|  |
| --- |
| LGE – CMU SW Architect course 2016 |
| Architecture Document |
| TEAM 2 – NOT YET |
|  |
| **Wonhong Kwon (wonhong.kwon@lge.com)** |
| **2016-06-22** |

|  |
| --- |
|  |

Table of Contents

[0. Document Description 2](#_Toc454337168)

[**Document Revision** 3](#_Toc454337169)

[**Terms and acronyms** 3](#_Toc454337170)

[**References** 3](#_Toc454337171)

[1. Project Overview 4](#_Toc454337172)

[2. Architecture Drivers 5](#_Toc454337173)

[**Influential architecture drivers** 5](#_Toc454337174)

[3. System Context Diagram 6](#_Toc454337175)

[4. Architecture Design 8](#_Toc454337176)

[**Design Decisions & Alternatives** 9](#_Toc454337177)

[#1: Patterns for high-level architecture 9](#_Toc454337178)

[#2: Tactics for promoting availability 11](#_Toc454337179)

[#3: Relief the negative effects of the tactics 12](#_Toc454337180)

[#4: Tactics for promoting security 13](#_Toc454337181)

[**Dynamic view of architecture design** 14](#_Toc454337182)

[High-level architecture 14](#_Toc454337183)

[Decomposition of Communication Manager 16](#_Toc454337184)

[Sequence diagrams 17](#_Toc454337185)

[**Static view of architecture design** 26](#_Toc454337186)

[High-level architecture 26](#_Toc454337187)

[Decomposition of Communication Manager 30](#_Toc454337188)

[**Physical view of architecture design** 31](#_Toc454337189)

[High-level architecture 31](#_Toc454337190)

0. Document Description

The intent of this document is to describe how Team-2 (NOT YET) has designed the architecture of the Sure-Park system. We hope that audiences can understand what the Sure-Park system is, how the system interacts with external entities, which decisions have been made for promoting important architecture drivers, and how we’ve designed the architecture of the system.

Section 1 is to describe what the Sure-Park system is and who stakeholders of the project are. Section 2 describes influential architecture drivers which impact on the system significantly. Section 3 is to define how the system interacts with external entities with system context diagram, and section 4 is dedicated to describe what the architecture of the system is in static, dynamic, and physical perspective.

## **Document Revision**

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Description | Author |
| 0.1 | 2016.06.08 | Initial version | Wonhong Kwon |
| 0.2 | 2016.06.11 | Updates design alternatives | Wonhong Kwon |
| 0.3 | 2016.06.13 | Updates the architecture design | Wonhong Kwon |
| 1.0 | 2016.06.19 | Version 1.0 release | Wonhong Kwon |
| 1.1 | 2016.06.22 | Refine architecture design | Wonhong Kwon |

## **Terms and acronyms**

Sure-Park system: parking garage management system for which the project is established

Parking facility: a parking garage including parking controller (Arduino); it doesn’t include the system SW running on the parking controller.

Grace-period: time parking spot will be held even if driver don’t show-up at the start of their reservation time

Parking controller: HW (Arduino) controller

CCPS: credit card payment system which is owned by credit card companies

## **References**

[1] [TEAM2\_REFERENCE\_01] Sure-Park Parking Garage Management System Project Description - V2

[2] [TEAM2\_DOC\_02] Architecture\_Driver\_Specification\_v1.2

[3] [TEAM2\_REFERENCE\_03] Architectural Patterns for Enabling Application Security

[4] Detail Design Document

[5] <http://mosquitto.org/>

[6] <https://github.com/knolleary/pubsubclient>

[7] <http://www.eclipse.org/paho/>

[8] [TEAM2\_DOC\_04] Protocol Design

1. Project Overview

The goal of this project (based on [1]) is to design and implement the Sure-Park system that enables drivers to find and reserve parking spaces quickly and efficiently and provides monitoring parking facilities operations. The project sponsor, Geoff’s Transportation and Parking Services (GTPS LLC), has concerns about inefficiencies in their parking garages, so key motivations for this project are to:

* enable drivers to easily find a parking facility and reserve it
* utilize the space in the parking facility more efficiently to increase profits
* reduce traffic congestion and the chance for accidents inside the parking facilities reducing liabilities
* utilize personnel more efficiently and reduce the number of people needed to operate the parking facility reducing operating costs

However, this project is NOT a real project and is only for educational purpose. Some requirements are out of our scope for the purpose to more focus on the architecture design within the time limit. You can see the actual requirements this project focuses on in the [2].

Many stakeholders are related to this project and they are:

* Geoff’s Transportation and Parking Services (GTPS LLC) also known as Owner
* Anthony J. Lattanze (Representative of GTPS LLC as well as the instructor)
* Attendants of the parking facilities also known as Dave
* Drivers who want to reserve parking spaces
* Operators who install and manage the parking facilities
* Credit card companies
* Developers who actually build the system
* Project manager who manages the project

2. Architecture Drivers

We’ve focused on the following architecture drivers to design our architecture. You can refer to [2] to see the entire list of architecture drivers.

## **Influential architecture drivers**

**Schedule & Resource:** We just have 5 weeks to finish the project, but we only have 6 developers.

**Scalability:** Easy to install new parking slots to an existing parking facility or set-up a new parking facility

A new facility controller is introduced; the system is able to recognize this new HW without having to re-start the system or changing existing processing components.

**Security:** Prevent unauthorized users from accessing the information related to operating facility

When a malicious attacker who doesn’t have any authorization tries to access the information, the system doesn’t allow it and notify it to an operator within 30 seconds when the system detects an attack.

**Availability:** Allow drivers to reserve parking spaces and use parking facilities even if the system or the facility has some problems

Power suddenly went out in a facility; the system is able to detect the power outage and calls an attendant of the facility in a minute to take care of the facility manually. Also, the system is able to call local power supplier to fix the facility’s power outage. During the repair of the facility, the system doesn’t allow any reservation on the facility.

A facility controller is out of order; the system is able to detect it and call an attendant of the facility in a minute to take care of the facility manually and fix/replace the controller to run the facility again. During the repair of the facility, the system doesn’t allow any reservation on the facility.

The system is down when driver is making reservation of a parking slot; the back-up system is able to detect this fault and works as usual without having any person’s intervention in a minute.

3. System Context Diagram

This section describes how the system interacts with external entities and the boundary of the system.

Figure 1 shows the system context diagram of our system. Driver is a person who wants to reserve parking spaces and the system requires driver’s information to reserve spots in parking garages. Attendant is able to get the system status such as which parking spots are open and which are occupied, how long a car has occupied a particular parking spots, and whether a driver parks in the wrong parking space and system reassign parking spaces and correlate associated reservations. Owner is a only person who is able to see the system statistics about facility usage such as average occupancy, peak usage hours, parking slot statistics (how much time cars were parked in parking slots), and revenue. Owner also can ask new analysis algorithm to developers in order to see other facility statistics like how many times drivers don’t show-up at the start of their reservation time. In order words, product developer can add the algorithm to the system by owner’s request. Additionally, only owner is able to configure the system to adjust parking fee (an hour basis) and grace-period. System operator is responsible to install and manage parking controllers and handle the exceptional problems like power outage on the parking facility. Parking facility is a parking garage including parking controller. That is, it sends sensor data to the system and the system controls the parking facility to control actuators and LEDs. Finally, CCPS is the system owned by credit card companies. When our system wants to verify driver’s credit card information, the system sends the information to CCPS and receives the result indicating whether the information is valid or not.

