Moitree: A Middleware for Cloud-Assisted Mobile Distributed Apps

Mohammad A. Khan, Hillol Debnath, Nafize R. Paiker,
Narain Gehani, Xiaoning Ding, Reza Curtmola, Cristian Borcea
Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102, USA
{mak43, hd43, nrp48, gehani, xiaoning.ding, reza.curtmola, borcea}@njit.edu

***Abstract*—This paper presents Moitree, the middleware of our Avatar platform for mobile cloud computing. Avatar leverages cloud resources to support fast, scalable, reliable, and energy efficient distributed computing over mobile devices. Each mobile device is augmented by an avatar, a per-user always-on software entity that resides in the cloud and acts as the surrogate of the mobile device, to the extent possible, thus reducing the workload and the demand for storage, bandwidth, and energy on the mobiles. Moitree provides: (1) a novel middleware which allows unmodified apps to execute seamlessly over mobile/avatar pairs with the provision of offloading computation and communication, and (2) a new programming framework to simplify mobile collaborative app development. The programming framework has two key features: (1) user collaborations are modeled using natural group semantics - groups are created dynamically based on context and are hierarchical. (2) data communication is offloaded to the cloud through high-level communication channels. A prototype of Moitree, along with several apps, has been implemented and evaluated on Android devices and on an OpenStack-based cloud running Android x86 avatars.**

I. Introduction

Execution and communication offloading from mobile devices to their software surrogates in the cloud has proven to improve app response latency, reduce wireless communication overhead and energy consumption at the mobiles, and improve the availability of mobile apps [23], [9], [10], [16], [24]. These surrogates can be instantiated as virtual machines (VMs), containers, or even processes. Microprocessor manufacturers have recently started providing shielded application execution over untrusted cloud platforms [1], thus offering guarantees that the surrogates are truly personal and protected from the cloud providers [6]. Therefore, the converging model for mobile cloud computing assumes that each mobile device of a user is paired with a user-owned and controlled surrogate in the cloud.

This scenario lends itself naturally to mobile distributed computing executed over sets of mobile/surrogate pairs. People collaborate within these apps by forming groups defined by friendship, common interests, geography, etc. Examples of distributed apps include discovering alternative routes to avoid traffic jam/congestion, finding people of interest in a crowd using face recognition techniques (e.g., a lost child), monitoring and stopping the spread of epidemic diseases, and mobile multi-player gaming.

This paper presents Moitree[1], a middleware that facilitates the execution of collaborative apps within groups of mobile users, with each group being represented by a collection of mobile device/surrogate pairs. In addition to runtime support, Moitree provides an API and a set of libraries for developing cloud-assisted mobile distributed apps. While the concepts of Moitree are general and applicable to any distributed mobile cloud platform, we have designed it and implemented it for our Avatar platform [7]. An avatar is an instantiation of a surrogate in the cloud for a mobile device.

Programming over mobile/avatar pairs is different from traditional mobile distributed programming. First, the end points in the computation are device/avatar pairs with different capabilities - mobile devices have multiple types of sensors and user interaction capabilities; avatars have more powerful computation, storage, unlimited power, etc. Programmers need an intuitive and common interface to transparently use these computing end points. Apps need to read sensor/user inputs on the mobiles, but should offload most of the communication/-computation to the avatars, without affecting negatively the user experience. Second, the apps require user collaboration based on natural contexts, such as location, time, social relationships, etc. Therefore, managing the collaboration in real time is important.

The Moitree middleware provides a unified view of the mobile/avatar pairs to the programmers, thus hiding the heterogeneity and complexity of the underlying system. Programmers can use the Moitree API to access resources, without any assumption of where the code is running (mobile or avatar). The Moitree communication API offloads the user-to-user communication from mobile devices to avatars.

User collaboration in Moitree is modeled using group semantics. Groups are formed based on factors such as location, time, and social connections. The key features of the user groups are: *hierarchy*, which allows programmers to naturally organize users into groups/subgroups and manage their collaborations within different scopes; *dynamic group membership*, which updates group membership based on the current context of users (e.g., a group for "visitors of the Statue of Liberty" changes dynamically over time); and *communication channels*, which facilitate the communication among the members in a

[1]The word "moitree" is taken from Bengali, which means alliance/collaboration.

group and offload the communication to the cloud.

We implemented a prototype of Moitree on Android devices and an OpenStack-based cloud running Android x86 avatars. To validate Moitree, we implemented two proof-of-concept apps: one finds a lost child at a crowded event by performing face recognition on the photos taken by people attending the event; the other is a video-conferencing app that allows users to create group hierarchies. The number of lines of code of the app implementations based on Moitree is reduced to less than half when compared to their implementations done in JXTA [11], a platform for peer-to-peer programming, and Android. We also performed experiments with micro-benchmarks on top of our prototype to evaluate the scalability and overhead of the middleware.

The rest of the paper is organized as follows. Section II presents a brief overview of the Avatar system. The Moitree programming model is detailed in Section III, and the design and implementation of the middleware is presented in Section IV. Section V validates the programming model with a proof-of-concept application and shows micro-benchmark results for the middleware. Related work is discussed in Section VI, and the paper concludes in Section VII.

II. Avatar Overview

The Avatar system [7] is motivated by the strong demand for fast, scalable, reliable, and energy efficient distributed computing over mobile devices. A few research efforts have investigated cloud support for mobile distributed computing [16], [14], [21]. These projects focus mostly on enabling communication among mobile users. Other efforts have investigated offloading mobile code execution to the cloud in the context of stand-alone apps [10], [22], [9], [15]. While Avatar leverages high-level ideas from these efforts, its goal is fundamentally different - integrating the benefits of modern mobile devices and cloud computing, to provide a novel platform for distributed collaborative apps (which are inherently peer-to-peer).

