DaaS: Cloud-based mobile Web service discovery

Khalid Elgazzar*, Hossam S. Hassanein, Patrick Martin

School of Computing, Queen’s University, Canada

ARTICLE INFO

Article history:
Received 25 February 2013
Received in revised form 29 October 2013
Accepted 31 October 2013
Available online 14 November 2013

Keywords:
Cloud computing
Context-aware
Mobile Web services
Service discovery

ABSTRACT

The proliferation of smartphones and the recent advancement in ubiquitous wireless access have made mobile Web services more possible than ever before. However, finding relevant Web services that can match requests and fit user context remains a major concern. The challenges facing Web service discovery are further magnified by the stringent constraints of mobile devices and the inherent complexity of wireless heterogeneous networks. Cloud computing, with its flexible design and theoretically unlimited computing resources, is a viable approach to bootstrapping Web service discovery. The cloud can build bridges between mobile devices, as a convenient ubiquitous interface, and a backbone infrastructure with abundant computing resources. This paper introduces “Discovery as a service (Daas)”, a novel cloud-based discovery framework that addresses the core components of mobile Web service discovery. The Daas framework lays the foundation of efficient mobile Web service discovery that takes into consideration user preferences and context. The experimental validation and performance evaluation demonstrate that Daas can effectively rank relevant services according to the various user context and preferences, in addition to enhancing the precision of the discovered services. The prototype also shows that Web service clustering for discovery significantly improves the overall response time, while the cloud maintains scalability according to prespecified performance criteria.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

The Web services approach is a key enabler of seamless integration between heterogeneous applications and software systems. Web services also can be consumed by users on the fly, given that a user-friendly interface is available that enables users to efficiently communicate with Web services. User-facing Web services are on the rise due to the proliferation of mobile devices and the advancements in ubiquitous wireless communications. RESTful Web services are capable of communicating with both applications and users via dispatching the appropriate service response to the type of service request. For example, if the request is sent by an application, an XML or JSON-formatted response is dispatched to the application, where a response with HTML format is sent to requests by Web browsers (i.e. users).

The successful implementation of Web services starts with finding relevant services that best accomplish a particular objective and are appropriate for the current context [1]. Thus, efficient discovery mechanisms for finding, ranking, and selecting the appropriate Web services are crucial to the success of adopting the Web services approach. However, due to the lack of such robust discovery techniques that understand the user preferences and context, Web services have failed to match the Web's growth.

Mobile environments present even more unique challenges for service discovery due to the intrinsic limitations of wireless network technologies and the limited resources of mobile devices, despite the advanced features and capabilities

* Corresponding author. Tel.: +1 6135314897.
E-mail addresses: elgazzar@cs.queensu.ca, k.a.elgazzar@gmail.com (K. Elgazzar), hossam@cs.queensu.ca (H.S. Hassanein), martin@cs.queensu.ca (P. Martin).

1574-1192/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.pmce.2013.10.015
of the new generations of smartphones and high-end mobile devices (e.g. laptops and tablets). Researchers over the past few years have focused on optimizing specific aspects of current Web service discovery approaches in isolation [2,1,3] or overcoming individual limitations (such as intermittent connectivity) [4,5] to fit the inherent constraints and dynamic context of mobile domains. However, the lack of a comprehensive understanding of both user context, and the various constraints of mobile environments, renders most of these approaches incapable of efficient and reliable discovery in mobile scenarios.

From another perspective, Web service discovery is commonly recognized as a resource-intensive process [6], which contradicts the resource limitations of mobile devices. For example, semantic service discovery approaches perform matching at the semantic level, which better understands the semantic of Web service functionalities and non-functional parameters. Therefore, semantic approaches go beyond the syntax level and offer better discovery results by successfully retrieving all relevant services [7–9]. However, semantic approaches add significantly to the resource requirements. As such, cloud computing is candidate to bridge the gap between resource-constrained environment and resource-intensive Web service discovery. It opens up new opportunities for mobile devices to efficiently perform service discovery, while substantially reducing their resource consumption. Cloud computing not only bootstraps the performance of service discovery in mobile environments, but also removes development constraints by expanding the horizon with more options to apply sophisticated techniques that might potentially result in better service discovery.

While significant research has focused on service discovery protocols to address constraints stemming from mobile environments, they mostly lack the capacity to holistically address the different limitations and user needs. This paper introduces Daas (i.e. Discovery as a service), a holistic discovery framework that addresses various aspects of efficient context-aware mobile Web service discovery.

The contributions of this paper are summarized as follows:

• We provide a comprehensive requirements analysis for mobile Web service discovery in resource-constrained environments.
• We introduce the concept of “Discovery as a service (Daas)”, i.e. Web service discovery as a cloud-based service. We demonstrate the viability of our framework with a use case and a proof-of-a-concept implementation.
• We integrate user preferences and context into service discovery to find services that best fit the user needs.
• We present analytical models to calculate the relevance of candidate services to a particular aggregated context.

The remainder of this paper is organized as follows. Section 2 presents a motivating scenario. Section 3 outlines related research. Section 4 discusses the current discovery approaches, points out the limitations, determines the essential requirements for efficient discovery in mobile environments, and identifies how the cloud can bootstrap service discovery. Section 5 describes the proposed framework and relevant research efforts that may be potentially applied for each component. The framework functionality is validated in Section 6 and a performance evaluation is presented in Section 7. Finally, Section 8 draws the concluding remarks of the paper.

2. Motivating scenario

Adam is visiting France on a vacation. Adam’s first language is English and he has a basic knowledge of French but cannot effectively communicate in French. While traveling, Adam prefers to pay using his credit cards to reduce the amount of foreign currency he has to carry. He follows a strict diet that limits his options. Adam spends most of his day outside visiting tourist attractions. He uses his 4G-connected smartphone for guidance, itinerary optimization, and searching for services such as attraction recommendations, restaurants, and currency exchange. Although Adam is connected to the Internet through his roaming plan, he sets his smartphone to connect to free Wi-Fi spots whenever applicable to reduce the cost and take advantage of the higher bandwidth WiFi offers.

During lunch time, Adam searches for restaurants in his vicinity. Adam wishes to find a restaurant that provides food that meets his dietary regulations and offers flexible payment options, English language communication, and complementary Wi-Fi access. Adam typically plans the remainder of his day while waiting for his meal to be served by exploring services that offer tourist recommendations that include photo snapshots, video trailers, and visitor feedback.

In such a scenario, Adam faces several challenges finding services that know about his context and can accommodate his preferences. For example, searching for a restaurant with traditional Web service discovery approaches may recommend a list that includes restaurants that are not within walking distance and perhaps others that do not offer service in English or flexible payment options. Additionally, such approaches provide no priorities to services that accommodate Adam’s preferences such as dietary constraints. For example, he may prefer to satisfy his dietary constraints over proximity. Existing service discovery approaches lack the ability to support such a level of convenience. Daas offers suitable service discovery in such a scenario through integrating the user context and preferences with the discovery process.

