14

Privacy-Aware Access Control System: Evaluation and Decision

Claudio Agostino Ardagna, Sabrina De Capitani di Vimercati, Eros Pedrini, and Pierangela Samarati

Università degli Studi di Milano

14.1 Introduction

The success of the Web as a platform for the distribution of services and dissemination of information makes the protection of users' privacy a fundamental requirement. The privacy issues affect different aspects of today's Internet transactions, among which access control represents the most critical. An important step towards the protection of privacy is then the definition of a privacy-aware access control system that, in addition to server-side resources protection, provides users with solutions for preserving their privacy and managing their data. Although considerable work has been done in the field of access control for distributed services [AHK+03, AHKS02, BS02a, eXt05, Wor02], available access control mechanisms are at an early stage from a privacy protection point of view. This situation reflects the fact that in the last years the variety of security requirements focused on addressing server-side security concerns (e.g., communication confidentiality, unauthorized access to services, data integrity). Here, we focus on the development of a privacy-aware access control system regulating access to resources and protecting privacy of the users.

Generally speaking, an environment well-suited for users that need a private and secure way for using e-services should support at least the following basic requirements.

J. Camenisch, R. Leenes, and D. Sommer (Eds.): Digital Privacy, LNCS 6545, pp. 377–395, 2011.
© Springer-Verlag Berlin Heidelberg 2011

- *Privacy.* A digital identity solution should be respectful of the users' rights to privacy and should not disclose and manage personal information without explicit consent.
- *User-driven constraints.* In addition to traditional server-side access control rules, users should be able to specify constraints and restrictions about the usage that will be made of their information once released to external parties.
- *Minimal disclosure.* Service providers must require the least set of credentials needed for service provision, and users should be able to provide credentials selectively, according to the type of online services they wish to access.
- *Interactive enforcement.* A new way of enforcing the access control process should be defined based on a negotiation protocol aimed at establishing the least set of information that the requester has to disclose to access the desired service.
- *Anonymity support.* As a special but notable case of minimal disclosure, many services do not need to know the real identity of a user. Pseudonyms, multiple digital identities, and even anonymous accesses must be adopted when possible.
- *Legislation support.* Privacy-related legislation is becoming a powerful driver towards the adoption of digital identities. The exchange of identity data should not violate government legislations such as the Health Insurance Portability and Accountability Act (HIPAA) or Gramm-Leach-Bliley Act (GLB).

In the following, we present the prototype of a privacy-aware access control system, which supports the above requirements and integrates traditional access control mechanisms with release and data handling policies. In particular, we focus our discussion on policy evaluation and composition. Our privacy-aware access control system deals with five main key aspects: *i) resource representation,* ability to specify access control requirements about resources in terms of available *metadata* describing them; *ii) subject identity,* the evaluation of conditions on the subject requesting access to a resource often means accessing personal information. This raises a number of privacy issues, since electronic transactions (e.g., purchases) require release of a far greater quantity of information than their physical counterparts; *iii) secondary use,* although users provide personal information for use in one specific context, they often have no idea on how such personal information may be used subsequently. Users should be able to define restrictions on how their information will be used and processed by external parties; *iv) context representation,* context information is a set of metadata identifying and possibly describing entities of interest, such as subjects and objects, as well as any ambient parameter (including location) concerning the technological and cultural environment where a transaction takes place. As far as policy enforcement is concerned, context contains information enabling verification of policy conditions and,

therefore, it should be made available to any authorized service/application at any time and in a standard format. A major factor harnessing the potential of context representation is the lack of a standard context representation metadata layer; *v) ontology integration,* a privacy-aware access control should exploit the Semantic Web to allow the definition of access control rules based on generic assertions defined over concepts in the ontologies, which control metadata content and provide abstract subject domain concepts [ADD+05]. A central element of semantic-aware privacy policies is the use of semantic portfolio supporting controlled access to contextual resources (e.g., personal, company, and public services) subject to user-specified privacy constraints.

The remainder of this chapter is organized as follow. Section 14.2 presents the interactions between parties for data and services release. Section 14.3 describes the architecture of the access control module. Section 14.4 presents how the policies are evaluated. Section 14.5 describes the prototypes of *Access Control Decision Function* and *Policy Management* components. Section 14.6 analyzes the performance of the decision process.

14.2 Interplay between Parties

The infrastructure in Figure 11.1 is aimed at managing two different interplays between the involved parties (see Figure 14.1):

- *User-Service Provider interplay*, when a User submits an access request for a resource managed by the Service Provider, and
- *Service Provider-External Party interplay*, when an External Party submits an access request for sensitive information of a User stored by the Service Provider.

The access request submitted by a client or an external party can be defined as follows.

