Semantic rule-based device recommendation for service-migration in multiple-device contexts

Dimeth Nouicer, Nizar Messai, Yacine Sam, Ikbal C Msadaa

Semantic rule-based device recommendation for service-migration in multiple-device contexts

Dimeth Nouicer
LIFAT, ISTIC
University of Carthage
Tunisia
dimethnoucer@gmail.com

Nizar Messai
LIFAT
University of Tours
France
nizar.messai@univ-tours.fr

Yacine Sam
LIFAT
University of Tours
France
ycaine.sam@univ-tours.fr

Ikbal C. Msadaa
LARINA, ENSTAB
University of Carthage
Tunisia
ikbal.msadaa@ensta.u-carthage.tn

***Abstract*—Over the last few years, multiple-device ownership have continuously and rapidly increased, driven by the continuous progress of Internet of Things solutions deployment. At the same time, the way people use their multiple devices is also changing making it common and even necessary to switch from one device to another for the same task. This evolution raises several issues regarding the migration process and the choice of the device to migrate to. In this paper, we address the problem of recommending the most suitable devices for a successful migration. We propose a semantic and rule-based approach for device recommendation that takes into account the characteristics of devices, services, and migration contexts. We propose an OWL ontology defining the necessary concepts describing devices, services, users, etc. and the relationships between them. Based on this ontology, we propose a set of SWRL rules defining the common requirements for a successful service migration across devices. The ontology is populated in real-time using appropriate APIs to get devices and context characteristics. This allows the rule-based reasoning to be held over updated values and provides relevant recommendations. The approach is implemented as a recommendation system within a web application to demonstrate its effectiveness and scalability.**

I. Introduction

Nowadays, multiple-device ownership is increasing more and more driven by the high availability of smart objects, their increasing performances and the progress of data management and transfer protocols. It is indeed expected that by 2025, more than 75 billion smart devices will be in use, according to Statistica[1] and Cisco[2]. It is now common to use simultaneously or in sequence at least 2 smart devices by the same user to perform daily tasks. Especially with the paradigm of Everything-as-a-Service (XaaS), people can access different services anywhere, anytime and from almost any device. In this context, multiple-device ownership raises new challenges when users run services over their different devices. One of the very common challenges to switch from one device to another while keeping interaction with the same service. An example that fits perfectly with the world's current situation is online conference attendance and animation. Because of Covid-19, users need to attend conferences online using their devices, but they also might need to go to work or switch location for some reason. For example, first a user will be attending with a laptop at home, then he would switch to his smartphone when driving to his office while still keeping the call on. And when he arrives, he would want to use the TV or the desktop for a better user experience.

In this scenario, a migration process needs to be triggered to allow to smoothly move user-service interaction between devices. Today, cloud/server-side synchronization allows such a migration but requires for each device to be individually connected to the server, as it is the case of Google suite (Docs, Meet, etc.), Android and others [1], [2]. Also, some approaches such as LiquidJS [3], which is based on Liquid Software principals [4], allow to adapt services to device characteristics. Recently, approaches such as CUBE [5] have proposed to deal with the migration of REST services from a user-side perspective without depending on server/cloud synchronization. However, whatever is the considered approach, **the user needs to manually select the target device to migrate to**.

Furthermore, in multi-device environments, client-side synchronization is not only a problem in user-service interaction scenarios, but also in data centers and Internet of Things (IoT) firms. This may concern for example the offloading tasks in an IoT Multi-Edge Computing (MEC) context. The need to migrate content freely between devices seems to become a must. In these specific scenarios, **the migration decision and the target device choice should be made automatically** as there are no users to monitor and trigger the migration all the time.

A successful migration decision, whether it is done automatically or manually, requires to follow some criteria to know which target device is the most suitable for the service/application that will be migrated. Also, it has to capture the users preferences, timeline and other features related to security, reliability, compatibility of devices, constraints of the services, etc., in order to be able to suggest the right device. This issue has been widely covered from the cloud/server perspective, as part of cloud resource management [6] or in the context of IoT device management [7]. **From user/edge perspective however, efficient device recommendation modules are still to be defined**. Such issue is more challenging as far as we consider the limited device computation performances

[1]https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/

[2]https://www.cisco.com/c/en/us/solutions/internet-of-things/future-of-iot.html

and the lack of access to large semantic resources, such as Linked Open Data, for similarity calculation.

