PiMiCo: Privacy Preservation via Migration in Collaborative Mobile Clouds

Kaushik Ravichandran  
Georgia Institute of Technology  
kaushikr@gatech.edu

Ada Gavrilovska  
Georgia Institute of Technology  
ada@cc.gatech.edu

Santosh Pande  
Georgia Institute of Technology  
santosh@cc.gatech.edu

Abstract

The proliferation of mobile devices and mobile clouds coupled with a multitude of their sensing abilities is creating interesting possibilities; the sensing capabilities are creating different types and fidelities of data in a geographically distributed manner that can be used to build new kinds of peer-to-peer applications. However, the data generated by these mobile devices can be personal and of a highly confidential nature. While very interesting possibilities exist for collaborating on the diverse, shared data in real time, privacy policies on the data sharing, transport, as well as usage must be clearly specified and respected. The goal of this work is to introduce a privacy preserving data centric programming model for building collaborative applications in large scale mobile clouds and discuss its design.

Our work introduces several concepts and leverages privacy annotations and a transparent execution migration framework to achieve our goals. We also present an evaluation using several applications demonstrating that overheads are minimal and can be used in a real-time setting.

1. Introduction

Mobile clouds are a combination of cloud computing systems and mobile networks [29,30]. Mobile clouds facilitate the execution of rich mobile applications on a plethora of mobile devices, in which the mobile devices themselves provide huge amounts of compute power. The individual mobile devices which the mobile cloud comprises of also produce large amounts of data. Typical data sources include on board sensors and information individually obtained from the network. Consider for example the data generated from the on board sensors; a large variety of sensing capabilities are creating different types and different fidelities of data in geographically distributed manner. There are an ever increasing number of on board sensors in today’s mobile devices ranging from GPS, accelerometers, temperature sensors to even oximeters. The information generated from these sensors create very interesting possibilities opening the door to building new kinds of peer-to-peer applications that span multiple devices. Such multi-device applications can collaborate in real time seamlessly utilizing many geo-spatio-temporal properties of the information opening interesting application possibilities. However, the information generated on such devices is often of a highly confidential nature and personal making this a challenging problem.

Current privacy specifications take a 0-1 approach towards the data; either data is available or is unavailable under a given policy. The current specifications do not allow reasoning about finer levels of data properties, some examples of which are: whether a particular view or usage of data is allowed, the context under which the usage is legal or illegal or whether the data can be combined with certain other data in a certain form and with certain operators, whether the data could be shared with certain entities under certain context or conditions and in turn whether such an entity can share it with other entities (friend of a friend). In addition, data originating on mobile devices exhibits peculiar properties associated with the underlying sensors, and those of time and space; typically such a data is associated with meta-information such as timeliness, and currency; optimizations must be done to attain these properties to build such applications. Further, in order to preserve the strict privacy policies associated with personal data, runtime systems must perform object migrations and orchestrate their movements across the devices in highly efficient manner to alleviate the overheads. Also, in mobile clouds, huge parallelism exists in the ensemble of thousands of participating devices which could be leveraged in a different manner to be able to achieve: rapid pruning of the seas of information and data and build on shared each other’s partial results to discover interesting artifacts in real time.

This work first focuses on the specification of policies with regard to data sharing and exchange on a peer-to-peer basis amongst collaborating Android Mobile Devices. In particular, we present programming idioms to specify data and meta-data properties and privacy and trust constraints through simple annotations on top of existing Android applications. Such an application is then be compiled into software for efficient execution onto mobile
devices enforcing the underlying privacy policies associated with the data that are stipulated by the owners of the device. We present an “owner computes” rule which means computation migrates from a device that generates or owns the underlying data; many interesting optimizations are needed to achieve this efficiently.

The applications that we target are collaborative applications that run in the mobile cloud environment. Communication in the cloud can be achieved through the construction of an overlay network for the devices (similar to Project JXTA [34]). The collaborative model facilitates the development of threads that execute on different devices and are able to build on each other’s state to achieve very fast pruning of data space. The privacy preserving model along with computation migration allows one to move the computational state from device to device building on partial results.

We also discuss how migrations are performed and the techniques we employ to make migration transparent, seamless and efficient to the programmer finally presenting some results on representative applications. We refer to our system as PiMiCo.