Definition 6 (Access request). *An* access request *is a 4-tuple of the form* ⟨user_id, action, object, purposes⟩*, where* user_id *is the optional identifier/ pseudonym of the requester,* action *is the action that is being requested,* object *is the object on which the requester wishes to perform the action, and* purposes *is the purpose or a group thereof for which the object is requested.*

For instance, the access request ⟨*Alice, execute, book_a_flight, service_access*⟩ states that *Alice* wants to *execute* the service *book_a_flight* for the purpose of accessing the requested service (*service_access*). On the other hand, the access request ⟨*Lufthansa, read, Alice_Credit_Card_Number, service_release*⟩ is a typical example of an External Party request, where external party *Lufthansa* wants to *read* personal data of a client.

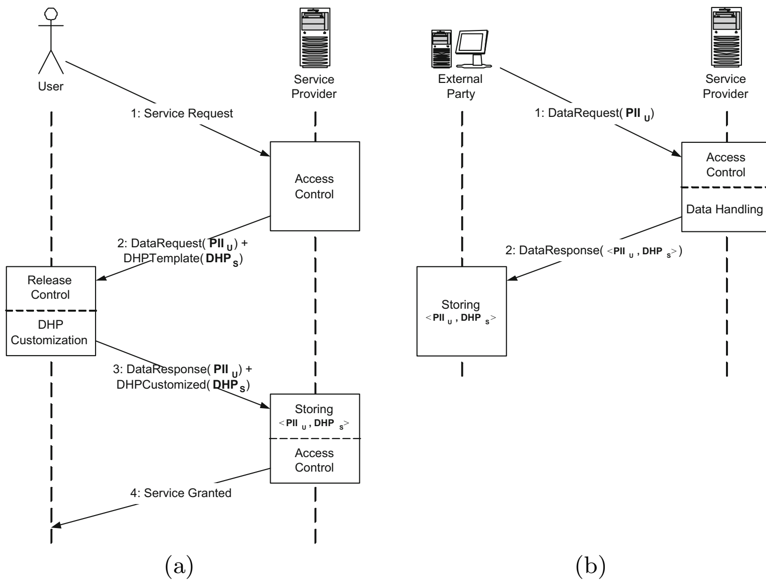


Fig. 14.1 User-Service Provider (a) and External Party-Service Provider (b) interplays

User-Service Provider Interplay

The User-Service Provider interplay is depicted in Figure 14.1(a) (step 1-4). Upon the reception of a service request (step 1), the service provider evaluates its access control policies and, if needed, requests some *PII* from the user (*DataRequest*(PII_U) in step 2); this happens if the information provided by the user is not sufficient for taking an access control decision. In this case, the service provider presents a data handling policy template to the user for customization (*DHPTemplate*(DHP_S) in step 2). The user receives the service provider request, evaluates her release policies, and customizes the data handling policy template. If the *PII* request satisfies at least one (user-side) release policy, the user sends back the required *PII* (PII_U) along with the customized data handling policy (DHP_S) (step 3). Otherwise, if the user's release policies deny the *PII* release, the transaction aborts. Finally, the service provider stores the user's data $\langle\text{PII}_U,\text{DHP}_S\rangle$ and re-evaluates the access control policies based on PII_U. If the evaluation succeeds, the service is granted to the user (step 4). In a more general setup, both the release of *PII* and the data handling policy customization could require multiple negotiation steps [YWS01], rather than a single request-response exchange. The user, for example, may require the service provider to release some *PII* as well, and,

in turn, the service provider may want to specify a data handling policy for such a *PII*.

Service Provider-External Party Interplay

Service Provider-External Party interplay (see Figure 14.1(b)) is temporally subsequent to the interplay between the user and the service provider. Suppose that an external party requests access to personal information of a user (i.e., PII_U) stored at the service provider ($\mathit{DataRequest}(\mathrm{PII}_U)$ in step 1). The External Party-Service Provider interplay begins with a mutual identification, followed, in the most general case, by a negotiation of the data to be released and attached data handling policies.[1] Differently from the previous interplay, in this case the service provider must protect the privacy of the user against the external party, by enforcing its access control policies and the data handling policies attached to the requested information. If the evaluation of access control and data handling policies returns a positive answer, the external party obtains the pair $\langle \mathrm{PII}_U, \mathrm{DHP}_S \rangle$ (step 2). Otherwise, the access is denied.

14.3 A Privacy-Aware Access Control Architecture

Figure 14.2 shows the privacy-aware access control architecture and related modules. Among them, the Access Control module is composed by two main components: *i) Access Control Enforcement Function* (*ACEF*) that is responsible for enforcing access control decisions by intercepting accesses to resources and granting them only if they are part of an operation for which a positive decision has been taken; and *ii) Access Control Decision Function* (*ACDF*) that is responsible for taking an access decision for all access requests directed to data/services.

In the remainder of this section, we focus our discussion on *ACDF* and *Policy Management* (i.e., the modules providing the privacy-aware functionalities), describing their characteristics and the interactions with others components.

14.3.1 Access Control Decision Function

The *Access Control Decision Function* (*ACDF*) component is responsible for taking an access decision for all access requests directed to data/services. *ACDF* produces the final response possibly combining the access decisions coming from the evaluation of different policies (access control, release, and data handling). The elements relevant to a decision are applicable *policies*,

[1] For the sake of clarity, Figure 14.1(b) does not show these steps that are similar to steps 2 and 3 of the User-Service Provider interplay in Figure 14.1(a).

