The Journal of Systems and Software 92 (2014) 20–31

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Mosco: a privacy-aware middleware for mobile social computing☆

Dinh Tien Tuan Anh*, Milind Ganjoo, Stefano Braghin, Anwitaman Datta

School of Computer Engineering, Nanyang Technological University, Singapore

A R T I C L E I N F O

Article history:
Received 15 October 2012
Received in revised form
16 November 2013
Accepted 22 November 2013
Available online 8 December 2013

Keywords:
Fine-grained access control
Social computing
XACML
Google App Engine

A B S T R A C T

The proliferation of mobile devices coupled with Internet access is generating a tremendous amount of highly personal and sensitive data. Applications such as *location-based services* and *quantified self* harness such data to bring meaningful context to users' behavior. As social applications are becoming prevalent, there is a trend for users to share their mobile data. The nature of online *social networking* poses new challenges for controlling access to private data, as compared to traditional enterprise systems. First, the user may have a large number of friends, each associated with a unique access policy. Second, the *access control policies* must be dynamic and fine-grained, i.e. they are *content-based*, as opposed to all-or-nothing. In this paper, we investigate the challenges in sharing of mobile data in social applications. We design and evaluate a middleware running on Google App Engine, named *Mosco*, that manages and facilitates sharing of mobile data in a privacy-preserving manner. We use Mosco to develop a location sharing and a health monitoring application. Mosco helps shorten the development process. Finally, we perform benchmarking experiments with Mosco, the results of which indicate small overhead and high scalability.

1. Introduction

The time of ubiquitous computing seems to have finally arrived. As computing devices are getting increasingly smaller, cheaper, more connected and more powerful, they gradually become indispensable to everyday life. In particular, smart phones equipped with numerous sensory capabilities, always-on network connectivity and powerful CPU have enjoyed a remarkable growth during the past few years. They can record user activities with their own sensors (GPS, accelerometer, etc.) or act as a portal to receive data from other devices (speedometer, heart-rate monitor, etc.) through short-range wireless communication.

Social computing has successfully latched on this trend and enjoyed a rapid growth. In the social computing paradigm, user behavior in social context is collected and analyzed by computing systems to derive new values to individuals, as well as new societal insights that benefit the community. While Facebook, Google+ , Twitter, LastFm bring individuals together using the off-line social connections, applications such as FourSquare, PatientsLikeMe, Nike+ , exploit data from ubiquitous devices to add meaningful context to facilitate social interactions. Other applications like PIER (Mun et al., 2009), CarTel (Hull et al., 2006) combine sensor data from users to generate real-time pollution and traffic reports which benefit the society as a whole.

An important premise to social computing is data sharing, either amongst friends (social networks) or to third parties (for societal services). However, sharing in social applications is challenging, because the nature of data and of the applications demand a rigorous treatment of user privacy. In particular, controlling data access in these settings is more troublesome than in traditional enterprise systems. First, a user may have many friends and connections, each associated with a unique access policy. Second, the policies are highly dynamic and fine-grained, that is they are content-based as opposed to the static, all-or-nothing policies. The vast amount of data, combined with a large number of users and complex social connections, add to the difficulty in designing privacy-aware social applications.

For applications that depend on data generated from mobile devices, user privacy must be addressed with foremost priority, because the data is of highly personal and sensitive nature. Two popular social applications that illustrate the needs of more fine-grained access control are location sharing (Foursquare, Find my friends) and quantified self (Quantified, Nike). In the former, a person may want to hide information from another based on their proximity or the time of day. One may also want to blur the location by concealing parts of the address, or to report only the statistics (number of check-ins at a particular place). In the latter, an

☆ This work was supported by A*Star TSRP *pCloud* project (grant no. 102 158 0038). Dinh Tien Tuan Anh, Milind Ganjoo and Stefano Braghin contributed to this work when they were at NTU Singapore.

* Corresponding author. Tel.: +65 96745847.
E-mail addresses: ug93tad@gmail.com (D.T. Tuan Anh), milind@gmail.com (M. Ganjoo), stefano.braghin@gmail.com (S. Braghin), anwitaman@ntu.edu.sg (A. Datta).

outpatient user may want to share his location and vital-sign readings to the doctors only when his heart rate exceeds a normal threshold. To his friends, insurance companies or research institutes, only the average readings per hour are revealed.

While the abundance and availability of data induce more social applications to appear, in many cases, different applications are created using different platforms and technologies. Even though cloud computing (Armbrust et al., 2009; Amazon; Google) can deliver the underlying computing infrastructure on-demand, these applications will need to be designed and written from scratch. We believe that there is an immediate need for a middleware designed specifically for social computing applications. Besides being scalable in handling large numbers of users and large amount of data, such a middleware will shorten the development and deployment process, at the same time provide easy mechanisms for addressing user privacy concerns. More specifically, it will come with easy-to-use, extendible interfaces for specifying and enforcing fine-grained access control with respect to other users of the system. Note that an end-user may also want privacy from the underlying service providers. The scenario of untrusted underlying service provider is interesting and more challenging (see for instance Tuan Anh and Datta, 2012 for a more comprehensive discussion on the system/privacy design space), but it is beyond the scope of the presented work.

In this paper, we present Mosco, a middleware designed for privacy-preserving mobile social applications. Specifically, the middleware runs on top of Google App Engine, thus the storage and management of data are done automatically by the cloud in a scalable manner. Mosco provides privacy with respect to the end-users, but it assumes that the service providing cloud platform is trusted. Though there are lot of ongoing research on securing services against untrusted cloud service providers (Tuan Anh and Datta, 2012), most real-life deployed applications are based on trusted services, and Mosco's aim is to augment such existing services with richer functionalities. Mosco accordingly makes it easier for developers to avail themselves of the middleware's primitives to easily develop applications which would allow end-users to specify dynamic and fine-grained access policies, that are efficiently enforced. It achieves this by extending the XACML framework. It provides template implementation for a core set of fined-grained access policies, so that new policies can be easily integrated. With data access being handled within Mosco, the application developers can turn their focus to the data semantics and application logics. Mosco provides an interface for data definition which can be readily extended for new applications. As a consequence of these, the development and deployment process are considerably shortened, while the resulting applications guarantee user privacy. These properties of Mosco are showcased in our implementation of a location sharing and a health monitoring application.

In summary, our contributions are as follows:

- We identify common scenarios for social computing applications that necessitate fine-grained access control.
- We present the design and implementation of Mosco (source code can be found at http://code.google.com/p/mosco), a middleware for developing privacy-preserving mobile social applications. To the best of our knowledge, Mosco is the first of its kind. The middleware runs on Google App Engine and utilizes XACML for specification and enforcement of fine-grained access control policies.
- To demonstrate Mosco's capabilities and flexibility, we implement two representative mobile applications using the middleware: a location sharing and a mobile health application. Mosco provides storage and access to data in a scalable manner and shortens the development and deployment process. Additionally, it allows users to share data in a flexible, secure manner.
- We benchmark Mosco using both real and synthetic data. The results suggest that it can scale gracefully with more users and more data, and that the overhead introduced by the access control mechanism is small.

The remainder of the paper is organized as follows. The next section describes motivating examples of mobile social applications, and presents the core set of fine-grained access control policies. Next, we detail the mechanism for defining and enforcing such policies, especially the implementation of policies in XACML framework. Section 4 presents the design of Mosco. Section 5 follows with the implementation details of two mobile social applications, and results from the benchmark experiments with Mosco. Section 6 highlights related areas of our work. Finally, Section 7 concludes and discusses avenues for future work.

2. Access control for mobile social computing

2.1. Motivating examples

There exists a plethora of mobile social applications, each providing a different social service either to the individual users or to the ensemble community. They rely on users to share data generated from mobile devices, which gives rise to concerns about data access control. In this section, we present some example applications which help identify and highlight the needs for highly dynamic, flexible and content-based finer-grained access policies.

2.1.1. Location sharing

Existing location-based social networks such as Foursquare, FindMyFriend, or check-in service (Facebook) employ all-or-nothing sharing policy of user location. Given that one's location is a sensitive piece of data, it is important to be able to determine not only to whom the data is shown, but also how much of the data is shown.

