J Internet Serv Appl
DOI 10.1007/s13174-012-0065-z



# Jano: location-privacy enforcement in mobile and pervasive environments through declarative policies

**José Simão · Carlos Ribeiro · Paulo Ferreira · Luís Veiga**

Received: 8 August 2011 / Accepted: 16 July 2012




**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. Therefore, location systems must support the specification and enforcement of security policies (including history-based) allowing users to specify when, how and whom may know their location. We propose a middleware platform named Jano [Jano (or *JANVS* in latin) is the god of doors and gates in the roman mythology. He is usually depicted with two or four faces turning in opposite directions.] 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 person's or object's location. Jano implementation integrates several location technologies (e.g. GPS, RFID, etc.) and deals with the related heterogeneity aspects. It provides a web-based interface that facilitates policy specification, and its evaluation shows good performance, embodying a number of optimizations regarding bandwidth, process and storage savings.

J. Simão
Instituto Superior de Engenharia de Lisboa,
Rua Conselheiro Emídio Navarro No.1,
1959-007 Lisbon, Portugal
e-mail: jsimao@cc.isel.pt

J. Simão · C. Ribeiro · P. Ferreira · L. Veiga (✉)
Instituto Superior Técnico, UTL / INESC-ID Lisboa,
Rua Alves Redol No.9, 1000-029 Lisbon, Portugal
e-mail: luis.veiga@inesc-id.pt

C. Ribeiro
e-mail: carlos.ribeiro@inesc-id.pt

P. Ferreira
e-mail: paulo.ferreira@inesc-id.pt

**Keywords** Location-awareness · Privacy · Declarative policies · Security · Middleware

## 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 located so that we can meet or find them, respectively. 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 [31], or to obtain detailed information about the work of art we stand by at a museum).

Even though the above-mentioned location-based applications are varied and very useful, privacy arises as a main concern. As a matter of fact, 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 under what condition, may someone be allowed to locate me or know I am nearby?". This can be as simple as restricting a time interval; for example: "Bob can only locate Alice 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 disclose her location at isolated instants but not being tracked, i.e., reveal several locations in sequence. In addition to the previous situation (e.g., knowing the location of Alice) there are cases in which it is important



Journal: **13174** MS: **0065** ☐TYPESET ☐DISK ☐LE ☐CP Disp.:**2012/10/6** Pages: **20** Layout: **Large**

that a user, e.g., Bob, is notified of some location-related event such as "Alice has arrived at the campus". Finally, when a person wants to disclose her location, e.g. Alice, she may do so with different precisions, depending on the requester or the situation Alice is involved. For example, in an emergency scenario, it is of utmost importance that Alice is located with maximum precision; in contrast, Alice may only allow her students to know her location with minimum precision (e.g., inside the campus or not).

To address all these scenarios, the location system must be capable of responding to location requests but, at the same time, evaluate each request and decide whether it is authorized or not, based on some previously specified policy. Thus, the goal of this work is as follows: for location-based services, to support the specification and enforcement of complex security policies, including those based on history events, without compromising usability and performance. Such policies are to be defined and enforced on a location service that supports both synchronous, i.e., *pull*, and asynchronous, i.e., *push*, requests.

It is worth noting that the goal stated above is to be attained while providing a widely applicable solution, i.e., that can be used in a generic location infrastructure. Thus, the policy monitor that enforces the above-mentioned policies cannot be made as a group of static or built-in rules. As noted in [7], organizations use different approaches and philosophies to structure and configure their units and collaborators (e.g., hierarchical, flat, etc.). Policies are to be defined independently of the location system and taking into account the existing organizational model of the site where location policies are to be enforced.

Given the dynamics of the location information, past events are particularly important to consider. When a user makes a query for someone's location, or when he arrives at, or leaves from any location, 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 (history-based in this case) mainly for performance and scalability reasons.

Other location services that enforce some notion of privacy [22,23] 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 location events produced by the location service in the notification decision process.

This paper presents Jano, a generic multi-technology Location Service, supporting the specification and capable of enforcing flexible declarative 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 left the cafeteria"). Thus, Jano supports two types of policies:

*Access Control policies* enforce the requirements of users and owners of places regarding the disclosure of location information. Such 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 the above-mentioned policies to determine if a notification is needed and allowed.

To define both types of policies (access and notification) Jano uses an extended version of the Security Policy Language (SPL) [25]. SPL is a policy language particularly suitable for location services, because it allows the definition of models comprised by elements specifically adapted to location semantics; namely, SPL allows for the definition of history-based policies which are an important element for the definition of location 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.

Regarding the precision of location information disclosure (which depends on the requester or on the current situation, as previously mentioned) in SPL, as in most authorization languages, one can only get this feature doing several queries with decreasing precisions until one is accepted. Therefore, we have extended SPL with an *awareness operator*. With this operator, the policy designer (and he alone) can define a logical expression or rule whose result may be used to determine the cause of a denied location request. When applied to the precision of location, as in the previous examples, Jano will only need to make one policy evaluation to determine if maximum precision is allowed and, if not, with which precision can the request be satisfied, improving efficiency while ensuring that there is no information leakage.

In summary, the contributions of this work are as follows:

(i) The specification and enforcement of privacy-related security policies using a multi-model language (i.e., not tied to any specific authorization model such as RBAC, MAC, etc.) [25]. 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.



Journal: 13174 MS: 0065 TYPESET DISK LE CP Disp.:2012/10/6 Pages: 20 Layout: Large

(iii) An extension to SPL logic and semantics, and core implementation, to improve the performance of location precision queries through the use of a new awareness operator.

The rest of the paper is organized as follows. Section 2 describes the overall architecture of Jano, focusing on the main components and their interactions. With increasing detail, Sect. 3 presents the solution to ensure policy enforcement, including the logics of history-based policies and the modifications in SPL to support the new awareness operator. Some of the implemented policies are described in Sect. 4 along with details about the implementation of Jano's core. Section 5 discusses the most relevant performance aspects of Jano and presents a use case illustrating its functionality. In Sect. 6, we discuss some related work. Section 7 concludes the article.

## 2 Jano architecture

Figure 1 presents the high-level overall architecture of Jano. Location applications request the user location and set notification conditions. The Location Server collects information from different sources (Location Generators); it works as a Policy Enforcement Point (PEP) delegating to the Policy Decision Point (PDP) (i.e., Rule Handler) the decision about returning location requests and location events. The figure also illustrates the workflow among all components.

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 precision level (step 4). Later, *Alice* uses the campus notification service (another example of a Location Application need) requesting to be notified by *Short Messaging Service* (SMS) when the book she ordered arrived at the reception desk after going through the library for registration. These two kinds of interactions with the Location Server are named *pull* and *push* requests, respectively.

The latter kind of interaction (i.e., push requests) is possible because the Location Server produces two location events: *i) A arrived* at place *P*, and *ii) A left* place *P*, *A* being a person or an object.

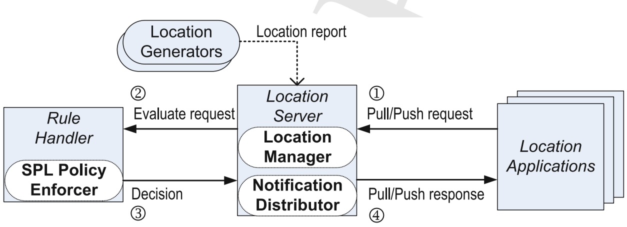



**Fig. 1** Jano architecture overview

Location events are based on the information collected from Location Generators. These components represent the source of location information and have the responsibility of translating it to a common hierarchical representation with the following format:

`< domain > / < sub-domain1 > / . . . / < sub-domainN >` .

Although not fundamental for this work, such an hierarchical representation has some advantages when compared to other approaches (which could also be used in Jano). As a matter of fact, due to this hierarchical common format, policies can be specified independently of the detail of the low level positioning technologies employed, and are able to encompass many locations with just a few rules.

Jano provides an efficient and adaptable Rule Handler, named SPL Policy Enforcer, that enforces both *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 common policy gives the opportunity for the site administrator to enforce a set of common global 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, at most, once for each time the Location Manager generates a *location event*. In the previously presented scenario, in which Alice is interested on a book she ordered, each time the book enters or leaves a place, the notification policies of *Alice* are evaluated to determine if a notification must be sent (or not).

It is worth noting that SPL is a language originally conceived for the specification of access control policies. Thus, in Jano, we extended SPL to support policies in a notification context. Following on the example previously described, Alice would choose an SPL policy representing the desired notification situation and specifies which book she was interested in. This results in the *instantiation* of the policy, which will then be associated to her notification policies. This resulted in a novel approach that, while taking advantage of SPL features, allows Jano to have the following novel properties: (i) it is easy to use past location events when determining the notification conditions (based on history-based policy support from SPL), and (ii) it is a general approach because the actual parameters that will be considered for notification purposes, regarding any given person and/or object, are not built-in or hard-coded in Jano.





Journal: 13174 MS: 0065 TYPESET DISK LE CP Disp.:2012/10/6 Pages: 20 Layout: Large

In both cases (i.e., access control and notification policies), the actions of the Policy Enforcer are governed by policies specified and enforced using SPL, with extensions described in the next section. 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 access control and notification policies, and policy templates or idioms. Users, then, only have to parameterize them according to their needs using a web-based interface (described in Sect. 4.5).

The next section gives a description of SPL, focusing on the language elements relevant to Jano, and those newly introduced (e.g., regarding policy awareness). Section 4 presents examples of history-based access control policies and notification policies, showing the generality and expressiveness of Jano's location policies.

## 3 Location privacy

An important characteristic of Jano is its adaptability, mostly due to the Location Server. This means 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 regarding the structure of policies governing the disclosure of location information. To accomplish this, as already mentioned, we extended SPL. The main goal of SPL is to support an environment where authorization policies can be expressed using a combination of known policy models (i.e., MAC, DAC, RBAC, history-based, etc.) among others [26]. The next sections show how the policies, relevant to the context of the Jano Location Service, can be built with SPL.

