SAM: A Semantic-aware Middleware for Mobile Cloud Computing

Harun Baraki, Corvin Schwarzbach, Stefan Jakob, Alexander Jahl, and Kurt Geihs
Distributed Systems Group
University of Kassel
Kassel, Germany
{*baraki, schwarzbach, jakob, jahl, geihs*}*@vs.uni-kassel.de*

***Abstract*—Mobile Cloud Computing (MCC) requires an infrastructure that is combining the capabilities of resource-constrained but mobile and context-aware devices with that of immovable but powerful resources in the cloud. Application execution shall be boosted and battery consumption reduced.**

Our goal is to relieve mobile devices by enabling applications to cooperate transparently with components provided by third parties remotely, locally, or both. For this purpose, we developed SAM - the Semantic-aware Middleware - and the SAM Store. The latter resembles a registry. It allows component providers to publish semantically enriched interfaces and to offer corresponding local and remote implementations. SAM makes use of the SAM Store to match local interprocess communication (IPC) messages transparently against local and remote components. This architecture facilitates the integration of powerful, but resource-intensive remote components, which may be prioritized over local ones, and fosters various patterns of cooperation that demonstrate useful solutions in the area of MCC. By means of a realistic scenario, our evaluation proves the efficiency with respect to latency and practicality.

***Keywords*-IPC; Mobile Cloud Computing; Ontologies;**

I. Introduction

Leveraging the capabilities of stationary, virtually unlimited cloud resources to support the battery life and processing power of resource-constrained mobile devices is the major target of MCC. Current approaches focus on assisting single applications [1], [2], [3], [4]. They boost an application by offloading and executing its resource-intensive parts in a virtual machine (VM) in the cloud. This first requires to partition the application, to monitor and identify the resource-intensive parts and to synchronize application code, state and context information with the allocated VM.

So far, interactions and interdependencies between cooperating applications have only received limited attention, although they form an ideal basis for outsourcing tasks to third parties in the cloud. IPC messages that ask for complex calculations, image processing, weather data, and so forth, can be redirected to remote endpoints capable of responding to these. Locally available components may serve as fallback solutions if communication link failures occur or if servers are down. The core of our contribution is to map local IPC to network IPC transparently. It empowers mobile devices to reduce the overall execution time of resource-intensive scenarios since remotely participating components may scale independently of one another. A specific advantage of the introduced IPC is that it is not bound to components registered locally. The message-oriented middleware SAM makes initially use of its local registry. If a target component is not found on the local device, the remote SAM Store will be queried for matching counterparts. Additional semantic information supports unambiguos matches. The result may be a downloadable component or remote endpoints satisfying the constraints given by the client application.

The concrete procedure, the middleware SAM, and the SAM Store are presented in section III. The interplay of SAM and the SAM Store allows applications to cooperate in various ways. Three different types of cooperation that are particularly suited for interweaving Mobile and Cloud Computing and relieving mobile devices, are shown in section IV. Henceforth, they are referred to as *distributed IPC patterns*. An example for a distributed IPC pattern and its benefits are shown in section II. Section V evaluates SAM and the IPC patterns and demonstrates their efficiency and suitability for MCC. A discussion of related work can be found in section VI. Conclusions are given in section VII.

II. Motivating Example

While the latest mobile devices and their operating systems are well equipped for common tasks and execute them in acceptable time, many resource-intensive scenarios and use cases are not realized or rare due to different restrictions. For example, let us consider a mobile application that is searching for certain faces or objects within the photo albums of the user. The application would read and analyze each image on the mobile device, which could be hundreds or even thousands of files. Additionally, if the images are not available locally, in order to reduce storage consumption and allow central access, they have to be downloaded first from the cloud storage. In summary, the task would ask for high capacity in respect to computational power, memory, storage, bandwidth, and energy, as well as for access rights for reading and processing the images.

The outlined scenario encompasses the following parties. Firstly, there is the image analysis app *IMA-APP* that requests initially images from the photo album app *PH-APP*. While the former belongs to the company *IMA*, the latter is a product of *PH*. The images are confidential and intellectual

property of the user who relies on the services provided by PH. In most MCC approaches, a device clone would run as a VM in the cloud [1], [2]. It would neither belong to IMA nor to PH but would be associated with the user and operated by a third party. Offloading a face detection task to the device clone would require a synchronization of images and application states. The image synchronization has to take account of images available on the device and in the PH cloud.

