Maicon Stihler

Decentralized UCON$_{ABC}$ with Cooperative Attribute Management for Cloud Computing

Curitiba

2016

Maicon Stihler

Decentralized UCON$_{ABC}$ with Cooperative Attribute Management for Cloud Computing

Submitted to the Graduate Program in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Informatics at the Pontifical Catholic University of Paraná

Pontifical Catholic University of Paraná – PUCPR

Graduate Program in Computer Science – PPGIa

Advisor: Altair Olivo Santin

Curitiba

2016

Reservado para a ficha catalográfica.

Reservado para a folha de aprovação.

Dedicated to Luiz, Maria, Scheila and Alice.

Acknowledgements

It is impossible to acknowledge all the people that contributed to making this work possible. However, some persons were decisive to its outcome. I would like to thank my advisor, Prof. Dsc. Altair Olivo Santin, who is one of these persons. He gave me constant encouragement and insight. He helped me to keep working while I was struggling to make progress and losing hope. I would also like to thank my parents, Luiz Carlos Stihler and Maria Laurete Stihler, for their support during all my academic journey. My better half, Alice — thank you — always tried to make the burden feel lighter by remembering me where I was headed to and that I was not alone. Without her, I would not have come this far. Last, but not least, I would like to thank all professors that contributed to my work, all my friends, colleagues, and coworkers, for helping me with insights, ideas, constructive criticisms or for simply cheering me up when I needed.

"Perfection is finally attained not when there is no longer anything to add,
but when there is no longer anything to take away"
Antoine de Saint Exupéry

Resumo

Há muitas evidências de que UCON$_{ABC}$ é um modelo de controle de acesso mais adequado para ambientes de computação em nuvem. Ele oferece avaliação contínua das regras de política e foi concebido desde o princípio com a mutabilidade de atributos em mente. Suas implementações para o mundo real devem lidar com o fato de que a reavaliação das regras de política deve ocorrer em intervalos discretos, ou seja, não é possível oferecer um processo de reavaliação realmente contínuo. Propostas anteriores de arquiteturas de UCON$_{ABC}$ para ambientes de nuvem optaram por abordagens baseadas em *outsourcing* (terceirização), que apresenta uma deficiência importante: a imprevisibilidade dos custos de comunicação em rede. Cada requisição de acesso deve ser enviada através da rede e, assim, está sujeita a um custo que pode mudar ao longo do tempo de maneira imprevisível. Deste modo, a freqüência de reavaliação das políticas deve ser ajustada de acordo com os custos do ambiente específico, o que se torna um processo contínuo em um ambiente que pode aumentar ou diminuir de tamanho conforme a demanda. Isto se torna ainda mais complicado pela necessidade de se sincronizar a reavaliação de políticas com um mecanismo de contabilidade distribuído e intrinsecamente assíncrono, usado para descobrir quanto recurso se está utilizando nos servidores remotos (onde a decisão de política deverá ser aplicada). Esta Tese de Doutorado descreve uma arquitetura descentralizada de UCON$_{ABC}$ com características mais apropriadas para ambientes de computação em nuvem. A descentralização é alcançada através de uma abordagem baseada em *provisioning* (configuração) modificada, sendo que avaliação de políticas e aplicação de decisões acontecem localmente, reduzindo o efeito da imprevisibilidade dos custos da rede. A complexidade no gerenciamento de políticas é evitada através do uso de uma combinação de credenciais do usuário e gabaritos de políticas locais, para derivar a política aplicável quando uma requisição de usuário é recebida. Portanto, a gerente não precisa controlar políticas individuais (ela apenas escreve os gabaritos de políticas que se aplicam a todos os servidores). Os atributos utilizados na avaliação de políticas são locais, tornando cada domínio local (servidor) independente dos demais (não há a necessidade de se compartilhar estado através da rede). A probabilidade da quota de usuário se tornar fragmentada entre vários sistemas (à medida que a quota está sendo configurada) é reduzida por um modelo cooperativo de gerenciamento de atributos, que permite que os domínios locais transfiram quotas não utilizadas entre aplicativos e entre os domínios locais. Isto é, quando o sistema detecta um situação próxima do esgotamento, um processo de transferência de quota é iniciado, localmente ou remotamente, para resolver o problema. A abordagem proposta apresenta um desempenho melhor quando comparada ao modelo de *outsourcing*, além de reduzir a maioria das deficiências do modelo de *provisioning* puro. Nós Implementamos o protótipo de alguns componentes e realizamos alguns testes de avaliação. Os resultados mostram que a proposta é viável e mais apropriada para a computação em nuvem do que abordagens anteriores descritas na literatura.

Palavras-chave: Controle de Uso, UCON$_{ABC}$, Sistemas Distribuídos, Computação em Nuvem, Gerenciamento Cooperativo de Atributos.

Abstract

There is plenty of evidence that UCON$_{ABC}$ is a better access control model for cloud computing environments. It offers a continuous evaluation of policy rules and is designed from the start with attribute mutability in mind. Real world implementations must deal with the fact that policy rules reevaluation must happen in discrete intervals, that is, it is not possible to offer a real continuous reevaluation process. Earlier proposals for UCON$_{ABC}$ architectures for cloud environments opted for outsourcing-based approaches, which has an important shortcoming: the unpredictability of network communication costs. Every access request must be sent over the network and, thus, is subject to a cost that can change over time in unpredictable ways. Therefore, policy reevaluation frequency must be adjusted for the specific environment costs, which becomes an ongoing process in an environment that can grow its size or shrink it on demand. This is further complicated by the need of synchronizing policy reevaluation with a distributed and intrinsically asynchronous accounting mechanism, used to discover how much resource is being used on the remote servers (where policy decision must be enforced). This doctoral thesis describes a decentralized architecture for UCON$_{ABC}$ with better characteristics for cloud computing environments. Decentralization is achieved by employing a modified provisioning approach, thus policy evaluation and decision enforcement happen locally, reducing the effects of unpredictable network costs. Policy management complexity is avoided by using a combination of user credentials and local policy templates, to derive the applicable policy upon receiving a user request. Therefore, the manager does not need to keep track of individual policies (she only writes policy templates that apply to all servers). The attributes used for the policy evaluation are local, making each local domain (server) independent from the others (there is no need for sharing state over the network). The probability of user quota being fragmented over the systems (as quota is being provisioned) is reduced by a cooperative attribute management model, that enables the local domains to transfer unused quota between applications and between local domains. That is, when the system detects a near starvation situation, it triggers a process to transfer unused user quota, locally or remotely, to solve the problem. The proposed approach shows better performance in comparison to the outsourcing model, besides reducing most of the shortcomings of the pure provisioning model. We implemented a prototype of some components and performed some evaluation tests. The results show that the proposal is feasible and better suited for cloud computing than earlier approaches described in the literature.

Keywords: Usage Control, UCON$_{ABC}$, Distributed Systems, Cloud Computing, Cooperative Attribute Management.

List of Figures

List of Tables

List of abbreviations and acronyms

LOW-AM	Low level accounting module
LPAP	Local Policy Administration Point
MID-AM	Mid level accounting module
PAP	Policy Administration Point
PEP	Policy Enforcement Point
PIP	Policy Information Point
PDP	Policy Decision Point
SAML	Security Assertion Markup Language
SG	Security Gateway
SLA	Service Level Agreement
TOP-AM	Top level accounting module
STS	Security Token Service
XACML	eXtensible Access Control Markup Language
XML	eXtensible Markup Language

Contents

1 Introduction

Cloud computing is revolutionizing the way organizations implement their information and communication technology. It creates the possibility of acquiring any amount of computational resources, such as storage space, network bandwidth, and processing power without bothering with how those resources are deployed. The consumer can redefine the amounts of contracted resources as the need arises, and all operations can be made through convenient service interfaces available on the Internet. These features provide great flexibility and ease of use for the consumer organizations (2).

However, as a recent study observed, better access control and accounting methods are essential to provide a trustworthy cloud computing environment (3). Some of the challenges are the creation of access control mechanisms that can dynamically adapt themselves to the cloud environment, meanwhile providing fine-grained policies and accurate accounting. Traditional access control methods are not appropriate for the dynamic and potentially large scale cloud computing environments, they lack the dynamism required to reflect the changes that may take place in the cloud (e.g., they cannot revoke an access authorization or change a user quota in a distributed environment after it was granted).

The usage control model (UCON_{ABC}, (1)) is one of the most advanced access control models found in the literature. Its ability to express fine-grained access control policies that are evaluated in a continuous fashion makes it a better match for cloud computing. Policy decisions can be modified in response to changes to user attributes, resource attributes, or other information used in the policy rules.

Even though the UCON_{ABC} model shows great potential for the cloud computing environment, it presents some implementation challenges. The usage control model does not deal with the practical aspects of distributed systems; the system's designer must define how to deal with all the implementation complexities, such as attribute consolidation, quota management and policy management, evaluation and enforcement.

Earlier attempts at implementing UCON_{ABC} for the cloud opted for centralized approaches (4, 5, 6, 7, 8). In these architectures, a central authorization facility receives authorization requests from enforcement facilities in the remote servers and the accounting data must be collected on these remote servers before a decision is taken.

Such works follow the well-known outsourcing model, which provides an easy to understand architecture with centralized management. On the other hand, that model can be hindered by network overhead and the risk of inconsistency in the attribute consolidation. Two major concerns arise from that model: the scalability of the central components and the possible degradation of the policy reevaluation frequency (due to increased costs required for gathering

accounting information from a large number of servers). As the number of entities increase, the central facility must be prepared to handle increasing numbers of requests. An overloaded authorization facility can degrade the response times seen on the enforcement facilities and, in the worst case: the central facility can become unresponsive and bring all dependent enforcement facilities to an undesirable halt.

Cloud computing resources cannot be controlled in the way it was done in traditional computing systems. An increase in the number of servers allocated to a company may require an increase in the accounting facilities as well. A centralized access control model requires the system's designer to take into account the asynchronous interactions between the cloud services. The authorization facility needs consistent consolidated accounting data to take adequate authorization decisions. The accounting consolidation process must be aware that each remote server may work at a different speed. Thus, it must implement measures to deal with servers sending accounting data too late or not sending accounting data at all (e.g., due to failure on the server or on the network link). Although the outsourcing approach provides a conceptual model that is easy to grasp, implementing UCON_{ABC} with it on a distributed environment forces the designer to deal with many challenges due to concurrency and shared state, as the UCON_{ABC} policy evaluation model is not inherently concurrent.

The implementation of policy reevaluation frequency is highly dependent on the environment at hand, the cost of network communication and accounting consolidation must be taken into account (4). It is hard to predict how these costs will behave when trying to scale up a cloud computing system. The asynchrony between the accounting and authorization mechanisms can lead to exceptional conditions, where some user exceeds the allowed resource quota due to the authorization facility's use of stale accounting data. That is, it is possible for the central facility to authorize a request based on outdated accounting information. This is a challenging problem, if the designer decides to force a synchronized operation on the decision facilities and the accounting mechanisms, the policy enforcement facilities may be blocked for a long time waiting for a reply from the decision facilities. On the other hand, if the designer chooses an asynchronous approach, she will have to design some method to deal with the possible inaccuracies in the accounting data.

1.1 Objectives and Challenges

The main objective of this work is to design an approach to decentralize the implementation of UCON_{ABC} in cloud computing environments. The proposed solution must enable subjects (users) to execute applications on any participating server, potentially using all the resources allocated to the user (i.e., her quota), without creating a strong coupling between the participating servers.

Specifically, each server must function as an independent entity, evaluating usage control

rules and enforcing its decisions. Users might preallocate different quota configurations (i.e. usable resource limits) on each server, when a near starving situation is detected the servers must collaborate to rebalance (i.e., reconfigure) the user quota to avoid the premature end of the application in need.

This thesis' hypothesis is that by decentralizing policy evaluation and enforcement, combined with a cooperative attribute management model, we can promote a predictable UCON$_{ABC}$ implementation for cloud computing, while eliminating the shortcomings of the pure provisioning and outsourcing approaches.

Bellow, there is a list of the specific objectives of this proposal:

- To design a usage control system that reduces the network overhead present in centralized approaches to produce a predictable behavior, despite the number of hosts in the environment.
- To design a cooperative attribute management model enabling the automatic reconfiguration of application quotas in a distributed system, to avoid process starvation when there is available quota elsewhere.
- Reduce the fragmentation of the user quota and improve the efficiency of global quota usage in relation to pure provisioning approaches.
- To design an administrative model for UCON$_{ABC}$ model to enable decentralization of policy evaluation.
- To design a policy management approach to reduce the complexity common to pure provisioning approaches.
- To enable the cloud manager to obtain a consolidated view of the resource usage based on attribute consolidation.
- To integrate the UCON$_{ABC}$ system with the attribute management.
- To evaluate the feasibility of the proposal by prototyping components and performing experiments.

1.2 Contributions

The main contribution of this work is the description of an UCON$_{ABC}$ administrative model based on this thesis' hypothesis, so that we can achieve a better usage control system by using decentralization with a self-configuring approach (i.e., the cooperative attribute management). Our proposal achieves that by creating a model with many policy evaluation facilities, all of them working over the same user attributes, but avoiding the need of synchronization between

servers. Our proposal's model allows the servers to evaluate user attributes in a decoupled fashion while ensuring that attributes remain consistent throughout the environment. We also provide an easier to manage policy architecture by letting users manage their own application resource quotas. This is important because cloud computing systems can grow to large numbers, possibly making it unfeasible to centralize this task on a few individuals. Policy synchronization is also improved, as synchronizing individual policies in large environments can become challenging using traditional approaches. Finally, the proposal ensures that, even though the servers are decoupled from each other, the user is able to take advantage of her full quota on any server, even though her quota might be spread over several servers.

1.3 Text Organization

This thesis is organized in the following chapters:

- Chapter 2 presents the fundamental concepts to understand the rest of this work, and follows with a discussion of the works related to this proposal;
- Chapter 3 details the proposed solution to realize this thesis' hypothesis. It includes an informal description of the proposed approach, followed by an UCON$_{ABC}$ administrative model description to enable the proposal. It finishes with the design of an architecture to implement the model;
- Chapter 4 shows details of the prototyped components. Afterward, it discusses the experiments performed with the prototype and the results obtained;
- Chapter 5 contains our concluding remarks, discussing the results achieved, limitations and future works.

2 Fundamentals and Related Work

This chapter presents a general overview of cloud computing, as well as about policy architectures and the usage control model. These concepts are essential to understanding the rest of this work. It also discusses some of the research works that bear affinity with our proposal. Most proposals found in the literature are based on the outsourcing model, and as we will see in the next sections, they suffer more or less of the same problems: the unpredictability of network communications and lack of scalability from the central facilities.

2.1 Fundamentals

2.1.1 Cloud Computing

The cloud computing paradigm can be traced back to the 1960s, when John McCarthy envisioned the idea of computer utility (9) — general computing facilities that would be provided to end users as general utility services (e.g., electricity). A similar concept emerged in the 2000s under the name of cloud computing, though it meant different things to different people (10). Since then, there has been research efforts to avoid misunderstandings in this area. In this paper, we adopt the NIST definition for cloud computing (11), as we believe it covers all the essential aspects of cloud computing:

> "**NIST definition of cloud computing:** Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

Under this definition, each cloud computing solution can implement one out of three service models:

- **Infrastructure as a service** (IaaS): clouds provide virtual hardware resources, such as processing power, storage, and networking. It presents a very low-level abstraction, allowing the user to deploy software such as operating systems and applications. The consumer does not have control over the underlying infrastructure, though she can control the operating systems and applications deployed over these virtual resources.
- **Platform as a service** (PaaS): providers hide the low-level resources with support services (e.g., security and user management), programming languages, libraries and other tools

that help the customer to develop and deploy applications on the cloud. The consumer's control is limited to her own applications and some configuration settings.

- **Software as a Service** (SaaS): clouds provide complete applications to the end user, who can only modify some of the application's settings. The applications can be accessed in different ways, such as through a web browser or a web service.

There is no obligatory coupling between these three service models. However, it is possible to use less abstract models, such as IaaS, to create clouds of more abstract models, like PaaS or SaaS. These models can be deployed and operated by private organizations, groups of organizations (i.e., communities), the general public or a mix of these situations.

2.1.2 Policy Architectures

In distributed system environments, the access control policies can be managed and evaluated in a variety of ways. The two most popular approaches are described below:

- **Outsourcing:** operates under a client and server model. It contains a central authorization facility (also called the reference monitor) that responds to policy evaluation requests from all the enforcement facilities (i.e., the resource guardians). Every access attempt made by a subject is captured by the resource guardian and triggers a policy evaluation request that is sent to the (remote) reference monitor. The guardian waits for a policy evaluation decision and, upon receiving it, performs the actions required to enforce it (12). The main advantage of the outsourcing model is the simplicity of the policy management and guardian implementation. However, it has the disadvantages of communication overhead and its fragility, as the reference monitor can become a single point of failure. In a cloud computing environment, its centralized approach can make it hard to achieve the scalability expected of cloud computing environments.

- **Provisioning:** its operation is based on the configuration of policies at the place where the controls will be enforced, the resource guardians. The policies are written on a central policy administration point, however, the enforcement mechanisms receive the policies to be used during initialization. These policies are stored in a local repository and the resource guardian requests policy decisions to a local reference monitor (13). The provisioning model presents the advantages of being robust, as it has no external dependencies, and of having a reduced number of requests going through the network. On the other hand, it is harder to keep policies synchronized on a distributed environment because the central policy administration point must keep track of every policy in the system.

2.1.3 Usage Control Model

The usage control model was initially described on (14) and further detailed on (1). The motivation for its development was the need for unifying modern access control techniques under a coherent formal model. The key differences from earlier access control models are the emphasis on continuous decision evaluation, policy rules based on predicates over attributes, and the possibility of describing modifications to those attributes in the policy itself (i.e., the so-called attribute mutability).

The conceptual model became known as the UCON_{ABC} model, where *A* stands for Authorizations, *B* for Obligations, and *C* for Conditions. These are the main types of predicates that may be used to create usage control policies. To evaluate these predicates, UCON_{ABC} uses attributes related to subjects (e.g., the user or application requesting some action), objects (i.e., the resource being protected), and the environment in which the usage takes place. Predicates are categorized according to the type of attributes used on its evaluation, and on the logical function it plays inside the policy:

Authorization is a predicate over subject and object attributes. It evaluates whether a given subject has the required rights to perform the requested action over the target object. For instance, an authorization predicate may evaluate whether the subject has the right for opening a file or for executing an application;

oBligation uses subject and object attributes as well. However, its logical function is to verify if some obligatory procedure is performed. These obligations can be any type of action to be performed by the requesting subject or someone else, such as filling an electronic formulary, depositing some credit, keeping a window open on the web browser, and so on.

