Scalable Authorization Middleware for Service Oriented Architectures

Tom Goovaerts, Lieven Desmet, and Wouter Joosen

IBBT-Distrinet
Katholieke Universiteit Leuven
3001 Leuven, Belgium
{tomg,lieven,wouter}@cs.kuleuven.be

Abstract. The correct deployment and enforcement of expressive attribute-based access control (ABAC) policies in large distributed systems is a significant challenge. The enforcement of such policies requires policy-dependent collaborations between many distributed entities. In existing authorization systems, such collaborations are static and must be configured and verified manually by administrators. This approach does not scale to large and more dynamic application infrastructures in which frequent changes to policies and applications occur. As such, configuration mistakes or application changes might suddenly make policies unenforceable, which typically leads to severe service disruptions.
We present a middleware for distributed authorization. The middleware provides a single administration point that enables the configuration and reconfiguration of application- and policy-dependent interactions between policy enforcement points (PEPs), policy decision points (PDPs) and policy information points (PIPs). Using lifecycle and dependency management, the architecture guarantees that configurations are consistent with respect to deployed policies and applications, and that they remain consistent as reconfigurations occur. Extensive performance evaluation shows that the runtime and configuration overhead of the middleware scale with the size and complexity of the infrastructure and that reconfigurations cause minimal disruption to the involved applications.

Keywords: attribute-based access control, policy enforcement, policy deployment, middleware, service-oriented architectures

1 Introduction

In open and dynamic distributed applications, it is common practice to enforce fine-grained authorizations using the Attribute-Based Access Control (ABAC) [13] model. The eXtensible Access Control Markup Language (XACML) [8] is the most widely deployed and used ABAC language in practice. In ABAC, policies are specified declaratively in terms of rules based on the attributes of the subject, resource and environment. Logically, the enforcement of ABAC policies is performed by three types of collaborating components, which we call *authorization* *components*: Policy Enforcement Points (PEPs) intercept access attempts, request authorization decisions and enforce those decisions, Policy Decision Points (PDPs) make authorization decisions for PEPs, and Policy Information Points (PIPs) provide values for required attributes. This work focuses on attributes that are pulled in by the PDP. Therefore, PEPs depend on PDPs for authorization decisions, and PDPs depend on PIPs for attributes.

In practice, managing and maintaining the effective deployment and enforcement of ABAC policies is a significant challenge. Besides the need for implementing custom PEPs and PIPs to bind policies to the business logic, policies must also be distributed to the relevant PDPs, and the interactions between PEPs, PDPs, and PIPs must be configured and managed such that the policies are correctly enforced. This problem is further complicated by the fact that (1) PEPs, PDPs and PIPs are located on distinct distributed nodes in the system and therefore must interact remotely and (2) frequent policy and application changes occur that affect the dependencies between authorization components and therefore require updates to the required interactions.

Existing work on ABAC enforcement mostly focuses on isolated PDP implementation issues, and does not address the fact that ABAC policy enforcement is effectively a complex and dynamically evolving workflow between distributed authorization components. The practical consequence is that PEP, PDP and PIP interactions are static and must be configured and verified manually by the security administrator. This approach does not scale to large and complex infrastructures, and can lead to severe application disruptions because policies might suddenly become unenforceable.

The main contribution of this paper is an extended and scalable distributed authorization middleware that supports the configuration and reconfiguration of interactions between remote authorization components in response to policy or application changes. To guarantee and maintain correctness of the configuration, the system performs dependency and lifecycle management based on policy evaluation contracts that state the capabilities and requirements of the authorization components. The middleware has been prototyped on the Apache ActiveMQ message broker. Extensive performance measurements show that the policy enforcement as well as the reconfiguration overhead scale to large and complex infrastructures of hundreds of authorization components and thousands of dependencies.