In Avatar, a mobile user is represented by one mobile device and its associated "avatar" hosted in the cloud. An avatar is a per-user software entity which acts as a surrogate for the user's mobile device, to the extent possible, thus reducing the workload and the demand for storage and bandwidth on the mobiles. Each avatar is instantiated as a VM (Android x86) in the cloud in order to provide resource isolation and to simplify per-user resource management. Avatars run the same operating system as the mobiles and can thus run unmodified apps or app components (e.g., functions, threads, etc.). Implicitly, they save energy on the mobiles and improve the response time for many apps by executing certain tasks on behalf of the mobiles. The avatars are always available, even when their associated mobile devices are offline because of poor network connectivity or simply turned off. Each avatar coordinates with its mobile device to synchronize data and schedule the computation of avatar apps on the avatar and/or mobile device. A mobile device does not interact directly with the avatars of other mobile users. User-to-user communication always goes through the avatars.

While the avatars could be hosted by any cloud provider, mobile network operators may be particularly willing to offer the service. They have already started to offer cloud services to back up data from mobile devices. At the same time, having information about their customers and their mobile devices, mobile network operators may be able to offer better services with higher efficiency than other cloud providers.

The Avatar platform provides the architecture for cloud-assisted mobile distributed computing, and Moitree offers the required runtime system (i.e., middleware) as well as the API to build mobile distributed apps over Avatar.

III. Moitree Programming Model

This section first describes the key ideas in Moitree and the distributed app execution model. Then, we present an example app, *LostChild*, to illustrate Moitree's advantages in developing distributed apps. Finally, we describe the complete API provided by Moitree.

A. Key Ideas

1) Groups and Group Hierarchies: A *group* is a fundamental unit over which mobile distributed computation is done. A group is a set of users selected and organized based on properties, such as user locations, time, and social connections. Each group is app-specific, and each app can create as many groups as it needs. All members of a specific group, by default, run the same app. A user can be part of multiple groups at the same time.

Groups in the same app can form hierarchies with the groups at lower levels being subgroups of the ones at upper levels, and are maintained in a tree structure. This helps programmers to structure the distributed computation for certain apps. For example, subgroups can be recursively created to solve location-based computations using the divide-and-conquer strategy (e.g., finding a free parking spot in a city).

2) Communication Channels: A communication channel provides high level messaging support and makes it easy to offload communication to the cloud. There are four types of channels: (i) *broadcast* for sending messages to all members of a group; (ii) *anycast:* for sending messages to a random member of a group; (iii) *point2point:* for sending messages to a specific member; and (iv) *scatter-gather:* for sending messages to all members of a group and then receiving answers from some group members as a function of their computation results. The *broadcast* channel is unidirectional, while the other three are bidirectional.

When a function invokes a communication channel on a mobile, the call is intercepted in the middleware and always forwarded to the cloud (avatars) to carry on. User-to-user communication always occurs via the avatars.

While some messages (e.g., point2point and anycast communication) are not persistent, the messages exchanged within the whole group (e.g., broadcast and scatter-gather communication) can be designated as *persistent* by the programmers.

These messages are useful for forwarding data to new comers to the group (e.g., a person who entered the region that defines the group after the group was formed). Persistent messages are stored in the group buffer in the cloud and distributed to new members when they join a group.

3) Dynamic Group Membership: Since people move and their context changes over time, group membership also changes. The dynamic group membership in Moitree shields the programmers from handling group dynamics. The middleware selects and maintains group members automatically based on properties specified by the programmers. For example, a group can be formed for a given geographic region during a specific time interval. The middleware will add/remove group members based on who enters/leaves the region during that interval.

B. App Execution

A distributed app is instantiated by all group members who collaborate within the app. Thus, the distributed app consists of app instances executed in parallel on the set of mobile/avatar pairs belonging to the group members. The distributed app runs in a single program, multiple data style. The app group is formed when the first user (i.e., the initiator) starts its app instance, which in turn invokes a group creation function. This user becomes the owner of the group. Other group members are selected and added to the group automatically by the middleware based on their properties and permissions. The permissions are defined in the users' collaboration policies and specify the conditions under which a user is willing to participate in a group. The app instances at the other group members are created in response to the request for group creation. This is done through events delivered to the Moitree components on the avatars/mobiles.

The app code is the same for each group member. However, due to factors such as resource availability on mobiles and privacy policies, the way in which the computation tasks are partitioned between avatars and mobile devices may differ for different users. Programmers do not have to worry about app partitioning because Moitree hides it from them. Tasks in an app may also differ with respect to different users based on their roles in the distributed computation.

During the execution, instances can send messages using the four communication channels already described. The communication is event-based. Messages are received asynchronously (i.e., similar to the "select" system call in Unix), using callback functions registered by the programmers with the Moitree middleware.

The app instance at the initiator is responsible to delete its groups before terminating. This operation triggers events in Moitree that will terminate app instances at all the other group members.

Let us finally note that Moitree apps invoke the Moitree API for group operations and communication. For everything else, it uses the local platform API (e.g., Android). Finally, it is important to point out that Moitree apps co-exist with native apps on mobile devices.

Code 1: Code running at the parent's mobile/avatar

```
1  searchForLostChild(){
2   MembershipProperties prop = new MembershipProperties();
3   prop.setLocationBound(LOCATION, RADIUS);
4   prop.setTimeBound(TIME_FROM,TIME_TO);
5   Group group = Avatar.createGroup(null,prop,false,
        LIFETIME);
6   ReadCallback callback = new ReadCallback(){
7     public void scatterGather(ChannelID cid,Message msg){
8       //msg contains results sent by participants
9       //add result to potential trajectories
10      //update user with latest trajectories
11      if (ChildFound){
12        DONE=true;
13      }
14    }
15    ... ... ... //other channel callbacks
16  }
17  group.setReadCallback(callback);
18  ChannelID cid = generateChannelID();
19  group.scatterGather(cid, childImage);
20  while (!DONE){ //block }
21  group.deleteGroup(group.credential());
22 }
```

C. Code example: LostChild App

To illustrate the development of a Moitree app as well as the main features of the middleware, let us consider a *LostChild* app. A parent could use it to locate a child lost in a crowded area using the child's photo(s) to search for the child among recent photos taken by nearby mobile users. Based on the places where the photos that contain the child have been taken, the app builds a real-time trajectory of the child's movement to aid the parent find her quickly.