Condition differs itself from the previous types of predicates for using only environmental attributes, such as time of day, the level of system load, geographical locations, etc. That is, conditions does not use subject and object attributes.

An illustration of the usage control model can be seen on Figure 1. On it, we can see that a usage decision is taken by combining the attributes of subjects and objects, usage rights, and the three types of predicates. Environmental attributes are not shown because such attributes cannot be changed by the usage control model itself.

UCON_{ABC} establishes that these type of predicates may be combined in many ways, and may be evaluated in different stages of the usage procedure. In this aspect, the usage control model differs from the traditional access control models that understand the policy evaluation decision as something that happens only once, before the actual access takes place. This change was required to deal with modern environments, where the actual context in which a given decision was taken may change at any time, quickly rendering the decision obsolete.

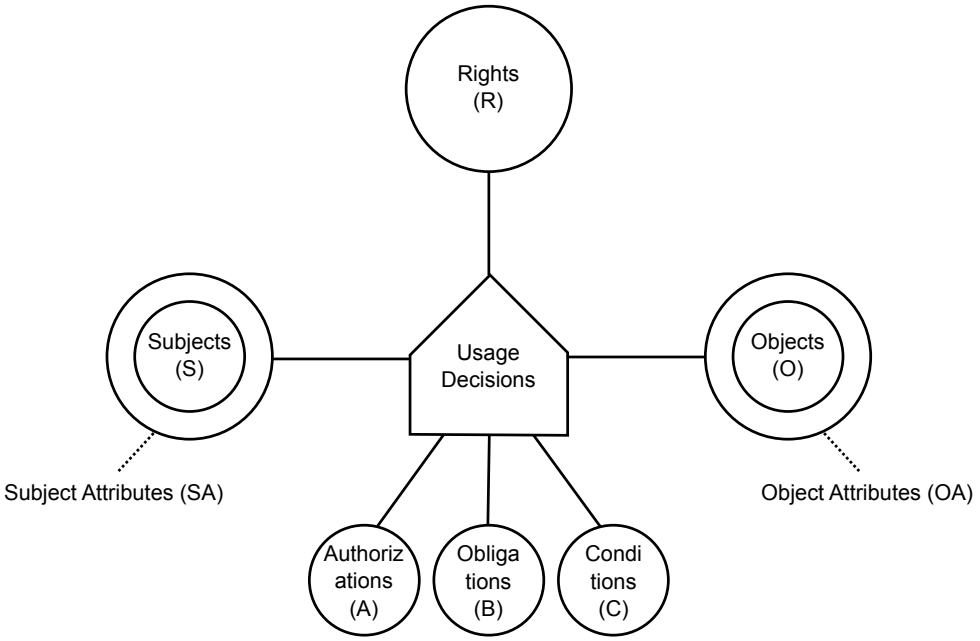

Figure 1 – Overview of UCON$_{ABC}$ model components, based on the original from (1)

Therefore, usage control predicates may be further categorized according to the point in time in which these predicates get evaluated. There are three possibilities for applying a predicate:

Pre controls are predicates that get evaluated before the actual usage starts. These predicates are equivalent to traditional access controls, as they are evaluated only once.

Ongoing controls are predicates that get evaluated during the usage process itself. In the conceptual model, this ongoing evaluation happens continuously and a new decision is taken whenever some attribute change changes the truth value of a predicate. Therefore, it is possible to revoke the access during usage itself.

Post controls these type of predicates are actually used to change some state after the usage process ends. As will be seen bellow, it would not make sense to describe actual controls at this stage, thus it is used only for attribute updates.

The usage control model can reflect modifications in attributes on the policy decisions during runtime (i.e., ongoing). This gives great flexibility for expressing highly dynamic usage control policies. However, it would not be expressive enough if these attribute modifications were only possible outside of the usage control predicates. Without controlled updates, a great deal of modern access control approaches would fall outside of the UCON$_{ABC}$'s expressive reach.

Predicates for attribute mutability were designed to solve this limitation. They are called update predicates and are used to change the value of subject and object attributes according to some arbitrary expression. It is possible to define a predicate to reduce the number of times a

	0 (immutable)	1 (pre-update)	2 (ongoing-update)	3 (post-update)
preA	Y	Y	N	Y
onA	Y	Y	Y	Y
preB	Y	Y	N	Y
onB	Y	Y	Y	Y
preC	Y	N	N	N
onC	Y	N	N	N

Table 1 – The 16 basic ABC models ((1)).

user is allowed to play a song or to increase its trust level if she is behaving well, as well as to reduce the amount of available resource (i.e., changing an object attribute).

Together with authorizations, obligations, and conditions, updates may be applied at any point in time. However, it is important to make clear that only updates are actually useful on the *post* stage, because the updates may affect future usage requests. It is also important to notice that updates can not change environmental attributes, which are outside of the model. Therefore, the combination of these predicates with the possible stages in which they get applied gives rise to the 16 basic ABC models:

As can be seen on Table 1, authorizations before (preA) or during (onA) the usage can be combined with immutable attributes (i.e., those that can not be changed by update predicates) and mutable attributes. The same is valid for pre and ongoing obligations (preB and onB). We can observe that there is no postA or postB, as that would not make sense. The conditions (preC and onC), on the other hand, can not be coupled with mutable attributes.

It is possible to describe any of the traditional access control models by combining the above ABC models. The highly expressive power of UCON$_{ABC}$ allows writing discretionary, mandatory and role-based access controls, as well as modern access control policies.

From the architectural point of view, the usage control reference monitor can be implemented on the client side, on the server side, or a combination of both approaches. Conceptually speaking, the reference monitor contains two facilities: one for policy decision, and one for policy enforcement. These components are subdivided in modules that handle that many aspects of the usage control process, such as attribute updates, usage accounting, obligations monitoring and so on.

Being a purely conceptual model, the UCON$_{ABC}$ does not define any of the technological details of its implementation. The implementor must select what type of architecture to use, how to write its policies, and how to implement important aspects such as continuous evaluation (e.g., will it be time-based or event-based), how to monitor obligations, how to update attributes and so on. As we will see further, this gives rise to many different approaches to implement UCON$_{ABC}$, with varying levels of compliance with the theoretical model.

2.2 Related Work

This section describes some of the most relevant works related to the implementation of the usage control model. From the limitations of these works, we derive the motivation for the proposed solution.

2.2.1 A UCON$_{ABC}$ Resilient Authorization Evaluation for Cloud Computing

On (4), we designed an architecture for the resilient evaluation of usage control policies in cloud computing environments using the outsourcing model. The resilience property, in this context, refers to the capacity of the proposed model to tolerate inconsistencies in the perceived attribute values used in the policy evaluation process, which may differ from the actual values read from the resources.

It is possible for the policy evaluation facility to perform policy evaluations based on inconsistent values. This is a direct result from the distributed and asynchronous model employed by the authors. The policy decision takes place in a central facility, that periodically receives accounting updates from the remote hosts where the resources are being used. The delay between reading the attributes in the remote systems, sending them to the central facility, consolidating and then evaluating the policy, may cause exceptional situations in which a policy violation is only discovered after a certain period of time.

The authors implemented an approach to cope with these possible exceptions: the model reserves part of the resource quota to be used as a safety margin. It allows usage sessions to exceed their actual quotas up to a certain limit, as long as a global constraint is fulfilled: as long as the sum of all used resources is kept under a certain limit, the situation is considered normal. The system manager is alerted whenever this constraint is violated, enabling her to take corrective actions.

The proposed architecture seeks to provide scalability to the central facilities by using tuple spaces. It allows the decoupling of the remote hosts from central entities and to handle the high demand created by the remote systems.

The paper describes a prototype implementation and performance evaluation. The researchers investigated the influence of network communications and message sizes on reevaluation frequency. Their conclusion was that the minimum time between reevaluations is highly dependent on the environment used, the message sizes and the number of requests.

However, we did not discuss the possibility of accounting messages drifting in time. That is, it was assumed that every involved system would send accounting information at the same time. It is possible that, due to the differing system loads, some messages will get progressively delayed and lead to unpredictable results. Under heavy loads the accounting agent may be unresponsive for a long time, while the user application is still executing and using resources,

thus violating the usage control policies far beyond from the tolerance provided by the resilience property of the model.

2.2.2 Usage Control in Cloud Systems

In the papers by Lazousky and colleagues (5, 6) a implementation of the usage control model is discussed in details. One of the main contributions of the authors is the definition of extensions to the XACML language to provide language constructs to express usage control ideas, therefore it is possible to express when the conditions and obligations must be evaluated.

The authors also approached the challenging problem of dealing with attribute changes. They introduced the concept of attribute retrieval policy to specify when to collect fresh attribute values and to consequently trigger the access reevaluation. The attribute retrieval policy is formed by a set of conditions that must hold true before an attribute can be fetched, when this happen, a component (the PIP) collects the attribute and pushes it to the policy evaluator.

The proposed framework intercepts every access request (e.g., to create, suspend, reactivate or delete a virtual machine) and determines whether the request is allowed or not. The authorization framework is able to cancel an ongoing usage session if a policy violation is detected.

These usage sessions are controlled by instances of the authorization service, which are created on each new usage session. The usage session is destroyed when the violation of the policy happens or the usage session ends normally. As the authors observed, the authorization service instances of different usage sessions run concurrently. However, the prototype implementation does not allow for cooperation between authorization service instances.

The proposed architecture works in an outsourcing manner. Therefore, it is safe to conclude that it suffers from the unpredictability present in outsourcing models. The authors did not cover the concurrency problem, not even from a conceptual point of view. Thus, their proposal lacks generality for use in cloud computing environments.

2.2.3 A Usage Control Based Architecture for Cloud Environments

Tavizi and colleagues (7) focused on the architectural details of a usage control mechanism. That is, they were concerned with organizing the function of each component, like policy decision points, decision enforcers, policy information points, while providing semantics closer to the usage control model.

Their proposed model is based on the outsourcing model, closely mimicking the XACML proposed architecture. One of the differing aspects of their proposal is that to provide the best security possible with the least computational overhead, the proposed model performs continuous control according to the sensitivity degree of the attribute.

The sensitivity is used to decide which type of attribute retrieval will be used. Highly sensitive attributes are required to be pushed actively to the policy decision facility whenever a change is detected. Lower sensitivity attributes can be pulled by the policy decision facility in a periodical fashion. The authors also provided an event handler facility, which may be configured to deal with various types of events that could trigger a policy evaluation.

A prototype was implemented using XACML language. The authors made some extensions to the language to enable the expression of some usage control constructs. However, no detail was given about any experimental results or the implementation itself.

2.2.4 Access Control of Cloud Service Based on UCON

An access control architecture based on the UCON$_{ABC}$ model was proposed by Danwei and colleagues (8). The architecture is different from the previous works for using an unusual approach in which the client and server contain a negotiation module. They modified the UCON$_{ABC}$ model to enable the dynamic negotiation of access levels.

According to their proposal (called the Nego–UCON$_{ABC}$), the user has the possibility of choosing another access option through a negotiation process. Therefore, in certain situations, the user can still get some access to the resources, instead of being promptly rejected when an authorization credential is insufficient.

The authors discussed a detailed architecture to implement the proposed model. They described how SAML can be used to transport the authorization attributes and the interactions between the various components. The proposed architecture follows the outsourcing model. No implementation details were given.

2.2.5 An Administrative Model for UCON$_{ABC}$

Farzad Salim and colleagues (15) argued that UCON$_{ABC}$ administration must be done through the management of attributes, which they termed the administrative model. It must define the meaning of attributes, valid attribute sources and who can manipulate these attributes. This is also valid for object attributes as well.

The administrative model is connected to UCON$_{ABC}$ because the former defines attributes and the latter employs them to evaluate usage control policies. On the authors proposed administrative model, properties and rights are defined by attributes, which are formed through assertions made by subjects or other entities with administrative capabilities.

The authors organized their model in a two-layer structure containing a peer model and an authoriser model. The first provides an expressive and unrestricted environment where every subject can make assertions about other subjects and objects. This allows the identification of who can modify or delegate properties and rights. The authoriser model determines whose

assertions are to be used for the decision process. The authoriser model depends on a policy agreed upon by every participant of the system.

The paper discusses trust aspects of these authoriser policies, including by using a concrete language (SecPal, (16)). The authors aimed at eliminating the assumption of a single administrator who issues attributes and the authorisation policies. They unified all attribute types (mutable and immutable) under one category, which defines mutability as a condition that may change under different circumstances.

Although the authors used a real language to express concrete examples, there was not prototype implementation to demonstrate the feasibility of the proposed model. Therefore, the authors described an interesting concept which can provide some insights into the administrative details of UCON$_{ABC}$. In our proposal, we developed an administrative model to employ temporary attributes to allow concurrent policy evaluations with cooperation between policy evaluators.

2.3 Conclusion

Cloud computing is a new type of distributed system. It was enabled by advances in technologies like hardware virtualization and the standardization of networking protocols. As we saw, cloud computing platforms can offer different levels of abstraction: the lowest level (infrastructure, IaaS), the intermediary abstraction (the platform, PaaS) and the highest level of abstraction (the software, SaaS).

We discussed the policy management architectures commonly seen on distributed systems: provisioning and outsourcing. We also described the usage control model (UCON$_{ABC}$) and its defining characteristics.

The related works make clear that UCON$_{ABC}$ is a very relevant topic for distributed systems and especially for cloud computing environments. As we said earlier, most proposals are based on the outsourcing model, which have clear disadvantages for the cloud computing environment.

Most works are focused on expressing usage control rules in some language extension, giving little attention to the other implementation details. Other authors focused on conceptual aspects, giving no details of possible implementations. Only one work took the time to address the shortcomings of the outsourcing model and, even then, there is still margin for inconsistent behavior.

3 Proposed Solution

The drawbacks mentioned in the previous chapters motivate the reduction of variable costs from the policy reevaluation process. This is required because if we want to implement a usage control system that behaves in a predictable fashion, the policy reevaluation frequency must remain stable no matter the number of servers present in the cloud computing environment.

We must also eliminate any shared state during policy evaluation. That is, considering we can have two or more policy evaluation facilities in the environment, those facilities must not share any state affecting the outcome of policy evaluation. This is a fundamental aspect; otherwise, we would create a strong coupling between servers. Decentralizing evaluation would not be any better if we were forced to synchronize state for policy evaluation.

Special care was taken to enable attribute mutability in a consistent fashion. This was achieved by partitioning the resource limits (quotas) among local domains, in a way that each local domain controls a subset of the whole (global) quota. Therefore, it is possible to provide attribute mutability in a controllable fashion, propagating updates between local domains and the administrative domain as needed.

In this work, we adopt the concept of quota as being a share or part of the total amount of a given resource available. Thus, we have a global quota that is the sum of all the defined user quotas plus the remaining amount that has not been allocated to any user yet. As will be described further, a given user quota might be further segmented in application quotas at the user discretion.

The decentralization of the authorization model (provisioning) can produce gains in performance by eliminating the network overhead present in centralized models. It shows potential for better scalability, because when each server does not depend on external entities to evaluate and enforce policies, the usage control system is expected to scale linearly with the number of servers without performance degradation. In a decoupled model each authorization facility can take decisions independently, which leads to a more adequate authorization system.

However, to enable the UCON$_{ABC}$ to be decentralized, we must first define how user and object attributes will be handled and used during policy evaluation. We must define an administrative model that provides these details, and also what constraints must be applied to it to ensure that decisions will be consistent. This new administrative model is discussed in details in the next section.

We designed an architecture inspired by the provisioning model to implement the new administrative model. The architecture provides decentralized usage control focused on reducing the latency present on the outsourcing approach. The architecture uses local policy evaluations to

significantly reduce network latency. This increases the predictability of the system's behavior, despite the number of servers in the cloud. The servers perform the local enforcement of policy decisions.

The policies themselves are generated from user credentials and policy templates. This provides a way to configure individual policies for each user application, each one with a custom defined resource quota, derived from the main user quota. Therefore, each policy is individualized and it does not share direct attributes with other policies.

As the application quotas are preallocated and configured on the server, this could lead to the main user quota becoming fragmented over several servers. To avoid this, we designed an attribute reconfiguration model that allows the servers to detect starvation conditions and to cooperatively rebalance the quota configurations previously defined by the users. This provides for overall better quota usage and better user experience, as the applications will be able to make use of all user quota before being denied further access to resources. This attribute reconfiguration happens in an asynchronous fashion, without negative impact on the policy reevaluation frequency.

The use of user credentials to provision application quotas was needed to avoid the high complexity that would be present in a pure provisioning approach. That is, by letting users define how much quota they want to allocate to each application, and then deriving the policy from a template combined with the user credential, we free the manager of tracking where the policy must be provisioned, of writing policies for each application, and from having to synchronize those policies every time an application quota changes. Therefore, the task of the system manager is much simpler.

3.1 Model

This section presents our proposal for an administrative model for the application of UCON$_{ABC}$ in distributed systems, more specifically those that are representative of cloud computing systems. We describe how some UCON$_{ABC}$ features might be used to create a model in which concurrent policy evaluations may take place without requiring state sharing between the decentralized policy evaluators.

As it has been shown on previous works, centralized approaches cause the policy reevaluation frequency to be highly coupled with the environment size. By enabling UCON$_{ABC}$ to be decentralized, we allow it to be implemented in a fashion that is scalable without coupling policy reevaluation frequency to the number of entities in the cloud.

Controlling resource usage in cloud computing environments is a complex task that cannot be solved with traditional access control models. One of cloud computing innovative features is its capacity of redefining resource allocation amounts dynamically, called its resource

elasticity. On one hand, it provides great flexibility for the consumer; on the other hand, it exposes the shortcomings of traditional access control models when faced with a continuously changing environment. Those models were designed based on the assumption that once a decision is taken, it would not need to be reconsidered. Therefore, once a user was authorized, that decision could not be taken back.

The limitations of traditional access control models can be clearly seen on the security mechanisms implemented by current cloud computing offerings, such as RedHat OpenShift (17), Pivotal Cloud Foundry (18) and Heroku (19). These cloud providers offer only static resource allocation, based on a predefined set of configurations. That is, the consumer is allowed to acquire, for instance, a virtual machine in a limited number of configurations. It is not possible to define a custom configuration nor to redefine it dynamically. Any changes to virtual machine resource configuration require a full restart, possibly involving installing the consumer's applications again. The only possibility for the user to dynamically acquire more resources is by initializing more virtual machines.