In this paper, we tackle the device recommendation challenge which will help users achieve a smooth service migration between devices from user/Edge side within a decentralized approach that roams between users devices. To address this challenge, we propose a declarative approach that relies first on semantic web ontologies and inference rules to provide a device recommendation system that can be integrated in existing migration solutions. The approach is built upon three main contributions: we first model the main concepts describing devices, services, users and contexts into a small OWL ontology that can be held on each device independently. Second, we define a set of typical semantic inference rules, in Semantic Web Rule Language (SWRL), formalizing the migration requirements based on the concepts of the ontology. Third, we implement the approach within a prototype that populates the ontology based on APIs and user interactions, applies a reasoning through the semantic rules, and infers device recommendations for service migration.

The rest of the paper is organized as follows: Section II discusses related contributions that align with our motivation and work. Section III presents the proposed declarative approach for device recommendation from user perspective, using ontologies and inference rules. Section IV illustrates the proposed approach on a typical migration scenario. Section V details the evaluation settings to demonstrate the effectiveness of the proposed approach. Finally, Section VI concludes the paper and highlights related future research directions.

II. Related Work

In the literature so far, the challenge of migrating running services/tasks between devices was defined in two manners: either with cloud computing paradigm or with user side (client repository architecture).

The first perspective is cloud computing where this challenge is referred to as part of Elasticity which consists of increasing or decreasing allocated resources for a given task or process depending on its execution needs (computing, storage, etc.). A detailed survey on cloud elasticity approaches and industrial solutions is given in [8]. The basic idea is that for each process, a Service Level Agreement (SLA) is defined, and resources allocation is then managed to avoid SLA violation. Adapted languages such as SYBL [9] allows to express necessary constraints to trigger resource allocation/reduction.

The second perspective is user side. A survey on approaches from this perspective is given in [10]. There, Palmeira et al. classify state-of-the-art approaches based on operating system, on synchronization method: server-side vs user-side, etc.
An interesting approach is LiquidJS [3]: a framework that enables the creation of distributed Web applications running on multiple devices based on Liquid Software principles, a paradigm that adapts services to devices to run on. The migration process consists in synchronizing only necessary parts of the service execution between devices using "Liquid components" on top of Web Components.
The approach that is most likely to work with our recommendation system is CUBE, described in [10] and illustrated through another prototype in [11]. CUBE also follows Liquid Software principles but from a user-side to migrate service states from one device to another without any server synchronisation. The framework is designed through 3 main components: A set of services, called OUTER CUBE, a set of user devices to interact with services, called INNER CUBE, and an interaction space between both, called POOL AREA. The migration relies on the transfer of REST service status between devices in the interaction space.

In this paper, we rather focus on approaches that manage service migration between devices from a user/edge perspective. These approaches focus on the migration process, that is service state synchronization, data transfer, etc., in a rather empirical way where the choice of the target device to migrate to is made by the user. Although Palemeira et al. describe a search space that offers a list of available devices when the user triggers the migration request, there is no an in-depth study of their characteristics and context of use nor a reasoning on their ability to host and successfully run the migrated service. The present work is intended to complete the state-of-the-art approaches by a recommendation phase that provides users and systems with the right choice of device to achieve a smooth and successful migration. The recommendation relies on a declarative framework to capture devices and services characteristics following a proposed ontology. It then defines a set of semantic rules over these characteristics to recommend or not a device for the migration.

Considering the semantic aspects, ontology-based user-side approaches are also to be defined. From server-perspective, however, many approaches have been proposed to manage cloud resources. Among them we can cite the mOSAIC ontology [12] for cloud resources annotation and management intended to ease multi-cloud oriented applications development. In the same context, the approach in [6] deals with job allocation in cloud systems using OWL ontology and SWRL rules. In both approaches, large ontologies (in terms of concepts and instances) are defined to cover all the aspects of the cloud systems. In the present work, limited computing capacities from user-side perspective require to use a rather light weighted ontology and a set of rules that can easily be run on users devices to identify the next device to migrate to. The designed ontology is therefore only focused on the main concepts to describe commonly needed characteristics for service migration.