Consider that Alice is on a night out on a weekend, and she would rather avoid sharing her location with acquaintances at this time, except for friends who happen to be *nearby* so that they could be able to find her and meet up. This involves matching locations of Alice and her friends to determine if they are in the same neighborhood or within a certain distance from each other. Allowing Alice to set such a similarity metric allows her to dynamically differentiate users who can and cannot see her location. During a workday, Alice is at work and is willing to share her locations so that her colleagues can find her during office hours. However, outside of office hours, she would rather her colleagues do not know her whereabouts. This can be achieved by defining a *time window*, so that certain friends can see her location only when their requests are made within this window. Alternatively, Alice may specify a set of locations as *workplace* and enable her friends to see her only when she checks in to one of those locations. When traveling on vacation, she may want to reveal the complete street address to her close friends and family, while sharing only the region or the country she is visiting to her other friends. In this case, Alice must be able to specify the *granularity* at which her data is revealed, so that some friends may see her exact locations while others only an approximate one.

2.1.2. Quantified self, or mobile health monitoring

The Quantified Self movement advocates the use of technology to record and analyze users' daily behavior. Its applications range from fitness (Nike), sleep pattern (Take), mood change (Moodscope) to medical conditions (PatientsLikeMe, Curetogether). At the current state of the art, users share their data to their friends in a coarse-grained, all-or-nothing manner. This practice might be acceptable for non-physiological data, but will fail to meet privacy requirement for clinical health data.

Lets say Bob has been diagnosed with a chronic heart problem and seeks to better manage his condition using wearable devices (Zephyr) to monitor his heart rate and blood pressure. These data along with his location is collected every 10 min and can be shared as it is to his family, who is deeply interested in the state of his condition. Since personal activities can be inferred from the data, Bob may only want to alert his physicians of abnormal signs, i.e. when his heart rate is unusually high or the pressure is unusually low. Thus, Bob needs to be able to set a *threshold value*, that ensures data is revealed only if its value exceeds the threshold. Bob also takes part in a clinical trial and undergoes some experimental treatments. The research institute overseeing the trial will be interested in the improvement or degradation of his condition, for which Bob would only want to share some statistical information such as average and 95th percentile readings on a daily basis, or even add some noise to obfuscate the precise values before sharing the data. To achieve this, Bob can specify a *sliding window* of one-day size which returns the appropriate information from within the specified window. Alternatively, he can set a granularity level which determines how much noise to add to the data. Finally, Bob has to interact with his insurance company to claim back the medical costs. The insurance company would like to be able to verify if his visits to hospitals were necessary. For this purpose, Bob may only want to reveal the maximum (or minimum) readings, which justifies his visits. To achieve this, Bob needs to restrict the insurance company to see only the statistics (max value, in this case) of his data.

2.1.3. Participatory sensing

Applications of this kind are based on sensor data collected from voluntary participants. In essence, participatory sensing pushes the tasks of data collection (and even possibly some basic processing) to the edge users, while mainly focusing on data analysis. PIER (Mun et al., 2009) collects air quality measurements to generate pollution warnings of the unhealthy areas, and also to offer insights for urban planning. Traffic monitoring and management likewise benefits from cars sharing their speeds and locations (Hull et al., 2006; Hoh et al., 2008). These systems assume that users are readily willing to share their data, which is over-optimistic. Hoh et al. (2008) and Cornelius et al. (2008) offer privacy to the participants by protecting their anonymity. However, it is not sufficient to hide user identity, as exposing sensor data unnecessarily can still reveal sensitive information. For example, a user's identity can be exposed by identifying the most frequently traveled route as the route from his house to work.

Suppose Alice commutes to work by driving. Her car is equipped with a multi-sensor device that can record her location, speed, energy consumption, air quality as well as road surface condition. She may want to share her speed and road conditions at pre-defined, usually congested junctions to the transportation authority, so that the latter can re-route the traffic in real-time or plan to expand the roads. However, Alice would rather not reveal her energy consumption which can be used to infer her car model. In this case, Alice must define a *filtering policy* restricting access to specific part of her data only when she comes within pre-defined regions. If Alice takes part in a research on a new energy-efficient fuel, the interesting information to share is how much fuel her car spends over certain distance. For this, Alice would only want to share her average fuel consumption per unit of distance traveled. In particular, Alice may combine a filtering policy which hides the other data fields, with a *summary policy* that shares only the statistics over each day. Finally, to support environmental initiatives, Alice would like to share her commute routes with her colleagues in order to identify opportunities for car-pooling. But she would like to share her route only to those whose routes significantly overlap with hers. To this end, Alice may want to define a *similarity policy* granting access only to requesters who provide inputs similar to her data.

2.2. Access control

The examples above illustrate that coarse-grained, all-or-nothing data sharing is insufficient for many mobile social applications. They further demonstrate that a *white-list* approach to access control which focuses on the question of what to share and with whom is preferable to the *black-list* approach that is mainly concerned with whom and what *not* to share. In the following, we delineate variables that characterize the access scenarios described above: *access policy* (or policy), *access subject* (or subject) and *policy combination*. An access control mechanism is secure if the subject only gets access to the data defined by its associated policy.

For simplicity, we assume user data has the following schema:

$\langle \text{userid} \rangle \quad \langle A_0 \rangle \quad \langle A_1 \rangle \ldots \langle\rangle$

where $A_i \in A$ is the ith attribute whose values belong to an ordered (possibly multi-dimensional) domain.

2.2.1. Fine-grained access policy

The set of all possible fine-grained access policies in social mobile applications may be very large. In Mosco, we consider the following set of four *primitive* policies. More complex policies can be achieved by composing these primitives. We do not claim that this set is complete, nevertheless it covers a wide range of access policies.

- **Filtering policy** is defined as the tuple (F, D) where $F = \{(c_i, A_i)\}$ specifies boolean functions $c_i(A_i)$ for $A_i \in A$. $D \subseteq A$ is a set of attributes which are returned if all functions in F are evaluated to `true`. In the location sharing example, Alice may create a policy $(\{c_i($`timestamp`$)\}, \{$`location`$\})$, where c_i`(timestamp):=timestamp>=9:00 am AND timestamp<=05:00 pm`. Similarly, in the health monitoring example, Bob's policy can be $(\{c_i($`blood_pressure`$)\}, \{$`blood_pressure`$\})$, where c_i`(blood_pressure):=blood_pressure > =t` for a threshold value t.
- **Granularity policy** is defined as $G = \{(g_i, A_i, T_i)\}$ where $g_i \in \mathbb{N}$ represents granularity level (0 being the highest), and T_i is the function that transforms values in A_i into a different level of granularity. This policy applies T_i to the attribute A_i using granularity g_i before returning A_i. When $g_i = T_i =$ `null`, the original value of A_i is returned. In the location sharing example, Alice may set T_i to return the street address for granularity level 1 and country name for level 5, then assign $g_i = 1$ and $g_i = 5$ to her family and other friends respectively. Note that the granularity levels need to and can be defined based on the attribute semantics.
- **Similarity policy** is defined as (S, D) where $S = \{(c_i, E_i, A_i, d_i)\}$ specifies boolean functions $c_i(E_i, A_i, d_i)$ returning `true` if the *distance* between an external input E_i and value of A_i is less than d_i. The policy returns attributes $D \subseteq A$ if all functions in S are evaluated to `true`. For instance, if Alice wants only friends within 5 km radius to see her location, her policy may be created with $D = \{$`location`$\}$ and c_i`(location,input,5):= dist(location, input) < = 5km` where `input` is the friend's location and `dist` computes the geometric distance between the two locations. Similarly, when sharing data for car-pool, Alice can hide her route information unless it is more than 80% overlapping with her colleague's, by setting $D = \{$`route`$\}$ and c_i`(route,input,0.8):= overlap(route, input) > = 0.8`. Appropriate similarity functions can be defined according to the attribute semantics.

