# Towards Improving the Availability and Performance of Enterprise Authorization Systems

by

Qiang Wei

B.Eng., The University of Science and Technology Beijing, 1998
M.Eng., Nanyang Technological University, 2003

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

The Faculty of Graduate Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

October, 2009

# Abstract

Authorization protects application resources by allowing only authorized entities to access them. Existing authorization solutions are widely based on the request-response model, where a policy enforcement point intercepts application requests, obtains authorization decisions from a remote policy decision point, and enforces those decisions. This model enables sharing the decision point as an authorization service across multiple applications. But, with many requests and resources, using a remote shared decision point leads to increased latency and presents the risk of introducing a bottleneck and/or a single point of failure. This dissertation presents three approaches to addressing these problems.

The first approach introduces and evaluates the mechanisms for authorization recycling in role-based access control systems. The algorithms that support these mechanisms allow a local secondary decision point to not only reuse previously-cached decisions but also infer new and correct decisions based on two simple rules, thereby masking possible failures of the central authorization service and reducing the network delays. Our evaluation results suggest that authorization recycling improves the availability and performance of distributed access control solutions.

The second approach explores a cooperative authorization recycling system, where each secondary decision point shares its ability to make decisions with others through a discovery service. Our system does not require cooperating secondary decision points to trust each other. To maintain cache consistency at multiple secondary decision points, we propose alternative mechanisms for propagating update messages. Our evaluation results suggest that cooperation further improves the availability and performance of authorization infrastructures.

The third approach examines the use of a publish-subscribe channel for delivering authorization requests and responses between policy decision points and enforcement points. By removing enforcement points' dependence on a particular decision point, this approach helps improve system availability, which is confirmed by our analytical analysis, and reduce system administration/development overhead. We also propose several subscription schemes for different deployment environments and study them using a prototype system.

We finally show that combining these three approaches can further improve the authorization system availability and performance, for example, by achieving a unified cooperation framework and using speculative authorizations.

# Contents

**Abstract** . . . . . ii

**Contents** . . . . . iii

**List of Tables** . . . . . vi

**List of Figures** . . . . . vii

**Acknowledgements** . . . . . ix

**Dedication** . . . . . xi

**Statement of Co-Authorship** . . . . . xii

**1 Introduction** . . . . . 1
- 1.1 Architecture of authorization solutions . . . . . 2
- 1.2 Problem motivation . . . . . 4
- 1.3 Research goals . . . . . 5
- 1.4 Contributions . . . . . 6
- 1.5 Dissertation outline . . . . . 8

**2 Background and Related Work** . . . . . 9
- 2.1 Access control policies and models . . . . . 10
  - 2.1.1 The Bell-LaPadula model . . . . . 10
  - 2.1.2 The role-based access control model . . . . . 11
- 2.2 Access control system implementations . . . . . 12
- 2.3 General fault-tolerance techniques . . . . . 14
- 2.4 Using replication to improve availability and performance . . . . . 15
  - 2.4.1 Group replication . . . . . 15
  - 2.4.2 Caching . . . . . 16
  - 2.4.3 Cooperative caching . . . . . 16
- 2.5 Caching in access control systems . . . . . 17
  - 2.5.1 Caching decisions . . . . . 17
  - 2.5.2 Caching polices . . . . . 18
  - 2.5.3 Caching attributes . . . . . 19

2.6 Secondary and approximate authorization model . . . 20
2.6.1 SAAM$_{\text{BLP}}$ . . . 22
2.7 Summary . . . 22

**3 Recycling RBAC Authorizations . . . 23**
3.1 SAAM$_{\text{RBAC}}$ design . . . 24
3.1.1 Assumptions . . . 24
3.1.2 Preliminaries . . . 25
3.1.3 Inference rules . . . 26
3.1.4 Naive algorithms . . . 26
3.1.5 Cache compression . . . 27
3.1.6 Optimized algorithms . . . 28
3.1.7 Discussion . . . 31
3.1.8 Using the role hierarchy in SAAM$_{\text{RBAC}}$ . . . 33
3.1.9 Handling policy changes . . . 33
3.1.10 Evidence construction and verification . . . 38
3.1.11 Implementation considerations . . . 42
3.2 Experimental evaluation . . . 43
3.2.1 Simulation-based evaluation . . . 43
3.2.2 Prototype-based evaluation . . . 52
3.2.3 Evaluating hit rate with real-world data . . . 54
3.2.4 Discussion . . . 57
3.3 Summary . . . 59

**4 Cooperative Secondary Authorization Recycling . . . 60**
4.1 CSAR design . . . 60
4.1.1 Design requirements . . . 61
4.1.2 System architecture . . . 61
4.1.3 Discovery service . . . 63
4.1.4 Adversary model . . . 65
4.1.5 Mitigating threats . . . 65
4.1.6 Consistency . . . 67
4.1.7 Eager recycling . . . 69
4.2 Evaluation of CSAR$_{\text{BLP}}$ . . . 70
4.2.1 Simulation-based evaluation . . . 70
4.2.2 Prototype-based evaluation . . . 73
4.3 Evaluation of CSAR$_{\text{RBAC}}$ . . . 82
4.4 Summary . . . 83

**5 Authorization Using the Publish-Subscribe Architecture** . . . 85
5.1 The publish-subscribe architecture . . . 86
5.1.1 The event notification service . . . 86
5.1.2 Subscription schemes . . . 87
5.2 System design . . . 87
5.2.1 Design requirements . . . 88
5.2.2 System architecture . . . 88
5.2.3 Subscription/unsubscription schemes . . . 91
5.2.4 Security considerations . . . 95
5.2.5 Consistency considerations . . . 97
5.2.6 Integrating pub-sub with authorization recycling . . . 98
5.3 Evaluation . . . 100
5.3.1 Evaluating availability . . . 100
5.3.2 Evaluating performance . . . 103
5.4 Summary . . . 113

**6 Conclusion** . . . 114
6.1 Future work . . . 115

**Bibliography** . . . 117

# List of Tables

3.1 Building $Cache^+$ and $Cache^-$ from primary responses . . . . . . . . . . . . . . . . 30
3.2 Building $Evidence^+$ and $Evidence^-$ from primary responses . . . . . . . . . . . . . 42

# List of Figures

1.1 General access control model based on the reference monitor [TS01]. . . . . . . . 2

1.2 Authorization systems based on the request-response model. . . . . . . . . . . . . 3

1.3 The tight coupling between PEPs and PDPs leads to fragile authorization architectures [Bez05]. Any PDP failure will affect all the PEPs that depend on that PDP. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.1 A BLP example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.2 RBAC model. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

2.3 Access control system implementations [Bez05]. The PEP intercepts application requests and generates authorization requests. The PEP can be implemented by the underlying middleware or be a part of the application. . . . . . . . . . . . . . 13

2.4 SAAM adds SDP to the request-response model. . . . . . . . . . . . . . . . . . . 20

3.1 SAAM$_{\mathrm{RBAC}}$ optimized recycling algorithms . . . . . . . . . . . . . . . . . . . . . 29

3.2 The decision algorithm in a hierarchical setting . . . . . . . . . . . . . . . . . . . 34

3.3 Cache update algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

3.4 SAAM$_{\mathrm{RBAC}}$ revised algorithms for evidence construction . . . . . . . . . . . . . . 40

3.5 The evidence verification algorithm . . . . . . . . . . . . . . . . . . . . . . . . . 41

3.6 Comparing optimized algorithm and naive algorithm of approximate recycling (AR), with precise recycling (PR), for an RBAC system with 100 users, 3,000 permissions, and 50 roles. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

3.7 The impact of various system characteristics on the hit rate. . . . . . . . . . . . . 47

3.8 The impact of removing a role from $R$ on hit rate. In both figures, the order of the curves (from top to bottom) matches that of the legends. . . . . . . . . . . . 50

3.9 The impact of cache warmness on the inference and update time. . . . . . . . . . 51

3.10 Response time variation with cache warmness. . . . . . . . . . . . . . . . . . . . 54

3.11 The experiments with the HP policy . . . . . . . . . . . . . . . . . . . . . . . . 55

3.12 The experiments with the real trace . . . . . . . . . . . . . . . . . . . . . . . . 56

4.1 CSAR introduces cooperation between SDPs. . . . . . . . . . . . . . . . . . . . . 62

4.2 The architecture enabling the support for policy changes. . . . . . . . . . . . . . 68

4.3 The impact of various parameters on hit rate. The requests for subfigures (a)—(c) follow a uniform popularity distribution. . . . . . . . . . . . . . . . . . . . . 72

4.4 Response time as a function of the number of requests observed by SDPs. The requests follow a uniform distribution. . . . . . . . . . . . . . . . . . . . . . . . . 75
4.5 Response time comparison between overlap rate of 10% and 100%. The requests follow a uniform distribution. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
4.6 The impact of policy changes on hit rate with a single SDP. The requests follow a uniform popularity distribution. . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
4.7 The impact of SDP cooperation on hit rate when policy changes. The requests follow a uniform popularity distribution. The overlap rate between SDPs is 100%. 80
4.8 Adding CSAR-based policy enforcement to TPC-W. . . . . . . . . . . . . . . . 81
4.9 The impact of policy enforcement on response time. . . . . . . . . . . . . . . . 82
4.10 The impact of various parameters on hit rate . . . . . . . . . . . . . . . . . . . . 83