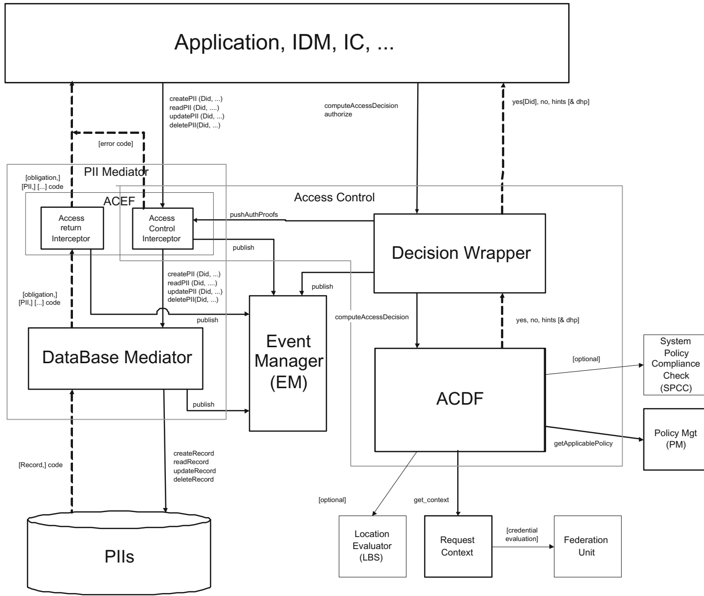


Fig. 14.2 Privacy-aware access control architecture

context which contains the information associated with the requester during a session, and *subject*, *action*, *object*, and *purpose* of the access request (i.e., the 4-tuple introduced in Definition 6). The decision component can return three different decisions:

- *Yes*: the request can be granted;
- *No*: the request must be denied;
- *Undefined*: current information is not sufficient to determine whether the request can be granted or denied. In this case, additional information is needed and the counterpart will be asked to provide such information.

ACDF mainly interacts with two modules: *Request Context* and *Policy Management* (*PM*). The *Request Context* module keeps track of all contextual information, aggregates information from various context sources, and deducts new contextual information from this aggregation. The main tasks of the *Request Context* module are then to provide contextual information from various sources in a standardized way, and to provide reasoning functionalities for boosting the evaluation process. Note that, *ACDF* does not interact directly with the *Request Context* module, but it relies on a Facade[2] component, called *Data Reader*. The Facade component has been designed to

[2] The Facade pattern [GHJV95] encapsulates a complex subsystem within a single interface object, and decouples the subsystem from its potential clients.

simplify the process of retrieving the information needed by *ACDF* for the evaluation. This solution adds a level of isolation that guarantees the simple integration of *ACDF* with different context formats or modules. The *Request Context* module also interacts with *Federation Unit*, a credential verification module in charge of verifying X.509 certificates and IDEMIX credentials [CL01c, CV02] (i.e., anonymous credentials) released by the counterparts. Only verified certificates are stored by the *Request Context* module, and used for policy evaluation.

The *Policy Management* module manages, stores, and distributes the policies to be used in the access control evaluation process. *ACDF* communicates directly with *Policy Management* for retrieving the policies applicable to an access request. We extensively discuss *Policy Management* in the next section.

ACDF also interacts with the *LBS Evaluator* and *System Policy Compliance Check* (*SPCC*) modules. *LBS Evaluator* is in charge of evaluating location-based conditions [ACD+06], such as `in_area(`**user**.Sim, ‘Milan’) that restricts access to requesters located in downtown Milan. The *SPCC* is in charge of handling assurance policies created by users and service providers. A policy in this context is the formalization of another party’s privacy compliance request.

14.3.2 Policy Management

The *Policy Management* (*PM*) module manages the overall policy life cycle by providing functionalities for administering policies. Also, it is the module that interacts with the *Access Control* (*AC*) module for filtering responses coming from *AC*, and for restricting the release of sensitive information related to the policy itself or to the status against which the policy has been evaluated. The definition of sensitive information and sanitization operations are issues managed in cooperation with the *AC* module. The *Policy Management* module addresses the following requirements:

- it provides operations for policy administration, such as search, store, update, check, and delete;
- it provides functions for searching policies applicable to a certain request;
- it provides filtering functionalities that restrict the release of sensitive information related to the policies, when additional requests have to be generated from AC;
- it provides policy storage.

