The Journal of Systems and Software 157 (2019) 110387

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

The Moitree middleware for distributed mobile-cloud computing

Hillol Debnath*, Mohammad A. Khan, Nafize R. Paiker, Xiaoning Ding, Narain Gehani, Reza Curtmola, Cristian Borcea

Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102-1982, USA

ARTICLE INFO

Article history:
Received 22 October 2018
Revised 4 June 2019
Accepted 23 July 2019
Available online 24 July 2019

Keywords:
Mobile cloud computings
Mobile apps
Middleware
Distributed systems

ABSTRACT

Commonly, mobile cloud computing assumes that each mobile device of a user is paired with a user-controlled surrogate in the cloud to overcome resource limitations on mobiles. Our Avatar platform leverages this model to support efficient distributed computing over mobile devices. An avatar is a per-user, always-on software entity that resides in the cloud and acts as the surrogate of the mobile. Mobile-avatar pairs participate in distributed computing as a unified computing entity in such a way that the workload and the demand for resources on the mobiles remain low. This paper presents Moitree, the middleware of the Avatar platform, which provides a common programming and execution framework for mobile distributed apps. Moitree allows the components of a distributed app to execute seamlessly over a set of mobile-avatar pairs, with the provision of offloading computation and communication to the cloud. The programming framework has two key features: user collaborations are modeled using context-aware group semantics - groups are created dynamically based on context; data communication among group members 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 a cloud running Android x86 avatars.

1. 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 on mobiles, and improve the availability of mobile apps (Satyanarayanan et al., 2009; Chun et al., 2011; Cuervo et al., 2010; Kosta et al., 2013; Zhang et al., 2014). 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 (SGXenabled, 2019), thus offering privacy guarantees that the surrogates are truly personal and protected from the cloud providers (Baumann et al., 2014). Therefore, a 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. The computation can be executed on the mobiles or in their surrogates in the cloud, based on performance or privacy requirements. Let us note that such a model is in line with new privacy laws, such as the European Union's General Data Protection Regulation (GDPR) (Gdp, 2019), which allow users control over their personal data.

Our Avatar platform (Borcea et al., 2015) leverages this model to support efficient *mobile distributed computing* apps executed over sets of mobile/surrogate pairs. 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. Mobile/avatar pairs participate in distributed computing as unified computing entities in such a way that the workload and the demand for storage, bandwidth, and energy on the mobiles remain low. Apps supported by Avatar allow people to collaborate within groups defined by friendship, common interests, geography, etc. Examples of distributed apps include finding people of interest in a crowd using face recognition techniques (e.g., a lost child), real-time dating based on learned facial preferences (Neog et al., 2016), and mobile multi-player gaming. Other areas for applications are mobile health, public safety, vehicular traffic management, and collaborative sensing.

This paper presents Moitree[1], the middleware of the Avatar platform. Moitree provides a set of APIs and libraries for

* Corresponding author.
E-mail addresses: hd43@njit.edu (H. Debnath), mak43@njit.edu (M.A. Khan), nrp48@njit.edu (N.R. Paiker), xiaoning.ding@njit.edu (X. Ding), narain.gehani@njit.edu (N. Gehani), reza.curtmola@njit.edu (R. Curtmola), borcea@njit.edu (C. Borcea).

[1] This article extends the work presented in the conference paper (Khan et al., 2016) published in IEEE Mobile Cloud 2016

https://doi.org/10.1016/j.jss.2019.07.089
0164-1212/

developing cloud-assisted mobile distributed apps, as well as runtime support to execute these apps. Moitree focuses on managing the distributed computation, the communication, and the mobile/avatar set that represents the group of users collaborating within the distributed app. For other functionality, the apps can use the programming support provided by the local OS.

Programming apps over mobile/avatar pairs is different from traditional distributed programming. The first difference is that the end points in the computation are mobile/avatar pairs with different capabilities - mobiles have multiple types of sensors and user interaction capabilities; avatars have more powerful computation, storage, "unlimited" energy, etc. Developers need an intuitive and common programming 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. Moitree's main novelty is that it allows the components of a distributed app to execute seamlessly over a set of mobile/avatar pairs representing a group of collaborative users, with the provision of offloading computation and communication to the cloud.

The second difference with traditional distributed computing is that the apps require user collaboration based on natural context, such as location, time, social relationships, etc. Therefore, managing the dynamic set of participating users (i.e., mobile/avatar pairs) in real time is important. Moitree facilitates the collaboration among participating users through two key features. First, user collaborations are modeled using context-aware group semantics; groups are created dynamically based on context and are hierarchical. Moitree 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). The group hierarchy allows programmers to naturally organize users into groups/subgroups and manage their collaborations within different scopes. Second, data communication among group members is offloaded to the cloud through high-level communication channels. This avoids communication problems due to mobility and the dual nature of a user end-point (i.e., mobile and/or avatar). In addition, it can reduce the bandwidth and energy consumption on the mobiles.

We implemented a prototype of Moitree on Android devices and an OpenStack-based cloud running Android x86 avatars. We have developed two proof-of-concept apps in order to evaluate the programming effort minimized by Moitree. One of the apps 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 dating app based on users' preferences about partners' faces. Although both apps use face recognition to achieve their goals, the work-flow and their use of Moitree API is different from each other. The number of lines of code of Moitree-based app implementations is significantly lower when compared to their implementations done without Moitree. In addition, experimental results demonstrate that Moitree helps these apps improve the overall response time through the use of cloud support. We also performed experiments with micro-benchmarks on top of our prototype to evaluate the efficiency, scalability, and overhead of the middleware. The results show that Moitree scales well with a large number of concurrent API calls, while consuming small amounts of resources (CPU, memory, and energy).

To the best of our knowledge, Moitree is the first middleware for distributed mobile cloud computing. To summarize, its main contributions are: (1) Common app execution environment on mobiles and avatars that allows seamless offloading of app components; (2) High-level distributed programming model for mobile cloud apps, which uses context-aware group-based abstractions to manage user collaboration; and (3) Transparent management of communication, including end-point mapping to mobiles or avatars.

Compared with our preliminary description of Moitree (Khan et al., 2016), this paper presents the updated design of Moitree, with a focus on the novel programming and system challenges and their solutions. In addition, it includes a new and detailed app usage case to illustrate the main features of the middleware. Furthermore, the paper describes the novel aspects of our prototype, which include app execution, offloading, and scalability. Finally, the evaluation presents new results, e.g., offloading, stress test, resource overhead, and benefits of reusing groups.

The rest of the article is organized as follows. Section 2 presents an overview of the Avatar platform and Moitree. Section 3 presents the key ideas of the Moitree high-level programming model. Section 4 presents the Moitree API and sample code snippets of the API usage. The design of the Moitree middleware is discussed in Section 5, and the implementation of the middleware is presented in Section 6. Section 7 validates the programming model with two proof-of-concept applications and shows macro & micro-benchmark results for the middleware. Related work is discussed in Section 8, and the paper concludes in Section 9.

2. Overview

2.1. Background: Avatar platform

In Avatar, a mobile user is represented by one mobile device and an 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. Avatars save energy on the mobiles and improve the response time for many apps by executing certain tasks on behalf of the mobiles. 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 apps on the avatar and/or mobile device. A mobile device does not interact directly with the avatars of other mobile users. All user-to-user communication is offloaded to the cloud (i.e., always goes through the avatars) in order to reduce wireless bandwidth usage at mobile devices.

Although containers can be used to host avatars, our prototype is implemented using virtual machines (Android x86) for flexibility and ease of prototyping. The OS and the runtime (Dalvik or ART) are the same in avatars and mobiles. Thus, the same app or app components (e.g., functions, threads, etc.) can run on both of them. For each user, the platform maintains a mobile-avatar pair. When a user installs an app running over the Avatar platform, the same app gets installed on both mobile and avatar. Users can install as many Avatar apps as they want on their devices. Standard Android apps can co-exist with Avatar apps on the mobiles without conflicts.

2.2. Moitree motivation and challenges

Programmers have to devote a substantial effort to develop distributed applications over mobile-cloud platforms such as Avatar if they use standard programming frameworks (e.g., developing apps directly in Android). For example, they have to write code for discovering potential participants for a collaborative app, managing participating users, distributing the computation and data among users and mobile/avatar pairs, and managing communication among the set of mobile-avatar pairs. All these tedious tasks are not directly related to the app logic. Therefore, designing a middleware that provides high-level abstractions and a programming model for collaborating users will help programmers to reduce the programming effort significantly.

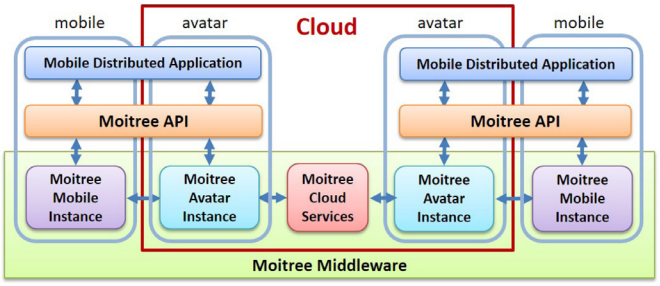


Fig. 1. Execution of distributed apps facilitated by Moitree.

Such a middleware has two types of challenges: programming/API challenges and system challenges. The programming challenges are: (P1) How to provide a high level abstraction (i.e., hiding low-level system details) of the distributed computation model provided by Avatar? (P2) How to model a collaboration among various participants? (P3) How to model the dynamic context of a distributed computation (e.g., compute only on participants present at a specific location during a specific time range)? (P4) How to offload user-to-user communication to the cloud and how to design an API to facilitate easy data communication among participating users?

In order to support such a high-level API, the middleware runtime system needs to overcome the following system challenges: (S1) How to run on heterogeneous resource constrained devices (e.g., smart phones, smart watches)? (S2) How to model and handle the expected asynchronous behavior of user-to-user communication? (S3) How to support applications that need fast/real-time data communication? (S4) How to maintain good scalability to ensure the middleware remains responsive at high loads?

2.3. Moitree at a Glance

While Avatar provides a specification for running mobile collaborative apps on a cloud-assisted distributed computing platform, Moitree provides an API set and a runtime system to develop and execute these apps. The API set is designed to addresses the programming challenges (P1-P4) described in Section 2.2. A context-aware group abstraction models the set of collaborating users in a distributed computation. The API employs dynamic group membership to support context changes for the group participants. Several abstract communication channels are designed to handle communication offloading and simplify data communication among participants. These communication channels are oriented toward group communication and abstract the low-level networking details.

