Managing Distributed UCON$_{ABC}$ Policies with Authorization Assertions and Policy Templates

Maicon Stihler*†,
*Federal Center for Technological Education – CEFET-MG
Department of Computing and Mechanics
Leopoldina, MG, Brazil
e-mail: stihler@leopoldina.cefetmg.br

Altair O. Santin†, Arlindo L. Marcon Jr.†
†Pontifical Catholic University of Parana – PUCPR
Graduate Program on Computer Science – PPGIa
Curitiba, PR, Brazil
e-mail: {santin,almjr}@ppgia.pucpr.br

***Abstract*—Managing UCON$_{ABC}$ policies in modern distributed computing systems is a challenge for traditional approaches. The provisioning model has trouble to keep track and to synchronize large numbers of distributed policies, outsourcing model may suffer from network overhead and single point of failure. This paper describes an approach to manage distributed UCON$_{ABC}$ policies, derived from the combination of authorization assertions and policy templates. It combines the benefits of provisioning and outsourcing, eliminating their respective drawbacks. Prototyping details and performance evaluation are shown, messages are 42.7% smaller than provisioning and response times are faster than outsourcing.**

I. Introduction

Cloud computing [1] provides a dynamic environment, in which client organizations can implement their own computing services using infrastructure provisioned on demand. This reconfigurable infrastructure allows the services to be scaled up or down as needed. Cloud providers are accessible from any computer connected to the Internet and can be used without requiring the installation and configuration of specialized hardware or software on the client side. These environments can also support many users on the same infrastructure (i.e., Multitenancy), improving resource utilization.

The abstraction level of these environments are commonly used to categorize them as being Infrastructure as a Service (IaaS), Platform as a Service (PaaS) or Software as a Service (SaaS) [2]. IaaS clouds provide virtual hardware infrastructure controlled by the users. The PaaS hides the virtual infrastructure and provides intermediary services (e.g. security and user management) supporting the development and deployment of services. SaaS provides complete applications to the end user, who can only modify some application aspects. Each model is conceptually independent, although, one may use a given model (e.g. IaaS) to build another model (e.g. PaaS or SaaS).

Cloud computing access controls show different degrees of granularity. IaaS environments, like Amazon EC2 [3], usually enforce access controls on whole virtual machines (VM), while PaaS providers, like Heroku [4], often enforce the controls on a lightweight process container. SaaS solutions, on the other hand, focus on individual application users.

This work was partially sponsored by the Brazilian National Council for Scientific and Technological Development (CNPq), grants 310671/2012-4 and 404963/2013-7.

Takabi [5] argued that cloud computing must use fine-grained access control policies, which require mechanisms able to capture the cloud dynamics, through the use of contextual information, attributes and credentials. The usage control model (UCON$_{ABC}$) [6] meets the aforementioned requirements, as it can reflect changes on attributes of users, objects and the environment, by continuously reevaluating the policies and applying the usage decisions during runtime.

Previous research on the application of UCON$_{ABC}$ in cloud computing [7], [8], [9] were centralized approaches, prone to communication overhead, single point of failure in the reference monitor and low scalability. Tuple spaces was investigated to solve these shortcomings [7], though its use of non-standard complex mechanisms [10], [11] hinders its applicability. Decentralized approaches based on the provisioning model, on the other hand, have difficulties to synchronize a large number of policies in distributed systems like cloud computing.

It is possible to protect any kind of abstract object (e.g. files, VMs) using UCON$_{ABC}$. The definition of what an object is can impact its access control granularity. VMs (IaaS) can only support coarse grained controls and is hard to reconfigure without restarts. Process containers (PaaS), on the contrary, allow for a fine-grained control, configuration parameters can be updated without restarts, and the computing overhead is low. SaaS could potentially offer an even higher degree of control, however, the control mechanisms would not be generic enough to be used on other services.

This work describes the management of usage control policies with an architecture based on provisioning of authorization assertions, policy templates previously configured in a runtime environment and process containers. Individual policies are derived on the local access control mechanisms, by combining information taken from the provisioned authorization assertions and the local policy templates. The proposal eliminates the need for synchronizing local policies with a centralized policy management system, enables the enforcement of individually customized policies on a lightweight container, and allows a higher frequency (almost continuous) policy reevaluation when compared to outsourcing approaches.