The rest of the paper is structured as follows. Section 2 discusses an example derived from an industrial collaboration that illustrates the configuration and policy deployment challenges that must be met. Section 3 presents the architecture of our middleware and Section 4 discusses its prototype implementation. We evaluate the presented solution in Section 5 and discuss related work in Section 6. Section 7 concludes the paper.

2 Motivation

This section illustrates the maintenance challenges of ABAC policies in a case study on a personal content management system. People acquire gigabytes of personal digital content (documents, pictures, videos, etc.) and store it on many different locations. The PeCMan project [5] has developed a Personal Content Management (PCM) platform that offers a uniform user interface for managing and sharing personal content that is scattered over various devices and services. A core feature of the PCM system is that it is open and extensible with value-added services such as cloud storage, web publishing, face recognition and social network integration.

In order to protect access to services and content, the PCM system supports two types of authorization policies. First, system-level policies are defined by the owner of the PCM system and protect access to internal and third party services. An example of a system-level rule is: *documents uploaded to storage provider X may not be larger than 100MB*. Secondly, via a high level graphical editor, end-users themselves can define fine-grained policies that control access to shared content. An example policy rule is: *grant read access to my pictures only to friends that are older than 12*.

Figure 1 shows a simplified deployment view of a PCM system. The core system consists of an *index service* that keeps track of content location, a *metadata service* that stores, caches and searches content metadata, a *workflow engine* that orchestrates and composes value-added services into reusable processes and a *controller* that processes and mediates incoming client requests. HTTP-based clients are handled by a servlet in the web tier. Value-added services or adapters to external services are deployed on service containers on separate nodes.

Figure 1 also overlays the authorization infrastructure with the required authorization components, consisting of three PDPs and multiple application-specific PEPs and PIPs. The solid arrows show the dependencies between the authorization components. It can be observed that there are already a large number of dependencies to manage in this (simplified) infrastructure. Furthermore, system must deal with two types of frequently occurring changes:

Policy changes Because policies are system as well as user driven, policy changes occur very frequently. A new or updated policy may require new types of attributes. This requires a reconfiguration of the involved PDP with a PIP that offers those attributes. Second, a policy can apply to other types of resources. In this case, the administrator has to ensure that all PEPs that protect those resource must use the PDP.

Application changes Application changes (for instance the deployment of a new third-party service) can introduce new PEPs that must be bound to one or more PDPs, they can change the interceptions and the required PDPs of an existing PEP, or they can break application-level PIPs that are required by existing PDPs.

To deal with such evolving, distributed software environments, the authorization infrastructure must be capable of (1) easily adding, reconfiguring or re-

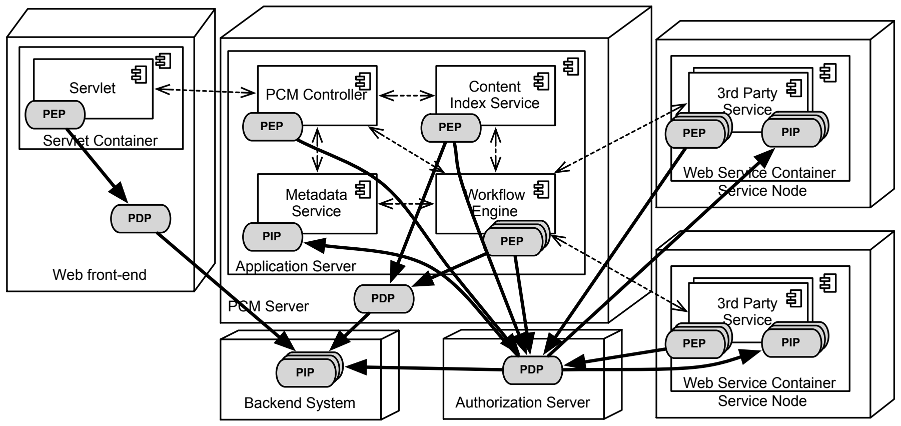


Fig. 1. Deployment view of the PeCMan architecture overlaid with the necessary authorization components and their dependencies. The dashed arrows represent architecture-level dependencies and the bold arrows represent dependencies between authorization components.

moving authorization components, as well as (2) guaranteeing the consistency of the deployed policies and their composed components. To the best of our knowledge, current authorization infrastructures, albeit modularized, do not support the necessary level of flexibility when changes must be managed and deployed because authorization components are tightly coupled to each other. Automatic (re)configuration has not been addressed in this context.