### 3.1 SPL policies structure

SPL [25] comprises 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. Each object has its own policies regarding location disclosure (i.e., access control) and notifications. These policies are referred by properties designated `accessControlPolicies` and `notificationPolicies`, respectively. Moreover, if Jano is to be used in an environment where users are also characterized by a clearance level, a new property is to be added to the object type. Next, we describe how these policies are defined.

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 provide SPL the necessary information for the properties of the external types, retrieving them by interfacing with different technologies.

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

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

**Fig. 2** Definition of entity types. `object` represents locatable entities (i.e., persons or objects). Each object belongs to a group and can define both access control and notificatio. policies

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

**Fig. 3** Definition of the type `event`

*Rules* are logical expressions that can take one of three values: `allow`, `deny` or `notapply`. Client systems communicate with SPL using *events*. The goal of each rule is to decide on the acceptability of a SPL event. Thus, as stated above, a rule may allow (`allow`) or deny (`deny`) an event; in addition, an event may be completely irrelevant for that event i.e., not applicable (`notapply`).

In Jano these events correspond to pull requests, originated from the users, and to location events, originated from the Location Server. SPL events are defined as described in Fig. 3, which presents the SPL event used in the interaction between Jano and SPL.

The event representing the current interaction is known as the *current event*, and a rule can access it as `ce`. The `action` field of the event element (Fig. 3) identifies the type of interaction. The `author` field is 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 to. Field `targetPlace` refers to the place inquired in a *pull* request or the place in a location event.

Rules can be simple or composite. A simple rule has two distinct logical binary expressions, separated by the symbol "::"—the domain expression and the decide expression. The *domain expression* determines the applicability of the rule. The decide expression decides on the acceptability of the





Journal: 13174 MS: 0065 TYPESET DISK LE CP Disp.:2012/10/6 Pages: 20 Layout: Large

```
SpecialRoom: // simple rule
   ce.action = "Get_Location" :: // domain expression
   ce.author = "alice@inesc.pt" & // decide expression
   ce.target.where = "inesc/office600" // decide expression (cont)

TheOwner: ce.target = ce.author :: true; // simple rule

CRule: TheOwner OR SpecialRoom // composed rule
```

**Fig. 4** Example of a simple (SpecialRoom) and a composed (CRule) rule

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

**Fig. 5** Example of a policy. The *question mark* identifies the first rule to be evaluated. The rule TheOwner is the same as presented in Fig. 4

event. A composed rule is a composition of other rules using tri-value logic operators, which are extensions of their first order binary counterparts (conjunction, disjunction, negation and logical quantifiers) with a global neutral element, the `notapply` value, i.e., the conjunction or disjunction of any rule with `notapply` is equal to the value of the rule, meaning that if the domain expression of one of the composed rules evaluates to FALSE, the value of its decide-expression is irrelevant for the the result of the composition, whatever the composition type (conjunction, disjunction, or quantification).

Figure 4 shows a composed rule (`CRule`) as it depends on two other simple rules named `SpecialRoom` and `TheOwner` (composed with the operator `OR`). As already stated, rules are designed to decide on a given location request which is represented by the construction `ce`.

Rule `SpecialRoom` evaluates to `allow` when, for the domain expression stated, the corresponding decide expression is TRUE. Regarding the rule `TheOwner`, we can see that, as long as the decide domain is verified (initiator of the request, the author, is the same as the person being located, the target) the decide expression always evaluates to TRUE meaning that the rule result is `allow`.

Thus, the composed rule `CRule` means that a location request is allowed if either:

1. a person is querying his own location; this is enforced by the `TheOwner` rule stating that `ce.target` must be equal to `ce.author` thus requiring the initiator of the request (the author) to be the same as the person being located (the target); or
2. the current location request, in the simple rule `SpecialRoom`, is a pull request (i.e., *ce.action* = `Get_Location`), the requester (i.e., *ce.author*) has the unique identifier of `alice@inesc.pt` and the last known location of the owner of the policy (i.e., *ce.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 its definition. This rule is the entrypoint of the policy. Figure 5 shows a policy that allows the location disclosure of the target if it is in one of the rooms contained in the set `allowedRooms`.

Different users can use this policy as a template but with different room names (i.e., a different set of rooms in `allowedRooms`), which allows flexibility and promotes extensibility. This is a difference between Jano and other policy enforcement systems, making it possible to define a set of meta-policies that can be particular to a domain, and letting users/administrators to instantiate them with the specific values they want.

SPL policies are not written by persons using the Location Service, but by the organization's *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, military installation).

### 3.2 History-based policies

As already mentioned, being able to locate someone or something has been a need over the times. In addition, there are cases in which it is relevant to track a person's location; for example, when security is a concern, it may be important to know if a person has been in rooms R1, R2 and R3 (possibly, for how long in each one). Obviously, such tracking raises important privacy issues; while such disclosure may be acceptable in an industrial environment during working hours, such tracking is not acceptable at week-ends or during other private activities (e.g., during leisure time).

Thus, the disclosure of location information can be dependent on previous location events or accepted pull requests. A usual scenario is to limit the number of location requests or, alternatively, the request frequency or the cardinality of the set of unique results provided, made by the same person, to a given target. This avoids tracking (as the scenario described previously) among other types of inference attacks.

Policy `TrackingLimit`, presented in Fig. 6, shows a rule (`?TrackingLimit`) where the location request is allowed (or not) based on past events. More precisely, 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 (`pe.author=ce.author & pe.target=ce.target`), on the same day. For example, if this policy is associated to *Alice* (`ce.author`) and instantiated with a value equal to three for `maxEvents`, Alice is allowed at most three such requests in sequence.





Journal: 13174 MS: 0065 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2012/10/6 Pages: 20 Layout: Large

```
policy TrackingLimit {
   ?TrackingLimit:
      EXIST AT_MOST maxEvents pe IN PastEvents {
         pe.action = "Get_Location" &
         ce.action = "Get_Location" &
         pe.author = ce.author &
         pe.target = ce.target &
         pe.date = ce.date :: true
   }
}
policy OnlyOutsideMailRoom {
   ?OnlyOutsideMailRoom:
      EXIST pe IN PastEvents {
         ce.action = "Get_Location"::
                        pe.target = ce.target &
                        pe.action = "Leave" &
                        pe.targetPlace = "MailRoom"
   }
}
```

**Fig. 6** Examples of history-based policies

Another example of history-based policies is policy `OnlyOutsideMailRoom`, also presented in Fig. 6. It takes into account the location event *leave* (`pe.action = "Leave"`). If this policy is applied to a mail object (the `target`), this means that such object can only be located after leaving the mail distribution room (`pe.targetPlace ="MailRoom"`). In addition to the examples previously described, Sect. 4.1 shows examples of history-based notification policies where location events (i.e., *arrive* and *leave*) are considered to decide whether a notification is needed.

A critical aspect of the above-mentioned policies (and history-based ones in general) is the size of the log where past events are kept. To enforce this type of policies, a virtual *event log* is used. This special log is referred as the *PastEvents* set; it does not match a concrete implementation of an *event log*, although the semantics is that of a global log [25]. 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.

In particular, in the policy `TrackingLimit` previously described (see Fig. 6) the PastEvents set is searched to determine whether, on the same day (`pe.date=ce.date`), a given requester (`ce.author`) has already made `maxEvents` successful location requests. Only events regarding location requests are relevant. If a user enters or leaves a place, that event will not be recorded by this policy *log*. Furthermore, each event from the same author, regarding the same place and for a given day will not be duplicated, reducing the *log* size. This log mechanism is fundamental as it promotes logs with reduced size thus fostering scalability (more details presented in Sect. 4.2).

### 3.3 Policy awareness

As already mentioned, sometimes, a person may be interested on being located with different precision levels depending on who is willing to know his location and also depending on the purpose. For example, Alice may allow his friend Bob to know her location within a 1 km radius but, for emergency purposes, Alice may allow an ambulance to know her location precisely, i.e., with much greater precision (i.e., 1 m). Jano takes all these scenarios into account while ensuring that there is no uncontrolled information leakage.

Thus, often, it is necessary to localize someone with the best precision allowed by the corresponding policy. However, with most authorization languages and also with (the original) SPL, the way to attain this is very inefficient: several requests must be issued, with decreasing precision, until one is accepted (as is the case with the above scenario where Bob wants to know Alice's location). This inefficiency happens because authorization engines must prevent any information leakage.

However, in some situations, providing the user (e.g., Bob) with the reason why his request was rejected (e.g., for Alice location) contributes to the policy awareness and turns the process of locating someone with the best precision possible much more efficient. Such increased efficiency results from the fact that the access control engine replies with an `allow` or with a `deny`, together with the best precision that makes the policy return `allow`. In fact, there is no need to ask the engine again with a different precision because the system already knows the answer. Therefore, with such an awareness mechanism the authorization engine is called just once, with clear efficiency gains. Once again, taking into account the above-presented scenario, Bob would issue a single request for Alice's location indicating a precision value that is acceptable by the corresponding policy.

In fact, this mechanism can be useful in a more general context to provide policy awareness to users, letting them know why their requests are being denied, without having to contact the help desk for that purpose [28].

In order to enhance SPL with the above-described awareness mechanism, we have extended it with the introduction of a new polymorphic operator: the awareness operator "$". This operator applies to logical expressions (e.g., $(`ce.precision < "Medium"`)) and rules (e.g., `$domain-exp::decide-exp`). It states that if an event is denied because of some condition inside the awareness scope, that information is transmitted back to the access requester, as additional awareness information. It's worth noting that, only the annotated expressions are transmitted to the requester; therefore, policy leakage is kept to a minimum and, more important, always strictly controlled by the policy designer.

The awareness information is provided to the requester as a symbolic binary logical expression indicating the conditions on the event request that are needed to change the policy result from `deny` to `allow` (or to keep the `allow`, if that is the result of the applied policy). If the applied policy returns `notapply`, the awareness information is not specified.