III. Declarative framework for describing devices and migration rules

A. Approach overview

In order to better position our contribution, we summarize in Figure 1 the overall architecture of service migration in multiple devices environment from user perspective. The architecture is decomposed into two main steps. First, the device discovery and recommendation phase, shown in the

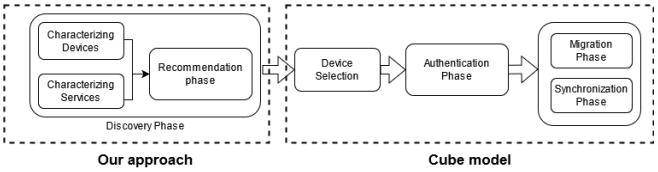


Fig. 1: Overall architecture for device recommendation and service migration in multiple device contexts.

left side of the figure. This part is composed of the following 3 modules:

- Requirements characterisation of services to be migrated;
- Device discovery and characterisation in the user context;
- Matchmaking check and recommendation of the most suitable device for migration.

Second, the service migration phase, shown in the right side of the figure, which consists of 4 main modules:

- Target device selection to start the migration;
- Authentication on the target device;
- Service state/data migration to the target device;
- Additional synchronization, if needed, to keep the interaction progress.

As discussed in the state-of-the-art section, service migration is covered by approaches such as CUBE, a user-side migration framework proposed in [5]. We will not therefore review it in details. We simply point out the main issue with device selection, prior to any service migration in CUBE as well as LiquidJS [3] which is the following; Usually, this selection is manually performed by the user regardless of the devices characteristics. However, this may cause failures in the service execution in the target device. To overcome this issue, we propose to complete the existing migration frameworks with a reliable device recommendation phase. The proposed approach should take into account two main aspects related to user-side perspective: (1) the highly dynamic context regarding devices availability, and (2) the least computation possibilities for the recommendation process.

Considering the first aspect, we propose a declarative approach that relies on semantic description of contextual and real-time device characteristics, as well as a set of logical rules to select the most suitable device to migrate to. The second aspect requires to define a rather lightweight ontology and a limited set of typical rules to allow running the ontology population and the reasoning tasks on the users devices. Each device deploying the recommendation module runs it independently to get its own migration context and keep it updated once in use. The recommendation module provides a first set of rules that covers the most common migration scenarios, and allows users to edit and enrich these rules for a personalized user experience. The following subsections provide a detailed description of the proposed approach.

B. Semantic description of the recommendation context: OWL ontology CORE concepts

In this section, we present an ontology that models our system and deals with the information related to our resources (users, devices and services). The task of discovering devices and recommending the potential device can be sensitive and complex as the services requirements differ from one another. Also, devices have a wide range of characteristics that can be very dynamic.

Thus, we conceptualize our resources into 4 main components: (i) the device, (ii) the owner, (iii) the service and (iv) the characteristics.

Device This class contains three sub-classes: UserCentric, CloudDevice and IoTdevice. Each of them contains specific sub-classes corresponding to the device type (smartphone, laptop, etc.).

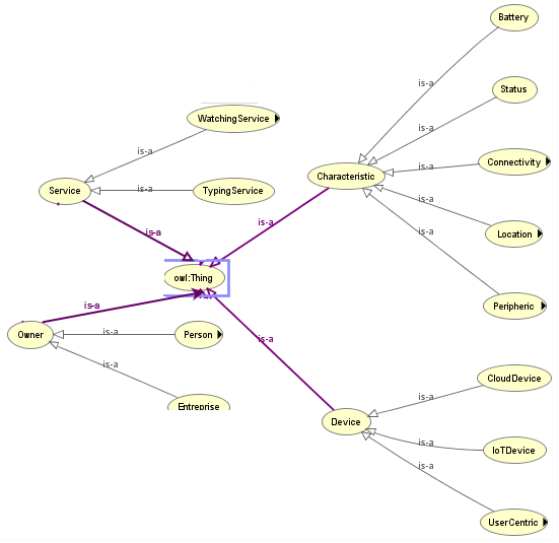
Owner This class contains individuals that interact with both the device and the service classes. Eventually, for every device there should be an owner, whether it is a person or an enterprise it does not matter. For security reasons, we cannot recommend a device owned by a stranger; so we need to know the owner of the devices. As mentioned in [13], this is called ownership device discovery.

