

|  |
| --- |
| **Perspective**: Dynamic  **Associated Drawings**: Figure 6.3  **Associated Responsibilities**: Tables 6.7 and 6.8 |
| The second level decomposition shows how the responsibilities of *Event Bus* is divided, and assigned to two separate elements; *MQTT Broker and MQTT Proxy*.  *MQTT Broker* is responsible for basic functionality of event bus; event pub-sub. *MQTT Proxy* is responsible for authorization and authentication of system access.  *MQTT Proxy* validates the session id which is included in the *Event Bus* connection request*.* Furthermore, *MQTT Proxy* provides the access control of publish/subscribe request. With session id, *MQTT Proxy* identifies the topics that are allowed to the client to publish or subscribe. *MQTT Proxy* verifies the topic of the message, and transmits only valid message to *MQTT Broker*. This promotes the security of pub-sub communications among the system entities. In addition, the communication channel to *MQTT Proxy* is secured by SSL. (*QA01 Security*) |

6.1.3 3rd Decomposition of IoT Server

The workload of reading and transmission of logs is large compare to the user authentication load or *SA node* management load. Therefore, the execution path should be separated to prevent performance degradation of *Web Server. Log Viewer* handles the log view requests. Upon the request, *Log Viewer* retrieves the identification information of *SA node*(s) associated with the requester account and the log time configuration from *Auth Manager*, then base on the information, search the log in the Sensor Data Store (NoSQL).

*Auth Manager* processes user account management operations, SA node management operations, and authorization/authentication.

User account registration sequence includes the user verification by the email. Auth Manager sends the verification email via SMTP service. Refer to section 9.1 for more detail on the user account registration sequence.

**Figure 6.4 Third Level of Decomposition - Decomposition of Web Server.**

Figure 6.5 shows the third level decomposition of *Event Manager*. *Event Manager* is the name of the group consists of *Event Listener*, *Logger*, *Service Broker*, *Rule Manager*, *Heartbeat Manager*, and *Push Manager*. *Event Listener* is a process, and *Logger*, *Rule Manager*, *Heartbeat Manager*, and *Push Manager* are event handler threads created by *Event Listener*. *Event Listener* establishes connection to the *Event Bus* and propagates the received event to the appropriate event handler. Below are the responsibilities of each handler.

* *Logger* stores sensor status and user commands.
* *Rule Manager* provides intellectual services by analyzing sensor data and the information from the external services. For example, Rule Manager gets weather information from the external service provider. When the weather forecast calls for a high probability of rain, and the door is open, *Rule Manager* requests to *SA node* to close the door. Yet, *Service Broker* helps *Rule Manager* to be loosely coupled from the external services.
* *Heartbeat Manager* monitors the heartbeat event of *SA node*. Upon three timeouts, *Heartbeat Manager* notifies user of *SA node* failure via *Event Bus*.
* *Push Manager* delivers an *SA node* alert message to user through an email or an SMS. *User App* subscribes alert events, but in many cases *User App* will not be available since the network reliability of mobile devices is relatively bad. *Push Manager* requests user information such as an email address or a phone number to *User Data Store* (RDBMS).

**Figure 6.5 Third Level of Decomposition - Decomposition of Event Manager.**

**Table 6.10 Element Responsibility Catalog for Third-Level Decomposition - focusing on the IoT Web Server.**

|  |  |
| --- | --- |
| **Perspective:** Dynamic  **Associated Drawings:** Figure 6.4 | |
| **Element** | **Responsibilities** |
| *Auth Manager* | * This element is a single process. * Authorization and authentication to access the event bus element. * Provide stateless REST APIs for following services. * Add new user. Response includes two types of tokens, called access token and refresh token. The access token expires after some period of time. The refresh token never expires and this token used to refresh access token when access token is expired. * Retrieve user information. * Validate token. * Refresh token. * Revoke token. * Manage user account. * Manage SA node * Refer section 9.3 for more detail on REST APIs. |
| *Log*  *Viewer* | * This element is a single process. * Provide REST API for the log of sensor status/user transaction history. * Refer section 9.3 for more detail on REST APIs. |

**Table 6.11 Element Responsibility Catalog for Third-Level Decomposition - focusing on the Event Manager.**

|  |  |
| --- | --- |
| **Perspective:** Dynamic  **Associated Drawings:** Figure 6.5 | |
| **Element** | **Responsibilities** |
| *Event*  *Listener* | * This element is a single process. * Create event handler threads; *Logger, Rule Manager, Heartbeat, and Push Manager.* * Propagate subscribed event to the handlers. |
| *Logger* | * This element is a single thread. * Save events to the data store. |
| *Push*  *Manager* | * This element is a single thread. * Request sending push message to external push service provider. The recipients are users of the *SA node*. |
| *Rule*  *Manager* | * This element is a single thread. * Provide intellect services by analyzing sensor data and external service information. * Create Service Broker thread to query weather information. |
| *Heartbeat Manager* | * This element is a single thread. * Monitors periodic heartbeat event from *SA node*. Upon three timeouts, *Heartbeat Manager* notifies user of *SA node* failure via *Event Bus*. |
| *Service Broker* | * This element is a single thread. * Query weather forecast to external weather service provider. |

**Table 6.12 Relationship Responsibility Catalog for Third-Level Decomposition**

|  |  |
| --- | --- |
| **Perspective:** Dynamic  **Associated Drawings:** Figure 6.4 and 6.5 | |
| **Relationship** | **Responsibilities** |
|  | * This relationship indicates that element A connects to element B over HTTP(S). |
| B  A | * This relationship indicates that element A send event message to B at runtime. |
|  | * This relationship indicates that element A push message to B’s queue. |
| B  A | * This relationship indicates that element A CURD (Create / Update / Read / Delete) B. The connector is based on a SQL query. |

**Table 6.13 Third Level of IoT Web Server and Event Manager Decomposition Rationale**

|  |
| --- |
| **Perspective:** Dynamic  **Associated Drawings:** Figure 6.4, 6.5  **Associated Responsibilities:** Tables 6.10, 6.11, and 6.12 |
| *Log Viewer* is responsible for reading and sending accumulated logs filtered by the given user information and timestamp within specified period. Since the workload of log access is vast, it may hurt the performance of *Web Server*. Therefore, *Log Viewer* is running in a separate process.  To reduce the burden applied to the *Event Bus*, the event logging is processed by *Logger*. *Logger* monitors and logs the events. In order to maintain scalability, event logs are stored in NoSQL data store. The scale out of NoSQL is easily achieved by adding new resource into system. *(QA06, Scalability)*  *Push Manager* is responsible for delivering *SA node* alert messages to user through an email or an SMS. *User App* subscribes alert events, but in many cases *User App* will not be available since the network reliability of mobile devices is relatively bad.  *Rule Manager* is responsible for providing intellectual services by analyzing sensor data and the information from the external services.  *Heartbeat Manager* promotes the availability of the system. (*QA03 Availability*) Heartbeat tactic is used to detect the failure of the SA node. This tactic is selected over Ping/Echo to reduce event traffics. However, the performance degradation still has to be addressed due to the extra traffics generated by the Heartbeat events. The rate of event and the scale of single server have to be identified. The performance experiment had been conducted on the matter and the results are as follows:   * A single server can support up to 1000 SA node at the same time. * The event rate should be 10 seconds or longer to avoid the performance degradation.   Refer to Appendix A to find more on the performance experiment. |

6.2 Static Perspective

Web Server uses Web Framework to handle REST requests and their responses. Responsibili­ties of REST request APIs are assigned to *Account Module*, *Profile Module*, *User Module*, *Log Module*, and *Session Module*. Common functionalities such as argument, session, and authoriza­tion validation are implemented in *Decorator Module*. *Decorator Module* uses *Argument Check Module* to check whether the mandatory properties are included in the Header and Body of the request. In addition, values of those properties are validated by *Argument Check Module*. Session management is implemented in *Session / Auth Check Module*.