While in the purely local case described in the first section images are present on the local device and in the PH cloud, in the latter case they also occur in the device clone of the user. Storage and bandwidth consumption and overall costs rise due to an additional VM and its extra synchronization efforts. Further QoS dimensions like trust, availability and error rate could suffer as a result too. Hence, the decision to offload has to be taken carefully.

A user-centric view where Virtual Machines or containers are assigned to individual customers and their devices, prevents interacting applications and their providers to weigh in and to exploit the full potential of their own cloud infrastructure. In the aforementioned example, the *PH-APP* is aware of the PH cloud endpoint and, thus, could offer partner applications to offload and execute their data-intensive operations remotely in the PH cloud. The *IMA-APP* would execute then its analysis function in a sandbox environment in the PH cloud and extract and return the URIs of images that match the submitted query. Images that are only available on the mobile device and are not yet synchronized, could still be analyzed on the mobile device or synchronized first with the PH cloud. Customers would not be in need of own VMs and images are not required to be transferred first to be analyzed. The evaluation section takes up this example and shows clearly the benefits of our IPC patterns. Before describing SAM and the distributed IPC patterns, the following section explains current IPC techniques on mobile devices. They deliver the starting point for our middleware.

III. SAM Middleware

A. Overview

Considering popular operating systems for mobile devices, message passing turns out as the common IPC technique. On Apple's iOS, where applications run in sandboxes, messages can be sent through XPC (Cross Process Communication) connections between extensions and their hosting applications. If third-party applications shall be addressed, data can be framed into messages and sent through BSD sockets.

In Android, communication between independent processes is typically performed by means of Intents. An Intent may include the following fields: a component name, an action, data, categories, extras, and flags. A dispatched Intent listing a component name can be assigned by the operating system to the associated application. The component will be started and may then execute the specified action on the data, mostly URIs, and process the extras given as key-value pairs. Intents without component names are called implicit Intents. In this case, Android determines the target application by taking the action, the MIME type of the data and the mentioned categories into account. They are matched against so-called Intent filters declared by the installed applications. If several applications comply with the Intent, the user has to select one of them.

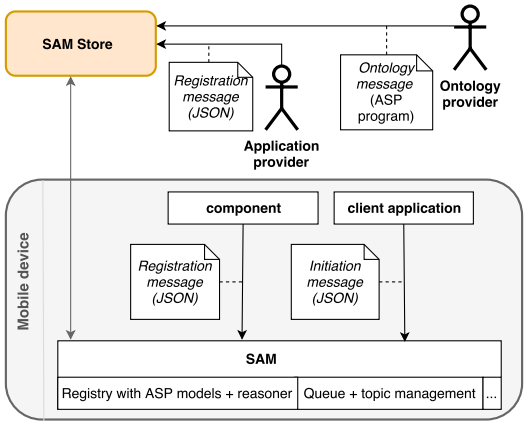

Figure 1. Message types and respective parties

At an entirely local level, SAM's IPC resembles to a certain extent Android's Intent mechanism. The message-oriented middleware SAM matches incoming messages to registered and eligible application components. However, SAM distinguishes between registration, ontology, initiation and usual IPC messages and provides various interaction modes. Registration and initiation messages are directly addressed to SAM, and, in certain circumstances, to the SAM Store too. Registration messages are intended for registering and de-registering application components, their interfaces and their endpoints. In this way, components can offer their functionality to client applications. Client applications may find and potentially start one or more matching counterparts through initiation messages. Thereafter, SAM will establish a message-oriented connection between these. Depending on the selected interaction mode, applications are sending each other IPC messages through message queues or by publishing and subscribing to topics. Registration messages may also reference ontologies they comply with. Referenced ontologies must exist at the SAM Store where they can be downloaded by SAM. This allows SAM more precise matches and motivates application developers to adhere to common interfaces and ontologies. Figure 1 illustrates the different message types and the parties using them. The following sections explain each message type and the middleware SAM. Section IV will switch over to the distributed IPC patterns that are particularly important for MCC.

