Mobile Cloud Application Models Facilitated by the CPA

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

Abstract – **This paper describes implementations of two mobile cloud applications, file synchronisation and intensive data processing, using the Context Aware Mobile Cloud Services middleware, and the Cloud Personal Assistant. Both are part of the same mobile cloud project, actively developed and currently at the second version. We describe recent changes to the middleware, along with our experimental results of the two application models. We discuss challenges faced during the development of the middleware and their implications. The paper includes performance analysis of the CPA support for the two applications in respect to existing solutions.**

Keywords: mobile cloud, applications, services, user experience

I. INTRODUCTION

Mobile cloud computing is a paradigm that aims to overcome the limited resources of mobile devices, such as battery capacity, processing power, and storage. By moving the responsibility for complex tasks from the mobile device into the cloud, demanding applications can be executed there, with the results delivered to the mobile device. Mobile cloud computing can also be seen as the use of cloud based applications and services from mobile devices to the benefit of their owners. Various cloud services and applications deliver their functionality to mobile devices either through an app installed on the device, or through the web browser. The use of cloud resources from mobile devices has resulted in new computing models being made available to mobile users.

Various applications synchronise files across the various mobile devices of the user so that they can be accessed from each device. Changes to a file on one device can be reflected on all the other devices. Dropbox [1] is one example of this model: cloud storage is used to store user's files, and then each mobile device can retrieve the files from Dropbox using an installed application. Similar services include Google Drive [2] and Microsoft SkyDrive [3]. Another example is Apple iCloud [4], which pushes files purchased from the iTunes store onto all of the user's "iDevices", or, additionally, the user could play media files from the cloud, without storing them on the mobile device at all. Many users also have social networking accounts such as Facebook [5] and Twitter [6], and upload media files to these services as a form of cloud storage. One benefit is that the limited storage space on the mobile device is saved. However, all of these services work in isolation. If a user has accounts with several of these providers, all files must be maintained separately. The user would have to upload files from the mobile device to each service individually using different applications, which costs time, money, and energy.

The mobile cloud can also be seen as a platform for demanding computations. Mobile applications with computations that cannot be performed on a mobile device due to their resource requirements can be offloaded into the cloud, executed there, with the results returned to the mobile device. Examples of this approach include cloudlets [7], which use virtual machines running on local infrastructure near the mobile devices to run user applications, which are then displayed on the mobile device. Application partitioning [8] uses a graph model to break a mobile application up into components, which are then distributed to nearby computing nodes. Code offload techniques [9, 10] run application code in the cloud, with resulting object states (in object oriented languages) being returned to the mobile device.

Execution of computationally long and intensive operations, such as large dataset processing and mathematical calculations, can also run on the cloud, with appropriate results returned to the mobile device. In this way, applications would not have to waste the battery and processing capacity of the mobile device, nor would it be at risk from interruptions such as being killed in low-memory scenarios or accidental shutdowns.

Our active work is on a project known as the Cloud Personal Assistant (CPA), which we introduced in a previous work [11]. It forms the backbone of a mobile cloud middleware we are developing, known as Context Aware Mobile Cloud Services (CAMCS), also introduced in a previous work [12]. Each user of CAMCS is given their own instance of a CPA, which can perform tasks given to it by the user, by using

MOBILWARE 2013, November 11-12, Bologna, Italy

DOI 10.4108/icst.mobilware.2013.254197

mobile cloud services. The tasks are described using a mobile thin client application, before being sent to CAMCS middleware, which forwards them to the CPA instance of the user. In our current work in this project, we have examined how to use the CPA to enhance two popular mobile cloud applications, synchronisation of files and data intensive processing.

This paper introduces the implementation of file synchronisation tasks among service providers using the CPA, with the aim of overcoming the mentioned shortcomings of the traditional applications which work in isolation. We also present our implementation of a data intensive processing task on an XML dataset pointing out the role of the CPA. For each, we discuss the implementation challenges and lessons learned.

The remainder of the paper is structured as follows. Section two describes the aim of the CAMCS middleware and the CPA along with its current development state. Section three describes our implementation of the file synchronisation and data processing models with the CPA. Section four details the challenges in implementing this functionality. Section five contains the results of our experimental implementations. Section six includes the related work, and the conclusions are found in section seven.

II. CAMCS AND THE CPA

The CAMCS middleware is a mobile cloud solution, and hence, is hosted on cloud servers. Cloud-based servers provide computing resources for consumers, which can include hardware resources (CPU time, memory, storage, networking capacity), developer resources (application platforms, tools and APIs), and software resources (user software with graphical user interfaces, normally accessed through a web browser). For mobile cloud, we leverage the resources offered in the cloud so that resources not available on the mobile device can be used from the cloud servers - in other words, the cloud resources are delivered to the mobile device as services. This normally requires the mobile device to have a continuous, high-quality network connection to cloud servers.