As shown in Figure 14.3, the *PM* module includes the *Policy Presentation* (*Ppres*) and the *Policy Processing* (*Pproc*) components. The *Policy Presentation* component acts as a policy presentation interface that receives as input an access request, and returns as output the applicable policies. This interface is used by *ACDF* to take an access decision. Note that *Ppres* communicates directly with *Policy Repository* to retrieve policies. The *Policy Processing* component provides filtering functionalities on the response that

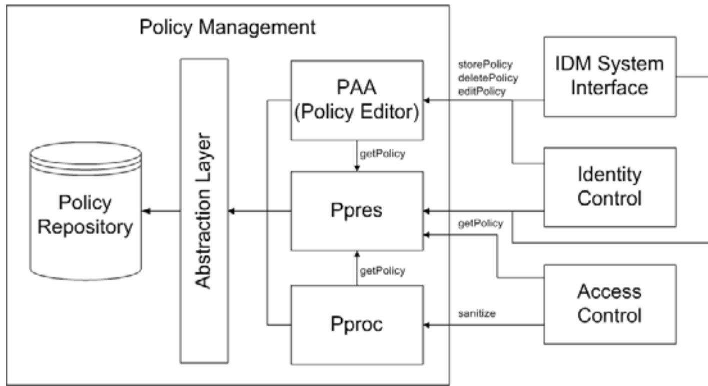


Fig. 14.3 High-level overview of *PM* interactions with other components

the *AC* module returns to the counterparts. This avoids the release of sensitive information related to the policy itself. For instance, suppose that the response returned by the access control is 'undefined' because current information is not sufficient to evaluate condition `equal(`**user**.Citizenship,'Italy'`)`. *PM* could decide to return to the user the response as it is, including `equal(`**user**.Citizenship,'Italy'`)`, or otherwise it could modify (sanitize) the condition by requesting the user to declare its nationality (e.g., 'give me your citizenship'). This way, the fact that access is restricted to EU citizens is not disclosed. The filtered information is called sanitized information. The definition of sensitive information and sanitization operations are issues managed in cooperation with the *AC* module. To conclude, *Policy Repository* (*PR*) provides policy storage and search functionalities. *PR* is based on the relational database concept and is designed to be independent from the real storage infrastructure. *PR* provides a fine-grained query infrastructure, based on policy constraints (i.e., object, multiple actions, subject, and so on). An *Abstraction Layer* hides low-level details and isolates *PR* from the outside. By default, *PR* works in an asynchronous way, allowing concurrent access to the data. Each request to *PR* is filtered by *Abstraction Layer*, which unifies the interfaces to the *PR* component.

14.4 Policy Evaluation

We now discuss how access control and data handling policies are evaluated together. Given an access request (see Definition 6) where the object can be a service or some *PII* associated with a user, the request is first received by *ACEF* and then evaluated by *ACDF* in two steps as follows.

Step 1. The access request is evaluated against the applicable access control (or release) policies (see Chapter 11 for more details) collected from the Policy Management module. Note that, if no policy is selected, the access is denied (i.e., the default access decision is *deny-all*). The current implementation is based on policies specified in a Disjunctive Normal Form (DNF), meaning that rules inside the policies are ORed and conditions inside the rules are ANDed. After collecting all applicable policies, each policy is evaluated by inserting the policy conditions in a Reverse Polish Notation (RPN) stack. At the end of the process, the system combines the results of each policy evaluation to reach a 'yes', 'no', or 'undefined' access decision. In case of a negative ('no') access decision, the access request is denied, and the process terminates. In case of a positive ('yes') access decision, *ACDF* has to verify whether there exists some restrictions on the secondary use of the requested object (see Step 2). In case of an 'undefined' access decision, the information submitted by the requester is insufficient to determine whether the request can be granted or denied. Additional information is required by communicating filtered queries to the requester. Such requests are called *claim requests*. It is important to highlight that a claim request could contain sensitive information. In this case, a sanitization process is performed before the claim request is sent to the counterpart, to avoid the release of sensitive information related to the policy itself.

Step 2. *ACDF* queries the PM module to retrieve all data handling policies attached to the object of the request. If no data handling policy is applicable to the request, Step 2 is skipped and the access is granted. Otherwise, applicable data handling rules are retrieved by using *action* and *purposes* specified in the access request as keys. For each applicable data handling rule, the system evaluates the conditions specified in the *recipients*, *gen_conditions*, and *prov* fields (see Chapter 11 for more details). A positive decision is taken if all applicable rules are satisfied by the requester.

Finally, *ACEF* enforces the final access control decision, which is generated by ACDF by composing the results of the above steps of evaluation. In case a positive ('yes') access decision is retrieved in both steps, the requested data/service is released to the requester together with corresponding data handling policies. The requester is then responsible for managing the received data/service following the attached data handling policies.

14.5 A Privacy-Aware Access Control System Prototype

We present the Java-based prototypes of the modules that are part of the privacy-aware access control system developed in the context of the European PRIME project. In particular, we discuss technical details of the *ACDF* and *PM* prototypes, which provide a solution to integrate traditional access

control mechanisms with release and data handling policy evaluation and enforcement. The *ACDF* and *PM* prototypes have been designed taking into account the following requirements:

- *ACDF* and *PM* have to present clear, well-defined, and independent interfaces;
- *ACDF* and *PM* have to support multi-threading, to manage multiple requests at the same time;[3]
- *ACDF* has to support composition of different types of policies;
- *PM* has to be independent from the adopted physical storage.

Our prototypes have been further integrated within the PRIME architecture in Figure 14.2, to provide a complete privacy-aware identity management solution.