Moitree is also designed to overcome the system challenges (S1–S4) described in Section 2.2. To increase responsiveness and scalability of the middleware, all the system components are kept loosely-coupled. The middleware is designed as a Message-Oriented Middleware (MOM)(Curry, 2004) which uses events and messages between components to perform a system-wide task. Both loosely-coupled and MOM design help the middleware to support heterogeneity and asynchronous behavior. Callback-based communication is employed to cope with asynchronous behavior. To make the middleware lightweight and data communication fast, the middleware is designed to (i) use lightweight and efficient data structures for events/messages, (ii) reduce the time for data serialization, (iii) reduce the amount of processing needed for making event/message routing decisions.

Fig. 1 shows an overview of Moitree, where two users' mobile-avatar pairs are running a distributed app. The app uses the Moitree API for performing collaborative computation on top of the Avatar platform. The middleware translates the API calls and executes them on the platform. Instances of Moitree middleware run on mobiles and avatars. Moitree also has a few cloud services, which facilitate the distributed execution environment.

2.4. CASINO: collaborative offloading framework

Moitree aims at solving the programming and system challenges to enable the development and execution of mobile-cloud distributed apps. However, the execution environment of a mobile-cloud distributed app consists of multiple mobile devices and multiple avatars, and thus it is substantially more complex than that of a conventional mobile app or a non-distributed mobile app with cloud assistance. Therefore, the combination of offloading tasks to the avatars and scheduling the tasks on a set of mobiles and avatars becomes a major challenge, particularly when trying to minimize the overall app completion time.

A few challenges that must be addressed include how to resolve the dependencies between different components of code (i.e., tasks), how to schedule them in a way that can optimize the total completion time (i.e., when and where each task should be executed), and how to orchestrate the execution of the scheduled tasks.

To address these challenges, we have designed CASINO (Debnath et al., 2018) to collaboratively offload and schedule the tasks in each mobile-cloud distributed app. CASINO supplements and work synergistically with Moitree. How programmers can utilize CASINO's API and how CASINO collaborates with Moitree are described in Section 5.3. The evaluation of CASINO and the findings are described in Section 7.5.

3. Programming model

In a Moitree application, a set of users collaborate with each other to finish a global task. Each user willing to participate in such a task installs the app; the installation results in app instances on both the user's mobile and the user's avatar. During the collaboration, participating users may play different roles and need to communicate with each other. When developing a Moitree program, in addition to implementing the core operations that fulfill app-specific functionality (e.g., face recognition), programmers must effectively manage collaborations, such that core operations can be orchestrated and conducted efficiently on sets of mobile/avatar pairs of the participating users. Moitree provides two key concepts for a program to manage collaborations: user collaborations are modeled using group semantics - groups are created dynamically based on context and are hierarchical; data communication among group members is offloaded to the cloud through high-level communication channels. To explain these concepts, we start by introducing a few app examples.

3.1. App examples

With Moitree's dynamic group management and easy communication mechanism, developers can build different types of distributed apps in a simpler and more effective way. For example, finding people of interest in a crowd can be very useful in certain scenarios. But, this can be very challenging to implement in a distributed app without infrastructure support due to the difficulties in organizing and coordinating the work done by the participating users. This section will present three apps that demand and receive very different capabilities from Moitree.

3.1.1. LostChild

One app example is to find a lost child by checking if the child happens to appear in photos taken by nearby people. Using

the location and time associated with the photos in which the child appears could help a parent locate the child or at least to narrow down the area where the child is. The likelihood of finding such photos is high in crowded/touristic places. The feasibility of such an app is further demonstrated by real-life situations in which police have caught criminals with the help of tourists' photos (News, 2019).

Let us consider that Alice is visiting Times Square in New York with her mother Mary, and she gets lost. Mary would immediately use this app, called LostChild, to try to find Alice. We also assume that many other people in Times Square have this app installed. Specifically, Mary broadcasts a set of photos of Alice's face through the app, which are received by the LostChild app instances of other visitors and are compared to faces extracted from photos they have taken recently. If a match is found, the location and time information of the photo is sent back to the LostChild instance on Mary's phone, where the information can be summarized to form a trajectory of Alice's movement on a map in order to find her.

The LostChild app can run only on mobile devices, only on avatars, or on a set of mobile devices and avatars. For example, the app runs on mobile devices when they have photos not backed up to the cloud yet or when the users choose not to share photos with the cloud due to privacy concerns. However, running instances of the app on avatars and processing the photos there can reduce the response time and the energy consumption of mobile devices. More importantly, this can increase the chance of finding matching photos in cases when mobile devices are not online, but the photos have been synchronized with their associated avatars (i.e., the synchronization happens in the background when wireless bandwidth and power are plentiful at mobiles). Let us note that privacy-preserving techniques can be used to store the photos in the cloud, as well as to perform face matching computations (Almalki et al., 2016).

3.1.2. FaceDate

FaceDate is another example to run face recognition and utilize Moitree's dynamic group management feature (Neog et al., 2016). With the app, each user uploads a few photos of herself as her profile photos, and trains the app with some other photos of faces she likes. Upon user request, FaceDate examines the photos of the users currently located in the proximity of the requester, and performs face matching in real-time to match the profile photos with the photos liked by users. If a mutual match between the requester and another user is found, these two users are notified and given the option to start communication.

This app uses the location of the requester as a context to find nearby users and form a dynamic group. It then runs mutual face recognition on the requester and other users in the proximity. Although both LostChild and FaceDate use face recognition to find people of interest, the pair-wise communication pattern used by FaceDate makes it more challenging to build.

3.1.3. Divert

Divert (Pan et al., 2017) is a distributed vehicular rerouting system for congestion avoidance, which consists of instances of a mobile app on the drivers' smart phones and a server in the cloud for app coordination. By diverting a number of drivers to new routes (i.e., rerouting), it can avoid traffic congestion and save valuable time for the drivers. DIVERT offloads a large part of the rerouting computation at the vehicles (i.e., smart phones in the vehicles) to improve driver privacy and overall system scalability. The vehicles exchange messages with each other to make collaborative rerouting decisions. The server is used just to determine an accurate global view of the traffic and distribute it to individual vehicles.

With Moitree, the problem can be modeled as follows. All cars in a city form a global group, and cars on each road segment form a local, dynamic group. The cars in each local group compute an average speed of that segment (in a distributed way). The global group periodically collects these data from the local groups and creates an up-to-date global map of the traffic. This map is represented as a directed, weighted graph, where the weights are the travel time for each segment, computed based on the reported average speeds on the segments. The global map is distributed to all vehicles, which then communicate within their local groups to make collaborative rerouting decisions.

3.2. Dynamic and hierarchical groups

A context-aware group represents the fundamental unit of computation and communication for apps developed over Moitree. Such a group is a set of users selected and organized based on app-specified context properties (e.g., location, time, social context, etc.). For example, in the LostChild app, a group will be formed with the users present in Times Square and surrounding areas during the time Alice was lost. Operations performed by the group members are similar (e.g., searching for Alice's face in the photos), and communication is conducted within the group (e.g., broadcasting Alice's photos). Groups are app-specific, and an app can create as many groups as it needs. Thus, a user can be a member of multiple groups within the same app or across different apps at the same time.

Members in a group may change dynamically due to context changes of participating users. The group concept and its dynamic group membership support in Moitree shield programmers from handling context changes. Moitree selects and maintains group members automatically based on properties specified by the programmers. For example, in LostChild, a group is formed for a geographic region around Time Square and for a one hour time interval. The middleware will add and remove group members dynamically based on the users entering or leaving the region during this time.

Groups in the same app can form a hierarchical tree structure with the groups at lower levels being subgroups of the ones at upper levels; the root is the group of all the participating users. Subgroups are created within a group since there are tasks to be finished collaboratively by only some of the users in the group. For example, subgroups can be recursively created in divide-and-conquer strategies when the problem space is partitioned, and each subgroup is in charge of the tasks in a partition. Specifically, Mary's phone may be overwhelmed by a large number of responses from participants in LostChild. To avoid this performance bottleneck, a more scalable design is to geographically partition the task as well as participating users (i.e., assign a subgroup for each sub-region). When each member in a subgroup finishes examining its photos taken in the region, it sends the matching information to the leading member in the subgroup (Section 4 defines the group leader concept), where the information is summarized and the summary is sent back to Mary. It's worth noting that the leading member can be chosen following different policies (e.g., based on resource condition, etc). Since summarizing the information is comparatively less expensive than doing facial recognition, there is no significant overhead on the leading member. Subgroups may overlap since a visitor may appear in different regions, but this does not affect the tree structure of the groups.

3.3. Communication channels

A communication channel is created by Moitree to support collaboration between the members in a group. It provides high-level messaging support to simplify programming and leverages

the avatars in the cloud to deliver messages so as to minimize the resource usage on mobiles. Moitree supports four types of communication channels: (i) *broadcast* for sending messages to all members of the group; (ii) *scatterGather* for sending messages to all members of the group and then receiving answers from some group members as a function of their computation results; (iii) *anycast* for sending messages to a random member of the group; (iv) *pointToPoint* for sending messages to a specific member of the group. The *broadcast* channel is unidirectional, and the other three are bidirectional. As an example, the LostChild app can use the scatterGather channel to distribute the child's photos and get back the result.

Messages can be designated as *persistent* by the programmers. Persistent messages are particularly useful for forwarding data to the members joining the group after the group has been established. For example, in LostChild, if Alice's photos were broadcasted using persistent messages, persons who entered the Times Square after Mary broadcasted Alice's photos could still obtain these photos. In Moitree, persistent messages are stored in the cloud and are distributed to new members when they join the group.

4. Moitree API

The Moitree API is designed to be asynchronous and event-driven. This design is compatible with the loosely-coupled, distributed, and message-oriented middleware design. At the same time, since Android SDK and Java Swing GUI toolkit also follow similar design principles, the Moitree API helps maintain a familiar programming style for Java/Android programmers.

4.1. API Overview

As summarized in Table 1, the API can be broadly divided into three categories: (1) creating and managing groups and group hierarchies, (2) managing group membership, and (3) supporting communication within each group. The API methods are organized and exposed to programmers using three main classes. The *Avatar* class provides methods for group creation and joining. The *AvatarGroup* class offers methods for group management (e.g., leave/delete the group, create subgroups) and group communication. The *MembershipProperties* class has methods for specifying the group properties.

To use the Moitree API, an app needs to first instantiate an Avatar object, which is a singleton representing the mobile-avatar pair. This object is used to invoke any subsequent Moitree API. To get the instance of the Avatar object, the programmer needs to call an API with two parameters: (i) the fully qualified name of the app's main class (e.g. "edu.njit.lostchild.MainActivity"), which is used by the middleware to connect to the app instance; (ii) the context object of the application (defined by Android), which is used by the middleware for communication with other app instances.