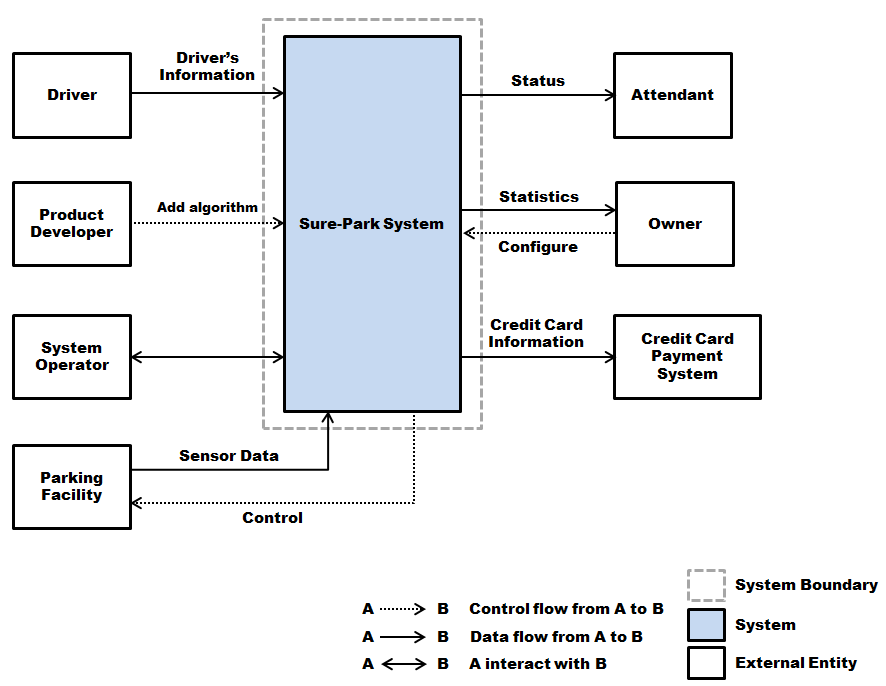


Figure 1 – System context diagram

To summarize, in system boundary perspective, the followings are not included in the system.

* Verify driver’s credit card information
* Sensors, actuators, and LEDs installed on the parking facility
* Develop new analysis algorithm

4. Architecture Design

This section describes the architecture design for Sure-Park system. First, it shows which alternatives have been considered and which design decisions we’ve made. And then architecture designs in static, dynamic, and physical perspective are described based on the decisions.

## **Design Decisions & Alternatives**

This subsection shows how we choose the architecture designs of the system and which patterns and tactics are applied to promote the major architecture drivers.

### #1: Patterns for high-level architecture

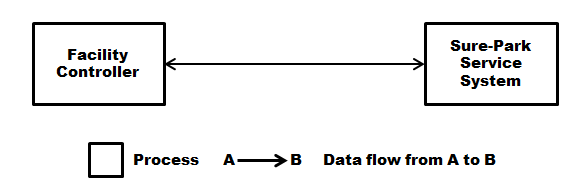


Figure 2 - 1st Decomposition of the system (dynamic perspective)

First, as Figure 2, we decomposed our system into 2 major categories - facility controller and Sure-Park service system. Facility controller is, as its name indicates, a process running on the facility controller HW in order to take care of the events on the HW. Sure-Park service system is to provide Sure-Park services such as parking space reservation, processing the facility statistics, and so forth.

And then we considered scalability which is one of our major quality attributes. According to [2], scalability means that our architecture should support not-predetermined parking facilities and new parking slots or facilities can be installed easily without having to restart the system or changing any existing processing components. To achieve this quality attribute, we chose two architecture patterns as Figure 3 and analyzed them.

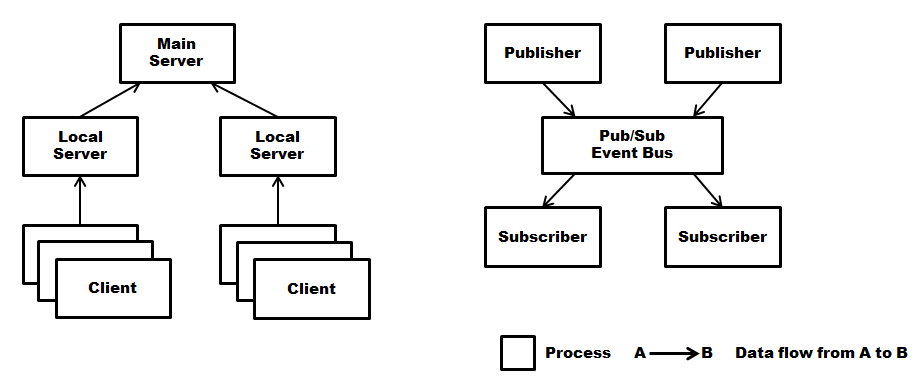


Figure 3 - Hierarchical client-server pattern vs. publish-subscribe pattern   
(dynamic perspective)

At the very first time, hierarchical client-server pattern seemed to be a good one since facility controllers (clients) running on the facility controller HWs in the same parking facility (in physical perspective) are able to have their own local server and clients can be connected to the server in a secured local network in order to satisfy security perspective also. Additionally, only local servers require to be connected to a main server for getting Sure-Park services like parking space reservation. This means that lots of client in a particular facility don’t need to communicate with a single main server – Also satisfying performance perspective (too many clients might cause scalability/performance issue – too many requests from the clients).

However, it turned out that hierarchical client-server pattern even didn’t satisfy QAS for scalability – any changes on existing processing components are not acceptable. Because we should use Arduino HW by technical constraint and it cannot support any configuration files to run same SW with different configurations, facility controller needs to be changed to set-up the information of the local server it will be connected. Also, there might be a lot of local servers and this causes lots of single point of failure problem – so bad for availability.

We therefore found another pattern for our architecture – publish-subscribe. In this case, only thing facility controller needs to know is pub/sub event bus and it can just participate in our system without having to re-start the system or change some existing processing components. The controller can be either publisher or subscriber and can be both if it’s required. This is a very natural way in IoT environment – event-driven, send/receive events to/from the system. However, still pub/sub event bus was a single point of failure and we needed to promote availability by using appropriate tactics.

### #2: Tactics for promoting availability

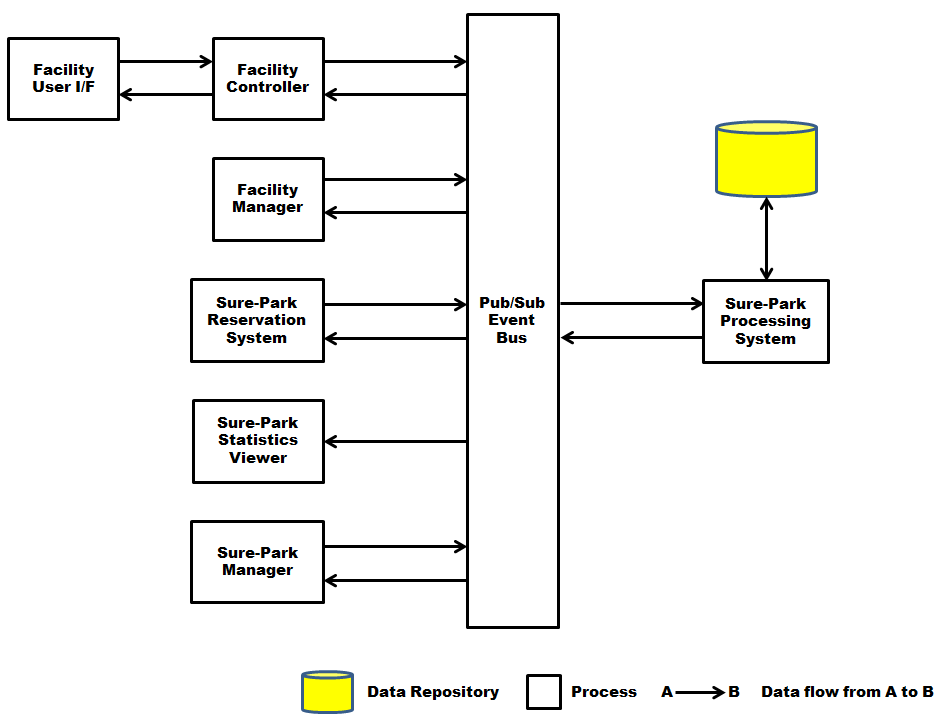


Figure 4 - 2nd Decomposition of the system (dynamic perspective)

We decomposed the system more as Figure 4 based on our 1st decision – use publish-subscribe pattern. We separated Sure-Park service system into 3 categories – server side (pub/sub event bus and Sure-Park processing system), client-side (facility manager, Sure-Park reservation system, Sure-Park statistics viewer, and sure-park manager), and facility-side (facility user interface). Server-side and client-side components usually do their works as subscriber and publisher respectively, but all components can be both publisher and subscribe when they want. Facility user interface is directly communicated with facility controller without having pub/sub connection. You can refer to “Dynamic view of architecture design” to get more details about each entity.