The paper is organized as follows: Section II presents the preliminary concepts; Section III discusses the proposal in details; Implementation and evaluation of a prototype, as well as its tests are described in Section IV; Related work are presented in Section V; Section VI shows our concluding remarks.

II. Preliminaries

This section presents a brief introduction to key concepts for understanding this paper. In particular, it addresses the concepts of policy architectures and the UCON$_{ABC}$.

A. Policy Architectures

Access control policies have been commonly managed following on of two approaches:

- **Provisioning:** requires the setup of policies at the place where the controls are enforced. The policy is stored in a local repository and then, on each access request, the resource guardian invokes a local policy decision point (PDP). The PDP is a reference monitor which produces policy evaluation decisions (i.e. permit or deny) to be enforced by the guardian [12]. Its main advantage is its robustness, as it has no external dependencies. On the other hand, it is harder to keep policies synchronized on a distributed system.
- **Outsourcing:** is based on a client and server model. An external service, a reference monitor, responds to requests from the guardians requiring the evaluation of access control policies, this happens on every access attempt. The guardian waits for a policy evaluation decision and enforces it [13]. Its advantage is the simplicity of policy management and guardian implementation. The disadvantages are the communication overhead and its fragility, as the reference monitor may be a single point of failure. Moreover, by being centralized it becomes harder to achieve the scalability expected of cloud computing systems.

B. Usage Control Model

The usage control model, UCON$_{ABC}$ [6], unifies prior ideas on access controls under a formal (predicate-based) cohesive model. It has two fundamental concepts:

- **Continuity** defines that policies are evaluated and enforced throughout the usage of an object (i.e. the resource being protected). The evaluation may occur before the usage starts (*pre*), while it happens (*ongoing*) and when it finishes (*post*);
- **Mutability** considers that certain attributes of the subject or object may be changed as a side effect of a subject's usage. Thus, policy rules may defined how some attributes get updated.

Usage controls are categorized as authorizations (the user must have rights to perform an action), conditions (constraints on environmental attributes, such as the time of day or geographic location), obligations (external actions that must be performed by the user) and updates (modifications that should be made to a user's or object's attributes).

III. Managing Usage Control

In this section we propose an infrastructure for usage control management in distributed systems, using policy templates and authorization assertions to derive policies that are locally evaluated. This approach eliminates the need of setting up policies, present in the provisioning model, and the communication costs of the outsourcing model.

The components that comprise the infrastructure can be categorized as being part of an *administrative domain* or of a *local domain*. The administrative domain offers services to manage the contents of authorization assertions and to create the policy templates to be preset in the *local domains*. The local domain contains the components for authorization assertion validation, derivation of usage control policies, its evaluation and decision enforcement.

User applications are executed in the local domain in isolated environments, called containers, which allow for fine-grained control and accounting, regardless of the number of processes running within the containers. Each container is controlled by a customized policy, derived from an authorization assertion and a policy template preset on the local domain.

The infrastructure was designed to eliminate the need of provisioning policies for each service, as well as the complex mechanisms required to keep those policies synchronized with a central repository. Local usage control mechanisms allow for shorter response times when compared to outsourcing approaches and reduces the likelihood of single point of failure, due to local domains being independent of each other.

A. Administrative Domain

The administrative domain consists of four services (see Fig. 1): Policy Administration Point (PAP), where policy templates are created; Attribute Manager (AM), used for managing each user's authorization attributes; Security Token Service (STS), issues authorization assertions; Security Gateway (SG), prevents assertions from being used more than once.

Policy templates are managed on the PAP and stored on the policy template repository. A template contains a set of authorization rules, and each rule contains some fields (identified by unique names, the *field-id*), to be filled with attributes obtained from the authorization assertions. The template must contain all the rules (e.g. quota for disk and CPU time) that an administrator needs to enforce on a service container. Different authorization assertions may activate different sets of rules, therefore, each user can have a customized policy.

All local domains are preset with a copy of the template repository, making them available as soon as the local domain is operational. This local repository can be updated at any time by a notification system (e.g. when a template changes in the administrative domain, a reliable notification system informs all the local domains affected, which then update their repositories and derive again the involved policies).