4.2. Group creation and group hierarchy

Group creation is done through the *createGroup* API, which takes four parameters. The *parent* parameter specifies the parent group and is used for forming the group hierarchy. The *prop* parameter is an instance of the *MembershipProperties* class. It specifies the context properties that must be satisfied by group members (i.e., region and time interval), as explained in Section 4.3. Some groups may need leaders to implement functions such as consensus or scheduling among their members. If the *enableLeader* parameter 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. The method *sendToLeader* allows any user to send messages to the leader without knowing the ID of the leader. This simplifies programming and improves execution reliability. To run leader-specific functionality, an app instance needs to know whether it is the leader. This can be achieved by calling the method *getLeader*. The *grpLifetime* parameter specifies that a group has to be deleted by the middleware in the absence of any group communication for the *grpLifetime* duration. In this way, when a group is inactive, it allows Moitree to remove it and de-allocate its resources. The app receives an exception when one of its groups is removed.

To respond to group creation and receive group information, a program can register a callback function as shown below:

```
avatar.setGroupCallback(new GroupCallback(){
  @Override
  public void onCreateGroup(AvatarGroup g) {
    //use group 'g' to communicate with members
  }
});
```

When a group is created, the callback function is invoked in the instances of the group members and an AvatarGroup object *'g'* is pushed to each instance by the middleware. The AvatarGroup object is used to communicate with app instances belonging to the group members. Different members in the group may receive *group* objects with different content, such as the ID and channel information. When a new member is added to a group at a later time (i.e., dynamic group membership), the callback function is also invoked on the instance of this member.

In group hierarchy, 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 receive references to its subgroups (i.e., one level down in the hierarchy) using the *getChildGroups* method.

4.3. Group membership

In addition to the method that populates members in a group automatically based on the *MembershipProperties* specified at group creation, Moitree allows users to join a specific group by calling the *joinGroup* method. For example, a new user is invited to a multi-player mobile game and is provided the group ID and the credentials; then the new user can invoke *joinGroup* to join the group. A user can leave a group by calling *removeFromGroup*.

4.4. Group communication

Moitree supports four types of communication channels, as described in Section 3.3. A channel is instantiated by Moitree upon its first invocation in the app. The communication on all channels is asynchronous. Programs can send messages by calling the corresponding API methods and receive responses later through callback methods. Thus, each sending communication channel is paired with a receiving callback method (except for broadcast, which is unidirectional).

We use the *scatterGather* channel to illustrate how channels are used in apps. The *scatterGather* channel combines OneToMany and ManyToOne communication topologies. The initiator sends data to all group members, and then, based on their computations, some group members will send back results to the initiator. To send data through a *scatterGather* channel, an app can invoke the following method: *group.scatterGather(data, true, MessageType.DATA);* the data is of type byte[], and the message is marked as persistent (i.e., the second parameter is set to *true*). The *scatterGather* API also allows the app to deliver data through a particular channel identified with its channelID (refer to Table 1). To send back results, the API can be called with results tagged with *MessageType.RESULT*.

To receive data sent through the channel, a program needs to implement and register callbacks as follows:

Table 1
Moitree API.

Group Management API - Avatar and AvatarGroup Class	
Method	Description
createGroup(AvatarGroup parent, MembershipProperties prop, boolean enableLeader, double grpLifetime)	Creates a group with members selected based on membership properties *prop*; if *enableLeader* is true, the group has a special member with leader role. *grpLifetime* specifies how long the group should exist without receiving any messages from the members.
changeParentGroup(AvatarGroup newParent)	This method is used to re-organize the group tree.
onCreateGroup(AvatarGroup grp)	Callback method registered to Moitree for delivering newly created AvatarGroup object. Moitree pushes this group parameter to callbacks registered by initiator's and participants' apps.
joinGroup(AvatarUser u, AvatarGroup g, Credential c)	Joins an already existing group. 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.
removeFromGroup(AvatarUser u)	Removes user *u* from a group.
deleteGroup(Credential c)	Deletes an existing group. Credentials are used to ensure that the callee has permission to delete the group.
getMembers()	Returns the list of group members.
getLeader()	Returns the group leader.
getRoot()	Returns a reference to the root of the group.
getParent()	Returns a reference to the parent of the group.
getChildGroups()	Returns the list of children groups of the group.
Group Membership API - MembershipProperties Class	
Method	Description
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).
setLocationBound(LatLng center, double radius)	Used to specify a circular region where a user is/has been/will be (typically used in conjunction with the time property).
setLocationBound(LatLngBounds rectRegion)	Used to specify a rectangular region where a user is/has been/will be (typically used in conjunction with the time property).
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.
setList(List⟨Users⟩ users)	Used to add specific users to a group.
Group Communication API - AvatarGroup Class	
Method	Description
setReadCallBack(ReadCallBack callBack)	Registers callback methods for incoming messages. *ReadCallBack* is an interface with four callback methods corresponding to broadcast, anycast, scatter-gather, and point2point.
broadcast(byte[] msg)	Used to broadcast messages to a group.
anycast(byte[] msg)	Used to send a message to a random member of the group.
scatterGather(ChannelID cid, byte[] 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* for different channels
pointToPoint(byte[] msg, AvatarUser to)	Used for user-to-user communication.
sendToLeader(byte[] msg)	Used for sending a message to the group leader.

```
group.setCallBack(new ReadCallBack() {
  @Override
  public void broadcast(AvatarMessage msg) {}

  @Override
  public void anycast(AvatarMessage msg) {}

  @Override
  public void pointToPoint(AvatarMessage msg) {}

  @Override
  public void scatterGather(AvatarMessage msg) {
    if (msg.getMessageType() == MessageType.DATA) {
      //run computation on msg.getData(), send back the result
      group.scatterGather(result, MessageType.RESULT);
    } else if (msg.getMessageType() == MessageType.RESULT) {
      //This is a result received by the initiator
    }
  }

  @Override
  public void getPersistentData(AvatarMessage msg) {}
});
```

In addition to communication channels between group members, Moitree provides a sync API for synchronizing data between the mobile and the avatar of the same user. Developers can use this API to add any specific directory for synchronization. For example, the developer could invoke *avatar.addSyncableDir(DIR)* to synchronize the directory 'DIR' on the storage of the mobile device and with the avatar. Once a file is created or modified in the directory, the Sync Service of the middleware (to be introduced in Section 5) starts to transfer the new file. Note that we also developed an advanced file system (Paiker et al., 2017) for achieving more sophisticated synchronization features on top of Avatar.

4.5. API usage example

In this section, we use Divert, the traffic congestion avoidance app described in Section 3.1, to illustrate how Moitree can significantly simplify distributed app development.

Since this app needs a server in the cloud for coordination, we use a dedicated avatar that works as the leader of the main group for the city (i.e., the group that contain all the current vehicles in the city). This avatar does not need to be paired with any mobile device. Under the main group, there are subgroups representing each road segments of the city. The first car entering an empty road segment (i.e., without vehicular traffic) creates the group and becomes its leader. Any subsequent cars entering that segment become part of that group. It is worth noting that both the leader and the participants in a subgroup can leave the road segment quickly. In such a case, Moitree removes them from that subgroup and adds them to the subgroup associated with their new road segment. When a leader leaves a subgroup, Moitree elects a new leader from among the remaining subgroup members.

The following pseudo-code demonstrates how the dedicated, main avatar operates. First, it creates a group for the whole city (line 7). Then, it sets up the *sendToLeader* callback function (lines 8–12), which is invoked when the main avatar receives updates of the travel times for each road segment from subgroup leaders. The main avatar uses these messages to update the global city map (line 10). Periodically, it broadcasts the up-to-date city map to all vehicles in the city (line 16).

```
1 //Runs at a dedicated global avatar, not tied with a mobile device
2 initialize() {
3   // create the mainGroup that covers the entire city
4   MembershipProperties prop = new MembershipProperties();
5   // "map" is the whole city's map with each street as one segment
6   prop.SetLocationBound(map);
7   Group mainGroup = Avatar.CreateGroup(null, prop, true, LIFETIME);
8   mainGroup.setReadCallback(new ReadCallback() {
9     public void sendToLeader(Message msg) {
10      cityAverageMap = updateCityAverageMap(msg.segmentAverageTime);
11    }
12  });
13
14  //periodically broadcast cityAverageMap
15  while(true) {
16    mainGroup.broadcast(cityAverageMap);
17    sleep(X);
18  }
19 }
```

Next, we show the code segments that run at every user/vehicle. The code segments prepare hierarchical group structures, manage communication, and compute new routes.

The following pseudo-code shows the hierarchical group construction. There are two types of *onCreateGroup* callbacks received by cars. One is for the main group creation. Every car becomes part of the main group (line 5). When a car enters a road segment, it tries to create a subgroup for that segment (with the main group as parent) and becomes the leader of that group. Moitree decides whether there is already an existing group with a leader for the corresponding segment (lines 8–13). If there is, the createGroup request is ignored, and the car is joined to the existing group by Moitree. The *sendToLeader* callback for this group is also configured in lines 16–21. The callback is invoked at the subgroup leader when subgroup participants update the subgroup leader with their current average speed (line 19).

```
1 onCreateGroup(Group group){
2   //check if this call is for the mainGroup
3   if(group.getRoot() == NULL) {
4     //this onCreateGroup() is a join group notification for the main groups
5     mainGroup = group
6
7     // create a new local/segment group and become its leader
8     isLeader = True;
9     MembershipProperties prop = new MembershipProperties();
10    currentSegment = getCurrentSegment(map, getLocation());
11    Prop.SetLocationBound(currentSegment);
12    //set the mainGroup as the parent of the local group
13    Group localGroup = Avatar.CreateGroup(mainGroup, prop, true, LIFETIME);
14
15    //set callbacks to receive averageSpeed updates from local group members
          (e.g., cars on the local segment)
16    localGroup.setReadCallback(new ReadCallback() {
17      Public void sendToLeader(Message msg) {
18        // userId is the sender, which is a member of the local group
19        AvgSpeed[msg.userId] = msg.userSpeed;
20      }
21    });
22  }
23  // else, the current onCreateGroup() is just a join group notification for a
        local group
24 }
```

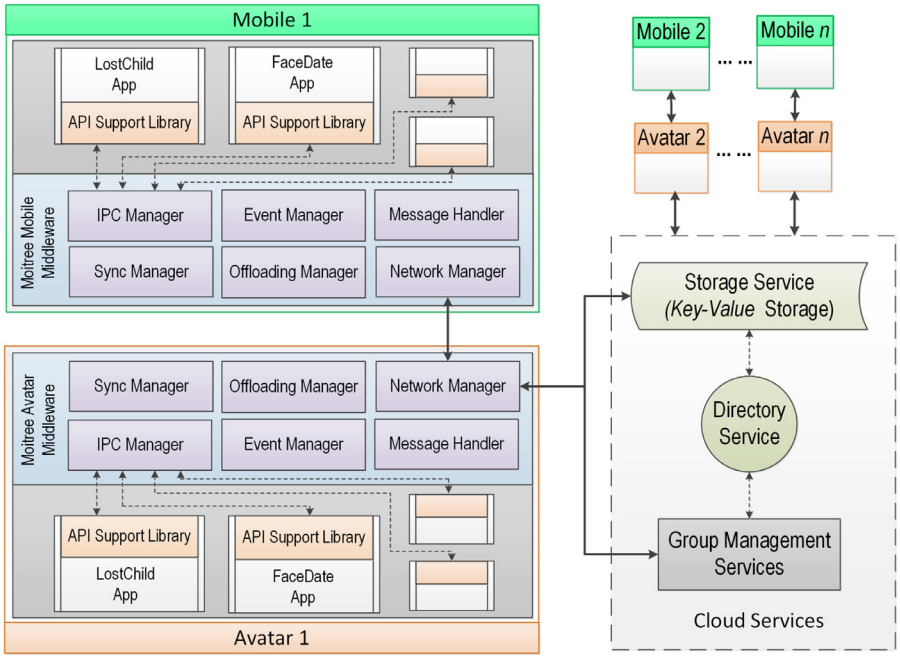


Fig. 2. Components of the Moitree Middleware.