We now briefly describe CAMCS and the CPA from our previous works. The CAMCS middleware is being developed to provide an integrated user experience for mobile cloud applications. This means that the difficulties of running mobile cloud solutions, such as time/energy costs, and network disconnections to name a few, have lessened significance on the user experience of mobile cloud applications. In addition, the use of the software is seamless for the end-user (part of the general public with no IT experience), and it intelligently responds to the state of the user and the mobile device. The thin client, which can be installed on a mobile device and provides communication between the user and CAMCS middleware, embraces this philosophy, whereas other approaches to mobile cloud, some of which mentioned in the introduction, have not. The user experience aspect of the CAMCS middleware is very important for the implementation of the file synchronisation and data processing models; the difficulties described of implementing mobile cloud solutions have a detrimental impact on both.

The CPA is the backbone of CAMCS. It is an active assistant that performs tasks for the user with mobile cloud services. It represents the user and their tasks in the cloud. It contains a discovery service to find cloud services for performing tasks set by the user. Once a user uploads a task from the thin client on the mobile device, discovery takes place to find an appropriate service. Once found, the CPA contacts the service with user-provided parameters, and waits for the result from the service. When the result is produced, it stores it until the user is ready to receive it on their mobile device. If the user became disconnected during this time, the execution and result of the task are safe in the cloud. As it has been completely offloaded, the mobile device is free for other work - see Figure 1.

Since the publication of our previous works, the CAMCS and CPA have undergone further development. The CPA is now a component of the CAMCS middleware; in our previous work the CPA was a standalone middleware [11]. The discovery service functionality will no longer be the responsibility of the CPA itself, but will be a component of CAMCS. The discovery service will take input from the context processor [12], with the aim of using context to enhance the quality and functionality of services discovered.

The most significant architectural change from our first version is the replacement of the MySQL database for storing information with MongoDB, a NoSQL database. MongoDB uses a document store for data - all information is stored in individual documents. This makes querying for data an easier task. The data itself is stored in JavaScript Object Notation (JSON) format. To contrast with the first version of the CPA,

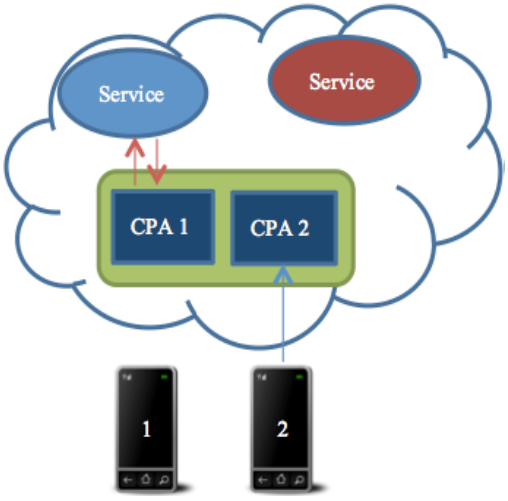

Figure 1: The mobile cloud provides computing services to mobile clients. Here, mobile devices 1 and 2 have their own instances of a CPA within the CAMCS middleware, which work with mobile cloud services, and deliver the results to the mobile devices.

there were separate MySQL database tables for user information (name, email address, password), CPA information (references to current and previous tasks), and task information (name, WSDL file location, result). To get this information into our Java-based middleware, cross-referencing using ID numbers or join queries would need to take place across multiple tables to bring related information into the result set.

With MongoDB, all information for each individual user of the middleware is located in one document. We simply execute one query to the document store for the user by their ID or username, and all their information, including CPA details and task information, are simply returned together in one document. If necessary, projection can take place to exclude data that is not required from being returned.

Aside from fewer queries required by the developer, other advantages also stand out. We exchange data between the Android-based mobile thin client and the CAMCS middleware in the cloud using RESTful web services, with JSON being the format of the data exchanged. We could simply insert this raw JSON into the MongoDB database, without any other overhead such as object conversions for Object-Relational mapping (although as it is easier to work with objects, so far we have not done this in practice). In addition, one of the central ideas of NoSQL databases is evident as mentioned above, no join queries are needed. In a cloud environment, join operations for relational databases can be difficult to implement and scale due to the distributed, or shared, nature of the data storage. When all information is stored in one document in NoSQL databases, joins are not required.

Our mobile thin client has also been updated with features to support the newly added functionality of the middleware, which will be discussed later.

III. FUNCTIONALITY MODELS

We now introduce the two applications described in this paper, which have been implemented as new features of the CAMCS middleware and the CPA, along with their advantages compared with existing solutions.

