Jano - Specification and Enforcement of Location Privacy in Mobile and Pervasive Environments

José Simão
ISEL
Rua Conselheiro Emídio Navarro $N^\circ 1$
1959-007 Lisboa
INESC-ID/Technical University of Lisbon
Distributed Systems Group
Rua Alves Redol $N^\circ 9$, 1000-029 Lisboa
jsimao@cc.isel.ipl.pt

Carlos Ribeiro, Paulo Ferreira,
Luís Veiga
INESC-ID/Technical University of Lisbon
Distributed Systems Group
Rua Alves Redol $N^\circ 9$, 1000-029 Lisboa
[carlos.ribeiro, paulo.ferreira,
luis.veiga]@inesc-id.pt

ABSTRACT

Today there are many location technologies providing people or object location. However, location privacy must be ensured before providing widely disseminated location services. Privacy rules may depend not only on the identity of the requester, but also on past events such as the places visited by the person being located, or previous location queries.

So, location systems must support the specification and enforcement of security policies allowing users to specify when, how and who can know their location. We propose a middleware platform named Jano[1] supporting both pull and push location requests while enforcing configurable security policies. Policies are specified using the *Security Policy Language - SPL*, facilitating the use of well known security models. In particular, Jano supports history-based policies applied to persons or objects location.

The system was implemented with the integration of several location technologies (e.g. GPS, Bluetooth, etc.), and dealing with the heterogeneity aspects. It provides an interface that facilitates policy specification and has good performance.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

Keywords

location-awareness, privacy, declarative policies

[1]Jano is the god of doors and gates in the roman mythology. He is usually depicted with two or four faces turning in opposite directions.

1. INTRODUCTION

Being able to locate someone or something has been a need over the times. Today, as in the past, the reasons why location is needed are multiple. We may wish to know where we are for self orientation. We may want to know where other persons or objects are placed so that we can meet or find them. Finally, and more recently, our location could also be used by third-party applications to send us contextual information (e.g. receiving advertisements related to the shop we are arriving at [18], or obtain detailed information about the work of art we stand by at a museum). In modern societies, privacy is a necessary condition for freedom, in the sense that where we are and who we are with, is related to what we are doing. The possibility of being located by others raises the question: "Who, and in what condition, may someone by allowed to locate me or know I am nearby?". This can be as simple as restricting a time interval: "*Bob* and *Alice* can only locate me between 10 a.m. and 4 p.m.". Sometimes the decision is not only based on the present situation but also on past events. For example, *Alice* may accept to disclosure her location in isolated instants but not being tracked, i.e. reveal several locations in sequence.

To address these scenarios the location system must be capable of responding to location requests but, at the same time, evaluate each request and decide if it is authorized or not, based on some previously specified policy. Our main goals are the specification and enforcement of complex security policies, including those based on history events, without compromising usability and performance. We want to define and enforce these policies to a location service that supports both synchronous, i.e. *pull*, and asynchronous, i.e. *push*, requests.

Given that we want to build a generic infrastructure, the policy monitor that will enforce policies can not be made as a group of static rules. Organizations will want to specify different policies and have different characterizations of their elements, not depending on the location system, but on a previously defined structure. Given the dynamics of the information, past events are particularly important to consider. When a user makes a query for someone's location, or arrives at, or leaves from a space, these events must be recorded by the system with the goal of applying policies to them; for example, "the administrator can know my location if I am in a dangerous place for more than one hour".

The way these events are represented and stored is crucial during the evaluation of policies.

Other location services that enforce some kind of privacy [15][14] do not present an integrated solution to deal with history based policies. In some of them, responses to push requests are also not handled as a first class issue, making it hard to use the location events produced by the service into the notification decision process.

This paper presents Jano, a generic multi-technology Location Service, supporting the specification and capable of enforcement of privacy policies on the location of persons or objects. Location information is gathered from an unlimited variety of sources. Two types of queries are available: *pull* and *push*. While the former answers with the last known location, the latter corresponds to an asynchronous notification request (e.g. "Notify me by e-mail when Alice arrives to room 19 after she has leaved the cafeteria"). *Control access policies* enforce the requirements of users and owners of places about disclosure of location information. These policies can be associated to users, objects or places. *Notification policies* are used to decide about the need for a notification. They are associated to a user when a *push* request is made. The movement of persons and objects makes Jano generate *location events* which are evaluated by these policies to determine if a notification is needed and allowed.

The definition of both types of policies is made through the use of the Security Policy Language (SPL) [16]. SPL is a policy language particularly suitable for localization services, because it allows the definition of models comprised by elements specifically adapted to the localization semantics; namely, it allows for the definition of history-based policies which are an important element for the definition of localization policies. SPL is also system agnostic which means that the representation of objects and events can be adapted to the specification of the location system.