5.1 Publish-subscribe architecture for delivering authorization requests and responses. 89
5.2 Basic sequence diagram in resolving an authorization request. . . . . . . . . . . 90
5.3 Trade-offs bettween (un)subscription schemes. . . . . . . . . . . . . . . . . . . . 94
5.4 Pub-sub for the authorization recycling system. . . . . . . . . . . . . . . . . . . 99
5.5 Analytical results on availability: number of nines. . . . . . . . . . . . . . . . . 101
5.6 Analytical results on availability: percentage decrease of failed requests. . . . . 102
5.7 PEP publishes a request in Siena . . . . . . . . . . . . . . . . . . . . . . . . . . 104
5.8 PDP subscribes to a request in Siena . . . . . . . . . . . . . . . . . . . . . . . . 104
5.9 Response time comparison for the four PEP subscription schemes: request-based, subject-based, session-base, and callback. . . . . . . . . . . . . . . . . . . . . . . 107
5.10 Response time as a function of the number of subscriptions. . . . . . . . . . . . 108
5.11 Results for the session-based subscription scheme. . . . . . . . . . . . . . . . . . 109
5.12 The impact of concurrent PEP subscriptions on the throughput. . . . . . . . . . 110
5.13 Response time as a function of overlap rate, when 4 PDPs exist. . . . . . . . . . 111
5.14 Response time comparison with approximate recycling . . . . . . . . . . . . . . . 112

# Acknowledgements

I would like to gratefully and sincerely thank many people who have helped me during my graduate studies at the University of British Columbia. Without their guidance, support and patience, the writing of this dissertation would not have been completed.

My deepest gratitude is to Professors Konstantin (Kosta) Beznosov and Matei Ripeanu, my research supervisors. We had many enlightening discussions. I am indebted to Kosta for his guidance and tremendous help in idea development and presentation skills. He invests much time toward providing his students with opportunities to gain a wider breadth of experience, organizing reading groups, and activities to cultivate a harmonious environment. Kosta also gave support in work and life, financial and otherwise. I am also grateful to Matei, who became my co-supervisor despite his many other academic commitments. He spent many hours contemplating my ideas, always coming back with inspiring comments. His participation led to great progress.

I would also like to thank Professor Sathish Gopalakrishnan, who served on my supervisory committee. He was accessible for valuable discussions about the ideas in this dissertation, and provided insightful comments and constructive criticism.

I would also like to thank Professors William A. Aiello and Norm Hutchinson. They served on my university examination committee and have provided many insightful comments to improve this dissertation. I am also grateful to Professor Carlisle Adams, who served as the external examiner of my doctoral examination. He read this dissertation very carefully and has provided many helpful suggestions to further improve it.

I would like to extend my sincerest thanks to Professor Jason Crampton, who is a rigorous mathematician. I am lucky for the opportunity to have worked with Jason on the first essay in this dissertation; he taught me a lot about how to be systematic and exhaustive in theory development. Without his expertise, this dissertation would lack some crucial ingredients.

My thanks go to all my colleagues from the Laboratory for Education and Research in Secure Systems Engineering (LERSSE). I am very grateful for their friendship and assistance in all aspects. Knowing them made my life richer.

I would like to give my special thanks to my wife, Xiaohua, for her love and support all these years. She accompanies me on this journey, sharing my happiness and stress, even while we were separated by the Pacific Ocean. This dissertation would not have been possible without her.

The last, and surely not the least, I want to thank my parents and my sisters, who always supported, encouraged and believed in me. I thank my parents for allowing me to be as ambitious, and for making me feel proud of my accomplishments. Their support has meant much to me.

*To my parents, sisters, and to my wife, Xiaohua*

# Statement of Co-Authorship

The materials in Chapters 3, 4 and 5 of this dissertation have each been either published or submitted for publication. The author of this dissertation performed all the design and evaluation related to chapters 3, 4, and 5. He also co-authored the corresponding papers, under the supervision of the co-authors who provided feedback and guidance throughout the research process. Below are the details for each chapter.

- Chapter 3: A preliminary version of this chapter has been published. A full version of this chapter has been submitted to a journal for publication. The author of this dissertation wrote all the sections of this chapter, except Sections 3.1.4-5 and 3.1.7-8 which were mainly written by Prof. Jason Crampton.

  Q. Wei, J. Crampton, K. Beznosov, and M. Ripeanu, Authorization Recycling in Hierarchical RBAC Systems, under second review, *ACM Transactions on Information and System Security (TISSEC),* 32 pages, 2009.

  Q. Wei, J. Crampton, K. Beznosov, and M. Ripeanu, Authorization Recycling in RBAC Systems, in *the Proceedings of the 13th ACM Symposium on Access Control Models and Technologies (SACMAT'08)*, Estes Park, Colorado, 11-13 June, 2008, pp.63-72.

- Chapter 4: An extended abstract of this chapter was published as a poster and a preliminary version of this chapter has been published at conferences. A full version of this chapter has been published in the IEEE Transactions on Parallel and Distributed Systems. The author of this dissertation wrote all the sections of this chapter.

  Q. Wei, M. Ripeanu, and K. Beznosov, Cooperative Secondary Authorization Recycling, *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 20 n.2, February 2009, pp.275-288.

  Q. Wei, M. Ripeanu, and K. Beznosov, Cooperative Secondary Authorization Recycling, in the Proceedings of the *16th ACM/IEEE International Symposium on High Performance Distributed Computing (HPDC'07)*, Monterey, IEEE, 25-29 June, 2007, pp.65-74.

  Q. Wei, M. Ripeanu, and K. Beznosov, "Poster: Cooperative Secondary Authorization Recycling," poster at the *15th USENIX Security Symposium*, August 2006, Vancouver, Canada.

- Chapter 5: A preliminary version of this chapter has been published at an IEEE conference. A full version of this chapter has been submitted to a journal for publication. The author of this dissertation wrote all the sections of this chapter.

Q. Wei, M. Ripeanu, and K. Beznosov, Improving Availability of Distributed Authorization Infrastructures Using the Publish-Subscribe Model, under review, *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 32 pages, 2009.

Q. Wei, M. Ripeanu, and K. Beznosov, Authorization Using Publish/Subscribe Models, In *the Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA'08)*, 10-12 December, 2008, Sydney, Australia, pp.53-62.

# Chapter 1

# Introduction

Enterprise application systems have become a necessary infrastructure component in most organizations. By supporting various applications that are developed independently, and even based on different designs and technologies, enterprise application systems promise to integrate all the information flowing through an organization to achieve organization-wide interoperation [Dav00]. This variegated information includes finance and accounting, human resources, supply chains, customers, partners, and more. As a result, organizations increasingly depend on reliable and efficient enterprise application systems [SV04].

At the same time, the magnitude of enterprise application systems is undergoing a revolution as large-scale commodity computing is becoming a reality. For example, eBay has 12,000 servers and 15,000 application server instances [Str07], and Google is estimated to have "more than 450,000 servers spread in at least 25 locations around the world" [MH06]. In other words, enterprise application systems are becoming massive-scale and distributed.

Nowadays, security is receiving greater attention from organizations than ever before [All05]. Designing dependable security mechanisms that protect the system and resources becomes an increasingly complex and difficult challenge as the magnitude and complexity of enterprise application systems increase, and as information resources are interconnected by the Internet and/or enterprise networks [Bez00].

Addressing security concerns in a distributed enterprise application system can be roughly divided into two aspects [TS01]. One aspect concerns the communication between components/processes that reside in different machines. The principal mechanism of ensuring secure communication is to build a secure channel. A secure channel is usually achieved by ensuring message confidentiality, mutual authentication, and message integrity, and it protects senders and receivers against interception, modification, and fabrication of messages.

The other aspect concerns authorization or access control,[1] which ensures that a subject gets only those access rights to the resources or objects it is entitled to. Resources can be files in an operating system or data entries in a database, and subjects can be application users or computer programs. The structure of traditional access control mechanisms is based on the conceptual model of *reference monitor* [And72]. As shown in Figure 1.1, a reference monitor is responsible for mediating access by subjects to system resources. The reference monitor is aware of the permissions that a subject has, and decides whether the subject is allowed to carry out a specific operation on specific resources. To provide full protection, the reference monitor

[1]Formally, access control is about verifying access rights, whereas authorization is about granting access rights [TS01]. In this dissertation, we use these terms interchangeably.

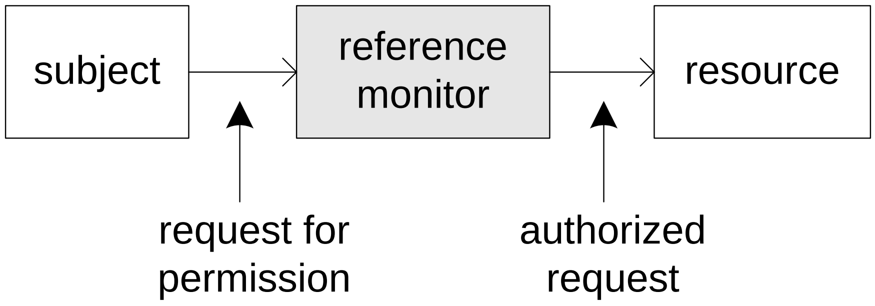


Figure 1.1: General access control model based on the reference monitor [TS01].

is invoked each time a request is issued.

Access control in enterprise application systems is the focus of this dissertation. In particular, we propose various approaches to improving the availability and performance of the access control infrastructure for enterprise application systems.