Journal: 13174 MS: 0065 TYPESET DISK LE CP Disp.:2012/10/6 Pages: 20 Layout: Large

The symbolic binary expression is kept on a tree structure where each leaf is a binary logical constraint (e.g., `ev.a > b`), and each node is a binary logical operation. The tree is reduced whenever one of the branches of a node is a constant value, but it is not further simplified; therefore, we may end up with an awareness expression stating that `ce.precision > "Low"& ce.precision > "Medium"`, which could be simplified further to `ce.precision > "Medium"`. However, currently, the present solution was deemed adequate. If one of the branches is constant, it is either the neutral element or the absorbing element of the binary operation, which means they can either be collapsed into the non-constant branch (neutral element) or to the constant branch (absorbing element).

The awareness information is provided to the requester in a tuple together with the policy result $\langle result, awareness\rangle$ in which: $result$ is the usual `allow`, `deny` or `notapply` values that result from the application of the SPL ternary logic to all the rules that comprise the policy; $awareness$ is the binary tree with the awareness information.

The awareness information is provided by the new extensions to the SPL logics; the ternary logic used to compose rules and the binary logic used in the decide expression of each rule. The elements of both logics are now tuples with the original elements and an awareness tag $\langle element, tag\rangle$. For the SPL ternary logic, the first element of the tuple is either `allow`, `deny` or `notapply`; and for the binary logic it is TRUE or FALSE. The tag corresponds to the awareness information and it is, in both logics (binary and ternary), a binary symbolic expression stored in a tree structure.

The following definitions provide the framework used to build the awareness information provided to the requester.

**Definition 1** The tag of a non-annotated binary logical expression (*ble*) is defined as $tag = \overline{ble}$, where $\overline{ble}$ is a value equal to TRUE or FALSE resulting from the evaluation of *ble* with the current event.

**Definition 2** The tag of an annotated binary logical expression ($*ble*), i.e., an expression that was preceded by the awareness operator "$", is defined as:

$$tag = \begin{cases} ble & if\,\overline{ble}. \wedge OnEvent(ble) \\ !ble & if\,!\overline{ble} \wedge OnEvent(ble) \\ \overline{ble} & if\,!OnEvent(ble) \end{cases}$$

where $OnEvent(ble)$ is a predicate that evaluates to TRUE if the expression depends on the current event.

Note that *ble* and !*ble* represent symbolic logical expressions, i.e., non-evaluated, whilst $\overline{ble}$ represents an actual binary value.

**Definition 3** The tag of a rule (rule = {domain-exp :: decide-exp}) is defined as:

$$tag = \begin{cases} \perp & if\,\overline{rule} = \texttt{notapply} \\ tag(\text{decide-exp}) & \text{otherwise} \end{cases}$$

where $\perp$ represents the empty symbolic expression, $\overline{rule}$ the evaluation of the rule with the current event and $tag(\text{decide-exp})$ the tag of the binary logical expression comprising the decide expression of the rule.

Extending the SPL ternary and binary logics to handle awareness tags implies defining two new sets of operators over two new tuples, respectively the $\langle ble, tag\rangle$ (or $\langle b, t\rangle$) for the extended binary logic, and the $\langle rule, tag\rangle$ (or $\langle r, t\rangle$), for the extended SPL ternary logic.

**Definition 4** The extended binary logical operators are defined as:

$$\langle b_1, t_1\rangle \triangle \langle b_2, t_2\rangle = \langle b_1 \triangle b_2, t_1 \bigcirc t_2\rangle$$

$$\textcircled{!}\langle b, t\rangle = \langle\, !b, !t\rangle$$

where $\triangle$ is a placeholder for the binary conjunction (&), disjunction (|) and exclusive disjunction (ˆ), and $\bigcirc$ is a placeholder for $\textcircled{\&}$, $\textcircled{|}$ and $\textcircled{\hat{}}$, which are symbolic operators that are equal to their binary counterparts, with the exception that they take $\perp$ as their universal neutral element. Similarly, $\textcircled{!}$ is equal to the logical negation but also takes $\perp$ as their neutral element.

**Definition 5** The extended ternary logical operators are defined as:

$$\langle r_1, t_1\rangle \text{AND} \langle r_2, t_2\rangle . \langle r_1 \text{ AND } r_2, \quad t_1 \textcircled{\&} t_2\rangle$$

$$\langle r_1, t_1\rangle \text{OR} \langle r_2, t_2\rangle . \langle r_1 \text{ OR } r_2, \quad t_1 \textcircled{|} t_2\rangle$$

$$\text{NOT}\langle r, t\rangle = \langle \textcircled{!}\ r, \ \textcircled{!}\ t\rangle$$

where AND, OR and NOT are the SPL ternary conjunction, disjunction and negation, respectively.

With the above definitions, it is not difficult to show that the new awareness operator "$" enjoys the distributed, commutative and associative properties over both the binary and ternary SPL logics. This means that annotating the complete policy with the awareness operator is equal to annotating each specific logical constraint. Note however that, both for policy privacy and efficiency reasons, the annotation of a full policy should be avoided.

We now show how to apply these rules to the policy in Fig. 9. The precise description of the policy is postponed to the next section. The tag-tree evaluation takes place from the last level to the first one. Therefore, the first step is to evaluate the elementary binary logical expressions, of the decide-expression (lines 21–24) using Definition 2. Assuming that





Journal: 13174 MS: 0065 TYPESET DISK LE CP Disp.:2012/10/6 Pages: 20 Layout: Large

all non-annotated binary expression evaluates to true for the current event their tags are all evaluated to tag=TRUE. On the other hand assuming that the annotated expression requires high accuracy while every `day.accuracy="Medium"`, then its tag is TAG = `ce.accuracy != "Medium"`. The next steps are the application of Definition 4 to calculate the tag of the conjunction of the elementary binary expressions that comprise the decide-expression of the rule. Then, Definition 3 is used to calculate the tag of the rule (lines 20–24) out of the tag of the decide-expression, which are both trivially equal to TAG = `ce.accuracy != "Medium"`. Finally, the tag of the policy is calculated applying Definition 5 to every rule disjunction resulting from the expansion of the `EXIST` quantifiers (lines 12–25). Assuming that the quantifiers groups have 2 and 5 elements, that is, two groups of friends and a condition for each working day, the tag of the policy is the disjunction of 10 equal tags, resulting in the deny reason of `ce.accuracy != "Medium"`, i.e., the expression is denied because the accuracy required is not equal to the `"Medium"`.

## 4 Implementation

Figure 7 provides a global view of Jano implementation. In the center, we can see the main modules (from left to right):

– The *Location reporting API* receives location information from the location generators (e.g., applications reporting GPS readings, RFID positioning systems) reporting the current position of each target.
– The *Location Manager* keeps the last location of each target, as received from the Location reporting API. Based on this information, it generates location events which are then stored in a first-in-first-out queue. Each event contains information about the target (e.g., *Alice*), the type of the event (*arrive* or *leave*) and the place to which the events refer (e.g., $P1$).
– The *Notification Distributor* receives events from the Location Manager and interacts with the Policy Enforcer (see next item) to know if there are users to be notified of a given event. As a consequence, the Policy Enforcer will evaluate each notification policy. If, for a certain notification request, a notification is needed, the Notification Distributor is responsible for doing so, using the previously configured communication channels for the user being notified.
– The *Policy Enforcer* evaluates access control and notification policies. Access control policies are evaluated for each pull request made through the Queries API (see next item) while notification policies are evaluated when a new location event is generated.
– The *Queries API* (for query and administration purposes) is used by other services or applications to get instant locations and configure the access control and notification policies.

On the left-hand side of Fig. 7, we can see a set of mobile devices (possibly attached to objects and/or persons) using different location technologies such as WiFi, GPS, etc. The right-hand side of Fig. 7 illustrates a Jano's user who issues location or notification requests through any computing device.

Thus, in summary, the Jano *programming interface* (i.e., API) supports two main services:

– the Queries API allows for location applications (web and rich clients) to: i) ask for a person or object location (pull requests), ii) manage access control location policies, and iii) manage notification policies;
– the Location reporting API is used by 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. A GPS and RFID

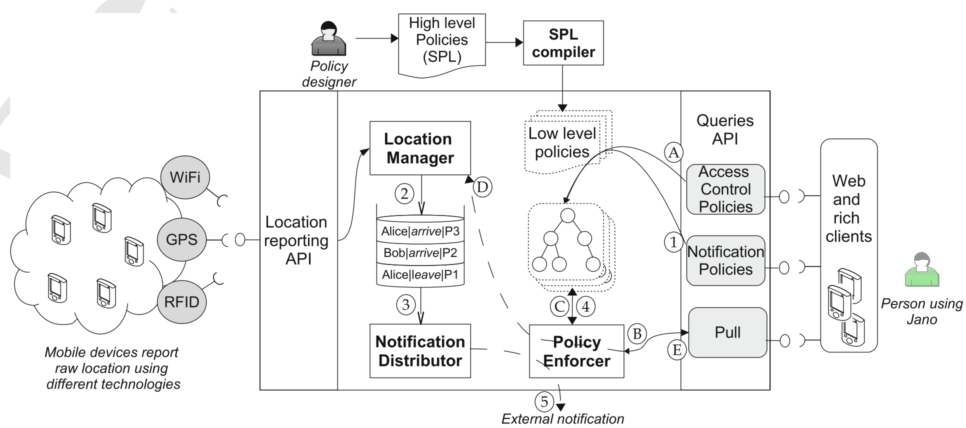



**Fig. 7** Jano implementation


Journal: 13174 MS: 0065 TYPESET DISK LE CP Disp.:2012/10/6 Pages: 20 Layout: Large

```
policy CommonPolicy {
accessControl: // Evaluate all access control policies of the target
  FORALL policy IN ce.target.accessControlPolicies
  { policy };
  TheOwner: ce.target = ce.author :: true; // Always allow the
                                            // request, if the author
                                            // is the target

  ?CommonPolicy: TheOwner OR accessControl;
}
```

**Fig. 8** Common policy of Jano. All policies of target are evaluated to decide if location can be disclosed

generator have been developed, both using the .NET platform and the C# language.