Code 1 shows the code executed by the app instance running at the parent's mobile/avatar. Lines 2-4 specify the context properties that must be satisfied by group members (i.e., region and time interval), and the group is created in line 5. The middleware is responsible for group formation and dynamic maintenance. Alternatively, a group may also be created from a specific list of users. Before adding a user to the group, the middleware verifies that its app/group collaboration policies allow it to become a group member (e.g., the user may refuse to collaborate due to location privacy reasons).

Lines 6-16 show the callback function implementation for Moitree message communication. In this example, we have only one channel type, *scatter-gather*. The callback function is registered with the middleware in line 17. Then, in line 19, the app sends a photo of the child to all the group members and then waits for responses until the child is found. When responses are received from individual users, the scatter-gather callback is invoked and the potential child trajectory is updated and presented to the parent. When the parent confirms that the child was found, the app deletes the group and terminates (lines 11-12 and 20-21).

Code 2 shows the app processing done at the participating users. The app is activated by the middleware at the participating users when they are added to the group (at group formation time or later). The app starts with the *onCreateGroup* method in line 1. A reference to the group is passed as a parameter

Code 2: Code running at the participants' mobiles/avatars

```
1  onCreateGroup(Group group){
2   ReadCallback callback = new ReadCallback(){
3     public void scatterGather(ChanelID cid, Message msg){
4       Image image = msg.getData();
5       //run face recognition with image
6       //if child image is recognized, set the location
            and time associated with the matching photo in
            result
7       group.scatterGather(cid, result);
8     }
9     ... ... ... //other channel callbacks
10  };
11  group.setReadCallback(callback);
12 }
```

Code 3: Creating hierarchical groups

```
1  //Top-Down Group Creation
2  Group newark = Avatar.createGroup(null,
       membershipProperties,true,lifetime);
3  Group zip07102 = Avatar.createGroup(newark,
       membershipProperties2,true,lifetime);
4  ... ... ...
5  //Bottom-up Group Creation
6  Group zip07102 = Avatar.createGroup(null,
       membershipProperties,true,LIFE_TIME);
7  Group zip07103 = Avatar.createGroup(zip07102.getParent()
       ,membershipProperties,true,LIFE_TIME);
8  Group newark = Avatar.createGroup(zip07102.getRoot(),
       membershipProperties,true,LIFE_TIME);
9  zip07102.changeParentGroup(newark);
10 ... ... ...
```

in this method. A *scatter-gather* callback is implemented in lines 2-10 and registered with the middleware in line 11. The callback is invoked when the photo of the child is received from the parent. Face recognition is performed for all the photos of this user that satisfy the group location and time criteria to see if they match the child image (line 5). If any of them does (line 6), its associated location and time are sent to the parent over the *scatter-gather* channel (line 7).

This example shows that Moitree makes mobile distributed computing concise and easy to program and understand.

D. API Description

The Moitree API, presented in Table I, is divided into three classes. The *Avatar* class provides methods for group creation and joining. The *Group* class offers methods for group management (e.g., leave/delete the group, create subgroups) and group communication. The *MembershipProperties* class is a utility class that has methods for specifying the group properties. The same Moitree API is used independent of the execution place (i.e., mobile or avatar).

1) Group Creation, Membership, and Deletion: In addition to the *prop* parameter already discussed, *createGroup* takes three other parameters: *parent*, *enableLeader*, and *lifetime*. The *parent* parameter is used for group hierarchies. If it set to *null*, it means the group is the potential root of a group hierarchy; however, groups are not required to be part of hierarchies. More details on group hierarchies are presented in III-D2.

Some groups may need leaders to implement functions such as consensus or scheduling among their members. If the *enableLeader* is set to *true*, then the user who creates the group becomes the leader. If the leader leaves the group, the middleware selects a new leader. Currently, Moitree selects a random user as a new leader, but other leader selection policies could be implemented. The method *sendToLeader* allows any user to send messages to the leader, without the need to use the ID of the leader, which improves fault-tolerance. The method *getLeader* is normally used to determine if the local user is the leader of the group; if yes, the app needs to run leader-specific functionality.

The *lifetime* parameter specifies that a group has to be deleted by the middleware in the absence of any group communication for the *lifetime* duration. In this way, Moitree deallocates the resources associated with a group when the group is not active. The apps receive an exception and terminate.

Groups are dynamic in that members can come and go. Users can join a group using the *joinGroup* method. The user invoking this method must know the group ID and have the right credentials. For example, a new user is invited to a multi-player mobile game and is provided the group ID and the credentials. A user can leave a group by calling *removeFromGroup*. Currently, this method is used only to remove the user invoking it. However, we plan to explore if this method should be allowed to remove other users; such functionality could be useful for group creators and/or leaders.

2) Group Hierarchies: By default, if a user belongs to a group, it is also a member of the parent group. For simplicity, we do not allow overlapping sibling groups or overlapping groups on different branches of the hierarchy tree.