To remove the dependency to specific database, Object-Relational Mapping(ORM) Model converts data between incompatible type systems. The *ORM Model* allowing the writing of code which naturally makes use of the features of the language. This creates, in effect, a "virtual object database" that can be used from upper layer module and promotes portability as a result.

Table 6.14 explains the responsibility of each module in more detail.

**Figure 6.6 IoT Server Class Structure Decomposition — Structure of Web Server.**

**Table 6.14 Element Responsibility Catalog for Static Decomposition**

|  |  |
| --- | --- |
| **Perspective:** Static  **Associated Drawings:** Figure 6.5 | |
| **Element** | **Responsibilities** |
| *Web Framework* | * Common structures and methods for web applications. * Common methods for sending and receiving data via HTTP/HTTPS. * Provide MVC template engine. |
| *Views* | * Provide UI (User Interface) using static assets such as HTML/CSS/images. For dynamic contents, the template engine from the Web Framework would be used. |
| *Account Module* | * Provide API for sign up. * Provide API for user identification (email confirmation) |
| *Profile  Module* | * Provide API for retrieve detail profile information. |
| *User  Module* | * Provide API for retrieve user information. * Provide API for register SA node. * Provide API for sharing and transfer SA node to other user. |
| *Session  Module* | * Provide API for validate token. * Provide API for refresh token. * Provide API for revoke token. |
| *Log Module* | * Provide API for get logged sensor status and user access history within configured time windows. |
| *Decorator* | * Common utility function for pre-request and post-request handling. |
| *Argument Check Module* | * Common utility function for check whether input argument (parameter) is valid or not. For example checking existence of mandatory parameter and its value. |
| *Session / Auth Check Module* | * Common utility function for check validity of input session and authentication of user app client. |
| *ORM Model* | * Common utility function for translate class definition with SQL query statement. |
| *mysql Binding Module* | * Common utility function for data transfer with the mysql database. |
| *Token handling Module* | * Common utility function for generating and management OTP token. |
| *mongoDB Binding Module* | * Common utility function for data transfer with the mongoDB database. |

**Table 6.15 Relationship Responsibility Catalog for Static Decomposition**

|  |  |
| --- | --- |
| **Perspective:** Static  **Associated Drawings:** Figure 6.6 | |
| **Relationship** | **Responsibilities** |
|  | * Dependency. * One class depends on another if the independent class is a parameter variable or local variable of a method of the dependent class. |

*Event Manager* uses *MQTT Binding Module* to receive or to send events via *Event Bus*. *Push Notification Module* and *Rule Management Module* uses *mysql Binding Module* to access *User Data Store* to retrieve user account information. *Logger Module* uses *mongoDB Binding Module* to access *Sensor Data Store* to store event logs. Rule Management uses Service Broker to find the external service information. *Service Broker* is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions. This structure could decouple components that interact by remote service invocations from others. (Increase *Portability*)

**Figure 6.7 IoT Server Class Structure Decomposition — Structure of Event Manager.**

**Table 6.16 Element Responsibility Catalog for Static Decomposition**

|  |  |
| --- | --- |
| **Perspective:** Static  **Associated Drawings:** Figure 6.7 | |
| **Element** | **Responsibilities** |
| *Event Daemon* | * Instantiate the event callback modules in accordance with the event message input. |
| *Logger  Module* | * Refer to the description of *Logger* in section 6.1.3. |
| *Rule Mgmt. Module* | * Refer to the description of *Rule Manager* in section 6.1.3. |
| *Service  Broker* | * Refer to the description of *Service Broker* in section 6.1.3. * Instantiate *Service Proxy*. |
| *Push Noti. Module* | * Refer to the description of *Push Manager* in section 6.1.3. |
| *Heartbeat Module* | * Refer to the description of *Heartbeat Manager* in section 6.1.3. |
| *MQTT  Binding Module* | * Common methods for publish and subscribe via message bus. * Common methods for register callback which related to subscribed topic. |
| *mongoDB Binding Module* | * Common utility function for data transfer with the mongoDB database. |
| *mysql  Binding Module* | * Common utility function for data transfer with the mysql database. |
| *SMTP Module* | * Composite SMTP mail and send it to user(s). |

**Table 6.17 Relationship Responsibility Catalog for Static Decomposition**

|  |  |
| --- | --- |
| **Perspective:** Static  **Associated Drawings:** Figure 6.7 | |
| **Relationship** | **Responsibilities** |
| X  Y  Module X  Uses Module Y | * Dependency. * One class depends on another if the independent class is a parameter variable or local variable of a method of the dependent class. |

6.3 Physical Perspective

The IoT Server infrastructure is serviced with Amazon Web Services (AWS). User App and Web Browser is connected to IoT Server via internet using MQTT and HTTPS protocol. Any port which not serviced is blocked by firewall. In order to support high availability and failover, IoT Server is deployed across multiple Availability Zones. Database also provisioned and maintained a synchronous standby replica in a different Availability Zone.

**Figure 6.8 IoT Server Deployment — Physical perspective.**

7. DECOMPOSITION OF USER APP

The detailed decomposition of *User App* is described through the sub sections, and the rationales of selected design choices are presented as well. *User App* is the one of the major component of *IoT System* as shown in Figure 5.1.