B. Registration of Components

Whenever applications are installed that offer functionalities to other applications, they need to register these through registration messages at SAM. To support the distributed IPC patterns introduced in section IV, a one-time registration at the SAM Store is required too. Messages sent to and received from SAM adhere to the JavaScript Object Notation (JSON). A registration message lists at least the component name, its path, an interface description, and the interaction mode. An application may register different components. The path is required to start the appropriate component when needed. The interaction mode distinguishes between single and multiple instances of the regarded component, and between queue- and topic-based communication. A single instance and queue-based communication would lead to a many-to-one communication through message queues. If for each inquiring partner a separate component instance shall run, for example to support stateful sessions, *multiple instances* has to be selected. Listing 1 shows a registration message fulfilling the minimum requirements.

```
1  {"sam:component": "com.ph.app.storage",
2   "sam:path": "/apps/ph/storage.comp",
3   "sam:interaction": {
4     "sam:message": "sam:queue",
5     "sam:instances": "sam:single" }
6   "sam:interfaces": {
7    "sam:types": {
8     "xs:schema": {
9      "-xmlns": "http://.../XMLSchema",
10      "-targetNamespace": "http://.../img/storage
       /monitoring",
11       "xs:element": [
12        {"-name": "getImageCountRequest",
13         "xs:complexType": {
14          "xs:sequence": {
15          "xs:element": {
16          "-name": "mimeType",
17          "-type": "xs:string"
18         }}}},
19         {"-name": "getImageCountResponse",
20          "xs:complexType": {
21           "xs:sequence": {
22           "xs:element": {
23           "-name": "count",
24           "xs:simpleType": {
25           "xs:restriction": {
26           "-base": "xs:integer",
27           "xs:minInclusive": { "-value": "0" }
28     }}}}}},...]}},
29     "sam:interface": {
30      "-name": "img.storage.monitoring",
31       "sam:operation": [{
32        "-name": "getImageCount",
33        "sam:input": {
34         "-element": "getImageCountRequest"},
35        "sam:output": {
36         "-element": "getImageCountResponse"
37 }},...]}}}
```

Listing 1. Registration message for a many-to-one communication

To enable compliance checks, a description of the interface is also part of the registration message. The interface description follows to a great extent the schema of the Web Service Description Language (WSDL) and can be seen as a JSON representation of the abstract part of a WSDL file. This allows an automatic derivation of the JSON interface description by making use of WSDL generators for the respective programming language. An excerpt of a generated interface description is shown in line 6 to 37 in listing 1. Namespace aliases are generally resolved into an ASCII representation of their effective character string. For the sake of clarity, listing 1 keeps the aliases.

```
1  { "sam:component": "com.ph.app.storage",
2    // registration message:
3    ...
4    // automatically added information:
5    "sam:permissions": [{
6    "sam:permName": "storage.files.meta.read",
7    "sam:permDescription": "Allows app to..." },
       ...]}
```

Listing 2. Permissions granted for an installed component

Listing 2 contains the permissions that are needed by the previously mentioned component and that have been granted by the user during its installation process. SAM captures and stores this information in its registry since partner applications are only permitted to initiate a communication with a component, if they hold its privileges too.

```
1  { "sam:component": "com.ph.app.storage",
2    "sam:path": "/apps/ph/storage.comp",
3    "sam:interaction": {
4     "sam:message": "sam:topic",
5     "sam:instances": "sam:single" },
6    "sam:topic": "/img/storage/monitoring" }
```

Listing 3. Registration message for a topic

The topic-based communication is of importance when it comes to well-defined interfaces that are additionally described through an ontology. Henceforth, they are referred to as *semantic interfaces*. Semantic interfaces have to be registered through *registration* and *ontology messages* (for example by ontology providers) at the SAM Store. Subsequently, the SAM Store checks first for cyclic dependencies, inconsistencies and naming conflicts with other referenced and registered semantic interfaces before publishing it. On the local device, the topic field of a registration message designates the semantic interface that the application's component implements (listing 3). If not already available on the device, the description of a referenced semantic interface is downloaded by the SAM middleware. The following section introduces ontology messages after describing how interface descriptions are represented internally in SAM.

C. Enriching Interfaces with Semantics

Semantic interfaces can be referenced, implemented and extended by application providers. Thus, the use of an ontology is essential to avoid ambiguities and to enforce strict typing also in a semantic sense. To enrich a structural description of an interface with semantic information, each term appearing in the description should have a counterpart