In the rest of this chapter, Section 1.1 provides an overview of typical authorization architectures used by the existing enterprise application systems, followed by a description of their problems (Section 1.2) that motivate this dissertation. Section 1.3 summarizes our research goals and their importance. Section 1.4 summarizes the contributions of this dissertation. Finally, Section 1.5 outlines the dissertation's structure.

## 1.1 Architecture of authorization solutions

To protect application resources, an obvious solution is for applications to define their own policies and authorization logic to enforce these policies [YB97], e.g., using Java Authentication and Authorization Service (JAAS) [LGK$^+$99]. In this design, the reference monitor is a single component within the application. While this architecture gives developers complete control over authorization during application development, it has a number of disadvantages. First, once an application is in production, it is hard to make changes to authorization functions (such as adding new requirements and modifying the authorization logic), as any change would require the developer to modify the application code [GB99, HGPS99, Ora08a]. Second, in an enterprise with a large number of applications, administrating authorization policies becomes challenging, as it has to be done on application-by-application basis [Bez98]. Besides, it is difficult to maintain policy consistency across multiple applications [HGPS99]. Third, the architecture leads to challenges in the area of audit and compliance [Ora08a]. For example, to comply with government regulations such as the Sarbanes-Oxley Act [Sar02], enterprises are required to report and review users' privileges. This implies that, in this architecture, an auditor needs to check all the applications across the enterprise, which will be a difficult task.

For these reasons, the reference monitor is commonly split into two components, a policy en-

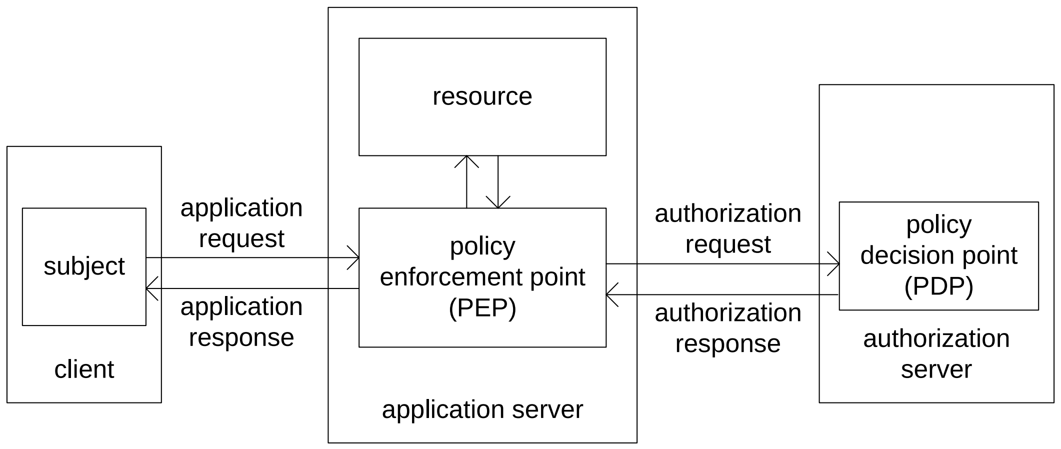


Figure 1.2: Authorization systems based on the request-response model.

forcement point (PEP) and a policy decision point (PDP), as shown in Figure 1.2. Coming from the Internet Engineering Task Force (IETF) and Distributed Management Task Force (DMTF) specifications [YPG99, DBC+00] and used by eXtensible Access Control Markup Language (XACML) [Com05], this design represents a common way of separating different functional components involved in authorization.

The *PEP* is the module responsible for enforcing policy decisions. A PEP logically consists of two parts: a front-end that intercepts the client requests and communicates with the PDP, and a back-end that forwards the requests to the resources. As soon as the PEP receives an application request for a resource access, it formulates an authorization request for an authorization decision and sends it to the PDP. The PDP returns the decision and the PEP then enforces the decision by either accepting and forwarding the request to the resources or denying the request.

The *PDP* is the module responsible for making decisions regarding access permissions to the requested resources. As soon as the PDP receives a request for an access decision from the PEP, it retrieves the policy associated with the protected resource from the policy store (not shown in Figure 1.2.) Based on the information in the request and the access control policy (and possibly other environmental or contextual data), the PDP decides whether to allow or deny access for the requested operation at the remote resource. Finally, an allow or deny message is sent back to the PEP.

The use of PEP and PDP enables the separation of authorization logic from application logic, which reduces the application complexity. Besides, this separation frees developers from dealing with the actual decision-making process, so that developers are able to concentrate on the business logic [Ora08a]. In addition, this architecture is more capable and flexible in dealing with the evolution of authorization requirements. Particularly, the authorization policy can be changed without requiring any modification to the application code.

Furthermore, most modern authorization solutions [CZO+08, Kar03, Ent99, Net00, Sec99, OMG02, DK06, BZP05, SSL+99, Ora08b] advocate the externalization of the PDP to a centralized authorization server. This externalization architecture enables the reuse of authorization logic at a centralized authorization server and consistent policy enforcement across multiple applications. Besides, it provides a central point for administrators to manage policies and for auditors to review users' privileges. Therefore, it helps achieve more efficient policy management, audit and compliance.

Therefore, this dissertation focuses on the architecture that uses the centralized PDP due to its popularity in enterprise application systems. For simplicity, we denote it as the *request-response model*, as the PEP sends each request to the central PDP and enforces the decision in the returned response.

## 1.2 Problem motivation

Although the request-response model enables authorization logic reuse and consistent policy enforcement, it commonly leads to a point-to-point communication architecture, where PEPs obtain decisions from PDPs through synchronous remote procedure calls (RPC). This point-to-point architecture, in turn, results in tight coupling between many PEPs and some particular PDPs. When the scale of enterprise application systems increases, as shown in Figure 1.3, a tightly-coupled point-to-point architecture suffers from two problems: fragility and poor performance.

- **Fragility**. In a distributed authorization system, the authorization server may not be reachable due to a failure (transient, intermittent, or permanent) of the network, of the software located on the critical path (e.g., OS), of the hardware, or even from a misconfiguration of the supporting infrastructure. In that case, all applications depending on that authorization server may not work properly. Hence, Fragility leads to reduced availability. A conventional approach to improving the availability of a distributed infrastructure is failure masking through redundancy of either information or time, or through physical redundancy [Joh96]. However, redundancy and other general purpose fault-tolerance techniques for distributed systems scale poorly, and become technically and economically infeasible when the number of entities in the system reaches thousands [KLW05, Vog04].

- **Poor performance**. In a massive-scale enterprise application system with non-trivial authorization policies, making authorizations is often computationally expensive due to the complexity of the policies involved and the large size of the resource and user populations. Thus, the authorization server often becomes a performance bottleneck [ND97]. Additionally, the communication delay between the application and the authorization server—added to the inherent cost of computing an authorization—can make the authorization overhead prohibitively high.

To address these problems, an alternative architecture is to co-locate PDPs with the corresponding PEPs. In order to achieve central administration of policies, the policy of each PDP

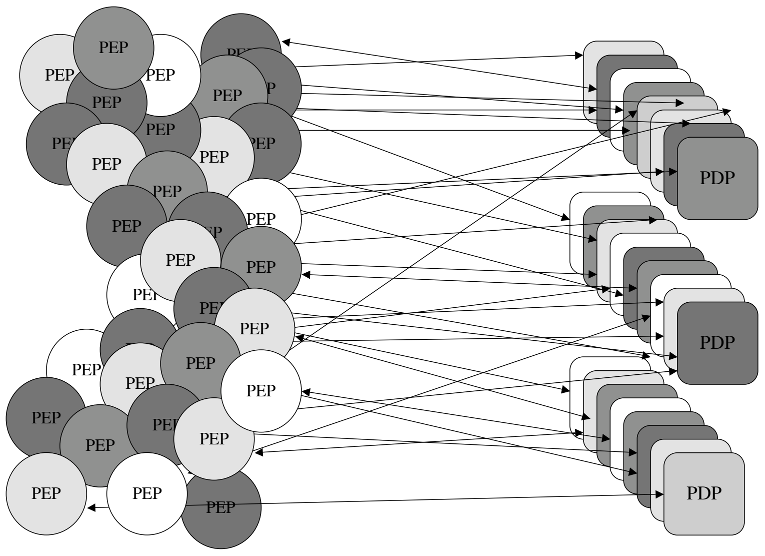


Figure 1.3: The tight coupling between PEPs and PDPs leads to fragile authorization architectures [Bez05]. Any PDP failure will affect all the PEPs that depend on that PDP.

can be delivered from a centralized policy store. Although this architecture reduces the network latency in resolving a request, it posts new challenges to the policy design: the policy should be expressed in such a way that it can be easily distributed to PDPs on-the-fly and then be verified and processed by them efficiently [KS07]. When the policy size is large, the cost of keeping up to date user, attribute, and permission data in multiple PDPs is also prohibitively high. In addition, this architecture is not efficient when the authorization logic is comprehensive. For example, in order to compute a response, the local PDP may still need to contact multiple remote databases/directories to retrieve the attribute information for the subject and object in each request, which might be time-consuming. For these reasons, this architecture is rarely used in enterprise-grade deployments.

## 1.3 Research goals

The goals of this dissertation research are to address the above-mentioned problems. More specifically, we want to investigate alternative solutions to improve the availability and performance of access control systems. By improving these two aspects of access control systems, we address the concerns of users, security administrators, and business managers in enterprises.