A. File Storage/Synchronisation

The first of the features added to the CPA allows the user to send files from their mobile device to different cloud service providers. Within the mobile thin client application, the user is given a choice to add their details for different provider accounts. This involves selecting from a list and authenticating with the selected provider, to give the CAMCS middleware access. Once the user has authenticated on the mobile device, the authentication keys used to access the accounts on the users behalf are sent from the mobile thin client to the CAMCS middleware in the cloud.

A user can upload a file from the mobile device using the Android share feature. This allows the user to select which of the provider accounts they have added would they like to send the files to. The user can always select file storage providers such as Dropbox or Google Drive. If at least one of the selected files is an image or video, it will also provide the option to upload to Facebook - Facebook only supports upload of image and video files. After the user has selected the accounts, the files are sent to the CAMCS middleware in the cloud, using a RESTful web service. Once the files have been received at the middleware, they are passed to the user's CPA, which will then send the files to the selected accounts - see Figure 2.

The advantage of such a feature is that if, for example, the user, possibly a company representative, wants to upload files such as promotional material to multiple social networks such as Facebook and Google+ to reach all possible consumers, they no longer have to spend resources such as time, money, and energy on their mobile devices individually uploading to each provider manually. Previous solutions in this regard upload files to each provider individually from the mobile device, using up the described resources during the upload to each provider. Taking advantage of this feature offered by CAMCS, they need only upload the file once to the CPA, which takes care of sending the files to the different providers. The resources are only used once for a single upload operation. If the user has client software for the providers on their desktop PC's, laptops, or mobile devices, the files will be synchronised to them via a push operation. As a result, the current implementation does not feature a download synchronisation to the mobile device as files may be duplicated, wasting more resources. Evaluation and some of the challenges in implementing this will be discussed in the next section.

B. Data Processing

One fundamental aspect of the CPA is that it can carry out work for the user asynchronously. The user can send it the details of some task to be completed, at which point the mobile device disconnects from the CAMCS middleware, and the work continues, with the results saved for when the user is ready to receive them.

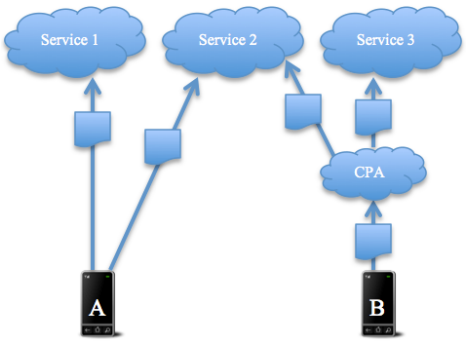

Figure 2: Rather than wasting resources uploading a file twice to two different services individually (device A), the user uploads the file once to the CPA (device B), which then sends the file to each of the user's service accounts.

One area where this may be particularly useful is intensive data processing, especially if it is expected to take a large amount of time. In our previous work [11], we implemented a solution, where we had the CPA perform database queries on relational databases running on Amazon RDS. The CPA would wait for the query to be executed and save the result set for the user. While it did work, it was tricky to implement well, due to the nature of the different types the query could take, and the simplicity of the form based user interface not being intuitive for the novice end-user to specify what they want. This time we decided on another direction; rather than taking data from relational databases and setting up the required authentication and connections, we decided to perform some processing on scientific data. A scientist could set a task to carry out some data processing on sets of experimental results, and get the result later.

In the absence of scientific data formats and software programs for various fields, we settled on processing data from XML datasets. We were able to find some publically available datasets on the website of the Computer Science and Engineering Department of the University of Washington [13]. These range from data on protein sequences, to data from NASA on star systems. These tended to be large in size, so we decided on experimenting with one dataset called Mondial, which contains information on different countries around the world, compiled from the CIA world factbook, and is smaller in size. We developed a RESTful web service, as a separate application deployment from the CAMCS middleware that could carry out statistical calculations on this data. To enable this, the Apache Commons Math library [14] was included in this service.

The flow of this work is as follows: the thin mobile client is used to specify the location of the data by URL, and to specify the type of processing they want to carry out from a list (in this case statistical). The data is sent to the CAMCS middleware, which hands them over to the CPA. It examines the task information, and it can see the user has requested statistical calculations, so it contacts our cloud statistical service, passing it the URL to the XML dataset. At this point in time, the CPA already knows the services available. The information is passed to the calculation service. A CAMCS call-back URL that the service should use to pass the result back is also passed. The service carries out the processing on the data (it calculates statistics like the mean and mode on population data for all cities in the countries part of the dataset), before calling back to the middleware with the result data, which the user can fetch when they are ready. The middleware marks the user's data processing task as complete - see Figure 3.

