Smart Home SDK

George Boulos, JiEun Kim, Daniel Mosse
gwf5@pitt.edu, JiEun.Kim@us.bosch.com, mosse@cs.pitt.edu

School of Computer Science,
University of Pittsburgh

April 14, 2011

Abstract

Though there are various Software Development Kits (SDK) for different technologies, very few exist for Smart Home. The reason is that while allowing third party developers to create Apps is desirable, it introduces various complexities and risks. These risks vary from framework malfunctioning to abuse of Smart Home devices. In a sensitive environment like home the chances for failures are not tolerated. In this paper we will be identifying main concepts for Smart Home SDK as well as explaining our proposed solution. The expected audience is software engineers, system designers, and software architects.

1 Introduction

Most technology providers nowadays move aggressively towards providing a pluggable framework, a framework that third party developers are able to easily extend/change its behavior. The more Apps the framework has, the more it attracts buyers; also the more Apps are able to do, the more developers get creative, and the more framework attracts buyers. To provide a pluggable framework, you need to have a framework that supports Apps, and Appropriate development tools to create those Apps. Framework that supports Apps includes the definition of an App structure and App capability, answering the question what is an App able to accomplish through the system? Development tools include an Integrated Development Environment, Libraries and APIs of the framework, testing tools for specific parts of the Apps (like unit, integration, and performance testing), and simulation tools to test the App as a unit. In this paper I will be identifying

main concepts for transforming Smart Home to a pluggable framework, vulnerabilities that are introduced by doing so. I will also be discussing SDK alternatives and our proposed model. The rest of this paper is divided as follows section 2 provides related work, section 3 stake holders for SDK, section 4 existing Smart Home Architecture, section 4.1 Overall Architecture section 4.2 Messaging section 5 how to plug an App to the system, section 6 Conflict section 7 Conflict detection

section 8 OSGi switching, section 9 IDE and libraries section 10 Simulation alternatives section 11 Simulation tool current status section 12 Future Work section 13 Conclusion Appendix A how to create an App Appendix B how conflict detector was implemented

2 Related Work

2.1 Control 4

There exist only one SDK for smart home(to the best of our knowledge); namely Control4 system. Control4 system is divided into Navigator, Server and Director. Navigator is a graphical user interface device running flash Application that supports plugins. After establishing connection between Navigator and Director, plugins can navigate through devices and issue commands to those devices. Server is a web service server where developers can place custom web service to facilitate connection between Apps running on Navigator to connect to web. Director is the centralized controller of smart home, it provides some means of defining new drivers to control new protocol devices, yet not Apps can be deployed on it. This solution is very limiting to Apps that could run on the system, as the App is only running on the Navigator so it cannot extend or change the behavior of the system all it can do is provide a GUI to control devices. According to the plugins page for Control4 only one App that provides smart home functionality is available (one that changes door lock combination) all other Apps are non-related to smart home like weather feed and social networking.

2.2 DogSim

DogOnt is a Smart Home solution that relies entirely on Ontology to define services, as part of their system they provide DogSim a state chart simulator. Since "Apps" would be defined in ontology, DogOnt would have as input the ontology and would generates state transition. DogSim is DogOnt simulator. This project aims to run simulations and test for model and services defined

in ontology. The simulation offers an event-based interface allowing external Applications to dynamically interact with the simulation and provides hooks for listening to events and state changes of the simulated machines, and for injecting new events. DogOnt alternative of providing Apps is strictly limited to the capability of the Ontology (as expressive as it may be), this is not comparable of the power to allow third party developers to provide custom Applications to run as part of the system.

3 Stake Holders

Apps would be exposed to the following stake holders Device manufacturer, App Developer, and End User. For a device to be able to run as part of Smart Home system, device manufacturer should provide necessary driver to interface device with Smart Home system. Manufacturer should develop device driver, and run it on real system where various forms of testing should be carried on. Manufacturer should submit the driver to App repository where end user can find it. App developer is concerned with providing high level services and Applications. Developer should start an App development by installing SDK, then using System API's, examples and community support develop App. Developer should be able to test App, also run a simulation of the system with the newly developed App. After development and testing is done, App developer should submit App to App repository. End User should be able to browse new Apps to discover the newly published App. User should purchase new App, after then it would be installed and working seamlessly.