In summary, the contributions of this work are *i)* the specification and enforcement of security policies using a multi-model language. These policies can be made dependent on history events without compromising usability and performance, *ii)* the implementation of an extensible and interoperable tracking and notification mechanism, with the possibility to define complex notification conditions.

In the next section we discuss some related work. Section 3 describes the architecture of the Location Service, focusing on the main components and their interactions. Section 4 presents the solution to ensure policy enforcement. Some of the implemented policies are described in Sect. 5, and conclusions are presented in Section 7.

2. RELATED WORK

The interest in location privacy has been growing with more services being able to take advantage of persons and objects locations. Mainly, three lines of research can be identified: one that takes the object location and blurs it [1], other that anonymizes users [2, 13] making them indistinguishable and finally, one that takes into account security policies defined by the users of the system. Typically obfuscation deals with the problem of what location accuracy should be reported to location consumers, not dealing with conditions like history of events or the origin of the location request. On the other hand, anonymization is applied in scenarios were the real identity of the user is not relevant, e.g. receiving advertisement when arriving to a defined shopping area. If the location consumer wants to know the location of someone or something in particular, it will not be possible with this technique.

Location privacy with the enforcement of security policy has been a topic of research for some time. Security policies of persons, objects or places, can be made dependent on several location privacy primitives (geographical area, time interval, historical access, etc.) [2, 17]. Each of these aspects can be combined to form a user, object or place, *security policy*.

Leonhardt and Magee [10] present a system where the access control is based on multi-target and multi-object policies. To simplify the management, the system has three levels of policy control: access, visibility and anonymity. The Aura project [5] incorporates a location module which, besides being able to handle multiple sources of positing information, is also structured to protect access to people's location [6]. Their option was to use the SPKI/SDSI infrastructure, giving the possibility, among other things, to delegate location access rights.

LocServ [14] represents each person policy by a group of validators responsible for the evaluation of each location request. The implementation of these validators can go from a software that interrogates the user for each request, to a generic decision maker based on, for example, a security policy file. Context Fabric [7] is a middleware to organize and promote communication between different *information spaces* where users keep their information (e.g. location). Associated to the information in each of these spaces is a description of privacy related actions that the middleware as to attend to, e.g. the requester of the information cannot be at a given building.

More recently, Opyrchal [15] focuses on adding support to location privacy in a content-based publish subscribe middleware. Their system let publishers, i.e. users, to control dissemination of location information they own. Publishers can do so by specifying to which users and in what conditions the disclosure of information is possible, using the KeyNote Trust-Management System [4]. People Finder [8] takes a different direction, applying techniques of machine learning to automatically adjust each user policy based on their satisfaction of the location information disclosure.

The work in [1] applies obfuscation techniques to location information based on user's privacy preferences. In our work, we do not attempt to tamper with location data, instead we allow users and administrators to define/use policies that rule the disclosure of location information for queries and notifications. The work in [13] assumes the existence of untrusted servers from which users want to hide their exact location; this is achieved by anonymizer nodes that reduce location precision to cloaked spatial areas. In Jano, location servers are trusted, nevertheless, the two works could be combined with enriched support for policies. Cooperative sensing is addressed in [3]: user nodes submit sensing tasks to accessible mobile devices of other users. To ensure privacy, all communication is anonymized. In Jano, we do not attempt to recruit other users' devices but deployment of sensing tasks could be defined, reused and enforced by taking advantage of Jano support for policy definition and enforcement.

Common to all these works is the lack of support to make decisions based on past events. The authors in [15] recognize the need to support history-based policies, but their work is unable to do so. Performance evaluation of the component used to evaluate policies is not mentioned, with exception of [15], where the authors conclude they need a more efficient policy evaluator. The adaptability of the policies to different organizations where users, objects and places have different characterization is also not the main issue.