From a user's perspective, the availability and performance of the access control system affect the user's experience in interacting with the application. Specifically, when the PEP is unable to get a decision for a request, the PEP may have to deny the request by default. Hence, the user is unable to perform the desired actions that should have been allowed, which leads to reduced user's satisfaction. Lower satisfaction could also be a result of poor performance of the access control system. Hence, it becomes important to tune the performance of enterprise access control systems, as evident by an article [BH05] that discusses various performance tuning techniques for IBM WebSphere Portal access control solutions.

Furthermore, as applications become more complex, the availability and performance of access control systems become even more critical to end users. Compared to authentication, authorization is usually required to perform for each application request. For example, Mercury [Mer05] shows that the total time used for authorization can be seven times longer than the total authentication time. As another example, as in Facebook and Amazon, a web page is usually constructed by multiple "portlets," each of which may display the data produced by some complex web services [Ora09]. As a result, the application request for a single web page may involve resolving multiple authorization requests at the back-end access control system. If any of these authorization process fails or is delayed, the result of the application request would be affected. Therefore, the application availability and performance increasingly depend on the underlying access control systems.

Additionally, improving the availability of the access control system addresses the concerns of security administrators. Bauer et al. [BCR$^+$09] conducted a series of interviews with thirteen administrators who manage access control policy in large universities/organizations. They show that security administrators were very interested in features such as availability and the ability to fail gracefully,[2] because highly available access control systems reduce the burden of security administration.

The availability of the access control system ensures the accessibility of enterprise application systems, which is critical to the proper functioning of an enterprise. When systems are down, the production and critical business processes come to a stop. The costs of being unable to deliver service and product to customers can be extremely high. For example, a large company claimed that the cost of downtime on their point of sale verification systems was in the order of \$5,000,000 per minute [MD99]. Therefore, it is important to make sure that the access control service continues to run uninterrupted in the event of software and hardware failures.

## 1.4 Contributions

The central contribution of this dissertation is the design and evaluation of three approaches to improving the availability and performance of enterprise authorization systems. This work is based on a study of current enterprise authorization systems and their requirements for availability and performance. In detail, this work contributes to research in enterprise authorization architectures as follows:

- *Approximate recycling of authorizations.* We propose mechanisms for authorization recy-

[2]In most of the interviews, administrators stated that availability and the ability to fail gracefully are their major concerns in choosing an access control technology.

cling in role-based access control (RBAC) systems [WCBR08, WCBR09]. The algorithms that support these mechanisms allow *approximate* authorization decisions (a kind of inferred decision) to be made at a secondary decision point (SDP), thereby masking possible failures of the PDP and reducing the authorization decision time. The approximate recycling can be integrated with the existing systems without modifying the PEP and PDP by integrating the SDP. Besides the algorithms for constructing cache and making approximate decisions, we also propose the algorithms for updating cache in case of policy changes and for constructing and verifying evidence. We also prove the correctness of cache recycling and update algorithms: we prove that the SDP will always produce the same decision (allow or deny) as the PDP. Hence, this mechanism is able to meet the audit and legal compliance requirements. Our evaluation results confirm that the number of authorization requests that can be served without consulting the remote PDP increases, compared to the traditional caching mechanism.

- *Cooperative authorization recycling.* We propose a cooperative authorization recycling mechanism where each SDP shares its capability of making decisions with other SDPs to further mask authorization server failures and network delays [WBR06, WRB07, WRB09]. Our design does not require cooperating secondary decision points to trust each other. To maintain cache consistency at multiple secondary decision points, we propose the alternative mechanisms for propagating update messages. Our evaluation results show that, by cooperation between SDPs, a large number of requests can be resolved even when only a small portion of responses are cached at each SDP.

- *Using the publish-subscribe architecture.* We propose to use a publish-subscribe architecture to replace the point-to-point communication channel between PEPs and PDPs. This reduces the dependency of each PEP on specific remote PDPs and also makes the management of authorization infrastructure easier [WRB08]. Our design is independent of the underlying access control policies and publish-subscribe technologies. For different application scenarios, we propose various subscription schemes that can be used to improve system performance. Our evaluation results demonstrate that this approach enables remote PDPs to form a reliable PDP cloud, and thus improve the system availability.

- *An integration of these approaches.* While deploying (cooperative) SDPs and using the publish-subscribe architecture can work independently, we also show that they can work together to support distributed authorization infrastructures. In particular, although the publish-subscribe architecture has a negative impact on the performance, using authorization recycling helps mitigate this negative impact. Meanwhile, using the publish-subscribe architecture helps achieve SDP cooperation as well as *speculative* authorizations (where PDP can pre-compute authorizations based on the request history and push them to the SDP).

## 1.5 Dissertation outline

The rest of this dissertation consists of five chapters. The outline of each chapter is described as follows.

- In Chapter 2, we present a background of enterprise access control systems and related work on improving their availability and performance.

- In Chapter 3, we propose the mechanisms that enable the inference of approximate authorizations for RBAC systems. We define inference rules specific to RBAC authorization semantics and develop the recycling algorithms based on these rules. We provide detailed analytical and experimental results on availability and performance.

- In Chapter 4, we describe a cooperative authorization recycling design which further improves the availability and performance of access control solutions. We address important issues like security and consistency in such a design. We also present the evaluation results using both simulation and a prototype system.

- In Chapter 5, we present an authorization architecture that uses the publish-subscribe model to improve the robustness and manageability of the access control system. In this architecture, PDPs can be viewed to form a reliable PDP cloud to PEPs. We also propose and evaluate different optimization techniques to improve the performance of this architecture.

- In Chapter 6, we conclude the dissertation by discussing the achieved results and outlining future work.

# Chapter 2

# Background and Related Work

Access control is “a security function that protects shared resources against unauthorized accesses” [Til05]. The sub-system that provides access control to the resources of an enterprise application system is called an access control system. This chapter presents the background on building access control systems and related work on improving their availability and performance.

A critical step of developing an access control system is a definition of a set of rules according to which access is to be controlled. Collectively, these rules are called *access control policy.* A formal representation of an access control policy is called an *access control model*, which enables one to formally prove the various properties of the policy. Section 2.1 provides background on the access control policies and models related to this dissertation.

An access control system usually provides two kinds of facility. First, it provides facilities for making decisions whether or not a request should be authorized based on security policies. Second, it provides facilities for managing user privileges, e.g., allocation and revocation of privileges. In this dissertation, we focus on the first kind. Section 2.2 presents the background on how decision and enforcement are usually implemented in enterprise access control systems.

Given the increase in both the number of servers and the scale of geographical areas, the existing enterprise access control systems that employ remote authorization servers are essentially distributed systems themselves. Section 2.3 and Section 2.4 review the solutions that use replication to improve the availability and performance of generic distributed systems. Furthermore, Section 2.5 describes the work that uses caching—a temporary form of replication—for access control systems. In particular, we discuss three caching solutions: decision caching, policy caching, and attribute caching.

Although caching is simple, its impact on access control system performance and availability is limited by a number of factors, e.g., cache size and the frequency of policy changes. In order to achieve a greater improvement, Crampton et al. [CLB06] proposed the secondary and approximate authorization model (SAAM), which extends the decision caching approach. SAAM defines an authorization inference framework where cache can resolve requests that have not been issued before. The approaches we propose in this dissertation are based on SAAM. Therefore, we provide an overview of SAAM in Section 2.6.

Finally, Section 2.7 summarizes this chapter.

## 2.1 Access control policies and models

Access control policies can be grouped into three main classes: discretionary, mandatory, and role-based. A *discretionary* access control policy (DAC) determines access based on identity of subjects and allows a subject with a certain access permission to pass that permission on to any other subject [SV01]. DAC is commonly used in both UNIX and Windows operating systems. The access matrix model [Lam71] provides a framework for describing DAC and it can be implemented in various mechanisms, such as access control lists (ACLs) and capabilities. An access control list is associated with an object and specifies who or what is allowed to access the object and what operations are allowed to be performed on the object, while a capability is associated with a subject and specifies the accessible objects and also a set of legal operations on those objects.

A *mandatory* access control policy (MAC) controls access based on mandated regulations determined by a central authority. The most common form of mandatory policy is the multilevel security policy modeled by a security lattice [San93]. Access to an object from a subject is granted only if the relationship between their security levels is satisfied. MAC has been used in military and other systems where security is the top priority, e.g., Trusted Solaris where MAC makes it possible to prevent anything in the operating system from accessing the disk driver [VV00]. Note that discretionary and mandatory policies are not mutually exclusive but can be applied jointly as in the Bell-LaPadula (BLP) model [BL73a, BL73b]. In this case, for an access to be granted, both a DAC policy and an MAC policy need to be satisfied. We elaborate on the BLP model below in Section 2.1.1.

A *role-based access control* policy (RBAC) controls access based on the roles a subject is assigned and the permissions that are allowed for those roles. Having been introduced more than a decade ago, RBAC [FK92, SCFY96] has been deployed in many organizations for access control enforcement, and eventually matured into the ANSI RBAC standard [ANS04]. In RBAC, instead of directly assigning permissions to users, the users are assigned to roles and the roles are mapped to permissions. A role normally represents the organizational position that is responsible for certain job functions. Users are assigned appropriate roles according to their qualifications. Permissions are a set of many-to-many relations between objects and operations. Roles describe the relationship between users and permissions through user-to-role assignment ($UA$) and permission-to-role assignment ($PA$). We elaborate the RBAC model below in Section 2.1.2.