Another feature is that the CPA can provide real-time status updates on the progress of the data processing. The mobile thin client contains a record of the offloaded processing task, and when they open it, the CPA feeds status updates to the thin client.

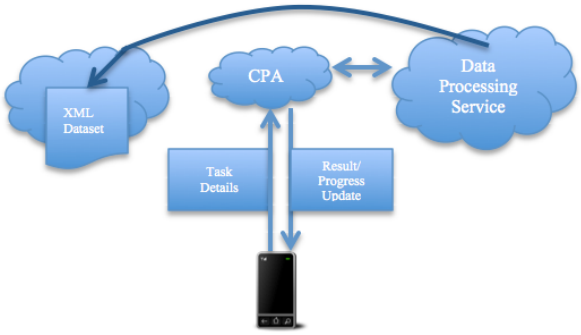

Figure 3: The user sends the data processing details, including the URL of the XML dataset to the CPA. The CPA then contacts the cloud data processing service with the details, which begins the processing. The result is sent back to the CPA. The user can also receive progress updates.

Of note is a difference between how we implemented our statistics web service compared to the database service in our previous work. The statistics service is a RESTful web service. In our previous work, the RDS service was a SOAP based web service. One of the difficulties encountered with that approach, was that for the long running calculation, the Apache CXF software used at the CPA to contact the web service, would time out while waiting for the result. Apache CXF does include an asynchronous call mechanism to overcome this. However for the REST approach, even though being easier to develop with, does not feature an asynchronous web service call. To avoid time outs, we implemented the call-back feature.

Advantages to this approach include the useful aspect that the web service will have libraries available to it that may not be present on the mobile device. As mentioned previously, we used the Apache Commons Math [14] library to calculate the statistics. Other scientific libraries available include JScience [15], which were also included but were not used. It would not be as trivial to calculate these statistics if done on the mobile device without these libraries.

Other advantages include the fact that the user does not need to sit waiting for a specific piece of client software to complete the data processing, which may be prone to interruptions. The user can set the task with the CPA, and go on to do other work or leave the office for the night and turn off the local equipment, which may have otherwise been left on and used for the processing task. They can then check in with the CPA on the go with the mobile thin client for progress when required.

There are some difficulties and limitations in this approach that will be evaluated in the next section.

IV. IMPLEMENTATION CHALLENGES AND EVALUATION

In implementing the two application models, we identified several difficulties and drawbacks to the approaches we discussed, as well as areas for

improvement in the API design for mobile devices. We will now evaluate the work with respect to these for each of the two applications.

A. File Synchronisation

1. OAuth Authentication

In order to access the accounts of the different cloud service providers, the user needs to allow CAMCS access by first authenticating themselves, and granting permission for the required operations. For all of the service providers we worked with for the middleware, OAuth [16] is the security access scheme employed. At development time, Facebook and Google used OAuth version 2, and Twitter and Dropbox used OAuth version 1 (by the time this paper was written, Twitter provided OAuth 2 support). In both versions of OAuth, the application requesting access to the user account with the service provider is given access credentials in the form of an access token/key/secret. With OAuth 1, a second access token/key/secret, sometimes called a "value", is also provided. When the application needs to access the user's account, they present the access token (and the value in the case of OAuth 1) with the request, and if the credentials are valid, the application is granted access. The main benefit of this approach is that the application that wishes to use the service provider on behalf of the user does not need to know/store the user's username and password for that service.

The flow of authentication and gaining an access key for most applications is as follows. The developer has to register their application with the service provider, and obtain an application key. The CAMCS was registered with each of the service providers we used and we obtained a key. When the user wishes to allow the application access to the provider, in Android, they are redirected from the mobile application to the website of the service provider through a WebView, presenting the application key. The user logs in with his/her own username and password. The user is then given a choice to grant access to the application for various operations (sometimes called "scopes"). Once the user has granted access, the mobile application is called back with the access key (and the value in the case of OAuth 1). These are then stored on the mobile device for future use.

The issue here for our middleware system, is that the mobile device does not need the access credentials at all. The CPA operating in the cloud is the entity that will be working with the service providers; therefore the CPA needs to be given the access credentials.

If this were a web application accessed from the desktop PC browser, the web application would receive the call-back and store the credentials. In this case, the credentials would be sent straight to the CPA. This however would not be optimal for the user experience. Asking the user to leave the mobile thin client, and open a corresponding website with a browser for our middleware to perform the authentication would defeat the purpose of it being a mobile thin client application.

To overcome this, we implemented a RESTful web service on the middleware. When the user has authenticated with the service provider on the mobile thin client, the access credentials are sent from the thin client to the CAMCS to be stored with their account. The CPA can then access credentials with the user's account details stored on the cloud middleware, to carry out operations with the service providers - see Figure 4.