In addition, the authorization infrastructure must scale to a large number of authorization components and dependencies without sharp performance degradations, and changes to the authorization components or their interactions (triggered by policy changes or application changes) should impose a minimal disruption of the authorization system and of the applications. In the following section, we present the architecture of our authorization middleware, taking these requirements into account.

3 Architecture

An overview of the architecture of the proposed authorization middleware is shown in Figure 2. The middleware consists of three core ingredients. First, the PEPs, PDPs and PIPs are represented as first class components that are governed by a well-defined lifecycle model. Second, a message-based distribution layer mediates and supports the remote interactions between authorization components and supports flexible and efficient runtime adaptations to those interactions. Third, the authorization middleware provides a management component that functions as a centralized administrative interface. The management component is responsible for configuring and reconfiguring PEPs, PDPs and PIPs and their interactions while guaranteeing that configurations are consistent with respect to the deployed policies and applications. Section 3.1 discusses the authorization components and the lifecycle model in detail. Section 3.2 discusses the distribution layer and Section 3.3 discusses the management component.

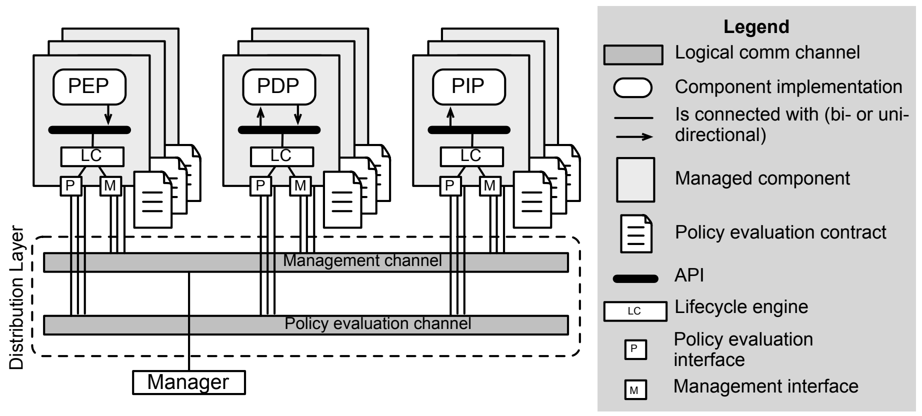


Fig. 2. Overview of the authorization middleware.

3.1 Managed Authorization Components

Authorization components are treated as first-class entities. As shown in Figure 2, an authorization component consists of several elements.

Each component exposes two interfaces towards the rest of the authorization middleware. The *policy evaluation interface* is used during policy evaluation and implements generic and stable protocols that define the interactions between PEPs, PIPs and PDPs: requesting and returning authorization decisions and missing attribute values. This interface is refined for each individual authorization component by means of a *policy evaluation contract* (further abbreviated as contract). The contract lists the dependencies and capabilities of each authorization component in terms of which authorization and attribute requests are required or provided. We refer to [4] for a detailed description of these contracts. Secondly, the *management interface* is used by the centralized manager to inspect and modify the states and to control the incoming and outgoing interactions of the authorization components. The management interface also has operations to distribute policies to PDPs.

In addition, the middleware uses a lifecycle model for authorization components, which is shown in Figure 3 and is implemented by the lifecycle engine (see Figure 2). The lifecycle model is similar to dynamic module systems such as OSGi[9] and Java Business Integration (JBI)[10] and consists of four states.

Initially, each component publishes its *capability contract* to the manager, which can then then deploy the component under a stricter version of its contract (with less provided elements), that is called the *deployed contract.* A component is only allowed to participate in policy evaluation once it has been activated by the manager.

