A practical implementation of smart home energy data storage and control application based on cloud services

Mihail Lyaskov[1], Grisha Spasov[1], Galidiya Petrova[2]
Department of Computer Systems[1], Department of Electronics[2], Technical University of Sofia, Plovdiv branch,
25, Tzanko Dustabanov, 4000 Plovdiv, Bulgaria
{ m.lyaskov@abv.bg, gvs@tu-plovdiv.bg, gip@tu-plovdiv.bg}

Abstract – **This paper presents the development and realization of distributed energy data storage and control application based on cloud technologies. The LAN is built by cloud connected devices and Olinuxino A20 single board computer. The developed application utilizes microservices based architecture which improves code maintenance, enables easier scalability and improves the reliability of the system. The implemented application presents three services: DeviceHive connector, Database and Schedule service. Each of them uses its own instance of a database for storing service related data. This approach enables the performing of local control over the demand response without using cloud control logic. The application is built using mostly open source software modules.**

Keywords – **Smart Homes, Home Energy Management Systems, Cloud-based Smart Homes, Microservices architecture.**

I. INTRODUCTION

The emergence of cloud technology greatly simplifies the creation and subsequently the expansion of infrastructure for communication between different devices and clients (users) in applications such as smart homes [1]. Consequently, some applications for Cloud-based Home Energy Management Systems (HEMS) have appeared where intelligent devices with hardware and software layers can monitor, control and provide feedback on a home‘s energy usage [2]. The main task in HEMS is the realization of Demand Response (DR) optimization programs by means of which to reduce peak loads at consumer-end and improve energy efficiency. The integration of HEMS in Smart Grid Management Systems (SGMS) expands the effect of DR in larger area (regions) [3]. Generally the development of these systems is based on hybrid cloud architecture. In order to increase the reliability, in case of temporary loss of connection, the information from HEMS is stored centralized in cloud environment and also decentralized on the side of the smart home devices. However, independently of the storing location, the information is required to be extracted and processed towards fulfilling any given needs. This approach further enables the performing of local control over the demand response without using cloud control logic.

This paper presents a practical implementation of an application for distributed smart home energy data storage and control [https://github.com/MihailLyaskov/HomeassistApp]. The main goal here is to continue the work done by M.Shopov [4], by developing an application which can perform the tasks of: cloud gateway for different appliances, local energy data storage and device control based on work schedules. This is achieved by using microservices based architecture, which enables: faster development, easier integration of new functionalities, scalability and reliability of the system.

II. SYSTEM INFRASTRUCTURE

In this chapter we will discuss the infrastructure needed to deploy and test the smart home energy data storage and control application. There are a couple of reasons for which DeviceHive [5] framework was chosen for building this system over other frameworks:

- It is an open source project, which can be further extended and developed.
- It can be hosted in privet and public clouds.
- It implements a M2M (Machine-to-Machine) type of communication protocol.
- Presents open source libraries for communication with the cloud server, which can be used on different devices and platforms.

The DeviceHive framework presents REST and WebSocket based application programmable interfaces (APIs) which are consumed by devices and clients in the system and enables them to communicate in real time. Both devices and clients use JWT (JSON Web Token) to authenticate in front of the server. The different type of appliances are grouped together into logical networks based on their location. Each appliance can authenticate as a device with DeviceHive cloud service directly by itself or with the help of a gateway. After authentication devices perform a registration or update by providing “Device Name” and ”NetworkID”. After that they can start polling for commands coming from DeviceHive server and send notifications about their status. On the other side client applications don’t need to to registration or update. They just authenticate with JWT and can start polling for device notifications, send commands and get information about the server.

A server running the DeviceHive framework is currently deployed in Technical University of Sofia, branch Plovdiv as a private sensor cloud infrastructure. Figure 1 shows the overall view of the implemented system. It consists of local area network containing cloud connected appliances and

978-1-5386-1752-6 /17/$31.00 ©2017 IEEE

Olinoxino A20 [6], DeviceHive server and client applications.

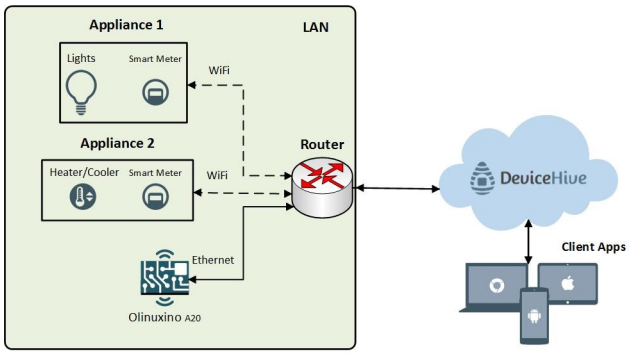

Fig. 1. The developed system infrastructure

Different appliances in LAN are connected to smart meters, which measure their electric power usage, calculate the consumed energy and send regular notifications to DeviceHive. Also they can receive commands and apply direct control over the appliances.

The smart home energy data storage and control application is being deployed on Olinuxino A20 single board computer. Olinuxino is running Armbian - Debian Jessie 8 build for Allwinner A20 SOC. The application uses DeviceHive REST API to communicate with the cloud server and uses JWT for authentication.