3. Related work

Service discovery spans multiple levels ranging from infrastructure service discovery (e.g. physical resources such as printers) to application-level service discovery, where end-users search for network functions that satisfy their objectives. The context is an inter-layer concept integrated in all discovery levels [1]. The framework presented in this paper is concerned with service discovery at the application level.
In mobile environments, dynamic context and resource constraints are key challenges to Web service discovery. Context information can help service discovery to find services that best fit a particular context [2]. In addition, the association of mobile devices with a particular user makes it possible to personalize services and applications according to user needs. However, Web service description standards lack robust models that can understand user preferences and context [10,11]. Garcia et al. [10] propose an abstract expressive model for defining user preferences that can be used to extend semantic Web service descriptions. This extension would enable Web service discovery mechanisms to discover and rank Web services that better match user preferences. From the same perspective, Yu et al. [12] are concerned with extending the semantic service profile with context information that can be used for better service matching. The focus of these research proposals is on context representation not on improving service discovery using user context, which our research pays more attention to.

Context-aware service discovery can benefit from various context information such as location, device profile, and environment parameters [13]. Location information can be used in selecting location-based services [14–16], or for maintaining service offerings based on their location [17]. Device profile (i.e. features) is important in mobile service discovery, since mobile users discover and consume services from their mobile devices. Discovery mechanisms need to be aware of the device capabilities in order to find services that, in addition to satisfying the user objective, function properly within the device constraints. MobiEureka [18] addresses this aspect with a device-aware discovery mechanism that integrates the device capability into mobile Web service discovery. MobiEureka ranks discovered services according to their fit to the device features. In contrast with MobiEureka, the work presented in this paper integrates multiple types of context information to find services that fit such context collectively and improve the quality of service discovery.

The other major challenge of service discovery in mobile environments is the resource constraints. Researchers over the past few years have addressed this challenge in two different directions. One direction aims at optimizing specific aspects of current Web service discovery approaches to overcome the inherent constraints of mobile domains [19,6,20,21]. The other direction relies on cloud computing to bridge the gap between limited resources of mobile environments and resource requirements [22,23]. Since computing resources in a cloud are allocated to customers through virtual machines (VMs) with determined configurations and deploying these VMs incurs implicit latency, time-sensitive applications face a real challenge with this approach. To avoid such latency, a physically nearby computing infrastructure (cloudlet) [24] could be used instead of a public cloud. However, the cloudlet approach is still in its infancy, facing challenges on how to find close to computing resources and manage running instances. To this end, the DaaS framework takes advantage of the cloud to execute resource-intensive discovery processes and manage system scalability.

Parallel to these efforts, Mobile Cloud (mCloud) is emerging to overcome the inherent limitations of mobile devices [25,26]. To date, mCloud is mostly used to refer to offloading computational intensive processing from resource-constrained devices to a more powerful computing infrastructure. This leverages the emergence of a new generation of mobile applications and services that are not bound by the limitations of mobile devices. For example, Giurgiu et al. [22] present a middleware that can distribute mobile applications between the mobile device and a remote server, aiming at improving the overall latency and reducing the amount of data transfer. Similarly, CloneCloud [23] offers a runtime partitioning approach for mobile applications based on a combination of static analysis and dynamic profiling techniques. MAUI [27] enables energy-aware offloading in order to reduce the burden on the limited energy resources of mobile devices while accommodating the increasing energy demands of mobile applications and services.

Although there have been many research contributions on the service discovery, little research has been conducted addressing service discovery that fits user interests and context. The research presented here builds on our previous efforts in Web service discovery for mobile environments [28]. The DaaS framework serves as the foundation of a whole new approach in finding Web services that best fit the user preferences and context, while shifting the burden of computations onto the cloud that can also support high scalability.

4. Service discovery approaches

Service discovery is the act of finding a relevant service for a particular request. Most of the existing techniques belong to one of three main discovery approaches [29]: UDDI Registry, specialized search engines, and generic search engines. Each one of these approaches has its own strengths and weaknesses.

- **UDDI**: Universal Description Discovery and Integration (UDDI) is the discovery approach used by the standard Web service architecture. It relies on centralized repositories that providers use to publish their services and customers use to discover services that satisfy their requirements. UDDI maintains information about the service description, publisher, endpoint, technical interface (tModel), implementation, etc. This approach has not been widely adopted by the Web services community which explains why major UDDI repositories (such as IBM and Microsoft) shut down their services in 2006 [30]. However, there are still a few public registries offering services with different capabilities such as RemoteMethods, Striklr, and X-Methods.

Problems with UDDI include the centralized architecture, limited scalability, single point of failure, consistency maintenance, searches that rely on keywords or category browsing, lack of support for non-functional and behavioral aspects, and outdated service records. Customers also need to be aware of the UDDI addresses to locate and query the repositories. However, UDDI enables service subscription for interested users to keep them updated. Some UDDI repositories add extra features such as service trial, transaction facilitation, WSDL parser, different pricing schemes, performance
monitoring, programmatic interface, ratings, categorization, documentation, etc. UDDI also allows searching for providers and tModels [30,29].

- **Specialized search engines**: This approach aims to distinguish a Web service search from a Web content search. The basic idea is to make use of Web services' functionalities, operations, and other information provided in the description files in order to perform a meaningful search for services that best match a particular request. These search engines collect Web services description files from public UDDI repositories and Web contents, extract the semantic meaning of these Web services from their description files, and perform semantic matching between requests and Web services capabilities. Woogle [31] and WSCE [32] are examples of these search engines.

Web services search engines are able to find services that are more relevant to users' requests as the search does not only rely on keywords but also on functionalities and other running parameters such as QoS. Additionally, the retrieved services should be valid and running, as these engines are able to catch any updates or status changes while continuously crawling the descriptions of Web services from the source. However, so far this approach supports only searching for non-semantic Web services.

- **Generic Web search engines**: Web content search engines are another alternative to find Web services using keyword search. Major providers of Web services, such as Google, Amazon, and Yahoo, have decided to publish their Web services through their own websites instead of using UDDI Business Registries (UBRs). This trend is forcing users to discover Web services through Web content search engines. Users can use search engines to locate Web services by customizing the search query to look for specific file types (ex. wsd and owl files).

The major drawback of generic search engines is that they cannot understand Web service functionalities outlined in the description files and only rely on keywords to find services. Their advantages include robustness, scalability, and no extra infrastructure is required.

4.1. Limitations of current discovery mechanisms: a mobile perspective

Notwithstanding the research efforts that have focused on Web service discovery, many limitations with respect to mobile environments remain, including the following:

- Current approaches lack agile architectures that can guarantee robustness and scalability.
- User experience and satisfaction, user preferences, and device features and capabilities are important factors that must be incorporated into Web service discovery in mobile environments. A few proposals provide limited consideration to these factors. They tend to focus on either user preferences or mobile capabilities and ignore environment context and the network status.
- Current approaches either do not cater to resource-constrained providers that may exist in mobile domains or address certain limitations in isolation, such as limited computational capacity [6] or intermittent connectivity [33]. Such constraints pose challenges to perform semantic matchmaking efficiently, for instance. Successful resource-aware discovery architectures must offer accommodation to the various constraints of mobile domains.
- In wireless networks, providers and customers communicate over wireless channels where signal quality is variable. Existing discovery mechanisms are unable to identify Web services that are able to promptly respond and appropriately adapt to such context change.
- Proximity-based service discovery approaches are able to capture the geographical scope to which services might be bound [34,35]. However, in mobile service discovery, proximity extends beyond just the geographical location. For example, selecting services that belong to the same home network may significantly reduce network traffic, cost, and perhaps guarantees a higher quality of service. Identifying services that provide such selection priority is not currently supported by existing approaches.
- Keyword-based service discovery works on the syntactic level and, is ineffective in retrieving the most relevant services to a specific request, while semantic-based discovery is resource-intensive and not affordable by resource-constrained providers. The gap between accuracy and resource demands can be approached either by developing lightweight semantic reasoners that require less resources while achieving reasonable results, or by offloading resource-intensive processes to a resource-rich computing infrastructure. A hybrid approach is also possible. With the widespread adoption of cloud computing, offloading seems to be more realistic to bootstrap service discovery.