2. PiMiCo Programming Model Overview

In this section we present an overview of the PiMiCo programming model. Figure 1 presents our design. The programmer annotates the Android program with privacy annotations. These annotations describe the policies that will be applied. Policies may be generated from a ‘Trust Base’ or ‘Social Graph’. For example, a policy may restrict usage of a data item to only a person’s friends.

A static compiler takes this annotated input and generates a privacy preserving, migration based program. These programs automatically migrate based on data item accesses and policy specifications using our migration framework.

Further, these individual programs are capable of collaborating by sharing information and building on each other’s execution state.

We now discuss the design of each of the main components in detail.

3. Privacy Annotations

Recall that the data distribution identifies the “owner” of the data (which is typically the smartphone or the device) and the policy specifies how it can be shared across the boundaries of ‘ownership’ amongst untrusted collaborating owners. A compiler and trusted data sharing and exchange substrate (see Figure 1) provisions for enforcing the data-flow of shared data as per the policy, and generates the necessary glue code for the movement of shared data via the substrate and determines where the necessary privacy enforcing ancillary computation takes place. In many cases, the server or cloud can enforce the privacy policy to making sure certain data-flows are never allowed; this system configuration assumes that the server is to be trusted with regard to the shared data. However, in some cases, a privacy policy might mandate that “extremely sensitive” data can never be taken out of the device ever or even the cloud or server is not to be trusted with regard to such data in such cases. In such cases, all the computation with regard to

![Figure 1: Programming Model Overview](image-url)
such data “must” be done locally. Under such policies the Android objects will migrate to the device and execute locally on the said device under the supervision of its local Dalvik VM. Dalvik is the VM in the Android operating system which executes Dalvik byte code (created from the original source).

3.1. Data Sharing Policy

Following are the assumptions we use for our mobile device system which are realistic settings encountered in practice for data sharing:

- A device may or may not be able to directly access the data present at another device as per the sharing policy of the data. If the data is publicly available or server is trusted, it can be kept in the server or the cloud (even if it not just read-only, it can be kept up-to-date with respect to its local copy on a device). However, as explained above, if the data policy does not allow taking the data out of the device ever, then any other device which wants to use it, must authenticate itself and can retrieve only those parts of data that are allowed under the access control under the supervision of local Dalvik after migrating its execution to the device that owns the data.

- The annotations identify the access control, privacy and trust issues apart from data distribution. For example, a given data (variable) might be available only at a given processing node. Moreover it could be made available only to a few devices through authentication and they may not be allowed to keep a copy of it but only use it in a certain manner. In some cases due to issues of privacy, only certain fields of data can be made available. For example, identity revealing fields of the device or its owner may not be accessible except to law enforcement such as in handling 911 calls. Only certain fields which cannot (either individually or collectively) reveal identity are allowed to be publicly available. That is if one field is accessed, another cannot be and vice-versa. In short, the access is context dependent. In some other cases, only certain views of data are allowed through a function interface (for example, a device might be able to check if someone's house is within a certain geographic area but will not get access to exact street address etc. due to privacy issues). In certain other cases, data rights are tagged to the data, and are securely propagated along with. These are then used for determining if the data is to be shared or not with a third entity.

3.2. Annotation and Policies

The goal of our compilation framework is to optimize efficiency (speed) of enforcing security, privacy and ownership properties of data expressed by data distribution annotations. Following are some examples of data access policies in our design. In the following discussion, an entity is either a user or a device. The trust is established by a relation. Examples of relations are social graphs (such as in Facebook, or LinkedIn) or they could be organizational (colleagues working in a given organization) or geographic (such as families living in a given sub-division).

- Each data item has a list of entities which are its are ‘owner entities’ who have control over value modification. In some sense these act like principals who own the data and can do value modification to it. Here is a sample annotation which specifies that the ‘meeting_date’ variable is owned by anyone who is participating in the meeting.

```
@owners [meeting_participants[]]
```

- Each data item has a list of trusted entities who are allowed to access its raw (actual) value in entirety using simple access control mechanism; these are trusted entities who can produce a certificate (such as X.509), whose credibility is verified, to get a value without going through the chores of full-fledged heavy duty access control. The following example provides gps access to people in the friends list.

```
@raw-access [friends[]]
gps current_location;
```

Any owner can write and destroy the current value of the data item at any time; it must inform about destruction of old value to all the entities who have copies but is not responsible to give new values. All other entities can get the value on demand from this entity from this point onwards.