### 2.1.1 The Bell-LaPadula model

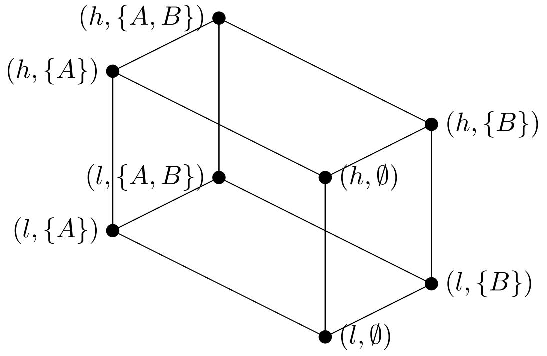
The Bell-LaPadula (BLP) model [BL73a, BL73b] combines mandatory and discretionary access controls. The mandatory component defines the following sets and functions:

- a set of subjects $S$;
- a set of objects $O$;

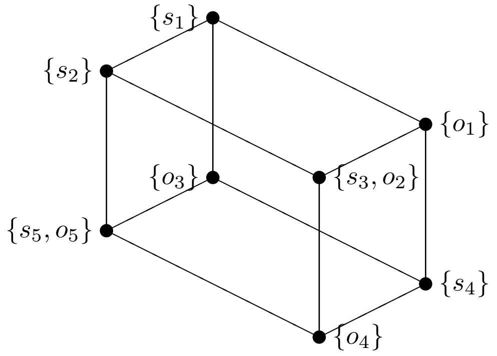
- a lattice of security labels $L$;[3]
- a security function $\lambda : S \cup O \to L$.

The *simple security property* permits a subject $s$ to read an object $o$ if $\lambda(s) \geqslant \lambda(o)$; the **-property* permits a subject $s$ to write to an object $o$ if $\lambda(s) \leqslant \lambda(o)$. The BLP model identifies three generic access rights to which these security properties apply: `read`, which is a read-only action; `append`, which is a write-only action; and `write`, which is a read-and-write action. Hence, the request $(s, o, \texttt{write})$ is only granted if $\lambda(s) = \lambda(o)$. Note that in the full BLP model, a request must be authorized by an access matrix (in addition to satisfying the simple security and *-properties).

For example, Figure 2.1(a) shows a typical security lattice with two security levels (high and low, abbreviated to $h$ and $l$, respectively) and two categories $A$ and $B$. Figure 2.1(b) illustrates the assignment of subjects and objects to security levels in $L$. Based on the simple security property, $s_2$ is only allowed to read $o_2$, $o_4$, and $o_5$. Based on the *-property, $s_5$ is only allowed to write $o_3$ and $o_5$.



(a) The security lattice.



(b) The assignment of subjects and objects to security levels.

Figure 2.1: A BLP example

### 2.1.2 The role-based access control model

There are a number of RBAC models in the literature, including RBAC96 [SCFY96] and the ANSI RBAC standard [ANS04]. All these models assume the existence of a set of users $U$, a set of roles $R$ and a set of permissions $P$. As shown in Figure 2.2, they also assume the existence of a user-role assignment relation $UA \subseteq U \times R$ and a permission-role assignment relation $PA \subseteq P \times R$. A user $u$ is authorized for a permission $p \in P$ if there exists a role $r \in R$ such that $(u, r) \in UA$ and $(p, r) \in PA$.

[3]Strictly speaking, the BLP model requires $L$ to have the form $C \times 2^K$, where $C$ is a linearly ordered set of security classifications and $K$ is a set of needs-to-know categories.

Many models also assume the existence of a role hierarchy $RH$, which is modeled as a partial order on the set of roles. That is $RH \subseteq R \times R$, where $RH$ is reflexive, anti-symmetric and transitive. It is customary to write $r \leqslant r'$ rather than $(r, r') \in RH$. In this case, $u$ is authorized for $p$ if there exist roles $r, r' \in R$ such that $(u, r) \in UA$, $r \geqslant r'$ and $(p, r') \in PA$.

An important innovation in RBAC96 and ANSI RBAC is the concept of *sessions*. A user initiates a session (typically when authenticating to the system) by activating some subset of the roles to which he is assigned. Access requests are evaluated in the context of the session that initiates the request. A request for permission $p$ is granted if the user session contains a role $r$ and there exists a role $r'$ such that $r \geqslant r'$ and $(p, r') \in PA$. The Core Specification of ANSI RBAC, similarly to RBAC96, also defines two mapping functions: $session\_user(s : SESSIONS) \rightarrow USERS$ that maps each session to a single user, and $session\_roles(s : SESSIONS) \rightarrow 2^{ROLES}$ that maps each session to a set of roles.

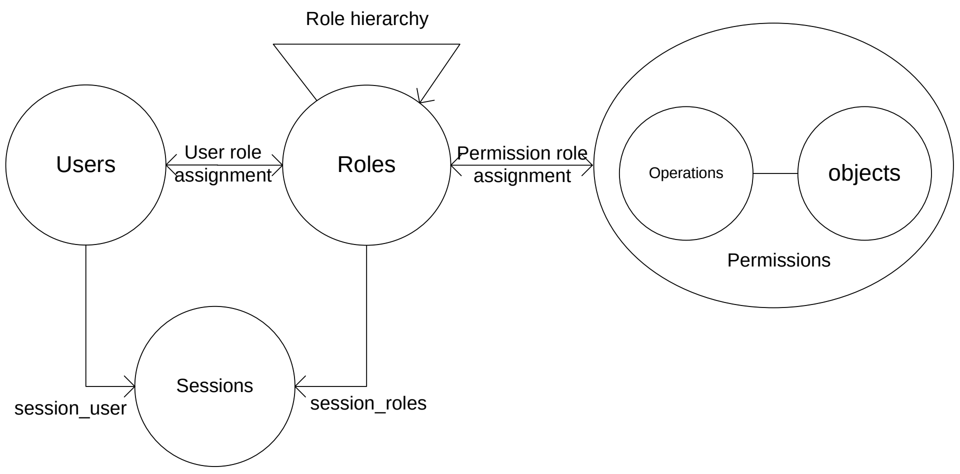


Figure 2.2: RBAC model.

## 2.2 Access control system implementations

Most modern access control systems are based on the request-response model [CZO+08, Kar03, Ent99, Net00, Sec99, OMG02, DK06, BZP05, SSL+99, Ora08b], where the PEP intercepts application requests from subjects and enforces decisions made by the centralized PDP. We distinguish between the *application request*, which is generated by the subject and is dependent on the application logic, and the *authorization request*, which is generated by the PEP and is independent from the application logic. This decoupling, for instance, is performed by the *context handler* in the XACML-compliant PEP [Com05]. The context handler generates an XACML *request context*, which is sent to the PDP for processing. Another example is the authorization request made by the CORBA Security [OMG02] interceptor to the AccessDecision object (ADO), which acts as a PDP. The request to the ADO supplies the subject's attributes, target object ID, implemented interface, and the operation to be invoked, but omits other parameters of the application request.

PEP implementations differ depending on the technology and the security requirements. As shown in Figure 2.3, A PEP can be a security interceptor (such as in CORBA Security [OMG02], ASP.NET [MMDV02], and most web servers), or can be a part of the component container (as in COM+ [Edd99] and EJB [DK06]). A PEP can also be a part of the corresponding application resource, e.g., implemented via static or dynamic "weaving" using aspect oriented software development techniques [KLM+97].

The PDP is usually implemented in the form of an appliction-independent authorization service using third-party components, such as IBM Tivoli Access Control Manager [Kar03] and Oracle entitlement server [Ora08b]. Shared by multiple PEPs, the authorization service provides APIs that allow PEPs to make calls to it. Additionally, the authorization service can be centrally managed and therefore easy to administer.

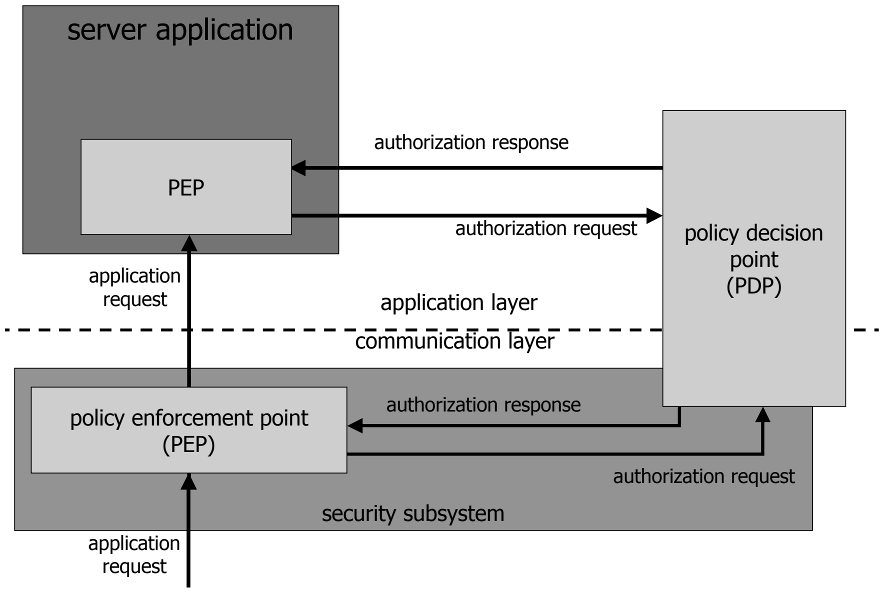


Figure 2.3: Access control system implementations [Bez05]. The PEP intercepts application requests and generates authorization requests. The PEP can be implemented by the underlying middleware or be a part of the application.