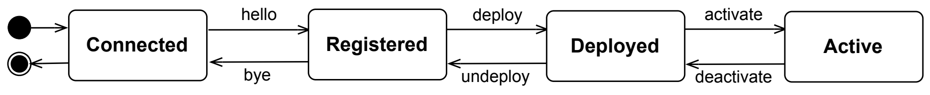


Fig. 3. State machine diagram of the authorization component life cycle model.

The middleware also provides an API for implementing authorization components that maps the underlying message-based interface to an object-oriented interface and that makes the entire middleware appear as a single PDP or PIP.

3.2 Distribution Layer

The distribution layer mediates the remote interactions between authorization components and the manager. Figure 2 shows that the distribution layer provides two logical channels, one for policy evaluation and one for management.

Typically, the interactions between authorization components are implemented using remote method invocations. However, this model tightly couples the authorization components in space: PEPs need to know which PDPs to contact and PDPs need to know which PIPs to contact. This makes it hard to dynamically adapt the interactions. Therefore, we have opted for a messaging-based distribution layer that loosely couples authorization components. Authorization and attribute requests are sent to the distribution layer as destination-less messages. The distribution layer automatically routes requests to compatible components. Routing is configured using component-side *subscriptions filters* that indicate which resource or attribute types the component is interested in. The subscription filters are completely shielded from the component implementation and can only be influenced by the lifecycle engine (and, thus, indirectly by the manager). Filters are derived from the policy evaluation contracts and can be updated dynamically.

3.3 Manager

The flexibility of the routing of authorization and attribute requests provided by the distribution layer ensures that any component can be dynamically connected to any other component at runtime. The main responsibility of the manager component is to deploy and activate authorization components and ensure that (re)configurations are carefully controlled such that the infrastructure is and remains able to enforce all deployed policies. At its core, the manager consists of a set of primitives that allow administrators to safely configure the system, load policies to remote PDPs and perform efficient reconfigurations in response to policy and application changes. The effective component-side (re)configurations are implemented as state changes and their execution is delegated to the lifecycle engines of the deployed authorization components. In the rest of this section, we discuss each of the management primitives in detail.

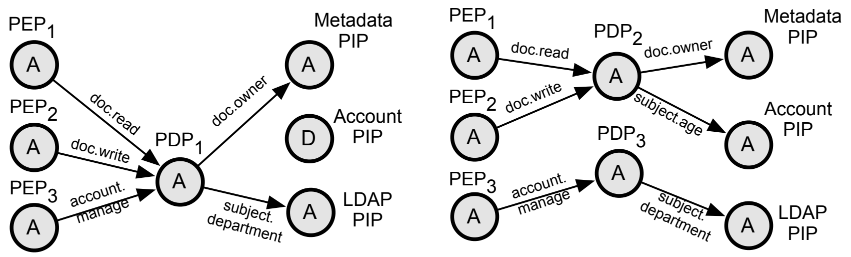
Deployment/Undeployment The deployment primitive assigns a deployed contract to a component. The deployed contract can provide less attributes or resource types in order to resolve conflicts (see further). Likewise, undeployment removes the contract associated with a given component.

Cascading Activation/Deactivation A deployed component needs to be activated before it can communicate with other components. The manager guarantees the following invariant for the set of services that are active: *each possible authorization or attribute request made by a component is handled by exactly one other component.* The manager uses the algorithm described in [4] to compute a dependency graph of all deployed components, based on the dependencies described in the contracts. When activating one component, the manager uses the dependency graph to obtain all components that are directly or indirectly required by the given component. Each of those components that is not already active, is also activated. In the case that it is impossible to satisfy all dependencies, the activation request is refused. If there is a conflict because multiple components handle the same requests (eg. two PIPs provide the same attribute), a selection must be made of one of the providers.