2. Service Provider APIs

This again relates to authentication with the service providers. To implement the authentication flow, we are using the Spring Android project, which uses components of the Spring Social project. They simplify the work required for authentication with service providers. The developers of Spring Social have only implemented official support for Facebook and Twitter authentications using OAuth. There are several community driven projects for other providers, such as Dropbox and Google. None of these community driven projects have been ported to the mobile platform, and their implementation remains solely focused for use with Spring Social on web applications. These could be ported to be compatible with the Spring Android components, but this requires some development effort.

Rather than doing this, we decided to use the Android APIs available from Dropbox and Google. This involves downloading JAR files from the different service providers, packaging the thin client with them, and using them in the code to carry out the authentication flows. Ultimately, we ended up having several JAR files; those for Spring Android, Spring Social, Spring Social Facebook, Spring Social Twitter, Dropbox, and Google Play services. The file sizes of these start to build up quickly. Moreover, all these

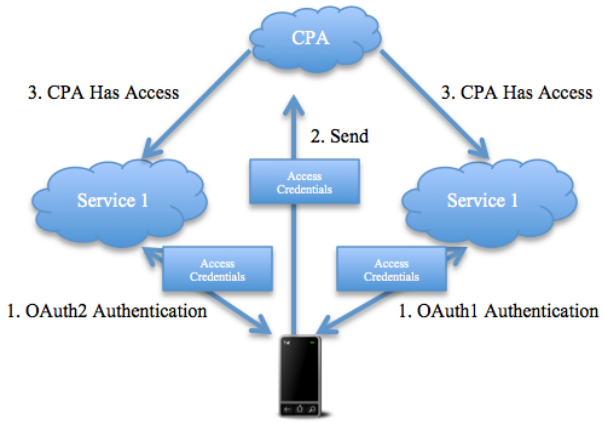

Figure 4: To get the access credentials to where they are needed with the CPA, the user must authenticate for each service on the mobile device (normally through a WebView). The keys are then sent to the CPA using a RESTful web service. The CPA can then access each service on the user's behalf.

services transfer data in JSON format, but these JAR files often contain different versions of JSON parsers, which all do the same thing, taking up even more space while doing so. One cannot set each of the APIs to use a single JSON parser of choice and remove the rest - see Figure 5. All the service providers authenticate using OAuth tokens, but each provider seems to implement the authentication flow differently, rather than using some standard. Spring Social aims at resolving this, but as described, only supports Facebook and Twitter, relying on community projects for other implementations, which have not been readily ported to Spring Android.

To authenticate with Google, we use Google Play services. This contains an AccountManager, which is supposed to again provide common features for getting access tokens, but, like Spring, requires community built authenticator modules for the different providers. Aside from the expected need for different interfaces for the different features of the different service providers, it would be much easier for developers, for the common task of authentication, if there was a standard API that would work for all out-of-the-box, since they all use OAuth authentication. In addition, if the user of the middleware wanted to add another provider not already supported that uses OAuth, our mobile thin client would need to be modified to support each new provider's different implementation of the authentication flow, so extension becomes difficult. If a standard API existed, they could add new service provider accounts without the need to modify the mobile thin client. The different APIs take more time to learn and implement, and increase the size of the applications deployed to mobile devices because of the required JAR files.

3. Synchronisation From Service to Device

As described, the current implementation does not implement a download mechanism to synchronise files from the various cloud services to the mobile devices. Many service providers already implement a push mechanism; this will automatically send a file uploaded to the provider, down to all the other devices that use a native application. On the mobile device, this would be a waste of resources if files were downloaded more than once both from the CPA and the native application.

If this were to be implemented, it would require a means for the CPA to be aware of when the user uploaded a file to the service from other sources, such as a web browser. This could be achieved by polling, but this introduces extra traffic to the service provider, which would be pointless if no new files or updates have been added to the service provider since the last poll. A better solution would be an event notification API, which could alert interested parties, such as the CPA, when a new file has been added or of any update to existing files. This requires the service provider to implement such an API. As an example, Dropbox provides the sync API, which allows notifications to be

Figure 5: Screenshot from the Eclipse IDE of all the required JAR files for the Android thin client for each service provider. Duplicated functionality can be seen; jackson JAR files are for JSON parsing, the json-simple JAR is required by another file but contains the same parsing functionality as Jackson.

sent after events such as new files being added occur. However, the API currently only exists for native Android and iOS implementations; the ability needed here is to inform a third party on the users behalf, in this case the CPA, so that it is aware of the file state at the providers.