Service This class is crucial in our model because based on the service needs and requirements, we will decide about the suitable device. It can also be decomposed into sub-classes based on the type of the service.

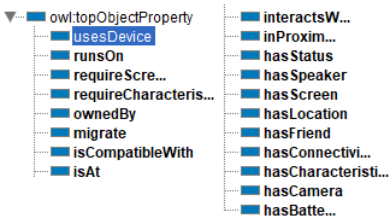
Characteristic This class includes all the necessary characteristics of both the services and the devices. Characteristics of the devices can be like location, battery percentage, screen resolution, etc. In the same way, services also have some characteristics, can be called requirements, like whether it requires Internet connectivity or not, or whether it needs a high screen resolution (e.g. 4k videos), etc. These characteristics are important as they can help us differ between the potential devices and match between the service and its suitable device. For example, the location characteristic can be used in a location discovery process, as explained in [13], that only discovers nearby devices.

Figure 2a illustrates an overview of the 4 main classes and few examples of their sub classes.

We also define necessary properties to model the interactions and relationships between theses concepts. For example, the owner class has two object properties with the device class: a device is **ownedBy** an owner, and the owner **usesDevice**. These two properties are different because a user can own several devices but only uses one of them at a time. Another property is: the user **interactsWith** the service. Furthermore, in order to define the relationship between the devices and their characteristics or the services with their requirement, we add two main properties. **hasCharacteristics** property: links between the device and the characteristic class, while the **requireCharacteristics** property: links between the service and the characteristic class. Moreover, between the device and the sub classes of characteristics there are also object properties as illustrated in the example in Figure 2b.



(a) The ontology main concepts.



(b) The ontology object properties.

Fig. 2: Overview example of the ontology

C. Real-time ontology population/update based on devices APIs and contextual information

After designing the ontology classes and properties, we need to fill the instances with the users devices and context information. The optimal way to do this would be to get APIs do the job, as we should keep users intervention as minimal as possible to facilitate the process. For instance, Battery API can give us: the battery percentage of devices, whether it's charging or not, etc. We don't want users to manually update the ontology with their devices information every time they need to do the migration. Also we cannot save the information once we get it because it's too dynamic (saving battery percentage is meaningless as it's always changing). That's why, ontology-update-process starts either when the user manually triggers a migration request or when predefined callback methods are automatically triggered through specific events related to service or device life-cycle. However, APIs cannot get all the information needed; that is why we might need users to enter/complete them manually.

D. Formalizing migration rules with SWRL

Now we add the inference rules which are based on the properties and classes we have defined above. Based on the service requirements, we will try to recommend the suitable device using SWRL rules. Table I presents some of the simplified inference rules that we can define to manage the recommendation. For example, rule number 1 states that devices that have more than 10% battery percentage are considered alive. Rule 2 for example gets us devices that are in proximity. Thanks to these rules, we can achieve various results and semantic meanings from all the information we have. However, sometimes we need to combine rules together to dictate certain orders like the migration rule. As an example, let us consider rule 4 that combines rule 1,2 and 3 in order to achieve a successful migration.

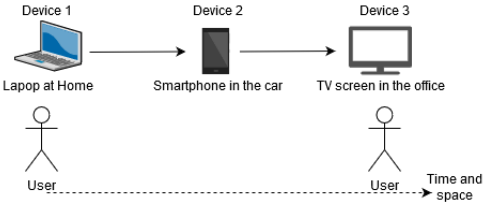


Fig. 3: Illustrative scenario.

No	Domain rules
1	$Device(?d) \wedge HasBattery(?d, ?b) \wedge swrlb:$ $greaterThan(?b, 10) \rightarrow Alive(?d, true)$
2	$Device(?d1) \wedge Device(?d2) \wedge HasLocation(?d1, ?l)$ $\wedge HasLocation(?d1, ?l) \rightarrow InProximity(?d1, ?d2)$
3	$Device(?d) \wedge HasScreen(?d, ?c) \wedge WatchingService(?s) \wedge$ $RequiresScreen(?s, ?c) \rightarrow IsCompatiblewith(?d, ?s)$
4	$Device(?d1) \wedge Device(?d2) \wedge Service(?s) \wedge Alive(?d2, true) \wedge$ $InProximity(?d1, ?d2) \wedge IsCompatiblewith(?d2, ?s)$ $\rightarrow Migrate(?s, ?d2)$

TABLE I: Examples of Inference rules

As seen, these inference rules can be very flexible and they give us efficient semantic meanings to achieve relevant device recommendation.