4.2. Requirement analysis of mobile web service discovery

Service discovery is a crucial component in Web services, especially in heterogeneous mobile environments. The process must be efficient and rapid to cope with the extremely dynamic nature of mobile domains. To ensure that service discovery in mobile domains is properly functioning, it must satisfy the following requirements:

1. Mobile devices frequently change their point of attachment to the network or engage in a vertical handover across different wireless access technologies. In mobile service provisioning scenarios, where service providers could be mobile, services may become stale or inaccessible due to changes in binding information. Therefore, efficient service discovery should fulfill two fundamental requirements. First, it should ensure that the discovered services are active and not
outdated. Second, the discovery process should remain running to support seamless provisioning, in case the selected service become unreachable while executing. In such a case, the service discovery either searches for an alternative access to the same service (e.g. uses mobile networks if a WiFi connection is lost) or re-selects the next candidate from the retrieved services.

2. With the existing diversity of mobile device form factors, capabilities and mobile platforms, and the limitations of wireless networks, devices may support limited operations from those currently offered by Web services. Discovery mechanisms should ensure that discovered (or selected) services are compatible with the device capabilities. In addition, service discovery schemes should be able to assign higher priority to services with multiple capacities that can adapt their behavior to fit certain constraints (e.g. Bandwidth and transmission rate), yet perform efficiently.

3. Location-based services are common and popular in mobile service scenarios, since users are typically on the move. Service providers and service consumers may offer or request location-based services, respectively. In such scenarios, robust service discovery should cater for the demand of service consumers by supporting location-dependent discovery upon request.

4. Typically, mobile devices are associated with context information and users who usually have preferences. One of the chief benefits of mobile services is the enabling of personalized services provisioning that takes into account the user preferences and context information. Service discovery in such cases must incorporate the user profile and context information in ranking and selecting Web services.

5. Semantic Web service discovery approaches present a significant challenge in mobile domains, despite the fact that they typically yield better results. The discovery of semantic Web services requires a heavyweight matchmaking process due to semantic reasoning, which cannot be supported by resource-constrained devices. Therefore, efficient discovery mechanisms require either highly optimized semantic reasoners to fit mobile capabilities or a way to offload resource-intensive tasks to resource-rich environments (such as the cloud).

6. Service composition is another alternative to fulfill a service request if no atomic service match exists. Service discovery schemes should be able to break down the request into primitive sub-tasks, if possible, and fulfill each sub-task separately before integrating them together to satisfy the original objective. However, service composition adds another level of burden on resource-constrained devices.

4.3. What can the cloud offer?

Cloud computing removes the boundaries of computing resources. Cloud computing features elastic resource management, where computing resources can scale up or shrink according to an application’s needs [36]. As such, cloud computing is able to augment the limited resources of mobile environments by offering remote access to computing resources over the network. The benefits that cloud computing may bring to mobile Web service discovery are summarized as follows.

- From the computational resources perspective, the cloud can bridge the gap between mobile devices and mobile applications that require resources beyond what mobile devices can afford.
- Offloading the resource-intensive processes of the service discovery onto the cloud enhances the overall response time and reduces battery consumption.
- Cloud computing, by design, promises to improve service availability, reliability, and quality of service.
- The cloud supports elastic resource provisioning, where resources can be scaled up and down efficiently on demand according to applications’ dynamic requirements.
- The cloud enables mobile Web service discovery to apply advanced discovery algorithms (such semantic matchmaking), while maintaining reasonable response time.

Taking advantage of the cloud, mobile Web service discovery can be provided as a cloud-based service. From the user’s perspective, DaaS can look like a native mobile application similar to other cloud-based mobile applications such as Google Gmail for Android or iPhone.

Fig. 1 illustrates an abstract architecture of DaaS. Customers submit service requests to the cloud, which handles these requests and finds relevant Web services that fulfill the request. Providers may inform the cloud of their valid Web service description endpoints, for immediate consideration. The cloud in turn creates local corresponding symbolic links for these Web service description endpoints in order to be used for discovery purposes. DaaS also clusters these services into functionally similar groups to reduce the search space and help efficient discovery.

5. DaaS: cloud-based service discovery framework

Based on the limitations of existing discovery mechanisms and the requirement analysis of successful mobile Web services discovery presented above, we propose a cloud-based framework for mobile Web service discovery for resource-constrained and mobile environments. DaaS has two main objectives: (1) pushing the resource-intensive processes of service discovery to the cloud in order to save the already scarce mobile resources, (2) incorporating the user preferences and context in mobile Web service discovery.
Fig. 1. WS-Discovery as a service.

Fig. 2 illustrates the essential DaaS components to perform effective Web service discovery in mobile environments. Components are depicted in layers to indicate where components are to be deployed. The customer layer encompasses the components that reside on the customer side, and the provider layer contains the components that run at the provider side. The cloud layer shows the components that run on the cloud, where resource availability and power consumption are not a concern.

To overcome the challenges that stem from the UDDI approach (especially service registration and de-registration), DaaS requires providers to advertise services they offer in a local service directory. Then, a Web service search engine [31,32] crawls services offered by different providers and maintains a cloud-based service repository. Web services are further classified into functionally related clusters to reduce the response time and achieve efficient matching [37].

5.1. Customer layer

5.1.1. Service request

Service Request is a multimodal mobile user interface that enables end-users to submit Web service requests that express their specific objectives. The request is submitted in a plain text that describes the user’s objective. Mobile users usually have limited input capabilities, which are unsuitable for a formatted service request or a formal service description language (i.e. in semantic services). Therefore, plain text fits well for the service request within mobile device constraints. Further analysis of the user input is done by the Service Request Handler to extract keywords and meaningful information. These keywords are used in keyword-based matching. In case of semantic matching, they can be used to construct a formatted service request.

5.1.2. Context manager

The Context Manager is responsible for collecting user environment context using mobile device embedded sensors and capabilities. It also handles the user preferences. Such context information is used to rank relevant Web services according to their best fit into such context. Context Manager also monitors available networks and their status.

5.2. Cloud layer

This layer contains the components that are deployed on the cloud where computational resources and power consumption are of less concern.