4 Existing Smart Home Architecture

4.1 General Architecture

In a protocol-agnostic smart home environment such as our system, different types of devices and services can be added and removed during runtime, and services within the system need to discover the existence of other services with which they need to interact. To attack this problem we use OSGi (Open Services Gateway initiative) as a basis for our framework. OSGi is a service oriented Java-based framework that enables modules(referred to as OSGi bundles) to expose services and consume exposed services. In addition to that OSGi has the ability to control bundle life cycle like start, stop, updat and uninstal without requiring a reboot. Other than the core

bundles, OSGi also has specification for services that are handy in several domains like Logging, Events, and Devices Management. Important architectural drivers for our smart home system are interoperability and dynamic integration of many types of drivers and devices developed by different vendors. For example, an X10-based lamp device need not to be discernible from an Insteon-based lamp device to such an extent that they provide the same functionality of on and off, and it should be exposed as a same abstract service. To support this paradigm we require a device stack and dynamic message-handling framework to process commands in different ways given various context. Inspired by OSGi that promotes the use of many interconnecting small modules making the system extensible (by adding new modules) and easily to manage (by changing existing modules), our system is not composed of small modules (managers), each communicating with other managers without depending on them. For example Device Manager does not even know about the presence of Semantic Manager in the system yet both of them communicate and accomplish a common task when a semantic service is invoked. This design is perfect for supporting Apps, actually each manager (other than core and Messaging) can be thought of as an App running on Smart Home. Here is a list of current managers within Smart Home System:

- Core provides interface definition, utility methods, and common constants. Any declaration that would be shared between smart home gateway modules (bundles) is defined in the Core.
- HIM (Hardware Interface Manger) is responsible to detect any controller device that is connected into the system. The OS reports events with information about devices to the HIM.
- Deployment connects to Application store in the cloud with properties of a controller devices discovered by HIM in order to download a proper driver for the new device. It manages software download and installation, and updates the home gateway profiles.
- Device implements our smart home device stack specification (which in its turn extends OSGi device specification), which includes devices, drivers, device discovery and command invocations.
- Configuration keeps track of all system configurations for persistent objects and ontology implementation.
- Ontology provides semantic knowledge about devices. Details are discussed in the Section VI.

- Database implements a persistent repository for device status and information.
- Scheduler provides capabilities for scheduling a job (e.g., control devices) such as at a given time or a recurring task.
- Authentication verifies a users credentials and returns the roles for the user if authenticated.
- Authorization is responsible for allowing or rejecting users requests by consulting the access control specification.
- Message FW implements our smart home message specification and framework specification. Details are discussed in the following section.
- UI Proxy provides web service interfaces to be used from within any client user interface devices such as smart phones or web browsers.
- Web server is a container for Restlet web server [25], which is a Java based Restful web server.
- Controller Discovery provides discovery of the controller devices together with HIM, Deployment and Device bundles.
- Device Discovery manages different discovery mechanisms specific to each protocol.
- Notify keeps track of all notification to be delivered to users in the home gateway, which include the notifications of new devices discovered, new software installation required and alert messages.
- Access Control manages authentication and authorization for different requests.

Once a device is discovered and registered, it is exposed through a Device Proxy object available through Device Manager. To invoke a command on that device (like turn on light) it is sufficient to issue the command to the device proxy where it would be forwarded through device stack till it reaches driver that physically (through sending bits over the wire) invokes the command. For more information about Device stack and discovery refer to original paper.