The next pseudo-code shows how the members of a subgroup report speeds to the subgroup leader (line 10) and how the leader sends travel time updates to the main group leader (lines 5–7). This code is executed as long as the vehicles are still on the same road segment (i.e., members of the same subgroup). If they are not, Moitree removes them from the subgroup and the current *onCreateGroup* ends. A new *onCreateGroup* will be invoked once a vehicle enters the next road segment and Moitree joins it to that segment.

```
1 previousSegment = currentSegment;
2 while (currentSegment == previousSegment) {
3   if (isLeader == True) {
4     // compute the average travel time for the street segment, as the leaderr,
          based on data received from members of the local group
5     segmentAverage = computeAverage(avgSpeed);
6     // send the average travel time on the segment to the global avatar
7     mainGroup.sendToLeader(segmentAverage);
8   } else {
9     //send the current car speed to the leader of the local group
10    localGroup.sendToLeader(new Message(userID, userSpeed));
11  }
12
13  //periodically check if the car is still on the same segment or not
14  sleep(X);
15  currentSegment = getCurrentSegment(map, GetLocation());
16 }
```

The following pseudo-code, executed by all members of a subgroup, sets up the callback functions for receiving the broadcast from the main group leader (lines 2–8), compute the new paths and perform rerouting (lines 5–6). For the sake of brevity, this pseudo-code assumes that the new path is selected locally based only on information from the global map. In reality, there is one more step (not shown) that includes coordination with other members of the subgroup in selecting the new path to avoid congesting nearby segments.

```
1 //setup broadcast callback for the mainGroup. Each update from the global
      avatar will be received here
2 mainGroup.setReadCallback(new ReadCallback() {
3  public void broadcast(Message cityGlobalTrafficMap) {
4    //A new rerouting path is computed according to AR* algorithm; the driver
       will be guided to follow this path
5    newPath = computeARStarPath(getLocation(), destination,
       cityGlobalTrafficMap))
6    guideDriver(newPath);
7  }
8 });
```

5. Moitree middleware design

The design of the Moitree middleware has several important objectives, which are crucial for keeping the Avatar platform usable and efficient. First, it must maintain a stable and seamless mobile-avatar pair for each user. Second, the design must hide low level details of the underlying system. Third, it must effectively translate high-level Moitree API calls to low level instructions (i.e., provide execution environment). Fourth, it must manage provisioning for computation offloading. Finally, it must glue all the participating entities together by taking care of the communications among them.

5.1. Structure and components

Moitree is designed as a Message-Oriented Middleware (MOM) (Curry, 2004) to keep the system loosely coupled. With this design, operations and system state changes are driven by asynchronous messages and events.[2] An RPC-based design is not selected since RPC uses blocking calls and different system components are expected to be strongly coupled.

As shown in Fig. 2, the Moitree middleware runs on mobile-avatar pairs. In each mobile-avatar pair, the mobile device runs the Moitree Mobile Middleware (MMM), and the avatar runs the Moitree Avatar Middleware (MAM). The two components work collaboratively to efficiently handle events and messages, manage data synchronization, and route network communications, making mobile-avatar pairs a stable, efficient, and unified environment for apps to run. Both MMM and MAM expose the same API for applications. The sub-components of MMM and MAM are introduced below.

5.2. App execution

The *API Support Library (ASL)* is embedded in each app. It translates API calls to corresponding events/messages and then sends these events/messages to MMM or MAM, depending on where the API calls are made. Since Moitree APIs are asynchronous and based on callbacks, ASL is also used to register and keep track of callbacks. For example, when an app makes a *createGroup()* API call from the mobile, the ASL translates it to an event named *CREATE_GROUP* and sends it to the MMM. When a response to the call arrives, ASL uses the registered callback to deliver the response to the app.

With the message-oriented middleware structure of Moitree, the state changes in an app are driven by messages and events, as illustrated in Fig. 3. The state transitions are managed by ASL.

When a user launches an Avatar app on her mobile, the app enters its initial state – "Started", which is equivalent to Android's onStart life-cycle state. Once the instance of the app on avatar gets initialized and callbacks are successfully registered on both the mobile device and the avatar, the app enters the "Initialized" state. In this state, the app can react to events and messages. When the app receives and handles an "onCreateGroup" event through its group callback method, it becomes a participant in a distributed computation and enters a "Ready" state. In this state, an app can participate in multiple computations (and thus its user is a member of multiple groups). All groups are maintained in separate threads with separate data structures for improved reliability and isolation.

[2] In a MOM design, the term message refers to both events and messages. However, the paper uses events and messages to refer to two different types of messages, with events mainly for delivering Moitree control information or commands (e.g., creating a group or deleting a group member) and messages mainly for delivering app data (e.g., Alice's photos in the LostChild app).

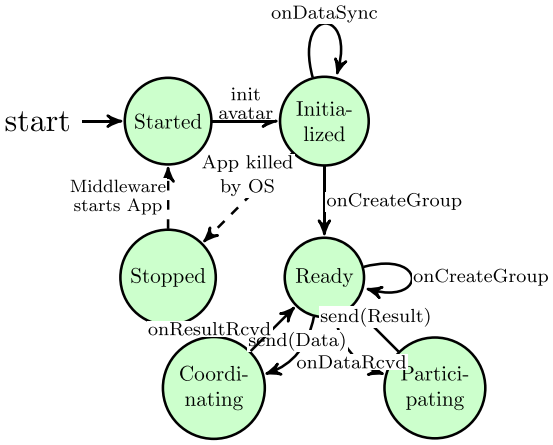


Fig. 3. App State Diagram.

The transition from the "Ready" state depends on whether the app runs at a group initiator or at a group participant. When group initiators start the first communication within the group, the app enters the "Coordinating" state. Once it receives all the results from participants, it goes back to the "Ready" state. When an app in "Ready" state at a participant receives data in its communication channel callbacks for the first time, it will enter the "Participating" state. After a participating app finishes the computation and sends the results back to the initiator, it goes back to the "Ready" state. Note that an app can have multiple active groups and those groups can be in "Coordinating" and "Participating" states. In this sense, "Coordinating" and "Participating" can be called group states rather than app states.

Android can kill the app if the mobile is short of memory. In that case, the app enters the "Stopped" state. The middleware can start the app again if necessary. These state transitions are shown with dashed lines in the figure.

5.3. App offloading

In the initial version of Moitree (Khan et al., 2016), apps used static partitioning to tell the middleware which parts are to be executed in the mobile and the avatar, respectively. In the current version of Moitree, we use the CASINO framework (Debnath et al., 2018), introduced in Section 2.4, to perform distributed computation offloading and scheduling. With this framework, programmers can mark any function or class with the *@Offloadable* annotation. During the execution, the CASINO uses mobile devices' profiling information (e.g., CPU, network, battery status, etc.) to decide whether an 'offloadable' component should be executed in the mobile or offloaded to the avatar.

CASINO provides a simple, but customizable way for synchronizing computational states between mobile and avatar. This is done by using distributed shared variables. Using CASINO's annotations, programmers can mark a variable with *@SyncrhonizedField*. This variable will be available in both mobile and avatar. The value is synchronized depending on how the programmer configures the synchronization policy, which can be either *lazy* or *eager*. The eager synchronization immediately makes the shared variable consistent in both mobile and avatar, but this naturally incurs more overhead. On the other hand, the lazy policy updates the value when it is requested/read. This has less overhead, but the latency is higher. Programmers can configure the nature of synchronization based on the use case and latency tolerance of the app.

We designed a scheduling algorithm to execute offloadable tasks in a sequence so that the overall completion time of the

distributed computation is minimized. The offloading framework interacts with Moitree through the *Offloading Manager* in Fig. 2. This middleware component provides dedicated communication support to offload tasks as necessary.

5.4. App communication and data synchronization

The apps, including the embedded ASLs, run in different processes from the mobile middleware/MMM or the avatar middleware/MAM. Therefore, inter-process communication (in the form of messages and events) is needed between apps and the middleware. The *IPC manager* (see Fig. 2) serves as the gateway that takes care of this communication.

The *Event Manager (EM)* and *the Message Handler (MH)* are in charge of handling events and messages, respectively. Each of them consists of a queue and a dispatcher. The queue is for organizing events/messages. The dispatcher watches the queue for incoming events/messages and dispatches them based on the meta-data embedded in their headers. EM and MH are designed following the observer pattern for a loosely coupled structure.

The *Network Manager (NM)* (see Fig. 2) is in charge of inter-device routing (i.e., route to an avatar or to a mobile device). NM performs object serialization/deserialization as necessary. The header fields used for inter-device routing are: *source* and *channelType*. Once the event/message is routed to the correct device, EM and MH are in charge of intra-device routing using the header fields *appId, groupId, channelId*.

The *Sync Manager (SM)* (see Fig. 2) takes care of data synchronization between the mobile and the avatar. Specifically, the apps specify the directories containing the data sets that are needed on both the mobile device and the avatar. These directories and files are synchronized by our overlay file system (Shan et al., 2016) that allows Moitree apps to access files concurrently at mobiles and avatars in a manner that is efficient, consistent, and transparent to locations (i.e., mobile or avatar).

5.5. Moitree cloud services

Moitree uses three cloud services, as shown in Fig. 2: *Group Management Service (GMS), Storage Service (SS), and Directory Service (DS).*

GMS is the most important, and it is designed to handle group operations, events, and communication. It runs as a cloud service on a group of dedicated servers, named GMS servers. As shown in Fig. 4, GMS consists of a group manager for maintaining the hierarchical structures of groups and a membership manager for maintaining the list of current members in each group. To support communication within a group, for each group, there is an event broker in charge of delivering events and a group communication manager for maintaining communication channels and forwarding messages to recipient avatars. The handling of events and messages is separated, since this helps preventing a large number of messages from delaying a few important events. GMS gives higher priority to event handling when allocating network resources because events are associated with important group/system state changes that must be reflected in real-time in apps.