IV. Illustrating the approach on a typical migration scenario

After explaining the recommendation process, the ontology and the SWRL rules, we provide in this section a full example to summarize and bring together the three aspects detailed above.

As illustrated in Figure 3, we consider a user who uses three devices: a laptop, a smartphone and a TV. As for the service, the user needs to attend a conference online so this is a videoconferencing service. First, he starts the call at home with his laptop. Second he switches to his smartphone when he's driving to the office, so that he doesn't miss anything from the conference. Third, he arrives at the office and he follows the video conference on the TV screen for a better user experience.

This scenario can be modeled with our ontology as illustrated in Figure 4. We have the device class and the three instances: device 1,2 and 3. In Figure 5 we have the service class and the conference instance. The service in this scenario requires a good connectivity, good screen resolution and a high battery percentage. That's why, in the device's characteristics

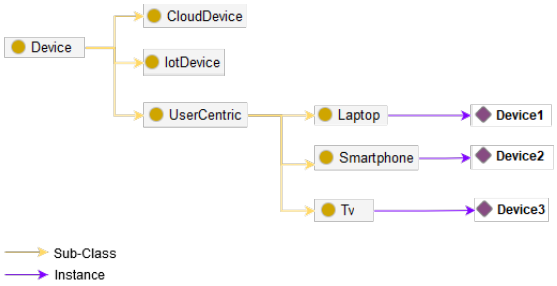


Fig. 4: Device class that illustrates the scenario.

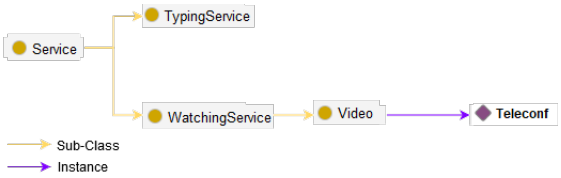


Fig. 5: Service class that illustrates the scenario.

we focus on satisfying the service requirements.
In this Table II we showcase few example rules that will help us recommend the suitable device for the service.

No	Domain rules
1	$Device(?d) \wedge HasBattery(?d, ?b) \wedge swrlb:$ $greaterThan(?b, HighValue) \rightarrow Alive(?d, true)$
2	$Device(?d1) \wedge Device(?d2) \wedge HasLocation(?d1, ?l) \wedge$ $HasLocation(?d1, ?l) \rightarrow InProximity(?d1, ?d2)$
3	$Device(?d) \wedge HasConnectivity(?d, ?c) \rightarrow IsOnline(?d, true)$
4	$Device(?d) \wedge HasScreen(?d, ?c) \wedge WatchingService(?s) \wedge$ $RequiresScreen(?s, ?c) \rightarrow IsCompatiblewith(?d, ?s)$
5	$Device(?d1) \wedge Device(?d2) \wedge InProximity(?d1, ?d2) \wedge$ $Alive(?d2, true) \wedge IsOnline(?d2, true) \wedge Service(?s) \wedge$ $IsCompatiblewith(?d2, ?s) \rightarrow Migrate(?s, ?d2)$

TABLE II: Inference rules of the video conference scenario

V. Implementation and Evaluation

In order to evaluate our ontology, We have implemented a simulation model. Protégé was used to design the ontology, while Apache Jena was used for reasoning and the inference engine to test the SWRL rules. To evaluate the recommendation system, we have conducted several experiments with different scenarios and compared the theoretical results with the output of the system and the ontology.