5.2.1. Request handler

The Request Handler performs advanced processing operations on the user request. It extracts keywords from the user request using text analytics techniques. These keywords are used to perform keyword-based service matching, or format the service request to match the different possible service description languages (WSDL, OWL-S, WSMO, WSDL-S, etc.), according to the available services in the cloud service repository. Since this component is deployed on the cloud, semantic reasoning, which represents a big challenge if performed on resource constrained environments, can be accommodated in the cloud layer. Service discovery, therefore, can benefit from the full potential that semantic reasoning and ontology-based matchmaking can bring to the discovery process.

5.2.2. Request analyzer

The Request Analyzer mainly gets involved when no atomic services are found satisfying the request objective. In other words, DaaS presents two levels of service discovery. Level 1: The service request is matched first with atomic services that totally satisfy the user’s objective. Level 2: If no relevant services are found, the Request Analyzer breaks down the request into primitive subtasks \( ST = \{ st_1, st_2, \ldots, st_n \} \), if possible, and satisfies each subtask separately. Breaking down the request (in collaboration with the Service Composer) may follow one of two approaches: (1) Start with the request inputs to get the request outputs with exact or partial matches [38], (2) Find services that have exact matches with subtasks, i.e. the matchmaking algorithm yields an EXACT or SUBSUMES match between the subtask inputs/outputs and the relevant service inputs/outputs. EXACT match means that all the subtask inputs and outputs match exactly with the service input and output parameters. A SUBSUMES match means that all the subtask inputs and outputs are subsets of the service input and output parameters, respectively.
5.2.3. Service composer

The Service Composer in DaaS is responsible for the orchestration process and generating composition plans for atomic Web services that satisfy the individual primitive subtasks, and together fulfill the original service request. Resolving users' requests via composite services (when applicable) meets requirement #6. Bhuveswari et al. [38] propose a framework for semantic Web service composition in mobile environments. Their framework converts WSDL files into OWL-S specifications and generates a service profile for the request. Then, it performs a semantic reasoning between the advertised service profile and the request service profile. The composer then generates composition plans and stores them in a plan repository in the cloud.

5.2.4. Search/matchmaking

The Search/Matchmaking module matches the functionalities described in the request and the capabilities offered by Web services. For non-semantic Web services, i.e., described by WSDL files, the matching between the request and Web services is keyword-based and uses information retrieval techniques [39]. In this case, the Web services are characterized by sets of keywords extracted from the description files. These keywords are used to index Web services and later matched with keywords extracted from the user request [40]. Semantically described Web services are discovered using high level matchmaking approaches. The most popular semantic discovery methods are OWL-S based and WSMO based.
approaches, which use the information provided in the service profile and domain ontologies to match the user’s requested functionalities [41].

Conducting the reasoning process on a resource-constrained device might possibly fail or produce out-of-memory/stack overflow errors due to insufficient resources. Although researchers have made proposals to address these limitations, DaaS employs the cloud to remove these boundaries and opens the possibilities of performing the matchmaking process efficiently. This opportunity meets requirement #5 for effective discovery mentioned earlier. Algorithm 1 illustrates an abstract process flow of a request matching in DaaS. The functions Match, Split, and Compose are out of the scope of this research.

Algorithm 1: Request Matching Algorithm

| Input: Service request SR |
| Output: a set of relevant services S |
| Function ReqMatch(SR) |
| 1 //search for atomic services first, |
| 2 //trying to fulfill the whole request with a single service |
| 3 S = Match(SR) |
| 4 if S isNull then |
| 5 //split the request into primitive objectives |
| 6 //subtasks ST |
| 7 ST = Split(SR) |
| 8 //match each subtask separately |
| 9 //collect services in an intermediate container |
| 10 Initialize S_intr |
| 11 foreach st in ST do |
| 12 S_intr = S_intr + Match(st) |
| 13 end |
| 14 if S_intr isNull then |
| 15 //generate & evaluate composition plans |
| 16 S = Compose(S_intr) |
| 17 end |
| 18 end |
| 19 return S |

5.2.5. Relevant services list

This is the list of services that match a user request. The list contains either atomic services or composite services that satisfy the user objectives. Services in this list are ranked based on their absolute relevancy to the user objectives. Although this list contains services that are relevant to the user request, some of them might not satisfy the user preferences or fit current context. Therefore, DaaS matches these services further against user preferences and context in order to improve the user experience.

5.2.6. Context management

DaaS uses four types of context information to rank the list of relevant Web services, user preferences, device profile, environment context, and user ratings. Incorporating this context information in mobile service discovery fulfills requirements #2, 3, and 4. Therefore, we provide more details on this aspect below.

User preferences: User preferences are not part of the Web service descriptions, which implies that retrieved services may satisfy a user’s request, but may not be quite relevant to the user’s preferences (requirement #4). User preferences are typically descriptive and strongly tied with the situation or service functionality. For example a customer might say “I prefer, or “require”, restaurants that accept credit card payments”. User preferences, however, lack a highly expressive model that can fit well and be adopted in Web service description and discovery. Towards this end, DaaS requires users to express their preferences in standalone files (.usr). This file remains on the user’s mobile device. Users can express their preferences using an interactive interface that implements an approach similar to the one proposed by García et al. [10].

Device profile: The objective of incorporating the device profile information with the Web service discovery process is to ensure the compatibility of the discovered/selected service with the device constraints. In this regard, Al-Masri et al. [18], develop a device-aware service discovery mechanism that is capable of selecting Web services that function properly within the mobile device constraints. The mechanism takes advantage of HTTP sessions to collect device information and store it at the server side. This information is compared later with the requirements that are set by the providers to rank services according to their fit to the device constraints (requirement #2). The solution is limited to services that are with the authors’ proposed extension, “WSDL-M”, to the standard WSDL definition [42] in order to include
the required device-specific features. DaaS adopts the approach of describing the “required” and recommended “to-have” features in mobile devices in a separate file (.mbl) for the same reasons mentioned earlier. Table 1 contains a non-exclusive list of sample attributes that might characterize the mobile device behavior [43].

Environment Context: In wireless networks, the environment context may change frequently during the execution of the Web service as mobile consumers and/or providers may be on the move. Such changes might substantially impact the performance of service provisioning in mobile environments, in contrast with traditional Web services provisioning. The objective of integrating the environment context into service discovery is to ensure that services run properly given these environment conditions/parameters. DaaS requires providers to identify the minimum and preferred values of parameters for reliable operation in a standalone file (.env). This file is to be associated with the other service description files. While at the customer side, the Context Manager collects all the environment information to better assess the customer’s environment status. The Service Ranking module employs the information in this file and that collected at the customer side to rank the relevant services. Accordingly, the discovery process may realize that the requester is able to execute limited functionalities of particular services and so rank them accordingly (requirement #2).

Exploiting environment context also opens the opportunity of supporting services of multiple capacity, which are services that can adapt their behavior to the current environment. In such a case, a discovery mechanism assigns higher priority to services with multiple capacities (requirement #2). Consequently, the service execution platform/architecture should support the appropriate mechanisms for assessing the environment status at runtime so that services with multiple capacities would be able to promptly respond to the environment changes. Environment context also includes location information, which can be used to enhance the service provisioning according to the user’s location (requirement #3). For example, providing a list of restaurants that are located nearby the requester. Table 2 lists sample environment contextual attributes that could of concern to the proper functioning of mobile Web services.

