Protocol Design

TEAM 2 – NOT YET

Revision History

Version	Date	Description	Author
0.1	2016.06.10	Initial version	Byounghoon (Beney) Kim
0.2	2016.06.15	Update rationales	Byounghoon (Beney) Kim
0.3	2016.06.17	Update channel category	Byounghoon (Beney) Kim
0.4	2016.06.20	Add server redundancy section	Byounghoon (Beney) Kim
0.5	2016.06.22	Update document purpose	Byounghoon (Beney) Kim

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

References

[1] [TEAM2_REFERENCE_04] Business Channels
[2] [TEAM2_DOC_03] Architecture Document
[3] [TEAM2_REFERENCE_02] MQTT_v3.1.1

1. INTRODUCTION

1.1 SCOPE OF DOCUMENT

This document describes Protocol of SurePark system which is designed by Not Yet Team. In SurePark system, there are physically distributed elements such as Arduino controller, business server, applications for end users including drivers, attendants and owners. At chapter 2, this document describe a way how an element to find other element, message structure is used to exchange data between elements. Also, chapter 2 introduces a channel which is abstracted communication way used in the system. At chapter 3, this document describes all channel's detail information.

1.2 PURPOSE OF DOCUMENT

The purpose of this document are as below.

- describing protocol specification used in SurePark system
- explaining decision and rationale while designing protocol.
- describing how this protocol work
- describing how this protocol hides the complex network communication behavior from applications.
- describing how an application to use the this protocol in implementation level for developers.
- explaining a comprehensive detailed description for all channels.

2. DECISIONS AND RATIONALE

This chapter describes what decision are made while protocol design and related rationale. Section 2.1 describes network/service discovery and message format. Section 2.2 shows abstraction level of channels. Section 2.3 explains how channels are categorized in SurePark system.

2.1 NETWORK DISCOVERY AND MESSAGE FORMAT

The distributed environment leads that each elements should network communication to send and receive message. In a view of network communication, at least, two methods should be defined: 1) a way how to describe and find other elements in network and 2) message's data representation which sender and receiver can understand.

2.1.1 Network element discovery

The simplest way to describe each element is using IP address itself. To use this way, each element should know the relationship between each element's IP address and its responsibility. Another way is using DNS name which hides actual IP address behind domain name. In SurePark system, publisher/subscriber pattern is used. Hence, each element needs not know each other, even existence of each element. Hence the only thing which all know should share is publisher/subscriber broker. In SurePark system, broker's IP address is known by other elements. But it can be changed to use DNS easily.

2.1.2 Message representation

Considering message which is used in network communication, there are two kinds of data. The one is used to decide the purpose of message (in other words, message type), for example, log-in, making reservation and notifying some state change in Arduino controller. The other one is contents for each purpose, for example, the driver application should send reservation date and parking lot location to business server to make reservation. Usually, for example, standard TCP/IP protocol use well-defined but fixed data structure for both. Also, there is TLV (Type, Length, Value) based message. The message is set of TLVs. It can improve extensibility in protocol layer, because adding one TLV is not a big problem and it has no effect on other content. One of several TLV in the message is used for message type. There is one more option we've considered, Json. Because developers need not know how message is structured in byte order with Json, it leads high portability, extensibility and very easy to use. But, it needs more bytes compared to above two schemes. Also Json library should be available in Arduino environment, because it has no enough time to implement it. Finally, after experimental, we decided that SurePark system use Json, because Json is better than others in almost characteristics and 'easy to use' is the most important to use

because we have only 5 weeks for this project. . Table 1 shows the comparison of candidates on message structures reviews by NOT YET team.

Characteristics	Fixed format message	TLV based message	Json
Extensibility (Add new message type)	Low	High	High
Efficiency (Message length)	High	Mid	Mid
Portability	Low	Low	High
Easy to use (Implementation level)	Low	Low	High

Table 1. Comparison of candidates on network communication message structure

2.1.3 Service discovery

In previous sections 2.1 and 2.2, this document described how to find destination - the destination IP address of broker - and message structure. This section describes how service discovery is done in SurePark system. Because, in publisher/subscriber pattern, it could inhibit performance if all message is reached to all elements. Hence, SurePark system uses channels in term of that messages are explicitly sent to known specific elements that requested to listen to the channel. Each channel has its own purpose, in other words, service capability, i.e. making reservation, notifying parking facility status, so on. That is, to use some specific service, it needs to select just proper channel. Moreover, this leads that message type is separated from message, because channel itself stands for own purpose - message type -, that is, there is no message type in message.