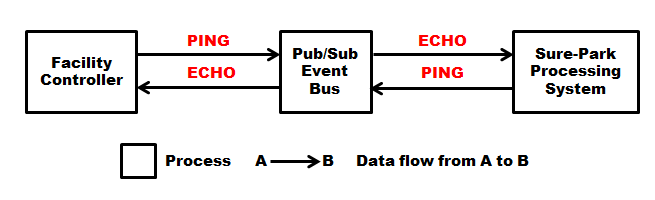


Figure 5 - Tactic #1: Ping/Echo (dynamic perspective)

As our system requires detecting the failure of both facility controller and server-side of Sure-Park service system, we decided that ping/echo tactic can be applied to the system like Figure 5. In publish-subscribe event bus perspective, all components connected to the bus are pub/sub client and the bus itself is pub/sub server. In order words, facility controller and Sure-Park processing system is the client and they send PING packet to the bus in order to indicate they’re alive and the bus sends ECHO back to the clients.

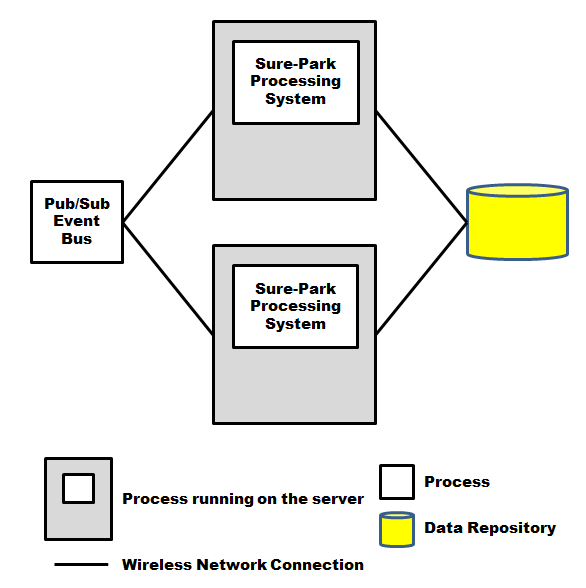


Figure 6 - Tactic #2: Passive Redundancy (physical perspective)

With this feature we could detect when the failure occurred on the clients, however, there was no way to detect the failure on the bus. Also, we needed back-up plan when the components were dead. Therefore, we put the bus and database into the cloud service like amazon firstly. (Actually, we just assume this because this is just a studio project and there is no need to use actual cloud service – it requires a lot of money!!). Commercial cloud service can detect the failure of the bus and database storage and they can be working even if the failure occurs by the redundancy which the cloud service provides. Following section named “Physical view of architecture design” shows more details about this.

Secondly, regarding back-up plan to make service available when some components are dead, we applied another tactic – passive redundancy as Figure 6. We have two servers physically and primary server is running to provide processing services. If the primary server is dead (which can be detected by ping/echo mechanism), secondary server is activated and it can provide services as if the primary server is still alive. You can refer to the following section named “Sequence diagrams” to get more details about how the system provides this redundancy.

### #3: Relief the negative effects of the tactics

Every tactic has its main effect and its side effects – tradeoffs. Because we applied two tactics to promote availability, we also needed to consider their side effects.

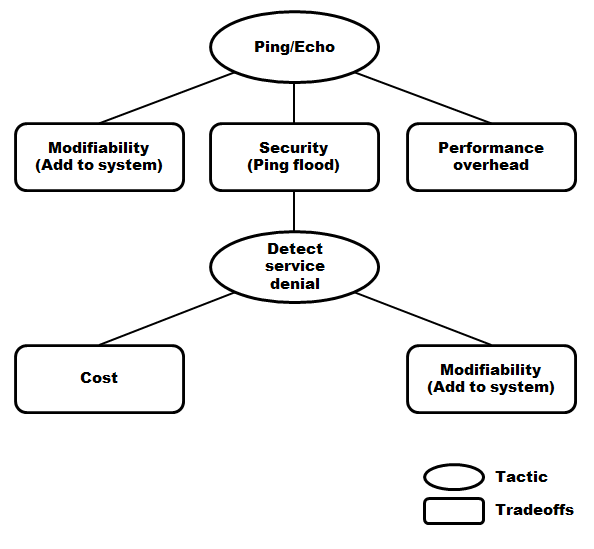


Figure 7 – Compensate the negative effects of ping/echo tactic

We considered 3 tradeoffs such as modifiability (add ping/echo to system), security (ping flood attack), and performance overhead in case of ping/echo tactic. Based on analysis of these tradeoffs, we figured out that modifiability and performance tradeoffs are easily compensated by pub/sub pattern. In case of modifiability, because facility controller and Sure-Park processing system are connected with pub/sub event bus, ping/echo feature can be added as one of pub/sub events. Performance overhead also can be relieved by using piggybacking – embedded ping/echo packet into normal pub/sub event packet. We therefore decided that security tradeoff is the most severe and detect service denial tactic is applied to relief this tradeoff. Detect service denial tactic has also its side effects and we considered two tradeoffs – cost (add front-end HW) and modifiability (add HW to system), however we decided that these tradeoffs are not severe.

In case of passive redundancy, only tradeoff we considered was cost (add more server machine), but we decided that cost is not severe.

### #4: Tactics for promoting security

In spite of applying several patterns and tactics to promote influential quality attributes, security still needed to be considered. In order to promote security, we applied 3 more tactics – authenticate actors, authorize actors, and encrypt data. Authentication and authorization are addressed by “Security Manager” module in static perspective and this module is responsible to ensure that actor who tries to access the system has authenticated identification and the proper right. You can refer to “Static view of architecture design” to get more details. In case of encryption, there are two parts to which we applied it – SSL for communication channel, AES for data encryption via MySQL.

## **Dynamic view of architecture design**

### High-level architecture

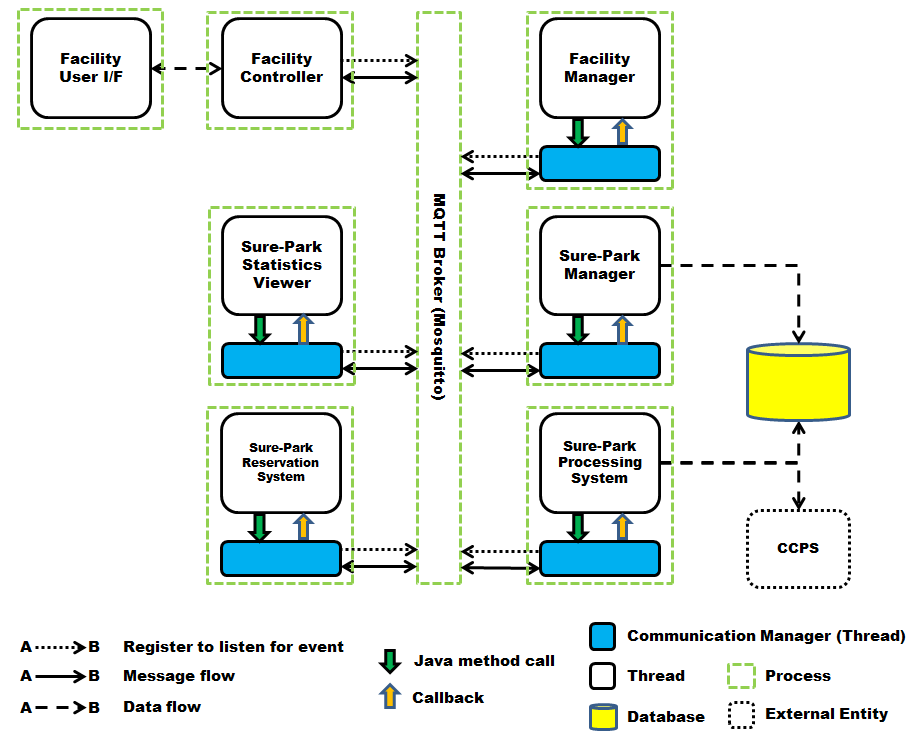


Figure 8 – Dynamic perspective of high-level architecture

Figure 8 shows high-level architecture in dynamic perspective and entities are as the followings.

