CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	Delivering mobile cloud services to the user: description, discovery, and consumption
Authors	O'Sullivan, Michael J.;Grigoras, Dan
Publication date	2015-06
Original Citation	O'SULLIVAN, M. J. & GRIGORAS, D. 2015. Delivering Mobile Cloud Services to the User: Description, Discovery, and Consumption. 2015 IEEE International Conference on Mobile Services (MS). New York City, USA, 27 Jun - 2 Jul. 2015. IEEE. pp. 49-56. doi:10.1109/MobServ.2015.17
Type of publication	Conference item
Link to publisher's version	10.1109/MobServ.2015.17
Rights	© 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
Download date	2026-11-06 06:59:47
Item downloaded from	https://hdl.handle.net/10468/1965



University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Delivering Mobile Cloud Services to the User: Description, Discovery, and Consumption

Michael J. O'Sullivan, Dan Grigoras
Department of Computer Science
University College Cork, Cork, Ireland
{m.osullivan, grigoras}@cs.ucc.ie

***Abstract* – The mobile cloud computing paradigm can offer relevant and useful services to the users of smart mobile devices. Such public services already exist on the web and in cloud deployments, by implementing common web service standards. However, these services are described by mark-up languages, such as XML, that cannot be comprehended by non-specialists. Furthermore, the lack of common interfaces for related services makes discovery and consumption difficult for both users and software. The problem of service description, discovery, and consumption for the mobile cloud must be addressed to allow users to benefit from these services on mobile devices. This paper introduces our work on a mobile cloud service discovery solution, which is utilised by our mobile cloud middleware, Context Aware Mobile Cloud Services (CAMCS). The aim of our approach is to remove complex mark-up languages from the description and discovery process. By means of the Cloud Personal Assistant (CPA) assigned to each user of CAMCS, relevant mobile cloud services can be discovered and consumed easily by the end user from the mobile device. We present the discovery process, the architecture of our own service registry, and service description structure. CAMCS allows services to be used from the mobile device through a user's CPA, by means of user defined tasks. We present the task model of the CPA enabled by our solution, including automatic tasks, which can perform work for the user without an explicit request.**

Keywords: mobile cloud, middleware, services, description, discovery, consumption

I. INTRODUCTION

Mobile cloud computing (MCC) research often explores the feasibility and methods for offloading the required computation by demanding applications from the mobile device to the cloud. By performing this offload, applications that could not run on the mobile device due to constraints such as limited CPU processing capability, low memory and storage capacity, can now be delivered to users of these devices. However, implementing the MCC model faces various difficulties of its own, such as continuous network connectivity requirements, and the high energy usage as a result. These demands add to the complexity of this model. Our approach to the mobile cloud focuses on an alternative route, by making use of web services already deployed in the cloud. Such services form the basis of service-oriented architecture (SOA), and can be used for delivering useful and relevant functionality and information to the mobile user.

As with other MCC application models, taking an SOA approach presents problems of its own. Services deployed in the cloud conform to web service standards, such as the Simple Object Access Protocol (SOAP) [1], and Representational State Transfer (REST) [2]. How to describe, discover, and consume these services from a mobile device, presents several challenges. In terms of description, services are often described using the Web Services Description Language (WSDL), which is XML-based, and cannot be understood by non-specialist users, and therefore, on its own, is useless for describing a service to an end user. As a result of these XML-based descriptions, the user has traditionally been unable to take part in the discovery process. However, it is widely agreed that XML was never supposed to be directly presented to end users, and is only for use by software and developers; a new approach is therefore required. RESTful services often do not have associated descriptions at all, aside from API documentation. In the area of service discovery, existing research has shown that automatic discovery of appropriate services is simply not mature enough for widespread use, and is therefore still a very manual, developer-oriented process. For service consumption, solutions must be able to work with various different kinds of services and web service technologies. Clients can be developed to access SOAP and RESTful services. To-date, standards do not exist for comparing similar services, nor for invoking services that may take similar input parameters, and output similar content.

We address these problems by implementing our own user-oriented service discovery process, which allows discovery of existing web services from our own service registry solution. The primary contribution of this particular registry is the means by which stored service description records are structured; the design goal being a user-oriented approach. The requirement for this work results from our MCC middleware, Context Aware Mobile Cloud Services (CAMCS) [3]. Each user of CAMCS is assigned their own Cloud Personal Assistant (CPA) [4]. By means of a thin client

application running on the mobile device, an end-user can request a task to be completed by their CPA, and view the result, saved at the CPA, at a later time. The CPA uses cloud-based services that it has discovered to complete these tasks. Users need to be able to select from discovered cloud-based services capable of completing their task, in a user-friendly way. His/her CPA will use the selected service for the task. Therefore, our solution is demonstrated through user CPAs.