Maamar et al. [44] discuss the development of capacity-driven Web services starting from description, discovery, and composition, to the invocation of desired functions. Similar research on services with different facilities to cope with environment context is presented in [45]. The authors named their approach “Service Differentiation”, which aims to develop/provide a single service with multiple variations instead of several independent services.

User ratings: In Web 2.0 and other open environments, users are encouraged to provide feedback and rate different services they have used based on their experience. User ratings reflect a user’s level of satisfaction and perceived quality of service(QoS). These ratings can be used as an effective tool to show the user-perceived quality of service operation. DaaS exploits user ratings to rank Web services, assuming that proper handling mechanisms for ratings, complaints, and leaving comments and feedback are applied.

5.2.7. Service ranking

The service ranking component receives a set of relevant Web services \( S = \{s_1, s_2, \ldots, s_i\} \) and ranks them based on the following four various types of context domains: user preferences \( P = \{p_1, p_2, \ldots, p_i\} \), device profile \( D = \{d_1, d_2, \ldots, d_j\} \), environment context \( E = \{e_1, e_2, \ldots, e_k\} \), and user ratings \( R_{s_i} \), where \( R_{s_i} \) represents the normalized average rating for a service \( s_i \). The user preferences are determined by the customer where device profile and required environment parameters are set by the service provider to ensure the proper operation of services. User ratings are collected by service providers through dedicated tools to realize the user perceived experience.

To show how ranking works, suppose that the matchmaking retrieves a set of relevant services \( S \) to a service request \( SR \). These services perform similar functions, but vary in their requirements for proper execution. During the request communication session, the discovery mechanism collects the features of the customer’s device \( Dc = \{dc_1, dc_2, \ldots, dc_j\} \) corresponding to the required device profile \( D \). Simultaneously, the device senses various ambient conditions to better assess

<table>
<thead>
<tr>
<th>#</th>
<th>Device features</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Browser (name, vendor, version, ...)</td>
</tr>
<tr>
<td>2</td>
<td>SoftwarePlatform (vendor, model, ...)</td>
</tr>
<tr>
<td>3</td>
<td>ScreenSize (width, length)</td>
</tr>
<tr>
<td>4</td>
<td>DisplayChar (width, height)</td>
</tr>
<tr>
<td>5</td>
<td>InputType</td>
</tr>
<tr>
<td>6</td>
<td>SupportsColor</td>
</tr>
<tr>
<td>7</td>
<td>SupportsCallback</td>
</tr>
<tr>
<td>8</td>
<td>SupportsCSS</td>
</tr>
<tr>
<td>9</td>
<td>NFCSupport</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#</th>
<th>Environment contextual information</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Network information Signal strength</td>
</tr>
<tr>
<td>2</td>
<td>Ambient attributes Temperature Humidity</td>
</tr>
<tr>
<td>3</td>
<td>Proximity Location Nearby objects Landmarks</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#</th>
<th>Parameter Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Network information</td>
</tr>
<tr>
<td>2</td>
<td>Ambient attributes</td>
</tr>
<tr>
<td>3</td>
<td>Proximity</td>
</tr>
</tbody>
</table>

K. Elgazzar et al. / Pervasive and Mobile Computing 13 (2014) 67–84
the environment status $Ec = \{ec_1, ec_2, \ldots, ec_k\}$. For each service $s_i$, there are features $ps$ corresponding to user preferences $P$ as expressed by the matrix in Eq. (1). These features can be extracted from the service description files.

$$
S_p = \begin{bmatrix}
p_{s_{1,1}} & p_{s_{1,2}} & \cdots & p_{s_{1,s}} \\
p_{s_{2,1}} & p_{s_{2,2}} & \cdots & p_{s_{2,s}} \\
\vdots & \vdots & \ddots & \vdots \\
p_{s_{l,1}} & p_{s_{l,2}} & \cdots & p_{s_{l,s}}
\end{bmatrix}
$$

where rows represent the set of service features for service $s_l$ that are related to the user preferences and columns represent a single feature across the entire set of relevant services. A similar matrix can be obtained for the other three context domains, $D$, $E$, and $R$.

The rank of each service $s_l$ then is represented by the following formula in Eq. (2).

$$
\text{Rank}_{s_l} = \sum_i w_i \times f(p_i, ps_i) + \sum_j w_j \times f(d_j, dc_j) + \sum_k w_k \times f(e_k, ec_k) + R_{s_l}
$$

where $w$ represents the weight of each corresponding feature. This weight indicates the level of importance of such a feature to either the service customer or the service provider. The function $f$ computes the relation between two objects as shown in Eq. (3).

$$
f(a, b) = \begin{cases} 
    b & \text{if } a, b \text{ are numbers} \\
    \frac{a + b}{\text{sim}(a, b)} & \text{if } a, b \text{ are strings}
\end{cases}
$$

where $\text{sim}(a, b)$ is a featureless similarity factor computed between objects $a$ and $b$ using Normalized Google Distance (NGD) [46]. $\text{sim}(a, b)$ is calculated by Eq. (4).

$$
\text{sim}(a, b) = 1 - \text{NGD}(a, b)
$$

$f(a, b)$ yields values $\in [0 \ldots 1]$. In the case where $a$ and $b$ are numbers, values greater than or equal 0.5 means that $b$ satisfies $a$. The closer the value to 1, the greater the satisfaction that condition $b$ achieves relative to the condition $a$. In the case where $a$ and $b$ are text (keywords), the function value close to 1 indicates that the terms $a$ and $b$ are semantically related, where values close to 0 indicates that they are not related. Algorithm 2 symbolically explains how ranking works.

---

**Algorithm 2**: Ranking Algorithm

**Input**: set of relevant services $S$,

various context $P$, $D$, $E$, and $R$,

and corresponding weights $W$

**Output**: ranked list of services $\text{RankS}$

1. **Function** $\text{Rank}(S, P, D, E, R)$
2. Initialize $\text{RankS}$
3. // calculate the rank of each $s \in S$
4. // based on all context domains, i.e. $P, D, E, R$, Eq. (2)
5. **foreach** $s$ in $S$ **do**
6.   $\text{Rank}_s = \text{Calculate Eq. (2)}$
7.   $\text{RankS} = \text{RankS} + \text{Rank}_s$
8. **end**
9. //sort the results
10. $\text{Sort}(\text{RankS})$
11. return $\text{RankS}$

---

5.2.8. Ranked services list

The ranked services list is the final set of services that match the user request and best fit the user preferences and current context. This list is typically shorter than the retrieved service list and services are re-ranked according to their aggregate relevancy to the user preferences and various context. Daas presents this list to the service requester in order to choose the proper service to invoke.

5.2.9. Relevant services cache

Mobile environments are characterized by intermittent connections, unreliable channels, and high transmission error rates. Services may become unavailable or function improperly due to the lack of resources (ex. bandwidth). In such cases, service discovery mechanisms need to support alternatives either by providing different access paths to the same service or finding functionally similar services (which satisfies requirement #1). From this perspective, Daas caches the retrieved
relevant services list for a specific user request in order to choose the next best candidate service in the case that the primary service fails to respond or perform adequately. This cache is cleared once the desired service is successfully executed.