- A trusted entity may or may not be able to hold the accessed value (from the other) beyond the point of its consumption. In other words, once the use of the value at trusted entity is over, the value must be destroyed or in some cases it might be retained as per the policy. This is specified using the @single-use annotation.

- If a value is retained, it might be shared with another one who is in the list of trusted entities; only the ‘owner entity’ can set the list of trusted ones and this list cannot be modified by any other entity. This is specified with @trusted-user-shareable. If a value is shared with another trusted entity, all the ‘owners’ are informed of this sharing. In other words, the ‘owners’ have an updated list of all trusted entities who have the current copy of the data item.

- In another case, a value in its raw form in its entirety may not be shared with an entity. The entity must first authenticate and then either an ‘owner’ or its ‘trusted entity’ can allow it to access a relevant part of it. This is described by using separate @raw-access and @owners annotations on different sub-elements of the data object.
• A policy can also preclude a value being accessed (in part or in its entirety) in raw form. In such a situation, an agent will get to use the value only under a given operation. In other words, only an operational view of a value is given to a device (and not the value itself). Different devices could be given different views. This is an especially powerful feature as it can be used in situations with especially sensitive data which simply cannot be transferred outside a device, say for legal reasons.

```java
@accessor-function is_closeby()
gps current_location;
```

The above annotation specifies that the current_location variable can only be accessed through the accessor function is_closeby.

• Finally, a policy can be context sensitive in the dynamic sense. That is, if access is granted to one datum, it could be denied/restricted to another “related” piece of data.

```java
@mutex-group A
float employee_salary;
@mutex-group A
string employee_name;
```

The above annotation specifies that the above two data fields are mutually exclusive. In that if an execution wants the employee salary it will not be able to retrieve the name and vice-versa. Some applications need this level of privacy when dealing with employee information to ensure k-anonymity property which requires that: "Given person-specific field-structured data, produce a release of the data with scientific guarantees that the individuals who are the subjects of the data cannot be re-identified while the data remain practically useful."

As one can see, most of the above privacy specifications can be defined via simple annotations on the shared data values using predefined types at static or dynamic time except the last one. For implementing the last one, we introduce a concept of conditional mutant types. The key idea is as follows: upon accessing a given data, the type of a “related” data item changes (mutates). Such a mutation disallows it being accessed via another request. The representation and implementation of a mutation is unique to each program and is a compiler internal and is protected as a read-only value to the outside world. This assures the security of it being maliciously changed by other non-trusted entities. Policy attributions can be inherited and modified via the class hierarchy mechanisms.

3.3. Compiler Analysis for Privacy and Trust

We now discuss the design of the compiler which uses the above annotations. The compiler performs the following analyses and code generation to implement the data sharing policies under the trust constraints:

It first classifies and then partitions the entities into different groups based on relations. As explained the goal is to keep free flow of values within a fully trusted base and push the authentication functions and transformers of values at the boundaries while enforcing the dependency constraints of the underlying computation and implementing the trust property. The idea is to generate maximal partition to be able to minimize this overhead of authentication at boundaries.

It then implements the necessary computation that is privatized on this data. Standard compiler techniques such as forward slices determine which computation should be privatized within the trusted boundary. A computation is privatizable provided its data-flow is completely satisfied by the values inside the trusted boundaries. The rest of the computation will be implemented outside the trusted boundaries.

The compiler analysis then implements the conditional mutant types on related data items. As discussed such types are marked read-only and a secure implementation is necessary. The security of these types is ascertained by the fact that they are only accessible to local privacy enforcement mechanisms and could be maintained in encrypted form to prevent tampering.

Finally, the compiler implements the glue code and the access control mechanisms at boundaries to implement the sharing and data transformation mechanisms. It also implements the calls to the data exchange. The data exchange implements the privacy policy. If the data is share-able through the server or cloud, it implements the sharing function in the server and at device end, simply generates data-synchronization calls to the server or cloud to keep the values consistent. If the data is to be kept strictly within a device though, it implements the sharing function locally on the device. The compiler implements the accesses to such private data by first checking if the access is local or non-local. For non-local accesses, it calls an object migration system to migrate and load a remote object onto the local Dalvik which executes under its supervision. We now discuss in detail our migration mechanism which provides a seamless and transparent migration from the programmer’s perspective.