The motivation behind CAMCS and user CPA's, is to deliver mobile cloud services to the user, in a disconnected fashion – if the user looses network connectivity, breaking their connection to cloud-based infrastructure, work assigned to the CPA will continue to execute without interruption. As the CPA works with cloud-based services, this approach to MCC brings the advantages of low data transfer between the mobile device and a CPA, and no requirement for a continuous connection between them, resulting in device energy savings. This is in comparison with other MCC solutions, which offload code-bases or virtual machines to cloud infrastructure, and require continuous connection and frequent data transfer. The focus of this work is in the problems of discovery and use of cloud-based services, by the mobile user, with their CPA.

In this paper, we introduce our MCC registry model, along with the service description structure used to store service information within. We present how a CPA uses our solution to discover services by querying the registry; a user can then choose from among these whichever service they believe suitable for the task. We show how the data required for consuming the service is collected from the user, and how a service is consumed for the purposes of completing a task. We will also present the model used by the CPA, for representing user tasks, and how these tasks run with discovered services. Enabled features are also presented, such as automatic task execution, which allows the CPA to execute tasks without any request from the user, enabling disconnected operation.

The remainder of this paper is organised as follows; Section II describes the MCC service registry and the structure of service descriptions used. Section III will present the discovery and consumption process used by the CPA. Section IV will present the task model of the CPA that uses our registry solution. Section V discusses related work, followed by conclusions in Section VI.

II. MCC SERVICE REGISTRY FOR USER ORIENTED SERVICE DESCRIPTIONS

Many services providing various functionalities already exist in the cloud. These conform to existing web service access technologies, and service registries exist containing descriptions of these services; these are not designed for human interaction. By means of the CPA, mobile users can take advantage of these services, stored in our own registry with our user-oriented description format. The user-friendly descriptions will allow an average user with little or no technical experience to easily find and use cloud-based services. We will first present the registry implementation, followed by the service description structure used to store service records.

A. The Service Registry

The service registry implementation is a JavaEE based application that is deployable to any application container running on a cloud-based server. The registry provides an API for querying services by search terms, which then returns a list of matched services. The registry is built on top of a NoSQL database, MongoDB. A NoSQL database was chosen because it is a document based data-store. Therefore, each service record is represented by a document in the database. This is easier to work with than having several various tables in a relational database such as MySQL, which need to cross-reference each other.

The search operation takes a string-based query provided by the user. The Apache OpenNLP [5] library is used to tokenise the query, and this is matched against a set of descriptive terms that are stored as part of each service record within the registry (discussed further in Section III). The results of the query, which will contain a list of matching services found in the registry, are returned to the caller in JSON format. The search algorithm of the registry can be modified in the future to include semantic matching techniques between the user query and the descriptive terms. For the time being, the current approach has proved to be more effective than querying a UDDI registry, which we used in a previous work [4]. The limited scope for querying with a UDDI registry (exact or approximate match queries on service names) was one of the issues that drove the creation of our own solution.

Service providers or developers need to be able to add services into the registry in order for users to find them. Currently, another of our API endpoints allows developers to add their services to the registry by means of a HTTP PUT request; if the service is described according to the format by which we store services in the registry, the service will be added. The service description will again be marked-up in JSON and sent in the body of the HTTP request. Future possibilities we are considering include automatic converters to extract the required information from existing WSDL files in order for them to be automatically added to our registry. Another possibility includes a web-based interface where developers can describe their service without having to provide any existing description file for the service. This would be far more appropriate for RESTful based services, which typically do not use service description files. Both of these approaches will allow existing web services to be added to the registry with little additional effort; they would not need to be manually converted to our own format.

The registry, along with CAMCS, is currently deployed on the IBM Bluemix platform, which is built on CloudFoundry.

B. Service Description Structure

The structure of the service descriptions we use within our registry gives power and flexibility to our MCC service solution more so than the registry itself – see Fig. 1. The aim of the descriptions in this registry is that they will allow simple user interaction to discover and utilise a service, with no technical details or mark-up of any kind presented to the user. The registry will store references to two types of service,