3. ARCHITECTURE

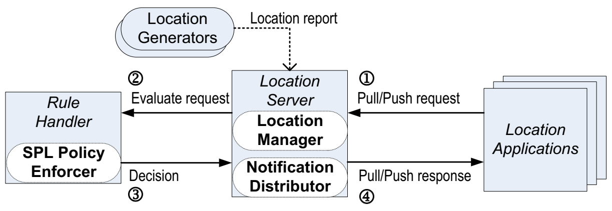


Figure 1: Jano Location Service Architecture

Figure 1 presents the high-level architecture of Jano. The figure depicts Jano's solution for the Rule Handler and the Location Server modules, complemented with a workflow between all parts. Consider the following scenario. A Location Application, e.g. a directory service, is used by *Alice* to ask where *Bob*, her project mate, is located in campus (step 1). Jano evaluates *Bob*'s policy (steps 2 and 3) and if the request is accepted, *Bob*'s location is disclosed with a certain degree of accuracy (step 4). Later, *Alice* uses the campus notification service (another example of a Location Application) requesting to be notified by *Short Messaging Service* (SMS) when the book she ordered arrived at the reception after going through the library for registration. These two kind of interactions with the Location Server are named *pull* and *push* requests, respectively.

The last kind of interaction is possible because the Location Server produces two location events: *i) A arrived* at place *P ii) A leaved* place *P*, being *A* a person or an object. Location events are based on the information collected from Location Generators. These components represent the source of location information and have the responsibility of translate it to a common hierarchical representation with the following format `<domain>/<sub-domain1>/.../<sub-domainN>`. Because of this common format, policies can be specified independently of the detail of the low level positioning technologies, and can encompass many locations with few rules. The information can be reported by persons or objects equipped with technology capable of determining their location, or some kind of tracking technology.

We focused our work on the development of an efficient and adaptable Rule Handler, named SPL Policy Enforcer. Our Policy Enforcer enforces *Access Control Policies* and *Notification Policies*. These policies are associated to persons, objects and places. They regulate if a *pull* or *push* response, can be given, controlling the disclosure of location information. The Policy Enforcer applies access control policies after a *common policy* is enforced. This policy gives the opportunity for the site administrator to enforce a set of common rules; for example, "mail objects can only be localized by their receivers and if the object has already left the distribution department".

Notification Policies are used by the Notification Distributor to evaluate the need and the authorization for a *push* response, i.e. a notification. This evaluation happens each time the Location Manager generates a *location event*. In the previously presented scenario, each time the book enters or leaves a place, the notification policies of *Alice* and other users of the system are evaluated to determine if a notification is needed. SPL is a language for the specification of policies regulating access control, thus the usage of such policies in a notification context is a novel approach. This feature has the following advantages: *i)* makes it easy to use past location events when determining the notification conditions and *ii)* it is a more general approach because the system needs not be hard-coded with the parameters that will be considered in the notifications.

In both cases, the actions of the Policy Enforcer are governed by policies specified and enforced using SPL. These policies depend on current and past interactions with Jano, e.g. location requests and *location events*. Users of Jano need not learn SPL, because Jano provides a library with a pre-defined set of location control and notification policies. Users only have to parameterize them according to their needs.

The next section gives a brief description of SPL, applying the language elements to the context of Jano. Section 5 also presents examples of history-based access control policies and notification policies.

4. LOCATION PRIVACY

The Location Service presented here is adaptable in the sense that the characterization of persons, objects and places can reflect the information available at the site where Jano is to be deployed, e.g. person's department, person's current activity, person's current security level.

Jano imposes minimum restrictions to the structure of policies governing the disclosure of locations information. To accomplish this we use SPL, a multi-model security policy language. The main objective of SPL is to support an environment where authorization policies can be expressed, using a combination of known policy models (i.e. MAC, DAC, history-based, etc.), but also other models that are specific to problem domains. The next sections show how policies are built with SPL, using the context of the Jano Location Service.

4.1 SPL Policies Structure

SPL[16] is composed by four basic blocks: entities, sets, rules and policies.

Entities are typed objects, described in the language as a group of properties. Figure 2 shows the definition of the types for the current implementation of Jano. If Jano is to be used in an environment where users are also characterized by a clearance level, a new property could be added to the `object` type.

During the evaluation of a policy, when a reference is made to a property of an entity (e.g. *where* of type *object*), this will result in consulting the Jano platform for the requested information. How this is done is not under direct control of SPL. Jano implements an adapter framework to give SPL the necessary information for the properties of the external types.

Rules are logical expressions that can take one of three values: `allow`, `deny` or `notapply`. Client system communicate with SPL using *events*. The goal of each rule is to decide on the acceptability of a SPL event. In Jano this events cor-

```
// characterization of Jano places
type place {
   string name;
   // policies regulating access to the place
   policy set accessControlPolicies;
}

// characterization of Jano objects
type object {
   string id;
   // last known location
   place where;
   // groups to whom this object belongs
   group set groups;
   // access control policies
   policy set accessControlPolicies;
   // notification policies
   policy set notificationPolicies;
}
```

Figure 2: Definition of entity types

```
type event {
   // kind of interaction between Jano
   // and SPL (pull, arrive, leave, accuracy)
   string action;
   // initiator of the request
   object author;
   // target of the request
   object target;
   // place, target of the request
   place targetPlace;
   // event generation time
   number time;
}
```

Figure 3: Definition of the type event

respond to pull requests, originated from the users of the location service, and to location events, originated from the Location Server. SPL events are typified as can be seen in Figure 3, which presents the SPL event defined to be used in the communicating between Jano and SPL.