The AM service consists of a repository of attributes used for issuing authorization assertions. Each attribute contains a *field-id*, which is matched with the *field-id*s on the policy template. The repository controls the amount of resources that can be issued for each user and the amounts already issued

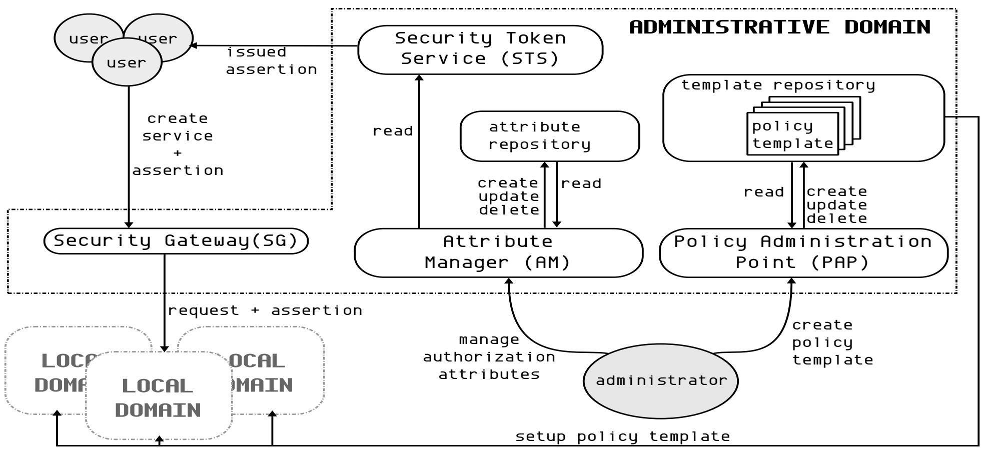

Fig. 1. Administrative domain components

(e.g. an user may have a disk quota of 100 GB, but 10 GB is already used). When an assertion is issued, the corresponding amount is accounted on the repository, thus an user cannot exceed his/her total quota.

Attributes are used by the STS to issue authorization assertions upon user requests. After user authentication, the STS requests the AM to validate the attributes claimed by a user. A valid request (i.e. authentic and not exceeding the administrative domain quota amount) updates the AM repository, reducing the available quota and allowing the STS to issue an authorization assertion with the requested attributes. This assertion is then signed by the STS, encrypted (only the local domains may decrypt it) and then returned to the user.

Security gateways control the interaction of users with the local domains, all application management requests must be made through this service. It transparently selects local domains to perform user requests, and prevents authorization assertions from being used more than once. Therefore, a security gateway needs to keep track of session state (e.g. in which local domain each request is being processed and what assertion is related to it) and to reliably share this information with other security gateways – the synchronization mechanism is considered a future work.

User application requests are forwarded directly to the application, bypassing the security gateway. Therefore, if the administrative domain becomes temporarily unavailable, the current applications will still be usable (though the user will not be able to change its parameters). A certain degree of fault tolerance is desirable on the administrative domain, to avoid delaying the creation and management of user applications, however, it is not as critical as in the outsourcing model, which may have a single point of failure.

B. Local Domain

A local domain is the runtime environment where services (i.e. user applications) can be executed and usage controlled. A service is executed in an isolated environment provided by the operating system, a container. This allows the execution of services from different users alongside each other. The local mechanisms validate authorization assertions and combine them with policy templates, generating the usage control policies to be applied to service containers. The evaluation and enforcement of these policies is made locally. For an illustration of the local domain components see Fig. 2.

Users can perform two request types: application and management. Application requests always target the service being executed inside the container. Management requests, on the other hand, are used to create, modify, delete or retrieve information about a particular service (e.g. request to start a web application). Local control mechanisms are focused solely on the second type of request, application requests are forwarded to the service itself.

When the user wants to perform a management request, he/she must submit the request with an authorization assertion to the SG (Security Gateway). The SG selects the appropriate local domain and forwards the user request, assuring the authorization assertion has not been already used elsewhere.

A Policy Enforcement Point (PEP) receives the request from the SG and ensures that only authorized requests get executed. The first step in the authorization process is to invoke a local security token service (STS) to validate the assertion (e.g. by verifying expiration dates, authenticity of signatures, trust relations, data integrity). An invalid assertion causes the user request to be rejected. After validation, the PEP submits an authorization request to the context handler (CH) along with contextual information (e.g. authorization assertion, request parameters). The PEP then waits for a reply with an authorization decision to be enforced.