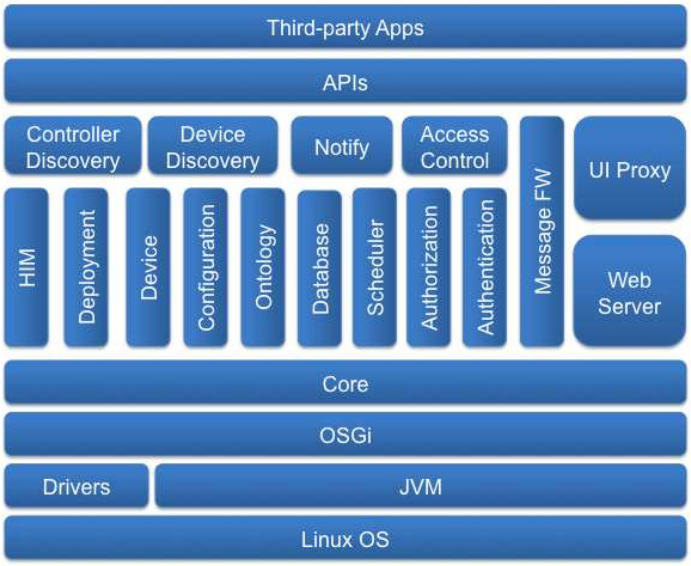

Figure 1: Framework Components

4.2 Messaging Framework

Now that there are several managers available through the system that are not even aware about the presence of other managers how would they communicate? We introduce a smart home messaging framework, the aim of this module is to provide a uniform way for process any event in the system. The content of the message varies depending on its purpose. For example, a message that is generated when the user wants to invoke a command to a certain device (e.g., turn on kitchen lights) is very different from a message signaling that a motion sensor has detected motion. The message is actually a simple wrapper for a generic hash table arranging fields as key, value fashion. To provide a uniform way of accessing the SHMessage, the concept of Message Helper is provided. Message Helper adopts the Decorator design pattern [26] that encapsulates SHMessage object and provides type-checking plane getter and setter methods to access the fields of that SHMessage. Messaging framework is composed of a universal SHMessage queue, where any entity would enque a message to process it. SHMessages are handled in a centralized yet dynamic fashion using a mixture of Chain of Responsibility and Strategy design patterns. A SHMessage is being handled by invoking a list of Message Handlers, which is a logical entity that is responsible for

executing a specific task such as raising an event throughout the system or log the message. The list of Message Handlers in this chain depends on the message type, which represents the purpose of the SHMessage. In this way we provide an extensible framework where handlers may be added and removed dynamically. More details on that in next section. Figure 2 illustrates how our message framework works for a native service. A web service creates a SHMessage with priority and timestamp to invoke an action on a device (set the temperature value) and put it in the queue. ChainsFactory knows the handler chains for Message Handlers. First, Event Handler notifies a new event to the listeners then passes it through a LoggingHandler, which logs this message with timestamp. Then a ServiceInvocationHandler, which sends the command of the action to the Appropriate ConrollerDriver. SHMessages are executed asynchronously, which is very useful for our system to enable SHMessage priority enforcement.

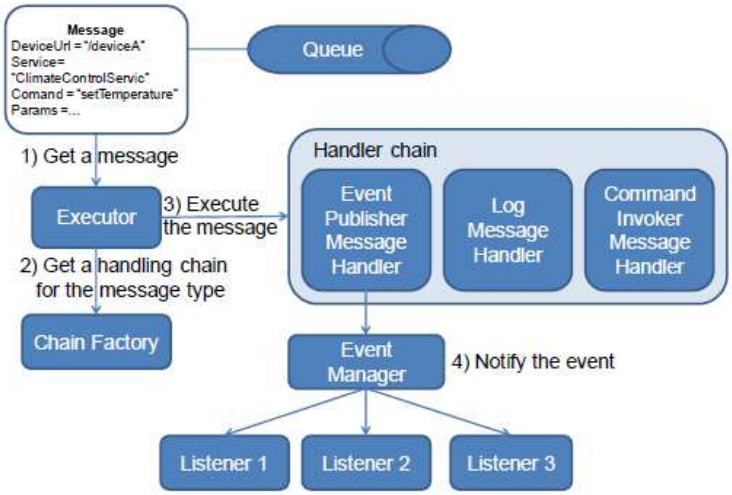

Figure 2: Messaging Framework

5 App Plugging

The core module of Messaging Framework is Chain Factory (Figure 2), this module is responsible to provide the correct chain of handlers to process a given message by examining the message. This module supports App plugin

in 2 ways, first requesting Event for a particular message type execution, second extending handling chain returned by main Chain Factory. Third party Apps can request to be notified whenever a specific message type is being executed, for example a turn on TV message can be detected enabling App to issue another message switching audio output to TV. This is done by simply registering a service with specific parameters through OSGi. Chain Factory by default does not include the handler to raise an event for any message execution, yet whenever a service requesting an event raising for some message execution is registered, Chain factory detects it and start raising events for requested message types. Chain Factory would raise an event if there is at least one App requesting the event to be raised. Sometimes Apps require more control over the executed message rather than just be notified when it is executed. Chain Factory enables third party Apps to do so when Apps register their own Chain Factory. Whenever this registration is detected main Chain Factory consult App Chain Factory when a new message is being executed and so App would be able to add Message handlers in a dynamic way. This would enable an App for example to limit the volume for TV to 70% after 8:00 PM. App Chain Factory can include smart algorithm do decide which message handlers to include in the processing chain, they can even invoke more Chain Factories composing a tree of Chain Handlers. Figure 3 shows an example of Handling Chain Tree. This