Multiple PDPs may be involved in resolving an authorization request. Beznosov et al. [BDB+99] present a resource access decision (RAD) service for CORBA-based distributed systems. The RAD service allows dynamically adding or removing PDPs that represent different sets of polices. In Stowe's scheme [Sto04], a PDP that receives an authorization request from a PEP forwards the request to other collaborating PDPs and combines their responses later. Each PDP maintains a list of other trusted PDPs to which it forwards the request. Mazzuca [Maz04] extends Stowe's scheme. Besides issuing requests to other PDPs, each PDP can also retrieve the policy from other PDPs and make decisions locally.

## 2.3 General fault-tolerance techniques

An important goal in designing a distributed system is to automatically mask partial failures without seriously affecting the overall performance. As defined (jokingly) by Lamport and cited in [Mul93, Pow94], a distributed system is "one that stops you getting your work done when a machine you've never heard of crashes." This "definition" reveals the uncertainty when users interact with the distributed system, due to the defective components and the unreliable communication channel. A well-designed distributed system should tolerate faults and continue to operate to some extent even in their presence.

The key technique for handling failures in a distributed system is to mask them through redundancy. There are three general classes of redundancy solutions [TS01]: information redundancy, time redundancy, and physical redundancy.

With information redundancy, extra bits are usually added to the message before transmission to allow recovery from corrupted bits (e.g., due to packet loss in an unstable communication channel.) An example is erasure coding, a technique for achieving high availability and reliability in storage and communication systems [LCL04]. It transforms a message of $n$ blocks into a message with more than $n$ blocks, such that the original message can be recovered from a subset of those blocks even when some blocks were lost.

With time redundancy, an action (e.g., the remote method invocation or the message transmission) is performed more than once if needed. Time redundancy usually deals with the failures that are transient or intermittent and are unlikely to recur in the next moment, such as transmission errors and transaction aborts. For example, checkpointing and rollback-recovery are well-known techniques that use time redundancy to allow processes to make progress in spite of failures [KT86]. With these techniques, a process takes a checkpoint from time to time by saving its state on a stable storage. When a failure occurs, the process rolls back to its most recent checkpoint, assumes the state saved in that check-point, and resumes execution.

With physical redundancy, extra equipment or processes are added to allow the system to tolerate the loss or malfunction of some components. Physical redundancy can be done in either hardware or software.

In general, the approaches proposed in this dissertation can be viewed as domain-specific approaches that employ physical software redundancy to improve performance and fault tolerance of distributed enterprise authorization systems. In particular, when a PDP fails, other PDPs, the SDP, or other SDPs may be able to mask the failure by providing the requested access control decision. The only efficient way to achieve physical redundancy in distributed systems is by replication. Below we elaborate on the use of replication techniques.

## 2.4 Using replication to improve availability and performance

Replication has been used in almost all related work that cope with failures [TS01]. Other than improving the system availability, replication also helps improve the performance of distributed systems. In this section, we review the general replication techniques. Based on who initiates the replication, there are two general approaches [TS01]: group replication and caching.

### 2.4.1 Group replication

Group replication usually occurs at the server side. In group replication, the server that handles requests is usually replicated to form a group. In the simplest case, a message is sent to the group itself. If one server in the group fails, another server can take over for it, thus improving the system availability.

Group replication can also help improve the performance of a system, especially when the distributed system needs to scale in the number of requests it can handle. By replicating the data in multiple servers and later dividing the workload among replicas, the system performance is improved, too.

Replicas can be created either permanently or dynamically. A permanent replica is created at the start of the service and will not change thereafter. For example, a company usually sets up multiple web servers to answer the incoming requests. Whenever a request comes, a load-balancing server forwards it to one of the web servers, for instance, using a round-robin strategy. In comparison, with dynamical replication, replicas are created dynamically to adapt to the environmental changes. For instance, in Content Distribution Networks (CDNs), the replica of web content is usually created dynamically to adapt to the change in clients' distribution and access patterns [CKK02].

Group replication schemes generally follow two streams, primary-backup and active replication [Mul93]. In the primary-backup approach, replicas are designed to include a primary and several backups and only the primary will interact with clients. After the primary receives a state update, it is responsible for propagating the update to all backups. In this sense, primary-backup schemes are also called passive replication because backups passively receive state changes without having any interaction with clients. In case that the primary fails, one of the backups takes its place. In comparison, all servers are viewed the same in active replication and each client request is processed by all of them. This however requires that the processes hosted by all the servers are deterministic.[4] Although this is hard to achieve in a real-world deployment, active replication is still preferable for those systems that require quick response time even under the presence of faults, or that must handle *byzantine faults* [Joh96].

[4]A deterministic process will produce a predictable response sequence when given an initial state and a request sequence. Deterministic processes in different servers will reach the same final state if they start from the same initial state and receive the same request sequence.

Although group replication helps improve system availability and performance, it has a few limitations. First, it has limited tolerance to network failures and partitioning as it usually happens at the server side. Second, although the system throughput is increased by replication, network latency is still inherent with each request. Third, group replication usually involves the deployment of extra hardware that may result in high cost. Finally, group replication usually scales poorly, and becomes technically and economically infeasible when the number of entities in the system reaches thousands [KLW05, Vog04]. We next describe caching solutions that partially address these problems.

### 2.4.2 Caching

Caching is a temporary form of replication. In essence, cache is client-side local storage facility that temporarily stores a copy of the data requested before. The data is usually expensive to compute at or retrieve from the original source. Therefore, cache may result in quick responses to clients by avoiding additional delay. At the same time, caching is also important in improving the system scalability and availability. As a number of requests can be served by the cache, the number of requests that reach the original source is reduced. Even when the original source fails, the cache is able to serve a portion of requests.

Caching has long been recognized as a powerful performance enhancement technique in many areas of computer design. Most modern computer systems include a hardware cache between the processor and main memory [Smi82], and many operating systems include a software cache between the file system routines and the disk hardware [NWO88]. Caching has also been extensively studied in the context of distributed systems, e.g., in the context of the web [Dav01].

When designing a caching mechanism, it is important to take consistency into consideration. The value of caching is greatly reduced if cached copies are not updated when the data at the original source changes. Cache consistency mechanisms ensure that cached copies of data are eventually updated to reflect changes in the original data. There are several cache consistency mechanisms currently in use: time-to-live fields, client polling, and invalidation protocols (e.g., using Lease [GC89]). A detailed discussion of each mechanism can be found in [TS01].

Although caching helps improve the system performance and availability, this improvement depends on a number of factors, e.g., the frequency of changes of the original data and the size limit of cache storage. As a result, a single cache may suffer from a low cache hit rate. We next describe cooperative caching techniques that are introduced to deal with these problems.

### 2.4.3 Cooperative caching

In cooperative caching, many caches work together to serve requests of a set of clients. When a cache is not able to resolve a data request, it may resort to other caches before forwarding the request to the original data source. Cooperative caching of data has its roots in distributed file and virtual memory systems in high-speed local area network (LAN) environments, where network transfer time is much shorter than disk access time to serve a miss. It has then been used to support a large-scale and wide-area distributed file system, e.g., Shark [AFM05]. In Shark, mutually-distrustful clients can exploit each other's file caches to reduce load on an origin file server. Using a distributed index, Shark clients find nearby copies of data, even when files originate from different servers. Performance results show that the cooperation greatly reduces server load and improves client latency for read-heavy workloads.

Cooperative caching has also been studied extensively in the context of the web, where caches are placed at multiple levels of the network. Through the sharing and coordination of cache state among multiple communicating caches, cooperative caching has been an important technique to reduce web traffic and alleviate network bottlenecks [WC98, XBH06].

A critical step in designing a corporative caching system is the design of the mechanism that allows a cache to be aware of the content/ability of other caches. In general, two mechanisms have been proposed. The first is a centralized mechanism. With this mechanism, a centralized directory server that contains location hints about the documents kept at every cache is set up, as proposed by Provey and Harrison [JH97]. The second is a distributed mechanism. With this mechanism, location hints are replicated locally at each cache, as proposed by Tewari et al. [TDV99]. Similarly in Summary Cache [FCAB00], caches use a Bloom filter to exchange compact messages indicating their content and keep local directories to facilitate searching documents in other caches.

## 2.5 Caching in access control systems

Caching has been an important technique for improving the performance and availability of access control systems. Based on the content that is cached, there are three general caching mechanisms: caching decisions, caching policies, and caching attributes. In this section, we briefly discuss each of these caching mechanisms. For each mechanism, we review the representative systems or protocols that have employed that mechanism.

### 2.5.1 Caching decisions

The state-of-the-practice and state-of-the-art approach is to cache authorization decisions—what we refer to as *authorization recycling* in this dissertation. This technique has been employed in a number of commercial systems, such as Oracle Entitlement Server [Ora08b] and Entrust GetAccess [Ent99], as well as several academic authorization systems, such as CPOL [BZP05] and Flask [SSL$^+$99]. In these systems, there usually exists a cache manager component that caches and manages the authorizations for previous authorization requests and uses them for future decisions.