Figure 4(a) shows a dependency graph of a part of the PCM system. In case no single component is activated yet, the activation of PEP_2 will trigger the activation of the PDP and its two required PIPs. Deactivation occurs similarly, except that a cascading deactivation is performed of all components that directly or indirectly require the given component. Thus, deactivating the LDAP PIP also deactivates the PDP and all PEPs.

It can be the case that it is impossible to find a proper selection because of overlaps. For instance, suppose that the PDP to activate requires the attributes `age`, `location` and `presence` and that one PIP provides `age` and `location` and another PIP provides `location` and `presence`). Such conflicts must first be resolved by restricting the deployed contract of one of the two PIPs.

Contract and Policy Updates We reconsider the example shown in Figure 4(a) in case of a policy update. Suppose that the document policy is updated with a new rule that is based on the `subject.age` attribute. The policy update is reflected in a contract update that makes the PDP dependent on that attribute. Suppose that the Account PIP deployed and provides `subject.age`. With only the basic management primitives, implementing such an update would require the deactivation the PDP, including all depending PEPs, change the contract and then reactivate the PDP and PEPs. To limit disruption in such cases, the



(a) Basic example topology. (b) Desired topology after migration.

Fig. 4. Dependency graphs of example topologies. PDP$_1$ is the Authorization Server PDP, PEP$_1$ and PEP$_2$ are third party service PEPs and PEP$_3$ is the PCM Controller PEP.

manager implements an optimized `updateContract` primitive. Because changes that trigger contract updates originate in the components, the components can trigger the contract update themselves.

We only discuss the non-trivial case in which the involved component is active. The `updateContract` primitive takes the affected component (with old capability and deployed contracts c_c and c_d) and the new capability contract c'_c as parameters and consists of the following steps:

1. Generate a new deployed contract c'_d that requires the same elements as c'_c and that provides the same elements as c'_c, except for the provided elements that c_d also omitted from c_c.
2. Remove all provided elements from c'_d that are already provided by other active components.
3. Try to replace c_d with c'_d in the dependency graph of active components.
   – If this maintains the invariant (one unique provider for every required element), the update is allowed and c'_d is loaded to the component.
   – If additional elements are required that are uniquely provided by deployed components, those deployed components are activated and then c'_d is loaded to the component.
   – If the new contract breaks incoming dependencies that can be uniquely satisfied by other deployed components, activate those components and then load c'_d to the component.
   – Otherwise, the update is refused and manual deconfliction is required.

If the algorithm is applied to the example, the PIP that provides `subject.age` would be activated and the PDP is briefly interrupted while committing the updated contract.

Migration Consider the basic topology after the update (the topology of Figure 4(a) with an added dependency of the PDP on the Account PIP for `subject.age`). Suppose that PDP$_1$ has become a performance bottleneck and that we want to move its policies to two new PDPs, called PDP$_2$ and PDP$_3$.

The desired topology after the migration is shown in Figure 4(b). The implementation of such a reconfiguration would again require the de- and reactivation of all PDPs and PEPs. The manager provides a `migrate` primitive to deal with migrations like these efficiently.

The migration algorithm migrates from a set of active components C to a set of deployed components C' and goes as follows:

1. Check for internal violations within C'. No element required by components in C' may be provided by more than one component in C'.
2. Check for external violations by verifying that the dependency graph for the components after the migration does not invalidate the invariant.
3. If these two conditions hold, deactivate all components of C and activate all components of C'. Otherwise refuse the migration and report the violating dependencies.

4 Prototype

We have built a prototype of our architecture in Java. The size of the prototype is around 12000 lines of code. The prototype is designed to be extensible with different implementations of the distribution layer. The distribution layer is based on the Java Message Service (JMS) API and Apache's ActiveMQ implementation thereof. We use the topic-based publish/subscribe of the JMS API for message delivery between components. Each JVM maintains one session with ActiveMQ to publish messages and one session per component for receiving messages. Per action on a resource type and per attribute type, there is one JMS topic. Components publish requests to the topic that corresponds to the resource or the attribute. Authorization and attribute requests are represented as objects that are embedded in JMS object messages. Because of the large number of topics, one change was made to the default configuration so that topics are processed by a thread pool instead of a single thread per topic. All JMS messages are sent non-persistently.