The event representing the current interaction is known as the *current event*, and a rule can access it as `ce`. Figure 3 shows the definition of the event type, used by the Policy Enforcer. The `action` field identifies the type of interaction. The `author` field can be the person making the location request or the originator of the location event. Field `target` is the person or object to whom the location request refers. Field `targetPlace` refers to the place inquired in a *pull* request or the place in a location event.

Rules can be simple or composed. A simple rule has two distinct parts: domain expression and decide expression. The *domain expression* determines the applicability of the rule. The *decide expression* decides on the acceptability of the event. The composed rule is a composition of other rules using tri-value logic operators. Figure 4 shows a composed rule that evaluates only to `allow` or `deny`. The policy evaluates to `allow` only if *i)* a person is querying its own location and *ii)* The current event is a pull request (i.e. *action* is `Get_Location`), the requester has the unique identifier of `alice@inesc.pt` and the last known location of the owner of the policy (i.e. the target) is `inesc/office600`.

Policies are groups of rules and sets, forming a logical unit. Each policy has one *query rule*, which is distinguishable by the question mark that precedes her definition. This

```
SpecialRoom:
   // domain expression
   ce.action = "Get_Location" ::
   // decide expression
   ce.author = "alice@inesc.pt" &
   ce.target.where = "inesc/office600"
TheOwner: ce.target = ce.author :: true;
Composed: TheOwner OR SpecialRoom

policy AllowedRooms {
    string set allowedRooms;
    InRoom:
       ce.action = "Get_Location" ::
             ce.author = "alice@inesc.pt" &
             ce.target.where IN allowedRooms;

    ?AllowedRooms: TheOwner AND InRoom
}
```

Figure 4: Rules and policy

rule is the *entrypoint* of the policy. Figure 4 shows a policy that integrates the previous rules with a modification: only allows the disclosure of the target location if he is in one of the rooms contained in the set.

Different users can use this policy but with different room names. This is a big difference when compared to other policy enforcement languages, making it possible to define a set of meta-policies that can pe particularized to a domain and letting users/administrators instantiate them with the values they want.

SPL policies are not written by persons using the Location Service, but by the site *policy designer*. The *policy designer* responsibility is to create a set of policies adapted to the domain where Jano is to be used (e.g. office building, university campus, hospital).

4.2 History-Based Policies

The disclosure of location information can be dependent on previously location events or accepted pull requests. A usual scenario is to limit the number of location requests (or in alternative, request frequency or the cardinality of the set of unique results provided), made by the same person, to a given target, to avoid tracking.

Other situation can be to avoid cross-referencing information about persons in two different rooms, e.g. if *Alice* already obtained a list of persons in *room A*, then she cannot obtain the list of persons in *room B*. In this scenario *room B* would have a symmetric policy. Figure 5 presents a rule where the request is allowed only if in the past there were no more than `maxEvents` push requests for the same target made by the same author. If this policy is associated to *Alice* and instantiated with `maxEvents` equal to `3`, then she accepts at most three requests in sequence, from the same author.

```
policy TrackingLimit {
   ?TrackingLimit:
      EXIST AT_MOST maxEvents pe IN PastEvents {
         pe.action = "Get_Location" ::
         pe.author = ce.author & pe.target = ce.target
   }
}
```

Figure 5: History based policies

To enforce this type of policies, a virtual event log is used. This special log is referred as the *PastEvents* set and does

```
policy OnlyOutsideMailRoom {
   ?OnlyOutsideMailRoom:
      EXIST pe IN PastEvents {
         pe.action = "Leave" &
         pe.targetPlace = "MailRoom" :: pe.target = ce.target
   }
}
```

Figure 6: History based policies

not match a concrete implementation of a global log, only the semantics are of a global log [16]. The log is associated to each user's policy. It is the responsibility of the Policy Enforcer to fill this log, adding successful pull requests and location events. Adding an SPL event to this log is an optimized operation in which only events relevant to the history policy are included. In the policy of Figure 5, only request location events are relevant. If, a user enter or leaves a place, that event will not be record by this policy log. This promotes logs with reduced sizes thus fostering scalability.