4. Execution Migration Framework

We leverage the Multiverse framework [31] to perform execution migration. The Multiverse framework provides a base which allows migrations to happen transparently and seamlessly without any programmer intervention. This allows the programmers to write code similar to how they would have written a simple traditional program and our framework will
automatically perform the migration of the application state between nodes as needed based on privacy constraints.

4.1. Overview
We first briefly describe how the Multiverse framework performs transparent migration; we refer interested readers to [31] to get full details. The Multiverse framework transfers the execution state of an application at the migration point to another node. The execution context is described by the various components of the memory and the architectural state of the processor on which it is running. Multiverse creates copies of these and sends them over to the other node. Only the stack is sent initially and the heap is copied on demand. Such a mechanism prevents the unnecessary transfer of memory especially since the part of the code which is executing on the remote node will be small. By transferring the execution as is, and by restoring at the same virtual addresses on the remote node, no memory translation needs to be performed (for example, for pointers) and the execution can continue transparently. Such a mechanism prevents the unnecessary transfer of memory especially since the part of the code which is executing on the remote node will be small. By transferring the execution as is, and by restoring at the same virtual addresses on the remote node, no memory translation needs to be performed (for example, for pointers) and the execution can continue transparently. This requires no programmer intervention and is orchestrated by the compiler when it detects a section needs to be migrated for privacy reasons. Under this scheme only the functions which were determined to be needed to execute remotely are automatically shipped to the remote node, execute and come back while minimizing data transfers by performing on-demand transfers. Figure 2 represents this setup.

4.2. Transparency and Efficiency
Multiverse transfers execution state transparently between devices. It does this even without any programmer intervention. To this it copies virtual memory pages from one node and restores it on the other node at exactly the same memory address. This allows executions to continue unhindered on the other node. To do this efficiently it adopts an on-demand transfer mechanism. Memory locations are initially protected by using the mmap system call on the remote node. Whenever, accesses are made to those memory regions a fault is handled by the system and the original memory is fetched from the original node and remapped into the memory space. This allows transparent migrations of executions between devices while respecting the privacy annotations that the programmer has specified.

4.3. Optimizations for Migration
The necessity of migrating objects to the devices that own the private data can pose substantial overheads due to transfer of data for Dalvik VM objects. In order to reduce the overheads, the number of migrations must be minimized. Note that while data and control dependences prescribe the legal program order that must be obeyed, the r-values and live ranges govern the use points of the data. The goal of the compiler analysis is to first determine which computation must take place where based on the data ownership and create essential migrations to carry out the execution. But the interesting challenge occurs when the partial results are generated that need to be disseminated to all the participating devices that “use” them under the dependence constraints. The compiler’s goal is to create a dissemination agent that could maximally sweep the live range of partial results under dependence constraints. Consider the following excerpt of a collaborative program:

\[
p = q.\text{check}(m); \quad (1) \\
= \ldots \quad (2) \\
= p.\text{field1}; \quad (3) \\
= \ldots \quad (4) \\
= p.\text{field2}; \quad (5) \\
= \ldots \quad (6)
\]

In the above example, object q, queries a sensitive piece of remote data m by migrating to its owner device and executing the check query there; the partial result is generated in p. In this example, it is also assumed that the state of object q is exactly identical on all devices or object q is stateless an assumption which is true in many cases. As per the original program order one must execute the program in the order (1) through (6), thus, the partial result must migrate back to the owner of q. In order to get the partial results for each q on each device, one must either get it regenerated for each object q by sending it to owner of data m or have an ability to disseminate the
partial result p. The former solution is infeasible and the device will crash if thousands of objects (q’s) visit it. In the current example, this is also unnecessary since objects q’s are identical or are stateless across all the devices. The solution adopted therefore focuses on disseminating the partial result generated to all participating devices. Such a solution saves a lot of migrations (for each of the q’s) as well as tremendous burden on the device that owns ‘m’ but this scheme has one drawback. Due to the asynchronism of execution (programs that are getting p may not even be at step (1) for example), in such cases a lots of copies of the partial results are created ahead of time due to a large extension of the respective live ranges. In other words, devices may receive partial results p’s even though they are not at step (1) in their execution which is akin to the effective upward motion of statement (1) in local execution context for example. In order to reduce the effect in the asynchronous setting, it is best to reduce the live range for effective shorter spans of partial result objects. Care must also be taken that local copy of q is not destroyed before its last use of course a condition that can be established by proper synchronization. Thus, program transformations that move statements (3) and (5) closer to (1) under the dependence constraints are desired to reduce the burden caused by copies.