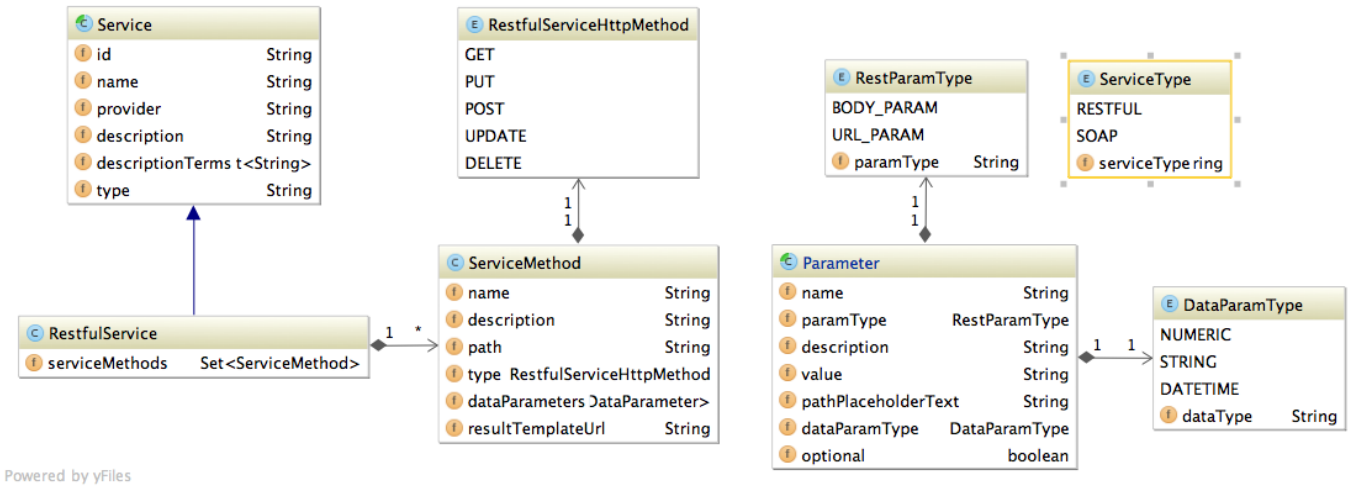


Fig. 1. Service Structure. Services contain ServiceMethods, which in turn contain Parameters. Each entity contains user-friendly names and descriptions for use during the discovery and service selection process by the user. In this implementation, we have placed our focus on RESTful services, and specific entities for these services can be seen.

SOAP and RESTful. In our current implementation, we have only used RESTful services for evaluation. Ultimately, the high level descriptions will be the same for both.

1) *Service*

A *Service* is simply a high level record/abstraction for a service offered by a provider. A Service features a *name*, a *description*, and a *provider*. These are in place for human consumption in the service discovery process. The Service abstraction also features a *type*, which can either be "soap" or "rest", indicating which web service technology is used to implement the service. As described in the previous subsection, a Service also contains a collection of *descriptive terms* that can be specified by the developer. The name and description are also queried for matching terms in the query, along with the collection of descriptive terms. A Service contains a collection of several *ServiceMethods*.

2) *ServiceMethod*

A *ServiceMethod* record corresponds to a function/operation that is offered as part of the service. If one were to compare with WSDL, there is one ServiceMethod for each operation. A ServiceMethod, like the Service record, also features a name and description. These are also high-level descriptions for human consumption in the discovery process, and describe what the operation does. They also feature a *path*, which is the URL endpoint for calling that ServiceMethod as a function/operation, over HTTP. Where the URL takes parameters, these are represented in the path string with "$" placeholders, such as /users/get/$userid. ServiceMethods also feature a type, which, for RESTful services in this implementation, represents which HTTP verb the ServiceMethod will respond to (a GET/PUT/POST/DELETE request). A *Result Template URL* can also be found in a ServiceMethod record. This is a URL to a HTML template webpage, which the service developer specifies. This template will be used to display the service results, and can be customised to a company's own branding. Results are described in more detail in Section IV. Finally, a ServiceMethod features a collection of *Parameters*.

3) *Parameter*

As the name would imply, a *Parameter* is an input data parameter to the service. Once again for human consumption in the discovery process, a Parameter features a name and description (how these are used is shown in Section III). A Parameter only contains one other attribute, a type. For a RESTful service, a parameter type indicates if it is a URL parameter to be passed to the service in the URL, or a body parameter, to be passed marked-up in the HTTP request body. If it is a URL type parameter, the name will match up to the respective placeholder used in the path attribute of the ServiceMethod, and will replace it with the actual parameter data value at call time. Parameters also feature a *Data Param Type* attribute. End users need not be concerned with required data types; our Android thin client application uses this for collecting valid data from the user. Currently, it supports Strings, numeric types, and datetime types. This is described in more detail in Section III.

For complex parameter data types, a Parameter can include a sub-collection of Parameters, which make up the complex type. These are not presented to the user in any different way in the discovery process compared to simple data type based Parameters; the user will not see anything which resembles the underlying data types.

One area that caused some trouble was the requirement of several web service API's to have a developer key sent with the request. Every end user with a CPA will not have a developer key. To overcome this, the CAMCS developer key was inserted into the registry as a Parameter of each ServiceMethod that required a key (unseen by the end user). This is not ideal, and we believe that more open web and cloud service API's will be required for an MCC approach based on SOA.

C. *Querying*

When queried for services, the registry can either return complete Service records as described, or reduced versions of the Service records, which just contain the names and descriptions for the ServiceMethods and corresponding Parameters.

III. SERVICE DISCOVERY AND CONSUMPTION WITH THE CPA

The CPA will query the registry, with the task description as the query string. Interactions can be seen in Fig. 4.

A. *Discovery*

With a user-provided description of a task to complete, consisting of a task name and description, the CPA will query the registry with this description (or "query-string"). The registry will return the full Service descriptions for matching services, including the ServiceMethods offered by a service, and the Parameters for each of the ServiceMethods. The initial query of the registry by the CPA, given the query-string provided by the end-user, is the beginning of the discovery process.