- **Summary policy** is defined as $W = \{(f_i, b_i, w_i, p_i, A_i)\}$ where f_i is a statistics function. b_i, w_i, p_i together defines a sliding window starting after b_i, with size w_i and advancing p_i steps every window. This policy returns only the summary of A_i obtained by applying f_i over the sliding windows. In the mobile health example, Bob may share his blood pressure from 01/01/2011 with the research institute by setting $f_i = \texttt{Average}$, $b_i = 01/01/2011$ $00:00\,\texttt{am}$, $w_i = 24$, $p_i = 24$, $A_i = \texttt{bloodpressure}$. For Alice to share her fuel consumption from the 01/01/2011 for the research project, she may set $f_i = \texttt{Sum}$, $b_i = 01/01/2011$ $00:00\,\texttt{am}$, $w_i = 24$, $p_i = 24$, $A_i \in \{\texttt{fuel}, \texttt{distance}\}$.

One can compose multiple policies to define more complex scenarios. For example, Alice wanting to share the hourly summary of her vital data during the day can specify a filtering policy (for data generated in between 9 am and 5 pm) followed by a summary policy (with window size of 1 h).

2.2.2. *Access subject*

Each policy described above must be associated with an entity to whom the access is granted. At one extreme, a policy is applied to anyone who is a friend of the user. At another, the policy is applicable to one specific user. In between, the user can define *groups* or *circles* of friends and bind each policy to a specific group, so that the same policy is applicable to any member of the group. An incoming request will be *evaluated* against all the policies applicable to the requester.

In the previous examples, both Alice and Bob can define at least one circle for family member, one for close friends, and another for work colleagues. When on holiday, Alice defines new granularity policies for these groups with different values of g_i (smallest value for family group and highest for colleagues). Bob may define a single-user group containing his physician, to which he assigns a filtering policy to only alert the subject of abnormal data. In the participatory sensing example, Alice may specify a group containing the research institute staff and assign it a sliding window policy which only reveals per-day total fuel consumption and travel distance.

2.2.3. *Policy combination*

It is not uncommon for a requester to be subject to multiple access policies. For instance, David is both a work colleague and a close friend of Alice, therefore belonging to two different circles. When multiple policies returning different sets of data to the same requester, the owner must be able to specify how to combine these results together, i.e. a policy combining algorithm. A default *return-all* algorithm could have a serious privacy implication. For instance, David and Alice have a few arguments and the latter decides to move the former to her weak-acquaintance circle, with the intention of restricting his access to her data. Unfortunately, David also belongs to Alice's running-friend and university-friend circles, from which Alice forgets or is not willing to remove David. A return-all policy combining algorithm will not serve Alice's purpose. However, she may define a *most-restricted* algorithm so that only the weak-acquaintance policy is applied to David's requests.

Notice that policy combination is not the same as policy composition. The former deals with how to derive result from outputs of multiple policies. The latter concerns with evaluating policies consisting of sequence of sub-policies: output of one sub-policy becomes input of another.

3. Specification and enforcement of fine-grained access control

We have described the core set of access policies necessary for ensuring user privacy in social applications. To support these, Mosco builds on an approach similar to what we recently proposed in Tuan Anh et al. (2012) which extends XACML—a popular XML-based standard for specifying and enforcing policies.

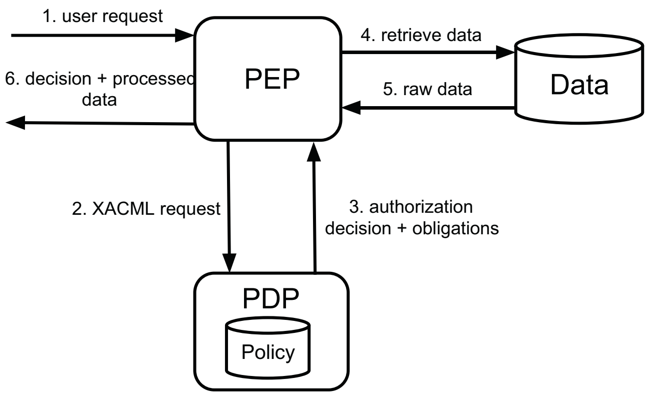


Fig. 1. Extending XACML framework.

3.1. *XACML framework*

At the high level, XACML consists of two components: a *Policy Enforcement Point* (PEP) and a *Policy Decision Point* (PDP). Requests first come to PEP, where they are marshaled into well-formed format before being forwarded to the PDP. The PDP maintains a set of policies against which incoming requests are evaluated. The result is an *authorization decision* and a set of obligations being returned to the PEP. Finally, PEP processes the obligations before sending data to the requester. In Mosco, this step involves accessing and transforming user data. Fig. 1 highlights the process of accessing data in XACML. This framework is flexible because PEP also handles application semantics, that is it does not only return `Permit/Deny` access decision but also a transformed version of the data. For more detailed description of XACML, we refer keen readers to the framework specification (Oasis). Here, we briefly explain the key elements for making and fulfilling requests.

1. Subject, Resource: a *subject* requests access to data of the *resource*. In Mosco, they are users of the social applications.
2. Request: consists of a series of *attributes* providing information about the subject, resource and external inputs. These attributes can be later extracted at PEP and PDP during request evaluation.
3. Policy and Policy set. a policy contains a *target*, a set of *rules* and a set of *obligations* (optionally). Every policy is indexed by its target element which contains a matching condition. The policy is *applicable* to a request if the request attributes satisfy the target matching condition. A rule element contains a boolean function, which returns an *authorization decision*: either `Permit` or `Deny` if the function is evaluated to `true`. When there are multiple rules, a *rule-combining algorithm* must be specified to determine the final decision. A policy set contains multiple child policies. When more than one child policies are applicable, a *policy-combining algorithm* is needed to determine the combined result.
4. Obligation. contains an operation that should be performed by the PEP when it enforces an authorization decision. The most frequent use of obligations includes notifications and logging of data access. In Mosco, obligations are vital for enforcing fine-grained policies, since they specify what functions to be computed over raw data.

3.2. *Fine-grained access control using obligations*

Obligations vs. Rules XACML allows for customized access control policies by letting users define *rules* and *obligations*. A rule is

Rules:	`is-a-friend` (subject is a friend of the resource owner)
Obligations:	ID: `SimilarityObligation` `sim-func-id`: Euclid distance `sim-range`: 5KM `col-idx`: location

Fig. 2. An example policy (Section 2.1.1). Alice sharing her location to nearby friends.

executed within PDP and returns a boolean value. One can implement a rule transforming or checking if a condition holds over the data, but such a rule cannot be used to retrieve data. On the other hand, an obligation is performed at the PEP and could return any object. Thus, for any given function, one can define a corresponding obligation ensuring only the results of that function are returned. In our settings, we utilize both rules and obligations: the former are to filter out requesters who are not friends or not in a specific friend group of the data owner, and the latter are for returning only the permitted data.

An obligation in XACML comprises an ID and a set of attributes. Table 1 summarizes the obligations designed for the policies in Section 2 (for simplicity, we assume that filtering and similarity policies return all data, i.e. $D = A$). These obligations can be combined to specify complex access scenarios. Every obligation contains an integer attribute `col-idx` representing the column to which the obligation is applied (`col-idx` is in fact the index of A_i as explained in Section 2.2.1).

- Filtering obligation: consists of an integer attribute `filtering-cond` specifying a comparison operator `comp`, and a real-valued attribute `filtering-value` containing the filtering value `val`. The obligation returns the data when A_i `comp val` = `true`.
- Granularity obligation: consists of an integer attribute `gran-level` specifying the value g_i, and an integer attribute `trans-func-id` specifying the transformation function T_i.
- Similarity obligation: consists of an integer attribute `sim-func-id` specifying a distance function `dist`, a real-valued attribute `sim-range` specifying the similarity distance between request inputs and values of A_i. A value $v \in A_i$ is returned if `dist(in, v)` ≤ `range` for user inputs `in`.
- Summary obligation: defines a sliding window over A_i. The string attribute `window-start` represents the starting timestamp. Integer attributes `stat-func-id`, `window-size`, `window-advance` define the statistic function to be applied over each window, the window size and advancing step respectively.