Figure 6 shows a policy where a location event *leave* is taken into account. If this policy is attached to a mail object it can only be located outside the mail distribution room. Section 5.1 shows examples of history-based notification policies where location events (i.e. *arrive* and *leave*) are considered to decide whether a notification is needed and in order.

5. IMPLEMENTATION

Figure 7 depicts some implementation details. Jano *programming interface* (i.e. API) has two main services: *i)* for query and administration purposes and *ii)* for reporting location information. The query service allows for location application to *i)* send pull requests *ii)* manage access control policies through the administration interface and *iii)* manage the push interface. The reporting service is used by the location generators.

Each consumer of location information and each location generator can be implemented in any language or platform. To facilitate this goal, Jano API is implemented as a Web Service, using the framework JAX-WS 2.0 [9]. On top of Jano API, a web application was developed to provide a human *Graphical User Interface* (GUI), to the functionalities previously described. Using this GUI, a non-SPL expert human user can make not only location requests but also select and provide the necessary parameters for his access control and notification policies. A GPS and RFID *translator* have been developed, both using the .NET platform and the C# language [11].

For demonstration purposes we describe how Jano can be used to implement a useful location control policy to ensure the intended privacy in location services. We have designed a model where there is a common policy, presented in Figure 8. This policy refers to every policy specified for each target or place (only the target policy is represented), together with another rule which states that every target can know its location. Policies specified specifically for a target or place are kept inside groups addressed by the targets or places associated with them (property `accessControlPolicies` in Figure 2).

Policies of targets and places can be specified independently, which can result in conflicting rules. For example,

```
policy CommonPolicy {
// Evaluate all access control policies of the target
   accessControl:
      FORALL policy IN ce.target.policies
      { policy };

// Always allow the request, if the author is the target
   selfPolicy:
      ce.target = ce.author :: true;

   ?CommonPolicy:
      selfPolicy OR accessControl;
}
```

Figure 8: Common policy

```
policy GroupsInterval {
   allowedGroupInfo set groupsInfo;
// Rule for evaluating about disclosure of location
   accessControl:
// for each node in the allowed list
      EXIST node IN groupsInfo {
// for each day in the allowed days
         EXIST day IN node.daysSet {
// search for the allowed group name in the
// list of groups whom the author of the request belongs
            node.allowedGroup IN ce.author.groups
            :: day.dayOfWeek = ce.dayOfWeek &
                 (ce.time.hour >= day.start.hour &
                   ce.time.hour <= day.end.hour)
         }
      }
// accuracy of the response
   normalAcc: ce.action = "Max" :: ce.time.hour >= 15
   defaultAcc: ce.action = "Low" :: true;

   ?GroupsInterval:
       accessControl AND (normalAcc OR defaultAcc);
}
```

Figure 9: Personal access control policy

Alice is not allowed to see *Bob*, but *Alice* can see who is at *P*. In this scenario, if *Bob* is located at *P*, and *Alice* makes a request to see who is at this location, Jano would not include *Bob* in the response. Jano will only disclose location when the combination of target's and place's policies allow it.

Target access control policies must answer to two types of requests: location and accuracy. The objective of the first request is to know if the disclosure of location is accepted in respect to a given requester. The second type of request inquires what kind of accuracy should be applied to the location information. For a location to be disclosed, the necessary SPL events are built and sent to the Policy Enforcer which will evaluate the target policy in order to decide on these two aspects.

The implementation of Jano has several access control policies. In this article we focus on one that could be applied to a variety of environments (e.g. university campus, enterprise building), presented in Figure 9. The policy regulates two events, *location request* and *accuracy.* To determine if the location can be know by the author of the request, the policy takes into account the allowed groups of users that can obtain the location of the target. For each of these groups, the policy determines if the author belongs to it. Each allowed group is represented as an object of the `allowedGroupInfo` type, describing the group name, the ac-

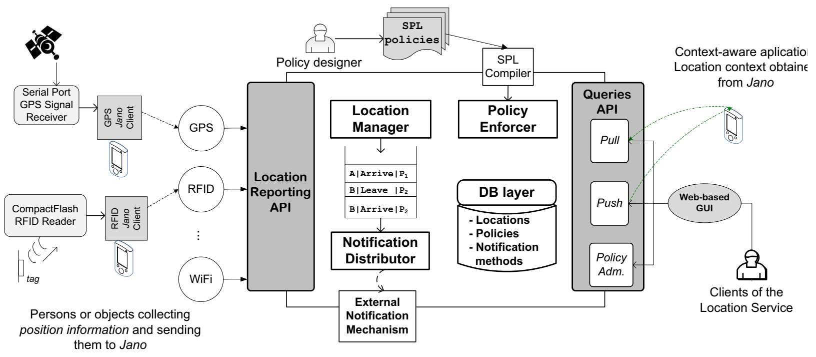


Figure 7: Jano implementation

```
policy SNotify(object id, place place, string evType) {
    ?SimpleNotify:
        ce.author = id ::
        ce.action = evType & ce.targetPlace = place;
}

policy VisitAfter(object id, place orig, place dest) {
   ?VisitAfter:
      EXIST pe IN PastEvents {
         ce.author = id & ce.action = "Arrive" &
         ce.targetPlace = dest ::
         pe.author = id & pe.action = "Leave" &
         pe.targetPlace = orig
    }
}
```