A user in a subgroup can get a reference to the parent group using the *getParent* method or to the root group using *getRoot*. Similarly, a user in a group can get a references to the child subgroups (i.e., one level down in the hierarchy) using the *getChildGroups* method. Group hierarchies can be created top-down or bottom-up as shown in Code 3. The top-down approach is used when the hierarchy can be defined before the app starts. The bottom-up approach allows for dynamic creation of hierarchies at runtime. The example in Code 3 illustrates the group hierarchy for an app that finds free parking spots around a given destination. The region around the destination can be divided in several levels of sub-regions, with each sub-region associated to a group in the hierarchy. The processing can be done in parallel for each subgroup, and the results can be gathered at the initiator.

3) Group Communication: Apps may use any combination of communication channels as needed. Each channel is instantiated by the middleware upon its first invocation in the app. Each app instance can use its own scatter-gather channel. To distinguish these channels, they have unique ChannelIDs.

The communication on all channels is asynchronous. Anyone can send a message anytime and receive messages through callback methods. Each sending communication channel is paired with a receiving callback method.

Group-related API - Avatar Class

Method	Description
Group *createGroup*(Group parent, MembershipProperties prop, Boolean enableLeader, Double groupLifetime)	Creates a group with members selected based on membership properties *prop*; if *enableLeader* is true, the group has a special member with leader role. *groupLifetime* specifies how long the group should exist without receiving any messages from the members.
Boolean *changeParentGroup*(Group newParent)	This method is used to re-organize the group tree.
Boolean *onCreateGroup*(Group group)	Callback method executed by Moitree to start the app for a user when the user is made member of the group. Parameter *group* is supplied by the middleware.
Boolean *joinGroup*(User user, Group group, Credential c)	Called to join a group after the group was created. The credential ensures that the user has appropriate permissions to join the group. Credentials are generated when a group is created and distributed to the members as part of group creation.

Group-related API - Group Class

Method	Description
void *removeFromGroup*(User usr)	Called to remove *usr* from a group.
void *deleteGroup*(Credential c)	Deletes an existing group. Credentials are used to ensure that the callee has permission to delete the group.
List⟨User⟩ *getMembers*()	Returns the list of group members.
User *getLeader*()	Returns the group leader.
Group *getRoot*()	Returns a reference to the root of the group.
Group *getParent*()	Returns a reference to the parent of the group.
List⟨Group⟩ *getChildGroups*()	Returns the list of children groups of the group.

Group Membership API - MembershipProperties Class

Method	Description
void *setTimeBound*(Time from, Time to)	Used to set the time property for identifying users active in the given time interval (typically used in conjunction with the location property).
void *setLocationBound*(Location center, Double radius)	Used to set the location where a user is/has been/will be (typically used in conjunction with the time property).
void *setSocialNetwork*(SocialNetwork network, Activity a)	Used to identify group members who are part of the user's social network based on activities such as friendship, work, sports, etc.
void *setList*(List⟨Users⟩ users)	Used to add specific users to a group.

Group Communication API - Group Class

Method	Description
void *setReadCallBack*(ReadCallBack callBack)	Registers the read callback methods for incoming messages. *ReadCallBack* is an interface with four callback methods corresponding to broadcast, anycast, scatter-gather, and point2point.
void *broadcast*(Message msg)	This method is used to broadcast messages to a group.
void *anycast*(Message msg)	Used to send a message to a random member of the group.
void *scatterGather*(ChannelID cid, Message msg)	Used to broadcast messages to a group and get responses from group members back to the broadcaster. An app can use as many scatterGather channels as required by using different *ChannelID* to different channel
void *point2point*(Message msg, User to)	Used for user-to-user communication.
void *sendToLeader*(Message msg)	Used for sending a message to the group leader.

TABLE I: Moitree API

IV. Middleware

The Moitree middleware is responsible for providing the runtime support for the API discussed in Section III. The overall structure of the middleware is shown in Figure 1. The middleware is implemented in libraries, which process app requests with the support of standard Android system services and a few Avatar-specific services.

The middleware has the following components. *System Consistency Support (SCS)* synchronizes data and system states between a mobile device and its associated avatar to create a consistent execution environment for apps. *Event and Message Services (EMS)* manage and dispatch events and messages to drive the app execution. *Group Management Service (GMS)* manages groups and implements all group related functionality. It also supports communication among group members by managing communication channels. *Directory Service (DS)* responds to queries requesting user information. *Storage Service (SS)* facilitates shared storage space for the

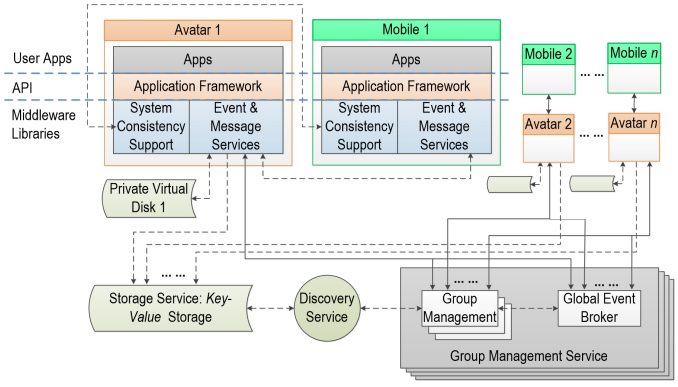

Fig. 1: Moitree middleware architecture

middleware. SCS and EMS reside on both mobile devices and avatars. The other components are provided as part of the cloud infrastructure and run on dedicated servers. The number

of servers is dynamically adjusted based on the workload.

A. System Consistency Support (SCS)

The avatar and the mobile device of a user adopt the same OS (Android in our current implementation). This allows the same app code to run unmodified on both platforms. SCS ensures a consistent system environment for the app instances on the platforms. SCS synchronizes the necessary data (e.g., photos, contacts, app-specific data) and system/app state (e.g., current location of the mobile device). To achieve low synchronization overhead, Moitree runs a daemon on each mobile and its avatar. The daemons rely on each app to define the data to be synchronized and specify when and how frequently the data should be synchronized. The cost of synchronization depends on user policies and the amount of data to synchronize. A policy that requires to keep everything always in sync will be costly in terms of energy and bandwidth usage. The other extreme policy could be to synchronize only using WiFi and while the phone is charging. Currently, we are working on fine grained policies which will provide optimized solutions in terms of usability and energy/bandwidth cost.