Notice that when obligation returns no data, the user receives an empty set of result instead of a Deny decision. We remark that returning Deny or an empty result both leak some information about the data, but addressing such leakage is not within the scope of our work.

Example. Fig. 2 illustrates how an access scenario described in Section 2.2.1 is mapped into XACML's rules and obligations. In particular, Alice wants to share her location only to friends when they are within 5KM from her. The corresponding policy contains a rule `is-a-friend` which returns `true` if the requester is a friend of Alice. The obligation is of type *similarity*. Suppose Bob, a friend of Alice, requests her location through the XACML framework. Bob's request contains his current location. PDP first evaluates the rule, whose result is `true`. Next, PDP forwards the obligation to the PEP, which reads Alice's current location and computes distance to Bob's. If the distance is less than 5KM, PEP sends Alice's location to Bob. Otherwise, Bob receives empty data.

3.3. XACML policy combination and composition

Letting a user to define a policy combining algorithm is another dimension of fine-grained access control. A more privacy-conscious user may want the most restricted policy to be selected, whereas an indifferent user may wish his friend to see as much data as possible. In XACML, a combining algorithm is identified by an ID and is included in the Policy element. The common options are:

- Deny-override or Permit-override: returns the `Deny`-policy immediately if there is one `Deny` policy, or the first `Permit`-policy. This algorithms are standards in XACML.
- Most-restricted: returns the policy with the most restriction over the data. The semantics of this algorithm depends on the application. In location sharing applications, granularity with the highest granularity value may be considered as most restricted, whereas in participatory sensing applications, summary policies with the largest window size may be the most restricted.
- Union: returns data from all policies. This is the most relaxed algorithm, especially since results from different types of policies may reveal extra information.

XACML does not support policy composition, as it is not possible to specify an order in which obligations are executed by the PEP. However, one can work around this by defining complex obligations which capture the composed policy. For example, a `FilteringAndSlidingWindowObligation` can be added to Table 1 such that when returned, the PEP will first evaluate the filtering condition and then apply sliding window over the output.

4. Middleware design

The goal of Mosco is to provide a middleware that requires minimal effort from the developers to create new mobile social applications with support for privacy-preserving capabilities, particularly by facilitating fine-grained access control. The resulting applications are scalable with respect to the number of users and sizes of data, while at the same time provide users with fine-grained control over their data. To achieve the former, we build Mosco on top a cloud platform, namely Google App Engine, which handles increases in system workload automatically and gracefully. We accomplish the latter by extending XACML, as discussed in the previous section.

4.1. System overview

Fig. 3 illustrates main components of an application built using Mosco. Information about a user—including personal details, friends and access policies—are stored within a *data store*. The data generated by each user is also managed by the cloud at its data store. Suppose a user *A* queries for data from his friends, the request first

Table 1
Obligations supporting fine-grained access policies.

Description	Id	Attributes
Filtering	`FilteringObligation`	`filtering-cond, filtering-val, col-idx`
Granularity	`GranularityObligation`	`gran-level, trans-func-id, col-idx`
Similarity	`SimilarityObligation`	`sim-func-id, sim-range, col-idx`
Sliding window	`SlidingWindowObligation`	`window-start, window-size, window-advance, stat-func-id, col-idx`

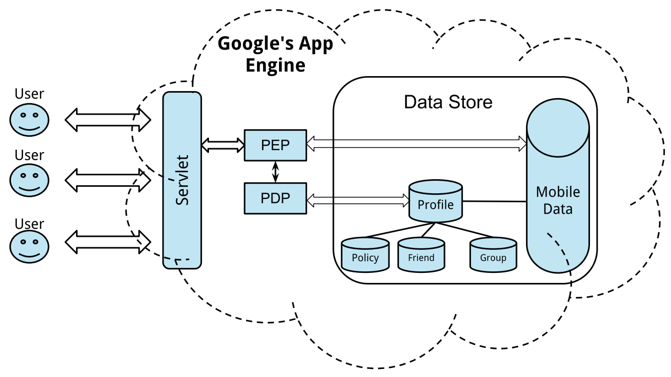


Fig. 3. System's overview.

arrives at the application servlet which forwards it to the PEP module. For each friend of *A*, say *B*, an XACML request is sent to the PDP, which retrieves B's policies from the datastore and evaluates them against the request. If evaluated to `Permit`, a set of obligations is sent back to the PEP. Finally, PEP retrieves raw data from the data store, performs the functions as specified in the obligations and returns the result to *A* (via the servlet).

4.1.1. *Trust model*

Security of the access control mechanism depends upon the access policies being evaluated correctly. In our case, the evaluation is done at the cloud which we assume to be honest. In particular, the cloud is honest in three respects: first, it carries out the access control enforcement correctly; second, it is allowed to access user data in clear; third, the cloud is secure from external attacks. We acknowledge that this is a strong assumption. Nevertheless, we argue that it is not unreasonable to expect the cloud to behave honestly either due to the need to protect its reputation or to fulfill its Service Level Agreements and legal obligations, and many existing systems do operate under similar assumptions. While it is possible to achieve some levels of fine-grained access control with semi-honest clouds (Tuan Anh and Datta, 2012), such approaches require expensive cryptographic operations. On the other hand, by assuming trusted cloud, we can design more scalable and efficient systems supporting very high level of fine-grainedness in access control with respect to other users.

4.1.2. *Why google App engine*

Being a platform-as-a-service cloud platform, Google App Engine (GAE) offers a scalable infrastructure for deploying social applications. It has been used for many large-scale social applications: BuddyPoke, Crystal, etc. Compared to the alternatives such as Windows Azure or Amazon's EC2, there are several advantages in using GAE when it comes to social applications. First, even though GAE runtime environment is restricted (no writing to files, no socket API, etc.), it handles scaling of the resource seamlessly making the application respond better to sudden increases in demand (in the presence of *flash crowd*, for instance). Second, with the large user base, one can enjoy the Google authentication service for free. This means the application users can be assumed to have already been authenticated, eliminating also any long drawn registration process. Third, the sand-boxing environment could indeed help secure the applications from common security or denial of service threats (such as side channel attacks or common software vulnerability) which are dealt with by Google underlying infrastructure. Finally, GAE comes with rich ecosystem for creating new applications: comprehensive support for multiple SDKs, easy integration with other Google's products, and ease of rolling out the finalized product since applications can be deployed to a real cloud directly from the development mode without any change.

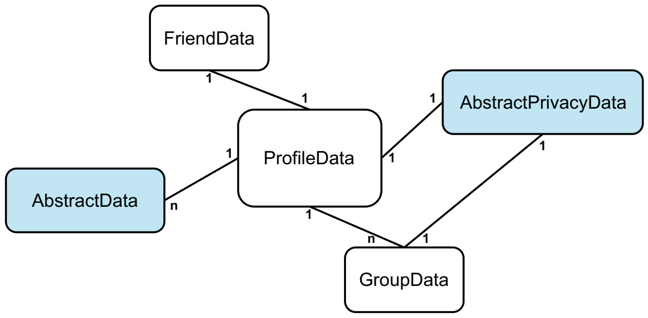


Fig. 5. UML diagram representing main entities in the data model. The shaded entities are to be extended when implementing a new application.

4.2. *Middleware design*

As seen in Fig. 3, a new social application consists of a client and a server component. The latter will be running on Google's cloud infrastructure and serving requests from mobile clients via HTTP. Mosco provides a set of API *hooks* so that new applications can be developed with minimal effort. Fig. 4 highlights the main modules that are to be extended when writing a new application. Suppose the application deals with data types and policies that are not already supported by Mosco. First, the data and policy definition are extended to accommodate the new types. Next, the *XACML policy builder* is extended so that XACML policies can be built from the `AbstractPrivacyData` instances. Finally, *XACML obligation handle* is extended to process obligations embedded in the new policies.