We have also developed an XACML-based PDP that is based on Sun's XACML implementation (version 1.2). Attribute retrieval is integrated by means of a custom `AttributeFinder` module that uses our middleware. Furthermore, to study the middleware in isolation from the environments and policy engines it must be integrated with, we have developed synthetic PEP, PDP and PIP components. These synthetic components do not contain actual interception, policy evaluation or attribute retrieval logic, but simulate it. The synthetic components are fully configurable and allow us to instantiate and study the behavior of the middleware for large topologies.

5 Evaluation and Discussion

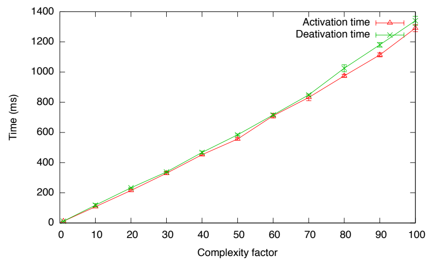
In this section, we evaluate the scalability of the proposed middleware by measuring the impact of the complexity of the instantiated authorization infrastructure on the performance of the supported interactions and reconfigurations. Furthermore, we discuss the applicability of the middleware in terms of the approach-specific development efforts and we indicate how these efforts can be reduced or partially automated.

One of the core goals of the proposed middleware is to scale to large and complex setups that are infeasible to manage manually. We have evaluated the scalability of (re)configuration operations as well as the runtime performance impact of the middleware in a distributed setup that consists of 12 identical Pentium 4 2GHz machines with 512MB RAM, interconnected by a 100Mbps switch and running a vanilla Kubuntu 10.04 install with the Sun Java 1.6.0_20 JVM. One node runs the ActiveMQ server, another node runs the manager component and collects the results, and the other 10 nodes run authorization components. To scale the complexity of the infrastructures, we have used the synthetic authorization components discussed in Section 4. We use the notation $[X, Y, Z]$ to represent an authorization infrastructure consisting of X PEPs, Y PDPs and Z PIPs. In the rest of this section, we discuss the obtained results in detail in logical chronological order.

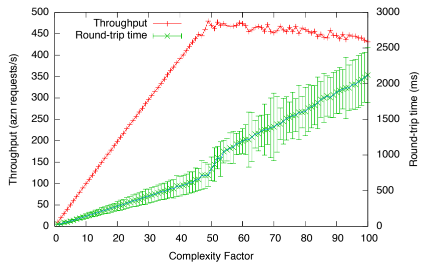
Activation/deactivation Before an authorization infrastructure can be used, it must first be activated. Therefore, we first study the scalability of the activation and deactivation times of an entire infrastructure in function of its complexity. We have measured the time it takes to activate and deactivate 10 infrastructures of increasing complexity, where the simplest configuration of $[10, 1, 2]$ is multiplied by a factor of 1 to 100 in steps of 10. The setup has worst-case dependencies (each PEP depends on all PDPs and each PDP depends on all PIPs). Figure 5(a) shows the resulting activation and deactivation times. For the largest setup of $[1000, 100, 200]$, the activation and deactivation times are both approximately 1.2s. The performance increase is polynomial because dependency calculations are affected by both the increase in number of contracts as well as number of dependencies. However, the relative overhead of the dependency calculations is very small in comparison to the communication overhead of the (de)activation.