14.5.1 ACDF Prototype

The *ACDF* component takes an access decision for all access requests directed to data/services, by evaluating all the policies applicable to the requests. *ACDF* has been designed to be thread-safe and then its implementation supports the execution of multiple *ACDF* instances running at the same time, without any interaction among them. After receiving the access request, *ACDF*:

1. retrieves the access control (release, resp.) policies by querying *PM*;
2. evaluates the access control (release, resp.) policies by using a Reverse Polish Notation (RPN) stack, and takes an access decision;
3. collects the data handling policies attached to the target of the request;
4. evaluates the data handling policies by using a Reverse Polish Notation (RPN) stack, and takes a decision;
5. composes the different evaluations to generate a single access decision.

The mechanism used to evaluate the different types of policies is the same and relies on a Reverse Polish Notation (RPN) stack. In particular, the peculiarity of the Reverse Polish Notation is that the operators follow their operands. For instance, a sum between three and four, which is usually written '3 + 4', in Reverse Polish Notation becomes '3 4 +'. Often interpreters of Reverse Polish notation are stack-based, that is, operands and operators are pushed onto a stack, and when an operation is executed, its operands are extracted from the stack and the result of the operator evaluation is then pushed on the stack. The current implementation of our *ACDF* RPN stack supports the following operands for access control/release policies (and implements similar ones for data handling policies).

[3] To provide multi-threading support, all the modules that interact with *ACDF* and *PM* should support it.

	Yes	No	Undefined
Yes	**Y**	**N**	**U**
No	**N**	**N**	**N**
Undefined	**U**	**N**	**U**

(a)

	Yes	No	Undefined
Yes	**Y**	**Y**	**Y**
No	**Y**	**N**	**U**
Undefined	**Y**	**U**	**U**

(b)

Fig. 14.4 3-state *and* operator (a) and 3-state *or* operator (b)

- **SEAnd:** it implements the logical *and* between two *3-state* values (Yes, No, Undefined) as shown in Figure 14.4(a);
- **SEOr:** it implements the logical *or* between two *3-state* values (Yes, No, Undefined) as shown in Figure 14.4(b);
- **SEAction:** it implements the evaluation of the *actions* field of the access/release policies;
- **SEEvidence:** it implements the evaluation of a complex statement (e.g., `equal(`**user**.Name.Given,'Alice'`)`) using certified information of the requester (i.e., *credentials*);
- **SEOperator:** it implements the evaluation of a complex statement (e.g., `greater_than(`**user**.age,18`)`) using declared information of the requester (i.e., *declarations*);
- **SEPurpose:** it implements the evaluation of the *purpose* field of the access/release policies;
- **SESubject:** it implements the evaluation of the *subject* field of the access/release policies.

The RPN stack-based evaluation has the main advantages of being very fast and of making the evaluation process independent from policy syntax and semantics. The translation from each policy language (both access/release and data handling languages) to the RPN format is made by the specific implementation of the *PolicyLoader* interface depicted in Figure 14.5. In particular, the *DHPolicyLoader* class interprets the syntax of DHPs, while the *ACPolicyLoader* class interprets the syntax of access/release policies. In this way, it is possible to add new policy languages by implementing the specific loading class, with a minimal impact on the current implementation.

To conclude this overview of the *ACDF* prototype, it is important to remark that *ACDF* supports conditions to be evaluated both on certified data, issued and signed by authorities trusted for making the statement, and uncertified data, signed by the data owner itself. The declared and certified information relevant to the evaluation process is retrieved from the *Request Context* module. In case of certified information, the *Request Context* module retrieves the information needed by *ACDF* by using the evidence specified within the statement of the policy to be evaluated. For instance, the following XML fragment requires a user with age greater than eighteen. The age attribute has to be certified (evidence) with an *identity document* released by the *Italian Public Administration*.

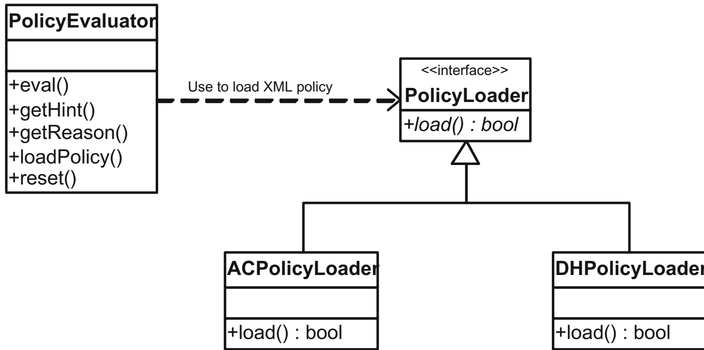


Fig. 14.5 Policy loader infrastructure

```
<group>
 <condition name="greater">
  <argument isLiteral="false">age</argument>
  <argument isLiteral="true">18</argument>
 </condition>
 <evidence>
  <issuer>ItalianPublicAdministration</issuer>
  <proofMethod>X.509</proofMethod>
  <type>identity-document</type>
 </evidence>
</group>
```

14.5.2 PM Prototype

The *PM* module provides functionalities for policy administration. However, considering a privacy-aware access control, the *PM* module accomplishes two main tasks: *i)* it provides a searching engine to retrieve applicable policies to a given request, and *ii)* it provides sanitization functionality.