MQTT Broker: We’re using Mosquitto [5] - open-source publish-subscribe event bus using MQTT protocol. This broker manages all publish/subscribe requests from clients participated in MQTT communication; also is responsible to response client’s PINGREQ to inform that current MQTT connection is still alive.

The broker only communicates with clients through communication manager thread. That is, a client just invokes Java methods provided by communication manager and it actually sends/receives events to/from the broker. Following sections for “Decomposition of communication manager” in both dynamic and static perspective will describe the reason why we’re using communication manager and how communication manager promotes quality attributes such as **performance, security, and modifiability**.

Facility Controller: This thread handles all events from HW (Arduino) and processes requests from Sure-Park Processing System such as blinking LEDs or lifting the entry gate. Also this communicates with Facility User I/F on dedicated communication channel to pass the input from Facility User I/F to Sure-Park Processing System and vice versa

Facility User I/F: This thread initiates verifying confirmation information. When a driver arrive an entry gate of a facility, this thread receives driver’s confirmation information and passes it to Facility Controller to verify it. And then Facility Controller gives whether the confirmation information is valid or not.

Sure-Park Statistics Viewer: This component is for showing statistics to an owner. It communicates with Sure-Park Processing System to get raw data and processes them to create and show statistics. Pipe-and-filter pattern is applied to this component to promote **extensibility** (see the following section “Decomposition of Statistics for extensibility”)

Sure-Park Manager: This component is for managing parking facilities. If new parking facility or parking controller is added to the system, an operator can see which facility or controller is added and configure it with proper attributes.

Facility Manager: This component is for showing facility status to an attendant and also receives events from Sure-Park Processing System when related parking facility is out of order.

Sure-Park Reservation System: This component is responsible to reserve a parking slot for a driver. It receives driver identification information, date and time, and credit card system from a driver and sends them to Sure-Park Processing System. It also shows confirmation information to a driver when reservation is done successfully.

Sure-Park Processing System: This thread processes reservation request from Sure-Park Reservation System and send confirmation information back to the system. It also communicates with CCPS and database to verify credit card information and send queries respectively. Besides, it sends control events to Facility Controller to lift an entry/exit gate or blink LEDs.

Communication Manager: All clients are only able to publish and subscribe events through communication manager thread. This thread provides 4 communication channels on each connection between a client and MQTT broker. A client can select which channel is used based on the intent of the communication and all channels can be used simultaneously if it’s required. See “Decomposition of Communication Manager” below to get more details about the channels.

### Decomposition of Communication Manager

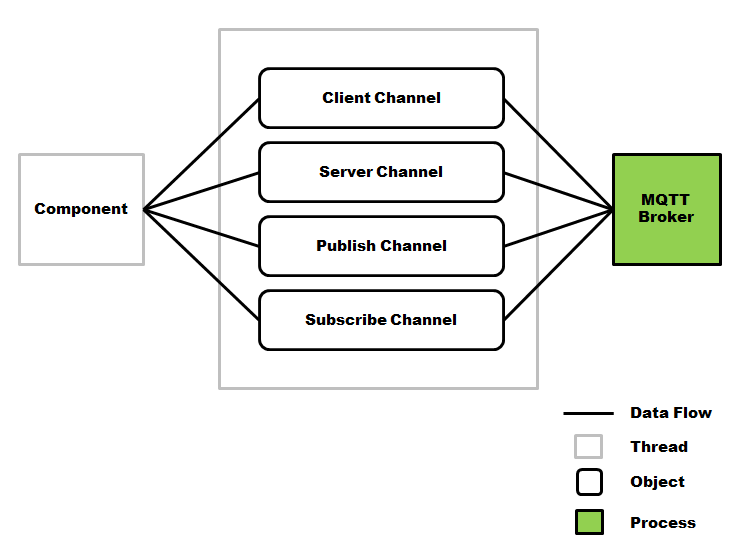


Figure 9 – Structure of Communication Manager (dynamic perspective, partial view)

Figure 9 shows partial structure of communication manager in dynamic perspective.

Basically, since we applied to publish/subscribe pattern to the system as described above, we have two channels for pub/sub communication – publish channel and subscribe channel. Using these channels, component can publish events to un-predetermined target components and subscribe any events which are interested – “1:N” communication. However, if many components publish/subscribe a lot of things or some components just subscribe everything, performance degradation might happen on the network and some security vulnerability can be exposed – some components are able to get un-permitted data from the broker by just subscribing the event.

We created two more channels – client channel and server channel – in order to compensate the weakness of publish/subscribe pattern. If a component uses these channels, it is only able to communicate with one target component – “1:1” communication. In order words, if component A publishes an event to B on client channel, only component B can receive the event and any other components cannot subscribe this event at all – promoting **security**. Also **performance** can be promoted since packets on these channels won’t go to unintended components even if they subscribe everything.

You can refer to [8] to get more details about this protocol.

### Sequence diagrams

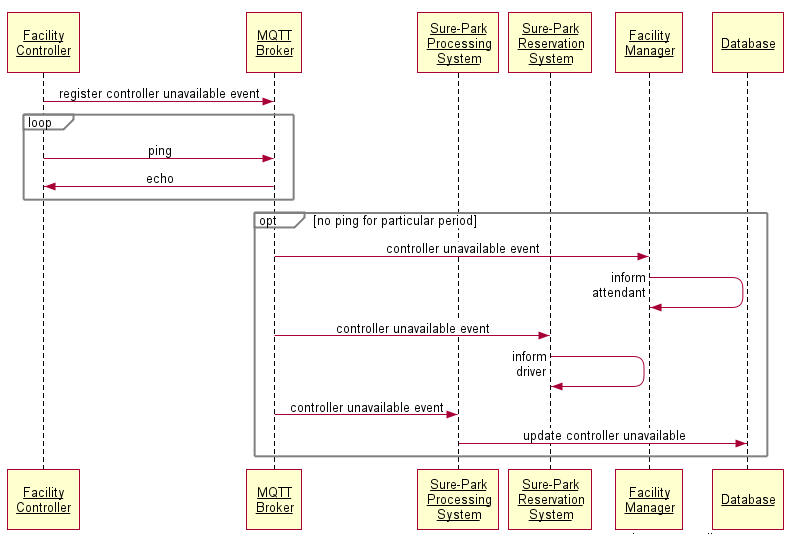


Figure 10 – Sequence diagram of “failure on the facility controller”

Figure 10 shows how the system reacts when facility controller is out of order. (MQTT broker is a process and others are threads.)

Term:

* WILL message: dying message on our communication protocol; facility controller and Sure-Park processing system register their own dying messages when they are being connected to MQTT broker

Preconditions:

* Facility manager subscribed WILL messages of facility controllers which were interesting.
* Sure-Park reservation system subscribed WILL messages of facility controllers on which reservations were made.
* Sure-Park processing system subscribed WILL messages of whole facility controllers on the system.