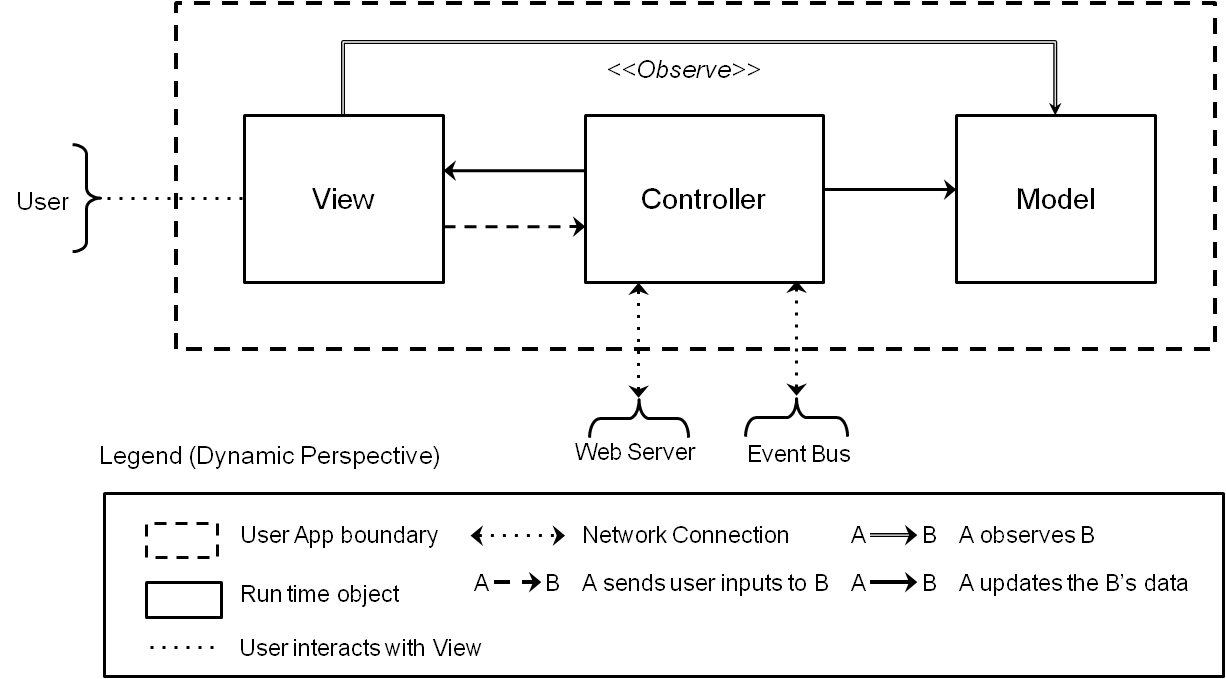
7.1 Dynamic Perspective

*User App* consists of three interconnected components; *View*, *Controller*, and *Model*. MVC pattern is adapted to the *User App* architectural design for 3rd party developers to build various user interfaces easily.

User interacts with *View* to determine sensor values or to control actuators. *View* presents the status of SA node by observing the data change of *Model*, and delivers user inputs to *Controller*. *Controller* posts actuator control events to *Event Bus* over network, subscribes sensor value update events, and updates the *Model* data. *Model* captures the SA node model and the model of analytic logs produced by the system components.

*Controller* connects to *Web Server* over the network for the access authorization, user account management, SA node managements, and log presentation. Operations for these tasks are as follows:

* Login
* User account registration / un-registration
* SA node registration / un-registration
* Log presentation



**Figure 7.1 First Level Decomposition of User App**

**Table 7.1 Element Responsibilities Catalog for First-Level Decomposition of User App**

|  |  |
| --- | --- |
| **Perspective:** Dynamic  **Associated Drawings:** Figure 7.1 | |
| **Element** | **Responsibilities** |
| *View* | * This element is run time object. * Show the status of SA nodes. * Provide SA node control interface to user. * Provide user interface for the access authorization, user account management, SA node managements, and log presentation. * Deliver user inputs to *Controller*. * Observe changes of SA node status, and refresh UI. |
| *Controller* | * This element is run time object. * Post actuator control events to *Event Bus* over network. * Subscribe sensor value update events, and update the *Model* data. * Connect to *Web Server* over the network for the access authorization, user account management, SA node managements, and log presentation. |
| *Model* | * This element is run time object. * Represent a SA node. * Capture the behavior of application. |

**Table 7.2 Relationship Responsibilities Catalog for 1st Decomposition of User App**

|  |  |
| --- | --- |
| **Perspective:** Dynamic  **Associated Drawings:** Figure 7.1 | |
| **Relationship** | **Responsibilities** |
|  | * This relationship indicates the data transition over the network. * Controller connects to *Event Bus* over MQTT – publish/subscribe. * Controller connects to Web Server over HTTPS. |
|  | * This relationship indicates that A sends user inputs to B. |
|  | * This relationship indicates that A observes B’s data. * *View* shows the latest status of SA node by observing the *Model*. |
|  | * This relationship indicates that A updates B’s data. * Controller changes the state of View upon the result of user input or *Web Server* response. * Controller updates the *SA node* status, upon status update events from *Event Bus*. * Controller updates the logs obtained from *Web Server*. |
|  | * This relationship indicates interactions between user and *View*. * User interacts with *View* to determine sensor values or to control actuators. * User interacts to login, register/un-register user accounts, register/un-register SA nodes, and view log. |

**Table 7.3 Rationale of First-Level Decomposition of User App**

|  |
| --- |
| **Perspective:** Dynamic  **Associated Drawings:** Figure 7.1  **Associated Responsibilities**: Table 7.1 and 7.2 |
| MVC pattern is adapted to *User App* design for 3rd party developers to build various user interfaces easily. The form of the applications varies in accordance with the purpose of the application. For instance, end-user applications may have user friendly GUI which enables user to easily determine sensor values and to control actuator installed at home. In the other hand, applications for the data analytic people or for operators of building management may require different UI presentation; can be GUI or command line interface. However, the model and the control of the applications are most likely similar. MVC is the exemplary pattern for the purpose. (*QA09. Extensibility*) |

7.2 Static Perspective

The static structure of the *User App* is presented by Figure 7.2. Components shown in Figure 7.1 are mapped into the packages of same name in Figure 7.2. (i.e. *Model Package* is compiled to the run time object, *Model*)

**Figure 7.2 Static Perspective of User App.**

**Table 7.4 Element Responsibilities Catalog for Static Perspective of User App**

|  |  |
| --- | --- |
| **Perspective:** Static  **Associated Drawings:** Figure 7.2. | |
| **Element** | **Responsibilities** |
| *View package* | * This package implements user interface. |
| *Control package* | * *Control* *package* uses *View* package. * *Control* package consists of *Controller*, *Server* interface, *HTTPAdaptor* class, *HTTPRESTClient* class, and *MQTTClient* class. |
| *Controller* | * This element is a class. * Use Server to connect to Web Server for following operations: * Login, user accounts management, SA nodes management, and log presentation. * Use MQTTClient to connect to *Event Bus* for following operations: * Publication of SA node control event, subscription of events from *SA Node* and *Event Manager*. |
| *Server* | * This element is an interface. * Interface of HTTP REST API to connect to *Web Server*. |
| *HTTPAdaptor* | * This element is a class. * Implement *Server*. * Use *HTTPRESTClient* class to implement HTTP REST APIs to connect to *Web Server*. * Refer to section 9.3 for HTTP REST API specification. |
| *HTTPRESTClient* | * This element is a class. * Implement HTTP client. |
| *MQTTClient* | * This element is a class. * Implement MQTT client to connect to *Event Bus*. |
| *Model package* | * This package implements the SA node model, and IoT dataset model. * *Model package* consists of *Node*, *SensorActuator*, and *IoTDataset*. |
| *Model* | * This element is a class. * Use Node to capture the model of SA node. * Use IoTDataset to capture the model of analytic logs produced by the system components. |
| *Node* | * This element is a class. * Have multiple *SensorActuator* to capture the model of sensors and actuators of SA node. * Define SA node properties; ID, serial number, nickname, and SA node access permission. |
| *SensorActuator* | * This element is a class. * Model sensors and actuators. * Define sensor properties; name, value, value range, controllable. |
| *IoTDataset* | * This element is a class. * Model IoT analytic logs. * Define structure of the log. * Implement analytic functions. (TBD – Out of the project scope) |

**Table 7.5 Relationship Responsibilities Catalog for Static Perspective of User App**

|  |  |
| --- | --- |
| **Perspective:** Static  **Associated Drawings:** Figure 7.2. | |
| **Relationship** | **Responsibilities** |
|  | * A and B are associated each other. * *View* observes the model. * Whenever model status changes, *View* updates the view presented to user. |
|  | * A uses B. |
|  | * A implements B |
|  | * A specializes B |

**[Table 7.6] Rationale of Static Perspective of User App**

|  |
| --- |
| **Perspective:** Static  **Associated Drawings:** Figure 7.2  **Associated Responsibilities**: Table 7.4 and 7.5 |
| The UI of the IoT applications varies in accordance with the purpose of the application, but the model and the control of the applications are most likely similar. *Control package* includes the implementation of essential functionalities that are required by applications; login, user accounts management, SA nodes management, log presentation, and pub-sub. *Model package* includes expandable SA node model, sensor model, actuator model, and analytic data model. These two packages are provided to the 3rd party developers for easy software development. (*QA09. Extensibility*)  While *IoT Server* only allows HTTPS - STATEless protocol for the log access, the STATEful protocol is required for real time data analytic operations. Later version of the system may support such protocols. Therefore, the implementation of *IoT Server* client, *HTTPAdaptor*, is hidden from the user by providing the *Server* interface. |

8. DECOMPOSITION OF SA NODE

The detailed decomposition of *SA node* is described through the sub sections, and the rationales of selected design choices are presented as well. *SA node* is the one of the major component of *IoT System* as shown in Figure 5.1.