in an ontology. SAM assumes unique names and abandons semantic annotations. Therefore, the XML Schema and the SAM interface vocabulary also exist as ontology (listing 4).

```
% Variables in capital letters
% Derived head :- if body true
% N = Name, T = Type, S = Sequence,
% O = Operation, I = Interface
xs_primitive(xs_string).
xs_primitive(xs_integer).
xs_primitive(xs_anytype).
...
xs_type(N,T) :- sam_input(N,T).
xs_type(N,T) :- sam_output(N,T).
xs_type(N,T) :- sam_fault(N,T).
...
xs_complexType(S) :- xs_sequence(S).
xs_sequence(S) :- xs_sequence(S,N,T), xs_type(N,T).
...
% Each type is connected to an operation or
% an interface and is used as input,
% output or fault.
sam_input(N,T) :- sam_input(O,N,T), sam_operation(O).
sam_output(N,T) :- sam_output(O,N,T), sam_operation(O).
sam_fault(N,T) :- sam_fault(O,N,T), sam_operation(O).
...
% A component requires a name, an interface
% and an endpoint or path.
sam_component(N) :- sam_component(N, I, E), sam_interface(I), sam_path(E).
```

Listing 4. Excerpt of the SAM and XML Schema vocabulary in ASP

SAM uses declarative logic programs formulated in ASP (Answer Set Programming) to represent knowledge [5] and to reason about it [6]. In contrast to Web Ontology Languages (OWL), ASP supports the demanded unique name assumption, which is needed to automatically derive ASP code from the interface description. Listing 5 illustrates the automatically generated ASP program when parsing and translating the interface of listing 1. Combining it with the available ASP program of listing 4 will derive additionally the respective heads in listing 4.

```
sam_operation(ism_getImageCount).
sam_input(ism_getImageCount,
  ism_getImageCountRequest,
  ism_mimeSequence).
sequence(ism_mimeSequence, ism_mimeType).
type(ism_mimeType, xs_string).
sam_output(....).
...
```

Listing 5. Generated ASP representation of the getImageCount operation

However, whenever new terms are introduced by application providers, they should also be described in ASP or should be connected with existing ontologies stored as ASP programs. This task is their own responsibility and is not mandatory. Yet semantic knowledge avoids misunderstandings and improves matching search requests or initiation messages against registered components. This not only allows an exact match of client goals and provided components, but enables also subsumption and intersection matches. Using ASP reasoning in SAM, components are even composable to achieve more complex goals. An example of a simple ASP program that could be supplied by an application provider for the aforementioned interface is shown in listing 6. It defines image/jpeg and image/gif as MIME types and then states that directories may contain other directories and files and that the type of the latter one is a MIME type.

```
ism_mimeType(jpeg).
ism_mimeType(gif).
...
ism_folder(F) :- ism_folder(D),
  ism_contains(D, F), not ism_file(F).
ism_file(F) :- ism_folder(D),
  ism_contains(D, F), not ism_folder(F).
ism_mimeType(M) :- ism_file(F),
  ism_hasType(F, M).
ism_mimeType(M) :- sam_component(N),
  ism_supportMimeType(N, M).
...
```

Listing 6. Added semantics for the registered semantic interface

The program may also include further attributes that may or must be set by components when registering as an implementing party through JSON messages at SAM. It must be considered as an extended schema for registration messages. Typically, the attributes shall describe the context of a component and allow SAM more precise matches. For example, line 3 in listing 7 shows a constraint that requires each component to support at least three MIME types. Since line 3 directly connects this attribute with the component, it has to be listed on the top level of the JSON registration message. This allows SAM to automatically link an entry like *"ism:supportMimeType":"png"* to the component itself. Attributes for operations and messages have to be listed accordingly on lower levels. Line 7 in listing 7 states that a component supports the MIME type image/jpeg by default. A component can invalidate it by explicitly stating the opposite: *"ism:supportMimeType":"-jpeg"* in its registration message. Defaults are well suited for setting standard values which can be overwritten when necessary. A minimum standard can be enforced through constraints.

The main reason for using ASP instead of OWL [7] is that OWL is based on Description Logics (DL) [8] and, hence, does not support integrity constraints and closed-world reasoning. In addition, OWL is monotonic. Updating existing rules or overwriting initial default assumptions necessitates first discarding the current knowledge base. Frameworks like WSMX [9] combine OWL with rule-based formalisms grounded in logic programming. However, logical conclusions that an OWL reasoner would draw from an ontology differ from those that would be obtained when using a logic program engine. ASP roots in logic programming, allows non-monotonic reasoning and provides

defaults for expressing standard representations. Public or given default knowledge may be overwritten by application providers by local versions without causing inconsistencies for any participating party. This is essential in dynamic and heterogeneous environments.

```
% At least three supportMimeType have to be
% set, otherwise, the constraint is violated
:- sam_component(N), #count{ism_supportMimeType(N
    , M)} < 3.

% Mime type supported by default
% unless stated otherwise
ism_supportMimeType(N, jpeg) :- sam_component(N),
     not -ism_supportMimeType(N, jpeg).
```

Listing 7. Added constraints and default values for components implementing the semantic interface and referring to it as a topic

SAM utilizes the given ontologies, registration messages and their representations in ASP to respond to search queries (initiation messages) and to execute compliance and permission checks. Further attributes and mechanisms of the registration procedure are presented in section IV.