B. Event and Message Services (EMS)

Moitree API calls from an app are translated by the middleware as a set of events or messages. EMS forwards the events and messages to the appropriate app instances in the group with help from GMS. On each system (mobile device or avatar), the EMS component is implemented as a set of Android services, which are shared by all the apps on that system. To handle events and messages, two queues are established: an event queue (EQ) and a message queue (MQ). Events generated from apps (e.g., group creation event in LostChild) are posted to EQ. MQ is used to distribute data and receive results (e.g., photos of the lost child, photos matching the lost child and the related location/time information).

Moitree events and messages do not depend on network addresses/names to reach their destinations. Instead, they rely on the groups and channels established in each group. Specifically, events are defined as (*app*, *group*, *type*, and *data*) tuples; messages are defined as (*app*, *channel*, *group*, *type*, and *body*) tuples. The recipients of an event are the members in the group of the app specified by the *app* and *group* fields. Within a group, the channel types (i.e., *type* field) help demultiplexing messages. GMS assists event/message forwarding with group and channel information.

When an event or a message is generated on a mobile device, it is first forwarded to its corresponding avatar before it is delivered to each recipient of a group. Compared to directly sending the event/message to every recipient, sending it only once to the avatar can significantly reduce the mobile data traffic and its potential monetary cost. When the event/message arrives at the avatar, EMS forwards it to a GMS server, where the recipient list can be determined. Then, the event/message is dispatched by the GMS server to the recipient avatars, and these avatars finally forward it to the corresponding apps.

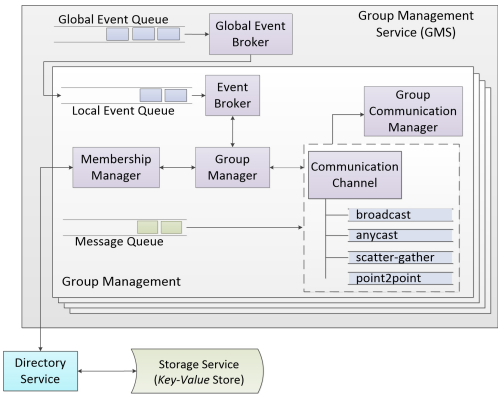

Fig. 2: Group Management Service architecture

C. Group Management Service (GMS)

GMS is the core of the middleware. It is designed to handle group operations, events, and communication. GMS runs on a group of dedicated servers, and its internal modules are shown in Figure 2.

For each group hierarchy, a *group manager* is used to maintain its tree structure, and a *membership manager* is used to maintain a list of the current members of each group or subgroup. These two modules are also responsible for handling requests, such as creating and managing groups, changing membership, etc. To keep the data structures up-to-date, they use keep-alive messages to find out failures of the members; members are removed from groups when failures are detected.

For each group, there is an *event broker* in charge of delivering events within the group, and a *group communication manager* to maintain communication channels and to forward messages to recipients. Thus, the handling of events and messages is separated to prevent a large number of messages to delay a few important events. The middleware gives higher priority to event handling compared to message handling because events are associated with important group/system state changes that must be reflected in real-time in apps.

The forwarding workload of the GMS servers can be offloaded to group leader avatars. However, this requires that group and channel information is duplicated to these avatars. This may lead to privacy concerns and additional overhead to maintain the information consistent. Another optimization aiming to reduce the load on GMS servers is to save large messages into a shared key-value store. Instead of forwarding complete messages, the GMS servers just forward the keys of the messages. When an avatar receives a message key, it reads out the message from the shared storage. However, this method increases the workload of the storage service. Thus, we have not included these optimizations in our current implementation because they need additional experimental evaluation.

D. Storage Service (SS)

SS provides a shared and permanent storage space for the middleware and is implemented as a key-value store. Specifically, this service uses the Redis [5] key-value database

Application	Moitree	JXTA
Lost Child	85	178
Video Conferencing	100	219

TABLE II: Number of lines of code for our apps using Moitree and JXTA

because it is fast, reliable, and can work on both a single node and a multi-node configuration.

SS maintains an app registry, which serves the purpose of finding which app is installed on which user's device and avatar. An entry in the app registry is created when an Avatar app is installed on a user's mobile device and avatar. The registry entry contains the avatar ID and the app's name. Information about the users' dynamic location and time is also stored by this service to assist the DS component. Finally, this service could be used for sharing large messages as described in section IV-C.

Each avatar has a virtual disk directly attached to it; these disks are not part of the storage service. Each disk serves as the private and primary storage for the avatar and the apps running on it.

E. Directory Service (DS)

DS provides answers for queries such as "which users have the LostChild app installed and were present in Times Square between 5PM and 6PM today?". DS uses SS as its data repository. Normally, we expect the mobile carrier to provide this service.

V. Evaluation

We implemented a prototype of Moitree and two apps: the LostChild app and a video-conferencing app that allows users to create group hierarchies. The Moitree prototype consists of 2722 lines of Java and Android code. The experimental evaluation has three goals: (1) verify the *effectiveness* of the Moitree programming model to reduce programming complexity by implementing two apps; (2) assess the *performance* of an app developed over Moitree, and (3) test the *efficiency* of the middleware through micro-benchmarks.

Our testbed uses Android-based mobiles and Android x86 VMs running in an OpenStack-based cloud. The mobiles are mix of Nexus 6, Nexus 5, Moto X Pure, Kindle Fire, and HTC One M7. Unless otherwise specified, each avatar VM runs Android 4.4 and is configured with 4 virtual CPUs and 3GB memory. The mobiles communicate with the cloud using our institution's secure WiFi network.