The CH integrates the many components of the local domain. After extracting the authorization assertion from the PEP's request, the CH invokes the local policy administration point (LPAP). The LPAP derives usage control policies from authorization assertions and policy templates stored in the template repository. The discovery of which rules must be created (e.g. pre-authorizations, ongoing conditions, etc.) is made by matching attributes contained in the assertions with the *field-id*s present on the policy template. Each rule with a matching *field-id* is configured with the corresponding attribute value. The resulting policy is stored in a policy repository and a success message is returned to the CH, allowing the authorization evaluation process to continue.

The next step for the CH is to contact the policy informa-

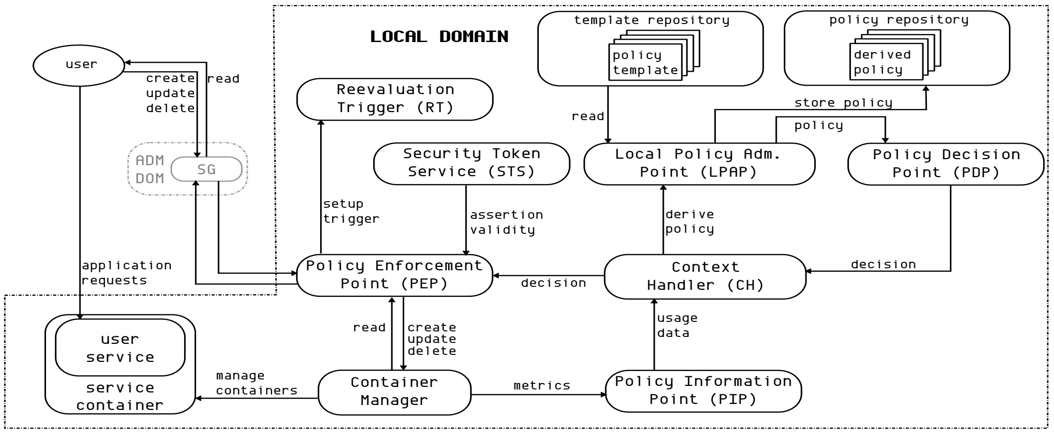

Fig. 2. Local Domain Components

tion point (PIP), which provides resource usage information through a well known interface. Data is collected by other components (e.g. accounting agents) and stored on the PIP's repository. These components use the operating system native APIs to discover the resource usage for each individual container, as well as the current state of the system (e.g. load average, number of running processes). UCON$_{ABC}$ obligations are treated as normal attributes stored on the PIP and must be updated by an external agent, because obligations cannot be controlled within the system.

The data retrieved from the PIP is combined with contextual information from the PEP's request to create an policy evaluation request, which is sent to the policy evaluator (PDP). This component evaluates access control policies, matching the data contained on the CH request with the applicable policy rules. The policy applicable to the service container is retrieved from the LPAP. Each policy is linked to a single authorization assertion, therefore, the PDP can select the right policy on the LPAP. The lack of an applicable policy causes the request to be denied. A request is authorized only if, after matching all attributes to the applicable policy rules, the rule combining algorithm produces a *permit* value. The decision is sent back to the CH, that forwards it to the PEP along with details of how the decision must be enforced.

The aforementioned process is repeated for each management request. This process can be better understood with Algorithm 1. The PDP retrieves the subject (S), object (O) and context (C) linked to an assertion. The data is used to retrieve the applicable set of policies from the LPAP address (A). A request gets rejected if no policy is available (line 4-6). A *deny overrides* algorithm is shown from lines 7 to 14: if any rule produces a *Deny* decision, the request is immediately rejected, otherwise the request is authorized.

The PDP's decision may contain a revaluation trigger (RT) and any attribute updates required. The CH invokes the PIP to update any attribute, effectively supporting attribute mutability for UCON$_{ABC}$. Failure to update the attributes causes the user request to be rejected. The decision is converted to the format used on the PEP and sent to it, after updating the attributes.

The PEP configures the RT in a component with the same name. The RT functions act as a timer to alert the PEP to repeat the authorization process periodically. The trigger is created with request data provided by the PEP. This data serves as contextual information to reevaluate the suitable policy. Therefore, the RT component implements the continuity of control, defined by UCON$_{ABC}$, as a configurable periodic reevaluation.