8.1 Static Perspective - 1st Decomposition of SA node

*SA node* is designed as the layered pattern for the modifiability and portability. Abstract Layer consists of two modules; *SA Controller* and *SA Network Manager*. *SA Controller* provides the abstraction of the sensor and actuator to support different hardware of SA. *SA Network Manager* provides the abstraction of publish-subscribe event bus to support emerging event bus and network protocol.

Modules - *Arduino Driver* and *MQTT Adaptor* in Concrete Layer implement *SA Controller* and *SA Network Manager* in the abstract layer respectively.

Finally, *SA Manager* in the middleware layer has the logic and policy of the functional requirements of the SA node (i.e. open or close the door upon the arrival of door control event), and they are implemented by using the modules in the abstract layer.

**Figure 8.1 First Level of Decomposition - Decomposition of SA Node**

**Table 8.1 Element Responsibility Catalog for First-Level Decomposition**

|  |  |
| --- | --- |
| **Perspective:** Static  **Associated Drawings:** Figure 8.1 | |
| **Element** | **Responsibilities** |
| *SA Manager* | * This element implements logics and policies of the functional requirements of *SA node*. * Responsible to monitor the status of the sensor from *SA Controller* and publish the status through *SA Network Manager*. * Uses SA controller to control the actuator after subscribing to the event through SA Network Manager. |
| *SA Controller* (HAL) | * This element hides underlying driver implementations of sensors and actuators. * Include the generic HAL interface that doesn’t need to be changed even if sensor and actuator are replaced. * Various H/W drivers can be supported by implementing this generic HAL interface. |
| *SA Network Manager* | * This element hides the implementation details of the event bus. * Define the generic publish-subscribe interface of the event bus. * *MQTT Adaptor* should be implemented from this interface. * Provide the communication channel for the emerging protocol. * This communication for the emerging protocol is encapsulated from *SA Manager*. * See “8.2 2nd Decomposition of SA Network Manager” for more detail. |
| *Arduino Driver* | * This element is the driver for the Arduino based sensor and actuator. * Arduino *Driver* should be implemented in accordance with *SA Controller*. |
| *MQTT Adaptor* | * This element is the adaptor that is implemented by MQTT event bus. * MQTT *Adaptor* should be implemented in accordance with *SA Network Manager*. |

**Table 8.2 Relationship Responsibility Catalog for First-Level Decomposition**

|  |  |
| --- | --- |
| **Perspective: Dynamic**  **Associated Drawings: Figure 8.1** | |
| **Relationship** | **Responsibilities** |
| B  A | * This relationship indicates that element A is allowed to use the function of B. The relations must be unidirectional. |

**Table 8.3 First Level of SA node Decomposition Rationale**

|  |
| --- |
| **Perspective: Static**  **Associated Drawings: Figure 8.1**  **Associated Responsibilities: Tables 8.1 and 8.2** |
| This decomposition is composed by the layered pattern to promote the modifiability and portability of SA node program.  First, it minimizes the effort required to transfer SA node program from one hardware of the sensor and actuator to another. *SA controller* in the abstract layer provides the HAL interface of the sensor and actuator that need to be changed if the hardware of SA is replaced,  SA node program can be easily ported to various hardware, if they are implemented from HAL interface. (High portability for the different hardware)  Second, it reduces the cost of adding emerging event bus and network protocol. SA network manager in the abstract layer provides the generic publish-subscribe interface. High layer modules use the interface, not using concrete MQTT event bus directly. So, the higher layer modules are not required to be changed, even though MQTT is replaced to other event bus. As a result, this structure makes it easy to add or change the emerging event bus and network protocols.(High modifiability for the emerging event bus and network protocols) |

8.2 Dynamic Perspective - 2nd Decomposition of SA Network Manager

SA node communicates with the *IoT Server* and *Event Bus* via WIFI using HTTP and MQTT protocol respectively. However, SA node is designed to support the emerging protocol without WIFI. *SA Network Manager* encapsulates the concrete network protocol-specific functionality from *SA manager* and other module instances. *Network broker* in *SA Network Manager* locates an appropriate network bus in accordance with the network capability of SA node. In addition, it provides the bridge between SA node with the emerging protocol and SA node with WI-FI. SA node with WI-FI communicates with the system in behalf of the node without WI-FI. They have the relationship of master-slave and are named “Master SA node” and “Slave SA node”.

**Figure 8.2 Second Level of Decomposition - Decomposition of SA Network**

**Table 8.4 Element Responsibility Catalog for Second-Level Decomposition**

|  |  |
| --- | --- |
| **Perspective:** Dynamic  **Associated Drawings:** Figure 8.2 | |
| **Element** | **Responsibilities** |
| *Network Broker* | * This element is the intermediary that locates an appropriate network bus in accordance with the network capability of SA node. * If SA node only supports the emerging protocol without WI-Fl, than the *Network Broker* sends the event to the other node’s *Network Broker* through the *Emerging Protocol Bridge*. (They are named as “Master SA node” and “Slave SA node”) * Master SA node communicates with the event bus in behalf of Slave SA node. |
| *Emerging Protocol Bridge* | * This element mediates between the local network broker and *Emerging Protocol Bridge* of remote network broker in other SA node. * *Emerging Protocol Bridge* encapsulates the network protocol-specific functionality from the network broker. |
| *Publish-subscribe Bus Interface* | * This element is the abstract of the event bus. * Call the MQTT *adaptor* for delegating the request from the network broker. |
| *MQTT Adaptor* | * This element is the adaptor to control MQTT event bus that has the real event bus mechanisms. |

**Table 8.5 Relationship Responsibility Catalog for Second-Level Decomposition**

|  |  |
| --- | --- |
| **Perspective: Dynamic**  **Associated Drawings: Figure 8.5** | |
| **Relationship** | **Responsibilities** |
| B  A | * This relationship indicates that element A calls functionality of B. This is just function call between the module instances at run time. |
| B  A | * This relationship indicates that element A communicates with by event at runtime. |
| B  A | * This relationship indicates the communication through emerging protocol. |

**Table 8.6 2nd Level Decomposition of SA Network Manager - Decomposition Rationale**

|  |
| --- |
| **Perspective: Dynamic**  **Associated Drawings: Figure 8.2**  **Associated Responsibilities: Tables 8.4 and 8.5** |
| This decomposition is composed by the light broker pattern to promote the modifiability and interoperability about adding new emerging protocols like Bluetooth without WIFI. Other module instances like SA manager only calls the *Network Broker* without knowing about the concrete network and protocol. The *Network Broker* locates an appropriate network bus in accordance with the network capability of SA node. If the SA node supports the WIFI, then the *Network Broker* calls the *Publish-subscribe Bus Interface* to send or receive the event. If the SA node supports the emerging protocol without WIFI, then the event is transferred to other SA node with WIFI and emerging protocol. (They are named “Master SA node” and “Slave SA node”.) At that time, The communication between Master SA node and Slave SA node are operated through the *Emerging Protocol Bridge*. Master SA node has the responsibility to communicate with the event bus on behalf of Slave SA node.  This structure provides the high interoperability between SA nodes based on different network protocol and system (High interoperability between SA node with different network protocol)  *Emerging protocol bridge* encapsulates the network protocol-specific functionality from the *Network Broker* and other module instances. So, it provides high modifiability for new emerging protocol requirement, by adding and developing new bridge without the change of other module instances. (High modifiability for new emerging protocol.) (*QA07 Modifiability*) |

9. SYSTEM DESIGN SUPPLEMENT

This section provides the supplementary system designs which can help developers better understand how the three major system entities – *IoT Server*, *SA Node*, and *User App* interact with each other in different scenarios.