Performance degradation The distribution overhead caused by the middleware once components are collaborating must degrade gracefully with the complexity of the setup. Therefore, we have measured the throughput and mean round-trip times of authorization requests in setups of increasing complexity. The same basic setup of $[10, 1, 2]$ was used as in the previous experiment, but the complexity factor was increased in steps of 1. Each PEP sends one random request per second and each PDP requests two random attributes per authorization request in parallel. From the results in Figure 5(b), we observe that the setup is unsaturated for all infrastructures smaller than $[500, 50, 100]$. From then on, the CPU of the JMS server becomes the bottleneck and the throughput starts to degrade from its maximum of 470 requests/s to 430 requests/s for $[1000, 100, 200]$. As desired, this degradation is graceful. Similarly, the round-trip times degrade linearly but the slope slightly increases after the saturation point.

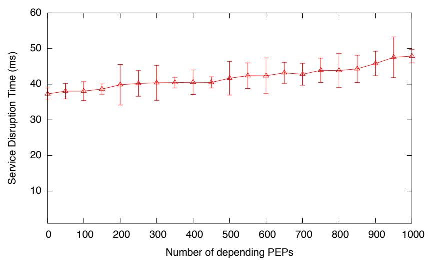
Contract Update Next, we consider the scalability of a dynamic policy (and, by consequence, contract) update in a running infrastructure with 1 PDP that



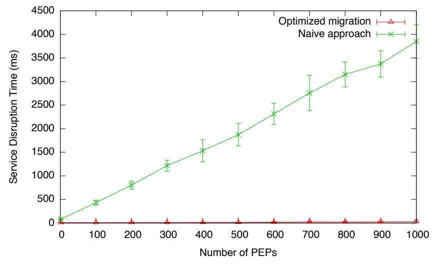
(a) Activation and deactivation time vs. infrastructure complexity.



(b) Round-trip times and throughput vs. infrastructure complexity.



(c) Service disruption time for a contract update of one PDP in function of the number of depending PEPs



(d) Service disruption time for migrating from one PDP to another in function of number of depending PEPs.

Fig. 5. Scalability results.

depends on 50 of 100 PIPs. Such an update briefly disrupts the PDP and this disruption time must scale with the number of PEPs that depend on the PDP. To evaluate this, we have changed the number of depending PEPs from 1 to 1000 in steps of 50. For each infrastructure, the PDP contract is updated 20 times to depend on a random subset of 50 PIPs. The mean of the resulting disruption times are reported in Figure 5(c). We observe that the disruption times are always lower than 50ms and that they increase linearly with the number of depending PEPs. The disruption time in case of 1000 depending PEPs is only 25% higher than for one depending PEP.

Migration In the second reconfiguration experiment, we study the scalability of component migration. The base setup consists of one PDP that depends on 50 of 100 PIPs and that is migrated to a second PDP that depends on a different subset of 50 PIPs, which disrupts all depending PEPs. The performance of the migration must scale with the number of PEPs that depend on the PDPs, which is varied from 1 to 1000 in steps of 100. We compare the optimized primitive versus a naive approach in which the setup is first completely deactivated and then reactivated. Figure 5(d) shows the resulting disruption times. The disruption time caused by the naive approach scales linearly with respect to the number of PEPs and takes almost 4 seconds for a 1000 depending PEPs. The optimized migration algorithm drastically reduces the disruption time to a maximum of 30.9 ms by avoiding cascading deactivation. The optimized version is still scales linearly, but the slope is so small that it is not visible in the plot.

The integration of a distributed application with the proposed authorization middleware requires some additional effort. First, custom PEPs and PIPs must be implemented. However, since this is not specific to our approach, we like to refer to existing work that addresses this problem [11, 1]. Secondly, each authorization component needs to be enhanced with a contract. This effort can be reduced because PDP contracts can be derived automatically from the policies based on syntactic analysis and because PEP/PIP contracts can be codeveloped with the components [11].

6 Related Work

Because SOAs are typically built using Enterprise Service Buses (ESBs), policies are often enforced at the the ESB level [3]. Such approaches work well for protocol-level policies but are not suited for enforcing more expressive higher level policies. Therefore, we see both approaches as complementary enforcement strategies for different types of policies.