B. Data Processing

1. Service Extensibility

The main question facing the development of the data processing is how to expand its operation, and make it easier to invoke. As it stands, our statistics service will only work with an XML dataset that shares the same schema as that of the Mondial XML dataset we developed it against (or any specific dataset we specifically develop a service for). The statistical service we developed for it has to parse the XML dataset, and expects to find certain tags and attributes that can be used for calculations. While there may be other datasets representing similar data (for example, ethnic population data on European countries) that uses the same markup, it is still a fragile service.

It may be prudent if a scientist who has data to process could easily specify their own calculations that they are interested in to a service, so that the service could readily work with different XML schemas. They could be uploaded to the CPA from the mobile device and be sent on to the processing service. The calculations would have to specify what data to work with, and what calculations should be performed on it. Ideally, the user should be able to express the desired calculation on the mobile device thin client interface. This may be achievable with cooperation from those who develop software specifically for data processing of large formatted data. If this was not the case, as it is now, a different service would have to be developed for each different XML schema, limiting the scale of the data processing service.

2. Discovery of Data Processing Services

Currently, our data processing service runs in another web application completely separate to the CAMCS middleware. This is because different service providers

will provide their own services for processing different types of data; it is not something that the CPA can do itself at present. In this situation, services that can perform different processing on data need to be known to the CAMCS middleware. The service needs to be able to describe what exactly it does, what data it expects, and how it will return the results. In addition, the CAMCS middleware needs to be able to compare the dataset and instructions passed by the user with these external services to find what service will match the request.

At the moment, locations and types of services are hardcoded onto the CAMCS middleware, so it knows where to find a specific set of services that carry out what calculations on what datasets. Clearly, a discovery solution would be of use here, which is part of our future work. In addition to describing common service attributes such as message formats and endpoints, it would need to describe how to specify the required data for the calculations (such as which mathematical calculations to perform on what specific data in the XML document).

V. RESULTS

Experiments were carried out to evaluate the timing performance of both the file synchronisation and the data processing functionality of the middleware, which we now present.

A. File Synchronisation

To evaluate the file synchronisation performance, over 5 different runs, the time to upload a PNG image file of size 112KB was measured - see Table 1.

Specifically, we measured: the time taken to upload the image to the CPA from the mobile device, and the time for the CPA to upload the image to Facebook and Dropbox. The mobile device used was a Samsung Galaxy S3, connected to the Vodafone Ireland operator. The mobile – CPA upload took place on a HSDPA+ cellular network connection. The CAMCS middleware was running with an Apache Tomcat version six servlet container on the cloud server. The cloud server is located within University College Cork, Ireland, and features a 1.7Ghz CPU and 2GB RAM. The timing data was collected from logging statements placed in the code. The upload of the image file to Dropbox and Facebook from the CPA took place in sequential order. If we had utilised threads to do this concurrently, the total time would have been smaller, the mobile to CPA communication time plus the maximum of the server upload times.

To compare this with the performance of uploading with the individual Android apps, we measured over five runs the time to upload the same image with the native Facebook and Dropbox Apps with the HSDPA+ connection. This timing data was obtained with a stopwatch, from the time the upload (or equivalent) button was pressed on each app, to the time when the notification came through that the upload was complete. The timing is less accurate as a result, but the greater duration is still clear - see Table 2. Clearly it takes even more time, energy and money, since the user has to upload the image twice using two different apps, whereas with our CPA the user only has to do this once.

In Table 1, the only times to user has to wait on their mobile device are the times for the Mobile – CPA communication in column two. So the total time for the images to reach the service providers from the mobile device in column 5 is not the total time the user has to spend waiting for upload on their mobile device. Contrast this with the total result in Table 2. The user has to manually upload with the applications for each individual provider, so the total time in the fourth column is the total time the user must spend waiting for uploads to complete.

Table 1: The time in seconds over 5 runs to upload a 112KB image from the mobile device to the CPA, and subsequently from the CPA to Facebook and Dropbox.

Run	Mobile – CPA (s)	Facebook (s)	Dropbox (s)	Total (s)
1	2.255	2.749	1.621	6.625
2	4.395	3.25	1.523	9.168
3	1.935	2.011	1.533	5.479
4	2.63	2.094	2.671	7.395
5	1.25	2.106	1.584	4.94

B. Data Processing

The data processing service was deployed in the same Apache Tomcat six servlet container and cloud server as the CAMCS middleware. The XML parser used was XMLPULL [17]. For a comparison test, we implemented a small Android application with a service, which would carry out the same XML parsing as the server. The Android XML parser is the aforementioned XMLPULL parser we used on the server, so the comparison is fair in this regard of implementation. For the cloud service, we used the XPP3/MXP1 implementation [18] of the XMLPULL parser, as we believe this to be the same implementation found on the Android platform, due to the same package structure (the other implementations have a different package structure to the version found on Android).