Steps:

* Facility controller registers its WILL message to MQTT broker. In order words, registered message will be broadcasted by MQTT broker when the broker detects that the controller is not available anymore.
* Facility controller and MQTT broker repeatedly sends ping and echo message respectively to inform that they are still alive.
* MQTT broker doesn’t get ping message from the facility controller for particular period. When this happens, it broadcasts registered WILL message of the controller.
* Sure-Park reservation system and facility manager just notify the failure on the controller to a driver and an attendant respectively when they receives the message. Also, Sure-Park processing system updates database to disable the controller until it’s fixed.

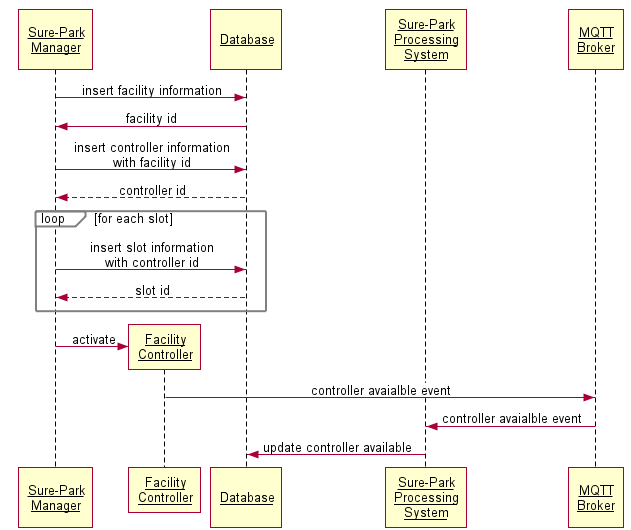


Figure 11 – Sequence diagram of “adding a new facility controller”

Figure 11 shows how the system reacts when a new facility is added on run-time. (MQTT broker is a process and others are threads.)

Preconditions:

* Sure-Park processing system subscribed an event indicating “new facility controller is added”.

Steps:

* Sure-Park manager sends a message indicating which parking facility newly added controller needs to be added to.
* Database processes the request and responses to the Sure-Park manager with facility ID.
* Sure-Park manager is now able to know the ID of the parking facility. The manager sends a message with the controller’s information. In order words, this message is for adding a new facility controller to the parking facility the facility ID indicates.
* Database processes this request and sends controller ID back to the manager.
* At this time, unique controller ID is assigned to the facility controller which is being added. Sure-Park manager sends slot information to database until all slots on the controller are added completely.
* Now Sure-Park manager can activate the new facility controller. The controller sends an event – “Now I’m available.”
* Sure-Park processing system receives this event and update database to enable this controller which means a driver can reserve a parking slot on this controller from now.

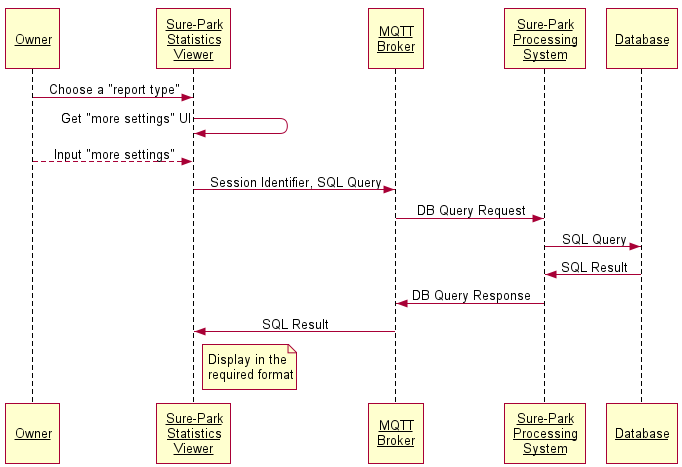


Figure 12 – Sequence Diagram of “Adding new analysis algorithm”

Figure 12 shows how the system reacts when new analysis algorithm is added. (Owner is an actor, MQTT broker is a process and others are threads.)

This sequence is very simple – just sends SQL query to Sure-Park processing system and get required data to process a new analysis algorithm.

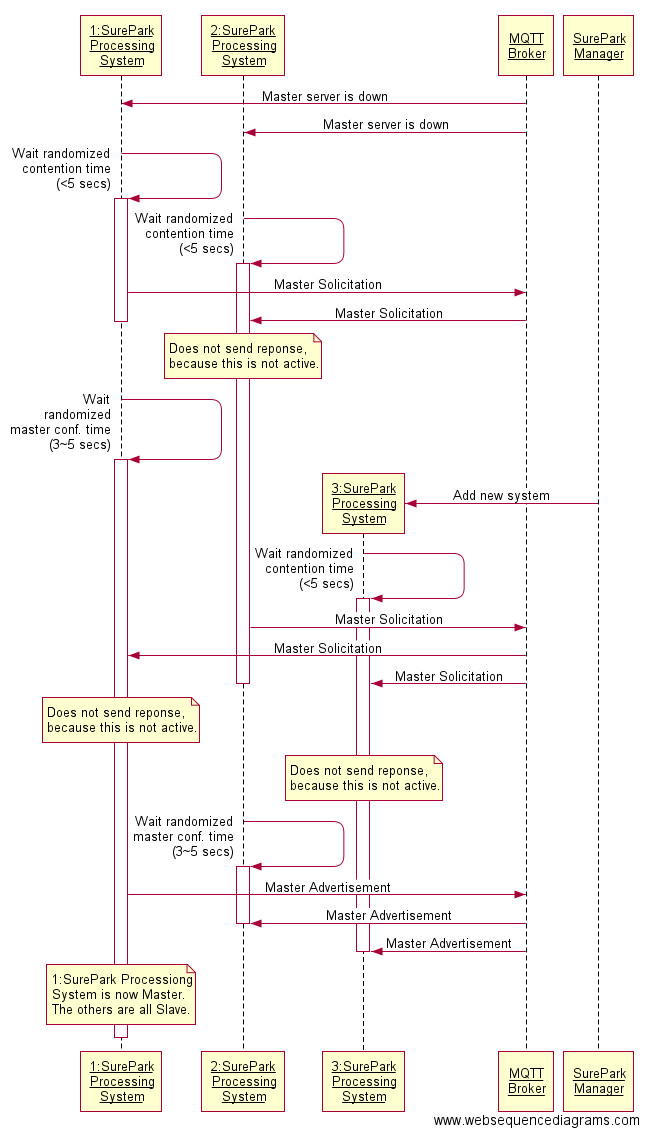


Figure 13 – Sequence diagram of “how passive redundancy works when the primary server is out of order”

Figure 13 describes how passive redundancy works when the primary server is out of service. (MQTT broker is a process and others are threads.)

Preconditions:

* Sure-Park processing system registered its WILL message when it was being connected to the broker.
* Sure-Park processing system subscribed WILL, master solicitation, and master advertisement messages of other processing systems.
* Each Sure-Park processing system is running on physically separated server machines.

Steps:

* MQTT broker detects the failure on the primary server and WILL message of the server is broadcasted.
* Each Sure-Park processing system has its own contention-avoid time which was assigned randomly within 5 seconds when it was up.
* When Sure-Park processing system receives WILL message, it waits until the time elapses. And then, it sends out a message – master solicitation – which is for figuring out “Does primary server exist?”
* For now, there is no primary server, so no one can response this message.
* Each Sure-Park processing system has another timer for “how much time I need to wait for master advertisement message?” This time was also configured randomly within 3 ~ 5 seconds and each processing system wait until this time elapses.
* After that, it sends out master advertisement message to insist “I’m primary server”.
* Anyone who receives this message immediately stops figuring out who is the primary server.