To avoid a potential bottleneck in the system, GMS is designed to scale according to the load. When the number of groups managed by a GMS instance reaches a threshold, a new GMS instance is created. The middleware directs any new group creation requests to the new instance. All GMS instances can work independent of each other. For example, if the cloud consists of geographically isolated clusters, one GMS instance is instantiated in each cluster for serving group related requests for that region. If the load increases in any cluster afterward, multiple instances can be instantiated to serve the new requests.

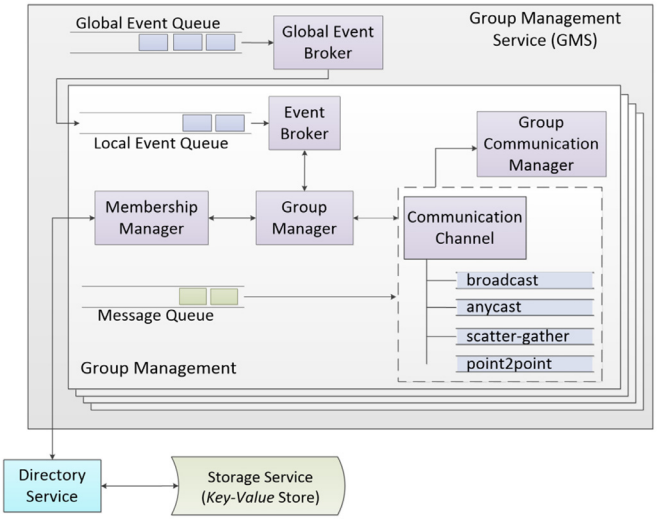


Fig. 4. Group Management Service.

There are alternative GMS designs that may help reducing the workload on GMS servers. For example, event/message forwarding can be offloaded to group leader avatars. However, this requires that group and channel information be duplicated to these avatars, leading to privacy concerns and additional overhead to maintain the information consistent. Another design is to save large messages in 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 adopted these designs. It is worth noting that privacy preserving operations (Borcea et al., 2016; Ghinita et al., 2007; Mokbel et al., 2006) can be used to hide users' location and context information from the cloud providers.

The Storage Service (SS) provides a shared and permanent storage space for the middleware and is implemented as a key-value store. SS maintains an app registry, which serves the purpose of finding which app is installed on which user's device and avatar. Other information about users (e.g., the locations where they have appeared and the corresponding time) is also stored in SS to assist the DS component. Finally, SS could be used for sharing large messages among participants in an optimized way. Note that each avatar has a virtual disk directly attached to it as its private and primary storage for the apps, which is not part of SS.

The Directory Service (DS) is used by GMS to select appropriate candidates for joining a group. It provides answers to queries such as "which users have the LostChild app installed and were present in Times Square between 5 PM and 6 PM today?". The mobile carriers can provide user location and time data to serve such queries. The directory service uses SS as its data repository.

5.6. Moitree app security model

Moitree apps use the same security model enforced by the Android platform. Moitree is designed to work at the application level, and it does not need any special system level permission. Moitree and all apps developed on top of it use the same permission model that regular Android apps use. In this way, Moitree ensures that all Android-enforced security and permission models are followed by Moitree apps.

Users install Moitree apps from the Moitree App Store. Although this type of app side-loading is permitted in Android, it skips the Google Play Store's security checking (e.g., malware

checking). There are two approaches Moitree can take to overcome this limitation: (i) introducing a similar security check for malware in Moitree App Store, (ii) distribute Moitree and its apps via Google Play Store. Either of them can be used in the future.

6. Moitree implementation

We have implemented a prototype of Moitree based on Java and Android. Various implementation techniques are used in different components. For example, the IPC manager is implemented based on the Binder mechanism of Android with each Avatar app working as a Binder client and the IPC manager as a Binder service. Compared to alternative implementation techniques, e.g., Android's BroadcastReceiver mechanism, the Binder mechanism offers faster communication. The network manager and GMS use a TCP library named Kryonet (2019) in their implementations. The Storage Service is implemented based on the Redis (2019) key-value database. In the rest of this section, we focus on implementation techniques that are general and can be applied to other programming languages and platforms.

Moitree runs on devices with different capabilities (e.g., smart watches, smart phones, tablets, and x86 virtual machines). The available main memory and maximum heap size are very different in such a wide variety of devices. For example, a Pixel smart phone has 4 GB of memory, whereas the Samsung Gear Live smart watch has only 512 MB of memory. Therefore, the footprint of Moitree has to be as small as possible, especially for supporting devices with small RAM size. To achive this goal, we made the following implementation decisions.

First, all the middleware components (except the networking module) are designed and implemented from scratch. We avoided using alternative open-source libraries due to their large footprints. Second, there is only one middleware instance per device. Only the API Support Library (ASL) is embedded in each app on the device. Furthermore, ASL contains only simple and lightweight parts (e.g., data definition, APIs, etc.), whereas the middleware instance contains the major functionalities. Finally, to further save memory, we implemented a different lower-footprint networking module for the Moitree instances running on smart watches.

7. Performance evaluation

To validate Moitree, we built and evaluated two apps: LostChild, already described in the paper, and FaceDate (Neog et al., 2016), which will be briefly discussed in this section. We also evaluated Moitree with micro-benchmarks. The evaluation has four goals: (1) verifying the effectiveness of the programming model in reducing programming complexity and effort by implementing two apps; (2) validating the app performance benefits when using avatars vs. using only mobiles by showing the improvements in running-time and resource usage; (3) testing the efficiency of the API support library; and (4) measuring the scalability, efficiency, and overhead of the middleware.

7.1. Experiment settings

We deployed the Moitree prototype in our OpenStack-based private cloud comprised of 8 servers with Intel Xeon E5-2620 CPU and 80GB of RAM. The avatars hosted in this cloud run Android x86 VMs, specifically Android Marshmallow version 6.0. Each VM is configured with 6 virtual CPU cores and 3 GB of RAM. We deployed GMS as a cloud service on a few servers, with one instance on each server. The number of servers running GMS is adjusted dynamically with the workload, and GMS requests are evenly distributed to these servers for load balancing. We used several types of Andoid mobile devices: Nexus 5, Nexus 6, Nexus 5X, and Moto X Pure. The API Support Library is distributed via our private *maven* repository; the apps import it with a single line of code in their Android build script (gradle).

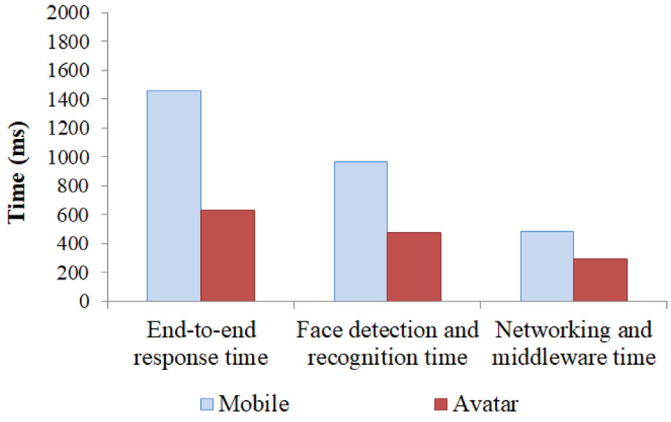


Fig. 5. End-to-end response time of *LostChild* app when major computation workload is at the mobile and avatar, respectively.

7.2. App performance and programming benefits

We evaluated the performance of two apps built using Moitree: LostChild and FaceDate (Neog et al., 2016). FaceDate is a mobile, location-based app that matches people based on their face preferences in real-time. Each FaceDate user uploads their profile face photo and trains the app with photos of faces they like. Upon user request, FaceDate detects other users located in the proximity of the requester and performs face matching in real-time. If a mutual match is found, the two users are notified and given the option to start communicating.

7.2.1. Running time improvement

We first measured the response time of the LostChild app in two scenarios: (1) the major computation workload in the app, including face detection and recognition, is handled on the mobiles, and (2) the major workload is executed at the avatars. Note that, in both scenarios, mobile to mobile communication is mediated by the Moitree services in the cloud.

In the experiments, we used one mobile device as the initiator of LostChild request and three other mobile devices as participants. Their avatars were instantiated on the same server. Each participant has a database of 47 images containing 60 faces stored in her avatar (i.e., some photos contain more than one face). 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.

Fig. 5 shows the end-to-end response time from submitting the initial request until receiving the final results, as well as the time spent on the major computation workload and the time spent on network and middleware operations. The results demonstrate that avatars help reduce the end-to-end response time by 51%, compared to the scenario in which the mobiles handle the major workload. A substantial part of the improvement is achieved by offloading the major computation workload to the avatars. We also observe that the time spent on Moitree and networking is reduced by more than 40% after offloading the major computation workload to avatars. This is because the network communications associated with the major computation workload move along with the workload and are conducted in the cloud.

We also varied the number of participants from 2–7, and varied the number of servers used to host the participant avatars in LostChild. It is worth noting that we wanted to study how the

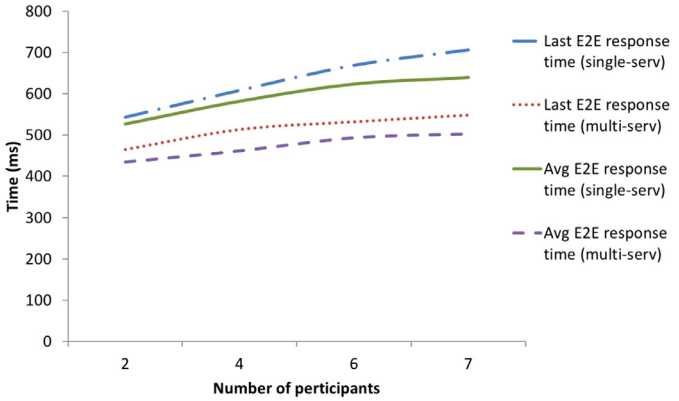


Fig. 6. End-to-end response time of *LostChild* app with an increasing number of participants. The curves marked with "single-serv" are obtained with all participant avatars running on the same server; and the curves marked with "multi-serv" are obtained with each participant avatar running on a different server.

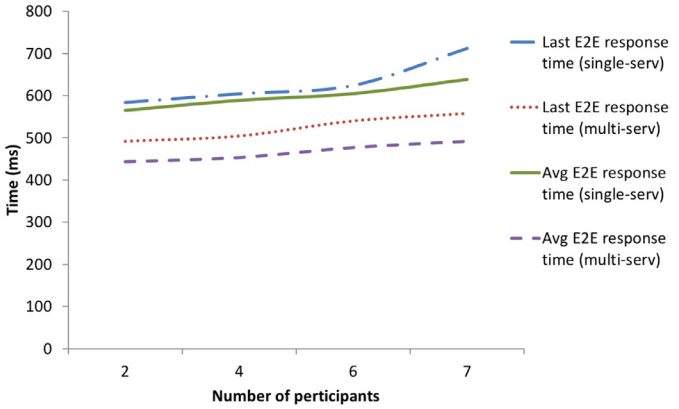


Fig. 7. End-to-end response time of *FaceDate* app with an increasing number of participants.