Figure 10: Notification policies

curacy that the location should be reported with and in what period of the week. Each of these objects are stored in the policy instance, in particular in the `groupsInfo` property.

Each policy, goes through the SPL compiler, which produces an *enforceable policy* in the form of a Java class. Instances of these classes, with proper initialization, are attached to each target, as access control or notification policies, that is, to the `accessControlPolicies` or to the `notificationPolicies` properties in Figure 2. The set of policies associated to each target forms a graph of objects which is updated each time a new policy is added or removed.[2]

5.1 Notification Policies

Jano sends notifications based on the evaluation of notification policies associated to users. Using SPL, notification policies can be specified with different conditions, adapted to the site where the location service is used.

Figure 10 presents a simple notification policy (`SNotify`), parameterized by the name of an object, the name of a place and the location event of interest. This policy could be used by *Alice* to be notified by Jano when *Bob* arrives to `inesc/floor6`. If so, a policy with the given parameters would be instantiated, like in the following example:

```
new SimpleNotify(bob,
   new place("inesc/floor6"), "Arrive").
```

When the Notification Distributor receives a location event containing information that *Bob* has arrived to `inesc/floor6`, he will contact the Policy Enforcer, with the objective of knowing who wants to be notified. For this, a new SPL event is built, where `author` is *Bob*, `action` is `Arrive` and `targetPlace` is `inesc/floor6`. Then, this event is used to evaluate each user pending notification policies. If the policies allow it, a notification will be sent, through a communication channel (e.g. web service, e-mail, sms) previous configured by the user.

Jano can efficiently enforce notification policies with history based rules. This could be used for a user to be notified about an object trajectory inside his organization. For example, a previously ordered book can arrive at the reception, but this event is only interesting if the same book has already been through the library to be cataloged. Figure 10 shows a parameterized history-based policy, called `VisitAfter`, which can be instantiated to represent the previously described scenario, and associated to *Alice*:

```
new VisitAfter("book:Understanding Privacy",
  new place("inesc/reception"),
  new place("ist/library"))}.
```

5.2 Support for Policy Dynamism and Log-size Management

Policy dynamism is an important issue in policy-driven systems. Users and administrators may want to install new policies, edit current ones, or remove some policies altogether from the system. Policy edition amounts to policy removal and reinstall. When a policy is removed, no special support is required except to take into account that some events stored in the log may no longer be required. When a new policy is installed, relevant events start being recorded.

If the system is required to analyze all past events in the context of any newly installed policy (*full-log mode*), then all events must be available persistently in some form (in some secondary offloaded storage), which raises issues of log-size and log processing. Log size can be addressed with partitioning, distribution and compression but eventually the whole data of events must be preserved until some threshold date for event garbage collection e.g., one week, month, a year).

To ensure that log processing for policy evaluation maintains scalability when in *full-log mode* (i.e., storing all events regardless of their relevance for the current policies), we may employ bloom filters[12] to allow efficiency in ulterior testing of such past events with newly installed policies. Thus, the occurrence of events of a given type in a specific time-frame (a partition of the log) is efficiently stored for later testing. To fetch actual event data and resolve false positives, the

[2]As an appendix, we address Jano web interface to support the configuration of policies.

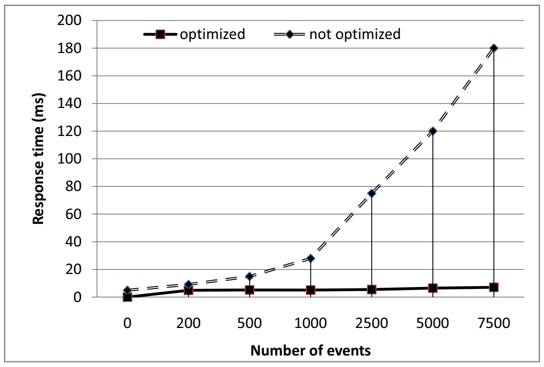


Figure 11: Results for growing number of history events

secondary storage must be accessed.