For demonstration purposes, we now 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 Fig. 8. This policy includes two rules:

- `TheOwner`: this rule is the same described in Sect. 3.1 which, as already mentioned, states that every target can know his location;
- `accessControl`: this rule enforces all the existing specific access control policies for the target being located (i.e., `ce.target.accessControlPolicies`) to be verified and enforced.

It is important that, policies for targets and places can be specified independently, which can result in conflicting rules. For example, *Alice* is not allowed to locate *Bob*, but *Alice* can kown 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 a certain location when the combination of target's and place's policies allows it.

The implementation of Jano supports several access control policies. In this article, we focus on one policy that could be applied to a variety of environments (e.g., university campus, enterprise building). The policy is presented in Fig. 9: when associated to a target (to be located), it defines the users who are allowed to know the target's location, with what precision and when.

The entry point of this policy is the rule `GroupsInterval` that, as stated previsouly, defines the circumstances under which a target's location can be disclosed. The target being located is the entity to whom the policy `GroupsInterval` is associated. The users requesting the target location are members of the group named `allowedGroup`. The `allowedGroupInfo` type contains the name of an `allowedGroup` along with the precision that the target location should be returned, and in what period of the week. Each `allowedGroup` is stored in the

```
type allowedGroupInfo { // type for GroupsInterval policy
    string allowedGroup; // users within this group can see location
    string precision; // maximum precision of disclosed location
    dayinterval set daysSet; // allowed intervals of the week
}

policy GroupsInterval {
  allowedGroupInfo set groupsInfo; // access control list of
                                   // allowed groups

  ?GroupsInterval: // rule for evaluating disclosure of location
    EXIST node IN groupsInfo { // for each node in
                               // the allowed list
      EXIST day IN node.daysSet { // for each day in the
                                  // allowed days, search
                                  // for the allowed group
                                  // name in the list of groups
                                  // whom the author of the
                                  // request belongs to
          node.allowedGroup IN ce.author.groups
          :: day.dayOfWeek = ce.dayOfWeek &
             (ce.time.hour >= day.start.hour &
              ce.time.hour <= day.end.hour) &
           $(ce.precision = day.precision)
      }
    }
}
```

**Fig. 9** Example of personal access control policy

policy instance, more precisely in the `groupsInfo` property.

This `GroupsInterval` policy is evaluated in two scenarios: i) following a *pull* for some target to be located, ii) when a location event is produced by Jano's core and a notification policy determines the potential location disclosure of some target.

As presented in Sect. 3.3, we have extended SPL with the *awareness operator* which allows the policy designer to identify, if relevant, the reason why the location cannot be disclosed (e.g., too much precision). In the policy presented in Fig. 9, when the author of a location request is denied access to a person's location, he will be informed about the precision that is demanded for the location to be disclosed. Consider a scenario where there are three levels of precision: "low", "medium", "high". If the requester wants "high" precision but the policy only allows "low", Jano returns the location with the allowed precision. Note that the original request will be denied but only because of incompatible precision. Therefore, the system can automatically return the location in accordance to the target policy.

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, i.e., to the `accessControlPolicies` or the `notificationPolicies` properties in Fig. 2, respectively.

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. In Sect. 4.5, we address Jano's web interface to support the configuration of policies.





Journal: 13174 MS: 0065 TYPESET DISK LE CP Disp.:2012/10/6 Pages: 20 Layout: Large

```
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
          }
}
```

**Fig. 10** Notification policies. `SNotify` is a simple parametrized policy. `VisitAfter` takes into account past location events to determine if a notification must be sent

### 4.1 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 notification policy (`SNotify`), that can be used as a template for policies, parameterized by the name of an object (`id`), the name of a place (`place`) and the location event of interest (`evType`). This policy could be used by *Alice* to be notified by Jano when *Bob* arrives at `inesc/floor6`. If so, a policy with the given parameters would be instantiated as follows:

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

When the Notification Distributor (see Fig. 7) receives a location event stating that *Bob* has arrived at `inesc/floor6`, it 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 notification policy determines that a notification should be sent and the access control policy of the moving target allows it, a communication channel (e.g., web service, e-mail, etc.), previously configured by the user, will be used to send the notification.

Jano can efficiently enforce notification policies with history-based rules. This is used, for example, when a user wants to be notified about an object trajectory inside his organization: a previously ordered book can arrive at the reception, but this event is only interesting if that same book has already passed 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("ist/library"),
new place("inesc/reception"));
```

### 4.2 Policy dynamism and log-size management

A monitor-like security service (as is the one implemented by Jano) has to decide, for each request, whether it should be allowed or denied. The decision must be taken at the time of the request with the information available. Thus, in order to implement history-based policies, any monitor-like security service has to store information about past requests and events.

Some security services store requests explicitly into a request log [4, 16] that can later be queried for specific requests; others, store them implicitly in their own data structures. For example, Sandhu [27] proposes the use of dynamic clearance levels, associated to each user, where the information about the classification of the information read is stored, and may be further used to decide if a user with a specific clearance level is allowed to access information with the specified classification.

The former solution is more flexible than the latter. However, if the request log becomes too big, the memory space required to keep that log may become unlimited, and the time required to execute each query could have a significant impact on the overall performance of the system. Jajodia [16] tries to solve this problem recording the requests that differ in time only once. However, this does not solve the problem because the number of requests to store is still huge and disallows the definition of policies based on request cardinality to be enforced (e.g., the user may only be localized by someone else three times in a row).

SPL implements the log solution through a compilation algorithm that optimizes the amount of information to be saved and the way that information should be queried. Although the algorithm does not obtain optimal results for all history-based policies, the results obtained for most frequent policies are equivalent to those obtained by label-based implementations [27].

The algorithm has three main aspects. First, the Policy Enforcer (shown in Fig. 7) selectively logs just the requests required by the concerned history-based policy; e.g., if a policy needs to know if a document was signed, there is no need to record requests that are not "sign requests". Second, the Policy Enforcer selectively logs just the fields of the requests required by the specified history policies, e.g. if a policy decision is based on whether or not the author of the current request has signed a document, it is not necessary to record the "time" or the "task" fields of signature requests. Third, the





Journal: 13174 MS: 0065 TYPESET DISK LE CP Disp.:2012/10/6 Pages: 20 Layout: Large

Policy Enforcer uses the best possible structure to maintain the log and the best type of query to search it.

Thus, the log is searched by entries with specific properties. These properties might be expressed using equality constraints, inequality constraints or membership constraints. Equality constraints can be searched in a hash table in $O(1)$, which makes them ideal to be used as index keys. However, if there is not a single equality constraint to look for, it is better to use a balanced tree to hold the log and use a different type of query.

Thus, with this solution, instead of building a single log for all history-based policies, the compiler in Jano builds a specific and fined tuned log for each history-based policy. This solution has several advantages. First, it reduces the number of requests required to be searched. Second, it allows for a better adaptation of the base structure to each query, because each log can be kept by a different structure. Third, it simplifies the insertion and the removal of policies.

The problem with this solution is the potential for maintaining redundant information in several logs. However, given that the information kept by each log is the minimum information necessary for the corresponding policy, the level of redundancy expected is similar to the one of label-based implementations, where the labels used by different policies may also be redundant. Nevertheless, this negative aspect can be further limited through the sharing of logs with the same signature (same requests to log, same fields in those requests to log, same base structure) between policies.

Given that, traditionally, each policy applies to a very limited number of users and places (see for instance ACL-based policies), and that the domain of event properties is usually limited (e.g., the localization may be all rooms in campus), the size of each policy log is not usually large. However, there are some types of policies that must be avoided. For instance, a policy that requires the logging of the time at which each past event took place should be avoided in favor of some alternative one (e.g., logging the relative order between a sequence of events), because it could potentially be very inefficient. Still, from our experience with SPL, most of these situations may be avoided, and often automatically detected, by the SPL compiler. The next section describes in detail the process used in Jano to minimize the log size.

Finally, the main drawback of the proposed solution is that history-based policies cannot decide on requests prior to their activation, i.e., the system only records requests for each history-based policy after the policy starts to exist. However, based on our and others' experience, we believe this is not a serious drawback.

### 4.2.1 Log size reduction algorithm

The process used to reduce the log size is comprised by three main algorithms: the compilation algorithm

```
CompileExistAtMost(rule) {
        Apply_e ←DomainExpression(rule)
        Elementar_e ←ExtractElementaryExpressions(Apply_e)
        Pe_ind ←Forall z in elementar_e that Independent(z, pe)
        Pe_dep ←Forall z in elementar_e that
               Independent(z, ce) and Dependent(z, pe)
        Ce_dep ←ReplacePeByCe(Pe_dep)
        Cpe_dep ←Forall z in Elementar_e that
               Dependent(z, ce) and Dependent(z, pe)
        First_e ←FindOne z in Cpe_dep that HaveEquality(z)
        Next_e ←Forall z in Cpe_dep that
               Conjuntion(z,First_e) and HaveEquality(z)
        Find_e ←First_e ∪Next_e

        LE_register ←Recombine(Ce_dep)
        LE_apply ←pe ≠NULL & pe._count ≤maxEvents &
               Recombine(Cpe_dep / First_e) & Recombine(Pe_ind)
        LE_decide ←DecideExpression(rule)
        TE_find ←Forall z in Find_e take PeFields(z)
        TE_register ←Forall z in Cpe_dep take PeFields(z)
}
```

**Fig. 11** Simplified compilation algorithm for history based rule based with a EXIST AT_MOST construction