Fig. 8. End-to-end response time for FaceDate app with an increasing number of queriers and one positive responder.

response time varies when multiple avatars are instantiated in the same server and in multiple servers. Due to our cluster setup with 8 servers, we varied the number of participants from 2–7. Fig. 6 shows the response time when face detection and recognition were executed on avatars. The curves marked with "single-serv" are obtained with all participant avatars running on the same server; and the curves marked with "multi-serv" are obtained with each participant avatar running on a different server. Since the initiator receives multiple responses, one from each participant, the figure shows both the median response time and the longest response time. In a real-life situation, most participants are not expected to send responses. Therefore, the curves for the longest response time represent the worst case scenarios.

As the figure shows, the response times range generally between 450 ms and 700 ms, which are reasonable values. At the same time, the application scales well with the number of participants for this experiment. Increasing the number of participants only slightly lengthens the median response times. However, it has more impact on the longest response times. There are two reasons. First, our current Moitree implementation sequentializes the communication among the members and adds a few of milliseconds to every message transmission. Second, participant avatars do not send responses back to the initiator at the same time; for most requests, we noticed one or two straggler responses with extra-long turn-around times. We are currently optimizing the message delivery part of the middleware.

We also notice that, as expected, running one avatar per server reduces response times, especially when the number of participants is high, because running all avatars on one server may cause the avatars to compete for the computational and network resources on the server.

We have also done the same experiment with *FaceDate*. The result is shown in Fig. 7. The difference with the *LostChild* experiment is that, in this case, each user has fewer photos (6 images per user) to run face recognition on. As a result, the time needed for face detection is significantly lower. However, it has one extra hop of network communication between the mobile-avatar pair (for FaceDate's pair-wise matching), which is part of the app design and cannot be further minimized.

7.2.2. Benefits of reusing groups

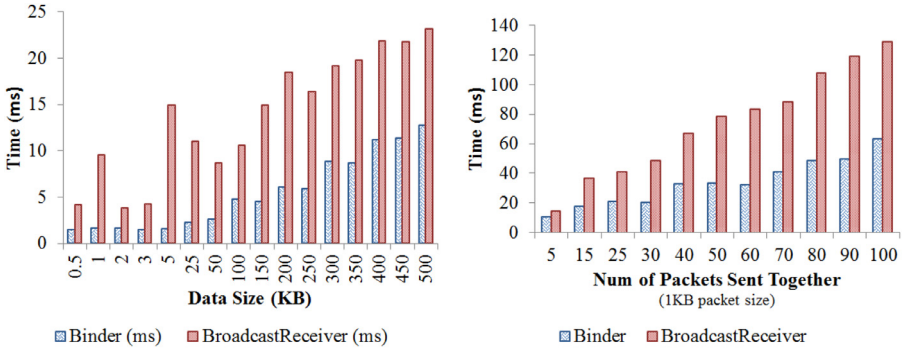
When multiple users initiate the creation of different groups described by the same properties (e.g., location, time), Moitree can verify if a group with the same properties already exists and simply return a reference to this group. Thus, the overhead associated with group creation and management is drastically reduced. This optimization is evaluated with the FaceDate app. For this experiment, there is only one participant who responds positively and there are a varied number of initiators/queriers. The participant is a member of all the groups initiated by the queriers. Fig. 8 shows the average end-to-end latency for the queriers. The results for the "No Group Reuse" case show a significant increase in latency for each addition of a querier because a new group is created for each new query. However, the latency for the "Group Reuse" case is substantially lower.

7.2.3. Programming effort comparison

We used the LostChild app to quantify the benefits of the Moitree programming model. The app can act as a good test for Moitree for two reasons. It represents typical mobile distributed apps that may involve a large number of mobile users. At the same time, its implementation must deal with the common issues, which are faced in developing other mobile distributed apps, such as, identifying and coordinating groups of participants.

We implemented two version of the app, one with Java and JXTA (Gong, 2001), and the other with Java and Moitree. JXTA is selected to compare to 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, which are different from those used in Moitree.

We compared the sizes of the source code of these two implementations. In this comparison, we only counted the lines written by our programmers. The code in other libraries (e.g., OpenCV, 2019 for face recognition and Kryonet, 2019 for network communication) is not counted toward the effort to develop the app. The app implementations include mostly group management



(a) Varied packet size (b) Varied packet numbers

Fig. 9. IPC latency of BroadcastReceiver-based mechanism and Binder-based mechanism.

and group communication features; the rest is done through library function invocations.

The implementation with Moitree has 85 lines, whereas the implementation with JXTA has 178 lines. Moitree decreases LOC by a factor of more than 2. Similarly, FaceDate implementation with Moitree has 109 lines, whereas the version without Moitree needs 231 lines. This is a promising result that illustrates how Moitree can simplify the programming of mobile distributed apps.

7.3. Stress tests on key operations

While Moitree shows good overall performance when tested with the two apps, we also wanted to understand if any of the Moitree components are potential performance bottlenecks when the middleware is under heavy workload. To investigate this issue, we tested Moitree with a few micro-benchmarks. The tests cover the major operations in Moitree for handling the interactions between the participants in an app, including inter-process communication, data serialization and de-serialization, and API calls.

7.3.1. IPC performance

For every event/message communication, several IPC calls take place throughout the system (e.g., between ASL and the middleware). This can potentially be a major bottleneck for real-time apps and apps that need a large amount of data communication. Thus, we tested the IPC performance in Moitree first.

In the Moitree prototype implementation, we tried two Android mechanisms for IPC. One is a lightweight Android mechanism named BroadcastReceiver (BR), which was used in an early version of the prototype. The other is a stable and efficient IPC mechanism named Binder, which is used in the current prototype. The experiments evaluated both IPC mechanisms to justify the use of Binder in Moitree.

We performed two different experiments using a test app, with one experiment testing latency and the other testing throughput. Specifically, to test the latency of IPC communication, we used the app to send a data packet to the mobile/MMM component of the middleware. Once the data packet is received by the MMM, it is sent back immediately. Using the round-trip latency, the app can get the latency of one way IPC. Fig. 9a shows the one way latency for both the BR-based and Binder-based mechanisms when the packet size is varied from 0.5 KB–500 KB. Generally, when the packet size is small (< 5 KB), the effect of meta-data is noticeable. Thus, the end-to-end latency does not increase significantly with packet size. However, when the packet size keeps increasing, the end-to-end latency starts to increase linearly, indicating that the IPC mechanism in Moitree is scalable to data size variations. The figure also shows that the IPC mechanism based on Binder incurs much lower latency than that using BroadcastReceiver.

To test the throughput of IPC communication, we used the app to send a batch of packets with the same size (1 KB) to the middleware, one after the other, which were then bounced back by the middleware. We varied the batch size from 1–100, and for each batch size, we measured the time from sending out the first packet to receiving the last packet. Fig. 9b shows that the time increases linearly with the batch size, indicating that the cost of Moitree handling each packet does not increase, and Moitree is capable to scale under heavy workload.

7.3.2. Serialization and deserialization performance

Through its high level API, Moitree receives and returns high level objects (e.g., Java objects). When these objects are exchanged through IPC and network communication, Moitree has to perform serialization and deserialization. These operations can be a potential bottleneck, particularly when objects are frequently exchanged.

In our implementation, objects exchanged between different Moitree components are written using the Parcelable (2019) interface for serialization/deserialization. For the objects exchanged through network across different devices, we used the Kry serializer (2019) in the Kryonet communication library. Both Parcelable and Kryo serializers are much faster than the standard Java Serializable interface (Serialization, 2019).

We used an app to test the performance of Moitree's serialization and deserialization. We ran an instance of the app on a mobile and another instance on an avatar. The instance on the mobile sends data packets of different sizes to the instance on the avatar. We measured the serialization time in the Network Manager of the mobile middleware and the deserialization time in the Network Manager of avatar middleware. Fig. 10 shows the time taken to serialize or deserialize each packet for packet size varied from 50 bytes to 800 kilobytes. Even for a packet as large as 800 KB, the serialization time is only a few milliseconds and deserialization time is under 1 ms, suggesting that Moitree can perform serialization/deserialization efficiently.

7.3.3. Performance of API support library

To test the performance of Moitree's handling of API calls, we used an app to issue a batch of API calls back-to-back from a mobile, and measured the end-to-end latencies of these calls. The APIs called in each batch were randomly chosen, and the batch size was increased from 10 to 100. The processing of each API call may involve many steps. Starting from the app layer, messages and events are generated and passed through the software layers in mobile and avatar as well as the GMS service layer before the results are returned. Thus, the experiment can assess the overall

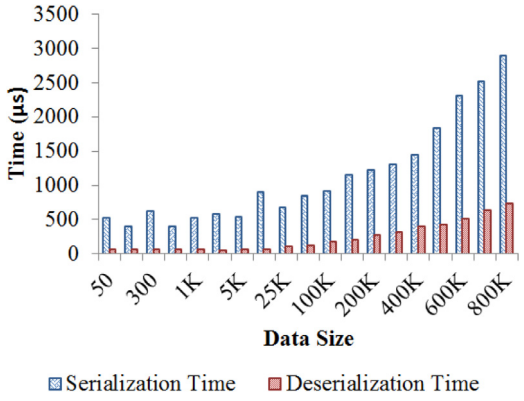


Fig. 10. Network serialization and deserialization delay.

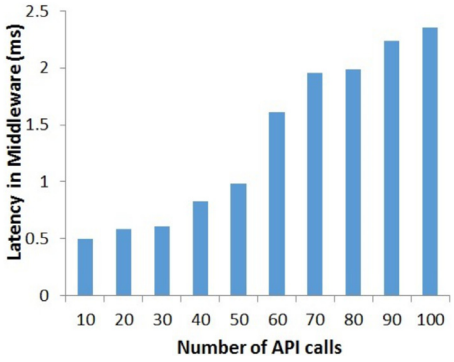


Fig. 11. Average end-to-end latency for concurrent API calls in Moitree (including network communication).