The PEP forwards authorized requests to a local Container Manager to setup the container and start up the user service (i.e., it manages the container life cycle). The service access details (e.g. IP address) are returned to the user after container creation. The Container Manager uses the operating system native mechanisms to configure the container limits in accordance with the values on the authorization assertion.

C. Templates and Policy Derivation

A template is a set of all the rules that can be used to control the behavior of a user service. For the sake of simplicity a version of a rule for controlling CPU time is shown on Fig. 3. Each rule is identified by a *RuleID*: when a rule identifier is present in the authorization assertion, the rule must be activated for this user. A rule may contain a variable number of *field-id*s (e.g. *TotalCpuTime*) that must be replaced by values with the corresponding *field-id*. Thus, the *TotalCpuTime* attribute must be present on the authorization assertion, otherwise no policy will be derived and the request will be refused. Accounting data can also be referred on the policy template through the

Algorithm 1 Policy Evaluation

```
1: $S \leftarrow subject(assertion), O \leftarrow object(assertion)$
2: $C \leftarrow context(request), A \leftarrow address(LPAP)$
3: $PolicySet \leftarrow retrieve\_policy(S, O, A)$
4: if $PolicySet = \emptyset$ then
5:     return Deny
6: end if
7: for $Policy\ in\ PolicySet$ do
8:     for $Rule\ in\ Policy$ do
9:         if $evaluate(Rule, S, O, C) = Deny$ then
10:            return Deny
11:        end if
12:    end for
13: end for
14: return Permit
```

use of variable names (e.g., *usedCpu* is the amount of CPU time already used).

The applicable rules are configured with the attributes from the authorization assertion and, after a successful derivation, the resulting policy is stored on the LPAP. This policy may contain rules to control the full life-cycle of the service container (i.e. *pre* and *ongoing* controls). Changes to the template forces the derivation of the affected policies – the obsolete policy is deleted and the new policy takes place. Policies may be grouped in Policy Sets, each policy representing a well defined stage of the usage session (e.g. *pre-authorization*, *ongoing-conditions*).

IV. Prototype

The local domain components were prototyped and evaluated, demonstrating that the local mechanisms are able to support controls from UCON$_{ABC}$. Furthermore, a performance analysis identified the container's accounting overhead and the PDP message size overhead when compared to a pure provisioning approach.

A. Implementation

To implement the prototype we used some open source libraries and the Java programming language. The application containers were provided by FreeBSD jails [14] mechanism, it features an API to monitor resource utilization and to manage the jails operation. The container manager uses these APIs to create the container where the user service is executed. Jails offer an execution environment that resembles a dedicated operating system. However, different applications are unable to observe or affect the environment outside of their own jails.

The authorization assertions employed the SAML specification [15], more precisely the *AttributeStatement* message type, and were created and handled by the OpenSAML [16] library. Usage control policies were created following the XACML [17] standard format and were evaluated through the WSO2 Balana [18] library. The templates were written as XACML rules with embedded variable names. A search and replace procedure was executed to fill the template with the corresponding attribute values, deriving the policies.

```
<Rule RuleId="CPURule" Effect="Permit">
  <Target><Any/></Target>
  <Condition>
    <Apply FunctionId="integer-less-than-or-equal">
      <Apply FunctionId="integer-one-and-only">
        <AttributeDesignator Category="access-subject"
        AttributeId="usedCpu" DataType="integer"/>
      </Apply>
      <Apply FunctionId="integer-one-and-only">
        <AttributeValue DataType="integer">
          ${TotalCpuTime}
        </AttributeValue>
      </Apply>
    </Apply>
  </Condition>
</Rule>
```

Fig. 3. Sample Rule

A REST Web service executing on the local domain receives requests containing SAML assertions and the desired action to be performed (e.g. make a new service instance). The local STS authenticates and validates the SAML assertion and a context handler is invoked to process the request. Embedded modules derive the XACML policy and save it on a private directory, gather the local information (i.e. on the PIP) and evaluate the policy. A background thread is configured with a parameter from the authorization decision to periodically request the reevaluation of a policy. Session data and policies are kept in a local directory readable only by the context handler. After successful evaluation, the request is executed by the container manager (it creates a container, configures its limits and IP address, starts the desired service and returns the access details). The resulting information is returned to the user.