Moreover, to make easier representation of channels, SurePark system uses REST (REpresentational State Transfer)-like representation, i.e. /facility/3/reservation, means reservation request in facility id 3. Finally, Table 2 shows an example of channel and message body which is exchanged in the channel.

Channel description	/facility/{facility_id}/reservation
Message body (Json object)	{ 'session_key' : 'session_value' // string, driver's session key 'reservation_ts' : reservation epoch / 1000 // integer }

Table 2. Reservation Channel's Request Format

2.2 ABSTRACTION LEVEL OF CHANNEL

Let's assume that the arriving car is allowed to pass the gate entry and assigns slot 3 by SurePark system. Now, parking facility, it needs to change entry gate LED to green, open the entry gate and blinking slot 3's LED. For this purpose, we examined two schemes about which level of message server send to parking facility. The first one is using low level 3 messages - one for entry gate LED, one for entry gate and the other for parking slot LED. The other is high level one message which means that the reservation is confirmed, then the parking facility, Arduino, does above behavior. If we use the first approach, server has all responsibility to control parking facility. Hence, it's easy to add more functionality without changing parking facility, server can control it. But, server has to know the details of Arduino, it inhibits portability and modifiability if it needs to introduce new kinds of microcontroller. It's too strong dependency. If we use the seconds approach, it has better performance than the first one, because the number of exchanging messages are obviously small. Also, it removes the dependency between microcontroller and the rest of system. Table 3 shows the comparison of candidates on message structures reviews by NOT YET team. According to customer, there is no scenario about extensibility and portability, we've focused on performance and modifiability and selected the second approach. Note that the number of message is important factor, because SurePark system is based on publisher/subscriber.

Characteristics	**Low level message**	**High level message**
Performance (the number of exchanging message)	Low	High
Extensibility (add new functionality without change facility)	High	Low
Portability (Add new kinds of microcontroller)	Low	High
Modifiability (Dependency between microcontroller and the others)	Low	High

Table 3. Comparison of abstraction level of channel

2.3 CHANNEL CATEGORY

The SurePark system uses two kinds of communication - notification and request/response. Notification is that one element notifies a message on a channel, some subscribers receive the message. Request/response is that one element notifies request something and one among subscribers respond to the requested message. This is used in case that a requester needs a response, i.e. making reservation needs the reservation result.

2.3.1 Channel Class Hierarchy

Figure 1 shows class diagram of channels. A *NetworkChannel* has *INetworkConnection* which is abstraction of Network Connection. It could be IP network, publish/subscribe over IP network, any kinds of network. The *NetworkChannel* has *Uri (Unified Resource Identifier)* which represents channel itself in *INetworkConnection*. It could be IP address in INET network or file name in local socket. Because SurePark system is based on publish/subscribe pattern, it uses REST-like string as described at section 2.1.3. Also, the *NetworkChannel* uses *NetworkMessage* to send/receive it through *INetworkConnection*.