With this list of discovered services from the registry, the CPA notifies the mobile user that services have been discovered for a given task – see Fig 2. The user is presented with the discovered Services, and each of their ServiceMethods, presented as "operations" on the device – see Fig. 3. The user will see the name and description of each operation. This is something of an MCC service market, where the user has indirectly searched for a service through their CPA, and can browse the results until they find an appropriate service. The user selects the operation to be used for the task, and this is sent to the CPA. At this point, the CPA differentiates between tasks that are new, and tasks that have run before, but this paper focuses on new tasks only (further information about tasks is given in Section IV). For the chosen ServiceMethod, the CPA extracts the parameter record from the ServiceMethod record. This is sent to the thin client. A form is displayed to the user with a field for each of the parameters that are described in the Parameter record – see Fig. 5. The user is provided with the name and description of each parameter, and is prompted to fill in each of the parameters required by that service. For parameters with DataParamType attribute numeric, the user is restricted to numeric data entry on the keyboard. For datetime parameters, a time and date selection spinner is presented to the user. Once the user has filled in this form, the values for the parameters are sent to the CPA for storage.

Service consumption with CAMCS, as hinted in the name, supports sending contextual data gathered from the mobile device, to services that can make use of this data. This works by means of the Context Processor, a component of CAMCS, described in a previous work [6]. This data is not collected through the form-based interface; this is gathered from a service within the thin client, and sent separately at user-defined intervals to CAMCS, for storage with the Context Processor, on behalf of the user. For services that take contextual data as parameters, the CPA will gather the relevant contextual data for the user, from the Context

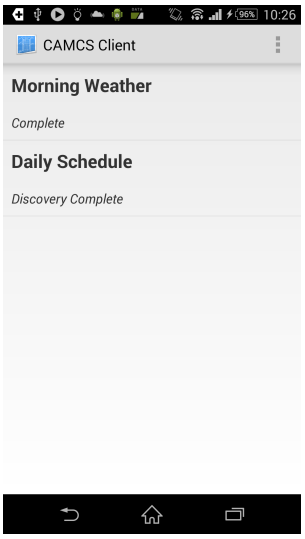


Fig. 2. Current Tasks List. Shows all tasks currently executing at the CPA for the user. Here, this task has finished service discovery.

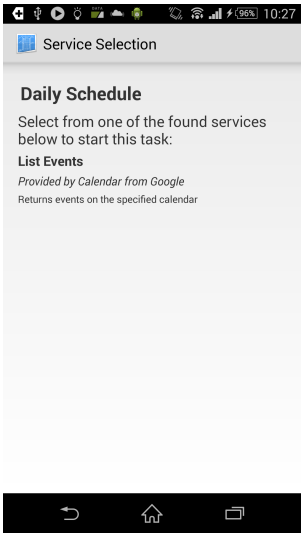


Fig. 3. Discovered Services List. This shows the operations (service methods) provided by discovered services that the user can choose from, including their names and descriptions from the registry.

Processor, and send this data to the service with the data parameters. The user must grant permission for a task to use contextual data first. This will be presented a future work.

Before task data is sent to the CPA, it is queued. The thin client observes the state of the mobile network, and depending on the evaluation (which takes into account signal strength, payload size, and current battery status), will either offload the data, or wait until the network quality has improved. This is explored in more detail, in a previous work [7].

B. *Service Consumption*

Now that the CPA has all the user-provided input values for the parameters required by the service, the CPA can consume the service, by sending it a HTTP request. As we are currently working with RESTful services, this request will use one of the previously described HTTP methods (GET, PUT,

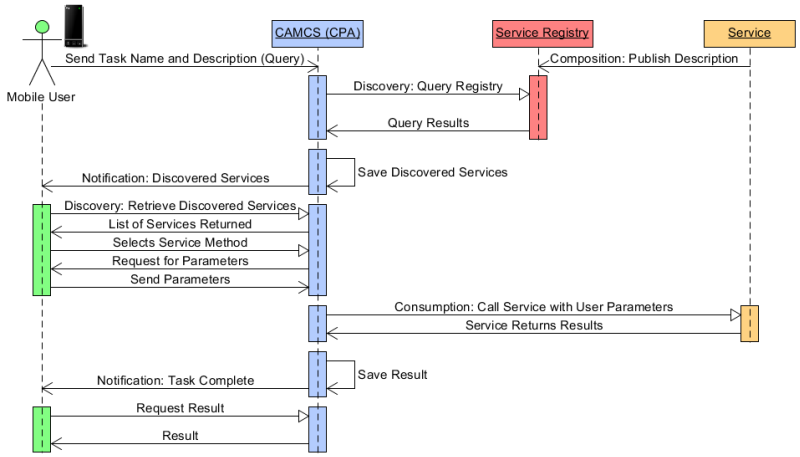


Fig. 4. Sequence/Interaction Diagram. This shows the sequence of events in the composition, discovery, and execution processes for a user task running for the first time. Tasks re-running, or tasks automatically started by the CPA, will not require service selection or parameter entry.

etc.); any parameters that are declared in the Parameter record as being a URL parameter are inserted into the location defined by the placeholders in the path attribute of the ServiceMethod record being used. Parameters provided that are declared as being of type body are converted into JSON, following a {"paramName": "paramValue"} format, and inserted into the body of the HTTP request.