4.2.1. *Data and policy model*

Fig. 5 shows five entities that make up the generic data model supported by Mosco. They correspond to five virtual *tables*

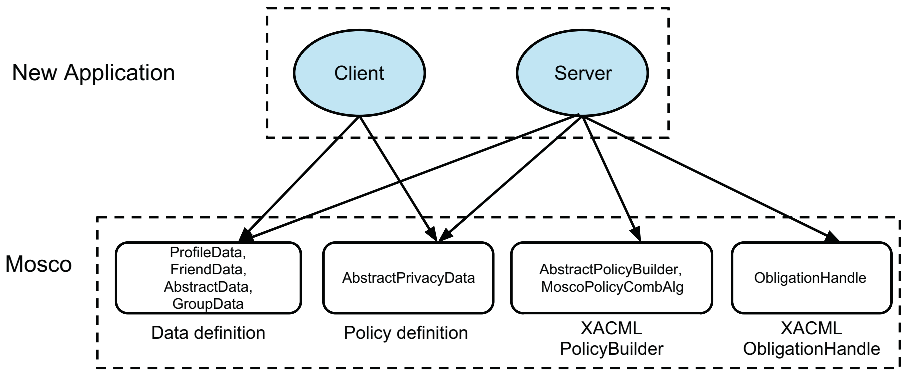


Fig. 4. Mosco design. Both client and server implementations extend four modules in Mosco: data definition, policy definition, policy builder and obligation handle.

maintained in Google datastore. Each data instance is indexed by a GAE-generated *key*.

`ProfileData` represents a user. It contains the user's email address and a `AbstractPrivacyData` key representing the *default* access policy applicable to all friends (as opposed to the group policies embedded in `GroupData` which are applicable to only group members). Using this key and the email address, one can retrieve all information pertaining to the specific user including the data.

`AbstractPrivacyData` represents a *generic* access policy. A new policy must extend this entity. It contains a `ProfileData` key pointing to the owner, and a `GroupData` key if this policy is associated with a group. Mosco comes with support for the four policies discussed in Section 2 (for real-valued data). We overcome GAE's lack of support for inheritance by storing a variable indicating the *type* of policy being stored, so that retrieval of a `AbstractPrivacyData` instance can be done by specifying the class name of the policy type.

`AbstractData` represents a *generic* data instance. In our settings, all data types extend this entity. It contains email address of the data owner, the data content and a timestamp variable. One can retrieve the owner's data by constructing a SQL-like query over the `AbstractData` table for the matching email address.

`FriendData` represents a friend relationship. It contains email address of the owner and of the friend, as well as a timestamp indicating the latest timestamp of `AbstractData` instance accessed by the friend. For example, a `FriendData` instance represented by the tuple (u_1, u_2, t) means that u_1 is friend of u_2, and u_2 most recent access to u_1's data is at timestamp t. The timestamp variable is necessary to avoid Mosco returning duplicate data. For example, (u_1, u_2, t) means that u_1 has accessed data of u_2 up to timestamp t. Next request from u_1 to u_2 will only return authorized data timestamped at t' where $t' > t$. Similar to `AbstractData`, one can retrieve all the friends of a certain user by constructing a SQL-like query.

`GroupData` represents a friend group (or circle) of a specific user. It consists of the email address of the group owner, the group name, and a list containing the members' email addresses. Additionally, it has a `AbstractPrivacyData` key pointing to the access policy associated with the group.

4.2.2. Data dependencies

In our design of the data model, the `ProfileData` instance contains no direct reference to other entities except to an `AbstractPrivacyData` instance. To retrieve groups, friends, and data associated with a user, one needs to execute a SQL-like query with the matching user email address.

In our first design (we refer to this as the *old data model*), each `ProfileData` instance maintains a key pointing to a `FriendData` instance which has a list of `ProfileData` keys pointing to other users. This approach seems to enable easy access to a user's entire friend list by retrieving one specific `FriendData` instance using its key. However, we later changed to the current design of `FriendData` that is similar to that of `AbstractData` because of the following reason. Adding or removing a friend relationship in the old model requires synchronized access to two `ProfileData` and two `FriendData` instances. For the latter, there needs to be two read and two write access. Since friend update is likely to be a high-frequency operation, and as the system scales, the cost of multiple read/write access and of locking will become expensive. In contrast, in the current Mosco model, updating friend require one write to the datastore, and updates for the same user can be done in parallel. We show in Section 5 that this indeed results in better update performance.

4.2.3. Enforcing new policies

Having defined the data and policy model, Mosco can now store the new data types and policy information in the cloud. When requests come in, they must be evaluated against the stored policies. To support evaluation against new policies, Mosco must be extended to construct well-formed XACML policies (for PDP evaluation) from the stored `AbstractPrivacyData` instance.

`AbstractPolicyBuilder` provides a template for constructing XACML policies from `AbstractPrivacyData` data instances. To build a concrete policy, one must implement the `createRules()` and `createObligations()` method, which defines Rule and Obligation elements of the resulting XACML policy. For each type of `AbstractPrivacyData`, Mosco comes with implementation of one policy builder.

`ObligationHandle` is an interface that must be extended for every type of obligation. An instance of `ObligationHandle` is executed at the PEP. In the `processObligation()` method, one can query raw data in `AbstractData` table and process it according to the function defined by the obligation ID and attributes. Mosco has four implementations of `ObligationHandle`, each corresponds to an obligation type defined in Section 3.

`MoscoPolicyCombAlg` provides a generic policy combining algorithm. Current version of Mosco supports Union algorithm. Other alternatives as listed in Section 3 can be added by extending `MoscoPolicyCombAlg`.

4.2.4. Datastore access and caching

For each data entity in Mosco there is a corresponding *backend service* handling storage, update and accessing of the data. In particular, `ProfileService, FriendService, GroupService, DataService` and `PolicyService` are singletons containing methods dealing with `ProfileData, FriendData, GroupData, AbstractData` and `AbstractPrivacyData` respectively.

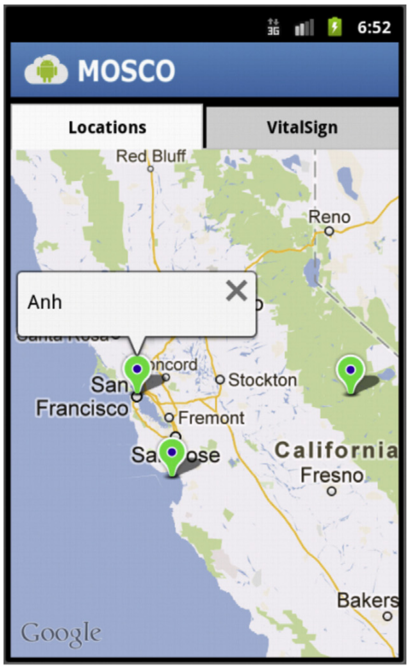
Accessing the datastore is an expensive and billable operation (Google). Mosco provides a cache layer for all of these services. Caching is useful in social applications since the same policy may apply to many users (for example, a default policy applies to all users in the friend list, or a group policy to all members in the group), hence data need be retrieved only once and used many times. It is particularly the case for non-`AbstractData` instances, since they seldom change. The cache is purged whenever the data is updated. For instance, when a new data is added, the current cache for `AbstractData` is cleared. We demonstrate caching effectiveness in the next section.

4.2.5. Push or pull

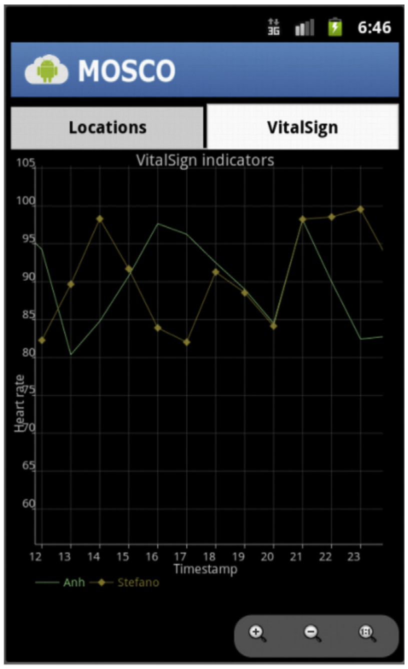
In many social applications, data can be pushed to the clients (Facebook, Twitter). Server pushing is an useful abstraction besides client pulling, which gives an impression of real-time updates. Underneath, however, pushing is implemented by client pulling periodically and by the server hanging on the HTTP request. The current implementation of Mosco supports data pulling only. It leaves it to the application to determine how often the client should query the server. This makes sense for non-realtime applications (such as location sharing) since it imposes no overhead on the server. Participatory sensing and mobile health applications which may demand instant access to the latest data (for realtime decision making) could benefit from the push interface.