The service cache remains with the service execution environment, depending on what mobile service architecture is adopted. If, for example, the service execution environment is deployed at the customer side, then it makes sense to have the service cache located at the customer side to be used in case the current service, for any reason, is disrupted. However, if the cloud manages the service provisioning as well as the service discovery, then the cloud (where the service discovery has been performed) is the right place for the service cache.

5.2.10. Service engine

The main objective of this component is to collect Web service references and description documents from all providers and maintain an up-to-date cloud-based service repository. Crawling Web service information from their original location is a continuous process similar to the Web content crawling. This means that services that become invalid or for any reason become unreachable are automatically removed or marked as inactive. Web service engines therefore serve as a reliable source of Web services.

While the service engine crawls Web services automatically at their providers' side, DaaS gives providers the ability to notify the search engine with their new offerings for the first time. This feature aims to make services known immediately as they become available to cope with the dynamic nature of the mobile environment. For example, suppose that someone decides, in an ad-hoc fashion, to offer a video streaming service for a football match s/he is attending. This Web service is of interest only while the match is playing and invalid otherwise. In such a case, the ultimate objective of the provider is to advertise this services immediately when it becomes available. By sending a notification with the Web service URI to the service search engine, the engine crawls and indexes the service immediately and adds it to the list of available services.

5.3. Web service clustering

This component clusters Web services into functionally similar groups based on their descriptions. Therefore, the Service Matching does not need to match the user request against all the service offerings in the corpus, but rather with a particular set of services that share similar functionality. The clustering and classification of new services are performed offline to eliminate any overhead during service matching [37]. The service clustering renders centralized repositories performing similar to distributed approaches in P2P architectures [47], where each domain-specific services are maintained and matched separately. Our clustering approach not only reduces the response latency by reducing the search space, but also improves the recall and precision via data mining and text analytic techniques. In addition, it takes advantage of the continuous crawling of service engines by enabling adaptive re-clustering and self organization in order to cope with the dynamicity of Web services.

Service clustering is carried out through multiple stages starting with extracting features from WSDL documents to calculate similarity between Web services and cluster them into functionally similar groups. Feature extraction includes parsing WSDL documents, tag removal, stemming words, removal of function words, and recognizing context words that contribute to the semantics of the Web service. We also extract the function names, parameter names and data types, message structures, and service name. These features are integrated to measure the integrated similarity factor between Web services. We use NGD [46] as a featureless distance measure between words. For more details about our clustering technique and its performance evaluation, the reader is referred to our previous research [37].

5.4. Provider layer

This layer contains the DaaS components that are implemented at the provider side. In mobile service scenarios, service providers could be a mobile entity with limited resources. DaaS removes the burden on such resource constrained providers by shifting resource-intensive processes to the cloud, keeping only necessary components on mobile elements. Therefore, in our framework the only component that providers need to run using local resources is the module responsible of announcing Web service availability.

5.4.1. Service advertiser

This component announces the existence of Web services in a local service directory. Publishing Web services in a local service directory is similar to publishing services in an enterprise UDDI but on an individual scale. The direct benefit of this approach is putting service providers in full control of their offerings. This also eliminates the burden of service registration and de-registration that service providers are required to do with the UDDI approach. It also makes it easier to keep local service repositories up-to-date and self-maintained.

5.4.2. Local services directory

Service providers maintain a local directory that contains all Web services they offer. DaaS adopts a distributed service directory approach, where each mobile device manages its own offered Web services and maintains references to services it knows about. From this perspective, there are two categories of services: local services and remote services. Local services are
Table 3
The structure of DaaaS service directory.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Boolean</td>
<td>&quot;0&quot; for local services, &quot;1&quot; for Remote services</td>
</tr>
<tr>
<td>ProviderEndPoint</td>
<td>String</td>
<td>The provider’s base Internet address</td>
</tr>
<tr>
<td>sID</td>
<td>Int</td>
<td>Service id</td>
</tr>
<tr>
<td>Title</td>
<td>String</td>
<td>Service title</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>A reference to the service description file</td>
</tr>
<tr>
<td>ServiceEndPoint</td>
<td>String</td>
<td></td>
</tr>
</tbody>
</table>
| Status          | Int      | 0 = “active” (default), 1 = “inactive”                                       

hosted and provided by the local system, whereas remote services are “active and running” services hosted on other mobile devices. Handling remote services requires a coordination protocol to manage link updates, advertisement notifications, invalid service/link removal, and duplicate reference avoidance. JXTA protocols [48] are commonly used in such scenarios in P2P networks.

The service directory, in DaaaS, represents the service descriptors, which are parameters that can express the service functionalities, such as (service title, text description, location or URI of a machine-readable description file, service URI, etc.). These parameters are used to discover services relevant to a particular user objective. Table 3 shows a possible table structure for the service directory.

The providerEndPoint identifies the base Internet address at which the service provider can be reached, where service search engines can get access to the service description documents.

In order to avoid false discovery when a service is temporarily suspended (on purpose) or becomes invalid (no longer offered), the provider sends an update message with the service ID to either deactivate or delete the service, respectively. The DaaaS service directory supports service deactivation in the case that providers want to temporarily suspend their services with a high possibility that these service will become active again later.

6. Experimental validation

To validate the functionality of DaaaS, we developed a Web service discovery prototype on Amazon EC2. We carried out a number of experiments to validate DaaaS operation in order to ensure that it functions as expected. The prototype consists of two parts: customer-side and cloud. The customer side portion is a mobile-based interface, and it handles service requests and presents results as shown in Fig. 3. This portion is deployed on a Samsung Galaxy II smartphone (Dual-core 1.2 GHz Cortex-A9, 1 GB RAM, Super AMOLED Plus 480 × 800 pixels display, 4.3 inches) with a rooted Android 4.0.4 platform [49], connected to a WiFi network. The cloud portion is deployed on Amazon EC2 and it implements most of the DaaaS components and handles service discovery processing including service ranking. On Amazon EC2, we created a number of instances of the type ‘t1.large’ with an EC2 pre-configured image (AMI) of ‘Ubuntu Server 12.04 LTS, 64 bit’ to host the cloud portion of the prototype. The prototype begins with a single instance and the cloud ‘autoscale’ feature activates other instances as needed.

Experiments are conducted with 1200 WSDL documents of (valid) online Web services that serve various domains, such as ‘currency exchange’, ‘weather forecast’, ‘address validation’, ‘e-mail verification’, and ‘credit card services’. These service description files are collected from real world Web service providers and online Web service directories, such as WebserviceList, WebserviceX, and xMethods. We have manually inspected the dataset and identified a set of 18 Web services which belong to the domain of ‘currency exchange’ as shown in Table 4. This set shows the expected response of a Web service discovery request for ‘currency exchange’ services. The experiments are not intended to evaluate the precision and recall of the information retrieval technique, but rather the experiments focus on the ranking aspect and adaptation to changes in context information.