In our experiments, we have developed an application (app) that consumes a Spring Boot API. We assume that the users of this application are the people who need to migrate their services. The API is responsible for handling the Jena reasoning and the SWRL rules. In other words, the API manages and stores the facts and rules (both defined in Apache Jena) to finally compute the recommendation results. For instance, the facts are very dynamic and change depending on the situation and the devices. They are stored in a ".n3" file. As discussed in Section 2, the facts can vary from one user to another. Thus, the facts in the n3 file change from one scenario to another. The characteristics, in the file, are added

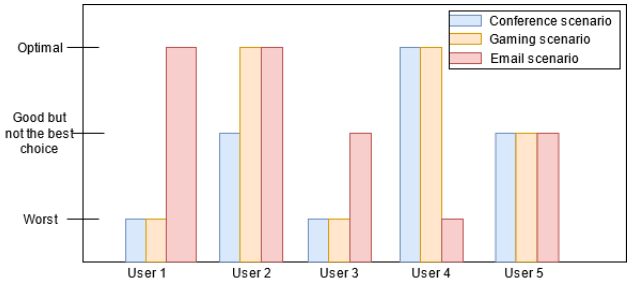


Fig. 6: Evaluation of users behaviour without recommendation system.

by the app that we have developed. During the configuration step, the app asks the user to add some of them manually from a web interface (for security and permission reasons), while some others can be extracted automatically from APIs (battery API, Bluetooth API, etc.) as detailed in paragraph Real-time ontology population. However, the rules are in a second file and they don't change dynamically. The rules are the same for all scenarios unless the user wants to modify them.
The code of the application we developed can be found in Github [3].

In order to validate the efficiency of our system, we have carried out several performance evaluation scenarios. As illustrated in Figure 6, we have tested the behaviour of 5 users in different situations with 3 services (video-conference, gaming and email). Usually, users would choose devices intuitively and without thinking through their choice. In these different situations, we've realized that the choice they make is not necessarily the best. As seen in Figure 6, in two scenarios user 1 made the worst choice of devices: device characteristics don't match the service requirements (e.g., device is not connected to Internet, or doesn't have enough battery).

Next, we gave users our recommendation system to test and measure their satisfaction with the recommended device. As illustrated in Figure 7, some users are totally satisfied with the result, while others are the complete opposite. For example with user1, although he made the worst choice in the conference scenario, he still insisted on using that device. Overall, users who weren't satisfied with the result are either users who insisted on using the same device and to not do the migration in the firs place, or the recommended device was already being exploited by someone else or by another service.

Here are few details of some of the users and how the recommendation system is making the choice. For every scenario there's a unique n3 file created in our app. For every new case of migration the n3 file resets the characteristics retrieved from both the APIs and the user as we previously discussed.

The first scenario is about attending an online conference. The first case is related to user 1. We have these devices and their characteristics as listed in Table III. In this case, the user

[3]https://github.com/dimeth-nouicer/WETICE.git

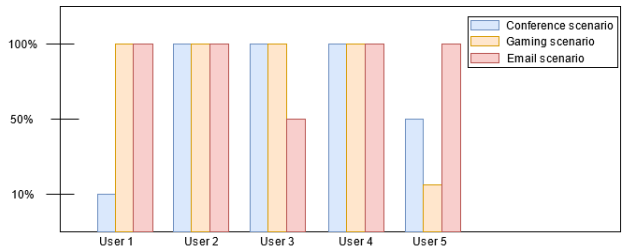


Fig. 7: Evaluation of users satisfaction with our recommendation system.

has chosen to watch from **Tablet1** in red while the system recommended **Tablet2** in blue.

devices	location	ownership	battery %	Internet connectivity
Laptop	home	user 1	100%	yes
Phone	car	user 1	80%	yes
Tablet1	car	user 1	10%	yes
Tablet2	car	User 1's son	70%	yes

TABLE III: Devices and their characteristics in the first case of the conference scenario

In the second case, with user 3, we modified some attributes: the location of Tablet1 and the Laptop, and the Internet connectivity of the Laptop as listed in Table IV. The system still recommends **Tablet2** in blue, but user 2 chooses his **Phone** in red.

devices	location	ownership	battery %	Internet connectivity
Laptop	car	user 1	80%	no
Phone	car	user 1	50%	yes
Tablet1	home	user 1	80%	yes
Tablet2	car	User 1's son	80%	yes

TABLE IV: Devices and their characteristics in the second case of the conference scenario