5. Implementation and evaluation

We have implemented Mosco in Java (source code can be found at http://code.google.com/p/mosco) which supports all four types of access policies discussed in Section 3. In the following, we describe the implementation of two mobile social applications using Mosco, and the experimental evaluation of our middleware.



(a) Location



(b) Mobile health

Fig. 6. Android client user interface.

5.1. Implementation

The goal of Mosco is to serve as a middleware for easy development of privacy-aware, scalable mobile social applications. To demonstrate this, we implement a location sharing and a mobile health sharing application for Android. The former allows user to control who gets access to his location, and to display nearby friends (who give him access to their locations) in a map (Fig. 6(a)). The latter enables user to specify fine-grained control over his physiological (or vital sign): namely the heart rate and chest volume (respiration force) which are collected during his sleep. Such information is useful for sleep study and can be shared (compared) between friends (Fig. 6(b)). It is worth emphasizing that the applications we discuss are not a contribution per se, and more sophisticated applications may be essential for making compelling real-life use cases. The purpose of developing these 'toy' applications was to demonstrate and test the efficacy and scalability of the Mosco middleware, and to showcase how it eases the development and deployment process for new privacy-aware social mobile applications.

To implement a new application with Mosco, one first defines the data type and registers it to the `IDs` class. Second, a new policy model is specified by extending `AbstractPrivacyPreference` class. Third, `AbstractPolicyBuilder` class is extended to support the new policy. Finally, obligation processing for the policy is defined by extending `ObligationHandle` interface. The `ObligationHandle` object extracts variables embedded in the XACML obligation and passes them as arguments to a user-defined function. This function implements application-specific processing of data, and it is declared at the `IDs` class. For the location sharing application, we define `LocationSimilar` and `MostFrequentLoc` as application-specific functions for similarity and granularity policies. The mobile health application requires `VitalSignSimilar` class for processing similarity obligation (other obligations are standard functions over real-value numbers).

Mosco comes with policy definition, builder and obligation handle of the four policies listed in Table 1. It also provides standard functions for processing obligations over real-value data. The implementation of the location and mobile health sharing applications take 413 and 292 lines of code (not including blank lines) respectively. This illustrates the ease of developing a new social application using our middleware. Including these two applications, Mosco amounts to 7019 lines of code.

Table 2
Parameters used for the benchmarking experiments.

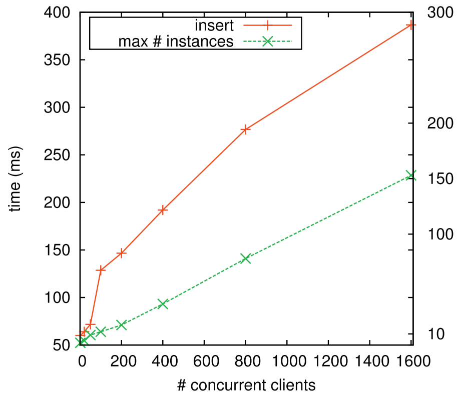
Parameters	Values
Environment	Single server, Google App Engine (GAE)
Application	Location sharing, mobile health sharing
Policy	Similarity, filtering
Dataset	SNAP, SantaFe
# Concurrent clients (nClients)	1–1600
Client request	Insert, delete, data query

5.2. Evaluation

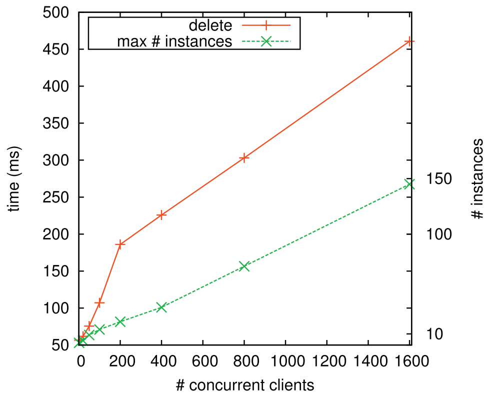
We carry out experiments to evaluate how well applications developed using Mosco perform, especially when running on Google App Engine (GAE) and under increased workloads. We consider the following performance metrics: **scalability** and **processing time** for updating application data and requesting user data (or data query). We also want to compare the performance between different applications.

5.2.1. Methodology

Table 2 summarizes the parameters used for the experiments. For the location sharing application, we use the SNAP dataset which contains real location information of over 5000 users. We experiment with similarity policies which grant access to location data only when the subject is within a certain radius. For the mobile health sharing application, we generate synthetic data based on the SantaFe dataset which contains real physiological data from a sleep study. For this application, we experiment with filtering



(a) Insert time



(b) Delete time

Fig. 7. Data update time.

policies which grant access to heart rate data whenever it exceeds a certain threshold.

We simulate a set of *clients*, each of which sends a stream of requests to the application. Specifically, a client sends a request (for a data insert, delete or query operation), waits for the response from the application and repeats again. Application *workloads* are simulated by varying the number of concurrent clients (between 1 and 1600). We note that a client represents an *extreme* user sending requests at maximal rate to the cloud. Hence, 1600 concurrent clients translate to 1600 concurrent requests at any given time. In practice, each mobile user issues requests at much lower rates, therefore the workload of 1600 concurrent requests may correspond to a much higher number of concurrent users.

We perform experiments in two settings: single-server and GAE environment. In the former, the application is deployed on one server running the development-version of the GAE server. In the latter, the application is running on the real Google App Engine cloud. The reason for experimenting with these two settings is to evaluate the limitation of running the application on a generic, non-cloud environment versus the benefit of automatic scaling of the GAE cloud platform. For the single-server setting, we hire one large (high-CPU and high-memory) Amazon EC2 instance to run the development-version GAE server, and 8 other medium EC2 instances that run the simulated clients. For the GAE environment, we purchase the F4 instance class offered by Google.

The results presented in the following are averaged over 3 runs. Unless stated otherwise, the graphs show results of the mobile health sharing application.

5.3. Results

Single-server vs. GAE environment. We started multiple clients that perform concurrent data insert. Each client inserts 1000 data tuples for each user. The single-server setting reaches its capacity at 50 concurrent clients, i.e. the application crashed after 50 clients. In the GAE environment, the application continues to run and scales gracefully. This result illustrates the differences of existing cloud platforms. More specifically, Amazon EC2 (as opposed to GAE) provides raw infrastructure, but its lack of automatic scaling hinders applications' availability when there is a sudden rise in demand. The following results are obtained by running the experiments in GAE environment.

Insert and Delete. Fig. 7 shows the insert and delete performance of application data, with each client sending a stream of 1000 data updates for each user. Overall, it takes 23 min to upload 1.6 millions data items with 1600 concurrent clients, and 25 min to delete them. The insert time per data item scales gracefully from

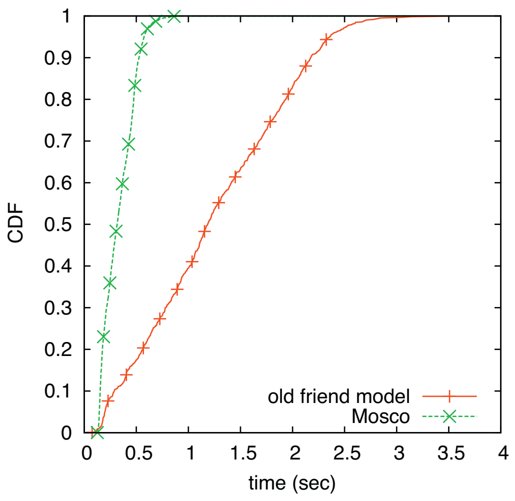


Fig. 8. Comparing the old vs. Mosco data model for `FriendData`, for 50 concurrent clients adding friend relationship from the SNAP dataset.