The result of the service execution is sent in the body of the HTTP response to the request. The CPA will store this result for the user, and a notification will be sent to the mobile device. Upon opening the notification, the result is retrieved and displayed on the mobile device. At this point, the task is considered complete, and is moved to a completed tasks list.

IV. CPA TASK MODEL: FEATURES AND IMPLEMENTATION

The task model of the CPA is implemented around the disconnected operation principle. Once task details have been sent to CAMCS, they are stored with the CPA of the user, and the mobile device does not need to remain connected to the cloud deployment for the task to execute. We will now present the structure of a task, followed by the task execution process, and how our model enables automatic task execution. We also describe results storage and presentation.

A. Task Structure

The CPA and all of its related data (such as details for users and tasks) are stored. Once a user has sent a task description to CAMCS, this task is created at their CPA, and placed into a current tasks list. This is represented as a subdocument within MongoDB. Each task contains several attributes:

- Name: a user specified task name, entered into the thin client by the user on the mobile device
- Query: the query associated with the task, which is the description entered into the thin client by the user on the mobile device. This is the query string sent to the registry for service discovery.
- Discovered Services: a list of discovered services returned by the registry in response to a query search.
- Service Record: this is the service selected by the user that will be used to complete the task, as chosen from the discovery results. Contains the technical data required to invoke the service.
- Operation Name: this is the name of the operation (ServiceMethod) offered by the service that the user has chosen to use for completing the task.
- Results: a list of results. Tasks can be executed more than once with different results each time.
- Task Data: a map which stores the values that the user has provided for each parameter required by the service.

B. Task Implementation

Within CAMCS, a runnable task is defined by a TaskExecutable class. We use the Quartz Scheduler [8], which provides a scheduler for "jobs". TaskExecutable subclasses the Quartz Job class, and is executed in its own thread of execution. All tasks in CAMCS, from all CPA's, are handled by a Task Handler, which is responsible for taking the tasks from the CPA's, and starting them as Quartz jobs using the TaskExecutable. The required information to execute a task is passed by the Task Handler to the job by means of a map. This data will include all the required data outlined previously.

Tasks can be executed either by an explicit request from the user (running a new task or explicitly re-running a completed task), or as an automatic task, whereby the CPA chooses to run a task without any explicit user request.

1) User Requested Task Execution

A user creates a new task with a name and query using the mobile thin client. Once this is sent to the CPA, the CPA starts the task execution by passing the task to the Task Handler. The Task Handler checks if a valid Service Record has already been associated with this task. Being a new task, this will not be the case, and so the Task Handler will begin service

discovery by taking the user provided query for the task, and contacting the MCC service registry.

Once the user has selected the appropriate service and operation, and task execution has completed, the task result is returned to the CPA from the Task Handler, and the task status is set to complete. When the user views the task result on the thin client, the CPA moves the task from the current tasks list to the completed tasks list.

The user can signal any completed task to run again from the thin client. In this case, the name, query, and parameters data, need not be collected from the user again, as they are already stored with the CPA. Service discovery will not take place again (unless the user specifies they wish to choose a different service); task execution with a Quartz job using TaskExecutable will take place immediately. If the user wishes to change any of the previously given input parameters to the service, they can request to do so, in which case they can enter them into the parameter data collection form again.

2) *Automatic Task Execution*

The benefit of the CPA model is the ability to perform work for the user without an explicit request. This furthers the disconnected operation goal. In this case, even if the user is disconnected from the cloud deployment, and hence the CPA, work can still take place for the user. Based on times specified by the user, the CPA can also automatically run a previously completed task again, using the previously chosen service and provided parameter data values.

In this implementation, when the user has re-executed a task a given number of times (three currently), the CPA sends a notification to the user, asking would they like to schedule the task to run regularly on an automatic basis. The user can set the days of the week, and times, when the task should automatically execute – see Fig. 6. The repeat task data is stored with the CPA. This data is used to schedule cron triggers, with Quartz. These use regular expressions to define when a trigger should run to start a given job (or task in our case). The user can stop the automatic task execution at any time. Whenever an automatically repeated task is executed, the result is added to the results list of the task with the given date and time of execution. The task is moved from the completed tasks list, back to the current tasks list. A completion notification is sent to the user. Another future possibility is that tasks can automatically run based on patterns of when the user has explicitly requested that the task should run in the past. This way, the user would not have to manually specify when a task should execute.