5. Putting It All Together: An Example

In this section we present an example of a mobile application which needs to maintain privacy.

Consider a collaborative application which alerts a user when any of his friends are nearby. Friends will most certainly be uncomfortable revealing their location at all times but will be more willing to alert their friends when they are close by (say within half a mile radius or so). For such an application it is important that the raw physical location of a user never leaves his own device. Code from another application will be permitted to run on this device allowing it to securely access the physical location of the device after it has been filtered by a user-defined function and authenticated to run on local Dalvik.

The design of this user defined function is such that it may just return a true or false value, indicating whether both devices are within half a mile radius of each other or not. Consider a scenario in which user B is trying to determine if user A (who is his friend) is nearby or not. In such a scenario, member function of an object B (belonging to user B) will first migrate to the user A’s device and execute under its local Dalvik to seek a yes/no answer given by the function’s implementation. This answer is then carried back to user B’s device. Thus, the GPS data never leaves user A’s device and access to its local GPS resource is only granted to its local function that are verified and are authenticated as per the setting of local Android security policy. This implementation thus, conforms to the way Android security mechanisms are designed. Such an implementation also does not reveal the raw physical location of the device at all for larger distances and will only return a generic true response in case the distance criterion is satisfied. The detailed implementation of this example is illustrated in figure 3 below.

```java
public class mobile_agent {
    @accessor-function is_closeby()
    @read-access friends[] location[] friends_locations; // GPS
    bool[] friends_closeby;
    double distance(location location1, location location2) {
        returns the distance between the two locations.
    }
    // Note that friend_location belongs to an array of privacy sensitive "location"s.
    // And is annotated with an accessor-function specification and a read-only specification.
    // Since this code access a privacy sensitive object the code is automatically migrated to the correct device after acquiring permissions.
    // Every invocation of this function will run on a different device.
    bool is_closeby(
        location my_location,
        location friend_location) {
        // This code executes on the remove // device.
        if (distance(my_location, friend_location) > 0.5) {
            return false;
        } else {
            return true;
        }
    }

    void find_all_closeby_friends() {
        for (i in range(0, friends)) {
            if (is_closeby(my_locations, friends_locations[i]) {
                friends_closeby[i] = true;
            } else {
                friends_closeby[i] = false;
            }
        }
    }
}
```
void main() {
    mobile_agent mobileAgent;
    mobileAgent.find_all_closeby_friends();
}

The pseudo-code in Figure 3 above illustrates a simple situation where code needs to migrate to access privacy sensitive information. The array location[] friends_locations is an array of the raw geographical position information of all the friends (GPS data). This data member is annotated to indicate that it is in fact distributed as per the entire friends[] array which is a distributed array and importantly it is also annotated as privacy sensitive allowing access only through an accessor function is_closeby. Note that the friends[] array itself is not annotated as privacy sensitive.

The privacy sensitive annotation automatically tells the compiler that any code which accesses this information needs to be run on the remote device itself and it cannot run locally. Using this information the compiler can automatically infer that the function "is_closeby" needs to run on the remote device without any additional programmer intervention. The runtime system will take care of acquiring any necessary permissions to run on the remote device and handle the migration of the code to the remote device and back. The transparent migration framework takes care of efficiently transferring the execution state between machines. Another point to note is that every instance of the is_closeby function runs on a different device. The compiler and the runtime orchestrate these migrations with the compiler statically determining the regions of code that need to be migrated and the runtime performing the actual migrations.

6. Evaluation

In this section we present an evaluation of the migration overheads in our prototype. We wrote four real world applications to measure different properties.