If the underlying access control mechanisms were more flexible, the user would be able to increase or decrease the resource amounts for its virtual machines, as the current virtualization technologies already allow it (it is called ballooning, see (20)). Therefore, the consumer is stuck with a situation that either might lead her to acquire too much resource or too little resource.

The usage control model, UCON$_{ABC}$, was designed to unify many research ideas that were looking to overcome the limitations of traditional access controls in the context of modern environments. The main aspects of usage control are the continuity of policy evaluation and attribute mutability. It means that, unlike traditional models, UCON$_{ABC}$ is capable of revoking a decision at any point in time if a policy rule violation is detected.

This new model is being proposed as a better option to implement security mechanisms in cloud computing environments, as it is able to express traditional access control restrictions, and is better suited to deal with the dynamism of the cloud. UCON$_{ABC}$ does not require the resource limits to be static; any change to those limits can be promptly reflected on the policy reevaluations affecting current usage sessions. This allows the consumers to allocate resources as they see fit, unlike having to fit her applications in a predefined configuration.

The original UCON$_{ABC}$ model, however, adopts a centralizing approach. Previous attempts to implement usage control architectures for cloud computing environments that faithfully followed the original model ended up with a centralized structure as well. The results were that the performance of the usage control systems is highly dependent on the environment in question. The policies reevaluation frequency must be set specifically for each environment size, and any change in the number of components involved may require further analysis and reconfiguration of this frequency.

The natural alternative to this problem is the adoption of a decentralized approach to

the evaluation and enforcement of usage control. The original model does not provide for concurrent policy evaluations using the same attributes on distributed evaluation facilities. That is, in the original model, it is not possible to express a distributed usage control to be applied and evaluated, for instance, on two virtual machines involving the same user and the same authorization attributes. In the original UCON$_{ABC}$ model it is only possible the expression of unified policies that must be evaluated by a conceptually centralized entity that is able to continuously observe the whole environment.

3.1.1 Operational Model

Before we discuss how the UCON $_{ABC}$ can be extended so that it can be implemented in a decentralized manner, it is fundamental to present the objectives of the new model. In the following paragraphs the main entities involved in the model will be presented with the constraints that must be respected by it.

The proposed model contains three types of entities:

Provider: It is responsible for providing the computing resources of the cloud. For the context of this proposal, the provider manages infrastructure resources (i.e., storage, network bandwidth, processing power, etc.);

Consumer: is the entity that acquires computational resources from the provider for use in her organization. The consumer allocates portions of the acquired resources for the execution of user applications;

User: also known as the subject, is the entity that performs the effective use of resources by running applications on the infrastructure acquired from the provider. The user has a dependency relationship with the consumer, as the latter defines each user's resource allocations (quotas).

It is assumed that the provider takes no notice of how the resources provided to a particular consumer will be used. The provider is only concerned with managing the infrastructure, letting the consumer define how the allocated resources will be used. On the other hand, the consumer determines how to best allocate the resources acquired for its users (whether they are employees or customers of the organization). With this in mind, we can formulate the first model definition:

Definition 1. *Global quota of a resource – $GQ(ResID)$*

- $ResID \in \mathscr{R}$ *where* $\mathscr{R} = \{$*resource identifiers*$\}$

It represents the total amount, or global quota, of a resource that is identified by ResID, which was acquired by the consumer from the cloud provider. It is a fundamental system constraint that the sum of the use of all consumer users do not exceed this value, as this could result in losses to the consumer in the form of additional charges by the provider.

One can identify the conflict of interest in resource allocation control if it was delegated to the provider. If we consider that the provider is benefited from the inefficient use of contracted resources, it is clear that the usage control of users should be implemented independently from the provider. In addition, for the provider to control consumer users a strong integration would be needed between different cloud categories: in the current proposal, we assume that the provider does not have knowledge of the existence of the users, it only recognizes the consumer.

Remark 1. *In a previous research paper, we identified the lack of integration between different cloud service models (21). To summarize, each cloud service model is isolated from the other models. For instance, the IaaS provider is not aware of the PaaS or SaaS user identities. Therefore, it is not possible to implement fine–grained resource controls outside one's own service model without some involved form of integration. Appendix A, on page 79, provides further details on the challenges of integrating different cloud service models and discuss a possible approach.*

It is the consumer's responsibility to implement the mechanisms to control user applications in order to prevent the global quota $GQ(ResID)$ of each resource from being exceeded. However, it is not possible for the consumer to anticipate the number of resources that should be allocated to each user application, or where in the cloud computing environment the application will be instantiated. For these reasons, the consumer is limited to setting the maximum amount derived from $GQ(ResID)$ that should be reserved for a particular user. This brings us to our second definition:

Definition 2. *User quota* – $UQ(UserID, ResID)$

- $UserID \in \mathscr{U}$ *where* $\mathscr{U} = \{$*user identifiers*$\}$

The user (identified by UserID) quota for a given resource (ResID), is a part of the global quota $GQ(ResID)$ *which was reserved for the given user, allowing it to make use of the resource in the contracted environment without exceeding this value, as she sees fit. Creating a user quota should respect the following constraint:*

$$UQ(UserID, ResID) \leq GQ(ResID) - \sum UQ(id, ResID) \; \forall \; id \; \in \; \mathscr{U} - UserID \tag{3.1}$$

The user can not just use her $UQ(UserID, ResID)$ directly in a cloud environment, as this would require the ability for all the systems involved to consistently share this attribute. That is, it is necessary that all involved systems be strongly coupled to ensure that the accounting of the various user applications does not exceed the permitted value. The computational cost and complexity involved make this type of approach unfeasible.

Therefore, the user must allocate a portion of her $UQ(UserID, ResID)$ for each application she wants to use. That is, the user must estimate the number of resources that each application instance will use and establish a reserve for each application. This leads us to the third definition:

Definition 3. *Application Quota – $AQ(AppID, ResID)$:*

• *$AppID \in \mathscr{A}$ where $\mathscr{A}$ = {user application identifiers}*

The application quota represents an amount of a resource (ResID) which is allocated by the user for a specific application instance (AppID). This value is derived from $UQ(UserID, ResID)$. The definition of AQ must comply with the following constraint:

$$AQ(AppID, ResID) \leq UQ(UserID, ResID) - \sum AQ(id, ResID) \; \forall \; id \; \in \; \mathscr{A} \qquad (3.2)$$

The proposed model should allow the consumer's manager to set the amount of resources that each user will be able to consume ($UQ(UserID, ResID)$), the manager should not worry about the way the user allocates these resources to her personal applications. Furthermore, the manager should not be required to write individual policies to be provisioned on each system where the users can run their applications.

Each user must decide how to use her resources. After defining how many resources each application will receive, the user must request the consumer to issue credentials containing the requested values to enable the application to be instantiated in any system of the cloud computing environment. At this point, the consumer should implement the accounting restrictions mentioned, i.e., to ensure that the sum of application quotas $AQ(AppID, ResID)$ of the requesting user will not exceed the user quota $UQ(UserID, ResID)$ defined.

The credential issued by the consumer must contain the definitions of $AQ(AppID, ResID)$ requested by the user. The system that receives the request to run the user application, with the credential issued, must perform the derivation of a usage control policy from a template policy. The resulting policy is customized to the application in question. The evaluation of the derived policy and the enforcement of the decision must take place in the local system, following the original UCON_{ABC} model. Thus, each user application will have its own local usage control policy.

Definition 4. *Policy Derivation:*

The usage control policy should be derived at run time from a policy template. The controls (i.e., $UCON_{ABC}$ predicates) are described on the template and must be configured with the attributes of the presented credential.

The proposed model assumes that users should be able to run applications concurrently in different systems. This means that the definition of $UQ(UserID, ResID)$ may become fragmented, that is, allocated inefficiently in various systems where the user has applications being executed.

The inefficiency stems from the fact that under certain circumstances, the application quotas can be set with imprecision, generating low resource situations for some applica-

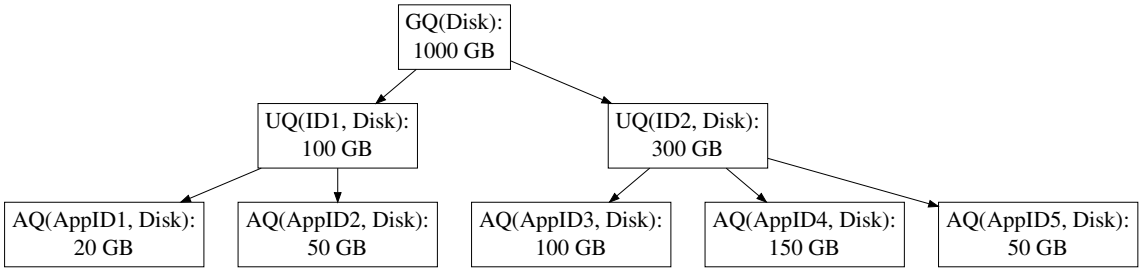

Figure 2 – An example of quota hierarchy

tions and excess of resources to other applications. This is possible because each definition of $AQ(AppID, ResID)$ is made as an estimate which may have been too optimistic or pessimistic.

With the aim of improving resource usage efficiency, the user must allow the application quotas to be reconfigured dynamically. This can happen between application instances on the same system or on different systems. The purpose of this requirement is to ensure that applications with an overestimated $AQ(AppID, ResID)$ may transfer part of it to applications that may suffer from lack of resources, thereby increasing system efficiency. Furthermore, it is important that the reconfiguration process occurs without the need of user intervention:

Definition 5. *Quota Reconfiguration:*

The system should enable automatic reconfiguration of application quota in cases explicitly allowed by the owner of the application, when the use of resources by the application is approaching the limits of its configured quota. The goal is to reduce quota fragmentation due to the unpredictability of user application requirements.

With the previous definitions in mind, it is possible to describe a hierarchy of resource quotas in accordance with the level of granularity. The Figure 2 illustrates this hierarchy. At the top is the *GQ* for the resource *Disk*, with assigned value of 1000 GB of storage. The consumer has decided to allocate 100 GB for user (*ID1*) and 300GB to another user (*ID2*), leaving 600 GB available to allocate to other users. User *ID1* has allocated 20 GB and 50 GB respectively for applications *AppID1* and *AppID2* while the user *ID2* has allocated all its quota for three applications (*AppID3* 100GB, *AppID4* 150GB and *AppID5* 50GB).

In this scenario, if the application *AppID1* needs more resources to complete its task, the user can transfer part of the free quota (30GB) present in $UQ(ID1, Disk)$. User *ID2*, on the other hand, has no quota available in $UQ(ID2, Disk)$: in this case, for instance, *AppID5* may require more resources to complete its task, and because *AppID4* has a large amount of unused quota, it could allow a portion of its quota to be transferred to correct the situation.

The model must comply with the above definitions and promote decentralization of activities (i.e., decoupling), in order to facilitate the administration of policies and user attributes

regardless of the environment size. To meet this goal, the application of usage control, including the evaluation of policies, should be performed at the local level, eliminating the unpredictability factor intrinsic to the remote communications that could negatively affect the performance of the model. In addition, the model should ensure the consistency of attributes, preventing the occurrence of repeated use of the same definition of $AQ(AppID, ResID)$ (potentially causing incorrect accounting, whether by accident or by malicious activities).

3.1.2 UCON$_{ABC}$ Administrative Model

The previously proposed objectives can be achieved by designing an administrative model to extend the operation of the original UCON$_{ABC}$ model. At the heart of this new model is the concept of attribute mutability provided by UCON$_{ABC}$.

This attribute mutability was conceived as a pillar of the usage control model (22). It makes it possible to perform user or object attribute modification at any time in a usage session (i.e., before, during or after the actual use), providing great flexibility to the usage control model.

There are two categories of mutable attributes that can be used in the usage control policies:

- **Persistent attributes**: are those that have a persistent nature and that can be shared over time in various usage sessions. A persistent attribute example is a prepaid card that is decremented on each use and that can be used several times.
- **Temporary attributes**: are those that exist only during a single usage session and, generally, are not shared with other sessions. When the session is finished, the temporary attribute ceases to exist. An example of a temporary attribute is the duration of a phone call, which is continuously updated during the call, but it ceases to exist as soon as the user hangs up.

Persistent attributes and temporary attributes can be related to each other, i.e., temporary attributes can be used to update persistent attributes. In the case of a prepaid phone card, the credit (persistent attribute) is updated after calculating the cost of the telephone call based on call duration (temporary attribute) and other cost factors of the telephone company. This allows different administrative models to be implemented in UCON$_{ABC}$, for instance, the *Dynamic Separation of Duty*, as can be seen in (22).

In this work, we developed a new administrative model for UCON $_{ABC}$ meeting the requirements set out in the previous sections. We used temporary and persistent attributes in a manner not envisioned in the original model. The objective of this new model is to enable the decentralization of the usage control evaluation, which otherwise could not be expressed in the originally proposed administrative model.

In our model, we consider the global quota *GQ* and user quota *UQ* as persistent attributes that are used use in several usage sessions. The application quotas *AQ*, on the other hand, are considered temporary attributes because they only affect a single usage session and are extinguished when the application is terminated.

Persistent attributes are not directly used during the usage control of an application. That is, they exist for the high–level management, ensuring that the sum of the quotas reserved for users does not exceed the global quota and the sum of application quotas of a user does not exceed her own quota. As these attributes are not used during the evaluation of policies, the constraints from Definition 2 and Definition 3 can be enforced at the time the user and application quotas are being created.

The infrastructure must be controlled with temporary attributes (the application quota *AQ*), allowing each application to be independent from the others. When submitting a request to instantiate an application in any system, together with the *AQ(AppID,ResID)* definition, the resource reservation is guaranteed for the desired application.

At the local level, the usage control model behaves the same way that was defined in the original UCON_{ABC} model. This is, policies are formed by predicates over sets of user and resource attributes, with the distinction that the attributes for users are temporary and exist only in the scope of the local system.

The values contained in the *AQ(AppID,ResID)* definition serve to derive the application's usage control policy from a local policy template, in which predicates must be configured before they can be evaluated. The amounts of each application quota become limits for local use, which will be matched with the application's accounting attributes.

The mutability of attributes is used in a different manner from the original vision from (22): the derived policy for the application must contain update predicates that change the persistent attributes on the administrative domain. Specifically, the attributes related to the user's quota utilization. Periodically, an update of the current quota usage is made and relayed to the administrative domain. At the end of the usage session, any quota amount that is left over can be reclaimed for later use for other applications.

Although the user quota might be updated by events from the local domains, it does not negatively impact the performance of the local evaluation facilities. That is, the user quota is not a state shared among the local domains. Each local domain needs only the application quotas provisioned prior to application execution to successfully evaluate the local policies. The updates sent to the user quota are there to improve the efficiency of user quota usage.

Although application quota attributes are considered temporary and should be used only in a single session, it does not mean they cannot be changed due to other concurrent sessions of the same user. The proposed dynamic reconfiguration of quotas is based on this assumption. It allows for independent local systems that can automatically transfer part of a user's quota as

needed. These transfers take place using conditional update predicates.

When a resource usage limit is achieved in a given *AQ(appID,ResID)*, an update predicate will transfer a part of the application quota from the system with the larger amount of unused quota, or from the user quota on the administrative domain, to the system where the starving application resides. If we consider a requesting application (*req*), a donor application with unused quota (*pro*) and a given amount of quota (P) to be transferred, we have the following operations that must be performed atomically: $AQ(req, ResID) \leftarrow AQ(req, ResID) + P$ and $UQ(UserID, ResID) \leftarrow UQ(UserID, ResID) - P$ (when there is free quota on the UQ) or $AQ(pro, ResID) \leftarrow AQ(pro, ResID) - P$ when the quota is transferred from another application. Priority is given for the second option (transfers between local domains). As will be better explained further, this aims at improving efficiency in quota usage.

This means that the local usage control policy contains a trigger to start the process of quota reconfiguration. When the user application reaches a predetermined amount of the application quota, a distributed attribute update must be performed in order to transfer an amount of quota (either from the user or another application quota), preventing the initial application from running out of resources.

The reconfiguration process only rewrites temporary attributes from a single user, in order to improve quota usage, reducing the fragmentation of user quota and improving user experience. Conceptually speaking, the reconfiguration process is represented only by the execution of a conditional update predicate that happens to be implemented as a distributed operation.

3.2 Architecture

The decentralization of the UCON$_{ABC}$ for cloud computing requires a design flexible enough to achieve the scalability common to these environments without compromising the security of the system. This section presents that design, answering the following questions:

- How does the architecture manage the policies among the participating servers?
- How does the architecture ensure a predictable UCON$_{ABC}$ reevaluation frequency?
- How is the resource allocation consistency enforced?
- How does the architecture avoid quota from becoming fragmented?
- How does the architecture manage resource allocation for the users?

3.2.1 Organization

To achieve the desired objectives, the architecture was designed with some features in mind: the use of local authorization and enforcement facilities, policy management based on the

use of user credentials and policy templates, and the use of an auto-configuration approach to allow the application hosts to reconfigure and to improve quota usage.

Remark 2. *In this work, a user credential is a digital document that carries various authorization related data, resembling security capabilities(23), though more general. A user credential does not automatically imply a permission. It must be analyzed by the usage control system, in light of the applicable policies, before any action can be permitted.*

An overview of the proposed architecture's organization can be seen on Figure 3. We designed it around the idea of two basic domain types: the administrative domain and the local domain.

- **Administrative domain** is used to provide high-level management services, such as user attribute management, policy management, issuing and tracking of user credentials and consolidation of accounting information.
- **Local domains** are the servers in which user applications are executed. A local domain contains a usage control system based on UCON$_{ABC}$, coupled with local accounting and attribute reconfiguration mechanisms, to control resource usage and avoid resource allocation fragmentation.

Each local domain is isolated from the rest of the local domains. That is, it does not share state (usage control information) with other local domains. This is a fundamental feature to preserve local domain autonomy and, thus, producing a predictable performance for the usage control system. Each local domain is a host equipped with special environments to execute user applications (i.e., application containers), controlled by usage control policies, and is able to manage local attributes with a LOW-AM, reconfiguring those attributes as needed.

We use policy templates combined with user credentials to derive the applicable policies on the local domain, making it self-sufficient, i.e., everything needed for policy evaluation and enforcement is available in the local domain.

This self-sufficient approach, however, has an important challenge: application quota may be spread (fragmented) over several local domains. Thus, some applications could starve without the use of some mechanism to enable the reconfiguration of the user's spare quota from the local domain or even from other domains (including the administrative domain itself).