Table 2: The time in seconds to upload a 112KB image to Facebook and Android using the individual native apps.

Run	Facebook App (s)	Dropbox App (s)	Total (s)
1	15.0	4.1	20.1
2	5.1	2.6	7.7
3	6.9	3.7	10.6
4	7.0	3.3	10.3
5	6.9	5.2	12.1

Before the tests were run, Tomcat was restarted. We measured the time with logging statements in the code to fetch the XML file, the time to parse the XML, and finally the total time (which included the time for preparing the XML parser, converting the XML file to a String for the parser, and the calculation of the statistics), over 5 runs - see Table 3. The majority of the time is spent on conversion of the XML to a String. The parse time decreases with each parse after the first. Another test was carried out by restarting the server again, and the same trend of decreasing parse time was repeated after an initial longer time for the first run. The larger the dataset in size, the larger the number of XML nodes that will need to be parsed, which will take up more of the limited memory if done on the mobile device. This will also take more time, and more energy from the battery.

As previously described, we implemented a small Android application to carry out the same XML parse as the cloud data processing service for comparison purposes. This ran a service thread, which executed the same Java code on the cloud service on the same XML dataset - see Table 4. The results show that the XML fetch over the cellular network connection took longer than the cloud service, as one would expect due to the poorer quality connection. The XML parse consistently took around half a second, and did not show the same decreasing time trend as the cloud service. Interestingly, this means that the first two runs of the parse on the cloud server were actually slower than the mobile device. We believe this to be an implementation detail of either the Java Virtual Machine running on the cloud server, or the Tomcat servlet container.

Both implementations use a Java InputStream for the fetch. The bytes from the stream are then read and converted into a String for the parser input. However, when the Android client fetched the XML dataset, it also brought along formatting characters, specifically,

Table 3: The XML fetch and parse times in seconds for the data processing cloud service along with the total time.

Run	XML Fetch (s)	Parse (s)	Total (s)
1	0.969	1.114	4.585
2	0.387	0.676	3.777
3	0.604	0.43	4.119
4	0.384	0.084	3.37
5	0.359	0.038	2.354

Table 4: The XML fetch and parse times in seconds for the data processing Android test application along with total time.

Run	XML Fetch (s)	Parse (s)	Total (s)
1	0.88	0.505	6.11
2	0.74	0.425	7.6
3	3.53	0.44	12.365
4	2.09	0.43	11.315
5	2.815	0.435	11.735

newline characters (\n) and whitespace. This interfered with the tokeniser of the XML parser, and they had to be removed from the String (using a String replace method) before the XML string was passed to the parser. This removal operation took around four seconds each time, and is the biggest contributor to the total time on the Android device. As a result, the total time was always longer on the Android test application, even for the two runs where the parsing operation was quicker than the cloud service. This removal process did not need to be performed on the cloud service; no newline or whitespace characters were fetched in the InputStream.

Once the work is complete, the call-back is made to the CAMCS middleware, which forwards the result to the user's CPA. The CPA then sends an email informing the user the result is ready, and then can then view the result in the thin client application on the mobile device. Future works here includes a push notification service from the CPA to the mobile device.

With the cloud service, the mobile user does not need to upload data from the mobile device over the network connection once the data URL is specified. No energy is used up on the mobile device for the parse, and the parse is unaffected by interruptions on the device; the device is also free for other work.

VI. RELATED WORK

Few middleware's exist offering mobile cloud services. One example is a middleware by Wang and Deters [19] that aims to optimise the consumption of web services from mobile devices. This involves the conversion of requests from RESTful JSON based, to XML SOAP based for contacting SOAP services. As JSON is more lightweight it is easier for the mobile client to consume. The mobile communicates with the middleware using JSON. XML based SOAP responses are converted to JSON before being sent to the mobile device. The work also tries to combine services by a mashup mechanism, feeding the result of one service as an input to another. The user must know something about the SOAP/REST nature of the service beforehand, and know where to find the WSDL file. Our system will be based on service discovery so the technicalities of the service are hidden (service type, WSDL locations).

Another work by Flores et al [20] provides a middleware which can plug in adapters to make requests to different web services. The request for a service is sent to the middleware, which will then substitute an appropriate adapter to make the service call. It is not known how the developer of mobile apps actually calls the middleware and specifies their request. The approach is limited by the adapter solution, where different adapters may have to be developed for each service. As our approach aims to use a discovery service, we believe our approach to be more scalable, and we will provide an interface for mobile app developers to request services from the CAMCS.

In both cases, our middleware aims to provide a range of services to the user that take advantage of cloud-based infrastructure and services, rather than just a middleware for a single purpose. CAMCS will be extensible so extra functionality can be plugged in.