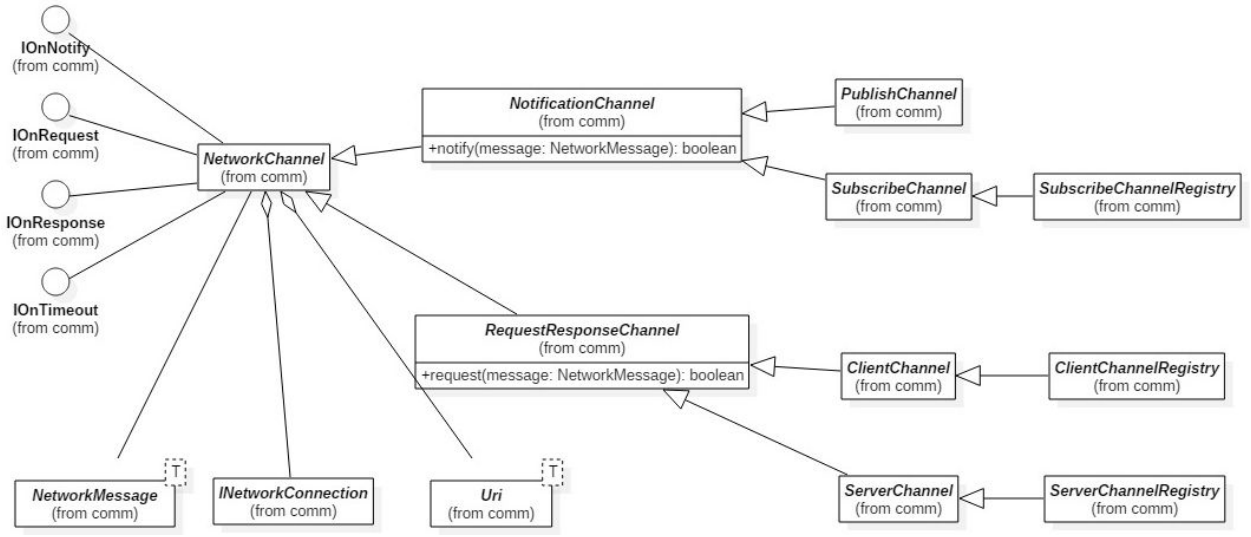

Figure 1. Comparison of abstraction level of channel (Static view, UML notation)

As mentioned in the beginning in this section, a *NetworkChannel* is categorized two types - *INotificationChannel* and *RequestResponseChannel*. *INotificationChannel i*s used for event passing pattern communication. With extending *INotificationChannel*, it's possible to send and receive notification message. Also, *PuslishChannel* is a kind of *INotificationChannel,* which it is allowed to send message only. A *SubscribeChannel*, it is allowed to receive message only. On the other hand, a *RequestResponseChannel* is used for call return pattern communication. Similarly, *ClientChannel* is allowed to request a service only. A *ServerChannel* is used to respond to the request. Registries such as *SubscribeChannelRegistry, ServerChannelRegistry, ClientChannelRegistry* provides callback registration which is called when there is received notification message, request message, response message.

2.3.2 Examples

For stream designer and implementation, this section show two examples. Figure 2 is an example, how to use *SubscribeChannel*. It's very simple. It needs just 2 steps. 1) Because SubscribeChannel is used to receive notification message only, it needs to implement *onNotify()* method. 2) Then, it needs to implement *getChannelDescription()* which is REST-like characters. Whenever, there is notified messages

from other elements on described channel, *onNotify()* is called with NetworkMessage object. It has Json object as explained at section 2.1.2.

```
private final class WillSubscribeChannel extends SubscribeChannel {

    WillSubscribeChannel(INetworkConnection networkConnection) { super(networkConnection); }

    @Override
    public Uri getChannelDescription() {
        return new MqttUri(getSelfConfigurationUri().getLocation() + MqttConstants.WILL_MESSAGE_TOPIC);
    }

    @Override
    public void onNotify(NetworkChannel networkChannel, Uri uri, NetworkMessage message) {
        Log.logd("WillSubscribeChannel", "onNotified:" + message.getMessage() + " on channel=" + getChannelDescription());
        if (!mIsMaster) doSelfConfiguration();
    }
}
```

Figure 2. SubscribeChannel usage example

Here is *ClientChannlRegistry* example. Similarly, it needs to implement *getChannelDescription()* as Figure 3. Then, Figure 4 shows how to add callback object. Then, if there is response from service provider, callback object *mUpdateFacilityListResult* is called. Otherwise, in case of no response, *mUpdateFacilityListTimeout* is called.

```
public class ReservableFacilitiesRequestChannel extends ClientChannelRegistry {
    private final static String TOPIC = "/facility/reservable_list";
    private final static String KEY_SESSION_KEY = "session_key";

    public ReservableFacilitiesRequestChannel(INetworkConnection networkConnection) { super(networkConnection); }

    @Override
    public Uri getChannelDescription() { return new MqttUri(TOPIC); }
}
```

Figure 3. ClientChannelRegistry usage example

```
ReservableFacilitiesRequestChannel reservableFacilitiesRequestChannel = ncm.createReservableFacilitiesRequestChannel();
reservableFacilitiesRequestChannel.addObserver(mUpdateFacilityListResult);
reservableFacilitiesRequestChannel.addTimeoutObserver(mUpdateFacilityListTimeout);
```

Figure 4. Comparison of abstraction level of channel (Static view, UML notation)

2. 4. SERVER REDUNDANCY

To improve availability, SurePark system uses business server redundancy. In publish/subscribe pattern, the message can be delivered to both or more servers if they are listening the message. This is a key point how to implement business server redundancy.