3) *Task Results*

Tasks can run several times. The results of a task can differ depending on the different times it is executed, or if the user changes the input parameters. Therefore, each result is stored with a timestamp of execution. When the user opens a task, they can view the results for each of the previous executions.

A result may range from textual data, (a hotel booking reference number), or something numeric (statistical results from a data processing service). To provide flexibility and

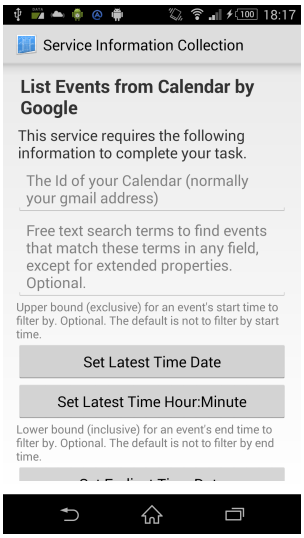


Fig. 5. Parameter Data Value Entry. The user enters the values for each parameter that the service requires for task execution. The description of parameters is displayed for the user. This is the list events API call from Google Calendar, which fetches events from a calendar. Upper/lower bound times are specified as datetime parameters, so buttons are presented for time/date selection spinners.

customisation for service results, a HTML solution was used. Companies such as Amazon could advertise other products that they offer within a task result HTML page. The Result Template of a ServiceMethod record contains a URL to a template HTML page. This page, which should be stored on a publically available server, makes use of JSON2HTML [9]. This library uses JavaScript to convert JSON data into a HTML representation. This is accomplished by means of specifying a transformation, which will convert a JSON string to HTML. The transformation is already stored in the HTML template page, along with any other mark-up/formatting details (CSS, JavaScript) that the service developer has chosen to include in the page. When the CPA has received the JSON result data from the service, it will fetch the result template page given by the URL. Using the JSoup library [10], the JSON service result is written into the <head> section of the page. The result HTML page is added into the result list for the Task, and stored in this marked-up format.

Upon task completion, the user is notified of the result; this is displayed in an Android WebView – see Fig. 7. When loaded, the transformation is applied to the JSON result data.

V. RELATED WORK

Many approaches to implementing MCC take the form of offloading entire applications or computations to a server, such as the Cloudlet approach [11]. Cloudlets are small servers deployed in areas where many people gather. Users interact with virtual machines running on the Cloudlet through their mobile device. Code offloading projects, such as MAUI [12], and CloneCloud [13], offload execution of mobile applications, normally at the method level, for execution to a server, with results returned to the device. Our approach focuses on offloading tasks, which are described with text queries, and services in the cloud are used to complete them.

Some work in the mobile cloud web service discovery and provisioning area includes a mobile web service provisioning framework [14], and a mobile cloud web service discovery solution, known as DaaS (Discovery as a Service) [15]. The work presented for DaaS is similar to the work in this paper, in that a discovery process takes place to find appropriate web based services for users; the discovery process takes into account even more features, such as user context, which is something we are also implementing in future work, building on our own previous work [6]. Where our work differs is that in DaaS, and the mobile web service-provisioning framework, the cloud is used for web service discovery, but the web services themselves run on mobile devices, rather than in the cloud. Another solution is the VOLARE middleware [16], which monitors the mobile device context, so as to dynamically change service providers at runtime to maintain a certain Quality of Service (QoS) level. The Location-Aware Service Provision and Discovery (LASPD) framework by Zhu et al [17] is focused on web services, like the DaaS approach, hosted on mobile devices, and implements locality based discovery and consumption of services for mobile devices located nearby. The work on SAMI by Sanae et al [18] uses an SOA approach to deliver services to users that are provided by mobile network operators.

Other solutions that deliver web based services to mobile devices include Google Now [19]. This can provide the user with information relevant to the user's situation, such as weather at given locations, and traffic en-route to a workplace. This is the same functionality our approach can deliver. However, it only works with Google services and products, whereas our solution can work with all kinds of cloud-based services. It also carries out work locally on the mobile device, whereas our architecture is completely cloud based. Apple Siri [20] retrieves data from web services for the user, based on voice queries. It does not work without an Internet connection. In a work by Wang and Deters [21], a cloud-based middleware was developed that can bring web services together into a mash-up form. This work stores service data in a MySQL database, and requires a user to enter WSDL file URLs for discovery. In our approach, we used structured documents in a NoSQL MongoDB database, and users do not need to know any service description mark-up language. The ALILI framework by Lomotey and Deters [22] used web services to bring file storage and sharing to mobile devices. The work by Sankaranarayanan et al [23] proposes the SMILE middleware, which aims to deliver relevant services to mobile devices, by allowing services to share related data with each other, such as travel dates for flight and hotel booking. The Mobile Web Services Mediation Framework by Srirama et al [24] implements enterprise service bus technologies to deliver features such as message compression, QoS guarantees and transaction support for mobile clients using web services.