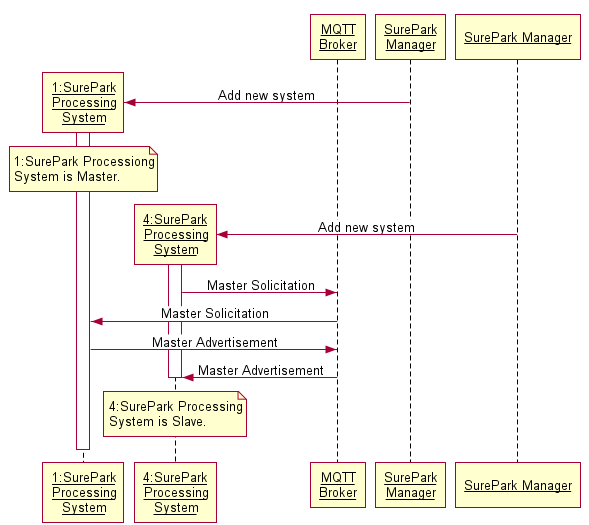


Figure 14 – Sequence diagram of “how the system reacts when new server is added”

Figure 14 is very similar to the sequence of Figure 13. (MQTT broker is a process and others are threads.)

When a new server is added, Sure-Park processing system running on the server sends out master solicitation message to figure out the existence of the primary server on the system. System #1 is a primary server in this case, so it responses the solicitation message through master advertisement message. When system #4 gets the advertisement message, it is able to know the existence of the primary server and all previous jobs are canceled.

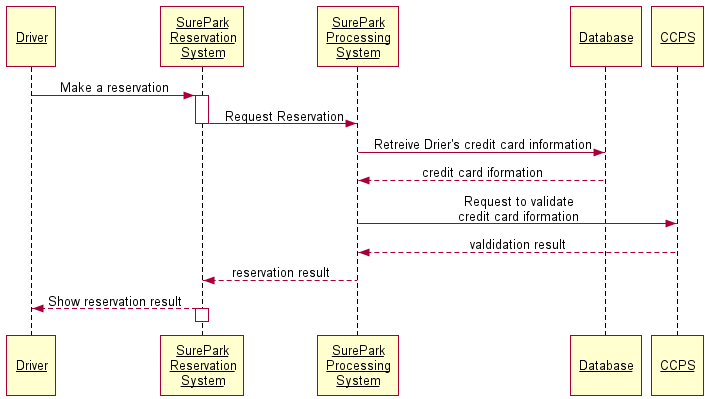


Figure 15 – Sequence diagram of “making reservation”

This figure describes the sequence when a driver makes reservation on particular parking slot. (Driver is an actor, others are threads.)

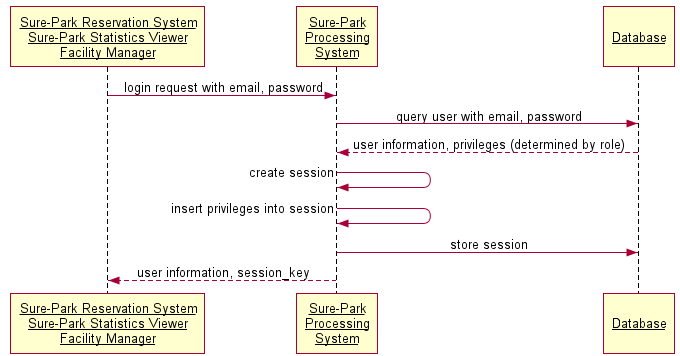


Figure 16 – Sequence diagram of “login process”

Figure 16 shows the sequence of login process. (All entities are threads.)

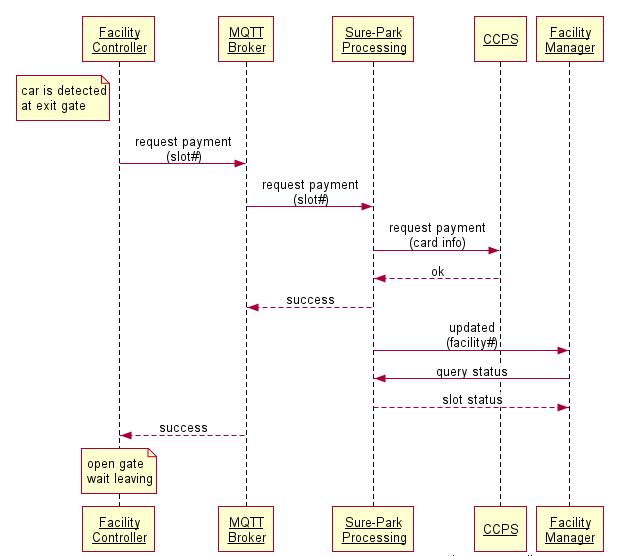


Figure 17 – Sequence diagram of “payment process”

This figure shows how parking fee is processed when a driver is leaving a parking facility.

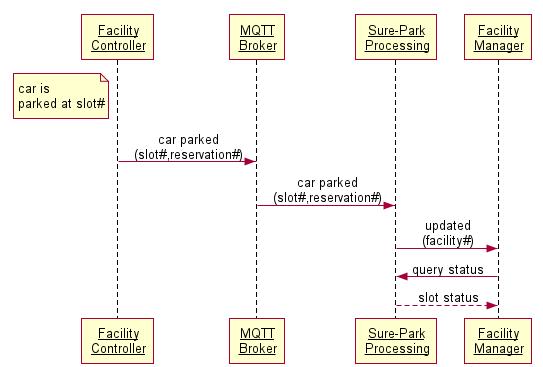


Figure 18 – Sequence diagram of “parking a car”

Figure 18 shows how the system reacts when facility controller detects a car newly parked. (MQTT broker is a process and others are threads.)

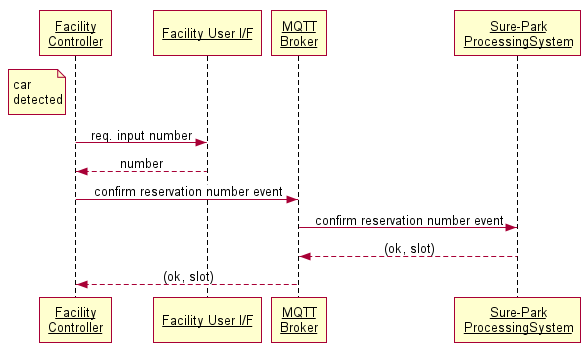


Figure 19 – Sequence diagram of “passing an entry gate”

This figure describes how the system reacts when a car is passing an entry gate. (MQTT broker is a process and others are threads.)