59 ms with 1 client to 386 ms with 1600 clients. Delete time scales from 56 ms to 460 ms. The maximum number of GAE instances launched during the experiments are shown in the graphs (they correspond to the right y-axis). As more concurrent clients are added, GAE spins new instances to deal with request: over 150 instances are active for 1600 clients.[1]

We have discussed in Section 4 the alternative approach for designing the data model, especially with respect to the `FriendData` entity. Fig. 8 compares the distribution of insert time for `FriendData` between the old data model and the Mosco model. The old model incurs significant overhead per insert, as it requires almost 1.5 s at median as opposed to 0.4 s for the Mosco model. This demonstrates the benefit of loosely-coupled data model, especially when using a cloud platform such as Google which employs key-value based storage.

Query. The experiments for data query time are run in the GAE settings. The following results are for the mobile health sharing application, with 1000 users and 20,000 data items. Each client sends data requests of the form (u_1, u_2) representing request originated from user u_1 for the data of user u_2.

[1] Compared with an old set of experiments (carried out in September 2012), we observe that GAE launches more instances to deal with a given workload, resulting in shorter upload time.

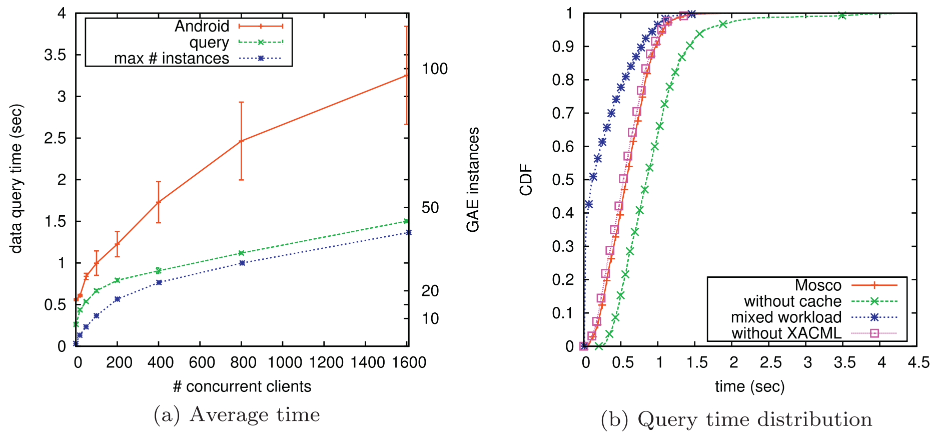


Fig. 9. Mean query time for varying workload, and query time distribution for 50 concurrent clients.

Fig. 9(a) shows how query time at GAE servers scales with increased workloads. In particular, it takes 0.26 s for 1 client and only rises to 1.5 s for 1600 concurrent clients. It can be observed that the maximum number of GAE instances goes up to 35 instances to accommodate higher workloads. The figure contains measurement from a real Android client application (running on a HTC NexusOne mobile phone with 2.3.6 Android OS, over a residential wireless network). The response time observed at the client increases and varies more with higher workloads. However, even with 1600 concurrent (simulated) clients, the average response time for the phone user is still under 3.5 s. We believe this latency (which can also be attributed to the network latency) is reasonable.

Fig. 9(b) shows the query time distribution for the workload consisting of 50 concurrent clients. There are three important observations that can be taken from this graph. First, the caching mechanism described in Section 4 is effective as it improves the query time up to 33% at median and 50% at the 90th percentile. Second, the overhead of XACML (including policy management and evaluation against requests) is negligible. It is illustrated as the time taken for direct queries (obligation functions is executed directly on receipt of the direct queries without going through the XACML process) being close to the query time observed in Mosco. The final observation comes from running a *mixed* workload. The result so far is presented for *normal* workloads in which the requests (u_1, u_2) are constructed such that u_1 and u_2 are friends. In the mixed workload, we let each client generate normal workload with the probability of 0.5 and a random workload (where u_1 and u_2 are random users who may not be friends) with the same probability. The query time for this workload indicates faster response time from GAE servers, which is as expected because many of the requests are rejected without accessing the datastore for the data.

Comparing two applications. Finally, Fig. 10 compares the update and query time between the location sharing and mobile health sharing applications. It can be seen that the metrics are almost the same for both applications. This is because the location and vital sign data are roughly of the same size, and that the time taken for processing the similarity and filtering obligation are also similar.

6. Related work

Our work concerns a middleware that facilitates sharing of mobile data in the context of social computing. There exists other systems such as SenseWeb, PatientsLikeMe, Foursquare whose main focuses are also on data sharing. However, their access control models are either coarse-grained (all-or-nothing sharing) or have little support for social settings. When it comes to

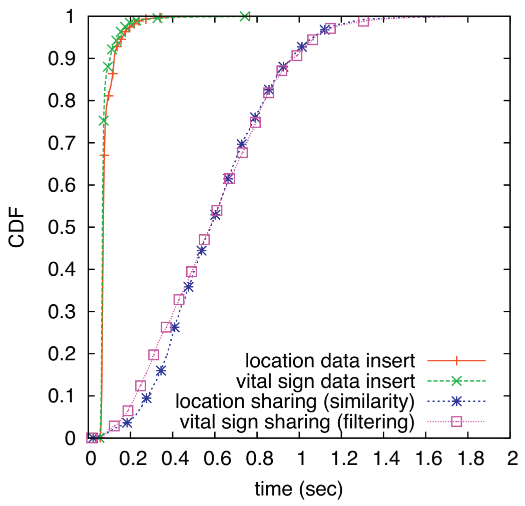


Fig. 10. Comparing location sharing and mobile health sharing, for 50 concurrent clients.

Internet-of-things applications, there are more in data sharing than merely giving access to the data. Other systems deals with other aspect of sharing, such as content matching (Guha et al., 2012), location proximity matching (Zhong et al., 2007; Narayanan et al., 2011), crowd-sourced sensing (Cornelius et al., 2008), or behavior classification (Lane et al., 2011). Frameworks such as CarTel (Hull et al., 2006), Virtual Trip (Hoh et al., 2008) address query and computation issues for specific applications (traffic control, in particular), while assuming data has already been shared.

Mosco guarantees user privacy in terms of fine-grained access control with respect to the end-users. We assume that the cloud where Mosco is running is trusted, that is it will neither violate data privacy nor collude with rogue users to do so. As competition amongst cloud providers are high, the need to maintain high reputation is a strong incentives for them to be trustworthy. Systems such as Airavat (Roy et al., 2010) or eXACML (Tuan Anh et al., 2012) builds on this assumption to provide differential privacy or fine-grained access control for archival data. When a general Service License Agreement (SLA) does not suffice, one must rely on cryptography to protect data from the cloud. CryptDB (Popa et al., 2011), Plutus (Kallahalla et al., 2003), CloudProof (Popa et al., 2011) ensure data confidentiality using encryption, which is the same as access control at a coarse-grained level. Recently, advanced encryption schemes such as Attribute-Based Encryption (Goyal et al., 2006; Bethencourt et al., 2007) enable more fine-grained access control. But these schemes incur high computational overhead.

Furthermore, policies that require transforming the data (the granularity policies, for instance) cannot be directly mapped to ABE. As noted in Tuan Anh and Datta (2012), the design space for outsourcing access control to the cloud can be characterized along three dimensions: trustworthiness of the cloud, fine-grainedness of policies and the work ratio between users and the cloud. CryptDB, Plutus and other systems employing ABE trade fine-grainedness and work ratio for a more relaxed trust assumption. In this design space, Mosco occupies a unique spot with the highest level of fine-grainedness and work ratio.

Mosco access control model can be considered as hierarchical, in the sense that it consists of two level: user and group. A popular, multi-level role-based access control (RBAC) model has been popular in enterprise systems, where delegation and dynamic group membership are important. For the time being, we believe the simple two-level model used in Mosco is sufficient for many social applications.