For service discovery, a study by Sun et al [25] compares various service description languages in terms of what features they provide, and suitability for a cloud environment. In the area of sematic discovery, works such as [26] and [27] use an extension to OWL, the Web Ontology Language, called OWL-S, to describe web services semantically. OWL is a W3C standard for marking up semantic content on the web.

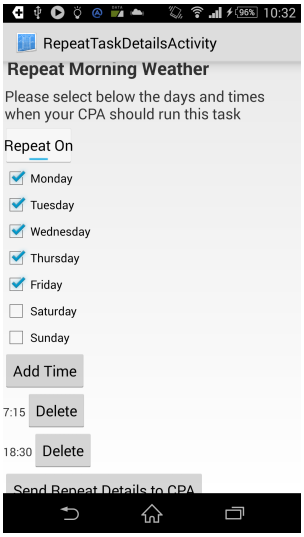


Fig. 6. Automatic Task Repeat. The user can enter the date and time details for when the CPA should automatically repeat a previously completed task. The Quartz Scheduler within CAMCS is then scheduled to re-execute the task, with results returned to the CPA.

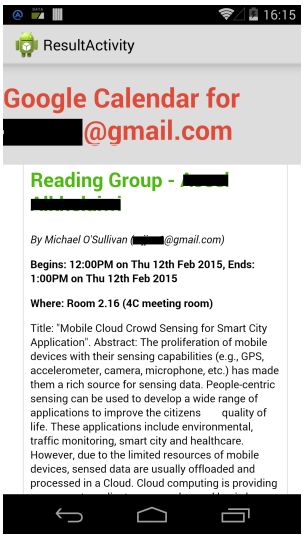


Fig. 7. Task Results Display. The result of the task execution is displayed as a HTML webpage using an Android WebView (personal details hidden). Here, the transformed output from the Google Calendar List Events API call is shown for the daily schedule of the user.

VI. CONCLUSIONS

In this paper, we have introduced an MCC based solution for enabling description, discovery, and consumption of mobile cloud computing (MCC) services. The motivation behind this solution is to utilise the existing web and cloud based services to deliver information and functionality to the mobile user. This is compared to existing MCC solutions which are more demanding on mobile resources, where virtual machines, or application code, are offloaded to remote cloud servers for execution.

Several contributions have been made in this paper. First, we have presented our service registry, which stores service descriptions in our own description format, which was designed with user-oriented discovery in mind. It will allow users to be a part of the discovery process, not previously practical with mark-ups such as XML. Secondly, the discovery process allows the user to benefit from cloud-based services through the device. We have shown how the solutions have been applied to our MCC middleware project, Context Aware Mobile Cloud Services (CAMCS). Through use of CAMCS with each user's own Cloud Personal Assistant (CPA), a user can search for services to complete tasks; our service descriptions from our registry are presented to the user for this, bringing them into the discovery process. We have shown how data for service parameters are collected from the user, and how task results from services are stored and presented. Finally, we presented the user task model for CPAs, as well as automatic task execution. Automatic tasks enable the design objective of disconnected operation for CPAs.

While this solution uses existing technologies and protocols, due to the user-oriented discovery process and services descriptions, this work opens a new direction in accessing cloud services by mobile users. In our future work, this MCC solution for describing, discovering, and consuming cloud based services will continue to be developed. Context data collected and stored with the Context Processor can be used for service consumption; details of such capabilities will be added to our service descriptions. We are also working on complex service interactions. This involves service flows, where a CPA makes many different calls to a service to complete a task. Each call completes a different objective of the overall goal. For example, a hotel booking service, where one call retrieves available rooms, a second call will send personal details of the user, and a final call will provide payment information. For each step, the CPA can request the required data from the user; this data can then be saved at the CPA for future calls to the service, so the user does not have to provide the same data again, supporting disconnected operation and automatic task execution. Once the implementation of CAMCS has been completed, user evaluation studies will take place.

ACKNOWLEDGMENTS

The PhD research of Michael J. O'Sullivan is funded by the Embark Initiative of the Irish Research Council.

REFERENCES

[1] Simple Object Access Protocol (SOAP) W3C Specification. http://www.w3.org/TR/soap/. Last Accessed 28/10/14.

[2] R. T. Fielding. Architectural styles and the design of network-based software architectures. PhD Thesis. University of California, Irvine, 2000.

[3] M. J. O'Sullivan, D. Grigoras. User Experience of Mobile Cloud Applications – Current State and Future Directions, in Proceedings of the 12th International Symposium on Parallel and Distributed Computing, Bucharest, Romania, 27-30 June, 2013, pp. 85-92.

[4] M. J. O'Sullivan, D. Grigoras. The Cloud Personal Assistant for Providing Services to Mobile Clients, in Proceedings of IEEE 7th International Symposium on Service Oriented System Engineering (SOSE), Redwood City, San Francisco Bay, California, USA, 2013, pp. 477-484.

[5] Apache OpenNLP. https://opennlp.apache.org/ Last Accessed 23/02/15.