D. Initiating the Communication

As with registration messages, there are two different types of initiation messages, depending on the interaction mode. An application may address a specific component by indicating the respective component name in the initiation message or it publishes, by means of a topic name, to a set of components implementing a certain semantic interface. In the former case, a queue-based communication is established by SAM, provided that the required privileges are given. IPC messages have to comply with the schema defined in the registration message of the component.

An initiation message that asks for a particular topic, may list several constraints. Otherwise, IPC messages published to a topic are broadcasted to all locally available components that implement the interface, adhere to the default values and that are accessible to the requesting application with respect to the permissions. Constraints may refer to generally admitted attributes, such as meta information, Quality of Service (QoS) and Quality of Experience (QoE) parameters and the number of addressed components. For example, an application may prefer among all components implementing a given interface, one that was installed or used recently by the user. In addition, constraints may pertain to attributes and features that are specific to the topic and, thus, to the semantic description of the interface. In the example of listing 7, support for certain MIME types can be demanded. SAM will match them against the values of the *"ism:supportMimeType"* attribute of the registered components. For other topics, for example in the context of weather apps, the precision of a weather component and its metric can be of interest.

As indicated in section III-C and listing 7, components do not need to register attributes as long as they stick to their default values. Unless otherwise specified in the initiation message, SAM also expects that only components shall be addressed that comply with the default values defined for the attributes. According to listing 7, SAM would publish IPC messages to components that support at least image/jpeg and two further MIME types. As a general rule, application developers do not require deep ontology expertise. They can be fully relieved when generating JSON registration message templates for a given semantic interface.

IV. Distributed IPC Patterns for MCC

Leveraging SAM and the SAM Store, different patterns for distributed IPC are implementable. The following sections present three patterns that, depending on the scenario, support common goals of MCC approaches, i.e., the reduction of the device's energy consumption and the acceleration of a task's execution. Thereby, the patterns abandon the requirement for device clones running in the cloud or cloud resources assigned to single customers and their application collection. Instead, this paper presumes the architecture proposed in [10], according to which application providers that aim at utilizing MCC, have to operate or lease their own cloud infrastructure. During runtime, parts of their mobile application can be executed in their cloud.

For a better understanding, the patterns are explained by means of scenarios and examples. They do not follow the typical structure of software design patterns, since this work emphasizes and evaluates the techniques behind them.

A. Pattern: Location Transparent IPC

Application providers initially register applications and their components at the SAM Store where users will search, find and download them. In case components are missing that are needed by local applications, the middleware SAM informs the user and suggests components available for download at the SAM Store.

However, under certain circumstances, for example, if remote information and access is needed either way, it may be advisable to provide a functionality through a remote endpoint instead of a downloadable application. In that case, a registration message sent to the SAM Store contains the *"sam:endpoint"* attribute and omits *"sam:path"*. Additionally, the interaction mode must be set to *"sam:queue"*. There is a variety of reasons for the latter restriction. Firstly, API keys and tokens are often required for remote endpoints. They differ for every provider. Secondly, an online publish-subscriber system would pose a central bottleneck. In contrast to that, the queue-based and encrypted communication takes directly place between the local SAM and the remote SAM running on the provider's server, avoiding any delays and insecurities through third parties. In this scenario, the SAM Store can be considered as a name resolver where SAM obtains and caches the registration message as the name resolution. For client applications location

transparency is achieved through addressing components by their names (see *initiation messages* in section III-D) and sending messages through queues managed by SAM.

Compared to service-oriented architectures, which mostly rely on HTTP and are mainly suited for single reply-response messages, *Location Transparent IPC* can be tailored to any desired message exchange pattern without extra configuration. Its asynchronous data transmission is predestined for unreliable communication channels. To support this essential feature further, a provider may supply a component through the *"sam:path"* attribute, but additionally list an endpoint by means of *"sam:endpoint"*. In this case, the locally available component only serves as a fallback if the remote endpoint is not reachable.