Finally, Mosco leverages Google App Engine, a platform-as-a-service cloud platform. There exists other services at the same level of abstraction (Amazon, Windows), or even lower-level abstraction (infrastructure-as-a-service, Amazon, Rackspace Hosting). One could implement Mosco using any of these services and enjoying different trade-off (Li et al., 2010). While we maintain that Google App Engine is a good choice for developing and deploying social applications, we envisage that porting Mosco to another environment would not be particularly challenging.

7. Conclusion and future work

In this paper, we have presented Mosco, a privacy-aware middleware for scalable mobile social computing. Mobile social applications requires fine-grained access control while also being able to scale gracefully with more users and data. We have designed Mosco to ease the development of new social applications while meeting both of these requirements. We have identified a core list of access policies that are common in many social applications. In Mosco, these policies are enforced by using an extension of the XACML framework. Mosco runs on Google App Engine to leverage the cloud's plentiful and scalable resources. We have demonstrated that Mosco shortens the development process for new applications. In addition, the resulting application scales gracefully to accommodate increased workloads. Our experiments also indicate that the overhead incurred by the access control mechanism is small.

Our immediate plan is to enhance the existing location sharing and mobile health sharing application with more features (mostly at the client side) in order to attract real users. Once having real users, we will be able to carry out user study and gain more insights into the performance of the application and of the middleware. The current version of Mosco supports only the *pull* abstraction for data retrieval. As discussed in Section 4, a time-sensitive application can benefit from a push abstraction. We plan to incorporate this into the future version of Mosco, which entails instrumenting the server to wait on long-lived HTTP requests and send new data to the client when it arrives. This extension is likely to incur overhead at the server side.

We plan to investigate how to enhance the current access control model to the full role-based access control (RBAC) model. Adding more hierarchy levels and delegation capability to the access subject will improve the flexibility of Mosco and make it more attractive to enterprise applications. We also intend to extend the current XACML framework with support for policy composition, which will increase its expressiveness as well as its chance to be adopted in the stream database community. In fact, the problem of composing simple policies into complex ones can be viewed in the same light as a well-known problem in stream database research: constructing query graph from query operators. As a consequence, we could borrow techniques from the vast number of works in this community when implementing our XACML extension.

Another interesting extension for Mosco is to raise the level of data access abstraction. Current applications of Mosco support simple abstractions involving none or very simple computation on the data, but higher-level abstractions requiring more complex computation may be desirable. One example is a policy that grants access only to results of certain data mining algorithms or statistical function. Incorporating these abstraction to Mosco seems straightforward, as one can define a new obligation for the required computation. The challenges lie on identifying the different levels of abstractions and implementing them on a cloud platform in an efficient way.

Last but not least, we will like to investigate the challenges when removing the trust assumption regarding the cloud. Access control enforcement will no longer be possible with XACML, instead a cryptographic approach must be considered. Recent work (Tuan Anh and Datta) shows that it is possible to support a number of fine-grained access policies. But to support all the policies listed in Section 3 in an untrusted environment remains a challenge.

References

Amazon Elastic Computing Cloud. http://aws.amazon.com/ec2.

Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M., 2009. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28. EECS Department, UCB.

Bethencourt, J., Sahai, A., Waters, B., 2007. Ciphertext-policy attribute-based encryption. In: IEEE Symposium on Security and Privacy.

BuddyPoke! Express Yourself. http://www.buddypoke.com.

Chillingo Crystal. http://www.chillingo.com/crystal.

Cornelius, C., Kapadia, A., Kotz, D., Peebles, D., Shin, M., Trandopoulos, N., 2008. Anonysense: privacy-aware people-centric sensing. In: MobiSys, pp. 211–224.

Curetogether. http://curetogether.com.

Facebook. http://www.facebook.com.

Find My Friends. http://itunes.apple.com/us/app/find-my-friends/id466122094?mt=8.

Foursquare. https://foursquare.com.

Google App Engine. https://developers.google.com/appengine.

Google+. https://plus.google.com.

Goyal, V., Pandey, O., Sahai, A., Waters, B., 2006. Attribute-based encryption for fine-grained access control of encrypted data. In: CCS.

Guha, S., Jain, M., Padmanabhan, V.N., 2012. Koi: a location-privacy platform for smartphone apps. In: NSDI.

Hoh, B., Gruteser, M., Herring, R., Ban, J., Work, D., Herrera, J.-C., Bayen, A.M., Annavaram, M., Jacobson, Q., 2008. Virtual trip lines for distributed privacy-preserving traffic monitoring. In: MobiSys, pp. 15–28.

Hull, B., Bychkovsky, V., Zhang, Y., Chen, K., Gorackzko, M., Miu, A., Shih, E., Balakrishnan, H., Madden, S., 2006. CarTel: a distributed mobile sensor computing system. In: 4th International Conference on Embedded Networked Sensor Systems.

Kallahalla, M., Riedel, E., Swaminathan, R., Wang, Q., Fu, K., 2003. Plutus: scalable secure file sharing on untrusted storage. In: FAST 2003.

Lane, N.D., Xu, Y., Lu, H., Hu, S., Choudhury, T., Campbell, A.T., Zhao, F., 2011. Enabling large-scale human activity inference on smartphones using community similarity networks. In: UbiComp, pp. 355–364.

Last fm. http://lastfm.com

Li, A., Yang, X., Kandula, S., Zhang, M., 2010. Cloudcmp: shopping for a cloud made easy. In: HotCloud.

Moodscope, With a Little Help from Your Friends. http://www.moodscope.com.

Mun, M., Reddy, S., Shilton, K., Yau, N., Burke, J., Estrin, D., Hansen, M., Howard, E., West, R., Boda, P., 2009. PEIR, the personal environmental impact report, as a platform for participatory sensing system research. In: MobiSys, pp. 55–68.

Narayanan, A., Thiagarajan, N., Lakhani, M., Hamburg, M., Boneh, D., 2011. Location privacy via private proximity testing. In: NDSS.

Nike+. http://nikeplus.nike.com.

Oasis Extensible Access Control Markup Language (xacml). https://www.oasis-open.org/committees/xacml/.

Patientslikeme. http://www.patientslikeme.com/.

Popa, R.A., Zeldovich, N., Balakrishnan, H., 2011. Cryptdb: A Practical Encrypted Relational dbms. Technical Report MIT-CSAIL-TR-2011-005. CSAIL, MIT.

Popa, R.A., Lorch, J.R., Molnar, D., Wang, H.J., Zhuang, L., 2011. Enabling security in cloud storage SLAs with CloudProof. In: USENIX Annual Technical Conference 2011.

Quantified Self: Self Knowledge Through Numbers. http://quantifiedself.com, 2012.

Rackspace Hosting and Cloud. http://www.rackspace.com.

Roy, I., Setty, S.T.V., Kilzer, A., Shmatikov, V., Witchel, E., 2010. Airavat: security and privacy for mapreduce. In: NSDI 2010.
SenseWeb. http://research.microsoft.com/en-us/projects/senseweb/.
Stanford Network Analysis Project. http://snap.stanford.edu/data/loc-brightkite.html.
Take Control of Your Sleep. http://www.myzeo.com/sleep.
The Santa Fe Time Series Competition Data. http://www-psych.stanford.edu/andreas/Time-Series/SantaFe.html.
Tuan Anh, D.T., Datta, A., 2012. The blind enforcer: on fine-grained access control enforcement on untrusted clouds. Data Engineering Bulletin.
Tuan Anh, D.T., Datta, A. Stream on the Sky: Outsourcing Access Control Enforcement for Stream Data to the Cloud. http://arxiv.org/abs/1210.0660.
Tuan Anh, D.T., Wenqiang, W., Datta, A., 2012. City on the sky: extending xacml for flexible, secure data sharing on the cloud. Journal of Grid Computing, 151–172.
Twitter. http://twitter.com
Windows Azure: Microsoft's Cloud Platform. http://www.windowsazure.com.
Zephyr Measure Life Anywhere. http://www.zephyr-technology.com.
Zhong, G., Goldberg, I., Hengartner, U., 2007. Louis, Lester and Pierre: three protocols for location privacy. In: PET, pp. 62–76.