To the best of our knowledge, there are two authors that have proposed to use messaging for interactions between security components. McDaniel [7] proposes a flexible security policy enforcement architecture for the Antigone group communication system in which security mechanisms are composed over an event bus. Wei [12] proposes the publish-subscribe model between remote PEPs and PDPs and confirms that this improves flexibility and availability. These approaches do not consider the consistency aspect. In our opinion, the flexibility gains of messaging must be carefully controlled by higher order management functionality that preserves consistency.

The deployment model of the Ponder policy language [2] manages policy deployment in distributed settings. Policies are translated into native access control configurations and are enforced local to the protected resources. This reduces the runtime overhead in comparison to or approach, but makes updates more complex: each policy update or new enforcement component requires policy redeployment. Moreover, although remote attribute retrieval could be implemented, it must be mapped to all native mechanisms, which is not always feasible. Additionally, if policies are heavily dependent on remote attributes, native enforcement might not be the best choice from a performance standpoint.

The run-time reconfiguration process discussed in this paper only ensures that a correct composition is achieved at the end of the reconfiguration process. In contrast and as a complement, safe reconfiguration approaches for distributed services (such as [6]) can guarantee that no single request is lost during the reconfiguration process or is processed by an incorrect composition.

7 Conclusion

We have presented a distributed authorization middleware for ABAC policy evaluation. The middleware supports the interactions between remotely deployed PEPs, PDPs and PIPs and manages their policy- and application-specific dependencies. By guaranteeing the consistency of the configuration with respect to the deployed policies, the system supports dynamic (re)configuration of the authorization infrastructure in response to policy or application changes. We have shown that the distribution and reconfiguration overhead scale to hundreds of authorization components and thousands of dependencies. In future work, we will improve the flexibility of the selection of required components and we will extend the system with more automated management processes such as PDP load balancing or performance optimization through automatic policy redistribution.

Acknowledgements. This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, IBBT, IWT and the Research Fund K.U. Leuven.

References

1. Konstantin Beznosov. Object security attributes: Enabling application-specific access control in middleware. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 693–710, London, UK, 2002. Springer-Verlag.
2. N. Dulay, E. Lupu, M. Sloman, and N. Damianou. A policy deployment model for the ponder language. In *Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on*, pages 529–543, 2001.
3. G. Gheorghe, S. Neuhaus, and B. Crispo. xESB: An Enterprise Service Bus for Access and Usage Control Policy Enforcement. *Trust Management IV*, pages 63–78, 2010.
4. T. Goovaerts, B. De Win, and W. Joosen. Policy Evaluation Contracts. Technical report, Department of Computer Science, Katholieke Universiteit Leuven, 2009.
5. IBBT. PeCMan project (Personal Content MANagement). `http://projects.ibbt.be/pecman`, 2007.
6. N. Janssens, W. Joosen, and P. Verbaeten. Necoman: middleware for safe distributed-service adaptation in programmable networks. *Distributed Systems Online, IEEE*, 6(7), 2005.
7. P. McDaniel and A. Prakash. A flexible architecture for security policy enforcement. *DARPA Information Survivability Conference and Exposition, 2003. Proceedings*, 2, 2003.
8. OASIS. eXtensible Access Control Markup Language (XACML) Version 2.0, December 2005.
9. OSGi Alliance. OSGi Service Platform, Core Specification, Release 4, Version 4.1, May 2007.

10. Rod Ten-Hove and Peter Walker. Java Business Integration (JBI) 1.0 Final Release, August 2005.
11. T. Verhanneman, F. Piessens, B. De Win, E. Truyen, and W. Joosen. A modular access control service for supporting application-specific policies. *IEEE Distributed Systems Online*, 7, 2006.
12. Q. Wei, M. Ripeanu, and K. Beznosov. Authorization using the publish-subscribe model. In *Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 53–62, Washington, DC, USA, 2008. IEEE Computer Society.
13. E. Yuan and J. Tong. Attributed based access control (ABAC) for web services. *Web Services, IEEE International Conference on*, pages 561–569, 2005.