We've evaluated both active and passive redundancy tactics. However, with active redundancy, all servers receives same message. In SurePark system, because business servers share data repository (databases), this leads a complex problem in terms of transaction problem, how to handle the request if the same request are already processed with other server. Because of business constraint, time is not enough to implement it. Also, same response in the network could inhibit performance. On the other hand, passive redundancy, it needs how to select the active server which is response to respond to the requests among all server at the moment. For this, please refer to [2] [TEAM2_DOC_03] Architecture Document. Also, when a active server is down, it needs 5~10 seconds to decide a active server. About this time delay, we discussed with customer and this latency is acceptable.

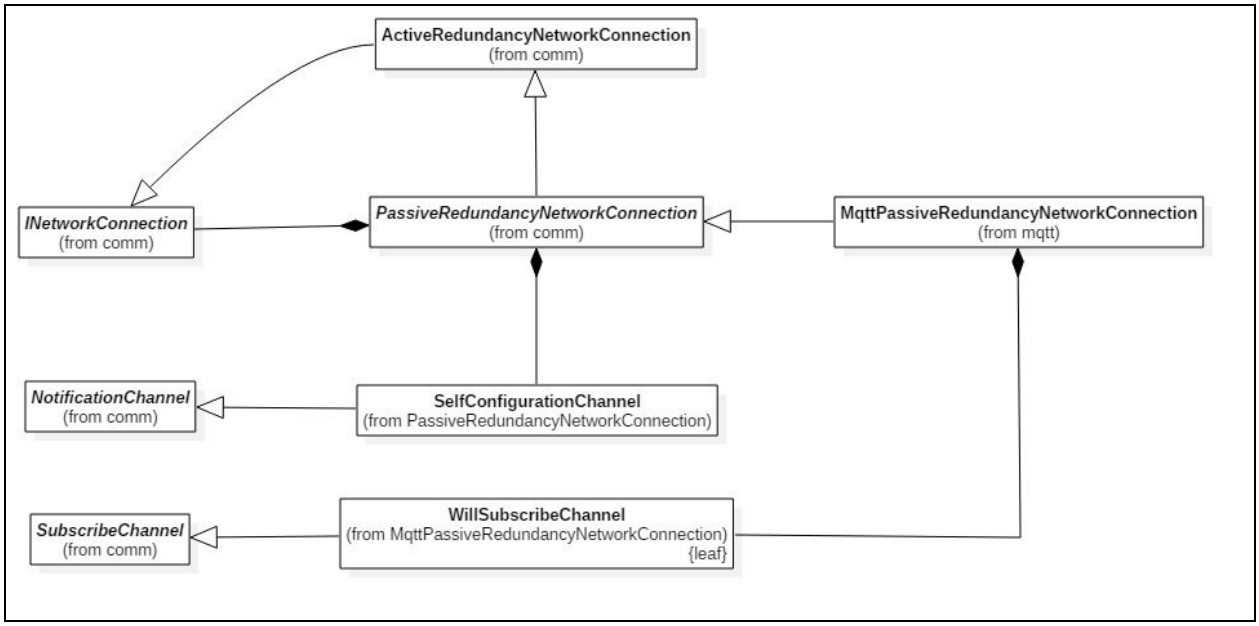

Figure 5. Passive Redundancy Class diagram (Static view, UML notation)

Figure 5 shows class diagram about passive redundancy network connection. Note that *PassiveRedundancyNetworkConnection* class is also *INetworkConnection*. Hence, even if the server does not use *PassiveRedundancyNetworkConnection* but use *INetworkConnection*, it needs not change any code. *PassiveRedundancyNetworkConnection* has *SelfConfigurationChannel* which is specified from *NotificationChannel*. As described in [2] [TEAM2_DOC_03] Architecture Document, each server competes with other servers by sending MS (Master Solicitation) message and MA (Master Advertisement) message. *SelfConfigurationChannel* is used to exchange MS and MA message between several servers. Because SurePark system uses MQTT (MQ Telemetry Transport) [3]

[TEAM2_REFERENCE_02] MQTT_v3.1.1 as a publish/subscriber protocol, MQTT based passive redundancy is implemented in *MqttPassiveRedundancyNetworkConnection*. In SurePark system, publish/subscriber broker use ping/echo tactics to detect unavailable servers. If there is unavailable server, publish/subscriber broker notify it to other servers through *WillSubscribeChannel.*

3. BUSINESS LOGIC CHANNELS

Refer to [1] [TEAM2_REFERENCE_01] Business Channels for detailed information about each channel used in SurePark system.