B. Pattern: Mobile Code IPC

The *Mobile Code IPC* pattern is particularly suited for data-intensive applications. Many of the current mobile applications already integrate cloud services to allow users to access centrally administered data and functions and to make additional storage available. With respect to the scenario presented in section II, some cloud storage providers offer a Web or REST service to third-party applications to access metadata and to upload and download data on condition that an appropriate token or key is given. However, they do not support in-situ execution of third-party operations in the cloud. The data has to be downloaded first, which leads to a significant rise of latency and bandwidth and energy consumption. Furthermore, mobile devices are not well equipped for processing big amounts of data.

To enable code migration to a third party in the cloud, the *Mobile Code IPC* pattern proceeds as follows. Starting from a client application that analyzes images or extracts and aggregates information from user files in the cloud, the client application has to send first an initiation message to SAM. The initiation message will ask for components that implement a certain semantic interface and, thus, for components that subscribed through registration messages to the respective *"sam:topic"*. This could be, for instance, *cloud.storage.img*. The registration messages have to adhere to the ASP program linked to the interface, in particular, by providing the requested attributes. Default and additional attributes list the platforms, libraries and the programming and script languages that are supported by the component's cloud counterpart. In our experiments, for instance, we set the default platform to Java VM, supporting Java 1.8 and 1.9. Alternatively, containers can be made available, or interpreters can be provided. For each domain, other platforms and libraries might be preferred as default or minimum configuration.

The client application finally sends its IPC message, which contains the mobile code and parameters for its execution, to matching components by publishing it to the topic. Each component equips the incoming IPC message with

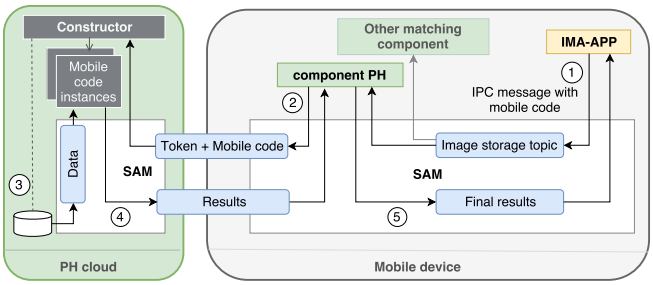

Figure 2. Mobile Code IPC pattern applied on the example of section II

information required for the remote access, such as tokens and device identifiers (step 2 in figure 2). Subsequently, each component applies the *Location Transparent IPC* to send the extended IPC message through SAM to the remote endpoint of the respective provider. In this way, tokens and credentials are hidden from client applications. They merely need the permission for the given scope.

The receiving component on server side checks authorization information and feeds, when access is granted, a queue with the requested data in the form of messages. Depending on the configuration, one or more instances of the mobile code are constructed. They are registered as subscribers for the queue and process the incoming messages through callback methods. Further access to server resources is restricted, except writing results to a queue. The results are returned to the provider's component on the local device, which may remove information or postprocess the messages before returning it finally to the client application.

C. Pattern: Local2Distributed IPC

The last pattern presented in the scope of this paper is applicable for typical MCC scenarios where resource-intensive parts of an application are offloaded to the cloud. For offloading, we apply the MOCCAA framework presented in [10]. As mentioned before, it presumes that partitions of an application may only be offloaded to the application provider's own cloud. In case of MOCCAA, objects and methods are annotated therefore by the application developers, and properties, like e.g. resource registry or server addresses, are set by them. Annotated objects are automatically proxied by a configurable evaluator which decides, before executing an invoked and marked method, if offloading would be beneficial or not. If offloading appears, the object's state and the parameters are transferred by means of an intermediate and compressed format to the cloud and the method is invoked. The applied approach is called Delta Synchronized Remote Method Invocation (DRMI) [10]. A device clone is not needed.

By the inclusion of SAM, even methods can be offloaded that send and receive IPC messages to third-party applications. Assuming that a client application applies the *Location Transparent IPC* or the *Mobile Code IPC* pattern on the local

device, it may use it the same way in the cloud. Since the operator of the cloud is also the application provider, SAM can be pre-installed and equipped with sufficient rights to access the SAM Store and to start components.