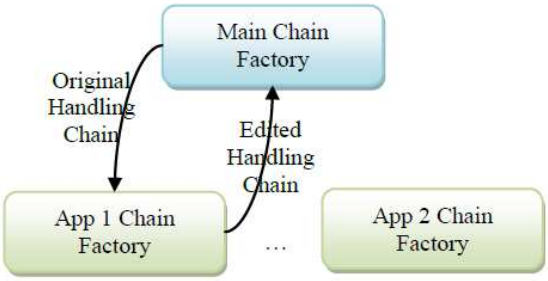

Figure 3: Chain Factory Tree

design is very dynamic and allows integration of third party Apps into the

system at runtime, yet as it is very powerful it may introduce some conflicts.

6 Conflicts

Smart Home system would be running several Apps at any time, and sometime these Apps would introduce a conflict with each other or cause some errors within smart home system. We identified three cases of malfunction. First, exceeding device limits; For example setting home temperature to 20 or 80 or switching a device on and off so frequently, such a case is easy to detect and safeguards should be implemented at the driver level to make sure no such case occurs. It should be placed at the driver level since the knowledge of the limits of this particular model of devices is only available at this layer. Second, App cycle negating each other effect, for example an App that whenever lights turn off turns them on, and another one that whenever lights turn on turns them off. Such a case is different than the first one, because we are not concerned here about device's limits yet about the convenience of Smart Home user. Sometimes the same user could have enabled two Apps that have this effect, some other times two different users started different Apps that contradicts one another. Third, Message Handlers overwriting values provided by other Message Handlers, for example if two profiles are running at the same time, given the time of day vacation profile (that should run when family is in vacation) tries to set temperature to 40 yet the following Message Handler of regular profile sets the same field to 50 overwriting the effect of the first one. This is different than the second case, in the second case we are talking about two Apps running in the background (passive Apps) listening for events and issuing other events. In the third case Apps are active they introduce their own ChainFactory and plug it at ChainFactory tree. There is no way to know before hand if two handlers contradict each other (Other than manual inspection of the code) since ChainFactories and handlers have all the freedom to implement conditions defining how to process the message at this part.

6.1 Conflict Detection

To support Smart Home Apps, we need to be able to detect such conflicts when they occur. The solution to the conflict should be centralized, and flexible. It should be centralized since it is not up to the device manufacturer to worry about this cases since it is introduced by system design and can be solved once and for all devices at the system core. I should be flexible enough to be able to detect different conflicts between Apps, actually

we are not even sure we identified all possible conflict situations so the solution should be able to accommodate new as well as very diverse conflict definition. We propose the use of a conflict policy (an input file) that defines what is a conflict, for example define that successive calls to turn light on or off for ten times within less than one second is a conflict, so is the call to adjust temperature for two times in ten minute period. We introduce Conflict Detection Manager, an App that reads this input file and detects this conditions when they occur. Conflict Detection creates InstanceConflictDetector for each rule specified in policy. InstanceConflictDetector keeps track of all messages executed for this particular service, and for each message execution check to detect whether a conflict is present or not. InstanceConflictDetector runs on a separate thread than the one executing the message to make sure this extra processing does not impose delay on current message execution, yet of course it increases the load of the system and may cause more delay for all system services. The policy should be able to provide a base case for all resources depending on their type, for example Apps change the light devices more than 5 times in a 15 min range are conflicting. This solution is able to detect second case conflicts. To be able to detect the third case we must keep track of all versions of a message processing, meaning comparing each message before being processed by a given handler and afterwards detecting fields that change and using similar policy to the case explained above detect conflicts. The computation for the latter case may be more than what gateway can afford since it requires comparing all message fields before and after each handler is invoked. Migrating these computation to the cloud may be a good idea. The results of conflicting Apps, and situations of when this conflict occurs could also be stored on a central repository so that the system can consult to alert users before installing those Apps.