```
LE_apply(ce) ←ce.action = "Get_Location"
LE_apply(ce,pe) ←pe ≠NULL & pe._count ≤
maxEvents & ce.action = "Get_Location"
LE_decide(ce,pe) ←true
TP_register(ce) ←{author(ce), target(ce), date(ce)}
TP_find(ce) ←{author(ce), target(ce), date(ce)}
```

**Fig. 12** Compilation result for the TrackingLimit policy (shown in Fig. 6)

(Fig. 11), and the register and decide algorithms (Fig. 13). The compilation algorithm takes the history based rule and builds three logical expressions (LE_apply(ce,pe), LE_register(ce), LE_decide(ce,pe)) and two tuple extraction functions (TP_find(ce) and TP_register(ce)). The result of the compilation applied to the TrackingLimit policy (Fig. 6) is shown in Fig. 12.

For clarity, the algorithms are presented in simplified pseudo-code. The actual implementation takes a slightly different approach to take in consideration all the different cases. For more details see [25,24].

The algorithm starts by extracting all elementary expressions out of the domain-expression in the policy (e.g., ce.action, = "Get_Location" in the TrackingLimit policy) which are composed using binary conjunctions and disjunctions. Then, it chooses those expressions which are independent from the past events (Pe_ind), the ones that are dependent of past events but independent from the current event (Pe_dep), and the ones that depend on both the current event and on the past events (Cpe_dep). From these last ones, it builds the set of expressions that are connected through equality constraints and are related to each other through conjunctions (Find_e). Each of these sets of expressions is then used to build the four logical expressions and two tuple extractors (Fig. 12).

These five functions are then used in the `RegisterExistAtMost` and `DecideExistAtMost` functions (Fig. 13). The first one is called for every event and decides





Journal: 13174 MS: 0065 TYPESET DISK LE CP Disp.:2012/10/6 Pages: 20 Layout: Large

```
RegisterExistAtMost(ce) {
        if (allowed(ce) & LE_register(ce)) then
                tuple ←TE_find(ce)
        pe ←find(tuple,LOG)
        if (pe ≠NULL) then
                pe._count ++
        else
                event ←TE_register(ce)
        add(event, LOG)
}