The administrative domain and local domains in conjunction provide a distributed and cooperative attribute management service. It enables the automatic reconfiguration of resource quotas among the local domains, thus reducing fragmentation of the user quota (e.g. small remaining quota in different user sessions) and improving the global quota usage efficiency. Users might also have a better experience as this approach reduces the risk of applications being interrupted by lack of usable quota.

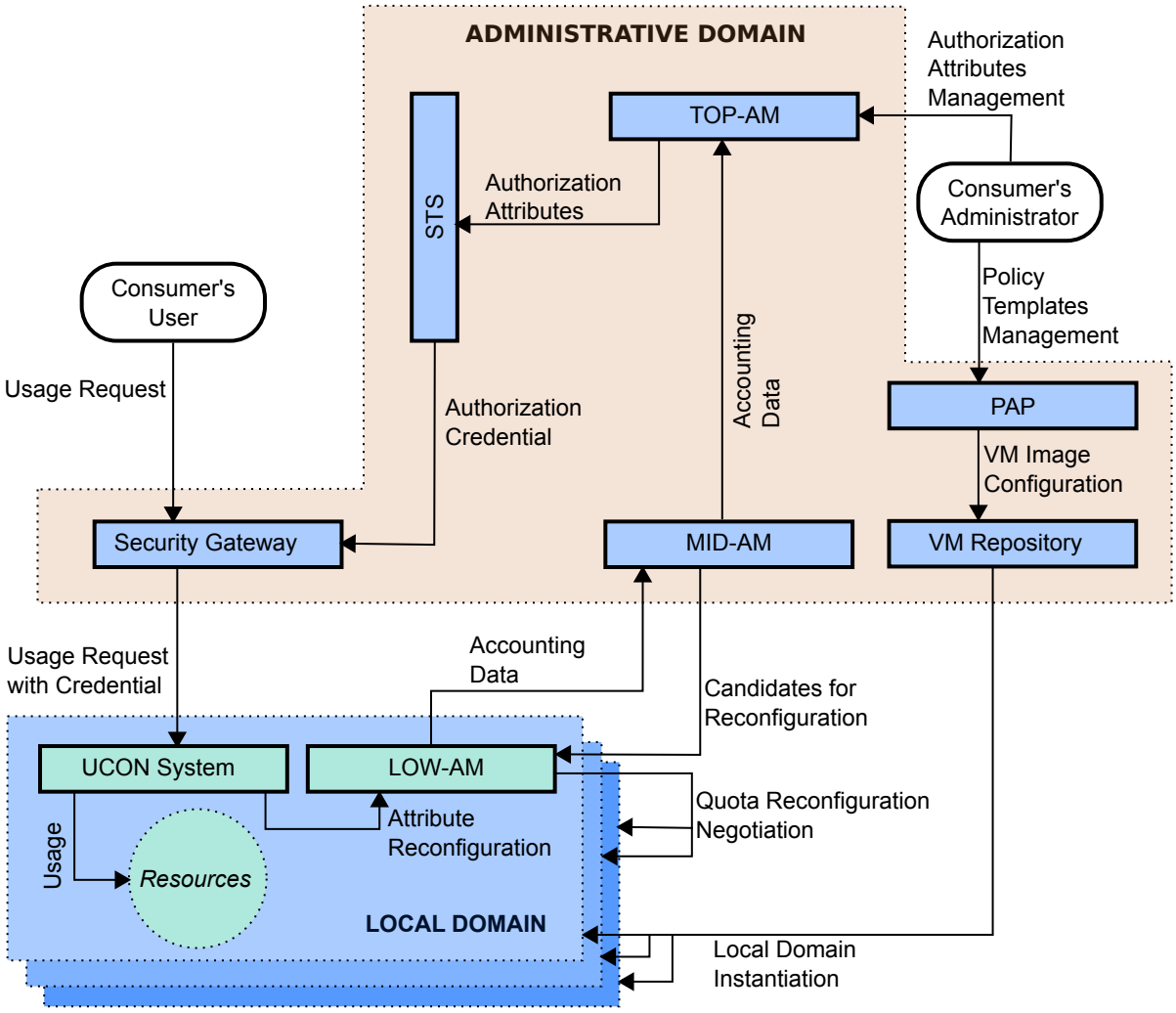

Figure 3 – General organization of the proposed architecture

The following sections describe the architecture of Figure 3 in a bottom-up style. We begin by discussing the organization and operation of a local domain and then we describe the details of the administrative domain.

3.2.2 Local Domain

The local domain implements the run–time environment in which user applications can be executed and individually controlled. A given service is executed in an isolated environment provided by the local domain mechanisms (we call them containers), which allows for multitenancy – the ability to execute applications from different users alongside each other. The local mechanisms validate user credentials and combine them with policy templates to generate the usage control policies that will be applied to application containers. The evaluation and enforcement of these policies are locally performed. For an illustration of the local domain components, see Figure 4.

Users can perform two request types on the local domains: application requests and

management requests. Application requests are always directed to the user application being executed inside the container. Management requests, on the other hand, are used to create, modify, delete or retrieve information about a particular application (e.g., request to start a web application). Local usage control mechanisms are focused solely on the second type of request, application requests are routed to the user application itself.

When the user wants to perform a management request, she must submit the request to the Security Gateway (SG). The SG acts as a broker, retrieving the user credential from the administrative domain, selecting the appropriate local domain and forwarding the user request to it. The SG must take special care to ensure the user credential has not been already used elsewhere, as it contains authorization data akin to a capability.

The above-mentioned user credential is crafted especially for each application instance. As we will see on Section 3.2.3, the user must specify the resource quotas she wants to reserve for the requested application. These quota definitions are encoded on the user credential that will be used to perform the user's management requests for the target application.

A Policy Enforcement Point (PEP), on the selected local domain, receives the request from the SG and ensures that only authorized requests are performed in the local domain. The first step in the authorization process is to invoke the security token service (STS) to authenticate and validate the user credential (e.g. by verifying expiration dates, the authenticity of signatures, trust relations, data integrity). Any user credential that fails to be validated causes the user request to be rejected. After successful validation, the PEP submits an authorization request to the context handler (CH) along with contextual information (e.g. user credential, request content). The PEP then waits for a reply with an authorization decision to be enforced.

The CH integrates the many components of the local domain. It translates the request format used on the PEP to a format useful to the other components. After extracting the user credential from the PEP's request, the CH invokes the local policy administration point (LPAP) and, in parallel, retrieves any resource utilization data associated with the user application (if it is already running) and the environment attributes from the Policy Information Point (PIP).

The LPAP derives usage control policies from the user credentials and policy templates contained in the local template repository. The discovery of which rules must be created (e.g. pre-authorizations, ongoing conditions, etc.) is done by matching attributes contained in the credentials with *field-id*s present on the policy template. Each rule with a matching *field-id* is configured with the corresponding value from the user credential. 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 purpose of the PIP is to provide resource usage information to the CH through a well-defined interface. Usage information is collected by the low-level accounting agent (LOW-AM) and made available to the PIP. The LOW-AM uses the operating system's native APIs

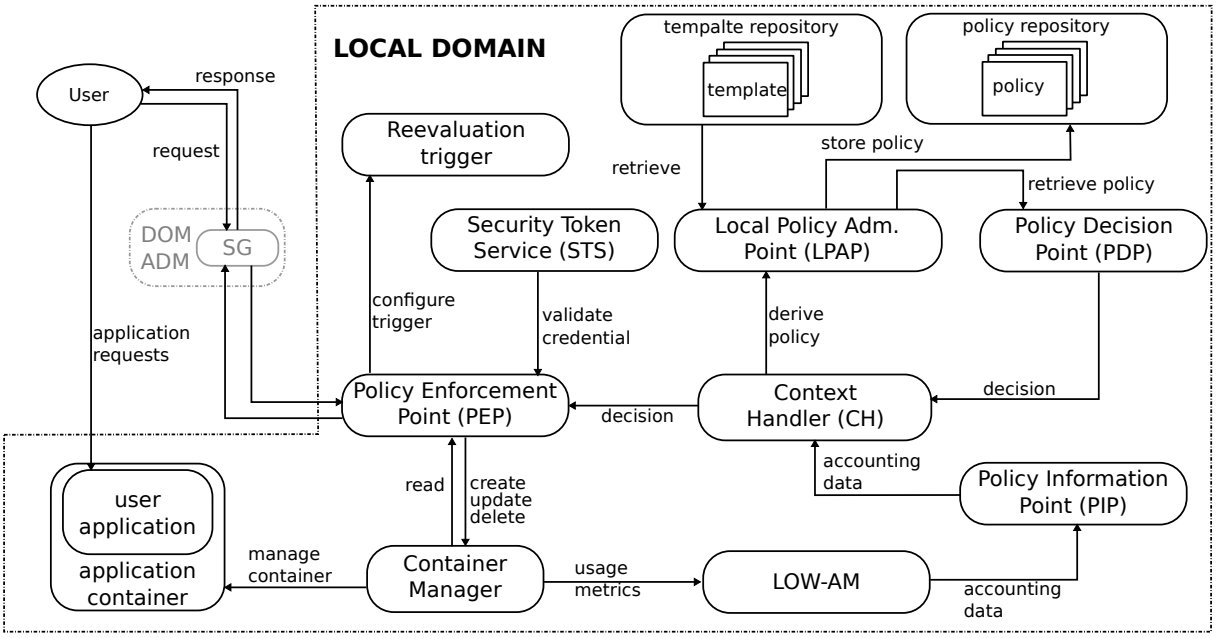

Figure 4 – Local Domain Components

to discover the resource usage for each individual container, as well as the current state of the system (e.g. load average, the number of running processes). UCON$_{ABC}$ obligations are treated as normal attributes stored on the PIP that must be updated by an external agent, because not all obligations can be observed from within the system.

LOW-AM is separated from the PIP because it implements other features not related to the PIP, whose sole function is to provide an attribute recovery interface to the context handler. The LOW-AM, on the other hand, collects accounting data and must register application quota definitions when the user presents her user credential. It performs cooperative quota reconfiguration between applications of the same user locally or remotely and sends accounting data to the MID–AM for further consolidation.

The next step for the CH is to verify if the request is related to an already running user application (subject to ongoing controls) or if it is a request for the instantiation of a new user application (subject to pre controls). An application container represents the object being controlled; therefore, its mere existence implies that a usage session is ongoing. The CH creates an authorization request with the gathered data, sends it to the PDP and waits for a decision.

The PDP is the specialized component for the evaluation of usage control policies, matching the data contained in the CH's request to the policy rules. The policy applicable to the application container is retrieved from the LPAP before evaluating the request. Each policy is linked to a single user credential; 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 if, after matching all attributes to the applicable policy rules, the rule-combining algorithm produces a *permit* decision. The decision is sent to the CH along with details of how the decision must be enforced.

Algorithm 1: Policy Evaluation

Input: *reqCtx* a request context
Output: *resCtx* a decision response context

```
1  S ← reqCtx.subject                                  // requesting subject
2  O ← reqCtx.object                                   // requested resource
3  C ← reqCtx.action                                   // requested action
4  A ← address(LPAP)                                   // LPAP address
5  PS ← retrieve_policy(S,O,A)                         // retrieve policy set
   // deny by default
6  if policies(PS,C) = ∅ then
7      return Deny
   // evaluate all policies
8  for P in policies(PS,C) do
9      for R in P do
10         if evaluate(R,S,O,C) = Deny then
11             return Deny
12 return Permit
```

The aforementioned process is repeated for any management request. This process can be clarified by studying Algorithm 1. The PDP retrieves the subject (S), object (O) and context (C) linked to the user request (represented as request context, *reqCtx*). The data is used to retrieve the applicable set of policies from the LPAP's address (A). A request is rejected if no policy applies to the current context (lines 6,7). A *deny overrides* algorithm is shown from lines 8 to 12: if any rule produces a *Deny* decision, the request is immediately refused, otherwise, the request is permitted.

The response from the PDP contains a decision to be enforced, a reevaluation trigger (RT) and any attribute updates required. The CH invokes the PIP to update any attribute, effectively supporting UCON$_{ABC}$ attribute mutability. Failure to update the attributes must cause the user request to be rejected. The decision and reevaluation trigger are converted to a format understandable by the PEP and sent to it after updating any attributes.

The PEP configures the reevaluation trigger (RT) in a component with the same name. The RT functions as a timer that invokes 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 performs the continuity of control, defined on UCON$_{ABC}$, as a configurable periodic reevaluation.

The PEP forwards authorized requests to a local Container Manager, which performs the required actions to prepare the containers and to manage user applications in it, as well as to manage the container life cycle. The application's access details (e.g. IP address) are returned to the user after container creation. The container manager uses the operating system's native mechanisms to setup the container limits in accordance with the user credential values.

```
<Rule RuleID="Storage" Effect="Permit">
<Target><Any/></Target>
<Condition>
  <Apply FunctionId="integer−less−than−or−equal">
    <Apply FunctionId="integer−one−and−only">
          <AttributeDesignator Category="access−subject"
           AttributeId="usedDiskSpace" DataType="integer"/>
        </Apply>
        <Apply FunctionId="integer−one−and−only">
          <AttributeValue DataType="integer">
           <%TotalDiskSpace%>
          </AttributeValue>
        </Apply>
  </Apply>
</Condition>
</Rule>
```

Figure 5 – Sample template rule

3.2.2.1 Templates and Policy Derivation

A template is a set of all the rules that can be used to control the behavior of a user application being executed in a container. To clarify this idea, a simplified version of a rule for controlling storage space is shown on Figure 5.

A *RuleID* identifies each rule unequivocally: when rule identifier is present in the user credential, the rule must be activated for this user. The rule may contain a variable number of attribute identifiers (e.g. *TotalDiskSpace*), which must be replaced by the value with the same *field-id* from the user credential. The *TotalDiskSpace* attribute, in this example, must be present on the user credential, otherwise no policy will be derived and the user request will be refused. Accounting data collected on the local domain can also be referred on the policy template with variable names, like *usedDiskSpace* for storage space already used.

The applicable rules are configured with the authorization attributes from the user credential and, after a successful derivation, the resulting policy is stored on the LPAP. This policy may contain rules to control the full lifecycle of the service container (i.e. pre and ongoing controls). Updates made to the template causes the derivation of policies to be repeated — the obsolete policies are deleted and the new policies take place. Policies may also be grouped in Policy Sets, each policy representing a well-defined stage of the usage session (e.g., pre-authorization, ongoing-conditions).

Remark 3. *The local domain architecture was the object of a research paper (24) published by us before the attribute management model was designed. As this proposal's main contribution is related to the autonomous self-configuration of these local domains, we included the research paper as an appendix to provide a more detailed view on the inner workings of the local domain. Therefore, to know more about the local domain details and the policy management approach*

based on templates and user credentials, please consult Appendix B, on page 85.

3.2.3 Administrative Domain

The administrative domain groups services that are related to high–level management. Local domains are autonomous in the sense that, at run-time, policy evaluation and enforcement are executed solely on the local domain, without the intervention of any external entity. However, the administrative domain plays an essential role in the functioning of the architecture. It provides the local domains with the required features to enable the cooperative attribute reconfiguration.

To achieve this task, there are two main components in the administrative domain. They are responsible for resource allocation (quota definition) and usage accounting. Together, they enable the local domains to reduce quota fragmentation. The components are:

- **TOP–AM**: it is the top level accounting module. It is hosted on the administrative domain and it has the responsibilities of managing user quotas and issuing application quotas upon user request. It also serves the purpose of monitoring global quota usage by receiving periodical updates from the lower level accounting agents.
- **MID–AM**: The mid–level accounting module has the purpose of consolidating accounting data received periodically from the local domains. The consolidated data is sent to the TOP–AM. The MID–AM helps on the process of quota reconfiguration between user instances in different domains, it also registers user credentials to avoid double spending.

3.2.3.1 Top–Level Accounting

The top–level accounting concentrates the activities related to coarse–grained quota management. When the consumer buys a certain amount of virtual resource from a provider, the consumer's manager configures the TOP–AM with the thresholds defined by the service level agreement (SLA) agreed by the provider. The manager distributes the available global quota among the consumer's users. Therefore, the TOP–AM is responsible for managing global quota (GQ) and user quotas (UQ) (Figure 2, on page 43).

On Algorithm 2 we can see details of the high-level accounting process. The TOP–AM module can be invoked by the STS from the administrative domain or by the MID–AM, anything else is rejected (see lines 2,14 and 18).

Messages from the STS are processed in a two–stage procedure. First, the TOP–AM verifies if there is enough free quota for each of the requested resource on the requesting user's quota (lines from 4 to 7). The whole request must be rejected if any of the requested quotas cannot be fulfilled (line 13), thus respecting the limits set up for the user. The requested quota amounts are debited in the user's account when there is enough free quota (lines from 8 to 11).

Algorithm 2: TOP–AM algorithm

```
   Input: A message request from the STS or MID–AM
   Output: A message reply with the status of the operation
1  user ← request.targetUser                                  // requesting user
   // treats requests from the STS
2  if request.source == STS then
3      ok ← true
       // enforce quota limits
4      foreach qr in request.quotas do
5          if freeQuota(user, qr.id) < qr.amount then
6              ok ← false
7              break
       // update user quota definition
8      if ok == true then
9          foreach qr in request.quotas do
10             debit(user, qr.id, qr.amount)
11         return success
12     else
13         return denied
   // applies updates from the MID-AM
14 else if request.source == MID–AM then
15     foreach qr in request.quotas do
16         updateUse(user, qr.id, qr.amount)
17     return success
18 else
19     return denied
```

Messages received from the MID–AM (line 14) are used to update the global accounting. The TOP–AM updates each user account with the data received from the MID–AM (lines 15 and 16), replying with a success message to the MID–AM (17).

This consolidation of user accounting allows the consumer's manager to see how much quota is being actually used by each user. The manager can detect when some users are suffering from quota starvation and, therefore, may reconfigure the quota definitions, taking spare quota available from some users and transferring it to the starving users. The consumer may also decide to buy more virtual resources when all users are working closer to their quota limits.

3.2.3.2 Intermediary Accounting

Intermediating the low–level accounting and the high–level accounting is the MID–AM component. It implements an auxiliary service with a couple of functions:

1. Keeps backup records of used user credentials to avoid double spending;

2. Receives accounting data from the local domains;

3. Consolidates the accounting data for each user and relays it to TOP–AM;

4. Keeps records on quota usage for each application/user on each local domain;

Item 1 is related to the double-spending problem. The local domains must verify if any user credential received was not used prior to the actual request. Unseen credentials (i.e., not found on the local records) must be first registered by the LOW–AM on the MID–AM service before they can be used in the local domain. The LOW–AM will receive an error message when it tries to register a user credential that has been registered earlier, otherwise, a success message is returned and the credential is registered.