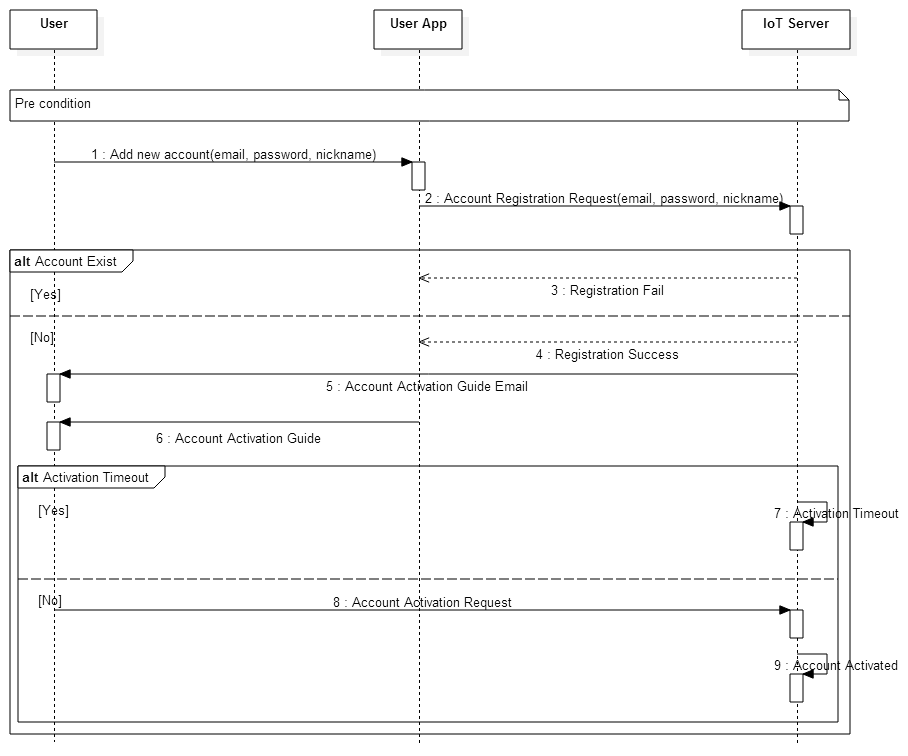
9.1 Sequence diagram

Email user verification is the general security measure that many web services use for the user account registration. *IoT System* uses similar technique to promote the security of the system in some extent. Upon the user account registration request, *IoT Server* sends an email to the user asking to activate the account by clicking the link included in the email. Figure 9.1 describes the sequence of user account registration, and how *IoT Server* and *User App* interact with user in the registration scenario.

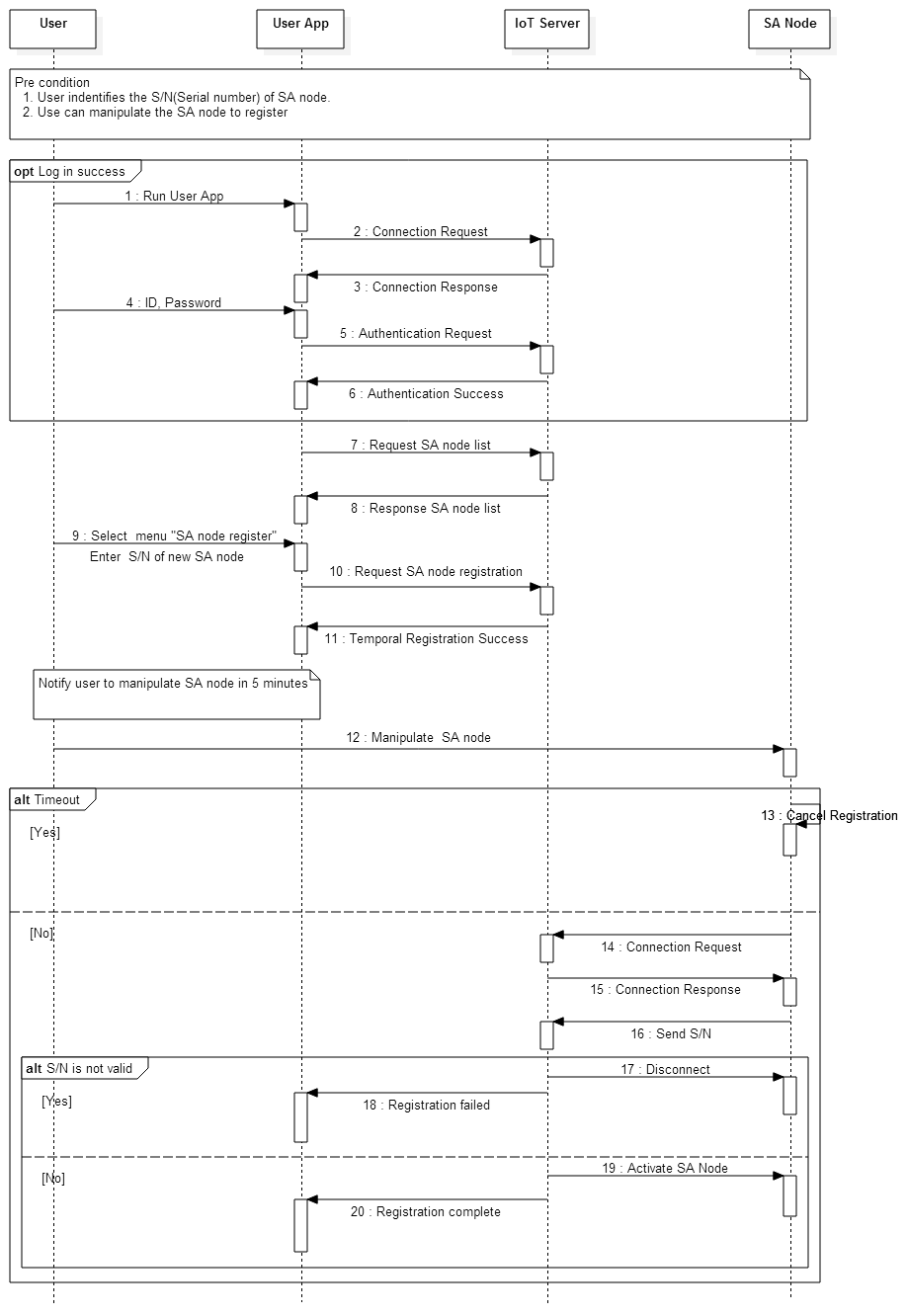
The system must provide best efforts to prevent the malicious use of the system. Only owner should be able to register the SA nodes. To achieve this quality attribute (*QA02 Security*), two key security steps are included in the SA node registration scenario:

1. User must know the serial number of the SA node to register.
2. User is required to manipulate the SA node within the specific time frame to send the message to the system to complete the registration. Mechanism may vary depending on the kind of input method that the SA node have. For instance, pressing the specific button on the node or restarting the node.