## **Static view of architecture design**

### High-level architecture

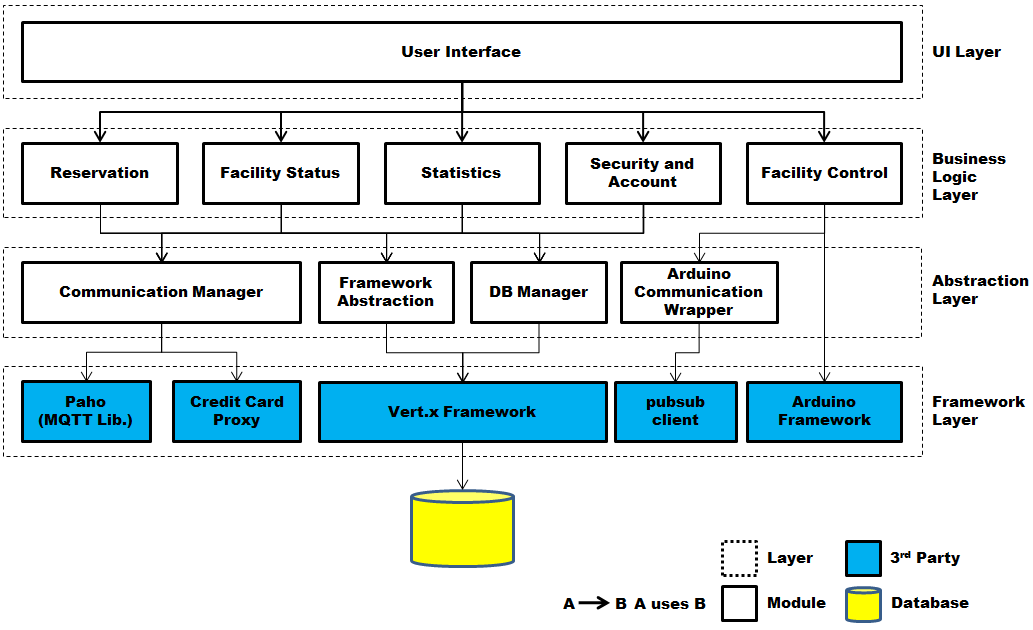


Figure 20 - Static perspective of high-level architecture

Figure 20 shows the static perspective of architecture design. We applied layered architecture pattern to satisfy one big constraint about “Schedule and Resource” – we don’t have much time (only 6 weeks) and have just 6 persons. Our software needs to be segmented that all modules can be developed and evolved separately with little interaction among the parts – separation of concerns. To achieve this, we separated the system into 4 layers such as UI Layer, Business Logic Layer, Abstraction Layer, and Framework Layer. We don’t need to change Business Logic Layer even if some technologies on Framework Layer don’t fit for us because Abstraction Layer abstracts underlying framework entirely; promoting **modifiability and portability**. Additionally, Abstraction Layer can provide common features to the modules in Business Logic Layer, so we promoted **re-usability** within the project scope for satisfying the schedule.

Each entity is defined as the followings.

**UI Layer**

User Interface: This module is responsible to handle user inputs and show the outputs of the system to a user. Inputs are:

* driver’s identification
* date and time when the system reserves a parking slot for a driver
* credit card information used for charging parking fees to a driver
* confirmation information to verify that a driver has valid reservation when a driver arrives a facility
* parking fee (an hour basis) and grace-period which can be configurable by an owner

Outputs are:

* statuses such as which parking slots are opened and which are occupied, how long a car has occupied a particular slot, and whether a driver parks in the wrong parking space and automatically re-assign parking spaces and correlate associated reservation
* statistics such as average occupancy, peak usage hours, parking slot statistics, revenue, and customized values
* confirmation information when reservation is done successfully
* notifications when exceptional issues occur like power outage on server

This module sends inputs to underlying Business Logic Layer to process them and receives data from the Business Logic Layer to show them to users like drivers, attendants, developers, operators, and owners.

**Business Logic Layer**

Reservation: One of the main features of the system is to reserve a parking slot and this module is in charge of the feature. Making reservation includes checking the availability of parking spaces in a garage, getting and processing the information from a user, and providing confirmation information to the user when reservation is done.

Facility Status: This module is responsible to provide features of the functional requirements for “Attendant” – FR-031, FR-032, FR-033, and FR-034. That is, the system is able to use this module for checking which parking slots are available and which are not, figuring out how long a car has occupied a particular slot, and re-assigning the parking spaces and correlate associated reservations when a driver parks in the wrong slot.

Statistics: There are functional requirements for “Owner” – FR-041, FR-042, FR-043, and FR-044, and this module is responsible to provide functions for these requirements. The module is designed to promote **configurability** – easy to change the parking fee or the grace-period, and **extensibility** – easy to add new analysis algorithm. Following section describes this more.

Security and Account: This module is designed to promote **security** – only authenticated and authorized user can access the system. The system satisfies the security in many ways such as authentication, authorization, and encryption; and this module is for authentication and authorization by allowing a user who has permitted ID and proper authorization to access the requested information. Security patterns (described in [3]) such as single access point, check point, roles, session, and limited view are applied to promote security.

Facility Control: This module is responsible to control a facility by processing data from sensors and controlling actuator and LEDs. By technical constraints – use dedicated HW (Arduino) and it won’t be changed, this module uses Arduino Framework directly and no abstraction module is considered.

**Abstraction Layer**

Communication Manager: This module provides common communication channels to send and receive events through pub/sub event bus. We created 4 channels on each connection between pub/sub event bus and a client to promote **performance and security**. Also delivery of messages can be guaranteed by this scheme to compensate the weakness of pub/sub pattern - no guarantees whether messages are delivered to a target. Besides, it abstracts underlying communication protocol to promote **modifiability**. Following section “Decomposition of Communication Manager” will describe this more.

Framework Abstraction: This module is responsible to provide an abstraction for underlying framework. Now we’re using Vert.x as our framework, but we can easily change the underlying framework without having modifications on Business Logic Layer.

DB Manager: This module is to abstract underlying database access scheme. We’re using MySQL library provided by Vert.x framework for now, but Business Logic Layer doesn’t know about this and just uses DB Manager to access the database.

**Framework Layer**

Paho: Paho [7] is an open-source client implementation of MQTT and MQTT-SN messaging protocols aimed at new, existing, and emerging applications for IoT.

Credit Card Proxy: Driver’s credit card information should be sent to CCPS to be verified and it requires secured communication channel between the system and CCPS. This module is responsible to provide secured connection and we assume that credit card company provides this module.

Vert.x Framework: Vert.x is an open-source, event-driven, and non-blocking framework and it’s easy to handle a lot of concurrency using a small number of kernel threads. We chose this framework to promote **scalability** of the system and Sure-Park Processing System is designed to use this in dynamic perspective. To get more details about how Vert.x serves scalability of the system, you can refer to [4].

pubsubclient: pubsubclient [6] is a client library for the Arduino Ethernet shield that provides support for MQTT.

Arduino Framework: Parking facility uses Arduino as its HW platform by technical constraint. So, Facility Control uses Arduino Framework to control Arduino and receive events from it.

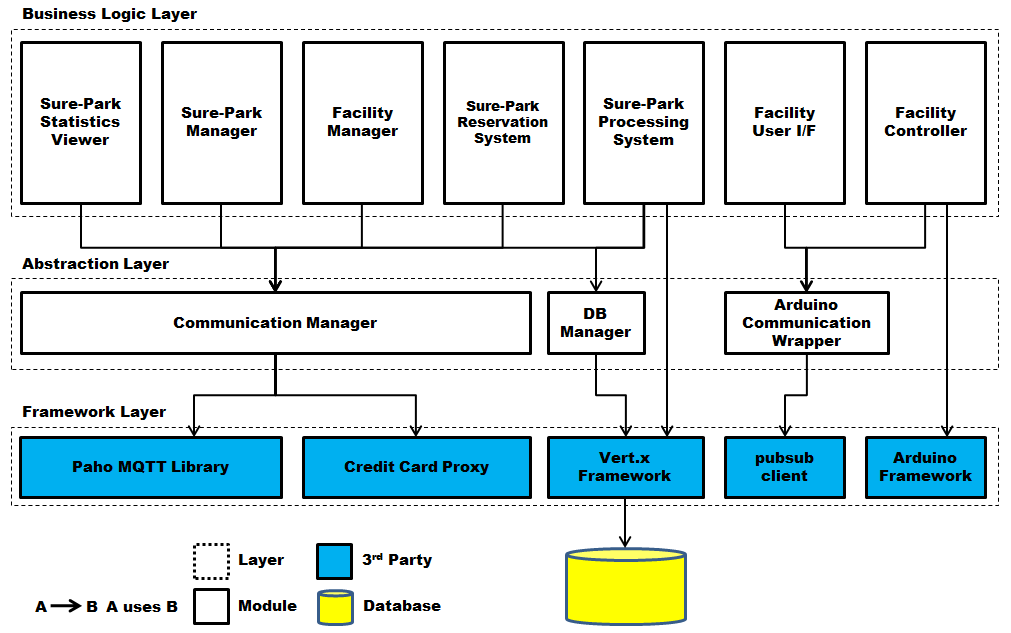


Figure 21 – Static perspective of the architecture for demo

In case of demo, we couldn’t implement the architecture described above completely since we didn’t have much time to do that and some features were not required for demo. Therefore, we simplified our architecture as Figure 21. First, we merged UI layer into Business Logic Layer because we didn’t need to focus on UI for demo – Business Logic Layer just can handle simple UI. And then, we split Business Logic Layer into 7 categories based on the features we need to provide. Also, Framework Abstraction module has been removed and Sure-Park Processing System just directly uses underlying framework – Vert.x (we don’t have some time for implementing the abstraction layer of Vert.x).