Remark 4. *Even though the user credential is targeted at a unique local domain, a malicious user could still try to use it more than once on the same local domain (assuming that she could exploit some vulnerability on the security gateway to perform such action). Therefore, it is essential that the MID–AM be fault–tolerant to some degree, to guarantee that even when a local domain loses its user credential records, such information can be recovered from the MID–AM.*

The user credential is double-checked before it is used (i.e., by searching local and external records). This can increase the credential validation cost, though it only incurs on the first request. The usage control system does not need to interact with the MID–AM to evaluate and enforce the usage control policies.

The MID–AM also receives accounting data from each user on each local domain. A given user can own many applications executing on different local domains, and each with its own application quota definition. Periodically, each local domain sends the accounting data about each user to the MID–AM, where it will be recorded, consolidated and sent to the TOP–AM (to be used for high-level management operations).

The recorded accounting data can be consulted by the LOW–AM modules to figure out where a given user owns unused quotas. That is, the LOW–AM modules can discover potential candidates to transfer part of the configured application quota for the target user (Candidates for Reconfiguration, Figure 3). The local domains access this information through a service interface that lists each domain where the target user owns application quota and how much of the quota is unused. Therefore, this information can be used for the quota reconfiguration process.

3.2.3.3 Application Quota Configuration Process

The applicable usage control policy can only be derived from attributes in the user credential and a policy template. As mentioned earlier, these user credentials are issued on the administrative domain.

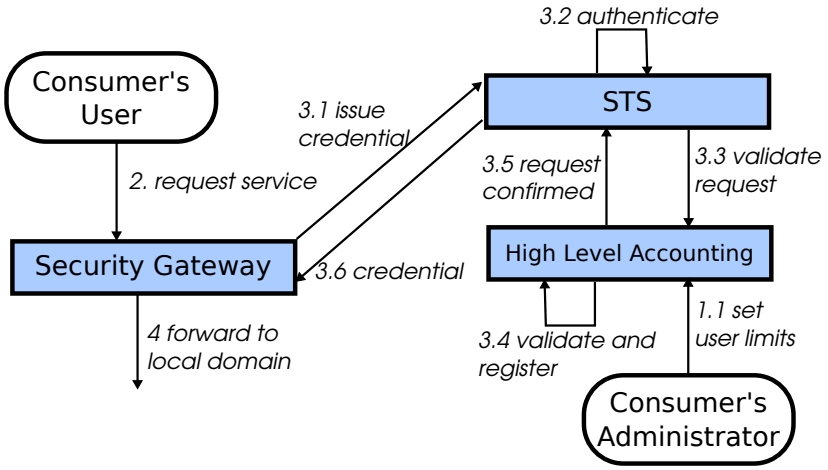

Figure 6 – Overview of application quota configuration

The *TOP–AM* (seen Figure 6) module is used for defining user-level resource limits (i.e., the user quota), as well as to keep track of consolidated usage data. The consumer's administrator must set the limits for each user (step 1.1). For instance, a given user can be granted 20 GB of disk space, 5 CPU cores, and 30 GB of network bandwidth. To further simplify matters, the administrator could choose to use a role-based approach: each user role can be configured with a predefined group of resource limits, thus when the user requests a credential, her user quota will be set as per the role definitions.

The user must make her request through the Security Gateway (step 2), and it will negotiate with the STS of the administrative domain before she can request any service from the local domains (step 3.1). The user request must contain the tentative limits the user wants to impose on the application to be run on the local domain (i.e. the application quota). The STS will authenticate the request (step 3.2) and forward it to the TOP–AM module to be validated (step 3.3).

The TOP–AM module first searches for the applicable user quota: if the limits are role-based, the system will derive the user quota from the role definitions. The system then verifies if there is enough available quota to be granted (see Definition 3, page 42), registering the requested amount in the positive case (step 3.4). A successful validation and registration process is followed by a confirmation to the STS (step 3.5). The requested attributes are then issued as a user credential for the requesting user and the target domain (step 3.6). The request is then sent to the local domain (step 4) to be processed.

3.2.3.4 Avoiding double-spending

The security token service of the administrative domain issues user credentials to requesting users, allowing them to perform usage requests on the local domains (Figure 3). The user credential contains application quota definitions that could be used to create an application instance in any of the available local domains. However, the user credential must be used only

once, otherwise, the user could easily bypass the usage control restrictions.

The Security Gateway was created to prevent double spending. Besides hiding the local domains from the direct contact of the outside world, it decides which local domain will perform the user request and, thus, receive the user credential. It requests the user credential on behalf of the user and selects where it will be used.

To select the local domain, the security gateway can base its decision on different kinds of strategy (e.g., round-robin fashion, first-fit, best-fit, etc). The suitable local domain's address is used when requesting the application quota to the STS. The user credential issued will contain the selected local domain's address as the target of the assertion, thus, only the targeted local domain can use this assertion. This feature eliminates the possibility of using the same credential in more than one local domain. However, there is still the possibility of using it again on the same local domain.

Double-spending the application quota contained in the user credential on the same local domain is avoided by using validity timestamps, local records of the user credentials used and remote records of the same information. Timestamps are required to eliminate the need of eternally remembering what credentials have been used: after the credential becomes invalid (i.e., stale), the corresponding record can be eliminated. The local records allow the local domain to identify already used credentials and quickly refuse the offending request. The external records are used as a safety prevention against failures in the local domain: should the local domain crash and loose its records of used credentials, it can recover the required data from a backup service. The level of replication of such records will define how many faults can be tolerated.

3.2.4 LOW–AM and Application Quota Reconfiguration

The LOW–AM module performs the local accounting tasks. It gathers accounting data for each application executed on the local domain and updates a local accounting repository. As said previously, when the user requests the instantiation of an application, the user credential is registered locally with the respective resource quota for the application and remotely on the MID–AM. As the application utilizes the resource, the LOW–AM module updates the application records to reflect this. The consolidated user accounting data is sent to the MID–AM, which further consolidates the user data received from the other local domains.

The biggest problem that arises from the decentralized approach is the fact that, under certain circumstances, an application can exhaust its configured quota and abort, even though the application's owner is entitled to more quota, which may be free elsewhere. When the user defines the application quota, it sets just a tentative limit for that application. The reasoning is that by bootstrapping the application with a tentative amount, we can avoid the large number of interactions that would be present in an architecture where no quota was previously defined and all the local domains would have to negotiate quota on demand.

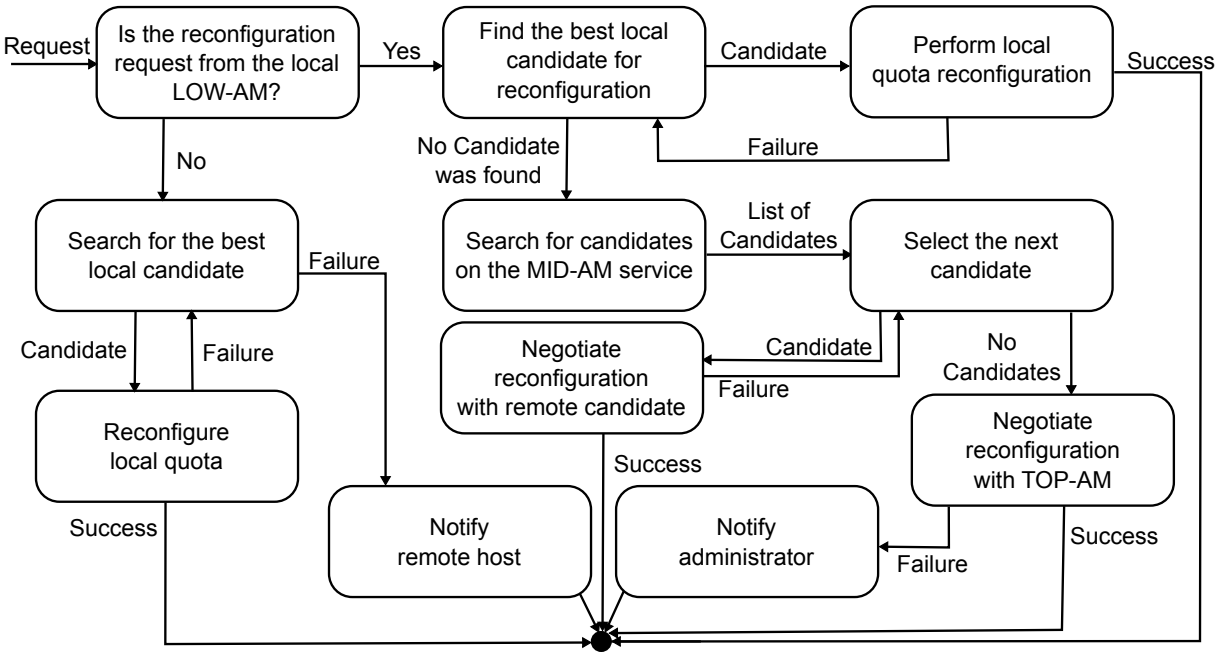

Figure 7 – Overview of quota reconfiguration process

Thus, the solution found was to give a large enough amount of resources to each application and to reconfigure this definition as needed to avoid fragmentation. The user still possesses the power to define that some applications are not allowed to have its quota reconfigured. The user might want to use such a limitation to control problematic applications (e.g., applications that might have memory leaks, services exposed to the outside world that might get short computing surges, and so on).

This requires the application owner to indicate if she wants to let the application quota to be reconfigured, and when this reconfiguration must take place. That is, the user must define what the trigger for the reconfiguration procedure is. The simplest form is to set the trigger as a proportion of the application quota: when the accounting data indicates that the application already used, for instance, 80% of its quota, the reconfiguration must start. This proportion is highly dependent on the application type and reconfiguration time. Applications that consume resources in a fast pace must require a lower proportion as a trigger (e.g., 60%) to give the LOW–AM enough time to finish the reconfiguration process, while slower applications could work with higher proportions (e.g., 90%).

The Figure 7 describes the LOW–AM process for reconfiguration of application quotas. It works with three types of reconfigurations: local reconfigurations between application quotas of the same user; reconfigurations between different local domains; and reconfigurations involving the TOP–AM.

To avoid race conditions on the quota reconfiguration process, we adopted a FIFO approach when dealing with reconfiguration requests. That is, the LOW–AM does not handle concurrent requests, every received request goes through a queue and waits for its turn to be

processed. Therefore, it is possible that a requesting local domain will need to make multiple tries before finding another local domain with enough quota to transfer.

Each local domain executes a process that continuously monitors each application reconfiguration trigger, as defined by the user. Every time the accounting data is updated, the corresponding trigger is verified. When a trigger is reached, a reconfiguration message is sent to the local LOW–AM with the reconfiguration parameters (AppID, ResID, UserID and requested amount). The owner of the user credential must also define the amount in order to provide an adequate increment, thus avoiding frequent reconfigurations. When the LOW–AM receives a message, it verifies if it comes from the local system or from a remote system.

Local requests should be first reconfigured with local application quota definitions. The reasoning behind this strategy is that by reconfiguring locally we can reduce network overhead, resorting to remote reconfiguration only when there is no locally available quota.

A local domain might have more than one possible candidate for quota reconfiguration. Thus, the LOW–AM must select the best candidate. A possible heuristic might be a candidate that: (i) must have enough quota to transfer to the requesting application and, after decreasing its quota, (ii) must have the highest proportional safety margin before reaching its own trigger. The idea is to take quota from the process with the smallest probability of hitting its own trigger in a short amount of time. This process can also be improved by employing a machine learning based classifier, as discussed on (25), to decide if a quota reconfiguration should take place or if it can be ignored (when the classifier detects a temporary computing surge).

Once a local candidate is found, the LOW–AM module tries to reconfigure the quotas from the requesting application and the donor application. This is done as a transaction and, as the quota might have changed in the interval between candidate selection and the actual reconfiguration, this operation may become infeasible. When the operation fails, the LOW–AM searches for the next best candidate and tries again. A successful reconfiguration results in two operations (not shown): updating the affected policies (of the requesting application and donor application) and updating the MID–AM records. When no local candidate is found, the external quota reconfiguration is initiated.

The LOW–AM from the requesting application must contact the MID–AM to receive a list of domains where the application's owner has configured application quotas. This list contains the local domains addresses and the amount of quota that is available on each domain, sorted by the largest safety margin to reach the reconfiguration trigger.

The requesting LOW–AM picks one candidate from this list and sends a reconfiguration message to it, with ResID, amount requested and UserID. The remote LOW–AM must select the best candidate for reconfiguration. When a candidate is found, the remote LOW–AM must coordinate with the requesting domain, so that both domains update their local quotas. A successful operation results in the applicable policies being updated message sent to the MID–

AM to update the accounting records. Care must be taken for this operating to be performed as a transaction (i.e., all or nothing); failure to do so might expose the system to inconsistent accounting.

The remote LOW–AM system follows the same principle of operation as local reconfigurations. It keeps searching for reconfiguration candidates until there are no more candidates available. The failure to find candidates results in an error message being returned to the requesting LOW–AM, which then must choose another candidate from the list of candidates returned from the MID–AM. This process is repeated until there is no more candidates available.

The last resort is the TOP–AM. The local domain's LOW–AM sends a message directly to the TOP–AM, requesting additional quota for the application. The reconfiguration is successful if the user owns enough free quota on the TOP–AM, otherwise, the administrator is notified and an error message is returned. The application will execute until its quota is not exhausted. This operation must be a transaction to avoid inconsistent accounting.

The reasons for leaving the TOP–AM as the last resort are two: (i) to avoid the bottleneck of using a centralized point for requesting reconfigurations — which would require some involved scheme to provide high scalability and fault tolerance to the TOP–AM; (ii) to allow the user to instantiate more applications with the quota available on the TOP–AM, which would not be possible if the user quota quickly became fragmented on the local domains.

For instance, suppose the user quota is defined as 100 GB of storage and the user instantiated two applications using 40 GB each one. Application one reaches its trigger (e.g., 38 GB), while application two used only 10 GB. Considering that the application reconfiguration block is, for example, 5 GB, the user would soon run out of free quota on the TOP–AM, even though there is a large amount of unused application quota defined for Application two. Therefore, the user would not be able to start another application.

3.3 Conclusions

This chapter described the proposed solution in light of the requirements identified in the previous chapters. It presented the conceptual model that allows the usage control model to be decentralized. UCON$_{ABC}$ was originally conceived as a centralized model, in which the policy evaluation is performed by a single entity that can observe all changes to attributes on the system.

For UCON$_{ABC}$ to be adequately implemented in cloud computing environments, the policy evaluation procedure must be essentially decentralized. The best way of achieving this is by avoiding information sharing between policy evaluators, as it would be unfeasible to do this in a consistent fashion in the scales involved in cloud computing environments.

We propose the use of temporary attributes that must be created following a small number

of constraints, to enable independent policy evaluators that avoid the above problem. We also show how these same attributes can be used to update the persistent attributes, thus affecting future usage sessions, though without suffering from the high coupling present in the originally envisioned UCON$_{ABC}$ administrative model.

Cloud computing environments require any proposed access control mechanism to be flexible and scalable. This is a major challenge for traditional access control approaches, as the conventional models can be made scalable though cannot offer the required flexibility. The administrative model we described for UCON$_{ABC}$ allows it to be decentralized, therefore improving its scalability.

We designed an architecture that allows the proposed model to be implemented in a fashion that promotes scalability without compromising service predictability and flexibility. We showed that, even though the local domains does not share usage session state (to prevent them from becoming highly coupled), they can cooperate to avoid quota fragmentation. The autonomy of the local domains provide for a fast policy evaluation frequency (which is not affected by the environment size) and resiliency in the face of faults on some local domains (as the other domains can keep functioning normally).

Policy management is made simple by using policy templates and user credentials. Each policy is derived specifically for the target application and it is updated whenever the user credential changes. The mechanisms proposed, avoid the need of remembering each user credential forever and ensures that there cannot be double-spending.

Therefore, the architecture proposed achieves the objectives laid out for the model. It combines flexibility, scalability, and ease of management, without requiring complex mechanisms. In these regards, it shows many advantages over previous proposals for applying UCON$_{ABC}$ to cloud computing environments.

In the next chapter, we will discuss a possible software implementation of the proposed architecture. We will also detail the prototype implementation of some components of the architecture that were used to evaluate its feasibility.

4 Implementation and Evaluation

This chapter discusses the software implementation of a prototype to evaluate the proposed architecture. It shows that it is possible to implement the envisioned features using open standards and public available software systems. It also shows the experimental evaluation results followed by its discussion.

4.1 Implementation

The prototype consists of mainly two components: the local usage control system and the MID–AM component. The first component was implemented to investigate the behavior of the system under normal conditions (i.e., without involving quota reconfiguration). The second component was designed to evaluate the feasibility and the performance of the intermediary accounting service, which is very important to the correct functioning of the proposed architecture.

We used two UNIX-like operating systems to implement the prototype. The FreeBSD operating system, version 10, was used to experiment with the Jails (26) environment, an operating system container that is capable of isolating groups of user processes, creating the illusion of a dedicated system to the process. It works by intercepting system calls and enforcing the access controls. The jail's system also provides a convenient application-programming interface to gather accounting information on the contained processes, and an easy way to enforce usage decisions on user processes.

The Linux operating system (27) was used to implement and evaluate the MID–AM service. FreeBSD could have been used in this step, though by using Linux we got access to a larger test bed. Linux can also be used to provide software containers (e.g., LXC (28), OpenVZ (29)), though these implementations are less mature than FreeBSD's jails. Linux also provides a feature called control groups, or *cgroups* for short, which allow the allocation of resources to groups of tasks defined by the user (30).

User credentials were encoded using the Security Assertion Markup Language (SAML)(31), which is a standard for writing various kinds of security assertions. More specifically, we used the AttributeAssertion type to carry user attributes describing the quota parts that should be configured for the user request. The OpenSAML toolkit(32) was used to craft and validate the assertions. Each assertion is digitally signed and encrypted using X509 certificates – those certificates must be signed by a trust anchor configured manually on the system's entities.