Figure 9.2 shows the sequence of SA node registration.



**Figure 9.1 Sequence Diagram - User account registration**



**Figure 9.2 Sequence Diagram - SA node registration**

9.2 User Case Scenario

User case scenarios, presented in this section break down the high level functional requirements into detailed scenarios.

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Add new user account | | **Use Case ID**:  01 |
| **General use case description**:  This use case describes how user adds a new account to the system. | | |
| **Entities involved**:  E01 – User  E02 - End user application  E03 – System | | |
| **Preconditions**: | | |
| **Primary use case flow of events**: | | |
| 1. | E01 elects to add a new user account. | |
| 2. | E02 prompts E01 to enter registration information including id and password. | |
| 3. | E01 elects to submit the registration information. | |
| 4. | E02 requests a new account registration with information to E03. | |
| 5. | E03 validates information, adds the temporal account, and sends the account activation mail to E01. | |
| 6. | E03 sends the confirmation to E02. | |
| 7. | E02 presents the guide to activate the account to E01. | |
| 8. | E01 activates the account by following the instruction described in the account activation mail. | |
| 9. | E03 activates the account. | |
| **Alternative Flows:**  **\* Invalid account registration information**  5.1. If the registration information is invalid, the request is rejected.  E03 sends error message to E02.  Proceed to step 7 of the primary use case.  **\* Account activation timeout**  8.1 If E03 doesn’t receive the activation request for a specific amount of time, the account registration is canceled. | | |
| **Postconditions**:  A new user account was registered. | | |

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Login | | **Use Case ID**:  02 |
| **General use case description**:  This use case describes how user logs into the system. | | |
| **Entities involved**:  E01 – User  E02 - End user application  E03 – System | | |
| **Preconditions**:  User account is registered to the system | | |
| **Primary use case flow of events**: | | |
| 1. | E01 elects to login to the system. | |
| 2. | E02 prompts E01 for id and password. | |
| 3. | E01 enters id and password. | |
| 4. | E02 requests the login with id and password to E03. | |
| 5. | E03 validates the entered id and password and logs E02 into the system. | |
| 6. | E03 sends the confirmation to E02. | |
| 7. | E02 presents the results to E01. | |
| **Alternative Flows:**  **\* Invalid id and/or password**  5.1. If id and/or password is invalid E03 rejects E02.  E03 sends an error message to E02. Proceed to step 7 of the primary use case. | | |
| **Postconditions**:  User was logged into the system. | | |

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Register new SA node | | **Use Case ID**:  03 |
| **General use case description**:  This use case describes how SA node is registered to the system. | | |
| **Entities involved**:  E01 – User  E02 - End user application  E03 – System  E04 – SA Node | | |
| **Preconditions**:  User is logged into the system. | | |
| **Primary use case flow of events**: | | |
| 1. | E01 elects to register E04. | |
| 2. | E02 prompts E01 for serial number of the node. | |
| 3. | E01 enters the serial number. | |
| 4. | E02 requests a new SA node registration with the serial number to E03. | |
| 5. | E03 validates the serial number, and waits for the connection request of E04. | |
| 6. | E03 sends confirmation of the temporal registration to E02. | |
| 7. | E02 guides E01 to manipulate E04 to establish a connection with E03. | |
| 8. | E01 manipulates E04 for the connection between E04 and E03. | |
| 9. | Proceed to step 1 of primary use case of UC05. | |
| 10. | E03 adds E04 to the user account. | |
| 11. | E03 sends the confirmation to E02. | |
| 12. | E02 presents the result to E01. | |
| **Alternative Flows:**  **\* Invalid serial number**  6.1. If the serial number is already registered, temporally registered, or invalid, E03 sends error message to E02.  E02 presents the result to E01.  The use case ends.  **\* Connection request timeout**  8.1. If E03 doesn’t receive the connection request from E04 for a specified amount of time, E03 revokes the temporal registration, and sends error message to E02.  E01 presents the result to E01.  The use case ends. | | |
| **Postconditions**:  A new node is registered to the system | | |

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Unregister SA node | | **Use Case ID**:  04 |
| **General use case description**:  This use case describes how SA node is unregistered to the system. | | |
| **Entities involved**:  E01 – User  E02 - End user application  E03 – System | | |
| **Preconditions**:  User is logged into the system. | | |
| **Primary use case flow of events**: | | |
| 1. | E01 elects to unregister E04. | |
| 2. | E02 prompts E01 to select a node to unregister. | |
| 3. | E01 selects the node. | |
| 4. | E02 requests the unregister of the node to E03. | |
| 5. | E03 unregisters the node, and disconnects it if connected. | |
| 6. | E03 sends the confirmation to E02. | |
| 7. | E02 presents the result to E01. | |
| **Postconditions**:  SA node was unregistered. | | |

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Request SA node connection | | **Use Case ID**:  05 |
| **General use case description**:  This use case describes how SA node establishes connection to the system. | | |
| **Entities involved**:  E03 – System  E04 – SA Node | | |
| **Preconditions**: | | |
| **Primary use case flow of events**: | | |
| 1. | E04 attempts to connect to E03. | |
| 2. | E03 grants the connection request | |
| **Alternative Flows:**  **\* Invalid connection request**  2.1. If E04 is not registered to E03, E03 rejects the request.  The use case ends. | | |
| **Postconditions**:  The node was connected to the system. | | |

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Discover SA Nodes | | **Use Case ID**:  06 |
| **General use case description**:  This use case describes how user discovers installed SA nodes, and equipped sensors and actuators. | | |
| **Entities involved**:  E01 – User  E02 - End user application  E03 – System | | |
| **Preconditions**:  Nodes are registered to the system.  User is logged into the system. | | |
| **Primary use case flow of events**: | | |
| 1. | E01 elects to discover SA nodes. | |
| 2. | E02 requests SA node information to E03. | |
| 3. | E03 sends the nodes information including the list of equipped sensors/actuators to E02. | |
| 4. | E02 presents the information to E01. | |
| **Postconditions**:  Node information were presented. | | |

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Determine presence/proximity | | **Use Case ID**:  07 |
| **General use case description**:  This use case describes how user determines presence/proximity sensor. | | |
| **Entities involved**:  E01 – User  E02 - End user application  E03 – System  E04 – SA Node | | |
| **Preconditions**:  User is logged into the system. | | |
| **Primary use case flow of events**: | | |
| 1. | E01 elects to determine presence/proximity. | |
| 2. | E02 requests presence/proximity sensor value to E03. | |
| 3. | E03 requests the value to E04. | |
| 4. | E04 reads the sensor, and sends the value to E03. | |
| 5. | E03 sends the value to E02. | |
| 6. | E02 presents the value to E01. | |
| **Postconditions**:  The value of the presence/proximity sensor was presented. | | |