A. A Case Study on Programming Effort

To quantify the benefits of the Moitree programming model, we used the LostChild app and a video conferencing app. In the video conferencing app, users share real-time video streams with friends, family or acquaintance. These three types of groups have different levels of permissions. Friends and family have permissions to see all the streams, while acquaintance can view only selected streams. Both apps are typical mobile distributed apps that may involve a large number of mobile users. Their implementations must deal with the common issues that are usually faced in mobile distributed apps (e.g., identifying and coordinating groups of participants). Therefore, they can act as a good test for Moitree.

We compared two implementations of each app: one done using Java and JXTA [11], and one done in Java and Moitree. JXTA is selected to compareto Moitree for two reasons: (1) JXTA is designed for peer-to-peer systems, in which peers are conceptually similar to sets of autonomous avatar/mobile pairs, and (2) it also has group concepts, although different from those used in Moitree.

For the two implementations of each app, we compared the sizes of their source code. In this comparison, we only counted the lines written by our programmers. The code in other libraries (e.g., OpenCV [3] for face recognition and Kryonet [2] for network communication) is not counted toward the effort to develop the app. The app implementations include mostly group management and group communication features; the rest is done through library function invocations.

We show the numbers of lines of code (LOC) in Table II. We found that Moitree decreases LOC by a factor higher than 2. This is a promising result that illustrates how Moitree can simplify the programming of mobile distributed apps. Currently, we are implementing several additional apps for a more thorough evaluation.

B. Performance of the LostChild App

To understand the real-time performance of Moitree apps and the effect of avatars on latency, we measured the response time for the LostChild app in two scenarios: (1) the major workload in the app, including face detection and recognition, is handled on the mobiles, and (2) the major workload is executed at the avatars. Let us note that mobile to mobile communication in the first scenario is mediated by Moitree services in the cloud. Figure 3 shows the end-to-end response time from the time of submitting the initial request until the time of receiving the final results. The figure also shows the breakdown of the latency between the face detection/recognition operations and the networking/middleware operations. In these experiments, we used one mobile device as initiator and three other mobile devices as participants. All the avatars were instantiated on the same server. In this experiment, each avatar VM runs Android 6.0 and is configured with 6 virtual CPUs. Each participant has a database of 47 images containing 60 faces stored in her avatar. In addition, each participant returns a result because all participants have photos of the lost child. The training process for face recognition is done before the app starts.

The results demonstrate that avatars help reduce the end-to-end response time to half when compared to the scenario where the mobiles handle the major workload. A substantial part of this improvement is due to offloading the computation for face detection and recognition to the avatars. We also observe that the latency incurred by Moitree and networking is reduced by 14.4%; this is due to offloading the communication part in these workloads to the cloud.

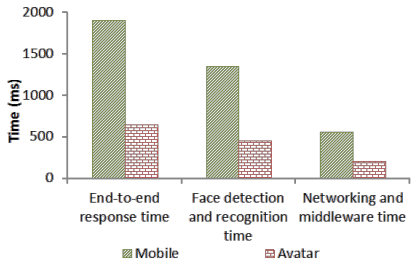

Fig. 3: End-to-end response time for *LostChild* app when the workload is at the mobile and avatar, respectively

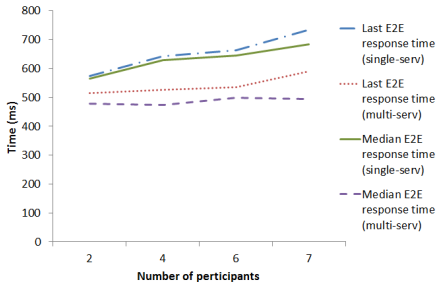

Fig. 4: End-to-end response time for *LostChild* app vs. number of participants: (i) single-serv: all participant avatars run on one server; (ii) multi-serv: each participant avatar runs on a different server

Figure 4 shows that the response time when we varied the number of participants from 2 to 7. In this experiment, the face detection and recognition are executed at the avatars. The experiment was conducted for two scenarios: (1) all the participant avatars run on a single server; and (2) each participant avatar runs on a different server. All the other parameters are the same with those on the previous experiment. The figure plots the median response latency and the latency of the last received response as experienced by the initiator. Let us recall that in this experiment each participant sends a response. In a real-life situation, most participants are not expected to send reponses. Therefore, the curves for the latency of the last received response represent the worst case scenario.

The results show that the absolute values are reasonable (generally, between 500ms and 700ms). In addition, the application scales well with the number of participants for this experiment. The number of participants has almost no effect on the median latency values. However, the latency of the last received response is affected by the number of participants. We found two reasons for this problem. First, our current Moitree implementation sequentializes the communication among the members and adds a few of milliseconds to every message transmission. Second, avatars do not send responses at the same time; in most experiemnts, we noticed one or two stragglers. We are working on improving the message delivery system of the middleware and planning more experiments to better understand the second problem. Finally, let us note that running one avatar per server improves the response time and, as expected, is relatively constant. Running all avatars on one server, on the other hand, leads to higher latency and this latency increases with the number of participants. This is caused by the resource contention incurred by the increasing number of avatars on the same machine.

Figure 5 shows the power savings achieved on a LostChild participant phone when face recognition is done on the avatar vs. on the phone. The power measurements in our experiments were taken with Power Tutor [4]. When face recognition runs on the phone, the energy consumption is approximately 225J. When the face recognition is run at the avatar, the energy consumption on the phone is about 9J. Thus, we conclude that Moitree and Avatar lead to significant improvements in response latency and energy savings on the mobiles.