The communication with the local domain was made through a web service interface, based on the *ReST* model (33). This model was selected for being fast to prototype and is

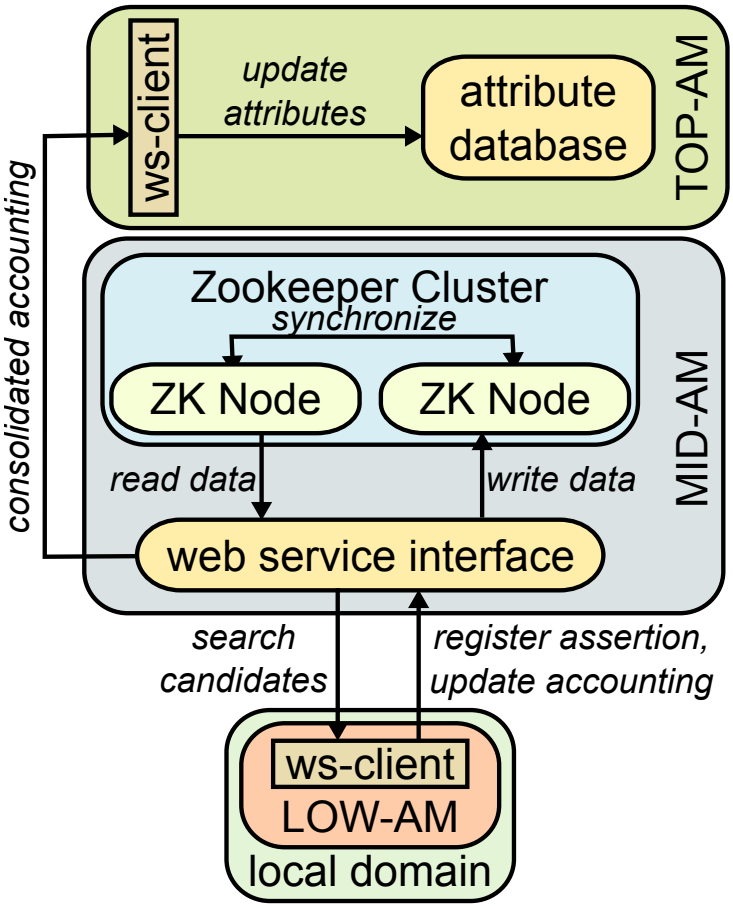

Figure 8 – MID–AM mechanism

well integrated into the web architecture. The web services were hosted on the NGINX web server (34). The services were implemented in a mixture of Java and Python programming languages. The communications were protected using the SSL protocol (35).

The policy templates are XML (36) files which represent XACML policies (eXtensible Access Control Markup Language(37)). The templates are not valid policies until they are processed by a substitution procedure which instantiates the usage control policies. The attribute identifiers present on the user's attribute assertion are matched with identifiers in the policy template, and the values are inserted as needed. The policy evaluation process is done with the Balana XACML library(38), which provides an XACML policy decision point implemented in the Java programming language.

Probably the most important part of the prototype is the MID–AM mechanism, where various information are kept to allow the cooperation between local domains (e.g., user attributes configured on each machine, assertion usage information, accounting data for consolidation and so on). As we mentioned earlier, this data must be replicated to provide some fault-tolerance to the local domains. When a local domain fails, it can recover its state from the MID–AM service and continue its normal operation.

The chosen architecture is shown on Figure 8. Each local domain is able to send requests

to write or read information to the MID–AM web service interface (e.g., search for candidates, register assertions and update accounting data). This interface can be replicated to provide some load balancing as well as to shield the back-end data storage services from misbehaving local domains. The web service validates the request and interacts with the back-end to perform the required actions.

The MID–AM storage is built using Apache Zookeeper(39), forming a cluster of Zookeeper Nodes (i.e., ZK Node). It provides a replicated storage based on the Paxos algorithm (40), thus consistency is guaranteed between ZK Nodes. Reads are faster than writes because reads can be relaxed in some situations, that is, a read can see older data in some situations. This, however, is not a problem, as Zookeeper allows the client to request a fully synchronized read when needed.

The services implementing the MID–AM web service interface can send read and write requests to any node from the Zookeeper cluster. The web service interface also sends consolidated accounting data to the TOP–AM upon request, thus allowing the administrator to obtain a consolidated view of the quota usage.

4.2 Evaluation

The experiments were divided into two scenarios: One for testing the local domain mechanisms and the other to evaluate the MID–AM performance. The next section details the tasks performed and the results obtained.

4.3 Local Domain Performance

The experiments were performed on a FreeBSD server running on an Intel core i7 machine with 4 processors, with two cores each, clocked at 2.67 GHz. The server had a total of 8 GB of RAM memory. All communications were made locally (i.e., no network overhead) to avoid external interferences on the results. The aim of these experiments were to evaluate the performance that each local domain could be expected to provide, as all quota reconfiguration process happens asynchronously. The variability coefficient remained under 5%.

One experiment measured the response time of the reference monitor (PDP) according to the number of policies stored locally. As it can be seen on Fig. 9, it can be considered low, always staying below 2 milliseconds on average. In the worst case, response times remained under 6 milliseconds in all scenarios, which still is a good response time. In this particular implementation, the limiting factor becomes the amount of memory available for caching policies. It must be noticed that the outsourcing model requires a request/response protocol, which is slower and may generate a larger amount of network messages.

The second test (Fig. 10) compares the hybrid template-based approach to traditional provisioning approaches. Provisioning is significantly more expensive when dealing with the same number of rules. The hybrid model used messages in the 4754-18524 bytes range; while in the provisioning model messages are in the 11314-42594 bytes range, the scenario involved the use of policies with 6 to 96 rules. The messages were 42.7% smaller on average.

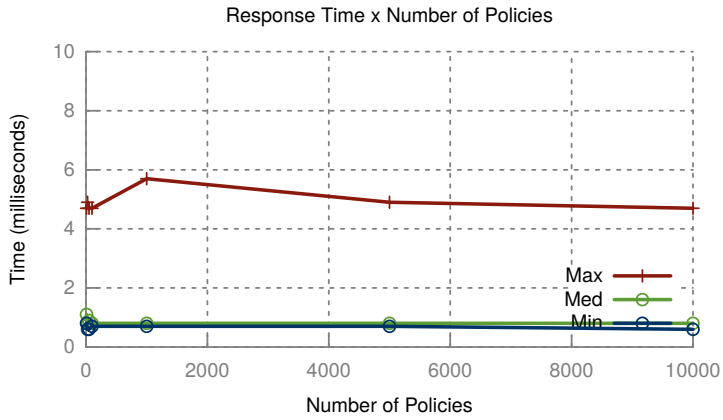

Figure 9 – PDP evaluation: response time versus number of policies in the repository

4.4 MID–AM Performance

The tests consisted of different scenarios of variable concurrency as well as requests of different sizes. The experiments were performed on eleven machines, one acting as the server and the rest acting as clients (i.e., local domains). The operating system on the hosts was Ubuntu Server LTS 12.04 for 64-bit architectures. The hardware platform was Intel core i5-3470 Quad Core, with 3.2 Ghz and 4 GB of RAM. The software on the clients used the official Apache Zookeeper Java client library to access the server. The server itself was the standard Zookeeper server, running in standalone mode (i.e., no replication). All tests performed 1000 requests to the server and the response times were measured in milliseconds.

The first test evaluated the response time for a single client writing a series of documents of variable size to the server node. The results are seen on Figure 11, on line 1. The test was repeated 1000 times for each message size. As can be seen, the response time is almost linear in the message range of 100 bytes to 5 kilobytes.

A variation of the first test was the measurement of the influence of the number of concurrent writes on response time, coupled with various message sizes. This can be seen also on Figure 11, represented by the other lines (the numbers on the lines represent the number of concurrent writers). Almost all scenarios show a similar behavior as the first test, though with higher response times directly related to the number of concurrent writers, except for the case of 10 concurrent writers, where the response times is better than when using 8 writers and 5 kilobyte messages.

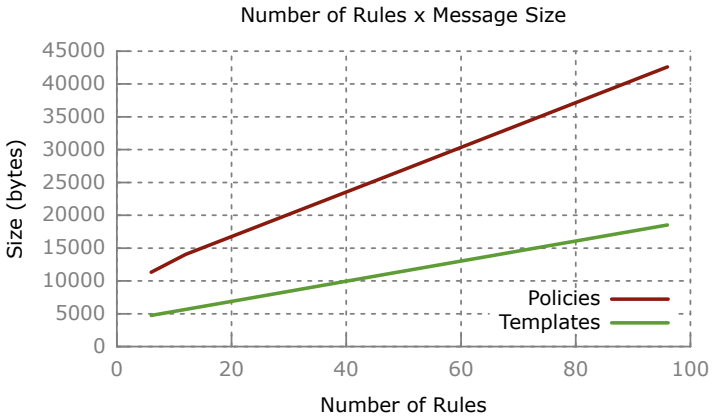

Figure 10 – PDP evaluation: message size in pure provisioning versus the proposed approach

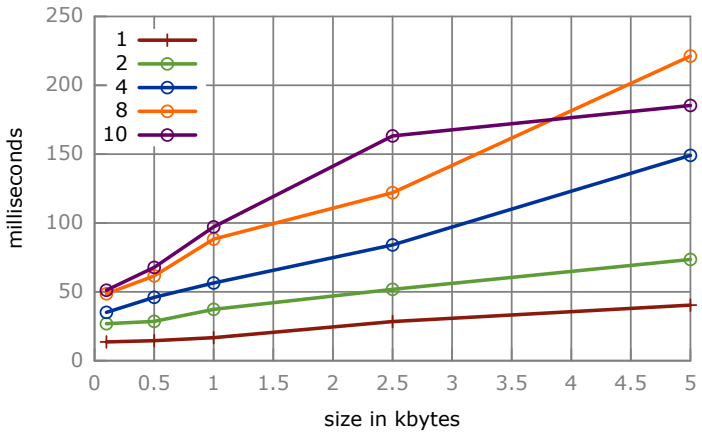

Figure 11 – Influence of writing concurrency and message size on response time

The reading performance was measured in two scenarios. The first one is shown on Figure 12, which shows the relation of concurrent writers (up to 8) on the response time of a single reader. All messages were 100 bytes long, and the reader kept iterating on the data from each writer (i.e., the target data changed at each request). The growth in response time is clearly correlated with the number of concurrent writes.

We also performed an evaluation of the message size on reading performance. This time, the scenario counted with eight concurrent writers and one reader. The message size was in the range of 100 bytes to 2.5 kilobytes. As can be seen from Figure 13, from 100 to 1000 bytes messages, reading performance is barely affected. The situation changes greatly when going to 2.5-kilobyte messages.

4.5 Conclusions

This chapter showed that it is feasible to implement a usage control system on current cloud computing environments. As our architecture is based on a Platform-as-a-Service approach, it is viable to implement it on actual cloud computing environments, as long as the host operating

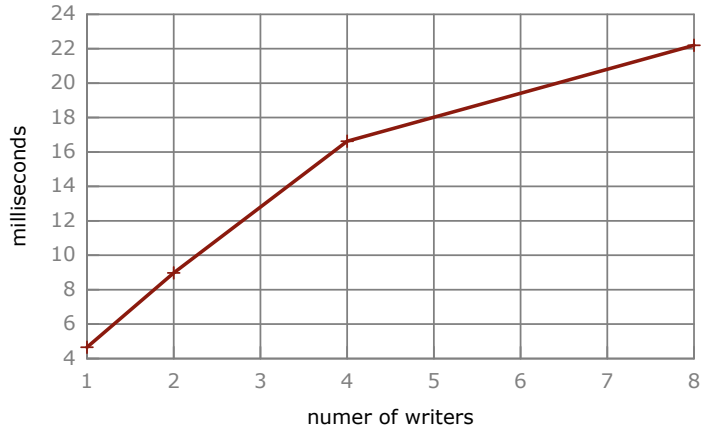

Figure 12 – Influence of writing concurrency on reading time for 100 byte messages

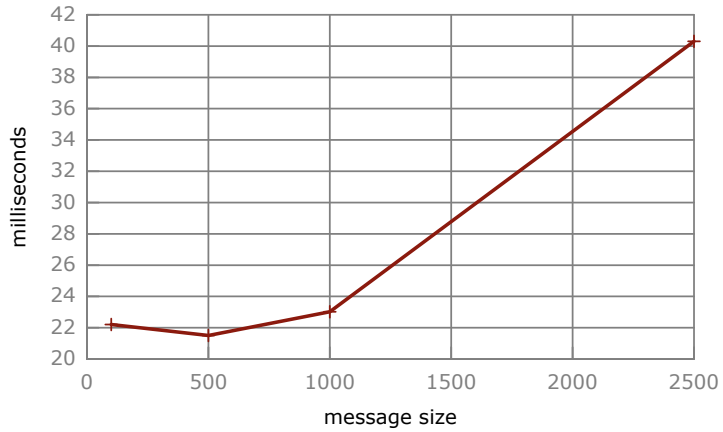

Figure 13 – Influence of message size on reading time for 8 concurrent writers

systems offer the required software containers. The missing components can be implemented using freely available software libraries and software implementations, everything based on open standards.

The experiments made showed the viability of the proposal. The experiments involving the local domain demonstrated that a template-based approach has advantages over pure provisioning approaches. Outsourcing approaches have to deal with the cost of network overhead, which increases as the number of servers grow, and pure provisioning must contend with higher message costs and a more complex management system. The proposed implementation combines the advantages of provisioning and outsourcing, without the aforementioned shortcomings.

5 Discussion, Conclusion and Future Work

We described an architecture that uses policy templates and user credentials to derive usage control policies. Templates are previously stored in the local domains, at which policy evaluation and enforcement are performed. The user credentials avoid the need of sending whole policies through the network for each user, reducing message sizes and keeping the flexibility for defining individual policies. Policy synchronization is done by sending new credentials with different attribute values – when the local domain detects the change, the affected local policies are derived again and reevaluated.

In this work, we designed a model and corresponding architecture to allow for decentralized usage control using a distributed cooperative attribute reconfiguration model. Our architecture behaves in an easier to predict way when compared to centralized approaches. The attribute reconfiguration protocol was developed to tackle the lack of attribute sharing intrinsic to decoupled applications. The same reason that makes decentralized architectures more predictable (decoupling) also makes it harder to correctly use the distributed resources.

We improved this problem by using temporary attributes that are provisioned to each local domain. These attributes can be reconfigured when the system detects a threshold condition, which triggers a process to reconfigure the temporary attributes of a given user. This allows the user quota to be better used on the system, even though each local domain is unaware of what is happening on other hosts.

This attribute reconfiguration model was discussed in the context of UCON$_{ABC}$ attribute mutability concept. We demonstrated that, even though the original authors did not discuss this scenario of independent authorization facilities and mutable distributed attributes, it is possible to include this variation by using the concepts of persistent and temporary attributes, coupled with update rules.

The problem of double spending that could happen when using digital credentials is avoided by using targeted credentials, defined during run–time, and a registration service that keeps track of the already used credentials while they are valid. The intermediary service (MID–AM) acts as a helper for the local domains. It can be regarded as a compromise between a completely centralized architecture and a fully decentralized one. Our experiments showed that it offers a reasonable performance, with little impact on the operation of the local domains, as most operations from the LOW–AM happen in the background (i.e., the user process does not stop while attributes are being reconfigured).

The experimental results show that the proposed architecture is viable; it shows promising performance and can be implemented with well–known technologies and standards. It provides advantages in comparison with centralized architectures, which must use complex technologies

to achieve scalability and that are harder to configure, due to dependency on variable parameters affecting the usage control implementation.

The tests showed a 42.7% reduction in message size compared to provisioning and faster response times compared to outsourcing (network latency is eliminated). The proposal shows the benefits of the provisioning without the complexity of synchronizing policies.

The contributions of this thesis can be summarized as follows:

1. It contains the description of a localized usage control system that eliminates the network overhead present in centralized approaches. The result of this is a higher frequency for policy reevaluations, as well as a predictable behavior, the reevaluation cost is stable no matter the number of hosts in the environment.

2. It describes a distributed and cooperative attribute management model that enables the automatic reconfiguration of application quotas in a distributed system, reducing the fragmentation of the user quota and improving the global quota usage efficiency.

3. It discusses the implications of distributed attributes reconfiguration in the scope of UCON_{ABC} model.

4. It describes a policy management approach to reduce the work of the system's manager by using policy templates and user credentials. The manager does not need to specify policies for each user, the policy limits are derived from the user credentials.

5. It proposes an attribute management approach that creates a consolidated view for the system's manager, with a fully distributed implementation to preserve the system's scalability.

6. It describes a reconfiguration protocol to allow the hosts to dynamically reconfigure application quotas to avoid process starvation when there is available quota elsewhere.

7. It shows how to integrate the reconfiguration protocol with the accounting architecture, enabling the local hosts to discover where the available attributes are and to proceed the reconfiguration. This process is carried in a proactive manner, avoiding the interruption of user applications to handle quota reconfiguration.

8. It demonstrates the viability of the proposed architecture by implementing a prototype based on open standards, coupled with discussion of experimental results from a performance evaluation.

Two aspects that were not covered in this work remain as future work: the investigation of thrashing conditions and the possibility of using a fully distributed model for the MID–AM component. The first topic is related to a possible cascading effect on attribute reconfiguration (e.g., host A takes some quota from host B, which takes some quota from host C, which takes

some quota from host A, and so on). The second topic refers to the possibility of using fully decentralized protocols to increase decentralization without increasing operational costs nor decreasing the architecture's robustness.

As part of the research works, the following publications were made:

- One conference paper discussing identity management integration between different Cloud Computing categories (21);
- One journal paper (4), a collaboration to investigate the use of flexible quota definitions in an outsourcing architecture to provide for resilience in the evaluation of usage control policies;
- two conference papers (41, 24) describing the organization of the local domain and the architecture for managing usage control policies with user credentials and policy templates.

References

1 PARK, J.; SANDHU, R. The ucon abc usage control model. *ACM Transactions on Information and System Security (TISSEC)*, ACM, v. 7, n. 1, p. 128–174, 2004. Cited 6 times on pages 15, 17, 23, 29, 30, and 31.

2 ARMBRUST, M.; FOX, A.; GRIFFITH, R.; JOSEPH, A. D.; KATZ, R.; KONWINSKI, A.; LEE, G.; PATTERSON, D.; RABKIN, A.; STOICA, I. et al. A view of cloud computing. *Communications of the ACM*, ACM, v. 53, n. 4, p. 50–58, 2010. Cited on page 23.

3 TAKABI, H.; JOSHI, J. B.; AHN, G.-J. Security and privacy challenges in cloud computing environments. *IEEE Security & Privacy*, IEEE, n. 6, p. 24–31, 2010. Cited on page 23.