V. Evaluation

The evaluation shall demonstrate the efficiency and expediency of SAM in comparison with a characteristic MCC approach and with a purely local IPC approach. The example presented in section II serves as evaluation scenario. The application *IMA-APP* shall return the URL of all images that contain a face from a front view. The images are managed by the photo gallery application *PH-APP*, which synchronizes images with the *PH cloud*.

The *Mobile Code IPC* pattern is applied in SAM, which would even allow to include several image providers in parallel, without adapting the client application *IMA-APP*. For the characteristic MCC approach described in section II, henceforth named *CMCC*, the user has a private VM with 4 x 2.6 GHz CPU, 8 GB of memory, and SSD storage, which has all applications pre-installed. The local device is simulated by a notebook with a 2 x 2.6 GHz CPU, 8 GB of memory, and SSD storage. The *PH cloud* has 12 nodes - each containing a six-core processor with 2.6 GHz and 64 GB of memory - at its disposal. The SSD storage is connected by means of an Infiniband switch. The latencies between all three parties were 20ms on average. The average bandwidth between the *CMCC VM* and the *PH cloud* were 12 MB/s, from client to the *CMCC VM* and the *PH cloud* 1.2 MB/s, and from the *CMCC VM* and the *PH cloud* to client 11.4 MB/s. The average size of an image was 1578 kB. The results show the arithmetic mean of ten iterations. SAM, Java 1.8 and the OpenCV library were available on all parties. SAM itself makes use of clingo [6], genson[1] and the OpenHFT[2] Chronicle Engine and Queue.

The first experiment expects a private person or a company as a user who prefers cloud-based storage to relieve mobile devices and to enable access by different devices via Internet. All images are stored in the *PH cloud*. According to table I, downloading images and analyzing them on the mobile device leads only to acceptable times for few images. For 10 MB around 3.77s are needed. Downloading and analyzing 1 GB of images requires 382.58s on average. If the mobile device starts the corresponding method of the *IMA-APP* in the *CMCC VM*, times are reduced to 1.47s and 140.73s respectively. The times include the download and analysis of the images and the return of the results to the mobile device.

Using SAM and applying the *Mobile Code IPC* pattern, it is possible to run multiple instances of the uploaded mobile code directly in the *PH cloud*. In the first configuration, the maximum number of instances is set to one. The second configuration, named *PH-Splittable*, makes use of a receiving component that first checks which nodes are idle. In this experiment, all nodes are free and, thus, 12 instances are constructed. Applying the pattern reduces the expended time to 0.38s for 10 MB and to 36.46s for 1 GB and one node, including the transfer of the mobile code and the response. *PH-Splittable* decreases the response time further to 0.07s and 3.19s respectively. In all experiments the analyzing code stayed the same.

Table I
Experiment 1: All images centrally stored in the PH cloud

	Time in seconds			
	Local	**CMCC**	**PH**	**PH-Splittable**
10 MB	3.769	1.464	0.377	0.067
50 MB	18.905	6.921	1.732	0.145
100 MB	37.722	13.632	3.594	0.331
250 MB	96.823	37.773	10.81	0.903
500 MB	187.267	67.928	18.123	1.541
750 MB	285.692	106.126	27.486	2.315
1000 MB	382.579	140.726	36.458	3.188

Table II
Experiment 2: Images available locally

	Time in seconds			
	Local100	**CMCC100**	**Local-PH**	**CMCC-PH**
10MB	0.723	0.622	0.372	0.313
50MB	3.675	2.684	1.855	1.356
100MB	7.172	5.172	3.683	2.688
250MB	20.998	16.748	9.897	8.389
500MB	35.267	25.828	21.455	16.801
750MB	59.042	42.676	29.801	21.442
1000MB	79.479	56.426	36.368	26.279

In the second experiment, images are already distributed between different parties. The first column in table II, *Local100*, shows the times in the event that all images are stored and analyzed on the mobile device. There is no delay by downloading images first. However, this eliminates the advantage of centrally accessible data, enhanced battery life, and significantly faster applications. Assuming that 50% of the images are stored and analyzed on the mobile device and 50% in the *PH cloud* (see column *Local-PH*), the benefit would be limited. The processing power of the mobile device constitutes the major bottleneck. It needs around 36s for 500 MB. Accordingly, the columns *CMCC100* and *CMCC-PH* list the times when images are stored and analyzed fully and partly in the CMCC VM. Here again, a single private VM cannot reach the capabilities and advantages of shared clusters and central cloud storages.