As depicted in Figure 14.3, the policy search engine is based on an *Abstraction Layer* that provides generic interfaces for accessing *Policy Repository* (*PR*). The *Abstraction Layer* is designed to support multi-threading access to the physical policy storage. The multi-thread supported in *PR* is based on the *Simple Concurrent Object Oriented Programming* (*SCOOP*) model [Mey93], where the concept of thread is extended to the concept of *active object*.[4] To support the *SCOOP* model, *PR* is designed as a singleton *Consumer* [GHJV95]. The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. Note that the Singleton

[4] Differently from traditional thread concept, *all* the methods of an *active object* run in the separate thread.

pattern can be extended to support access to an application-specific number of instances. *PR* then runs in a separate thread and waits asynchronously for the requests that a set of *Producers* insert in the *PR*'s request queue. Each *Producer* registers a callback method within the request. Finally, *PR* processes the requests one-by-one, and the results are returned to the original requesters.

The *PR* has been designed to support different kinds of repositories. In the current implementation two storage engines are supported:

- **MySQL** [MyS07]: it represents the world's most popular open source database because of its consistent fast performance, high reliability, and ease of use.
- **HSQLDB** [hsq07]: it is a leading SQL relational database engine written in Java. It has a JDBC driver and supports a rich subset of *ANSI-92 SQL* plus *SQL 99* and *2003* enhancements. It offers a small, fast database engine which gives both in-memory and disk-based tables, and supports embedded and server modes.

New engines can be added, if necessary, by implementing the *StorageDriver* interface. This interface works as a *Facade* class on the physical storage, assuring the following basic functionalities: policy addition, policy update, policy deletion, and policy searching.

Focusing on sanitization, *PM* gives a thread-safe process, which provides filtering functionalities on the response to be returned to the counterpart to avoid release of sensitive information related to the policy itself. In particular, it obfuscates conditions in the response to be returned to the counterpart as the access response, according to three possible levels of sanitization: *i) strong sanitization* meaning that a full sanitized condition is generated (e.g., given the original condition '`equal(`**user**.Name.Given,'Alice'`)`', a request for declaring **user**.Name.Given is returned); *ii) weak sanitization* meaning that a partial sanitized condition is returned (e.g., given the original condition '`greater_than(`**user**.Age,18`)`', a request for declaring **user**.Age together with the applied operator is returned); *iii) no sanitization* meaning that the full, un-sanitized condition is returned (e.g., '`equal(`**user**.Citizenship,'Italy'`)`' is sent to the counterpart as it is).

14.6 Performance Analysis

In this section, we analyze the performance of our access control prototype based on experimental data coming from the testing of the PRIME integrated prototype. The testing has been performed using two different system configurations. The first system configuration (**A**) used a desktop PC with a 3GHz *Intel Core 2 Duo E6850* processor, 4GB of RAM, a *SATA2* disk interface combined with two disks at 7200rpm, and *MySQL* version 5.0.45 installed. The second system configuration (**B**) used a notebook PC with a 2GHz *Intel*

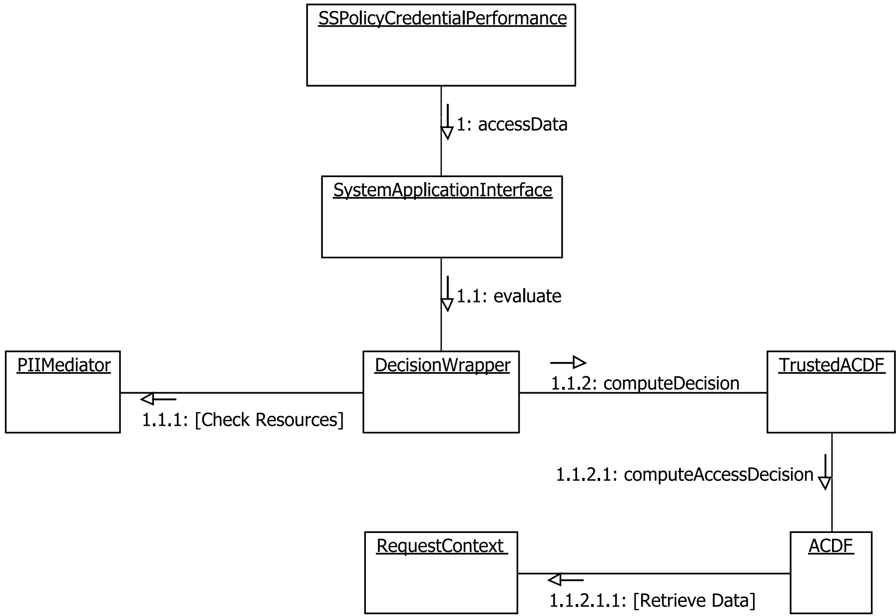


Fig. 14.6 High-level comunication diagram of the evaluation flow

Centrino Duo T2500 processor, 2GB of RAM, a *SATA* disk interface combined with one disk at 7200rpm, and *MySQL* version 5.0.37 installed.