The last component in this infrastructure comprises the client applications. The current implementation of a client application is in a very basic state. It can plot time based graphics for consumed energy and can create working schedules for different devices using the services provided by the smart home energy data storage and control application.

III. Application Architecture

The presented application is written with JavaScript on NodeJS [9], which is an event-driven I/O server-side JS environment based on Google's V8 JavaScript engine. Because of its event-driven architecture NodeJS is capable of asynchronous I/O, which proves to be a good choice for real-time applications. By design NodeJS uses a single thread for execution and this could be considered as disadvantage to the system, but with the help of native child processes or JS modules like PM2(Process Manager) [10], many NodeJS processes can be started in parallel and can be monitored and controlled if needed. This led to the choice of using microservices architecture [11] for this application over standard monolithic architecture.

The usage of microservices over monolithic architecture brings some advantages on developing and maintaining such an application. Functionalities are grouped together to form services, which enables easier management over time and the possibility of easier and faster scaling of the application with new functionalities [12,13]. Each service can have its own database which helps in isolating data with different essence and assuring that if one database fails it will not affect the rest of the services. That also introduces a higher level of complexity when deploying the application and requires more memory. Different services can communicate with each other using inter-process communication (IPC), different publish/subscribe message buses or via REST APIs.

The application presents three services (Figure 2): DeviceHive connector, Database and Schedule service. Each of them uses its own instance of a database for storing service related data. They are interconnected by HTTP based IPC module SenecaJS [14]. Each service creates server on a specific port and opens client which can connect to services on other ports. Because of the small size of this application service discovery is not needed, so each client and server configuration is described in configuration file which all services have access to when initializing. SenecaJS provides pattern matching mechanism implemented in the bus driver for every service, so they can differentiate which messages are intended for them. Message patterns are being sent in JSON format and typically begin with service role, functionality and arguments. Each service uses its own database to save service related data. DeviceHive connector service and Schedule service use document-orientated database MongoDB to store respectively notification subscriptions and work schedules. The Database service is using InfluxDB, a time series database, to store power and energy consumption measurements sent from other devices.

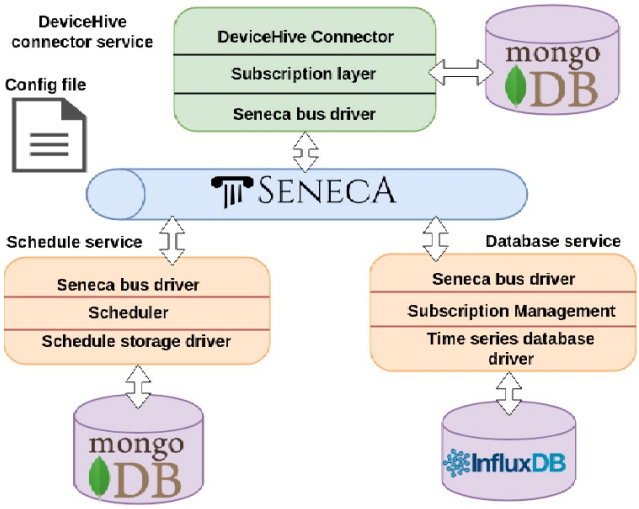

Fig. 2. The developed application architecture

The role of the DeviceHive connector service is to route commands between DeviceHive and the other application services. It registers the application as a device in DeviceHive framework and starts polling the server for commands. The incoming commands are mapped to command endpoints exposed by the other services from SenecaJS bus. The mapping is done in the configuration file. This service exposes its own endpoints on SenecaJS bus, which enables other services to: send commands to other devices, send and subscribe for notifications. Every time a new notification subscription is made, the connector starts polling the server for updates on this specific notification. The configuration of each subscription is saved to MongoDB database as a JSON object so it can be recreated after application restart. This persistent storage

ensures that the application will be able to restore its previous working subscriptions.

The supported functionalities by the schedule service are to: create daily schedules based on a JSON configuration, remove schedules and show all active schedules. On every new schedule the service subscribers, through DeviceHive connector, for notifications from the monitored device. The service uses a NodeJS module called “node-schedule”, which takes care of sending start and stop commands according to the working schedule. It also aggregates the consumed energy value from every notification and sends stop command if the device has reached the threshold energy value. All active schedules are stored in MongoDB database in order to be recreated after application restart or service failure.

Example schedule configuration:

```
{
      "DeviceID": "TestDevice",  // DeviceID
      "start": {              // DeviceHive command to turn
    on the device
          "command":"device/control",
          "parameters":{
            "state":"On"
          }
      },
      "stop": {                       // DeviceHive
    command to turn off the device
          "command":"device/control",
          "parameters":{
            "state":"Off"
           }
      },
      "schedule": [{                  // Array of start and
    stop times
          "beginTime": "13:00:00",
          "endTime": "14:00:00"
      }],
      "maxEnergy": 1500.0,          // Consumed energy
    threshold
      "notification": "device/init"   // Subscribe for this
    notification
  }
```

The database service supports the following functionalities: make notification, subscriptions through DeviceHive connector, remove notification subscriptions,
show all logged devices from database and present power and energy measurements from date to date. All the measurements are saved in a time series database (InfluxDB).