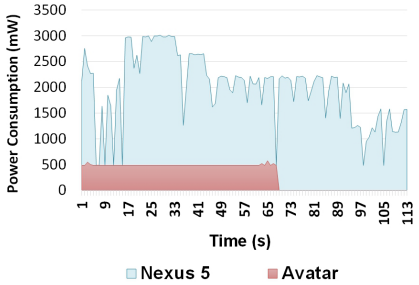

Fig. 5: Power consumption comparison for participant *LostChild* app phone with and without avatar help

Component	Energy/Power Consumed	Comment
Moitree in idle state	5.5 mJ/sec	Middleware could run for two and half month before draining the battery
Moitree API calls	2.3 mJ/call	One and half million API calls with a full battery
Data transfer by middleware & Plain TCP	0.5 mJ/KB & 0.15 mJ/KB	Energy consumed in addition to WiFi being ON for the transfer

TABLE III: Moitree's energy consumption on phones

C. Experiments with Micro-benchmarks

Moitree itself consumes very limited resources. The mobile part of Moitree takes 35 MB of memory and registers almost no CPU usage in the idle state. However, to maintain the avatar pairing, it periodically checks for data synchronization and keeps waiting for communication requests. That is why we see the low energy usage in Table III. This result shows that energy consumption on mobiles in the idle state is negligible for all practical purposes. Similarly, Table III shows that the energy consumed per average API call is very low (a full battery charge allows one and a half million calls). In terms of data transmission, Table III shows that Moitree introduces a relatively high overhead when compared with plain TCP. This is due to the Kryonet [2] library used for communication, which simplifies programming at the cost of overhead. Nevertheless, the absolute values are still low.

Our next set of experiments show the scalability and the

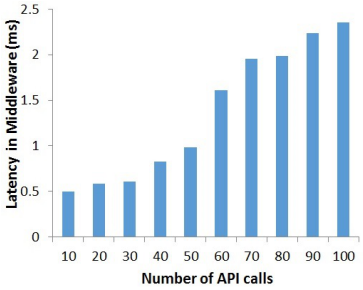

Fig. 6: Average end-to-end latency for concurrent API calls in Moitree (including network communication)

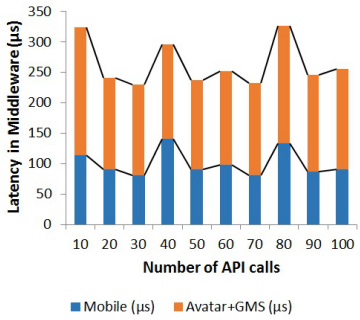

Fig. 7: Average processing time for API calls in Moitree on mobile and avatar (no network communication)

Sensor	Reading latency (ms)
Accelerator	3.26
Gyroscope	1.96
Magnetometer	3.60

TABLE IV: Sensor data reading latency in Moitree

pairing capability of the mobile/avatar pairs. For the scalability tests, we measured end-to-end latency for concurrent API calls. Test programs concurrently called random APIs from mobiles and avatars, and then measured the latency to complete these calls. The API calls start at the *App* layer in Figure 1, pass through the *API and middleware* layers, reach the GMS service layer, and then the results are returned via the reverse path.

Figure 6 shows the latency of the concurrent API calls. We used up to 100 API calls at the same time, which is a significant load. The aim of the experiment is to assess the delay imposed by the middleware to process the calls. The results show that even 100 concurrent API calls can be resolved in 2.4ms. Next, we instrumented the middleware and discarded the communication time (used by the kryonet [2] library). The results are shown in Figure 7. We see that the execution of the API calls remains approximately constant at about 320μs. This demonstrates that Moitree scales well at a load of up to 100 concurrent API calls.

Our final set of experiments show the pairing capability of the middleware. The most important metric for pairing is the latency to resolve API calls spanning across the mobile and the avatar. For this experiment, our test program runs on an avatar and reads sensor data on the corresponding mobile. The middleware on the avatar intercepts the calls, requests data from the mobile, and transparently provides the data to the running programs. Table IV shows the average latency incurred for three of the sensors present in the Nexus 5 phones. We deducted the network delay from the readings as it depends on external factors not related to the Moitree middleware. All the latencies are under 4ms, which is fine for most real-time apps. We are working on decreasing the networking latency which may hamper the performance of real-time apps.

VI. Related Work

While assisting mobile devices with cloud resources is a very active research area [23], [9], [10], [16], [24], Moitree is the first middleware for cloud-assisted mobile *distributed* apps. Very recently, a few works have investigated cloud support for mobile distributed computing [16], [24]. Clone2Clone [16] offloads peer-to-peer networking to the cloud, thus enabling more efficient communication among mobile users. Moitree, on the other hand, provides full system support for the execution of mobile distributed apps and a high-level API for programming distributed apps over mobile/avatar pairs. Sapphire [24] is a distributed programming platform for mobile cloud applications that separates the application logic from the deployment logic. Thus, programmers can modify distributed application deployments without changing the application code (e.g., change the caching behavior). This work is complementary to Moitree and could be leveraged by the Avatar platform to allow for dynamic management of non-functional app features. It should be noted that Moitree is not just for offloading of computation and communication to the cloud; rather, it is a programming model to build mobile distributed apps based on dynamic context such as social groups, time, and location, while providing computation and communication offloading to improve efficiency.

Moitree has clear advantages in terms of latency, energy-efficiency, and availability over middleware platforms for programming distributed apps designed for purely mobile environments [17], [18], [19], [20]. Among the middleware for distributed programming over mobile ad hoc networks (MANET), LIME [20] and TMACS [17] propose group abstractions similar to Moitree. LIME [20] provides a framework in which mobile agents can form groups based on context-awareness. Moitree's programming model has two main advantages over LIME: more flexible communication abstractions, and its supporting middleware performs transparent dynamic group management. TMACS [17] proposes an object-oriented distributed middleware framework for MANET. In Moitree, groups are defined based on users and their activities activities rather than the types and scopes of objects as in TMACS. This makes mobile distributed programming simpler and more natural.