rule DecideExistAtMost(ce) {
        tuple ←TE_find(ce)
        pe ←find(tuple, LOG)
        return LE_apply(ce,pe) :: LE_decide(ce,pe)
}
```

**Fig. 13** Register and decide functions for EXIST AT_MOST construction

which events get to be logged to the specific LOG of the policy. If there is an identical tuple registered in the log, the counter with the number of occurrences of that event is incremented; otherwise, the event is logged. The `DecideExistAtMost` enforces the policy. It returns a simple tri-value rule, built upon the four logical expressions generated by the compiler. The resulting rule will be evaluated together with the other rules not dependent on history.

### 4.3 Optimizing policy design, processing, networking

In vast organizations or deployment scenarios, the number of entities (e.g., members of the organization, locations) tends to be very large, with the consequent increase in the number of policies required to enforce overall location-privacy settings. Thus, the task of policy definition may become too heavy for a handful of administrators. Also, the total number of events will also increase (entering, leaving locations). This imposes further load on policy processing and increases network traffic, even when events are irrelevant for the active policies. To address these issues, in Jano, we include the following additional mechanisms: Policy Inheritance, Hierarchical Policy Applicability and Local Filtering.

With Policy Inheritance, besides policies being parameterized in Jano, a policy can also be defined as an extension and/or composition of other policies, to foster reuse of rules (that are more intricate to develop or code), with the possibility of overriding rules. The resulting policy is validated during compilation.

Hierarchical Policy Applicability in Jano, further simplifies policy development, by allowing the rules of policies to refer to entities according to a hierarchical namespace. Thus, whenever a policy is applicable to, e.g., a specific building, department, role/category, it will automatically be applicable to all its subelements, e.g., rooms in the building, people of the department, sub-roles or categories. This can be regarded as a form of inheritance across the entity space (encompassing people, places, roles), instead of the rule space above.

Events, e.g. regarding entering and leaving locations, when appropriate, can be subject to Local Filtering, i.e., not sent to the Location Manager (see Fig. 7) by the location generators (e.g., RFID tag readers). This lowers the load of policy processing, and saves bandwidth. This action can only be taken when it is known that the person or location (or both) are not relevant for the currently active policies.

The Location Manager also stores a policy digest that states, in summarized form, the entities (and entity namespaces) that are mentioned in all active policies (an entity or set appears only once in the digest regardless of the number of occurrences in policies); for that purpose, Jano uses a bloom filter [6] storing hashes of strings (entity names or namespaces).

Thus, on each event reaching the server (i.e., not filtered by the mobile clients), the Location Manager checks a global bloom filter for each entity mentioned in the event, to know whether it is referred to in any policy (or any of its high-level namespaces). This allows filtering out the events that are not related to any policy. It also ensures that, while any filtering done at mobile devices is useful (to save their bandwidth), it does not render Jano dependent on the cooperation of mobile devices, in order to reduce the load of event processing at the server. Note that, the low rate of false positives does not hinder correctness.

Events surviving the filtering are then checked for every notification policy, but only its domain of applicability (that will rule out most of them), and not the entire policy evaluation. This imposes less overhead than maintaining, for each individual policy, an additional dedicated bloom filter (implying the calculation of several hashing functions), as applicability conditions are usually a simple test that accounts for only a fraction of the overall policy evaluation processing. Whenever the coverage of entities involved in policies is enlarged (due to loading of a new policy), an updated digest is sent to the Location Manager.

### 4.4 Interaction between Jano and SPL

SPL is composed by a language, a compiler and a library. The compiler parses the policy definition files and generates Java classes with the evaluation of the policy, including the data structures used to keep track of location requests (i.e., history log). Figure 14 shows the interaction between Jano and the SPL-generated modules; the location access control or notification policies (on top) illustrate the Java code produced by the SPL compiler.

Policies are instantiated by means of a pull or a push operation invoked on the Queries API (as shown in Fig. 7) resulting in the instantiation of either: i) a location policy as the one presented in Fig. 9, or ii) a notification policy as the one presented in Fig. 10.





Journal: 13174 MS: 0065 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2012/10/6 Pages: 20 Layout: Large

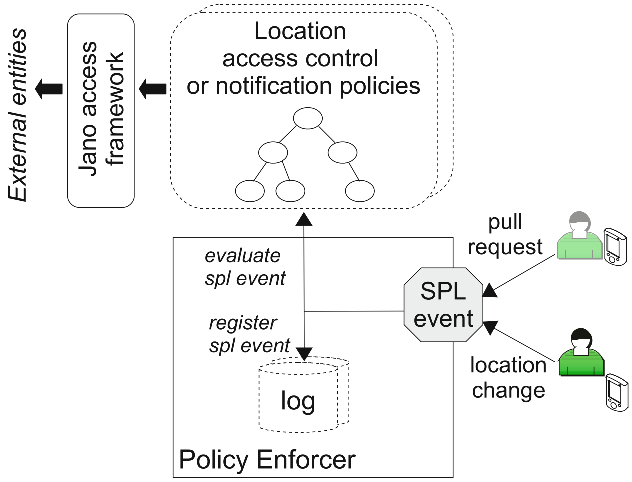



**Fig. 14** Location requests and events are evaluated by policies. To this end, the policy enforcer accesses the domain model through a the Jano access framework

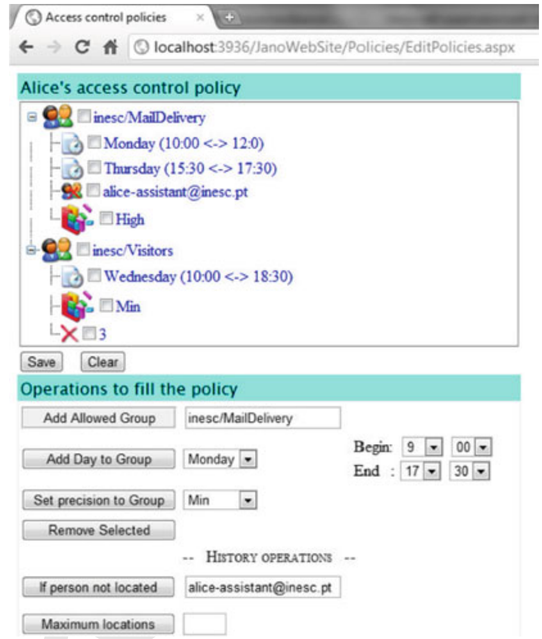



**Fig. 15** Web-based GUI to configure Jano's policies

Policies are used to decide about a pull request or location event. During the evaluation of a pull request or location event, the Policy Enforcer needs to obtain information about the target of the event. Because SPL is designed to be agnostic with regards to the enforcement site's information system, SPL policies rely on a bridge framework, called *Jano access framework* (top left-hand of Fig. 14) to access the relevant information. This framework interacts with the implementation of the SPL *external* entities presented in Sect. 3.1 which, in our system, represents Jano's locatable objects (i.e., persons, mobile objects and places).

### 4.5 Web-based GUI

In order to enhance the usability of Jano, a web-based GUI runs on top of the Jano API, using the ASP.NET platform. Using this GUI, a non-SPL expert user can make not only location requests but also select and provide the necessary parameters for his access control and notification policies.

Figure 15 shows the GUI, during the configuration phase of the access control policy illustrated in Fig. 9 enriched with some history-events described afterwards. The user, in this case Alice (`alice@inesc.pt`), configures the above mentioned policy with just a few "*clicks*".

The top part of the GUI shown in Fig. 15 (named `Alice's Access Control Policy`) allows Alice to indicate, for two groups of users (`inesc/MailDeliveryandinesc/Visitors`), the circumstances under which they are allowed to know Alice's location. More precisely, Alice specifies the acceptability of a location request or location event (regarding her location) as follows:

- a user who is member of the group `inesc/MailDelivery` is allowed to know Alice's location if, and only if, the current day is Monday or Thursday (on the time slots indicated) and he has not succeeded previously on locating Alice's assistant (`alice-assistant@inesc.pt`); in addition, the precision allowed for Alice's location is `High`.
- a user who is member of the group `inesc/Visitors` is allowed to know Alice's location if, and only if, the current day is Wednesday (on the time slots indicated) and the location requests so far performed have not exceeded three reports; in addition, the precision allowed for Alice's location is `Min`.

The bottom part of the GUI allows Alice to add or remove new user groups, date intervals, and parameterize the history-based rules, mentioned above.

In conclusion, the GUI developed in Jano supports a large number of operations so that most policy specifications can be easily done without knowing SPL.

## 5 Evaluation

In this section, we present the evaluation of Jano. There are two types of evaluation: quantitative, by means of performance tests, and qualitative, by means of a use case with two applications.

With respect to the quantitative results, we evaluated the most important performance aspects regarding the system



Journal: **13174** MS: **0065** ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:**2012/10/6** Pages: **20** Layout: **Large**

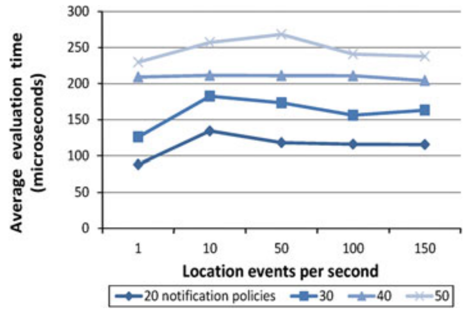



**Fig. 16** Growing number of groups of requesting author

behavior, with an increasing number of: users issuing location requests, history events and location events, and load of concurrent requests. The results that were obtained can be seen as a worst-case scenario, measuring the most performance demanding modules and operations. While the conditions tested are more demanding than typical usage scenarios, Jano's manages to operate within small response times.

Regarding the use case and applications, these were chosen as they provide two usage scenarios that illustrate real user needs in terms of location privacy policies, and are related to other scenarios described in the literature [29].

### 5.1 Performance evaluation

In this section, we report the performance of two crucial interactions between users and Jano. First, we present the results obtained in the evaluation of the Policy Enforcer, while enforcing an access control policy, with and without history-based rules. We use this type of policy because it must always be evaluated, even before a *push* response. Second, we present the performance of the system in a scenario in which there are several location events generated per second (similar to the use case presented in Sect. 5.2), in order to test Jano under load conditions.

Overall, Jano performs adequately in every scenario tested, including under load. In particular, Jano's performance under scenarios with increasing load (number of policies, and of concurrent requests and events), is always within the constraints required for interactivity and perceptual tasks [5,8] (namely, replies under 50 ms, the usually referred latency for good interactivity, e.g. in multi-player games or cooperative work).

*Access control policies* The policy considered for evaluation of the Policy Enforcer is the one presented in Fig. 9; the results are shown in Fig. 16. In the referred policy, the set of groups allowed by the target (`GroupsInfo`) must be traversed linearly. Recall from Fig. 9: for each member of the set, there is the indication of in which intervals the target can be located. Each of these allowed groups are looked up in the groups to which the author of the request belongs to (designated hereafter $AuthorGroups$). The cost of this search is $O(log(AuthorGroups))$ (search on the ordered set).