4 MARCON, A. L.; SANTIN, A. O.; STIHLER, M.; BACHTOLD, J. A UCON$_{ABC}$) Resilient Authorization Evaluation for Cloud Computing. *Parallel and Distributed Systems, IEEE Transactions on*, IEEE, v. 25, n. 2, p. 457–467, 2014. Cited 4 times on pages 23, 24, 32, and 71.

5 LAZOUSKI, A.; MANCINI, G.; MARTINELLI, F.; MORI, P. Usage control in cloud systems. In: IEEE. *Internet Technology And Secured Transactions, 2012 International Conference for*. [S.l.], 2012. p. 202–207. Cited 2 times on pages 23 and 33.

6 LAZOUSKI, A.; MARTINELLI, F.; MORI, P. *A prototype for enforcing usage control policies based on XACML*. [S.l.]: Springer, 2012. Cited 2 times on pages 23 and 33.

7 TAVIZI, T.; SHAJARI, M.; DODANGEH, P. A usage control based architecture for cloud environments. In: IEEE. *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. [S.l.], 2012. p. 1534–1539. Cited 2 times on pages 23 and 33.

8 DANWEI, C.; XIULI, H.; XUNYI, R. Access control of cloud service based on ucon. In: *Cloud computing*. [S.l.]: Springer, 2009. p. 559–564. Cited 2 times on pages 23 and 34.

9 PARKHILL, D. F. Challenge of the computer utility. Addison-Wesley, 1966. Cited on page 27.

10 VAQUERO, L. M.; RODERO-MERINO, L.; CACERES, J.; LINDNER, M. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, ACM, v. 39, n. 1, p. 50–55, 2008. Cited on page 27.

11 MELL, P.; GRANCE, T. The nist definition of cloud computing. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, 2011. Cited on page 27.

12 DURHAM, D.; BOYLE, J.; COHEN, R.; HERZOG, S.; RAJAN, R.; SASTRY, A. *The COPS (common open policy service) protocol*. [S.l.]: rfc 2748, January, 2000. Cited on page 28.

13 SELIGSON, J.; CHAN, K. H.; GAI, S.; SMITH, A.; HERZOG, S.; MCCLOGHRIE, K.; DURHAM, D.; REICHMEYER, F. Cops usage for policy provisioning (cops-pr). 2001. Cited on page 28.

14 PARK, J.; SANDHU, R. Towards usage control models: beyond traditional access control. In: ACM. *Proceedings of the seventh ACM symposium on Access control models and technologies*. [S.l.], 2002. p. 57–64. Cited on page 29.

15 SALIM, F.; REID, J.; DAWSON, E. An administrative model for ucon abc. In: AUSTRALIAN COMPUTER SOCIETY, INC. *Proceedings of the Eighth Australasian Conference on Information Security-Volume 105*. [S.l.], 2010. p. 32–38. Cited on page 34.

16 BECKER, M. Y.; FOURNET, C.; GORDON, A. D. Secpal: Design and semantics of a decentralized authorization language. *Journal of Computer Security*, IOS Press, v. 18, n. 4, p. 619–665, 2010. Cited on page 35.

17 Red Hat, Inc. *OpenShift*. 2016. <http://www.openshift.org/>. [Online; accessed 24-April-2016]. Cited on page 39.

18 Pivotal Software, Inc. *Pivotal Cloud Foundry*. 2016. <http://pivotal.io/platform>. [Online; accessed 24-April-2016]. Cited on page 39.

19 Heroku, Inc. *Heroku Cloud Application Platform*. 2016. <https://www.heroku.com/>. [Online; accessed 24-April-2016]. Cited on page 39.

20 LIU, H.; JIN, H.; LIAO, X.; DENG, W.; HE, B.; XU, C.-z. Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines. *Parallel and Distributed Systems, IEEE Transactions on*, IEEE, v. 26, n. 5, p. 1350–1363, 2015. Cited on page 39.

21 STIHLER, M.; SANTIN, A. O.; JR, A. L. M.; FRAGA, J. D. S. Integral federated identity management for cloud computing. In: IEEE. *New Technologies, Mobility and Security (NTMS), 2012 5th International Conference on*. [S.l.], 2012. p. 1–5. Cited 2 times on pages 41 and 71.

22 PARK, J.; ZHANG, X.; SANDHU, R. Attribute mutability in usage control. In: *Research Directions in Data and Applications Security XVIII*. [S.l.]: Springer, 2004. p. 15–29. Cited 2 times on pages 44 and 45.

23 LEVY, H. M. *Capability-based computer systems*. [S.l.]: Digital Press, 2014. Cited on page 47.

24 STIHLER, M.; SANTIN, A. O.; MARCON, A. L. Managing distributed uconabc policies with authorization assertions and policy templates. In: *2015 IEEE Symposium on Computers and Communication (ISCC)*. [S.l.: s.n.], 2015. p. 619–624. Cited 2 times on pages 52 and 71.

25 SEGALIN, D.; SANTIN, A. O.; MARYNOWSKI, J. E.; SEGALIN, L.; MAZIERO, C. An approach to deal with processing surges in cloud computing. In: *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*. [S.l.: s.n.], 2015. v. 2, p. 897–905. Cited on page 59.

26 MCKUSICK, M. K.; NEVILLE-NEIL, G. V.; WATSON, R. N. *The design and implementation of the FreeBSD operating system*. [S.l.]: Pearson Education, 2014. Cited on page 63.

27 Linux Kernel Organization. *The Linux Kernel Archives*. 2016. <https://www.kernel.org/>. [Online; accessed 24-April-2016]. Cited on page 63.

28 Canonical Ltd. *Linux Containers*. 2016. <https://linuxcontainers.org/>. "[Online; accessed 24-April-2016]". Cited on page 63.

29 Virtuozzo. *OpenVZ Virtuozzo Containers – Wiki*. 2016. <https://openvz.org/>. "[Online; accessed 24-April-2016]". Cited on page 63.

30 RedHat, Inc. *Chapter 1. Introduction to Control Groups (Cgroups)*. 2016. <https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html>. "[Online; accessed 24-April-2016]". Cited on page 63.

31 COMMITTEE, O. S. S. T. et al. *Security Assertion Markup Language (saml) 2.0*. 2012. Cited on page 63.

32 Shibboleth Consortium. *OpenSAML Toolkit*. 2016. <https://wiki.shibboleth.net/confluence/display/OpenSAML/Home/>. [Online; accessed 24-April-2016]. Cited on page 63.

33 FIELDING, R. Representational state transfer. *Architectural Styles and the Design of Netowork-based Software Architecture*, p. 76–85, 2000. Cited on page 63.

34 REESE, W. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, Belltown Media, v. 2008, n. 173, p. 2, 2008. Cited on page 64.

35 FREIER, A.; KARLTON, P.; KOCHER, P. The secure sockets layer (ssl) protocol version 3.0. 2011. Cited on page 64.

36 BRAY, T.; PAOLI, J.; SPERBERG-MCQUEEN, C. M.; MALER, E.; YERGEAU, F. Extensible markup language (xml). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, v. 16, 1998. Cited on page 64.

37 RISSANEN, E. *eXtensible Access Control Markup Language (XACML) version 3.0 (committe specification 01)*. [S.l.], 2010. Cited on page 64.

38 WSO2. *WSO2 Balana Implementation*. 2016. <https://github.com/wso2/balana>. "[Online; accessed 24-April-2016]". Cited on page 64.

39 The Apache Software Foundation. *Apache Zookeeper*. 2016. <https://zookeeper.apache.org/>. "[Online; accessed 24-April-2016]". Cited on page 65.

40 LAMPORT, L. et al. Paxos made simple. *ACM Sigact News*, v. 32, n. 4, p. 18–25, 2001. Cited on page 65.

41 STIHLER, M.; SANTIN, A. O.; MARCON, A. L. A usage control platform based on rule templates and authorization credentials. In: SBC. *Computer Networks and Distributed Systems (SBRC), 2015 XXXIII Brazilian Symposium on*. [S.l.], 2015. p. 50–59. Cited on page 71.

Appendix

APPENDIX A – Integral Federated Identity Management for Cloud Computing

Integral Federated Identity Management for Cloud Computing

Maicon Stihler, Altair Olivo Santin, Arlindo L. Marcon Jr.
Graduate Program in Computer Science
Pontifical Catholic University of Paraná
Curitiba, Brazil
{stihler,santin,almjr}@ppgia.pucpr.br

Joni da Silva Fraga
Department of Systems Automation
Federal University of Santa Catarina
Florianópolis, Brazil
fraga@das.ufsc.br

Abstract—**Cloud computing environments may offer different levels of abstraction to its users. Federated identity management, though, does not leverage these abstractions; each user must set up her identity management solution. This situation is further aggravated by the fact that no identity federation solution is able to integrate all abstraction layers (i.e. IaaS, PaaS, and SaaS). On this paper we describe a new architecture offering integral federated identity management, to support multi-domain clients in a multi-provider environment. We also present some implementation details. The proposed architecture offers significant advantages over current offerings: it eases identity management without losing flexibility, offers better user tracking through the whole cloud computing layers, and enables the implementation of multi-provider environments through account data replication.**

Keywords: cloud computing; federated identity management; single sign-on.

I. INTRODUCTION

Federated identity management deals with the establishment of trust relationships between various security domains, to share authentication data to reduce management complexity and security risks. It also helps to simplify authentication procedures for end users (e.g. by employing single sign-on, SSO) [1]. This subject has been studied and applied to many environments, such as web resources [2], web services [3], and grid computing [4], an evidence of the high relevance of federated identity management.

The emergence of cloud computing created a new environment that is not completely addressed by previous works. Cloud computing can be categorized as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), or Software as a Service (SaaS) [5], respectively by increasing level of abstraction. It is a common approach for higher levels of abstraction to leverage functionalities provided by the lower levels. However, current federated identity solutions are isolated to a single level (e.g. identity is federated only for IaaS or SaaS). Thus, if a SaaS provider wants to employ user identification at a lower level (e.g. to track user actions for auditing) she will have to come up with her own *ad hoc* solution, as the lower levels (i.e. PaaS and IaaS) are completely unaware of such user. The matters are further complicated if the environment spans multiple IaaS providers, as no available solution can offer a federation that is both horizontally (i.e. between multiple IaaS providers) and vertically (i.e. through all abstraction levels – SaaS, PaaS, and IaaS) integrated.

We designed a new architecture for federated identity management aimed at IaaS users, who wish to provide services and resources to other subjects. A defining characteristic is the transparent translation of high level identities (i.e. from SaaS level), authenticated by a third party identity provider (IdP), to lower level identities (i.e. for PaaS/IaaS usage). This allows SaaS users to perform authentication on their IdP and interact with the SaaS with SSO. Furthermore, the SaaS provider is able to track the user actions on the lower levels of abstraction, as the architecture provides the means to attach a unique credential that is valid on the IaaS. This also enables the provider to create applications tailored to each user, running under their own identity (i.e. no shared application), with individual accounting.

An interceptor installed in front of the SaaS application captures the user identity, received from the user's IdP, and exchanges it for an internal token on a security token service (STS). This token contains a unique identification that is digitally signed and registered on a central repository. The interceptor attaches this token to the user's request; another component operating on the IaaS level can then, for example, start a user processes under this identification. The central repository provisions this account data to the low level components, and is able to replicate data to other IaaS providers, effectively allowing the SaaS provider to track user activity over the entire environment.

This work makes significant contributions to the field of identity management. Previous works generally deal with relatively homogeneous scenarios (e.g. every resource is a web site) or makes some assumptions (e.g. authentication is interactive and password-based). We present a proposal to tackle a much more complex scenario, allowing sharing of information through all cloud abstraction layers, as well as on environments spanning multiple IaaS providers. The end users (i.e. from SaaS) are free to use their own IdPs, while the SaaS provider translates their identities transparently. The proposal brings various security advantages, like better auditing, accounting, and facilities for access controls.

The paper is organized as follows: Section 2 discusses some related works; Section 3 describes the proposed

This work was partially sponsored by the Program Center for the Research and Development on Digital Technologies and Communication (CTIC/MCTI), grant 1313 and National Council for Scientific and Technological Development (CNPq), grants 310319/2009-9 and 478285/2011-6.

978-1-4673-0229-6/12/$31.00 ©2012 IEEE

architecture. On Section 4 we present some implementation details. Section 5 presents our conclusions.

II. Related Works

Shibboleth [2] offers federated identity management based on two main components: the service provider (SP) and the identity provider (IdP). SPs protect web resources (i.e. web sites) and establish trust relationships with IdPs that act as authoritative authenticators. Once a user is authenticated on the IdP, she can access resources protected by SPs that rely on this IdP, without need for further authentications; she just needs to inform which one is her IdP. Under the covers, IdPs and SPs exchange authentication data according to the SAML specification. Identity providers have autonomy to decide, according to their privacy policies, which SPs may access the user's authentication data. OpenID [6] resembles Shibboleth on some aspects. The building blocks are called Relying Party (RP) and OpenID providers. OpenID providers store authentication data, and RPs play a role similar to what SPs do on Shibboleth. However, the user is the one who decides who may have access to her authentication data. This approach empowers the user discretion to choose who is to be trusted and, therefore, makes the trust relationships more flexible.

Kerberos [7] is a distributed authentication service very popular for operating systems. It employs encrypted messages, following the authentication protocol by Needham and Schroeder [8], with the addition of timestamps. Kerberos provides single sign-on on systems that trust its authentication mechanisms (i.e. the authentication service and the ticket granting service). It is also possible to perform federated authentication between different security domains, if the authoritative authentication servers have trust relationships established with each other [9].

A model for distributed identity management for digital ecosystems is described on [10]. Authentication data is abstracted through the usage of user profiles that are encrypted and replicated to trusted peers. It describes a model to support dynamic digital ecosystems, with single services and service compositions.

The above mentioned works represent a good approach when dealing with a relatively homogeneous environment (e.g. only operating systems with Kerberos, or only web sites with Shibboleth). Although the proposal on [10] works with credential translation, it is heavily based on SAML. This is not easily supported on popular operating systems, thus restricting its usage mostly to web based resources and systems that are SAML compatible.

III. Federated Identity Management Architecture

First, we need to define the entities belonging to the scenario considered in this work:

- *SaaS User*: any entity that wants to access the resources exposed as a SaaS application. Her identity is unknown outside of the SaaS context.
- *IaaS Contractor*: someone that contracts resources from IaaS providers, to deploy the SaaS Application to be provided to *SaaS Users.*
- *IaaS User*: the entities recognized on the IaaS resources (e.g. operating systems) as valid accounts.
- *Contracted Resources*: infrastructure resources like processing time, disk space, and network bandwidth.
- *SaaS Application*: a system that employs *contracted resources* to provide some specific functionality to the *SaaS users.*
- *Identity Providers*: any entity that is responsible for managing the *SaaS users'* authentication data.

IaaS contractors employ *contracted resources* to offer *SaaS applications* to the *SaaS users*. It is a popular approach to sell such services in a *pay-as-you-go* manner. However, as the *SaaS user* is only known inside the *SaaS application* itself, it is hard to measure with precision how much of the contracted resources a given *SaaS user* has consumed. Without the unification of *SaaS user* identities and *IaaS user* identities, it is only possible to obtain estimated (average) individual resource consumption.

In this proposal it is important to know which *SaaS user* is executing which actions on the infrastructure level. This not only enables a fair accounting system, it also permits an accurate identification of users for fine grained auditing and access control in IaaS.

With these requirements in mind, we propose a new architecture for federated identity management that integrates the IaaS and SaaS layers. Some components are operated on the PaaS layer to conceal the implementation details of identity translation from the *IaaS contractor*. We also designed an approach to permit the *SaaS user* to use the *SaaS application* without any interference.

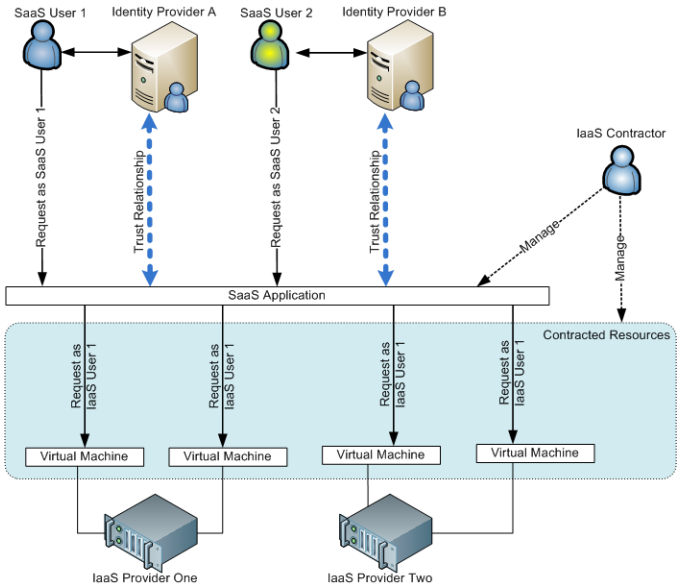

Figure 1: Example Scenario

A. Example Scenario

On Figure 1 we show a scenario involving all the entities described above. The *IaaS contractor* acquires the *contracted resources*, in the form of virtual machine instances, from two different providers (IaaS Providers One and Two). The *IaaS contractor* then deploys her *SaaS application* over the *contracted resources*; she already has the burden of managing both the infrastructure and the application. It is also important to set up trust relationships with the many *identity providers* responsible for the *SaaS users*' authentication.

Figure 1 exposes the limitation of current approaches: even though the *IaaS contractor* can delegate *SaaS users*' authentication to third party *identity providers*, she cannot be sure of which user is consuming what resources on the infrastructure. An outside view, perceives the SaaS application consumption as referring to a single entity (e.g. *IaaS User 1*). The *IaaS contractor* must design her *SaaS application* specifically to track *SaaS users*' actions, for accounting and auditing purposes, as well as for authorization and access control enforcement.

B. The Architecture

The proposed architecture is shown on Figure 2. It is composed of four identity domains: *User domain* corresponds to the security domain in which a given external user exists; *IaaS domain* is where the *contracted resources* exist, and may be spread over different *IaaS providers*; *SaaS domain* refers to the security domain of the provided *SaaS application*; The *PaaS domain* contains the components necessary to perform identity translation and data replication over different *IaaS providers*. On real world scenarios there may be more occurrences of the same kind of domain (e.g. many *IaaS and User domains*), we presented only one of them for the sake of simplicity.

A trust relationship between the given *user's domain* and the *SaaS domain* is prerequisite for the *SaaS user* to access the provided application and resources. The configuration of this relationship depends on the type of *Identity Provider* employed. The proposed architecture can support different *identity providers* (e.g. Shibboleth, OpenID) by using different application access points (i.e. *interceptors*).