The second scenario is about gaming service. The first case is about user 1 and we have the different devices and their characteristics listed in Table V. The user chooses to play on the **Desktop** in red while the system recommends the **Laptop** in blue.

devices	location	ownership	battery %	Screen Resolution
Phone	home	user 1	50%	HD
Laptop	home	user 1	70%	HD
Tablet	car	user 1	80%	HD
Desktop	home	user 1	10%	Best quality

TABLE V: Devices and their characteristics in the first case of the gaming scenario

As explained in these scenarios, our recommendation system is dynamic and adapts to different services and different use cases. In some cases, it verifies the battery percentage, in other it prioritizes the screen resolution and Internet connectivity. However, it's important to mention the limits of our system and our ontology. As mentioned above, in one of the scenarios our system recommended a device that was already being used by another user. For instance, based on the characteristics alone, our system is not able to detect, for now, whether a device is being used or not.

VI. Conclusion

In this paper, we present a declarative approach for device recommendation to allow smooth and successful service migration in multi-device contexts for both user-centric and MEC IoT scenarios. The approach consists in defining an OWL ontology for describing the devices, the services to migrate, the user and the context. Based on the ontology concepts, we define a set of SWRL semantic rules to ensure relevant recommendation of target devices while keeping user-service interactions on. We've illustrated and validated the approach through different scenarios to show its relevance and how it can adapt to different environments. We've implemented the proposed framework in a simulation application to showcase real life scenarios and results.

Regarding our future steps, we intend to combine our work with few prototypes such as CUBE or LiquidJS to provide a full migration solution framework in user-centric contexts as well as in Edge IoT contexts. Further tests will be done to provide a full user experience that's automatic, to improve our ontology and its limits.

References

[1] P. Hamilton and D. J. Wigdor, "Conductor: Enabling and understanding cross-device interaction." Association for Computing Machinery, 2014.

[2] D. Wolters, J. Kirchhoff, C. Gerth, and G. Engels, "Cross-device integration of android apps." Springer International Publishing, 2016.

[3] A. Gallidabino and C. Pautasso, "The liquid.js framework for migrating and cloning stateful web components across multiple devices," 04 2016, pp. 183–186.

[4] J. Hartman, U. Manber, L. Peterson, and T. Proebsting, "Liquid software: A new paradigm for networked systems," Tech. Rep., 1996.

[5] C. Palmeira, N. Messai, Y. Sam, and T. Devogele, "User-side service synchronization in multiple devices environment," in *Web Engineering - 20th International Conference, ICWE 2020, Helsinki, Finland, June 9-12*, ser. LNCS, vol. 12128, pp. 451–466.

[6] Y. Ma, S.-H. Jang, and J. Lee, "Ontology-based resource management for cloud computing," 04 2011.

[7] F. Komeiha, N. Cheniki, Y. Sam, A. Jaber, N. Messai, and T. Devogele, "Towards a Privacy Conserved and Linked Open Data Based Device Recommendation in IoT," in *International Conference on Service Oriented Computing, ICSOC 2020*, 12 2020.

[8] E. Coutinho, F. Sousa, P. Rego, D. Gomes, and J. Souza, "Elasticity in cloud computing: a survey," *annals of telecommunications - annales des télécommunications*, 2014.

[9] G. Copil, D. Moldovan, H.-L. Truong, and S. Dustdar, "Sybl: An extensible language for controlling elasticity in cloud applications," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 05 2013.

[10] C. Palmeira, N. Messai, and Y. Sam, *CUBE System: A REST and RESTful Based Platform for Liquid Software Approaches*. Web Information Systems and Technologies, Springer International Publishing, 06 2018, pp. 115–131.

[11] ——, "Liquid mail - a client mail based on cube model," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 07 2018.

[12] F. Moscato, R. Aversa, B. Di Martino, T.-F. Fortiş, and V. Munteanu, "An analysis of mosaic ontology for cloud resources annotation." in *Proceedings of the Federated Conference onComputer Science and Information Systems*, 2011.

[13] A. Gallidabino, C. Pautasso, T. Mikkonen, K. Systä, J.-P. Voutilainen, and A. Taivalsaari, "Architecting liquid software," *J. Web Eng.*, 2017.