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Determine temperature/humidity | | **Use Case ID**:  08 |
| **General use case description**:  This use case describes how user determines temperature/humidity. | | |
| **Entities involved**:  E01 – User  E02 - End user scenario  E03 – System  E04 – SA Node | | |
| **Preconditions**:  User is logged into the system. | | |
| **Primary use case flow of events**: | | |
| 1. | E01 elects to determine temperature/humidity. | |
| 2. | E02 requests temperature/humidity sensor value to E03. | |
| 3. | E03 requests the value to E04. | |
| 4. | E04 reads the sensor, and sends the value to E03. | |
| 5. | E03 sends the value to E02. | |
| 6. | E02 presents the value to E01. | |
| **Postconditions**:  The value of the temperature/humidity sensor was presented. | | |

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Control door light | | **Use Case ID**:  09 |
| **General use case description**:  This use case describes how user turns on or off the light. | | |
| **Entities involved**:  E01 – User  E02 - End user application  E03 – System  E04 – SA Node | | |
| **Preconditions**:  User is logged into the system. | | |
| **Primary use case flow of events**: | | |
| 1. | E01 elects to control the light. | |
| 2. | E02 requests light control to E03. | |
| 3. | E03 requests to E04. | |
| 4. | E04 controls the light, and sends the status of light to E03. | |
| 5. | E03 sends the status to E02. | |
| 6. | E02 presents the status of light to E01. | |
| **Post conditions**:  The light was turned on or off as user requested. | | |

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Control door | | **Use Case ID**:  10 |
| **General use case description**:  This use case describes how user opens or closes the door. | | |
| **Entities involved**:  E01 – User  E02 - End user application  E03 – System  E04 – SA Node | | |
| **Preconditions**:  User is logged into the system. | | |
| **Primary use case flow of events**: | | |
| 1. | E01 elects to control the door. | |
| 2. | E02 requests door control to E03. | |
| 3. | E03 requests to E04. | |
| 4. | E04 controls the door. | |
| 5. | E04 sends the status of door to E03. | |
| 6. | E03 sends the status to E02. | |
| 7. | E02 presents the status of door to E01. | |
| **Alternative Flows**  **Alarmed**  4-1. If the alarm is set, step 4 is skipped and proceeds to step 5 of the “Primary use case flow”. | | |
| **Postconditions**:  The door was opened or closed as user requested if the alarm was not set. | | |

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Control alarm | | **Use Case ID**:  11 |
| **General use case description**:  This use case describes how user sets the alarm on or off. | | |
| **Entities involved**:  E01 – User  E02 - End user application  E03 – System  E04 – SA Node | | |
| **Preconditions**:  User is logged into the system. | | |
| **Primary use case flow of events**: | | |
| 1. | E01 elects to set the alarm. | |
| 2. | E02 requests alarm setting to E03. | |
| 3. | E03 requests to E04. | |
| 4. | E04 sets the alarm, and sends the status of the alarm to E03. | |
| 5. | E03 sends the status to E02. | |
| 6. | E02 presents the status of the alarm to E01. | |
| **Postconditions**:  The alarm was set or unset as user requested. | | |

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Log user commands. | | **Use Case ID**:  12 |
| **General use case description**:  This use case describes how user commands are stored in IoT infrastructure | | |
| **Entities involved**:  E02 – End users application  E03 – System | | |
| **Preconditions**:  User is logged into the system. | | |
| **Primary use case flow of events**: | | |
| 1. | E02 sends command to E03. | |
| 2. | E03 stores the command. | |
| **Postconditions**:  User command logs were stored. | | |

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Log sensor values | | **Use Case ID**:  13 |
| **General use case description**:  This use case describes how sensor values are stored in IoT infrastructure | | |
| **Entities involved**:  E03 – System  E04 – SA Node | | |
| **Preconditions**:  SA Node is registered to the system. | | |
| **Primary use case flow of events**: | | |
| 1. | E04 reports sensor value to E03 when changed | |
| 2. | E03 stores reported sensor value. | |
| **Postconditions**:  Sensor values were stored. | | |

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Configure log time window | | **Use Case ID**:  14 |
| **General use case description**:  This use case describes how log time window is set. | | |
| **Entities involved**:  E01 – User  E02 - End user application  E03 – System | | |
| **Preconditions**:  User is logged into the system. | | |
| **Primary use case flow of events**: | | |
| 1. | E01 elects to set log time window. | |
| 2. | E02 requests E03 for the current configuration. | |
| 3. | E03 sends the configurations to E02. | |
| 4. | E02 presents the configuration to E01. | |
| 5. | E01 enters the new configuration, and elects to update the configuration. | |
| 6. | E02 requests configuration updates to E03. | |
| 7. | E03 applies configurations. | |
| 8. | Repeat 2 ~ 7 as necessary | |
| **Post conditions**:  New configurations were applied. | | |

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Review logs | | **Use Case ID**:  15 |
| **General use case description**:  This use case describes how user reviews sensor state and user command history. | | |
| **Entities involved**:  E01 – User  E02 - End user application  E03 – System | | |
| **Preconditions**:  User is logged into the system. | | |
| **Primary use case flow of events**: | | |
| 1. | E01 elects to review the logs. | |
| 2. | E02 requests logs to E03. | |
| 3. | E03 sends logs of specified time window to E02. | |
| 4. | E02 presents logs to E01. | |
| **Post conditions**:  User command and sensor state logs were presented to the user. | | |

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Send emergency message | | **Use Case ID**:  16 |
| **General use case description**:  This use case describes how an emergency message is sent to user when the door is opened manually or the house is occupied suddenly while the alarm is set. | | |
| **Entities involved**:  E01 – User  E02 - End user application  E03 – System  E04 – SA Node | | |
| **Preconditions**:  The alarm is set. | | |
| **Primary use case flow of events**: | | |
| 1. | E04 detects door opening or detects the presence of moving object(s). | |
| 2. | E04 sends the emergency message to E03. | |
| 3. | E03 sends the message to E02. | |
| 4. | E02 presents the emergency message to E01. | |
| **Postconditions**:  Emergency message was presented to the user. | | |

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Lock house automatically | | **Use Case ID**:  17 |
| **General use case description**:  This use case describes how the house is locked automatically. | | |
| **Entities involved**:  E01 – User  E02 - End user application  E03 – System  E04 – SA Node | | |
| **Preconditions**:  The alarm is not set. | | |
| **Primary use case flow of events**: | | |
| 1. | E04 detects the vacancy of the house for 5 minutes. | |
| 2. | E04 sends inform message to E03. | |
| 3. | E03 sends the message to E02. | |
| 4. | E02 presents the message to E01. | |
| 5. | E01 elects to lock the house. | |
| 6. | E02 requests E03 to lock the house. | |
| 7. | E03 requests to E04. | |
| 8. | E04 closes door and sets the alarm. | |
| **Alternative Flows:**  **\* Not to lock the house**  5.1. If E01 elects not to lock the house, E02 requests E03, and E03 requests E04 not to lock the house.  The use case ends.  **\* Response timeout**  5.1. If E04 does not receive the response from E01, proceed to step 8 of the “Primary use case flow”. | | |
| **Postconditions**:  The door was closed and the alarm is set. | | |

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Turn off light automatically | | **Use Case ID**:  18 |
| **General use case description**:  This use case describes how the light is turned off automatically. | | |
| **Entities involved**:  E04 – SA Node | | |
| **Preconditions**:  Light is on. | | |
| **Primary use case flow of events**: | | |
| 1. | E04 detects the vacancy of the house, and starts timer for 10 minutes. | |
| 2. | E04 turns off lights when timer is expired | |
| **Alternative Flows:**  **\* Not to lock the house**  1.1. If E04 detects the light being turn off, or presents of people, the timer is canceled.  The user case ends. | | |
| **Postconditions**:  Lights were turned off | | |