We performed all our experiments on a mobile cloud of 144 entities connected by a mobile network. The set-up was Dalvik nodes consisting of Samsung Galaxy Notes connected to mobile servers and while some were peer-to-peer connected. The mobile servers consisted of AMD Opteron 8431 processors and 64GB of RAM. The server nodes were running RedHat Enterprise Linux 6.3. All applications are compiled with gcc 4.4.6 with the O3 flag.

Migrated code needs to be signed to prevent those with malicious intentions from hacking the system. There is an execution time cost but with current processor speeds it is minimal.

**Application 1:** implements the pseudo-code that was presented in Figure 3. The application attempts to find which friends are close by. This necessitates the migration of the execution to each of the individual nodes and then back. At the end of the execution the application has the count of friends who are nearby. This application allows us to measure simple migration overheads and its different components. This application was run on 50 devices. Here are the average migration overhead results we obtained. One can see that the total time is much lesser than a fraction of a second meaning that the application executes in near real time.

<table>
<thead>
<tr>
<th>Component</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Migration</td>
<td>38.92</td>
</tr>
<tr>
<td>Migration Start</td>
<td>24.37</td>
</tr>
<tr>
<td>Stack Transfer</td>
<td>6.01</td>
</tr>
<tr>
<td>Execution on Remote Node</td>
<td>4.35</td>
</tr>
</tbody>
</table>

**Application 2:** implements an application which needs to transfer large amounts of data between devices while maintaining privacy. In this application, a user is allowed to go and fetch a copy of photo or audio or video file from his friend but he/she must do so, by first migrating to the device that “owns” the data and then authenticating and then transferring. We evaluate the overheads that transferring image, audio and video content incurs. We chose these types of media since these are in fact some of the most personal pieces of information on mobile devices and often access can only be granted in a privacy preserving manner. The image was an image of a flower (1765 px * 1602 px) encoded in the JPEG format. The audio was an MP3 file encoded at 256kbps using MPEG ADTS layer III in stereo. The video file was an MPEG sequence, encoded with v1 (system multiplex). Again, one can see that in spite of the large workloads, one can see that the time involved is less than a fraction of a second, implying that the application executes in real time.
Application 3: implements an application where each device must contact only the “owner” of the data to get it. In other words, this piece of data being very sensitive, a device is able to get a copy only from the owner. Due to this requirement, a scenario is created where a large number of devices are simultaneously communicating with a single node trying to acquire some information on that node. This situation simulates a “hot-spot” and describes how the latencies involved in migration can increase in situations where hot-spots arise. In this example an image is being requested from several devices. Again one can see that inspite of hotspots the total time of execution of the application is much smaller than a fraction of a second. The application in this scenario again executes in realtime for even as high as 8 concurrent requests creating a hot spot on a device.

Application 4: implements a situation where a node is seeking out some information and it tries a particular node first. However, by the time node’s request reaches another node, the context changes and the data is no longer available at that node for dissemination (such situations are common when the data is highly time sensitive and may not be current). Hence, this execution needs to search other nodes for the particular privacy sensitive information. The request keeps visiting different nodes and thus the latency of the request increases as it attempts to visit different nodes. We show demonstrate how the execution time increases as the number of nodes the original program has to contact increases (due to context changes). Again, we see how in spite of larger number of searches the execution time always remains less than a second, demonstrating suitability in real time situations.

7. Related Work

Mobile cloud architectures [29, 30] can be roughly classified into two types. In the first setting mobile devices just use the cloud resources but do not provide any services. The second, is a cooperation or collaborative paradigm where data centers and mobile nodes share resources and partake in a computation. Our work focuses on the second type and aims to address privacy issues in this setting.