The usage of time series database gives some advantages over the standard relational databases. InfluxDB gives a less complex schema for storing and retrieving time series data. It also gives the possibility for extending the schema without affecting previously stored data. This database also presents two functionalities that give the chance to store data for bigger time intervals without exceeding the memory. They are called “continuous queries” and “retention policies”. The overall idea for using those functionalities is to down-sample the stored data. The continuous queries perform aggregation over the data and can produce one measurement per hour averaging the data from sixty measurements and store them in different database with no retention policy. The retention policies are used to describe for how long data can stay into the database before it is automatically deleted. In the end we can have a smaller database that can have high precision data (one measurement per minute) for the last 24 hours and lower precision but long term data for the last week, month or even a year.

IV. Testing the Application

There are two tests written, that exercise the schedule service and the database service. Both tests are can be found in the github repository for this project in directory /services/Devicehive/tests. The names of the tests are: integration TestDatabase.js and integration TestSchedulde.js. Both tests authenticate with DeviceHive, register a device named “TestDevice” and start sending commands to the application.

Pseudo code is used to represent the basic actions made by both tests.

```
START  integrationTestDatabase.js:
        Authenticate with JWT;
        Register “TestDevice”;
        Get test begin time;
        Send command “database/startLog” for
“TestDevice”;
        Send command “database/showSubs”;
        IF there is no a subscription for “TestDevice”:
                ERROR subscription not made;
                STOP TEST;
        ENDIF
        Send 3 notifications with power and energy from
“TestDevice”:
        Send command “database/stopLog” for
“TestDevice”;
                Send command “database/show Subs”;

        IF there is a subscription for “TestDevice”:
                ERROR subscription not removed;
                STOP TEST;
                ENDIF

                Get test end time;

                Send command “database/get Data” for
    “TestDevice” between begin and end time;

    TEST END


START  integration TestSchedule.js:
        Authenticate with JWT;
        Register “TestDevice”;
        Start polling for commands directed to
“TestDevice”;
        Send command “schedule/create” for
“TestDevice”  with 1 minute length and energy threshold
of 1500 W;
        Send command “schedule/show All”;
        IF there are no schedules:
                ERROR schedule not made;
                STOP TEST;
        ENDIF
```

Send notification from "TestDevice" with energy 1000W;
Send notification from "TestDevice" with energy 1000W;
IF "TestDevice" doesn't receive stop command:
ERROR energy threshold exceeded but application is not stopping "TestDevice";
STOP TEST;
ENDIF
Send command "schedule/remove" for "TestDevice";
Send command "schedule/show All";
IF there are schedules:
ERROR schedule not removed;
STOP TEST;
ENDIF
TEST END

IV. Conclusions And Future Work

The paper presents one practical implementation of distributed energy data storage and control application based on cloud technologies. It utilizes microservices based architecture which improves code maintenance and enables easier scalability of the system. The application is built using mostly open source software modules which makes it easier to run not only on embedded devices such as Olinuxino A20 but also in the cloud. In the future, this architecture will be added to a DeviceHive gateway and more services for connecting with appliances and devices will be provided. This will give the chance to control and monitor those devices even if cloud services go down or connection to WAN is not reachable.

V. Acknowledgements

The presented work is supported by National Science Fund of Bulgaria in the frame of the project "Investigation of methods and tools for application of cloud technologies in the measurement and control in the power system" under contract NSF E02/12 (http://dsnet.tu-plovdiv.bg/energy/).

References

[1] Zubair Nabi, Atif Alvi, *Clome: The Practical Implications of a Cloud-based Smart Home*, Cornell University CoRR, abs. 2014, ISSN:1405.0047.
[2] Poornima Padmanabhan and Gary R. Waiss, *Cloud - based Home Energy Management (HEM) and Modeling of Consumer Decisions*, International Journal of Smart Home Vol. 10, No. 8 (2016), pp. 213-232, http://dx.doi.org/10.14257/ijsh.2016.10.8.21.
[3] Y. Simmhan et al., *Cloud-based software platform for data-driven smart grid management*, Comput. Sci. Eng. (CiSE'13), vol. 15, no. 4, pp. 38–47, 2013.
[4] Shopov, M., *An M2M solution for smart metering in electrical power systems*, 39th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2016 - Proceedings, art. no. 7522311, pp. 1141-1144.
[5] http://devicehive.com
[6] https://www.olimex.com/Products/OLinuXino/A20/A20-OLinuXino-MICRO/open-source-hardware
[7] https://www.mongodb.com/
[8] https://www.influxdata.com/
[9] https://nodejs.org/
[10] http://pm2.keymetrics.io/
[11] Microservices architecture - https://martinfowler.com/articles/microservices.html
[12] Newman S., *Building Microservices: Designing Fine-Grained Systems*, O'Railly, 2015.
[13] Nadareishvili I., Mitra R., McLarty M., Amundsen M, *Microservice Architecture: Aligning Principles, Practices, and Culture*, O'Railly, 2016.
[14] http://senecajs.org/