6.2 Conflict Resolution

Once a conflict is detected we must disable one of the Apps causing the conflict. Determining which App to disable can be based on App priority, user initiating App, user decision, or even another policy defining which App should have the priority based on current system parameters. For example, Apps initiated by parents have by default higher priority than those created by children, yet in the case that parents are not home children Apps have the priority.

7 OSGi Switching

OSGi is a specification created by OSGi Alliance, so there are many implementations of OSGi. Our first Smart Home prototype was implemented on top of Prosyst's OSGi implementation, which is a commercial OSGi implementation. In order to allow more App developers to try our system and create more Apps, we needed to migrate the code to an open source OSGi implementation like Equinox. Equinox is by far the most popular OSGi implementation, it is maintained by Eclipse IDE community. Despite the fact that OSGi is a specification, so all implementations should be the same as far as an OSGi bundle is concerned, there were some few differences between Equinox and Prosyst that made this migration hard. These differences are caused by OSGi specification limitations. Though OSGi is a very popular specification, it missed specifying some utilities which caused some differences between different OSGi implementations. The ones that caused problems were, DeviceAccess, OSGi shell, and class Loaders.

7.1 DeviceAccess

According to OSGi, Device Manager is the entity responsible to detect newly registered devices and drivers, and match them together. DeviceManager should never be referenced, it is not a registered service, just a background thread. It happens that Applications (like ours) would need information about which devices or drivers are available and current mapping. OSGi specification did not anticipate this case, on the contrary Prosyst implementation did. Prosyst implementation provided a custom DeviceAccess module that implemented default OSGi specification and defined a way to listen to DeviceManager events like device attachment begin, end or fail. Our implementation depended on these events as this is how Smart Home keeps track of devices. We ended up editing Equinox DeviceAccess plugin to include a similar functionality, and edit Smart Home Device Manager module to use the new implementation.

7.2 OSGi shell

OSGi specifications also did not provide a specification of how to integrate with OSGi shell, like how to add commands that can be invoked from OSGi shell. Each implementation provided it's own way of how to integrate with OSGi shell, so we needed to change Prosyst shell dependencies to Equinox dependencies.

7.3 Class Loader

OSGi bundles depend on other classes provided in other bundles, OSGi is the one to intercept class loading calls and tries to locate the requested class within other bundles visible to requesting bundle. Unfortunately, there is a difference between Equinox and Prosyst for class loading. Equinox only loads any package starting with java from JVM, yet the limitation here is that it does not load any package starting with javax or any other JVM package that does not start with java prefix. In order to import these packages, they need to be specified in bundle manifest files, which was not required for Prosyst. The tricky part about this is that it was not documented at all, we actually found out about this reading exchanged mail between Equinox community. Java prefers lazy initialization to eager initialization, meaning that it would not inspect to see if all dependencies are met unless it had to instantiate this class right now. In order to complete the migration, we had to manually make sure that all processing paths for libraries we use imports from core libraries are available in manifest files.

8 IDE

Integrated Development Environment have been traditionally running as a desktop stand-alone Application. As web programming became more and more popular, and cloud computing infrastructure become more affordable, people move towards cloud based IDE. The advantages of this alternative is the increased portability, accessibility, ease of real-time collaboration, and ease of testing and deployment since the project is already available on the cloud. This alternative remains specially popular only for web development. Example of cloud based IDE are Ace, CodeRun and Kodingen. All of these IDE provide only web technologies. When considering how to present Smart Home IDE, the idea of placing it on the cloud seemed Appealing compared to stand alone IDE for many reasons like, ease of configurations, easy support for real time collaboration, and very small overhead to get developer starting. On the other hand stand alone IDE would give more control to the developer, and would be easier to implement. The App developer should be able to test his App on the system, so a simulation would be very helpful. Our Smart Home actually depends on some system libraries (though this dependency could be removed later), so the option of running the simulation in the cloud is really easier for the developer, as an alternative for following a 22 step install process. We decided to opt for a balance between the two alternatives, by using a standalone IDE and running the simulation on a virtual machine

(that could be running on the cloud). Also we could offer the option for App developer to use an all ready real Smart Home setup, including smart devices and system running through the cloud. This type of testing would be beneficial for hobby developers since they are not required to own a Smart Home solution to run real tests of their Applications, we call that Smart Home as a Service. We currently use Eclipse, the most popular IDE, as it supports OSGi plugins developemnt. We provide our framework as OSGi plugins to be referenced. Third party developers should only depend on two core Smart Home plugins, namely SHCore, and SHServices.