Privacy is critical in the design of large scale systems which deal with personal information. Many techniques which work on the principle of anonymizing or hiding the source have been proposed [1,2,3,4]. However, our approach works by preventing unfettered access to sensitive information in the first place and only exposing acceptable pieces of information. Mun et al. [5] introduce the notion of Personal Data Vaults which can be used to filter access to sensitive information but at a courser granularity. Our approach incorporates such principles but exposes these using simple annotations at the code level which automatically ensure privacy is maintained. Yang et al. [27] introduce a language called Jeeves, a language for automatically enforcing privacy policies. Resin [28] allows checking code to be executed at output channels. Myers et al. [25] employ programmer annotations but focus on how to ensure that private information is not leaked beyond trusted recipients. However, these systems deal with the privacy issues in a 0-1 manner and do not address the role-based or relation based privacy that allows only select views or
Several schemes [6, 7, 8] discuss privacy and data aggregation in wireless sensor networks. Bertino et al. [9] present an overview of privacy and data management scheme for geospatial data. Terrovitis [10] discusses the importance of privacy preservation of location data in the context of communication devices such as smart phones which provide an abundance of such information. Much of the data on mobile devices is extremely personal and can simply be not transmitted out of the device in any form; this creates a mandate that in a distributed environment, the code needing the data must migrate to it and execute under local VM environments; migration can be very expensive and in the privacy setting poses unique constraints – how to optimize the migrations under the privacy and dependency constraints is one of the important techniques proposed in this work; to our knowledge this important issue has not been addressed by any of the above work or in literature. Funf [17] is an Android framework which can be used to collect sensor data. Kun et al. [26] discusses the nuances of migration and employs compiler techniques to optimize migration points in a program.

Device-cloud interactions are a hot topic of research lately; most of the work explores aspects offloading functionality or parts of code to the cloud. The CloneCloud system [18] offloads certain parts of an execution on to the cloud based on compiler techniques and profiling. The MAUI system [19] also performs method shipping on to a remote cloud. Other systems which pre-partition applications between remote and local execution include [20, 21, 22, 23, 24]. However, the key question of how to leverage the “common states” of thousands of threads (that are performing essentially similar computation – such as route determination in traffic example) is a very important problem to be solved – to save resources as well as speed up the computation as well as reduce energy. This aspect has been left out by all the research on offloading or computation outsourcing type of work.

The collaborative aspects of the system rivals traditional parallel models and provides a novel alternate view of parallelism, moving away from the traditional paradigm of breaking the work down. Traditional parallel models include OpenMP, which is an API providing for the development of portable scalable shared memory parallel applications. It provides a programmers with simple and flexible interface for developing shared-memory parallel programs [11]. Intel Thread Building Blocks, or TBB, offer programmers with a C++ library with which they can write parallel programs easily and lets programmers program in a higher-level task-based parallelism [12]. Intel Concurrent Collections, or CnC, allows programmers to specify the higher-level structure of the program in C++. In the CnC model, programs are expressed in terms of steps, tags and items. NESL is another programming model which makes writing parallel programs easy and portable [13]. X10 [15] and Chapel [16] provide abstractions of partitioned global memory (PGAS) on a large scale. Unlike the above, the collaborative model promote collaborations between “processes” using the cloud using the notion of shared address space for meta-information. The privacy preserving model along with computation migration allows one to move the computational state from device to device building on partial results. Since the meta-information is not actual data that we are sharing, we can relax the issues of synchronization and hope to be able to scale to thousands of devices and their processing requests.

Multiverse’s [31] approach of transparently transferring execution state across machines bears similarity to that of PM2 [32] and Charm++ [33].

Cheriton [35] also introduced similar concepts in the System V distributed system for object code on a LAN. Chen et al [34] present a technique in which Java byte codes migrated within a P2P network with an itinerary.

9. Conclusion

In this work we tackle the issues which hinder the development of collaborating mobile cloud applications which use information of personal or highly confidential nature. We presented a privacy preserving data centric programming model for building collaborative applications in large scale mobile clouds and discussed its design. We introduced several concepts which leverage privacy annotations and transparent execution migration framework to achieve its goals.

We proposed some important compiler optimizations to minimize the number of migrations; the migration overheads are substantially reduced by the system. We have evaluated the framework on some key sample applications that stress test the overheads and response time. In each of the instances, the execution time of the application is well below a fraction of a second proving that the application executes in real time. It is thus, empirically shown that it is possible to build privacy preserving collaborative application in mobile cloud that span multitudes of devices that execute in real time. High efficiency can be made possible by the highly optimizing compiler transformations and due to the high execution
efficiency offered by the Multiverse migration framework.

10. Acknowledgements

The authors would like to thank the anonymous reviewers for their comments and suggestions. The authors also gratefully acknowledge the support of NSF grants CNS-1320752, CCF-1018544 and CNS 1148600.

11. References