**Oracle entitlement server (OES)** OES [Ora08b] is a component of Oracle Fusion Middleware that provides a fine-grained authorization engine for enterprise applications. The OES authorization server is able to cache the result of an authorization call, and use that result if future calls are made by the same subject. The authorization cache can automatically invalidate itself if there is a change in policy or user profile. It is important to note that the decisions are cached at the PDP in OES. This is specifically useful when the authorization logic is so complex that the time for making a decision is much longer than the network latency. Besides, the same cache can be shared by multiple PEPs.

**CPOL** Borders et al. [BZP05] propose CPOL, a flexible C++ framework for real-time high-throughput policy evaluation in sensor networks or location-aware computing environments. The goal of CPOL's design is to evaluate policies as efficiently as possible and caching has been used to achieve this goal. In particular, CPOL uses cache to store previous access tokens returned from the authorization engine. Access tokens represent the rights that are given to an entity in the system. The evaluation results show that CPOL was able to process 99.8% of all requests from the cache, reducing the average handling time from $6\mu$s to $0.6\mu$s.

**Distributed proof** Bauer et al. [BGR05] present a distributed algorithm for assembling a proof that a request satisfies an access-control policy expressed in a formal logic. As a different form of decision, a proof assures that an access request to an object by a subject is allowed. They introduce a "lazy" cooperating strategy, in which a party gets the help of others to prove particular subgoals in the larger proof, versus merely retrieving certificates from them—yielding a proof that is assembled in a more distributed fashion. They introduce caching as an optimization technique. Since the lazy scheme distributes work among multiple nodes, each node can cache the subproofs it computes. As future access to the same or a similar resource (even by a different principal) will likely involve nodes that have cached the results of previous accesses, caching leads to significant performance improvement. The evaluation results demonstrate that the number of requests made by the second access is reduced by a factor of two.

### 2.5.2 Caching polices

Another caching mechanism is to cache/replicate policies by the PEP so that each PEP can make authorizations locally. Based on who initiates the caching process, there are two possible ways to distribute policies from PDPs to PEPs.

First, the PDP initiates the policy replication by pushing the policies directly to the PEP. While this approach is simple, it requires the PEP to store all policy information locally, which however may not be feasible when the policy size is large. In addition, under this scheme, any update to the policy (e.g., deleting a user) requires all affected PEPs to update their local copies of the policy. If such updates are frequent or the number of affected PEPs is large, the cost is prohibitively expensive. Finally, the PEP will incur additional processing cost for examining potentially useless policy entries when trying to resolve the request from a specific user.

Second, the PEP initiates the policy replication by pulling the policies from a policy repository as needed and then stores them locally as proposed by [TC09], for example. This scheme exhibits better behavior in terms of storage requirements. However, this scheme also leads to additional delays in evaluating requests and adds additional burden to the PEP. For example, now the PEP needs to perform some additional processing when evaluating an access control request, which may also incur extra communication overhead.

Similar to the approach where PDPs are co-located with the corresponding PEPs and their policies are delivered from a centralized policy store, this approach posts new challenges to the policy design and is inefficient when the authorization logic is comprehensive.

**COPS** The Common Open Policy Service (COPS) Protocol, defined by the IETF's RFC 2748 [DBC$^+$00], is a simple query and response protocol that is used to exchange policy information between a policy server (PDP) and its clients (PEPs). The COPS protocol defines two modes of operation: outsourcing and provisioning. In the outsourcing mode, the PDP receives policy requests from the PEP, and determines whether or not to grant these requests. Therefore, in the outsourcing mode, the policy rules are evaluated by the PDP. In the provisioning mode, the PDP prepares and "pushes" configuration information to the PEP. In this mode, a PEP can make its own decisions based on the locally stored policy information. The provisioning mode has been used to transfer policy between network devices in the network environment where the scale of the policy is usually quite small [SPMF03].

**Tivoli Access Manager** IBM Tivoli Access Manager [Kar03] provides an access control infrastructure for a corporate web environment. Using the application programming interface (API) provided by the Tivoli Access Manager, one can program Tivoli Access Manager applications and third-party applications to query the Tivoli Access Manager authorization service for authorization decisions. The authorization API supports two implementation modes. In remote cache mode, one uses the authorization API to call the centralized Tivoli Access Manager authorization server, which performs authorization decisions for the application. In local cache mode, one uses the authorization API to download a local replica of the authorization policy database. In this mode, the application can perform all authorization decisions locally. "Overhead of policy replication" is mentioned in the technical documentation of the Access Manager [BAR$^+$03], but no evaluation is reported.

### 2.5.3 Caching attributes

The third approach is to cache attributes. Attributes provide a generic way for referring to users and resources in access control systems. For example, the XACML [Com05] uses attributes in order to specify applicable subjects, resources and actions in access control policies. User identity, group memberships, security clearances, and roles can all be expressed using attributes. The evaluation engine denies or allows access to resources based on matching the attributes held

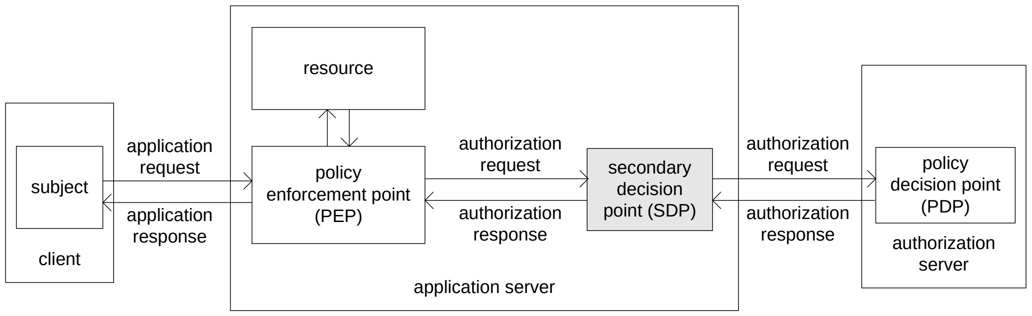


Figure 2.4: SAAM adds SDP to the request-response model.

by the user with those required by the policy.

Attribute certificates, defined by RFC 3281[FH02] and based on the popular X.509 standard, are digitally signed documents that bind a user identity with a set of attributes. In a highly distributed access control system that involves different domains, such as Grid computing [Cha05], in order to enforce local policies for external users, the access control system needs to be capable of fetching users' attribute certificates from external sources. As attribute certificates are usually distributed across multiple hosts, e.g., Internet-based web and remote LDAP servers, a challenge is the length of time it takes to gather all the certificates from remote servers. As a result, caching has been introduced in a number of access control systems [TEM03, Jim01, BDS00, CEE$^+$01, BSF02] to reduce certificate-gathering time.

## 2.6 Secondary and approximate authorization model

In this dissertation, we are particularly interested in the decision caching mechanism as it is simple and widely used. All existing approaches that use decision caching however only employ a simple form of authorization recycling: a cached authorization is reused only if the authorization request in question exactly matches the original request for which the authorization was made. We refer to such reuse as *precise recycling.* In addition, no cooperative caching approach has been proposed before in the context of access control systems.

Our approaches proposed in this dissertation are based on the secondary and approximate authorization model (SAAM), which extends the traditional decision caching mechanism by proposing *approximate recycling.* First proposed by Crampton et al. [CLB06], SAAM is a general framework for making use of cached PDP responses to compute *approximate authorizations* for new authorization requests. SAAM adds a secondary decision point (SDP) to the request-response model (Figure 2.4). Collocated with the PEP, the SDP can resolve authorization requests not only by reusing cached precise authorizations but also by utilizing them to compute approximate authorizations.

SAAM formalizes the definition of authorization request and response. An authorization request is a tuple $(s, o, a, c, i)$, where $s$ is the subject, $o$ is the object, $a$ is the access right, $c$ is the contextual information relevant to the request, and $i$ is the request identifier. Two requests are *equivalent* if they only differ in their identifiers. An authorization response is a tuple $(r, i, E, d)$, where $r$ is the response identifier, $i$ is the corresponding request identifier, $d$ is the decision, and $E$ is the evidence. The evidence can be used in some SAAM implementations to aid the response verification.

In addition, SAAM defines primary, secondary, precise, and approximate authorization responses. A *primary* response is a response made by the PDP, whereas a *secondary* response is produced by an SDP. A response is *precise* if it is a primary response to the request in question or a (secondary) response to an *equivalent* request. Otherwise, if the SDP infers the response based on primary responses to other (non-equivalent) requests, the response is *approximate*.

In general, the SDP infers approximate responses based on cached primary responses and any information that can be deduced from the application request and system environment. The larger the number of cached responses, the more information is available to the SDP. As more and more PDP responses are cached, the SDP will become a better and better PDP simulator.

An SDP is *safe* if any request it allows would also be allowed by the PDP [CLB06]. A safe SDP returns either undecided or deny for any request for which it cannot infer an allow response. A safe SDP can be configured or designed to implement a closed world policy[5] by simply denying any request that it cannot evaluate. More generally, the SDP may return an undecided response; it is then up to the PEP to decide how such a response should be handled. In most cases, the PEP will deny the request, thereby "failing safe"—one of the important principles identified by Saltzer and Schroeder [SS75]. An SDP is *consistent* if any request it denies would also be denied by the PDP.