For each system configuration, two batteries of tests have been defined and executed. To minimize spurious cases three runnings for each battery have been performed and their results have been aggregated using the average function. The first battery (battery 1) used an initially empty database to store *PII* and *policies*, while the second one (battery 2) used the database created from the first battery. Each test has then performed 1000 cycles of access requests to resources and, at each cycle, one new policy and one new credential have been created and stored in the MySQL database. Finally, the tests have assumed the scenario where each request is evaluated to 'yes' and the access is granted.

14.6.1 The Evaluation Flow

We provide a performance analysis based on the evaluation flow depicted in Figure 14.6. In particular, the test class requests an access to a resource by calling the *SystemApplicationInterface.accessData()* method. This method forwards the call to the *DecisionWrapper.evaluate()*, and then manages the *DecisionWrapper* result to support interaction with the counterparts. The *DecisionWrapper* component checks if the resource is stored in the *PII* database via *PIIMediator*, which is the component responsible for the management of

the *PII* stored in the database. If the resource is in the database, *Decision-Wrapper* routes the access request to the *ACDF* component[5] preparing the *Request Context* module needed by *ACDF* itself. The *ACDF* component is then responsible for evaluating the policies. To accomplish this task, *ACDF* interacts with the *Request Context* module to retrieve all information needed during the evaluation.

To provide a reliable estimation of the time spent during the evaluation, a number of probing points have been added to measure the following parameters:

- time spent by the *DecisionWrapper* component;
- total time spent by the *ACDF* component to evaluate the policies;
- time spent by the *ACDF* component to evaluate *access/release policies* only;
- time spent during the *access/release policy* evaluation to retrieve the information needed for evaluation, that is, the time spent to access to *Request Context* module;

The above measurements are used in conjunction with the performance data retrieved by *SSPolicyCredentialPerformance* class[6] to measure the total time required to generate a *positive* evaluation response.

14.6.2 Performance Results

Figures 14.7 and 14.8 show the results of our experiments when a *positive* evaluation is reached, and battery 1 and battery 2 are used, respectively. In particular, they show the average percentage of evaluation time spent: *i)* before the access request is sent to *DecisionWrapper*; *ii)* in the *DecisionWrapper.evaluate* method; *iii)* in *DecisionWrapper*; *iv)* in the *TrustedACDF.computeDecision* method (with respect to the total time); *v)* in the *TrustedACDF.compute Decision* method (with respect to the quantity measured at point *ii)*).

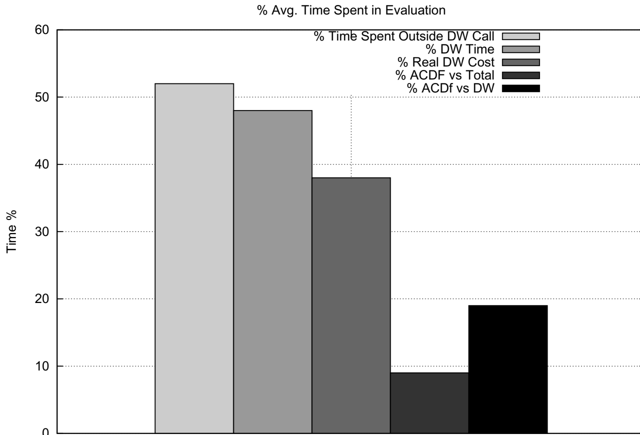
By a first analysis of the results shown in Figures 14.7 and 14.8, it comes with no surprise that independently from the batteries of tests under evaluation:

- different amount of RAM used in the two systems configurations do not influence the evaluation performance;[7]

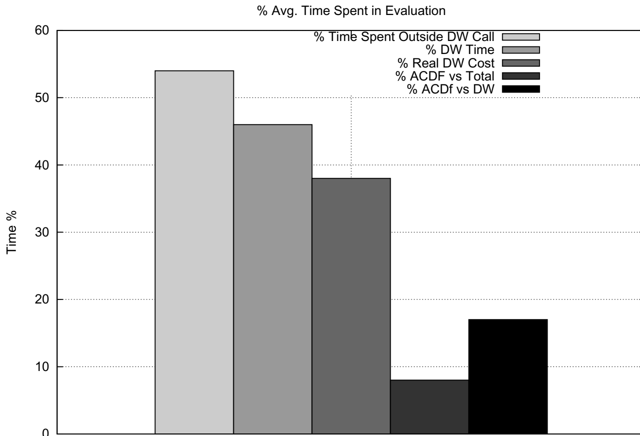
[5] Note that, the *DecisionWrapper* does not call directly the *ACDF* component, but the call is wrapped in the *TrustedACDF* class.

[6] *SSPolicyCredentialPerformance* is the test class developed by *BluES'n* application prototype (see Chapter 24 for more details) and used as a starting point in our experiments.