In terms of file synchronisation, most mobile apps for this purpose, such as Dropbox [1], Google Drive [2], and Microsoft SkyDrive [3], all work in isolation, and do not provide support for uploading to multiple services. Our approach goes over that limitation, as CAMCS works with multiple services, and as mentioned, we are saving time, money, and energy by uploading files the CPA once, rather than uploading to each service provider separately.

In regards to data processing, cloud based solutions to data processing are available, especially in the area of big data, but we are unaware of other work in this area from the mobile cloud context.

VII. CONCLUSIONS

In this paper, we presented how the CAMCS middleware with the CPA can be used to implement two mobile cloud applications, namely file synchronisation and data processing. We presented the current development state of the middleware, along with some of the changes to support these applications. We then went on to evaluate and discuss the challenges in implementing these functionalities. For the file synchronisation, these include OAuth security implementation issues and heterogeneous APIs for different service providers. For data processing, they include scalability and calculation specifications.

We presented timing results for both applications. For the file synchronisation, the timing results showed fast performance over the cellular network. When contrasted with uploading files individually using native Android apps, the time-savings were evident.

For the data processing application, the time to fetch and parse XML datasets on the server was also quick, with results comparable to or faster than the same parsing operation on our Android testing application.

We highlighted how effective the middleware can be as an enabler of these two applications, compared to existing approaches. For file synchronisation, the CPA can save resources such as time, energy, and money, by quickly performing the synchronisation with different service providers; resources need not be wasted uploading multiple times to different service providers from the mobile device. For data processing, heavy processing work can be offloaded to the CPA, so as not to use up the hardware resources of the mobile device. This can relieve the need for dedicated software running on the mobile or desktop that needs to be left on for long periods of time, with progress updates available on the move. Network disconnections or dead batteries will interrupt neither application after the initial upload to the CPA. Tasks can progress normally.

In our future work we will continue with implementation of the CAMCS middleware and the CPA, which will involve adding context processing, and subsequently service discovery. We will also be exploring how CAMCS and the CPA can be used to facilitate real-time applications that may have low-latency requirements.

The challenges we highlighted in this paper such as authentication, API design, and lack of data standards for processing, will be of crucial importance going forward as mobile cloud development increases, and mobile client software adopts the paradigm.

ACKNOWLEDGMENT

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

REFERENCES

[1] Dropbox. https://www.dropbox.com/
[2] Google Drive. https://drive.google.com/
[3] Microsoft SkyDrive. https://skydrive.live.com/
[4] Apple iCloud. https://www.icloud.com/
[5] Facebook. http://www.facebook.com/
[6] Twitter. https://twitter.com/
[7] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The Case for VM-Based Cloudlets in Mobile Computing", IEEE Pervasive Computing, 2009; 8(4), pp. 14-23.
[8] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso, "Calling the cloud: Enabling mobile phones as interfaces to cloud applications", Middleware 2009, pp. 83-102.
[9] 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.
[10] B-.G. Chun, S. Ihm, P. Maniatis, M. Naik and 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.
[11] M. J. O'Sullivan, D. Grigoras. The Cloud Personal Assistant for Providing Services to Mobile Clients, IEEE MobileCloud, Redwood City, San Francisco Bay, USA, 2013, pp. 477-484.
[12] M. J. O'Sullivan, D. Grigoras. User Experience of Mobile Cloud Applications – Current State and Future Directions, Proceedings of the 12th International Symposium on Parallel and Distributed Computing, Bucharest, Romania, 27-30 June, 2013, pp. 85-92
[13] XMLData Repository, Department of Computer Science and Engineering, University of Washington
. http://www.cs.washington.edu/research/xmldatasets
[14] Apache Commons Math Library.
http://commons.apache.org/proper/commons-math/
[15] JScience Library. http://jscience.org/
[16] OAuth. http://oauth.net/
[17] XMLPULL Parser. http://www.xmlpull.org/index.shtml
[18] XPP3/MXP1 XMLPULL Parser Implementation.
http://www.extreme.indiana.edu/xgws/xsoap/xpp/mxp1/
[19] Q. Wang and R. Deters, "SOA's Last Mile-Connecting Smartphones to the Service Cloud", Proceedings of the 2009 IEEE International Conference on Cloud Computing, 1632969, IEEE Computer Society, 2009, pp. 80-87.
[20] H. Flores, S. N. Srirama and C. Paniagua, "A generic middleware framework for handling process intensive hybrid cloud services from mobiles", Proceedings of the 9th International Conference on Advances in Mobile Computing and Multimedia, Ho Chi Minh City, Vietnam, 2095715, ACM, 2011, pp. 87-94.