MELON [12] is a general purpose coordination language for MANET that supports asynchronous exchange of persistent messages. Although MELON provides an API similar to

Moitree, it does not support group management or different types of communication between group members.

Pogo [8] and MobiSoC [13] are closer to Moitree because they use server-side resources to provide middleware platforms for specific areas of mobile computing. Pogo [8] proposes a middleware for distributed mobile phone sensing. Unlike Pogo which focuses on sensing, Moitree provides a general programming model for mobile distributed computing. Furthermore, Pogo does not explicitly use group abstractions such as Moitree. Also, the assignment of mobile sensing devices to a particular researcher is done by an administrator in Pogo, while Moitree groups are handled dynamically by the middleware. MobiSoC [13] supports mobile social computing and provides a high-level API based on people and places, similar in nature with the one provided by Moitree. Both platforms use groups as main abstractions. But unlike MobiSoC which maintains global state about communities at the server-side, Moitree provides a distributed architecture in which apps work in peer-to-peer fashion. Furthermore, MobiSoC focuses on mobile social apps, while Moitree enables general-purpose mobile distributed apps.

VII. Conclusion and Future Work

To the best of our knowledge, Moitree is the first middleware for mobile distributed apps assisted by the cloud. We implemented Moitree on top of our Avatar platform and tested it with two apps. The results of our evaluation are promising. Moitree is able to reduce the number of lines of code to less than half when compared to an existing solution. In addition, Moitree scales well when multiple APIs are invoked concurrently and helps users save energy on mobile devices at the cost of a reasonable latency overhead.

As future work, we are building a new distributed file system, with the goal of making the data available and consistent for the computation on both the mobile and the avatar, while reducing the energy and bandwidth costs. We are also working on privacy aware computation techniques and seamless computation offloading for Moitree.

Acknowledgment

This research was supported by the National Science Foundation (NSF) under Grants No. CNS 1409523, CNS 1054754, DGE 1565478, and DUE 1241976, the National Security Agency (NSA) under Grant H98230-15-1-0274, and by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under Contract No. A8650-15-C-7521. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, NSA, DARPA, and AFRL. The United States Government is authorized to reproduce and distribute reprints notwithstanding any copyright notice herein.

References

[1] Intel begins shipping Skylake CPUs with SGX enabled. http://www.intel.com/content/www/us/en/processors/core/6th-gen-core-family-desktop-brief.html.

[2] Kryonet. https://github.com/EsotericSoftware/kryonet.

[3] OpenCV. http://opencv.org/.

[4] Power Tutor. http://ziyang.eecs.umich.edu/projects/powertutor/.

[5] Redis. http://redis.io/.

[6] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 267–283. USENIX Association, 2014.

[7] C. Borcea, X. Ding, N. Gehani, R. Curtmola, M. A. Khan, and H. Debnath. Avatar: Mobile distributed computing in the cloud. In *The 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud '15)*, March 2015.

[8] N. Brouwers and K. Langendoen. Pogo, a middleware for mobile phone sensing. In *Proceedings of the 13th International Middleware Conference (Middleware '12)*, pages 21–40, December 2012.

[9] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: Elastic execution between mobile device and cloud. *Proceedings of the 6th EuroSys Conference (EuroSys 2011)*, pages 301–314, 2011.

[10] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. *Proceedings of the 8th international conference on Mobile systems, applications, and services (MobiSys '10)*, pages 49–62, 2010.

[11] L. Gong. Jxta: A network programming environment. *Internet Computing, IEEE*, 5(3):88–95, 2001.

[12] S. Guinea and P. Saeedi. Mobile application development with MELON. In *Proceedings of 13th International Conference ADHOC-NOW 2014*, pages 265–278, June 2014.

[13] A. Gupta, A. Kalra, D. Boston, and C. Borcea. Mobisoc: A middleware for mobile social computing applications. *Springer MONET*, 14(1):35–52, Jan. 2009.

[14] K. Kim, S. Lee, and P. Congdon. On cloud-centric network architecture for multi-dimensional mobility. *Computer Communication Review*, 42(4):509–514, 2012.

[15] 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. *Proceedings of the IEEE Infocom 2012*, pages 945–953, 2012.

[16] S. Kosta, V. C. Perta, J. Stefa, P. Hui, and A. Mei. Clone2clone (c2c): Peer-to-peer networking of smartphones on the cloud. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, Berkeley, CA, 2013. USENIX.

[17] J. Lin, E. Shing, W.-K. Chanand, and R. Bagrodia. TMACS: Type-based distributed middleware for mobile ad-hoc networks. In *Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, July 2008.

[18] J. Liu, D. Sacchetti, F. Sailhan, and V. Issarny. Group management for mobile ad hoc networks: design, implementation and experiment. In *Proceedings of the 6th international conference on Mobile data management (MDM '05)*, page 192199, May 2005.

[19] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications with the tota middleware. In *Proceedings of of the second IEEE Annual Conference on Pervasive Computing and Communications (PERCOM 2004)*, pages 263–273, March 2004.

[20] A. L. Murphy, G. P. Picco, and G.-C. Roman. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(3):279–328, July 2006.

[21] Y. Qin, D. Huang, and X. Zhang. Vehicloud: Cloud computing facilitating routing in vehicular networks. *Proceedings of the 11th IEEE TrustCom*, pages 1438–1445, 2012.

[22] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: Enabling interactive perception applications on mobile devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 43–56, New York, NY, USA, 2011. ACM.

[23] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[24] I. Zhang, A. Szekeres, D. V. Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy. Customizable and extensible deployment for mobile/cloud applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 97–112, Broomfield, CO, Oct. 2014. USENIX Association.