[7] This is probably due to the fact that the testing environment is Windows-based, and then the *Java Runtime Environment* does not take advantages from amount of RAM greater than 2GB.



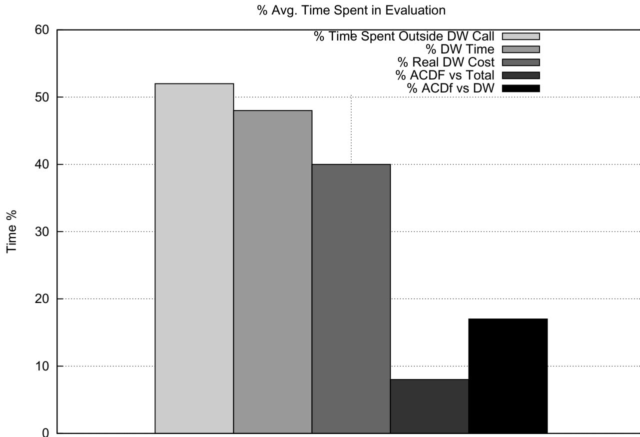
(a)



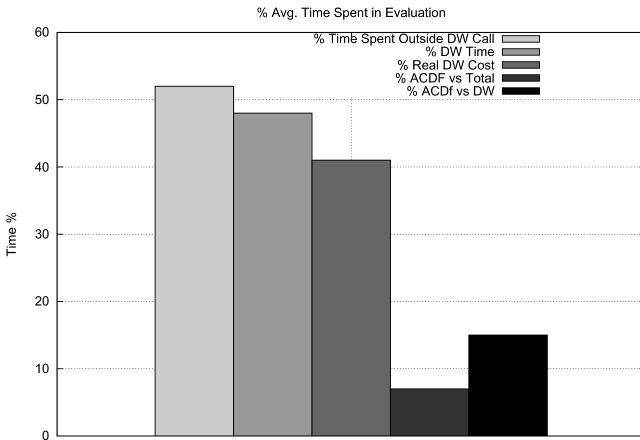
(b)

Fig. 14.7 Average percentage of evaluation time used during the test assuming an initially empty database (battery 1) for the reference system A (a) and B (b)

- different *MySQL* configurations (using the caching support or increasing the quantity of RAM used as cache) seem to do not impact much on the overall test performance. Some empirical tests show an improvement around the 3-5% with respect to the total evaluation time;



(a)



(b)

Fig. 14.8 Average percentage of evaluation time spent during the test assuming an existing database (battery 2) for the reference system A (a) and B (b)

- *SATA2* disk interface seems to guarantee a measurable improvement of the performance; *SATA2* interface in fact should guarantee a doubled bandwidth with respect to *SATA* one.

In more detail, the experimental results in Figure 14.7, which used an initially empty database (i.e., battery 1), show that the distribution of the times over the different components is independent from the system configuration used to perform the analysis. By contrast, the real time spent in the two system configurations is very different. Specifically, the average of the total time spent by system A (Figure 14.7(a)) is 9.1s, with minimum and maximum times 1.0s and 17.7s, respectively, while the average of the total time spent by system B (Figure 14.7(b)) is 24.9s, with minimum and maximum times 3.5s and 53.6s, respectively.

Also the experimental results in Figure 14.8, which used the database filled in the battery 1 experiment (i.e., battery 2), show that the distribution of the times over the different components is independent from the system configuration used to perform the analysis, while the average of the total time spent is very different in the two systems. In particular, the average of the total time spent by system A (Figure 14.8(a)) is 23.5s, with minimum and maximum times 18.4s and 42.8s, respectively, while the average of the total time spent by system B (Figure 14.8(b)) is 66.4s, with minimum and maximum times 50.6s and 101.0s, respectively. In both cases, the differences are due to the different hardware characteristics of the two system configurations.

Comparing the outcomes of the two batteries of tests, it results that the time needed for the overall evaluation flow depends mainly on the database status, that is, how much *PII* data and how many *policies* are stored.

To conclude our performance analysis, we analyze the time required by the *ACDF.computeDecision* method for taking an access control decision. As shown in Figures 14.7 and 14.8, the time used by *ACDF* to compute an access decision represents a minimal part of the time taken by the overall evaluation flow. Also, most of the time required by *ACDF* is used to retrieve the information needed to perform the evaluation, from the *Request Context* module. In our tests, in fact, the access to *Request Context* module consumes the 90-95% of the total time required by *ACDF*. It is important to highlight that, as policies become more and more complex and the number of credentials increases, the time spent to evaluate the request is likely to increase with respect to the time necessary for retrieving the needed credentials.

14.7 Conclusions

In Chapter 11, we have defined an access control model and language for restricting access to resources/data managed by a service provider and release of PII managed by the users, and a data handling model and language

allowing the users to pose restrictions on secondary use of their private data. Here, we have described the prototypes of the components, and the overall architecture providing functionalities for integrating access control/release and data handling policy evaluation and enforcement.