The evaluation has just demonstrated one example for one pattern. The pattern allows providers of cloud storages, crowd and sensor data, and others to serve as platform

[1]https://owlike.github.io/genson/
[2]https://github.com/OpenHFT

for third-party applications, which may apply, for example, aggregation, analysis and planning functions. Examples and evaluation results for further patterns are planned for an extended version of this paper.

VI. Related Work

In [1] and [2], a device clone running as a VM in the cloud is employed to switch between a local and a remote execution. Since the clone is fully synchronized, a local IPC on the device would be executable on the VM clone, too. However, the speed-up would be restricted by the capacity and configuration of the used VM since all applications run on the single machine. The same argument applies to other MCC approaches. Kosta et al. introduce in [3] their framework ThinkAir that allows developers to execute an application's methods in parallel on different VMs. Yet, IPC is limited to the respective VM and is not distributed. In [4] different parts of a single application may be offloaded to different servers in the cloud. The remote communication between the individual parts of the application is established by the framework. Unfortunately, IPC is not considered by the authors and, hence, would also be bound to the actual VM. Other highly cited works in the area of MCC also do not leverage IPC to support mobile devices and enable new application fields. In the scope of cluster and grid computing and in general in distributed systems, message-oriented middlewares are widely used for local as well as distributed communication. They even find use in combination with mobile devices [11]. Nonetheless, IPC is applied in the common way of sending and receiving messages without refinements that support patterns like those presented in this paper. These patterns are enabled by SAM particularly through the assistance by non-monotonic ontologies and a shared knowledge and reasoning platform.

VII. Conclusion

This paper presented SAM, a middleware that enables various IPC patterns for Mobile Cloud Computing. Together with the SAM Store, it unveils a new dimension of scalability and flexibility for mobile applications and breaks the existing barrier of isolated applications and VMs in MCC. In addition, semantic interfaces were introduced that allow the definition of minimum standards and defaults for implementing parties. Client applications may search and communicate with particular components or all components implementing a certain semantic interface. The communication is location transparent and allows leveraging different patterns for realizing well-performing solutions. The current work did not consider version management and software evolution. Both are considered in separate works [12]. Furthermore, feedback loops through ratings and monitoring are planned as future work for the SAM Store. They shall remove inferior components and descriptions and establish robustness. A cost model shall additionally prove the economic feasibility.

References

[1] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 301–314.

[2] S. Yang, D. Kwon, H. Yi, Y. Cho, Y. Kwon, and Y. Paek, "Techniques to minimize state transfer costs for dynamic execution offloading in mobile cloud computing," *IEEE Transactions on Mobile Computing*, vol. 13, no. 11, pp. 2648–2660, 2014.

[3] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Infocom, 2012 Proceedings IEEE*. IEEE, 2012, pp. 945–953.

[4] K. Sinha and M. Kulkarni, "Techniques for fine-grained, multi-site computation offloading," in *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Computer Society, 2011, pp. 184–194.

[5] M. Gelfond and Y. Kahl, *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach*. Cambridge University Press, 2014.

[6] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Clingo = ASP + Control: Preliminary Report," in *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*, M. Leuschel and T. Schrijvers, Eds., vol. 14(4-5), 2014.

[7] S. Bechhofer, "Owl: Web ontology language," in *Encyclopedia of Database Systems*. Springer, 2009, pp. 2008–2009.

[8] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical owl-dl reasoner," *Web Semantics: science, services and agents on the World Wide Web*, vol. 5, no. 2, pp. 51–53, 2007.

[9] A. Haller, E. Cimpian, A. Mocan, E. Oren, and C. Bussler, "WSMX - A semantic service-oriented architecture," in *Proceedingsof the 2005 IEEE International Conference on Web Services (ICWS)*. IEEE, 2005, pp. 321–328.

[10] H. Baraki, C. Schwarzbach, M. Fax, and K. Geihs, "MOCCAA: A Delta-synchronized and Adaptable Mobile Cloud Computing Framework," in *Proceedings of the 8th International Conference on Cloud Computing and Services Science*, 2018.

[11] S. Bagchi, "The software architecture for efficient distributed interprocess communication in mobile distributed systems," *Journal of grid computing*, vol. 12, no. 4, pp. 615–635, 2014.

[12] A. Jahl, H. Baraki, H. T. Tran, R. Kuppili, and K. Geihs, "Lifting low-level workflow changes through user-defined graph-rule-based patterns," in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2017, pp. 115–128.