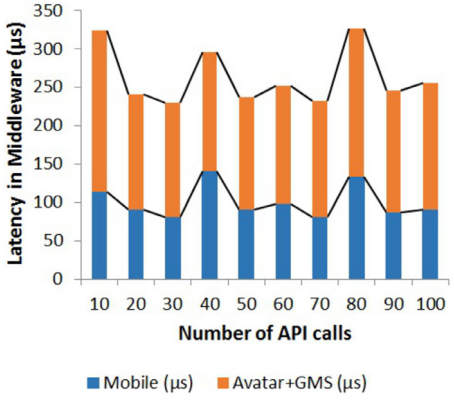


Fig. 12. Average processing time for API calls in Moitree on mobile and avatar (no network communication).

performance exhibited by various components in the middleware during the processing of the API calls.

Fig. 11 shows the average latency for each batch size. The average latency increases with batch sizes, since larger batches incur more network communication. When the batch size is 100, the average latency reaches 2.4 ms. Most of the time is spent on network communication (i.e., queuing and data transferring). To exclude the time on network communication, we instrumented the middleware to measure the network communication time. Fig. 12 shows the time incurred only by the middleware components without the part incurred by network communication. Processing an API call takes approximately 320 μs, and the time does not change much with batch sizes. The average time per API call is computed for each batch. This demonstrates that Moitree can efficiently handle API calls at a high speed.

7.4. Resource overhead incurred by Moitree

We want to understand how much resource Moitree consumes on mobile devices to ensure that Moitree does not affect user experience or other functionalities on mobile devices. We measured the resource overhead of Moitree during all its different phases, specifically (1) how much CPU resource and energy is consumed by launching and initializing the Moitree middleware, (2) after Moitree is initialized, how much memory resource, CPU resource, and energy Moitree middleware consumes, (3) when loading an app, whether using Moitree may introduce significant delay, and (4) during the exeucution of an app, how much memory, CPU, and energy Moitree middleware consumes.

Resource Usage for Initializing Moitree. When Moitree is launched on a mobile device, resources are needed to start various background services in the mobile middleware instance. This incurs little resource consumption, only 6% CPU usage and 0.16 mAh of energy. The energy consumption was measured using Qualcomm's Trepn profiler (Tre, 2019) on a Nexus 5X device with a battery capacity of 2700 mAh.

Resource Usage for Maintaining Moitree Services. After Moitree has been launched and initialized, memory is needed by the services in the mobile middleware (MMM) to maintain the data structures. At the same time, the services must wake up periodically to check data synchronization and communication requests and to maintain the mobile-avatar pairing. The MMM component consumes 22.60 MB of memory, which is very low when comparing to the DRAM capacities in popular Android phones (1 GB–3 GB). The CPU usage is minimal (about 1%) without requests from apps. As shown in Table 2, the energy consumption during this phase is also negligible (5.5mJ/s, or 2.5-month of usage to drain a fully charged battery).

Application loading time. An app running over Moitree must register with the IPC mechanism and initialize an Avatar object and other necessary data structures before it can call any Moitree API. This may increase the time used for loading an app. To measure the potential increase of loading time, we implemented an Android test app in two ways. First, we implemented it into a standard Android app without Moitree initialization, and measured the time taken to fully load the app with Android ActivityManager's log. The time was mainly spent on loading standard Android support framework and initializing and making the user interface visible. Then, we re-implemented the app with Moitree and measured the loading time spent on Moitree initialization.

As shown in Table 3, initializing Android's support framework and initializing user interface take 228.1ms and 41.9 s, respectively. Moitree-related initialization takes only 4.1 ms, which is less 2% of total app loading time, indicating that using Moitree only minimally affects the app loading time.

App execution energy consumption. As shown in Table 2, the energy consumed per average API call is very low. With a 2.3 mJ energy consumption for each call, a full battery charge allows one and a half million calls. When serving API calls, most energy is consumed on data transmission. As shown in Table 2, Moitree introduces a relatively high overhead (3.3x) on network communication when compared to using only plain TCP (0.5 mJ/KB with Moitree and 0.15 mJ/KB for plain TCP). This is mainly because Moitree uses the Kryonet (2019) communication library to simplify programming at the cost of increased energy consumption. The other reason is that MMM supports high level data communication channels, which introduce overhead, and makes routing decisions, which incur additional energy consumption on tasks such as queuing, dispatching, and routing.

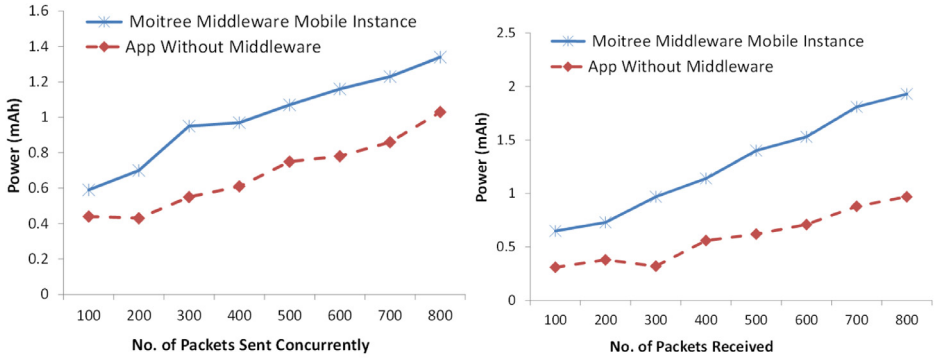
Since conventional apps may also use Kryonet for data communication, we wanted to know how much energy consumption is actually incurred by Moitree. For this purpose, we developed

Table 2
Moitree's energy consumption on phones.

Component	Energy Consumed	Comment
Initializing MMM	0.16 mAh	16,875 times with a fully charged battery
MMM in idle state	5.5 mJ/s	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 3
App initialization time.

	Android Framework Initialization (ms)	Moitree Initialization (ms)	Android App UI Initialization (ms)
Average	228.1	4.1	41.9
St Dev	23.6	0.5	5.0



(a) Sending packets (b) Receiving packets

Fig. 13. Power consumption overhead caused by sending and receiving data with a Moitree-based app compared to an app without Moitree support. The packet size is 50 KB.

two apps, which send and receive packets using Moitree APIs and Kryonet library, respectively. We run the two apps separately on a mobile device and measure the energy consumption incurred by the data communication in the apps using the Trepn profiler. The packet size is 50 KB. Fig. 13 compares the energy consumption incurred by the two apps sending and receiving packets, when the number of packets is varied from 100–800. Compared to the gap between Moitree communication and TCP communication (3.3x), the energy consumption gap between Moitree communication and Kryonet library communication is much narrower (50%–100%).

7.5. Computation offloading performance

Moitree uses CASINO (Debnath et al., 2018) for collaborative computation offloading and scheduling. CASINO works in two steps. First, it computes a near-optimal solution for the distributed scheduling problem, which can be shown as a $Q_m|prec|\sum_{j=1}^{n} C_j$ class problem. The schedule decides where the tasks of a distributed mobile-cloud app execute (i.e., at the avatars or the mobiles) and when they execute based on dependency constraints. Second, the tasks are offloaded as decided by the schedule and executed. The scheduling problem is NP-hard. By employing a greedy algorithm, CASINO can generate a good schedule in polynomial time: $O(mn^2 logm)$, where n is the number of jobs and m is the number of devices (i.e., mobiles and avatars). Although it cannot generate an optimal solution, CASINO achieves reasonably near-optimal results in a realistic time frame, which is essential for a dynamic offloading scheduler. Our evaluation of the scheduler in CASINO (Debnath et al., 2018) shows that using a greedy algorithm is a reasonable compromise between schedule optimality and execution overhead.

Table 4 shows the overhead introduced by CASINO's execution manager. For different state sizes (shown in column 1), the time needed to execute the offloaded computation varies dramatically (column 2). However, the overhead incurred by code interception and state initialization is minimal and kept almost stable (column 3). Code interception is needed to make the annotation-based API work (i.e., @Offloadable). The overhead needed to transfer the states locally (column 4) is even lower than the overhead of interception and state initialization. Across all the state sizes, the overhead is less than 2% of the execution time. This indicates that the runtime overhead of CASINO is very low.

8. Related work

Although assisting mobile devices with cloud resources is a very active research area (Satyanarayanan et al., 2009; Chun et al., 2011; Cuervo et al., 2010; Kosta et al., 2013; Zhang et al., 2014), Moitree is the first middleware for cloud-assisted mobile *distributed* apps. Recently, a few works have investigated cloud support for mobile distributed computing (Kosta et al., 2013; Zhang et al., 2014). Clone2Clone (Kosta et al., 2013) 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 (Zhang et al., 2014) 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

Table 4
Overhead Introduced by CASINO's Execution Manager.

State Size (Kb)	Execution Time Including Offloading (ms)	Overhead - Interception and State Initialization (ms)	Overhead - State Sync (ms)	Overhead Percentage
237.31	3212	2.11	0.06	0.06%
61.36	664	2.75	0.07	0.40%
36.44	490	2.93	0.08	0.61%
6.70	145	2.79	0.06	1.97%

management of non-functional app features. It should be noted that Moitree is not used 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 (Lin et al., 2008; Liu et al., 2005; Mamei and Zambonelli, 2004; Murphy et al., 2006). Among the middleware for distributed programming over mobile ad hoc networks (MANET), LIME (Murphy et al., 2006) and TMACS (Lin et al., 2008) propose group abstractions similar to Moitree. LIME (Murphy et al., 2006) provides a framework in which mobile agents can form groups based on context-awareness. Moitree's programming model has two main advantages over LIME: it provides more flexible communication abstractions, and its supporting middleware performs transparent dynamic group management. TMACS (Lin et al., 2008) proposes an object-oriented distributed middleware framework for MANET. In Moitree, groups are defined based on users and their activities rather than the types and scopes of objects as in TMACS. This makes mobile distributed programming simpler and more natural. MELON (Collins and Bagrodia, 2014) 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 (Brouwers and Langendoen, 2012) and MobiSoC (Gupta et al., 2009) are closer to Moitree because they use server-side resources to provide middleware platforms for specific areas of mobile computing. Pogo (Brouwers and Langendoen, 2012) 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 (Gupta et al., 2009) 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.

CAMCS (OSullivan and Grigoras, 2016) presents a mobile-cloud middleware which uses a software entity called CPA (similar to avatars) in the cloud. CPAs represent mobile users within the mobile-cloud platform and use cloud-based services to complete tasks assigned to them in a disconnected, asynchronous fashion. Although CAMCS is similar with Moitree in using CPAs, it does not provide a programming framework for collaborative computating among a group of users.

Other mobile-cloud middlewares and frameworks have been proposed to secure resource discovery (Reiter and Zefferer, 2016), allocate response resources during disaster scenarios (Guerdan et al., 2017), support mobile crowdsensing (Girolami et al., 2017), provide hierarchical trust management protocols (Guo et al., 2017), and use IoT sensors in the cloud (Das et al., 2017). Moitree differs from all of them in one important aspect: it provides programming and execution support for mobile distributed apps assisted by the cloud.