### Decomposition of Communication Manager

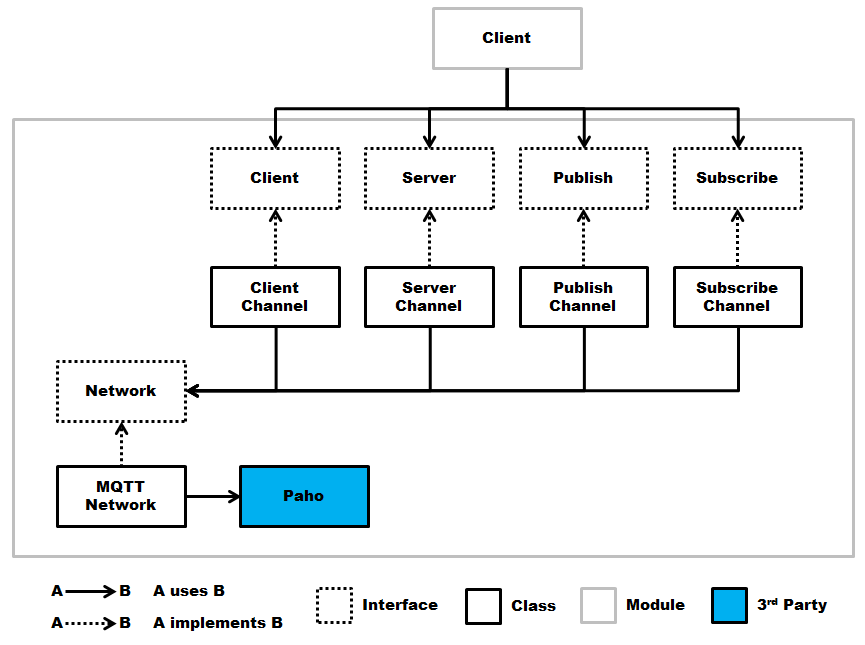


Figure 22 – Structure of Communication Manager (static perspective, partial view)

Figure 22 shows the partial structure of Communication Manager in static perspective. As described above, we have 4 channels encapsulating the connection between MQTT broker and a client. For these channels, 4 interfaces are defined such as Client, Server, Publish, and Subscribe; and each channel class implements corresponding interface. Client of Communication Manager just uses the interfaces; therefore the client doesn’t need to be modified even if actual implementations of channel classes are changed. Also, Communication Manager hides its underlying communication protocol through Network interface. All channel classes use the Network interface and a class for real protocol (MQTT for our case) implements the interface.

To summarize, there are two concepts to promote **modifiability** of this module.

* Interfaces for 4 channels: client of this module doesn’t need to be modified even if actual implementations are changed.
* Network interface: hiding underlying communication protocol; protocol can be easily changed without having severe impacts on the structure of this module.

You can refer to “Decomposition of Communication Manager” in dynamic perspective.

## **Physical view of architecture design**

### High-level architecture

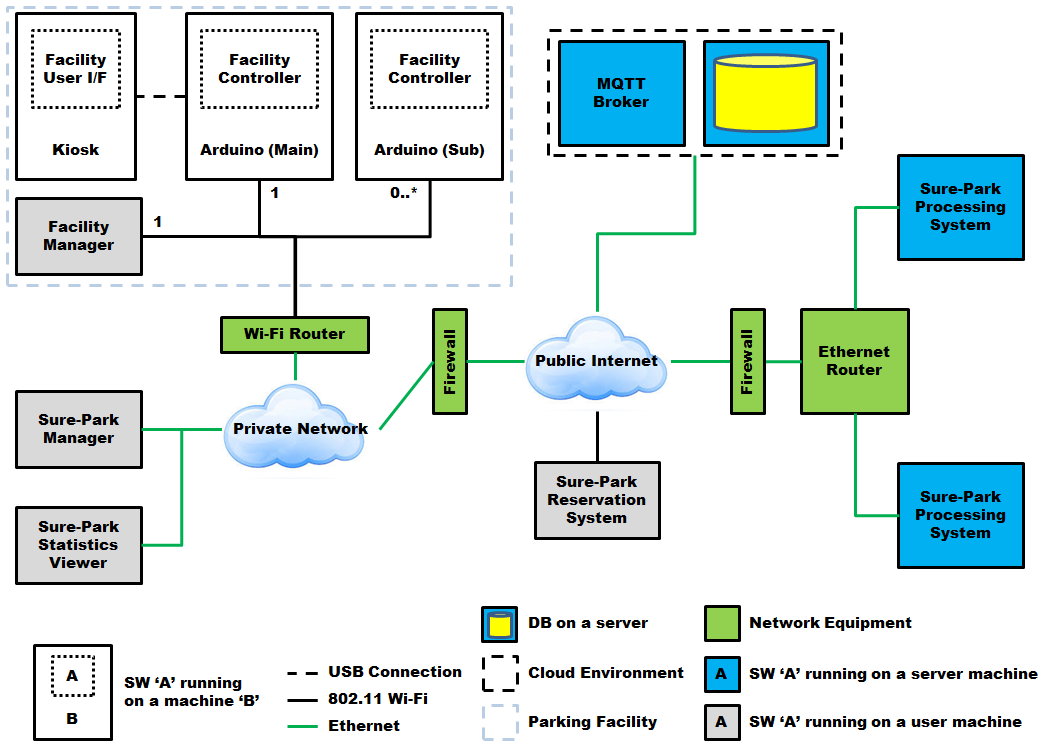


Figure 23 – Physical perspective of high-level architecture   
(mapping between dynamic and physical)

Figure 23 shows the physical view of high-level architecture. Each entity is as the followings.

Kiosk: Kiosk is a machine on a parking facility and is connected to Arduino (Main) through USB connection.

Arduino (Main, Sub): Each parking facility has one Arduino (Main) and zero, one, or more Arduino (Sub). Arduino (Main) is a primary machine and it is only one which is connected to Kiosk. Count of Arduino (Sub) depends on how many parking slots exist in the facility – it can be zero, one, or more as the occasion demands. All these Arduinos are connected to private network through 802.11 Wi-Fi.

Facility Manager: Each parking facility has one machine for an attendant who manages the facility. This machine is connected to private network through 802.11 Wi-Fi.

Sure-Park Manager: There are machines for operators. These machines are connected to private network via Ethernet connection.

Sure-Park Statistics Viewer: There are machines for owners. These machines are connected to private network via Ethernet connection.

Private Network and Firewall: In order to promote **security**, the system uses private network for parking facility and machines for owners and operators. Firewall is a single access point to prevent attackers from accessing the system without authority.

MQTT Broker and DB: Machines for MQTT broker and database are placed on the commercial cloud environment like Amazon to promote **availability**.

Sure-Park Reservation System: There are machines for drivers. There machines are connected to public Internet via 802.11 Wi-Fi.

Sure-Park Processing System and Firewall: There are two machines and each machine runs Sure-Park processing system thread independently. Normally, one of the machines is a primary one and other machine is deactivated until primary machine goes wrong (out of service). When the failure happens, second machine is activated and thread running on the machine takes care of the jobs which the primary machine processed previously. Additionally, all machines are connected to public Internet through Firewall to promote **security**.

In case of demo, we couldn’t realize all these configurations, so following assumptions were made.

* Only Arduino (Main) exists.
* No firewall
* No public network (we can only use private network in a classroom)
* No commercial cloud environment (too expensive to use this for demo)
* No Ethernet connection (we only have 802.11 Wi-Fi connection)