Experiments where done with different numbers of groups allowed and groups to which the author (the user making the location request) belongs to. In our test scenarios, we considered that $AuthorGroups$ tend to be much bigger than $GroupsInfo$, as can be seen in the four series of Fig. 16. In the more demanding scenario, the series with 100 allowed groups (a number that would stress the capacity of the owner of the policy to manage it), and taking into account that the author of the request belongs to 3,000 groups, the considered policy takes nearly 2.5 ms to be evaluated.

Figure 17 presents an analysis of the percentage slowdown, in performance, in the presence of load conditions. In each subfigure, we evaluate the percentage slowdown, regarding an increasing number of concurrent location requests (i.e., 10, 100, 1,000), for each of the sets of groups allowed by the target (i.e., 1, 25, 50 and 100), against the results previously presented in Fig. 16. Although response times increase when the load increases, which is expected, the results show that the slowdown does not grow linearly, as it remains below 100 % (and most often around 40–60 %), when the load of concurrent requests has been raised up to 1,000-fold. Moreover, the obtained slowdowns result in average response times always below 5 ms, which is still very low.

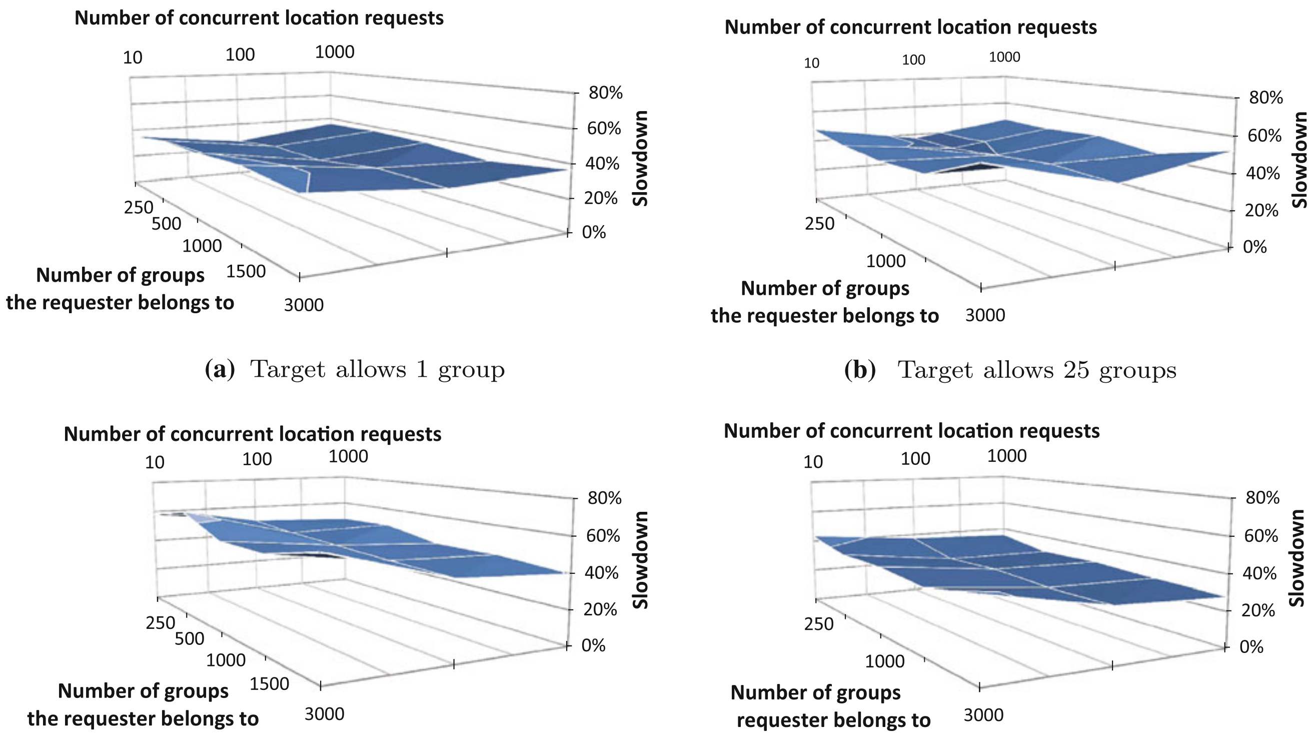
A critical aspect in the evaluation of the Policy Enforcer is the measurement of the delay introduced by the evaluation of history-based policies. As explained in Sect. 4.2.1, the SPL compiler produces specific data structures to store the events needed in the evaluation of history-based policies, such that it minimizes the time to evaluate history-based policies.

Figure 18 shows the delay introduced by the evaluation of a policy based on the history rule presented in Fig. 6. Tests were made using the (optimized) log of SPL and a non-optimized log. With the SPL log, if a user makes 1000 location requests to 20 different targets, only 20 events will be effectively stored, instead of the 1,000 of a non-optimized implementation. 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 to the non optimized log, where all events are recorded regardless of their redundancy. Therefore, evaluating history-based policies with this log takes much less time because there are orders of magnitude fewer entries in the log to be evaluated, when compared with the non-optimized log. Even so, the evaluation time will eventually grow but with a sub-linear progression and dependent only on how





Journal: 13174 MS: 0065 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2012/10/6 Pages: 20 Layout: Large




**(a)** Target allows 1 group **(b)** Target allows 25 groups

**(c)** Target allows 50 groups **(d)** Target allows 100 groups

**Fig. 17** Percentage slowdown of average evaluation time for policy `GroupsInterval` with an increasing number of concurrent location requests

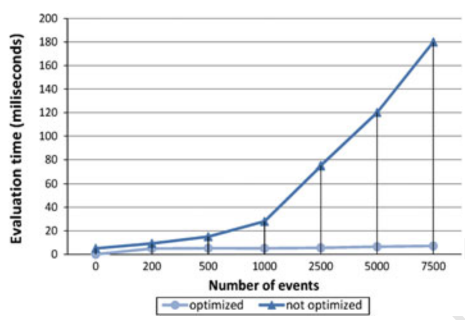



**Fig. 18** Evaluation time for growing number of history events

new events are compressible or not, due to their previous occurrence.

*Multiple location events* In Fig. 19, we show the results obtained when evaluating the response time of Jano in a scenario where multiple targets (persons or objects) are moving. As a consequence of these movements, several location events will be generated (e.g., leaving or arriving at some place). For this scenario, we consider a single user who has between 20 and 50 notification policies (a rather high number in reality, as other literature often considers only five [29]). Such policies are instantiations of the `SNotify` policy, previously presented in Fig. 10.

The results presented in Fig. 19a show the evaluation time (in logarithmic scale) of the notification policies. We can see that as more notification policies need to be enforced, the average evaluation time does increase, although roughly in a linear fashion with the number of polices. We recall that 50 active policies is nonetheless a rather large number (compared to others found in literature [29]), representing up to 50,000 concurrent policy evaluations, and that even then, all times are below 50 ms.

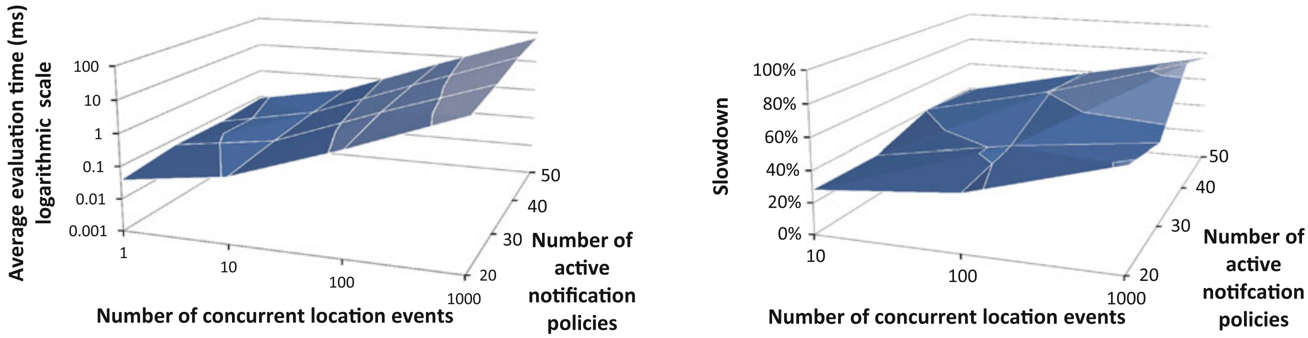
Jano's scalability regarding evaluation of notification policies is further illustrated in Fig. 19b, detailing the percentage slowdown of notification policy evaluation as the number of concurrent events increases tenfold, each time, from 10 up to 1,000, against serial execution. The slowdown observed is never smaller than 25 % but is always under 100 %, being most of the time between 40 and 80 %. This shows that when the load is increased by a factor of 1,000, policy evaluation times do not even double, which demonstrates the scalability of Jano notification policies evaluation under load. Once again, these slowdowns represent absolute times under 50 ms, within the latency constrains for interactivity and perceptual tasks [5,8].

More important, these results illustrate the performance of Jano when events have to be processed against at least one of the notification policies. This does not take into account





Journal: 13174 MS: 0065 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2012/10/6 Pages: 20 Layout: Large




(a) (b)

**Fig. 19** **a** Average evaluation time of notification policies with an increasing number of concurrent location events. **b** Percentage slowdown of average evaluation time of notification policies with increasing load (number of concurrent location events) and number of active notification policies



the effect of the filtering mechanism, based on using bloom filters (described in Sect. 4.3), that minimizes the number of events to be considered. As already mentioned, this mechanism ensures that only those events needing to be further processed (because they are relevant for active policies) do have to be considered.

In a conservative scenario, we consider that this filtering mechanism achieves an average of 40 % reduction in the number of events to be processed. Thus, the results shown in Fig. 19, when considering the filtering mechanism, are in fact valid for scenarios up to 2,500 concurrent location events (i.e., from these 2,500, only 60 % do correspond to active policies and have to be effectively processed).

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

### 5.2 RFID use case

The qualitative evaluation of Jano was done by implementing several location applications using different location technologies (e.g., wifi, GPS, RFID) demonstrating Jano's capability to deal with location technology heterogeneity. One of the most representative use cases is described in this section along with two applications in which a wide set of access control and notification policies were used.

We have implemented an RFID location based system in the Jano architecture. In addition, we developed two prototype applications: i) campusLocation allows a student/professor not only to find out his location, but also to obtain information regarding how to reach a particular place and where other colleagues are; ii) transportLocation allows users, at each bus stop, to receive a wide range of information regarding their own location as well as the buses that may be used. Similar applications are also considered in [29] and their relevance evaluated with user questionnaires.

Figure 20 provides a view of the several main aspects regarding the campusLocation application. At each relevant physical place in the campus, there is a fixed RFID tag (FT). Also, at some particular places (such as room entrances) there are fixed RFID readers (FR). Each user (students and professors) holds a mobile phone with RFID reading capabilities that also has an RFID tag.

A user can explicitly read a FT and send via Wi-Fi the corresponding identification to the Jano server. Thus, Jano knows where a user is as long as he decides to read a FT and send its identification. In addition, the FRs previously mentioned are also capable to automatically read the RFID tags on the user's mobile phone (when a mobile phone is close to the reader, obviously). These two mechanisms allow the Jano server to know the students'/professors' location. Figure 21 illustrates the interface of the campusLocation application (e.g., finding a way to a particular place in the campus).

Based on the users' locations, it is possible to offer a set of location-based added-value services which do raise privacy concerns (as stated in Sect. 1). For example, professor Alice may be notified when, his colleague, Bob enters in a particular room or arrives at the campus. On the other hand, Bob does not want students to know his location when he has no teaching duties. To ensure this, Bob simply configures the corresponding policy, regarding the disclosure of his location, using the Jano web-based GUI: allowing Alice to be notified in the circumstances indicated above, and allowing students to know where he is located only when he has teaching duties (e.g., from Monday to Thursday from 14 to 19 hours). These policies are similar to those described in Sect. 4, and in line with those considered as complex in the literature [29].

In the particular case of IST (the engineering school of the Technical University of Lisbon), there are approximately 900 professors organized in 9 departments.



Journal: 13174 MS: 0065 TYPESET DISK LE CP Disp.:2012/10/6 Pages: 20 Layout: Large

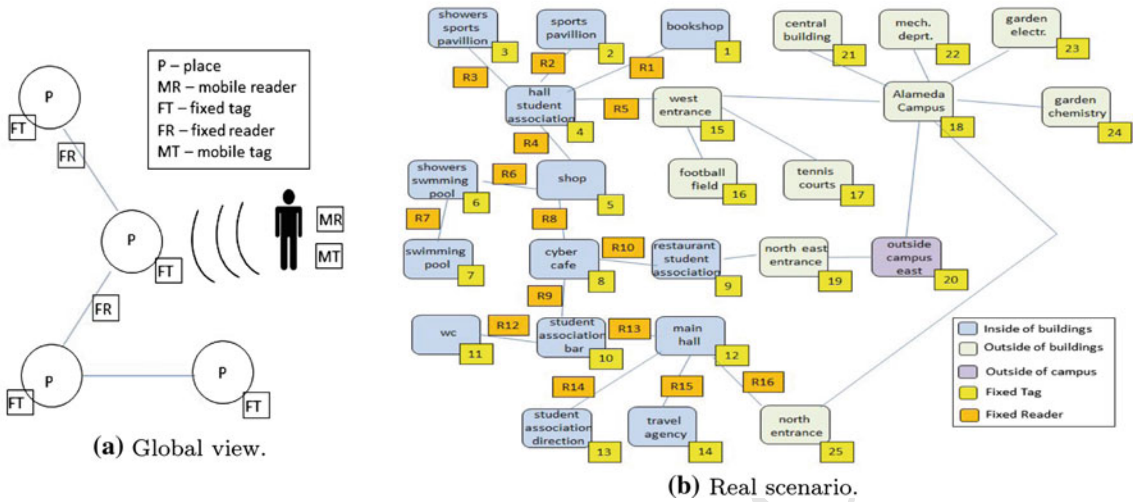



**Fig. 20** Application campusLocation

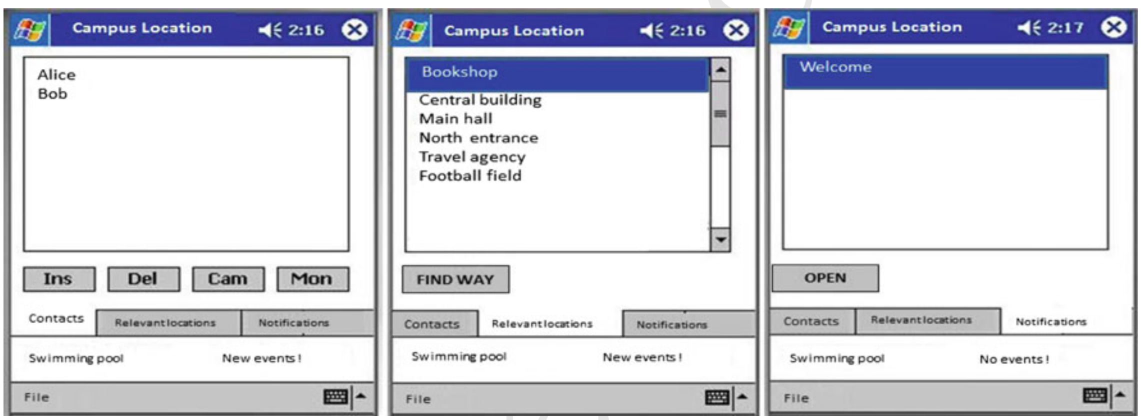



**Fig. 21** Interface of campusLocation application

Therefore, the number of groups they belong to, for policy specification purposes, is in most cases five (department, scientific area, research institute and research group). In some cases, a single user belongs to more groups depending on the management duties. However, the number of groups found in this case is much smaller than those that were used for the performance tests previsouly described.

Regarding the transportLocation application, on each bus stop there is both a fixed RFID tag (FT) and an RFID fixed reader (FR). As in the campusLocation application, each user holds a mobile phone with RFID reading capabilities that also has an RFID tag. In addition, each bus has an RFID tag (called bus tag) that is read by the FR at each bus stop; this reader sends the identification of the bus tag to the Jano server so that the location of all buses is known in real time.

A typical usage scenario ("findWay") is the following. At the bus stop, using his mobile phone, the user reads the fixed tag FT which uniquely identifies the location. Then, the user contacts the Jano server (using GPRS) sending it the fixed tag identification (thus, allowing the system to know where the user is) along with the desired final destination; the system then replies with the most appropriate bus the user should take. Regarding location information privacy in this application, Jano suports the following (among others): a bus driver may access all buses locations, contrary to bus clients. This requires the specification of distinct location policies, accordingly. Once again, such policies can be easily defined using the web-based GUI interface previously presented (see Sect. 4.5).

In conclusion, these two applications require a careful set of location policies (both access control and notification) to ensure privacy. For this purpose, Jano provides the adequate features: only in some application specific circumstances are users (e.g., students, professors, bus drivers, etc.) allowed to know the targets location.