B. Evaluation

One test measured the cost of retrieving accounting attributes from 400 containers (jails) mimicking a production environment. Each jail was executing services like e-mail, SSH and cron, while the attributes were being retrieved, thus causing a heavy load on the host system. Figure 4-A shows that sequential reading is the best method, with 0.13 ms on average for each jail, while 16 parallel requests spent 1.61 ms on average. The worst case remained stable at 7.81 ms, up to 4 threads, going up to 18.06 ms with 16 threads. Due to this small overhead, process containers are better suited to implement UCON$_{ABC}$ controls to processes not requiring the isolation of full virtual machines.

The second test (Fig. 4-B) compares the proposed approach to traditional provisioning approaches. Provisioning is significantly more expensive when dealing with the same number of rules. The hybrid model used messages ranging from 4754 to 18524 bytes; in the provisioning model messages are ranging from 11314 to 42594 bytes. The scenario involved the use of policies from 6 to 96 rules. The messages for the hybrid model were 42.7% smaller on average.

V. Related Work

In our previous work [7] we designed an architecture for resilient usage control for cloud computing based on the outsourcing model. Tuple spaces were employed to handle the high demand created by the remote PEPs. The architecture handles the discrepancies in the use of resources, in between policy evaluations, by using a share of the resource as a threshold. This proposal employs a hybrid-provisioning model with authorization credentials and policy templates, with local evaluation and enforcement of the dynamically derived policies. By using local mechanisms, we can offer now a much tighter control, avoiding the need for such resilience.

Lazouski and co-authors [8] applied usage control to a IaaS cloud system and created XACML language extensions in order to express attribute updates and reevaluation constraints. Their architecture is based on the outsourcing model and periodically reevaluates policies affected by attribute changes. The authors also consider the possibility of an event-based reevaluation. The possible overload of policy evaluation mechanisms is not addressed. This proposal does not employ extensions to

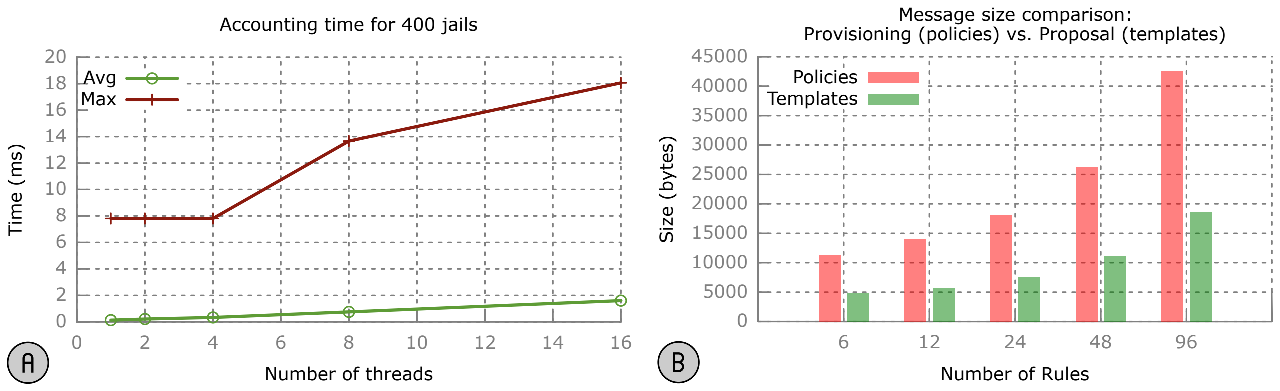

Fig. 4. Prototype evaluation for accounting overhead (A) and message size (B)

the XACML specification; it offers individually configurable reevaluation periods for policies, and uses mechanisms for fine-grained control (i.e. application containers, not full virtual machines).

An access control architecture based on the UCON$_{ABC}$ model was proposed by Danwei and his colleagues [9]. Their main contribution is the inclusion of a negotiation module coupled with the authorization architecture. The user has the possibility of choosing another access option through negotiation, in certain situations, instead of being promptly rejected when an authorization credential is insufficient. Our proposal does not need such fall-back functionality because policies are derived dynamically, thus it is not possible to encounter a mismatch between user attributes and the enforced policy.