6. EVALUATION

In this section we present the results obtained in the evaluation of the Policy Enforcer, while enforcing an access control policy, with and without history rules. We choose this type of policy because they must always be evaluated, even before a *push* response. The results that will be discussed can also be applied to notification policies.

The policy considered was the one presented in Figure 9. In this policy, the number of groups allowed by the target ($TargetGroups$) must be transversed linearly. Each of these groups is looked up in the groups to which the author of the request belongs ($AuthorGroups$). This search has a worst time computation of $log(AuthorGroups)$. Experiments where made with different numbers of groups allowed and groups to which the author belongs to. In our test scenarios we considered that $AuthorGroups$ tends to be much bigger than $TargetGroups$. For example, if a policy defines 15 allowed groups (a number already hard to manage by the owner of the policy), and the author belongs to 600 groups, the policy will be evaluated in approximately $5ms$.

A critical aspect in the evaluation of the Policy Enforcer is the measurement of the necessary delay introduced by the evaluation of history-based policies. The SPL compiler produces specific data structures to store the events needed in the evaluation of history-based policies. These data structures are optimized when compared with a simple list of events because: *i)* they take into account the aspects considered in the policy and only store the relevant information for each event, *ii)* location events that are equivalent are stored as just one object, referring that the event occurred a certain number of times.

Figure 11 shows the delay introduced by the evaluation of a policy based on the history rule presented in Figure 5. Tests were made using the optimized log of SPL and a non optimized log. In the optimized log, if a person makes 1000 location requests to 20 different objects (i.e. targets), only 20 event objects will be record, instead of 1000. This optimization has a significant impact in the space needed to store the history log, and more importantly, in the evaluation time of history policies, as can be seen when compared with the non optimized log where all events are record without regarding their redundancy.

Therefore, these results are very encouraging regarding the scalability and performance of Jano's policy evaluation and enforcement core.

7. CONCLUSION

In recent years, location information has been increasingly used in *context-aware* applications with the goal of augmenting the mobile services offered to the end user. Some examples are: advertisement on mobile devices from the shop we are visiting and presentation of more information related to the product we are shopping or the work of art we stand by.

For an effective deployment and acceptability of location services, they must support the specification and enforcement of security policies. Users want to specify under what conditions their location can be disclosed. In some scenarios this can depend on past events like, how many times a location request was made or what places have been visited. Finally, the kind of properties that are relevant to characterize each object or event is different for each location service.

In this document we have presented Jano, a Location Service capable of enforcing security policies. Although the instant reporting of locations (*pull* requests) is essential, in many situation users want to be notified about some kind of location related event, i.e. *push* requests. The policies enforcing the access to location information and the conditions used in the specification of *push* requests are made through SPL, a multi-model authorization platform. Using SPL, policies can be implemented using a variety of different security models and can be made dependent on the resources of the implementation site, and not the opposite. We have presented some of the policies developed, and evaluation results which have shown that performance is not compromised. The usability of the system is enhanced by the simple GUI developed for users to control their security policies.

8. REFERENCES

[1] C. A. Ardagna, M. Cremonini, E. Damiani, S. D. C. di Vimercati, and P. Samarati. Location privacy protection through obfuscation-based techniques. volume 4602 of *Lecture Notes in Computer Science*, pages 47–60. Springer, 2007.

[2] A. R. Beresford. Location privacy in ubiquitous computing. Technical Report 612, University of Cambridge, January 2005.

[3] C. Cornelius, A. Kapadia, D. Kotz, D. Peebles, M. Shin, and N. Triandopoulos. AnonySense: Privacy-aware people-centric sensing. In *Proceeding of the 6th international conference on Mobile systems, applications, and services*, pages 211–224. ACM, 2008.

[4] M. B. et. a. Rfc 2704: The keynote trust-management system version 2, September 1999.

[5] D. Garlan, D. P. Siewiorek, A. Smailagic, and P. Steenkiste. Project aura: Toward distraction-free pervasive computing. *PERVASIVE Computing*, pages 22–31, June 2002.

[6] U. Hengartner and P. Steenkiste. Protecting access to people location information. In *First International Conference on Security in Pervasive Computing*, pages 25–38, 2003.

[7] J. I. Hong. An architecture for privacy-sensitive ubiquitous computing. In *In MobiSYS Š04: Proceedings of the 2nd international conference on mobile systems, applications, and services*, pages 177–189. ACM Press, 2004.

[8] P. G. Kelley, P. H. Drielsma, N. M. Sadeh, and L. F. Cranor. User-controllable learning of security and privacy policies. In *AISec*, pages 11–18, 2008.