Journal: 13174 MS: 0065 TYPESET DISK LE CP Disp.:2012/10/6 Pages: 20 Layout: Large

## 6 Related work

The interest in location privacy has been growing with more services being able to take advantage of persons' and objects' locations. This is possible because of the universality of location technologies and their integration with every day devices. Hightower et al. [14] provide a systematic review of location technologies. In their work, these technologies are divided in proximity sensing, triangulation and scene analysis. Examples of proximity sensing include the radio frequency identification technology. Triangulation is the basis of the Global Position System (GPS) and scene analysis has been used to take advantage of Wi-Fi infrastructures [2,32].

In [20], Minch points out that location privacy can be defined in terms of: Intrusiveness, Seclusion, Boundaries, Control, and Limitation. In our work, users are willing to share their location to third parties and so we focus on control and limitation in the disclosure of location privacy. Different approaches have been considered to promote privacy when disclosing and sharing personal location and, mainly, three lines of research can be identified: one that takes the person's location and blurs it [1], one that anonymizes users [3,21,12] making them indistinguishable and finally, one that takes into account security policies defined by the users of the system [19,22,23].

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 an 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 policies has been a topic of research for some time [19,13,18]. Security policies of persons, objects or places, can be made dependent on several location privacy primitives (geographical area, time interval, historical access, etc.) [3,30]. Each of these aspects can be combined to form a user, object or place, security policy.

Leonhardt and Magee [19] 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 [11] incorporates a location module which, besides being able to handle multiple sources of positioning information, is also structured to protect access to people's location [13]. Their option was to use the SPKI/SDSI infrastructure, giving the possibility, among other things, to delegate location access rights.

LocServ [22] represents each person's policy by a group of validators responsible for the evaluation of each location request. The implementation of these validators can range from a software that interrogates the user for each request, to a generic decision function based on, for example, a security policy file. Context Fabric [15] 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 has to attend to, e.g. the requester of the information cannot be at a given building.

More recently, Opyrchal [23] focuses on adding support to location privacy in a content-based publish subscribe middleware. Their system allows publishers (i.e., mobile 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 [10]. People Finder [17] takes a different direction, applying techniques of machine learning to automatically adjust each user's policy, based on their satisfaction with 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 [21] 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 cloak 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 [9]: 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 [23] recognize the need to support history-based policies, but their work is unable to do so. In [29], a study is presented indicating that users tend to develop more elaborated policies as they continue to use a location service. In the same paper, a social location service is presented, as in our example, integrating a rule editor. Nevertheless the authors do not show how history-based rules can be used and how the system could be adapted to other contexts besides the social network. Performance evaluation of the component used to evaluate policies is not mentioned, with exception of [23], where the authors conclude they need a more efficient policy evaluator. The adaptability of the policies to different organizations where





Journal: 13174 MS: 0065 TYPESET DISK LE CP Disp.:2012/10/6 Pages: 20 Layout: Large

users, objects and places have different characterization is also not the main issue.

## 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: advertisements on mobile devices from the shop being visited, and presentation of more information related to the product being purchased, 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 such as, 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 privacy-related security policies. Although the instant reporting of locations (*pull* requests) is essential, in many situations, 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 an extension of SPL, a multi-model authorization platform. Using SPL, policies can be implemented using a variety of different security models, and their deployment can be made dependent on the resources of the organization site. That is, the location policies are tailored to the domain model where the location service is to be deployed. Regarding evaluation, results 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.

**Acknowledgments** This work was partially supported by national funds through FCT-Fundação para a Ciência e a Tecnologia, under projects PTDC/EIA-EIA/102250/2008, PTDC/EIA-EIA/113993/2009, and PEst-OE/EEI/LA0021/2011.

## References

1. Ardagna CA, Cremonini M, Damiani E, De Capitani di Vimercati S, Samarati P (2007) Location privacy protection through obfuscation-based techniques. In: Lecture notes in computer science, vol 4602. Springer, Berlin, pp 47–60
2. Bahl P, Padmanabhan VN (2000) Radar: an in-building rf-based user location and tracking system. In: INFOCOM 2000. Nineteenth annual joint conference of the IEEE Computer and Communications Societies. Proceedings, vol 2. IEEE, pp 775–784
3. Beresford AR (2005) Location privacy in ubiquitous computing. Technical Report 612, University of Cambridge
4. Bevier WR, Young WD (1997) A constraint language for Adage. Inc, Technical report, Computational Logic
5. Bhola SK, Banavar GS, Ahamad M (1998) Responsiveness and consistency tradeoffs in interactive groupware
6. Bloom BH (1970) Space/time trade-offs in hash coding with allowable errors. Commun ACM 13:422–426
7. Burton RM, DeSanctis R, Obel B (2006) Organizational design: a step-by-step approach. Cambridge University Press, Cambridge
8. Cheshire S (1996) Latency and the quest for interactivity. In: White paper commissioned by Volpe Welty Asset Management, LLC., for the synchronous person-to-person interactive computing environments meeting
9. Cornelius C, Kapadia A, Kotz D, Peebles D, Shin M, Triandopoulos N (2008) AnonySense: privacy-aware people-centric sensing. In: Proceeding of the 6th international conference on mobile systems, applications, and services. ACM, New York, pp 211–224
10. Blaze et al M (1999) Rfc 2704: the keynote trust-management system version 2
11. Garlan D, Siewiorek DP, Smailagic A, Steenkiste P (2002) Project aura: toward distraction-free pervasive computing. PERVASIVE Computing, pp 22–31
12. Gedik B, Liu L (2008) Protecting location privacy with personalized k-anonymity: architecture and algorithms. IEEE Trans Mobile Comput 7:1–18
13. Hengartner U, Steenkiste P (2003) Protecting access to people location information. In: First international conference on security in pervasive computing, pp 25–38
14. Hightower J, Borriello G (2001) Location systems for ubiquitous computing. IEEE Comput 34(8):57–66
15. Hong JI (2004) An architecture for privacy-sensitive ubiquitous computing. In: MobiSYS 04: Proceedings of the 2nd international conference on mobile systems, applications, and services, pp 177–189. ACM Press, New York
16. Jajodia S, Samarati P, Sapino ML, Subramanian VS (June 2001) Flexible support for multiple access control policies. ACM Trans Database Syst 26(2):214–260
17. Kelley PG, Drielsma PH, Sadeh N, Cranor LF. User-controllable learning of security and privacy policies. In: Proceedings of the 1st ACM workshop on workshop on AISec, AISec '08, New York, NY, USA. ACM, New York, pp 11–18
18. Langheinrich M (2002) A privacy awareness system for ubiquitous computing environments. In: UbiComp '02: Proceedings of the 4th international conference on Ubiquitous Computing, London, UK, 2002. Springer, Berlin, pp 237–245
19. Leonhardt U, Magee J (1998) Stability considerations for a distributed location service. J Netw Syst Manage 6(1)
20. Minch R (2011) Issues in the development of location privacy theory. In: Proceedings of the 2011 44th Hawaii international conference on system sciences, HICSS '11, Washington, DC, USA. IEEE Computer Society, pp 1–10
21. Mokbel MF, Chow CY, Aref WG (2006) The new Casper: query processing for location services without compromising privacy. In: Proceedings of the 32nd international conference on very large data bases. VLDB Endowment, p 774
22. Myles G, Friday A, Davies N (2003) Preserving privacy in environments with location-based applications. Pervasive computing, pp 56–64
23. Lukasz O, Atul P, Amit A (2007) Supporting privacy policies in a publish-subscribe substrate for pervasive environments. J Netw 2:17–26
24. Ribeiro C (2002) Uma Plataforma Para Politicas de Autorizacao Para Organizacoes Complexas. PhD thesis, Instituto Superior Tecnico, Lisbon, Portugal
25. Ribeiro C, Zuquete A, Ferreira P, Guedes P (2001) Spl: an access control language for security policies and complex constraints. In: NDSS, The Internet Society





Journal: **13174** MS: **0065** ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:**2012/10/6** Pages: **20** Layout: **Large**

26. Samarati P, De Capitani di Vimercati S (2001) Access control: policies, models, and mechanisms. In: Revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design: tutorial lectures, FOSAD '00, London, UK. Springer, Berlin, pp 137–196
27. Sandhu R (1993) Lattice-based access control models. IEEE Comput 26(11):9–19
28. Stiemerling O, Wulf V (2000) Beyond "yes or no"—extending access control in groupware with awareness and negotiation. Group Decis Negot 9:221–235. doi:10.1023/A:1008787208430
29. Toch E, Cranshaw J, Drielsma PH, Tsai JY, Kelley PG, Springfield J, Cranor L, Hong J, Sadeh N (2010) Empirical models of privacy in location sharing. In: Proceedings of the 12th ACM international conference on ubiquitous computing, Ubicomp '10, New York, NY, USA. ACM, New York, pp 129–138
30. Tsai JY, Kelley PG, Cranor LF, Sadeh N (2009) Location-sharing technologies: privacy risks and controls. In: Research conference on communication, information and internet policy (TPRC)
31. Varshney U (2003) Location management for mobile commerce applications in wireless internet environment. ACM Trans Interet Technol 3(3):236–255
32. Zaruba GV, Huber M, Kamangar FA, Chlamtac I (2007) Indoor location tracking using rssi readings from a single wi-fi access point. Wirel Netw 13:221–235





Journal: **13174** MS: **0065** ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:**2012/10/6** Pages: **20** Layout: **Large**