The basic steps needed for a given *SaaS user* to access the provided *SaaS application* are described bellow (see Fig. 2):

1) The *SaaS user* tries to access the *SaaS application* interface with her browser.

2) The *interceptor* cannot find a valid session for the access request, thus it requests the *SaaS user* to authenticate on her choosen *identity provider*.

3) The *SaaS user* is forwarded to the *Identity Provider* authentication interface, and performs her login if needed.

4) The *Identity Provider* redirects the *SaaS user* to the *interceptor*, embedding the required *proof of authentication* (e.g. a SAML token).

5) The *SaaS user's* browser tries to access the provided *SaaS application*, though now the request has an embedded *proof of authentication*.

6) The *interceptor* validates the *proof of authentication* and requests a new *security token* from the *Security Token Service (STS)*. The trust relationship between *interceptors* and the *STS* is established by the PaaS administrator, who owns both components.

7) The *STS* verifies if the *SaaS user* is allowed on the IaaS security domain. A new token containing the *IaaS user*' *ID* is issued for valid users and is signed with the *STS* private key:

a) First time *SaaS users* get their account registered on the *IaaS Identity Provider*, this is needed only once. A unique identification is created, derived from the *SaaS user*'s identification. Thus the *STS* can always issue new security tokens whitout need for storing the IaaS identities locally.

b) The *IaaS Identity Provider* centralizes the accounts for all *IaaS users*, and shares this data with the authorized mechanisms (e.g. operating systems) eliminating the need for account creation on each virtual machine's operating system.

8) The *interceptor* forwards the *access request* to the application endpoint (e.g. an internal URL), embedding the *security token* obtained from the *STS*.

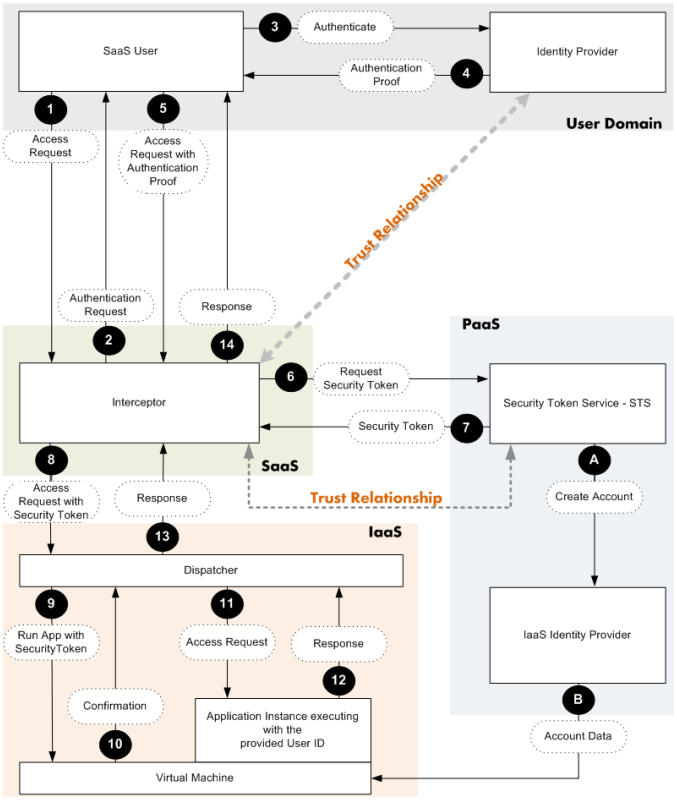

Figure 2: Architectural Overview

9) A local component (i.e. the *dispatcher*), running with administrator rights, captures the request and extracts the security token. If it is signed by the *STS*, it is considered authenticated. The *dispacher* uses the IaaS identification contained on the request to execute actions on behalf of the requesting user (e.g. to start an application instance owned by

the provided user identity). The trust relationship with the *STS* is configured by the *PaaS administrator*.

10) As the underlying mechanisms recognize the IaaS identification, the action is performed with success.

11) The *access request* is delivered to the *SaaS application instance* owned by the *IaaS user* provided on the *security token.*

Steps from 12 to 14 refer to the forwarding of SaaS application responses to the SaaS user's browser *or client application.*

The actual *SaaS user* authentication procedure depends on the solution used by the *user domain*. For approaches that offer *single-sign on*, we consider that the *SaaS client application* will be running on top of a web browser, thus inside the authenticated *SaaS user's* session. The application requests are tunneled inside this authenticated session and the *interceptor* enriches the *access request* with the *security token* from the *STS*.

For *SaaS users* who want to use custom developed applications (e.g. web service clients) it is possible to provide a separate gateway (i.e. *interceptor*). The only difference from the mentioned scenario is that the *SaaS user's* application will have to actively authenticate itself on the *identity provider* and cache the *authentication proof* for future use. Therefore, the application requests will not be tunneled inside browser authenticated HTTP sessions.

Account data stored on the *IaaS identity provider* can be replicated to other instances running on different *IaaS providers*. This allows for the unique identification of any user, no matter where the *SaaS application* instance is created, and provides what we called horizontal federation of identification. Data replication is desirable for reducing communication costs (e.g. network latency), though it is possible to set up an environment in which each *IaaS IdP* would be responsible for some subset of the identities, delegating the rest to the remaining *IaaS IdPs*.

Vertical federation is achieved by supporting different user authentication mechanisms in the front-end (i.e. *interceptor*), enabling different *SaaS users* to use their chosen identity management approaches, though maintaining identification unity through the abstraction levels by using the *STS* in cooperation with the *IaaS identity provider.*

The *dispatcher*, running on each virtual machine, acts as a proxy, performing the actions the *SaaS user* requested. It is needed to start the *SaaS application* with the user's credentials or to import (*mount*) the user's network storage devices on the target system, and so on. This is fundamental for the individual accounting, for auditing purposes, and to employ access controls tailored to specific users.

IV. Implementation Details

Bellow we present brief descriptions of the software components for the prototype implementation:

- OpenID authentication system [6]: offers a flexible solution for web authentication. It is used in the role of the *Identity Provider* from Figure 2. We selected this implementation for its ease of use and because it is a well-known framework for SSO purposes.
- JAX-WS RI [11]: an implementation of the Java API for XML Web Services. It used to create Java Applets running inside a web browser's HTTP session, making SOAP requests to the *SaaS application*.
- Apache Tomcat [12]: a popular servlet container, used for hosting web based components, like the *interceptor* and the *STS*.
- Apache Axis 2 for Java [13]: a web services toolkit used in conjunction with the Apache Rampart [14] security module to implement the *STS* component, compliant with the WS-Trust specification.
- OpenLDAP [15]: an LDAP implementation, supporting data replication to slave servers. This component is used on the role of *IaaS Identity Provider*.
- OpenSAML [16]: a popular SAML toolkit, used by the *STS* to create *security tokens* according to the SAML specification.

We are using Eucalyptus [17], an IaaS platform capable of deploying a group of virtual machines running the Debian GNU/Linux [18] operating system.

The OpenLDAP directory provides account information to the Linux operating systems through a component called NSS_LDAP. This enabled us to configure the operating systems with account information without the need for locally creating user accounts. It also enabled the data replication to other *IaaS providers* with native protocols. We also considered using a Kerberos implementation, but it seemed very inflexible if compared to an LDAP directory, because it depends on clock synchronization and user accounts must be manually created on each machine, what is not reasonable in a dynamic environment as cloud computing.

The *dispatcher* module runs as the root user, it spawns user processes which drop the administrative rights, using only the credentials provided on the *security token*. It evaluates *security tokens* in the SAML format, and considers tokens signed by the trusted *STS* to be authentic.

The *STS* implements a scheme for deriving unique user identification from the *SaaS user's* name (e.g. applying a hash function). This identification is then registered on the LDAP directory, and the operating systems are notified to synchronize their data with the directory server. It then creates SAML assertions and signs them with its private key.

The *interceptor* is a servlet that acts as a proxy. It receives requests from *SaaS users* that want to access the *SaaS application* interface (i.e. an URL pointing to a provided applet). The URL is protected by a module implementing the OpenID authentication, so the user must go through the authentication procedure, which ends up creating an authenticated session in the servlet. The *interceptor* interacts with the *STS*, and requests the *security token* using the WS-Trust specification. After successfully receiving a *security token*, it stores the token on a session attribute for further

reference; sessions that already contain a *security token* does not need to repeat this step. The token is attached to the application request before it is forwarded to the *dispatcher*.

The *SaaS user's* application is composed of an applet that executes inside an OpenID authenticated session. The application uses SOAP as its protocol, using the JAX-WS implementation. When a request is made, the *interceptor* considers it just an XML document. However, we are able to embed SOAP headers on it before it arrives on the *dispatcher*. Thus, the *SaaS user* herself is unaware of the credential translation taking place. The proposal could, as well, support Shibboleth users by providing a different *interceptor* able to interpret this protocol.

Even though we are working mainly with Linux-based operating systems, most modern operating systems have mechanisms to enable impersonating other users when given the appropriate credentials. Besides, LDAP is a very well supported technology, so the proposed architecture can adequately support different implementation scenarios.

We also devised the architecture to be application agnostic. That is, it can support SOAP based web services, HTTP based applications like CGI, REST, and so on. The use of web browsers was selected to show that it is possible to offer integral federated identity management transparently, though without losing the single sign-on functionality.

V. CONCLUSIONS

In this work we presented a new architecture and platform for federated identity management, mapping high level identities (i.e. from SaaS) to low level identities (i.e. for IaaS). By eliminating this integration gap we enable the SaaS application developer to track resource usage on a user based fashion, that is, it is possible to do fine grained accounting.

Besides, the application developer also gains advantages as: possibility of applying security policies for individual users on the infrastructure level; obtain accurate auditing records, as the users are known on the whole cloud computing layer; get the ability to deploy applications on multiple IaaS providers without losing identity unity.

The end user is not disturbed by the presence of the proposed architecture. We showed that it is possible to use interceptors to offer credential translation transparently. This also permits the proposal to support many authentication approaches in the front-end by just changing the interceptor.

The implementation aspects of prototype showed that the proposal is reasonable. There are plenty of freely available technologies and components, based on open standards, which supports the implementation of the proposed architecture.

References

[1] Shim, S.S.Y.; Geetanjali Bhalla; Vishnu Pendyala; "Federated identity management," Computer , vol.38, no.12, pp. 120- 122, 2005.

[2] Morgan, R. L.; Cantor, S.; Carmody, S.; Hoehn, W.; Klingenstein, K.; "Federated Security: The Shibboleth Approach," EDUCAUSE Quarterly, vol. 27, no. 4, pp. 12-17, 2004.

[3] OASIS; "Web Services Federation Language (WS-Federation) Version 1.2," Available at: http://docs.oasis-open.org/wsfed/federation/v1.2/ws-federation.html, Retrieved on January, 2012.

[4] Mikkonen, H.; Silander, M.; "Federated Identity Management for Grids," International conference on Networking and Services, pp. 69, 2006.

[5] Badger, L.; Grance, T.; Patt-Corner, R.; Voas, J.; "Cloud Computing Synopsis and Recommendations," Available at: http://csrc.nist.gov/publications/drafts/800-146/Draft-NIST-SP800-146.pdf, Retrieved on January, 2012.

[6] OpenID Foundation; "OpenID Authentication 2.0," Available at: http://openid.net/specs/openid-authentication-2_0.html, Retrieved on January, 2012.

[7] Steiner, J.G.; Neuman, B.C.; Schiller, J.I.; "Kerberos: An Authentication Service for Open Network Systems," Proceedings of the Usenix Conference, 1988.

[8] Needham, R. M.; Schroeder, M. D.; "Using encryption for authentication in large networks of computers," Commun. of the ACM. vol. 21, no. 12, pp 993-999, 1978.

[9] Neuman, B.C.; Ts'o, T.; "Kerberos: An Authentication Service for Computer Networks," IEEE Communications, vol. 32 no. 9, pp 33-38, 1994.

[10] Koshutanski, H.; Ion, M.; Telesca, L.; , "Distributed Identity Management Model for Digital Ecosystems," The International Conference on Emerging Security Information, Systems, and Technologies, pp.132-138, 2007.

[11] GlassFish Community; "The JAX-WS Reference Implementation," Available at: http://jax-ws.java.net/, Retrieved on January, 2012.

[12] The Apache Software Foundation; "Apache Tomcat," Available at: http://tomcat.apache.org/, Retrieved on: January, 2012.

[13] The Apache Software Foundation; "Apache Axis2/Java," Available at: http://axis.apache.org/axis2/java/core/, Retrieved on: January, 2012.

[14] The Apache Software Foundation; "Apache Axis2 Security Module," Available at: http://axis.apache.org/axis2/java/rampart/, Retrieved on January, 2012.

[15] OpenLDAP Foundation, "OpenLDAP Software," Available at: http://www.openldap.org/, Retrieved on Jan 2012.

[16] OpenSAML; "Open Source SAML Implementation," Available at: https://wiki.shibboleth.net/confluence/display/OpenSAML/Home, Retrieved on: January, 2012.

[17] Eucalyptus Systems, Inc.; "Eucalyptus Open Cloud Platform," Available at: http://open.eucalyptus.com/, Retrieved on: January, 2012.

[18] Debian Foundation; "Debian GNU/Linux," Available at: http://www.debian.org, Retrieved on: January, 2012.

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

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

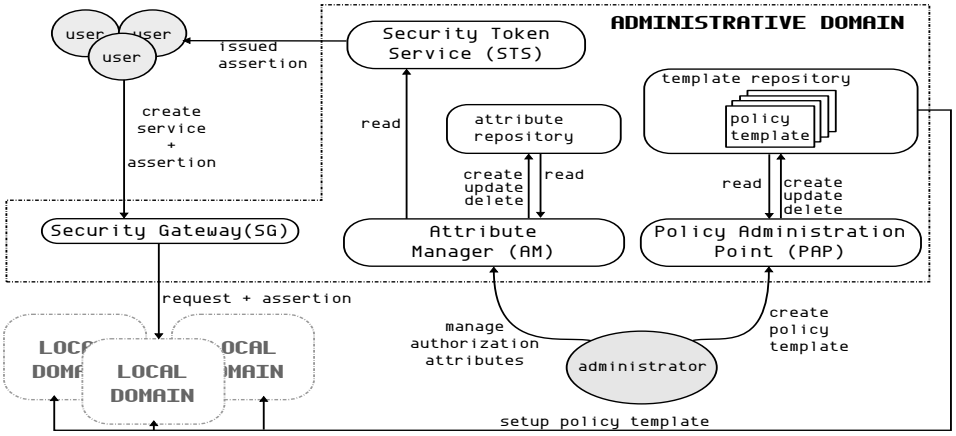

Fig. 1. Administrative domain components

(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-

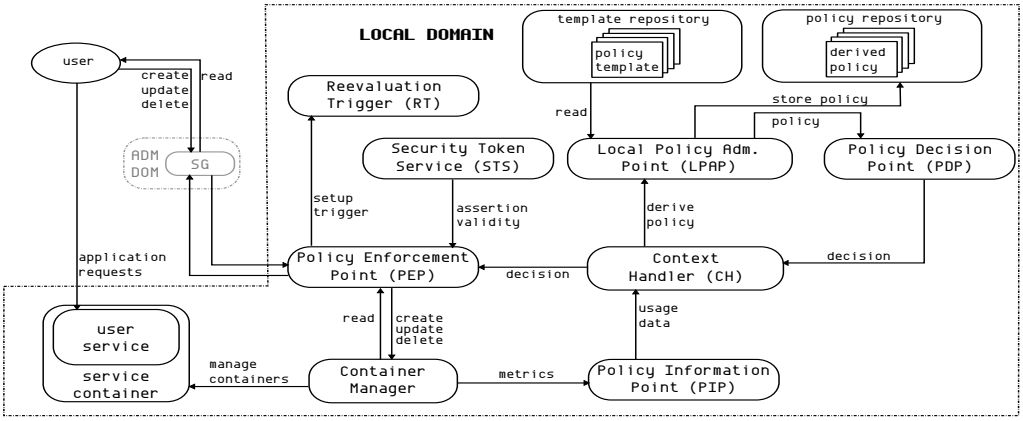

Fig. 2. Local Domain Components

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.

```
1 <Rule RuleId="CPURule" Effect="Permit">
2   <Target><Any/></Target>
3   <Condition>
4     <Apply FunctionId="integer-less-than-or-equal">
5       <Apply FunctionId="integer-one-and-only">
6         <AttributeDesignator Category="access-subject"
7         AttributeId="usedCpu" DataType="integer"/>
8       </Apply>
9       <Apply FunctionId="integer-one-and-only">
10        <AttributeValue DataType="integer">
11          ${TotalCpuTime}
12        </AttributeValue>
13      </Apply>
14    </Apply>
15  </Condition>
16 </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

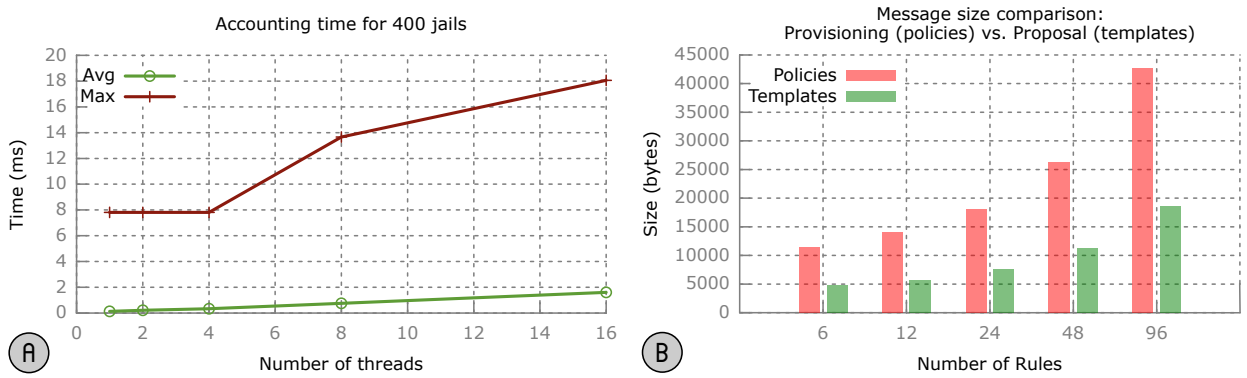

Fig. 4. Prototype evaluation for accounting overhead (A) and message size (B)

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.