For each of these services, three context information files are generated with random values for the context parameters: serviceName.env, serviceName.mbl, and serviceName.usr, where serviceName is the name of each corresponding Web service. In the user profile, we added a set of preference parameters such as payment method, food, restaurants, shopping places, travel interest, etc. Listing 1 illustrates an excerpt of XigniteCurrencies.env. The ‘type’ of the bandwidth element indicates that this requirement is mandatory, which means that if the customer’s context does not satisfy this requirement, then the associated service does not fit the user context, despite matching the request. In case these context files or a specific context parameter are absent, the ranking assigns the value 0 for the overall service rank relevant to such context type or for the missing particular parameter, respectively. If no context files exist, service ranking is based exclusively on their relevancy to the user objectives.

The matching module measures the similarity between the request and Web service description based on a non-repeated keyword match scheme. First, it extracts the keywords from the service description, wherever the tag <wsdl:documentation> is found in the description file, whether the tag belongs to the service or to individual elements. Then, it performs a keyword pair-matching to measure the similarity between keywords obtained from the service request and those obtained from the service description. Further details on keyword/feature extraction can be found in [37].
Table 4
The manually identified set of services that serve the domain of ‘currency exchange’ in the test dataset.

<table>
<thead>
<tr>
<th>#</th>
<th>Currency exchange web services</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><a href="http://www.webservicex.net/CurrencyConvertor.asmx?WSDL">http://www.webservicex.net/CurrencyConvertor.asmx?WSDL</a></td>
</tr>
<tr>
<td>2</td>
<td><a href="http://server1.pointwsp.net/ws/finance/currency.asmx?WSDL">http://server1.pointwsp.net/ws/finance/currency.asmx?WSDL</a></td>
</tr>
<tr>
<td>3</td>
<td><a href="http://allysoft.ru/Blccurrency/currency.asmx?WSDL">http://allysoft.ru/Blccurrency/currency.asmx?WSDL</a></td>
</tr>
<tr>
<td>4</td>
<td><a href="http://freewebs.com/jimmy_cheng/CurrencyExchangeService.wsdl">http://freewebs.com/jimmy_cheng/CurrencyExchangeService.wsdl</a></td>
</tr>
<tr>
<td>6</td>
<td><a href="http://www.currencyserver.de/webservice/currencyservervbservice.asmx?WSDL">http://www.currencyserver.de/webservice/currencyservervbservice.asmx?WSDL</a></td>
</tr>
<tr>
<td>7</td>
<td><a href="http://currencyconverter.kowabunga.net/converter.asmx?WSDL">http://currencyconverter.kowabunga.net/converter.asmx?WSDL</a></td>
</tr>
<tr>
<td>8</td>
<td><a href="http://ws.soatrader.com/gama-system.com/1.0/CurrencyExchangeRates?wsdl">http://ws.soatrader.com/gama-system.com/1.0/CurrencyExchangeRates?wsdl</a></td>
</tr>
<tr>
<td>11</td>
<td><a href="http://ws2.serviceobjects.net/ce/CurrencyExchange.asmx?WSDL">http://ws2.serviceobjects.net/ce/CurrencyExchange.asmx?WSDL</a></td>
</tr>
<tr>
<td>12</td>
<td><a href="http://www.petermeinl.de/CurrencyConverter/CurrencyConverter.asmx?wsdl">http://www.petermeinl.de/CurrencyConverter/CurrencyConverter.asmx?wsdl</a></td>
</tr>
<tr>
<td>13</td>
<td><a href="http://currency.niekutis.net/currency.asmx?wsdl">http://currency.niekutis.net/currency.asmx?wsdl</a></td>
</tr>
<tr>
<td>15</td>
<td><a href="http://www.xignite.com/xCurrencies.asmx?wsdl">http://www.xignite.com/xCurrencies.asmx?wsdl</a></td>
</tr>
<tr>
<td>17</td>
<td><a href="http://ws.strikeiron.com/ForeignExchangeRate3?WSDL">http://ws.strikeiron.com/ForeignExchangeRate3?WSDL</a></td>
</tr>
<tr>
<td>18</td>
<td><a href="http://ws.soatrader.com/haydonhill.com/0.1/Currency?wsdl">http://ws.soatrader.com/haydonhill.com/0.1/Currency?wsdl</a></td>
</tr>
</tbody>
</table>