9 Simulation

Smart Home is a real system interconnecting with real devices, in this section I am describing the work done to provide a simulation solution for the system. To accurately run the simulation, we decided to use the real system. To make it easier for the developer to test, we use a virtual machine to run the system. The virtual machine image would have in place all libraries required by Smart Home system. As discussed in previous section, the simulation can be actually running on a real smart home system, or on a virtual one on the cloud. Figure 4 shows the complete simulation diagram. We created virtual driver a class of virtual devices that would contain a virtual device similar to every real device that can be connected to the system. Other than virtual driver and devices, we also needed to implement the following, UI Device Proxy, Control Terminal, and Virtual Smart Home GUI. UI Device Proxy is responsible for forwarding UI requests to simulation, request can either originate from smartphones on local area network or even smart phone simulators. Control Terminal is the OSGi terminal of Smart Home system that can be used to issue various commands to the system. Virtual Smart Home GUI is a GUI that displays all details about virtual devices, commands issued to these devices, and also provides an interface to create these devices. Figure 4 shows the diagram of simulation running on a virtual machine.

This setup would enable us to provide the following scenario

- Developer Machine:
  - After developing code, through the IDE, the developer should trigger a simulation start
  - It would run UI Device Proxy, Control Terminal, Virtual Smart Home GUI

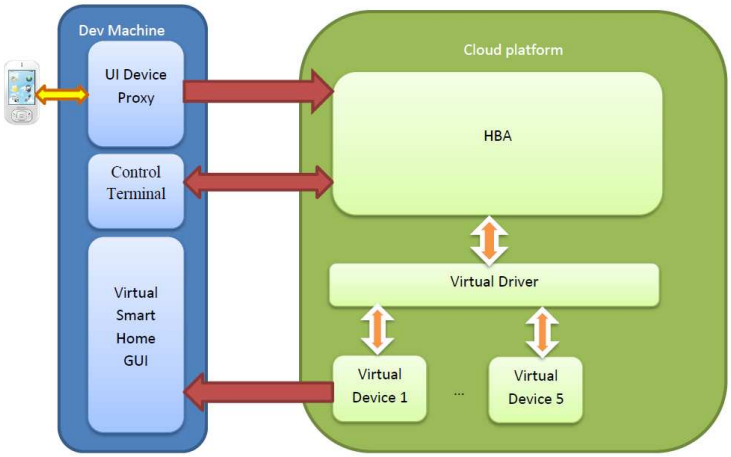

Figure 4: Simulator Diagram

  - UI Device Proxy forward request from smart phone or user to system, can even be a smart phone simulator
  - Control terminal, add, remove and control SHApps
  - Virtual Smart Home GUI is an interface that shows virtual devices, their statuses and commands they receive

- Cloud Platform:
  - A platform already set up and running Smart Home.
  - Smart Home system is controlling virtual rather than physical devices, the change is totally transparent for it
  - The virtual Driver and devices would be connected to Virtual Smart Home GUI updating it whenever status is changed or a command is recieved

10 Current Status

We currently have Smart Home system running on both Equinox, and Prosyst. We have several simulation devices along with the Virtual Smart

Home GUI. We have a conflict Manager proof of concept that is able to detect second case conflicts (and provides a model to follow to implement the one that detects third case conflicts). No policy is defined yet.

11 Future Work

We plan on starting to develop third case conflict detector, and also implement Conflict resolution. Some research needs to take place before deciding the policy to define conflicts. A more intuitive interface can be implemented for the Virtual Smart Home GUI, like a virtual reality inteface or even an intuitive map showing devices. We plan on doing some more research about how to be guarantee the correctness of an App, is there an automated way to authorize an App? We plan on creating a test scenario to simulate the performance of the system in real smart home environment.

12 Conclusion

In this paper we have presented the work done towards implementing a Pluggable Smart Home system that supports that can be extended through third party Apps. We have shown how we designed our system to be able to accomodate Apps, what kind of conflicts Apps could introduce and how to detect them. We also presented our simulation, and Smart Home as a Service concept.