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Grant SA node access permission | | **Use Case ID**:  19 |
| **General use case description**:  This use case describes how the owner of the SA node grants node access permission to other user account. | | |
| **Entities involved**:  E01 – User  E02 – End user application  E03 - System | | |
| **Preconditions**:  User is logged into the system. | | |
| **Primary use case flow of events**: | | |
| 1. | E01 elects to grant node access permission to other user. | |
| 2. | E02 requests a list of owning nodes to E03. | |
| 3. | E03 sends the list. | |
| 4. | E02 requests a list of “friend” account to E03. | |
| 5. | E03 sends a list of “friend” account to E02. | |
| 6. | E02 presents the list of nodes and account. | |
| 7. | E01 selects a node and an account. | |
| 8. | E02 requests a node access permission assignment to E03. | |
| 9. | E03 updates the account, and sends a confirmation to E02. | |
| **Postconditions**:  An access permission of the selected node was assigned to the selected account. | | |

|  |  |  |
| --- | --- | --- |
| **Use case title**:  Transfer SA node ownership | | **Use Case ID**:  20 |
| **General use case description**:  This use case describes how the owner of the SA node transfers the ownership to other user account. | | |
| **Entities involved**:  E01 – User  E02 - End user application  E03 – System | | |
| **Preconditions**:  User is logged into the system. | | |
| **Primary use case flow of events**: | | |
| 1. | User elects to transfer the node ownership to other user account. | |
| 2. | E02 requests a list of owning nodes to E03. | |
| 3. | E03 sends the list. | |
| 4. | E02 requests a list of “friend” account to E03. | |
| 5. | E03 sends a list of “friend” account to E02. | |
| 6. | E02 presents the list of nodes and account. | |
| 7. | E01 selects a node and an account. | |
| 8. | E02 requests the ownership transfer to E03. | |
| 9. | E03 updates accounts, and sends a confirmation to E02. | |
| **Postconditions**:  An ownership of the selected node was assigned to the selected account.  User lost the ownership of the selected node, but kept an access permission of the node. | | |

9.3 API Specifications

*User App*, *SA Node*, and *IoT Server* communicates by exchanging event messages. The event message contains “topic” and “payload”. The topic is the subject of the message. System entities publish and/or subscribe specific topics, and the system enforces the entities to publish and/or subscribe only permitted messages. In this sense, the message is delivered to rightful clients. The payload contains the type of the event and necessary values. “Team1\_03\_Event Bus Interface Specification\_v1.0.doc” [3] specifies the event message format.

*User App* and *SA Node* access *IoT Server* by REST web APIs. Following services are provided by the API: add new user, retrieve user information, validate token, refresh token, revoke token, manage user account, and manage SA node. “Team1\_04\_REST Web API Specification\_v1.0.doc” [4] specifies the APIs.

APPENDIX A – EVENT BUS EXPERIMENTS

This section provides the experiments result of the performance of *Event Bus*. The experiments affect the following two decisions.

* The *Heartbeat* event rate.
* Max number of *SA node* in single server

A.1 Test Environment

**Table A.1 Test Environment of IoT Server and SA node**

|  |  |
| --- | --- |
| **Element** | **Environment** |
| IoT Server | * Cloud : AWS EC2 Instance (m3.medium) * OS : Ubuntu 14.04.2 LTS (GNU/Linux 3.13.0-48-generic x86\_64) * CPU : 1 x Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz * MEM : 3.75GB |
| SA node | * Virtual SA nodes that run in same environment with Server. |

A.2 Test Scenario

* *User App* sends the control event to single *SA node*.
* *SA node* controls the *Actuator* and sends the status of *Sensor*.
* *User App* receives the status event.

**Figure A.1 Dynamic Perspective for Test Scenario**

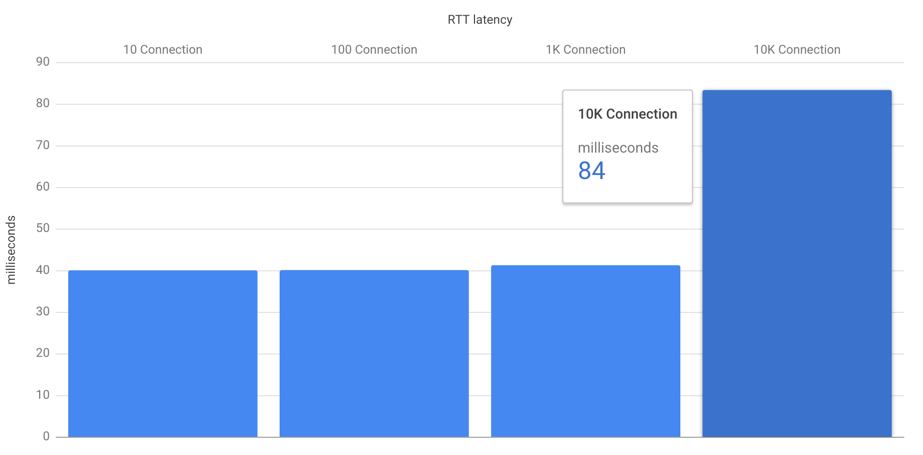
A.3 Experiment [1]

* Pre-Condition

All of *SA node*s send the Heartbeat events every 10 seconds.

* Test Result

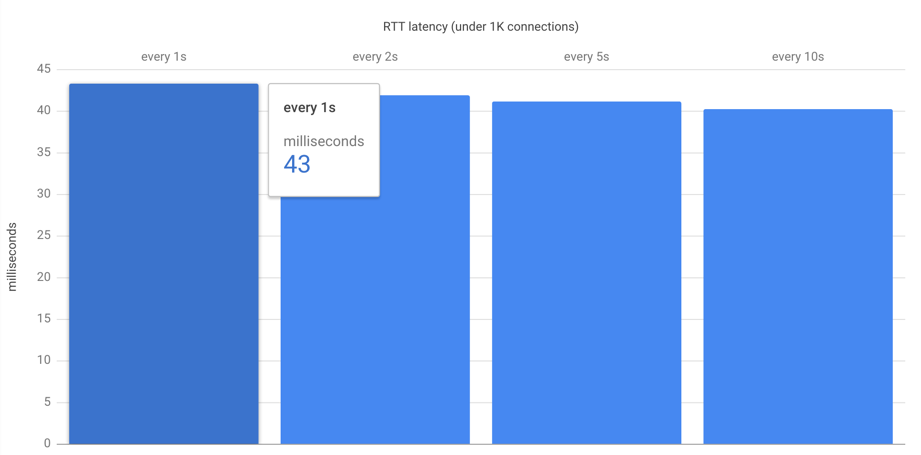
Max 1000 SA nodes don’t effect on the performance. (Figure A.2)



**Figure A.2 Measurement Result ( X-axis : N of SA node, Y-axis : t4 – t1 )**

A.4 Experiment [2]

* Pre-Condition  
  1000 *SA nodes* are connected to the *Event Bus*.
* Test Result  
  Less than 10sec *Heartbeat* does not effect on the performance. (Figure A.3)



**Figure A.3 Measurement Result ( X-axis : N sec cycle, Y-axis : t4 – t1 )**