VI. Conclusion

This paper presented and evaluated an architecture for the distributed management of usage control in distributed systems (e.g. cloud computing). Our contribution is the use of policy templates and authorization assertions to derive access control policies. Templates are previously stored in the local domains, where policy evaluation and enforcement are performed. The authorization assertions avoid the need of sending whole policies through the network for each user, thus reducing message sizes and keeping the flexibility for defining individual policies.

Policy synchronization is done by sending new assertions with different attribute values – when the local domain detects the change, the affected local policies are derived again and reevaluated.

Tests showed the suitability of containers to implement lightweight UCON$_{ABC}$ controls, and that the proposal significantly reduces message size (42.7% when compared to provisioning). The proposal shows the benefits of the provisioning without the complexity of synchronizing policies.

As future works, we plan to develop a decentralized architecture for sharing attributes between local domains, aiming to enable the dynamic setup of authorization attributes.

References

[1] B. Hayes, "Cloud Computing," *Commun. ACM*, vol. 51, no. 7, pp. 9–11, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1364782.1364786

[2] P. M. Mell and T. Grance, "SP 800-145. The NIST Definition of Cloud Computing," Gaithersburg, MD, United States, Tech. Rep., 2011.

[3] Amazon Web Services, Inc., "Amazon Elastic Compute Cloud (EC2) - Scalable Cloud Hosting," https://aws.amazon.com/ec2/?nc1=f_ls, accessed: 2015-05-03.

[4] Heroku Inc., "Dynos and the Dyno Manager," https://devcenter.heroku.com/articles/dynos, accessed: 2015-05-03.

[5] H. Takabi, J. B. Joshi, and G.-J. Ahn, "Security and Privacy Challenges in Cloud Computing Environments." *IEEE Security & Privacy*, vol. 8, no. 6, pp. 24–31, 2010.

[6] J. Park and R. Sandhu, "The UCON$_{ABC}$ Usage Control Model," *ACM Trans. Inf. Syst. Secur.*, vol. 7, no. 1, pp. 128–174, Feb. 2004. [Online]. Available: http://doi.acm.org/10.1145/984334.984339

[7] A. L. Marcon Jr., A. O. Santin, M. Stihler, and J. Bachtold, "A UCON$_{ABC}$ Resilient Authorization Evaluation for Cloud Computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 2, pp. 457–467, Feb. 2014. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2013.113

[8] A. Lazouski, G. Mancini, F. Martinelli, and P. Mori, "Usage Control in Cloud Systems," in *IEEE 2012 International Conference for Internet Technology And Secured Transactions*. IEEE, 2012, pp. 202–207.

[9] X. R. Danwei Chen, Xiuli Huang, "Access Control of Cloud Service based on UCON," in *Cloud Computing*. Springer, 2009, pp. 559–564.

[10] S. Capizzi, "A Tuple Space Implementation for Large-scale Infrastructures," PhD thesis, Università di Bologna, 2008.

[11] S. Capizzi and A. Messina, "A Tuple Space Service for Large Scale Infrastructures," in *IEEE 17th Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*. IEEE, 2008, pp. 182–187.

[12] K. Chan, J. Seligson, D. Durham, S. Gai, K. McCloghrie, S. Herzog, F. Reichmeyer, R. Yavatkar, and A. Smith, "COPS Usage for Policy Provisioning (COPS-PR)," RFC 3084, IETF, Mar. 2001.

[13] D. Durham, J. Boyle, R. Cohen, S. Herzog, R. Rajan, and A. Sastry, "The COPS (Common Open Policy Service) Protocol," RFC 2748, IETF, Jan. 2000, updated by RFC 4261.

[14] The FreeBSD Project, "The FreeBSD Project," https://www.freebsd.org, 2015, accessed: 2015-05-03.

[15] OASIS Security Services TC, "Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0," http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf, 2014, accessed: 2015-05-03.

[16] OpenSAML Project, "OpenSAML 2 for Java," https://wiki.shibboleth.net/confluence/display/OpenSAML/Home, 2015, accessed: 2015-05-03.

[17] OASIS eXtensible Access Control Markup Language (XACML) TC, "eXtensible Access Control Markup Language (XACML) Version 3.0," http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html, 2014, accessed: 2015-05-03.

[18] WSO2 Inc., "Balana XACML for Authorization," https://svn.wso2.org/repos/wso2/trunk/commons/balana/, 2015, accessed: 2015-05-03.