[9] J. Kotamraju. Jsr 224: Java api for xml-based web services (jax-ws) 2.0. JSRs: Java Specification Requests.

[10] U. Leonhardt and J. Magee. Stability considerations for a distributed location service. *J. Network Syst. Manage.*, 6(1), 1998.

[11] Microsoft. Standard ecma-335 common language infrastructure (cli), 2006.

[12] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networking (TON)*, 10(5):604–612, 2002.

[13] M. Mokbel, C. Chow, and W. Aref. The new Casper: query processing for location services without compromising privacy. In *Proceedings of the 32nd international conference on Very large data bases*, page 774. VLDB Endowment, 2006.

[14] G. Myles, A. Friday, and N. Davies. Preserving privacy in environments with location-based applications. *Pervasive computing*, pages 56–64, 2003.

[15] L. Opyrchal, A. Prakash, and A. Agrawal. Supporting privacy policies in a publish-subscribe substrate for pervasive environments. *Journal of Networks*, pages 17–26, February 2007.

[16] C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes. Spl: An access control language for security policies and complex constraints. In *NDSS*. The Internet Society, 2001.

[17] J. Y. Tsai, P. G. Kelley, L. F. Cranor, and N. Sadeh. Location-sharing technologies: Privacy risks and controls. In *In Research Conference on Communication, Information and Internet Policy (TPRC*, 2009.

[18] U. Varshney. Location management for mobile commerce applications in wireless internet environment. *ACM Trans. Interet Technol.*, 3(3):236–255, 2003.

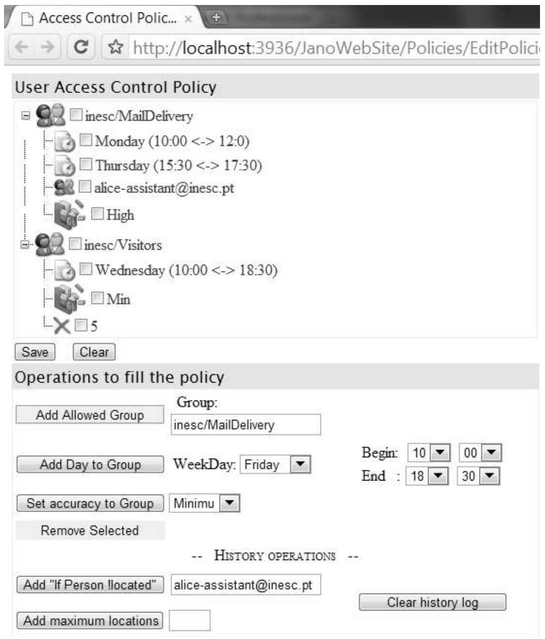
APPENDIX

Web-based GUI

In order to enhance the usability of Jano, a web-based GUI was implemented on top of Jano API, using the ASP.NET platform.Using this GUI, a non-SPL expert human user can make not only location requests but also select and provide the necessary parameters for his access control and notification policies. Figure 12 shows the GUI, during the initialization of the access control policy of user `alice@inesc.pt`. The structure of this policy is based on the one presented in Figure 9, considering also some history-events we will describe next.

The policy parameters in the GUI presented in Figure 12, in the left side of the figure, are organized as a tree. Each root node represents a group of users. Each leaf node is a rule restricting access in some aspect to the location of the user (who is interacting with the GUI).

In this policy the user can specify the acceptability of a location request or location event (regarding his location) dependent on two history-based conditions *i)* the location requests or location events haven't exceeded a certain limit, e.g. users of group `inesc/Visitors` can get a maximum of 3 location reports *ii)* the group to whom the requester belongs has not yet located someone else, e.g. only report location to group `inesc/MailDelivery` (the mail distribution workers), if anyone in this group has not yet located a person with *id* `alice-assistant@inesc.pt`. Using the bottom part of the GUI, the user can add or remove new groups, date intervals, and parametrize the history-based rules. The GUI developed in Jano supports a large number of operations so that most of policy specifications can be easily done without knowing SPL.



http://localhost:3936/JanoWebSite/Policies/EditPolici

inesc/MailDelivery
Monday (10:00 <-> 12:0)
Thursday (15:30 <-> 17:30)
alice-assistant@inesc.pt
High
inesc/Visitors
Wednesday (10:00 <-> 18:30)
Min
5

Group: inesc/MailDelivery
WeekDay: Friday
Begin: 10 00
End : 18 30
Minimu
alice-assistant@inesc.pt

Figure 12: Web-based GUI of Jano