[6] M. J. O'Sullivan, D. Grigoras. Mobile Cloud Contextual Awareness with the Cloud Personal Assistant, Proceedings of the 2nd International Conference on Future Internet of Things and Cloud (FiCloud-2014), Barcelona, Spain, 27-29th August, 2014.

[7] M. J. O'Sullivan, D. Grigoras. Integrating Mobile And Cloud Resources Management Using The Cloud Personal Assistant, Simulation Modelling Practice and Theory, Vol. 50, pp. 20-41, January 2015, ISSN 1569-190X.

[8] Quartz Job Scheduler for Java. http://quartz-scheduler.org/. Last Accessed 28/10/14.

[9] JSON2HTML. http://json2html.com/ Last Accessed 23/02/15.

[10] jsoup. http://jsoup.org/ Last Accessed 23/02/15.

[11] M. Satyanarayanan, P. Bahl, R. Caceres, N. Davies. The Case for VM-Based Cloudlets in Mobile Computing, IEEE Pervasive Computing, 2009; 8(4), pp. 14-23.

[12] E. Cuervo, A. Balasubramanian, D.-K. Cho, A. Wolman, S. Saroiu, R. Chandra, et al. MAUI: making smartphones last longer with code offload, Proceedings of the 8th international conference on Mobile systems, applications, and services, San Francisco, California, USA. 1814441, ACM, 2010, pp. 49-62.

[13] B-.G. Chun, S. Ihm, P. Maniatis, M. Naik,A. Patti. CloneCloud: elastic execution between mobile device and cloud, Proceedings of the sixth conference on Computer systems, Salzburg, Austria, 1966473, ACM, 2011, pp. 301-314.

[14] K.Elgazzar, P. Martin, H. S. Hassanein. A Framework for Efficient Web Services Provisioning in Mobile Environments, The 3rd International Conference on Mobile Computing, Applications, and Services (MobiCASE 2011), Los Angeles, CA, 24-27 October, 2011.

[15] K. Elgazzar, H. S. Hassanein, P. Martin. DaaS: Cloud-based Mobile Web Service Discovery, Pervasive and Mobile Computing, Vol. 13, pp. 67-84, August 2014, ISSN 1574-1192.

[16] P. Papakos, L. Capra, D. S. Rosenblum. VOLARE: context-aware adaptive cloud service discovery for mobile systems, in Proceedings of the 9th International Workshop on Adaptive and Reflective Middleware (ARM '10). ACM, New York, NY, USA, 2010, pp. 32-38.

[17] J. Zhu, M. Oliya, H. K. Pung, W. C. Wong. LASPD: A Framework for Location-Aware Service Provision and Discovery in Mobile Environments, Services Computing Conference (APSCC), 6-10 December, 2010, pp. 218-225.

[18] Z. Sanaei, S. Abolfazli, A. Gani, M. Shiraz. SAMI: Service-based arbitrated multi-tier infrastructure for Mobile Cloud Computing, 1st IEEE International Conference on Communications in China Workshops (ICCC), 15-17 August, 2012, pp.14-19.

[19] Google Now. https://www.google.com/landing/now/ Last Accessed 23/02/15.

[20] Apple Siri. http://support.apple.com/en-ie/HT4992 Last Accessed 24/02/15

[21] Q. Wang, R. Deters. SOA's Last Mile-Connecting Smartphones to the Service Cloud. Proceedings of the IEEE International Conference on Cloud Computing (CLOUD), Bangalore, India, 21-25 September, 2009, pp. 80-87.

[22] R. K. Lomotey, R. Deters. Reliable Consumption of Web Services in a Mobile-Cloud Ecosystem Using REST, IEEE Seventh International Symposium on Service-Oriented System Engineering (SOSE), Redwood City, California, USA, March 26-28, 2013, pp. 13-24.

[23] J. Sankaranarayanan, H. Hacigumus, J. Tatemura. COSMOS: A Platform for Seamless Mobile Services in the Cloud, 12th IEEE International Conference on Mobile Data Management (MDM), 2011, pp. 303-312.

[24] S. N. Srirama, M. Jarke, W. Prinz. MWSMF: a mediation framework realizing scalable mobile web service provisioning, in Proceedings of the 1st international conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications (MOBILWARE '08), Brussels, Belgium, Article No. 43.

[25] L. Sun, H. Dong, J. Ashraf. Survey of Service Description Languages and Their Issues in Cloud Computing, Eighth International Conference on Semantics, Knowledge and Grids (SKG), 22-24 October, 2012, pp.128-135.

[26] V. Suraci, S. Mignanti, A. Aiuto. Context-aware Semantic Service Discovery, 16th IST Mobile and Wireless Communications Summit, 1-5 July, 2007, pp. 1-5.

[27] M. Klusch, B. Fries, K. Sycara. Automated semantic web service discovery with OWLS-MX, in Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems (AAMAS '06). ACM, New York, NY, USA, pp. 915-922.