9. Conclusion and future work

To the best of our knowledge, Moitree is the first middleware for mobile distributed apps assisted by the cloud. Even though the concepts of Moitree are general and applicable to any distributed mobile cloud platform, we have designed and implemented it for our Avatar platform. 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 with faster response times and lower energy consumption on mobile devices at the cost of a reasonable latency overhead.

As future work, we plan to merge Moitree with our other systems that support the Avatar architecture: the CASINO (Debnath et al., 2018) framework for dynamic offloading of Moitree apps, the OFS (Shan et al., 2016) file system that allows Moitree apps to consistently and concurrently access and share files at mobiles and avatars, and the P2F2 (Almalki et al., 2016) system for privacy-preserving face finding in mobile cloud apps.

Acknowledgment

This research was supported by the NSF under Grants No. CNS 1409523, SHF 1617749, and DGE 1565478, and by DARPA/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, DARPA, and AFRL.

References

Almalki, N., Curtmola, R., Ding, X., Gehani, N.H., Borcea, C., 2016. P2F2: privacy-preserving face finder. In: Proceedings of the 37th IEEE Sarnoff Symposium, pp. 214–219.

Android Parcelable Interface, 2019. https://developer.android.com/reference/android/os/Parcelable.html, online, accessed May 28.

Baumann, A., Peinado, M., Hunt, G., 2014. Shielding applications from an untrusted cloud with Haven. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI '14, pp. 267–283.

Borcea, C., Ding, X., Gehani, N., Curtmola, R., Khan, M., Debnath, H., 2015. Avatar: mobile distributed computing in the cloud. In: Proceedings of the 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud). MobileCloud '15, pp. 151–156.

Borcea, C., Talasila, M., Curtmola, R., 2016. Mobile Crowdsensing. Chapman and Hall/CRC.

Brouwers, N., Langendoen, K., 2012. Pogo, a Middleware for mobile phone sensing. In: Proceedings of the 13th International Middleware Conference. Middleware '12, pp. 21–40.

Chun, B.G., Ihm, S., Maniatis, P., Naik, M., Patti, A., 2011. CloneCloud: elastic execution between mobile device and cloud. In: Proceedings of the 6th Conference on Computer Systems, EuroSys '11, pp. 301–314.

Collins, J., Bagrodia, R., 2014. Mobile application development with MELON. In: Proceedings of the 13th International Conference on Ad-hoc, Mobile, and Wireless Networks, Vol. 8487 of ADHOC-NOW 2014, pp. 265–278.
Cuervo, E., Balasubramanian, A., Cho, D.k., Wolman, A., Saroiu, S., Chandra, R., Bahl, P., 2010. MAUI: making smartphones last longer with code Offload. In: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services, MobiSys '10, pp. 49–62.
Curry, E., 2004. Message-oriented middleware. Middleware for communications. Wiley Online Library, pp. 1–28.
Das, R.B., Bozdog, N.V., Bal, H., 2017. Cowbird: a flexible cloud-based framework for combining smartphone sensors and IoT. In: Proceedings of the 5th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering. MobileCloud '17, IEEE, pp. 1–8.
Debnath, H., Gezzi, G., Corradi, A., Gehani, N., Ding, X., Curtmola, R., Borcea, C., 2018. Collaborative offloading for distributed mobile-cloud apps. In: Proceedings of the 6th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering. MobileCloud '18, pp. 87–94.
European union's general data protection regularion, 2019. https://ec.europa.eu/commission/priorities/justice-and-fundamental-rights/data-protection/2018-reform-eu-data-protection-rules_en, online, accessed May 28.
Ghinita, G., Kalnis, P., Skiadopoulos, S., 2007. PRIVE: anonymous location-based queries in distributed mobile systems. In: Proceedings of the 16th International Conference on World Wide Web. WWW '07, pp. 371–380.
Girolami, M., Chessa, S., Adami, G., Dragone, M., Foschini, L., 2017. Sensing interpolation strategies for a mobile crowdsensing platform. In: Proceedings of the 5th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering. MobileCloud '17, IEEE, pp. 102–108.
Gong, L., 2001. JXTA: A network programming environment. IEEE Internet Comput. 5 (3), 88–95.
Guerdan, L., Apperson, O., Calyam, P., 2017. Augmented resource allocation framework for disaster response coordination in mobile cloud environments. In: Proceedings of the 5th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering. MobileCloud '17, pp. 45–52.
Guo, J., Chen, I.R., Tsai, J.J.P., 2017. A mobile cloud hierarchical trust management protocol for IoT systems. In: Proceedings of the 5th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, MobileCloud '17. IEEE, pp. 125–130.
Gupta, A., Kalra, A., Boston, D., Borcea, C., 2009. Mobisoc: a Middleware for mobile social computing applications. Mobile Netw. Appl. 14 (1), 35–52.
Khan, M.A., Debnath, H., Paiker, N.R., Gehani, N., Ding, X., Curtmola, R., Borcea, C., 2016. Moitree: a Middleware for cloud-assisted mobile distributed apps. In: Proceedings of the 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering. MobileCloud '16, pp. 21–30.
Kosta, S., Perta, V.C., Stefa, J., Hui, P., Mei, A., 2013. Clone2Clone (C2C): peer-to-peer networking of smartphones on the cloud. In: Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud '13. USENIX.
Kryo serializer, 2019. https://github.com/EsotericSoftware/kryo, online, accessed May 28.
Kryonet, 2019. https://github.com/EsotericSoftware/kryonet, online, accessed May 28.
Lin, J., Shing, E., Chan, W.K., Bagrodia, R., 2008. 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. Mobiquitous '08, pp. 21:1–21:12.
Liu, J., Sacchetti, D., Sailhan, F., Issarny, V., 2005. Group management for mobile ad hoc networks: design, implementation and experiment. In: Proceedings of the 6th International Conference on Mobile Data Management. MDM '05, pp. 192–199.
Mamei, M., Zambonelli, F., 2004. Programming pervasive and mobile computing applications with the TOTA Middleware. In: Proceedings of the 2nd IEEE Annual Conference on Pervasive Computing and Communications. PERCOM '04, pp. 263–273.
Mokbel, M.F., Chow, C.Y., Aref, W.G., 2006. The new casper: query processing for location services without compromising privacy. In: Proceedings of the 32Nd International Conference on Very Large Data Bases. VLDB '06, pp. 763–774.
Murphy, A.L., Picco, G.P., Roman, G.C., 2006. LIME: A coordination model and middleware supporting mobility of hosts and agents. ACM Trans. Softw. Eng. Methodol. 15 (3), 279–328.
Neog, P., Debnath, H., Shan, J., Paiker, N.R., Gehani, N., Curtmola, R., Ding, X., Borcea, C., 2016. Facedate: a mobile cloud computing app for people matching. In: Proceedings of the 11th EAI International Conference on Body Area Networks. BodyNets '16, pp. 184–190.
News, C., 2019. Police caught murder suspect with the help of tourist photos. http://www.cbsnews.com/news/police-tourists-took-photos-of-san-francisco-pier-murder-suspect/, online, accessed May 28.
OpenCV, 2019. http://opencv.org/, online, accessed May 28.
OSullivan, M.J., Grigoras, D., 2016. Context aware mobile cloud services: a user experience oriented Middleware for mobile cloud computing. In: Proceedings of the 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering. MobileCloud '16, IEEE, pp. 67–72.
Paiker, N.R., Shan, J., Borcea, C., Gehani, N., Curtmola, R., Ding, X., 2017. Design and implementation of an overlay file system for cloud-assisted mobile apps. IEEE Trans. Cloud Comput. doi:10.1109/TCC.2017.2763158.
Pan, J., Popa, I.S., Borcea, C., 2017. Divert: a distributed vehicular traffic re-routing system for congestion avoidance. IEEE Trans. Mob. Comput. 16 (1), 58–72.
Qualcomm, Trepn power profiler, 2019. https://developer.qualcomm.com/software/trepn-power-profiler, online, accessed May 28.
Redis, 2019. http://redis.io/, online, accessed May 28.
Reiter, A., Zefferer, T., 2016. Flexible and secure resource sharing for mobile augmentation systems. In: Proceedings of the 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering. MobileCloud '16, IEEE, pp. 31–40.
Satyanarayanan, M., Bahl, P., Caceres, R., Davies, N., 2009. The case for VM-based cloudlets in mobile computing. IEEE Pervas. Comput. 8 (4), 14–23.
Serialization performance evaluation, 2019. https://github.com/eishay/jvm-serializers/wiki, online, accessed May 28.
SGXenabled, 2019. http://www.intel.com/content/www/us/en/processors/core/6th-gen-core-family-desktop-brief.html, online, accessed: May 28.
Shan, J., Paiker, N. R., Ding, X., Gehani, N., Curtmola, R., Borcea, C., 2016. An overlay file system for cloud-assisted mobile applications. Proceedings of the MSST. 1–14.
Zhang, I., Szekeres, A., Aken, D.V., Ackerman, I., Gribble, S.D., Krishnamurthy, A., Levy, H.M., 2014. Customizable and Extensible Deployment for Mobile/cloud Applications. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, pp. 97–112.

Hillol Debnath received his Ph.D degree in 2017 from New Jersey Institute of Technology, and his BSc Engineering degree from Bangladesh University of Engineering & Technology. His research interests include mobile computing, mobile sensing, programming frameworks, and mobile-cloud middleware.

Mohammad A. Khan received his Ph.D degree from New Jersey Institute of Technology in 2016. His research interests include distributed programming, network analysis, and machine learning algorithms.

Nafize R. Paiker received his his Ph.D degree from New Jersey Institute of Technology in 2019. His research interests include mobile computing, cloud computing, parallel and distributed computing.

Xiaoning Ding is an Assistant Professor at New Jersey Institute of Technology. His interests are in the area of experimental computer systems, such as distributed systems, virtualization, operating systems, and storage systems. He earned his Ph.D. degree in computer science and engineering from the Ohio State University.

Narain Gehaini was a Professor of Computer Science at New Jersey Institute of Technology. Previously, he was with Bell Labs. Narain has worked extensively in programming languages, software, and databases. He has authored several software systems, holds several patents, and has written many books and numerous papers in computer science. Narain got his Ph.D in computer science from Cornell University.

Reza Curtmola is an Associate Professor in the Department of Computer Science at NJIT. He received his PhD degree in Computer Science in 2007 from The Johns Hopkins University and spent one year as a postdoctoral research associate at Purdue University. He is the recipient of the NSF CAREER award. His research focuses on storage security, applied cryptography, and security in wireless networks.

Cristian Borcea received his Ph.D. degree from Rutgers University in 2004. He is currently a Professor with the Department of Computer Science, New Jersey Institute of Technology. He is also a Visiting Professor with the National Institute of Informatics, Tokyo, Japan. His research interests include mobile computing and sensing, ad hoc and vehicular networks, distributed systems, and cloud computing.