Listing 1: The environment context file generated for the Web service XigniteCurrencies.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:env="http://www.example.sample/env#">
  <rdf:Description
    rdf:about="http://www.xignite.com/xCurrencies.asmx?wsdl">
    <env:serviceName>DOTSCurrencyExchange</env:serviceName>
  </rdf:Description>
  <rdf:Description
    rdf:about="http://www.example.sample/env/network">
    <env:signalStrength>good</env:signalStrength>
    <env:bandwidth env:type="required">2</env:bandwidth>
    <env:dataRate>20</env:dataRate>
    <env:errorRate>10</env:errorRate>
  </rdf:Description>
  <rdf:Description
    rdf:about="http://www.example.sample/env/proximity">
    <env:location>
      <env:name>city_center</env:name>
      <env:longitude>44.242102</env:longitude>
      <env:latitude>−76.516589</env:latitude>
    </env:location>
  </rdf:Description>
</rdf:RDF>
```

Fig. 3(b) presents the discovered services that match a service request searching for “currency exchange” Web services. The list of relevant services is ranked according to the randomly generated context information. A manual inspection of the retrieved services and their generated files shows that the Web service CurrencyRates fits the best with respect to the current user context. We then introduce two changes in the user context: (1) the user prefers services that detect the user location and puts the local currency as the default value, (2) the user device does not support PayPass transactions, which basically means the device does not support NFC (NFCSupport=False Table 1). DaaS re-ranks the same list of retrieved services according to the user’s new context, as shown in Fig. 3(c). A closer look at the context files of the Web service that DaaS ranked first, CurrencyServerWebService in this case reveals that this service supports the local currency detection, and its generated context file CurrencyServerWebService.env does not specify NFC-enabled as a required feature of the customer’s device. On the other hand, services such as Service1, CurrencyExchangeService, and DOTSCurrencyExchange are removed because their context files (.mbl) mandate the NFC support.

7. Performance evaluation

In order to evaluate the performance of the DaaS framework, several experiments are conducted. These experiments investigate various aspects including, overall response time of the framework for different service requests, the contribution
of various framework components to the response time, system scalability, overhead cost of context processing, relative improvements in the overall precision, and the quality of context-aware service recommendations. Experiments also demonstrate how clustering Web services into functionally similar groups improves the overall response time, while obtaining a better quality of service recommendations.

7.1. Response time

The response time shown in these experiments is end-to-end, including both communication and processing time. The service matching process is keyword-based. The matching is carried out using NGD and services are initially ranked according to their absolute relevancy to the user’s objectives. Once relevant services are retrieved, DaaS applies context matching to check how relevant the retrieved services are to the user’s context.

Fig. 4(a) depicts the average response time for two different service requests vs. varying numbers of simultaneous customers. The two service requests are: “email verification” and “currency exchange”. The response of the first request comprises ‘5’ relevant services, while the second request resulted in a list of ‘18’ relevant Web services. Fig. 4(a) also highlights two points. The first is how response time is proportional to both the number of retrieved matched services and concurrent requests. The second point demonstrates that the increase in response time is non-linear, which means that response time tends to rapidly increase when the number of relevant services increases; more specifically when more context is processed. Such an increase in response time stems from the overhead incurred by context matching mainly in service ranking.

There are several DaaS components which contribute to this response time. The break down of the overall response time with regard to the contributions of the various DaaS components involved is illustrated in Fig. 4(b). We observe that, the highest contributions belong to service matching, communication time (both ways), and context processing, respectively. Since the communication time is relatively constant, the response time is dominated by the service matching and context processing. Therefore, we can infer that those two components are the key differentiators between various service requests. To this end, we implement our clustering approach aiming to reduce the impact of the corpus size on the matching time.

7.2. Impact of clustering

To show the impact of clustering Web services on the system performance, we measure the break down of the overall response time with and without clustering the Web services in the test corpus. Fig. 4(c) shows the significant improvements in the processing time of the service matching component, which in turn significantly reduces overall response time. From Fig. 4(c), we can observe: (1) the service matching time is independent of the corpus size, as the matching is only performed on the number of services in the target cluster, (2) using Web service clustering renders the context processing dominating the overall response time. However, we can conclude that although the context processing time is inevitable, the potential performance benefits are remarkable.
7.3. Context overhead

During context matching, services that do not satisfy any of the mandatory/required context attributes are removed from the relevant list, while the remainder are re-ranked according to their integrated degree of relevancy to the user context. Incorporating the context into the service discovery process brings benefits into the quality of recommendations, while adding some overhead. Fig. 5(a) shows the overhead incurred by the context processing for the two different queries.

7.4. Scalability

DaaS takes advantage of the elastic resource provisioning offered by the cloud, in our case Amazon EC2’s autoscale feature, to provide the scalability and accommodate increasing numbers of requests. The service provider sets particular quality of service (QoS) constraints and the cloud scales up to accommodate increasing demands and satisfy those constraints.

We carry out two experiments to evaluate the scalability of the DaaS framework. The first experiment tests how DaaS performs when a large number of requests are received and handled by a fixed hardware setup. We dedicate only one instance of cloud server to handle incoming requests and we measure the end-to-end response time versus an increasing number of requests. We simulate the number of requests using multithreading. Fig. 4(a) shows the results of this experiment. We observe that DaaS serves incoming requests on a best effort basis, where the response latency becomes higher as the load increase.

The second experiment evaluates the DaaS scalability based on QoS constraints. We set the response time not to exceed 240 ms as a quality of service measure. This means that when a violation occurs, our cloud setup brings in more resources (i.e., launches another instance of the type t1.large from our pool of VMs) to maintain the threshold below the prespecified value. In this case no matter how large the number of requests becomes, the response time does not exceed this threshold.
value. Fig. 4(d) shows how DaaS scales up to accommodate increasing requests, while maintaining the response time below the desired threshold. We observe that the new instance takes up to 60–90 s to become active and ready to handle requests. This explains why the response time reaches beyond 240 ms. The EC2 load balancer (ELB) then splits the load between participating instances. Fig. 4(a) also shows that there is a non-negligible overhead incurred by the scheduling of the load balancer, which explains why the response time of 2X requests with 2 instances is slightly more than the response time of X requests with a single instance. However, the scheduling overhead is constant and does not increase with further addition of instances.

7.5. Discovery precision

To measure the positive impact that the context processing brings to the service discovery, we calculate the Context Precision (CP). The CP is defined as the percentage of improvement in the result that the context handling achieves over the originally retrieved services. To nullify the impact of the matching algorithm, we assume that the retrieved services are all relevant to the service request (precision 100%), and all the relevant services to the service request, that are supposed to exist in the corpus, are retrieved (recall 100%). Let us assume that the \( |S| \) is the number of retrieved services and the context consideration reduces the number of relevant services to only \( X \) services out of \( |S| \). Then, the CP is calculated by Eq. (5).

\[
CP = \frac{|S| - X}{|S|}.
\]

(5)

The context precision is illustrated in Fig. 5(b) for the two queries we use. It is worth noting that incorporating context to the discovery process does not affect the recall within our assumptions.

7.6. Quality of discovery

The quality of service discovery reflects the user-perceived quality of service recommendations. In this experiment, we measure the deviation between the ranking score that DaaS assigns to service recommendations and the actual user ratings. To evaluate the quality of our service recommendations, we use the Mean Absolute Error (MAE). MAE represents the average absolute deviation between the rank DaaS assigns to relevant services according to their fit to the various context and the actual user ratings. MAE is calculated according to Eq. (6).

\[
MAE = \frac{\sum_{s \in S} |\text{Rank}_{s} - \text{Rate}_{s}|}{|S|}
\]

(6)

where \( S \) is the set of relevant services to a request \( R \), \( \text{Rank}_{s} \) and \( \text{Rate}_{s} \) are the rank and the actual user rating of a service \( s \), respectively, and \( |S| \) is the number of retrieved services. In this experiment we run 10 different service requests. Each request results in a response that contains a set of relevant Web services as shown in Table 5. DaaS ranks the services of each response according to three different user profiles. Each profile encompasses a random set of features of a user’s device and preferences. We manually inspect the first 5 services in each response along with the context profiles and rate each service according our personal judgment, which basically projects the user satisfaction. We then calculate the average MAE for each service request based on the formula given in Eq. (6). Fig. 6 shows the MAE for the 10 different service requests. The lower the value of MAE, the better the quality of the ranking.
8. Conclusion

Web service discovery is a key enabler for the adoption of Web services technology in mobile environments, where limited resources and unreliable communication of wireless networks pose unique challenges for service discovery. With recent expansions aiding ubiquitous access of networked resources, cloud computing can take mobile applications far beyond the resource limitations of their host devices. In this paper, we identified the limitations of current discovery approaches and the requirements of sound and reliable discovery mechanisms that can efficiently fetch Web services that match the user context within resource-constrained environments and provide a differential user experience.

This research presents DaaS (WS-Discovery as a service), a cloud-based context-aware service discovery framework for mobile environments. DaaS removes the burden on mobile resources by shifting resource-intensive processes to the cloud. DaaS is presented in a three-layer structure and takes into consideration network characteristics, user preferences and context, and device profile. A comprehensive description is provided for each component along with the possible approaches from existing technologies that could potentially implement such a component; as well as some of the research efforts in that respect.

The experimental validation of the DaaS framework demonstrates its ability to rule out retrieved services that do not fit the user’s context and to rank relevant services that are best suited to the user’s needs. Performance evaluation reveals that the overall response time is dominated by the service matching process, while context processing contributes a relatively small component, yet this varies according the number of matching service. However, such context matching overhead is negligible in contrast with the improvements of the service discovery quality. Experiments also show that clustering Web services into functionally similar groups improves the overall response time, while obtaining better quality of service recommendations due to the fact that the service matching is restricted to relevant groups. Experiments also demonstrate the effectiveness of scalability provided to DaaS by the underlying cloud platform. The experimental results support the conclusion that the proper utilization of context information aids finding services that are in the best interest of the requester, and most appropriate for the current situation.

In the future, we plan to study the tradeoff between performing the context matching on the mobile device and offloading it to the cloud, where service matching occurs. We also plan to examine how DaaS scales across distributed data centers in the cloud.

References