In general, one would wish to implement a safe and consistent SDP, which returns the same response as the PDP would have for any request that it can evaluate. Clearly, any SDP that only returns precise decisions—by only returning responses for equivalent requests for which decisions have been cached—is safe and consistent. However, such an SDP is rather limited. SAAM seeks to extend the functionality of the SDP so that it can generate approximate responses and remain safe and consistent. However, the limitations of the underlying access control policy, time or space complexity of the inference algorithms, or business requirements could limit an SDP implementation to being either safe or consistent, but not both.

SAAM abstractions are independent of the specifics of the application and access control policy in question. For each class of access control policies, however, specific algorithms for inferring approximate responses—generated according to a particular access control policy—need to be provided. A main contribution of this dissertation is the development of $\text{SAAM}_{\text{RBAC}}$—SAAM authorization recycling algorithms for RBAC. Below we review $\text{SAAM}_{\text{BLP}}$—SAAM authorization recycling algorithms for the BLP access control model, which is proposed in [CLB06].

[5]A closed world policy allows a request if there exists an allow response for it, and denies it otherwise.

### 2.6.1 SAAM$_{\mathbf{BLP}}$

The SAAM$_{\text{BLP}}$ inference algorithms use cached responses to infer information about the relative ordering on security labels associated with subjects and objects. The evaluation results on these algorithms show that the employment of approximate authorizations improves the availability and performance of the access control sub-system.

For example, three requests, $(s_1, o_1, read, c_1, i_1)$, $(s_2, o_1, append, c_2, i_2)$, $(s_2, o_2, read, c_3, i_3)$ are allowed by the PDP and the corresponding responses are $r_1$, $r_2$ and $r_3$. It can be inferred that $\lambda(s_1) > \lambda(o_1) > \lambda(s_2) > \lambda(o_2)$.[6] Therefore, a request $(s_1, o_2, read, c_4, i_4)$ should also be allowed, and the corresponding response is $(r_4, i_4, [r_1, r_2, r_3], allow)$. SAAM$_{\text{BLP}}$ uses a special data structure called *dominance graph* to record the relative ordering on subject and object security labels, and evaluates a request by finding a path between two nodes in this directed acyclic graph. Note that SAAM$_{\text{BLP}}$ inference algorithms only focus on the mandatory part of the BLP model.

## 2.7 Summary

This chapter has presented background information on different access control models, and discussed how caching has been used in distributed systems and access control systems in particular. Our review shows that the state-of-the-practice approach to improving overall system availability and reducing the authorization processing delays is to cache authorizations at each PEP. However, only precise recycling is employed: the cache is only used to serve returning requests.

The SAAM model [CLB06] has been proposed to extend the precise recycling approach by computing approximate authorizations at SDPs. However, SAAM is only an abstract model. For each type of access control policies, specific inference algorithms need to be provided. The next chapter presents a set of SAAM recycling algorithms for RBAC systems.

[6]Note that if such policy information is sensitive, e.g., in a military or government environment, all the PEP-SDP-PDP channels need to be secured so that casual observers cannot make such inferences and only the SDP can.

# Chapter 3

# Recycling RBAC Authorizations

This chapter presents an authorization recycling approach to improving the availability and performance for role-based access control (RBAC) systems. Based on the secondary and approximate authorization model (SAAM), the algorithms that support these mechanisms allow a local secondary decision point (SDP) to not only reuse previously-cached decisions but also infer new decisions based on certain rules, thereby masking possible failures of the central authorization service and reducing the network delays. We evaluate these algorithms experimentally. The results suggest that authorization recycling improves the performance of distributed access control systems.

Compared to approaches that proactively pull or push the policy to each SDP, our approach—based on on-demand caching of authorization responses—offers two advantages. First of all, if the working-set of the policy decision point (PEP) is a significantly smaller subset of the whole policy, it may well be the case that the SDP cache is "warm" enough and hence able to answer a significant proportion of authorization requests more quickly than the policy decision point (PDP) (since the cache size is significantly smaller than the whole RBAC policy and the SDP is collocated with the PEP). Thus, depending on application workload and deployment scenario, our approach offers the possibility of rapid response times without the need for large caches. Second and more importantly, our approach allows for the PEP and PDP remain unchanged. That is, the middleware used for PEP-to-PDP communications can be reconfigured in such a way that the SDP is interposed between the PEP and PDP. As a result, the SDP can act as a PDP's proxy for the PEP, without requiring any modification at the PEP or at the PDP. This means that one can retrofit existing authorization systems with our SDP without changing PEPs or PDPs. For this purpose, dynamic weaving [SPLS$^+$06] or other existing techniques, such as meta-objects [ASA01], for automatically generating custom RPC stubs are readily available. Those RBAC systems that already employ SDPs for precise recycling are even more amenable to being retrofitted with the SAAM$_{\text{RBAC}}$ aproximate recycling logic proposed in this chapter.

In Section 3.1, we first define inference rules specific to RBAC authorization semantics and develop the recycling algorithms from these rules. Second, we study how to modify these algorithms to support the role hierarchy. Third, we propose the policy update algorithms that update the cache to handle different types of policy changes. Fourth, we propose the evidence construction and verification algorithms that can verify the correctness of a secondary response. Finally, we study the deployment strategies of our algorithms to achieve different performance-related goals.

In Section 3.2, we present the evaluation results. To evaluate the properties of SAAM$_{\text{RBAC}}$ algorithms we used an experimental testbed with 100 subjects, 3,000 permissions and 50 roles. The evaluation results demonstrate an 80% increase, compared to precise recycling, in the number of authorization requests that can be served without consulting the access control policies stored remotely at the PDP. These results suggest that deploying SAAM$_{\text{RBAC}}$ improves the availability and scalability of RBAC systems, which in turn improves the performance of the enterprise application systems.

Finally, we summarize this chapter in Section 3.3.

## 3.1 SAAM$_{\text{RBAC}}$ design

SAAM$_{\text{RBAC}}$ applies SAAM concepts to RBAC systems. In a system using SAAM$_{\text{RBAC}}$, the SDP caches authorization requests and the corresponding authorization decisions, and computes new authorization decisions based on the cache when the PDP is unable to make a timely decision. As these decisions are not obtained from the PDP, they are by necessity secondary. In this section we present the algorithms that can be employed by an SDP in the context of RBAC systems. We show that an SDP that implements these algorithms will make safe and consistent secondary decisions.

### 3.1.1 Assumptions

In general, we assume that the PDP is the only component that has access to the entire authorization policy and the SDP is not aware of the policy. In particular, we assume that the SDP does not have direct access to the permission-to-role assignment relation (PA). It is the job of the SDP to try to “reconstruct” PA on the basis of information that can be inferred from primary responses to previous requests. If we relieve this assumption, e.g., the PDP is able to “push” the entire policy to the SDP, then the SDP may compute a precise response using the same authorization logic as the PDP. However, pushing entire policy is rarely done in enterprise-grade deployments due to the limitations of the underlying security protocols, the scale of the authorization policies, the administrative constraints, or the cost of keeping up to date user, attribute, and permission data at multiple SDPs. Providing SDP with no direct access to PA also enables transparently interposing SDP between PEP and PDP without having to modify the protocol between the two. This is an important practical benefit when it comes to retrofitting existing authorization systems with SDP-like components.

We further assume that authorization request sent to the SDP (and the PDP) includes the set of roles that are relevant to this request, this information being supplied by the PEP. This role information arrives at the PEP in a number of different ways [LPL$^+$03]. First, it can be “pushed” from the client’s security subsystem, as in CORBA [OMG02], DCE [GH95], SESAME [Kai98], and GAA API [RN00], where the security attributes are a part of the security context of the application request. Second, it can also be “pulled” by the PEP from external attribute services such as LDAP or the Shibboleth Attribute Authority [Int08].

For most of Section 3.1, we assume that the SDP does not have access to the role hierarchy relation and does not try to reconstruct hierarchical relationships between roles. In Section 3.1.8, we drop this assumption and show how our approach needs to be modified when the SDP is aware of the role hierarchy structure.

### 3.1.2 Preliminaries

We must first consider how to map SAAM notions of subject and request to appropriate RBAC concepts. The notion of session is important in RBAC96 and ANSI RBAC for implementing the principle of least privilege [SS75]: by activating a strict subset of the roles to which she is authorized, a user may limit the privileges that she can exercise while interacting with a computer system. It is a session that is synonymous with a subject in identity-based access control systems, since access decisions are made on the basis of the permissions that are available to the activated roles. Accordingly, SAAM$_{\mathrm{RBAC}}$ models a subject as a set of roles.

As we mentioned in Section 2.6, the Core Specification of ANSI RBAC defines two functions that map sessions to users and roles: $session\_user(s : SESSIONS) \rightarrow USERS$ and $session\_roles(s : SESSIONS) \rightarrow 2^{ROLES}$. SAAM$_{\mathrm{RBAC}}$ abstracts a subject as a set of roles activated in a given session.[7] In the terms of above functions, a subject in SAAM$_{\mathrm{RBAC}}$ is an output of $session\_roles(\dots)$. Therefore, if user $u$ started two sessions $s_1$ and $s_2$, they are treated as two separate—and possibly unrelated—subjects in SAAM$_{\mathrm{RBAC}}$, unless the same roles are activated for both of these sessions. On the other hand, if another user $u'$ started session $s_3$, and the sets of activated roles in $s_2$ and $s_3$ are equal, then SAAM$_{\mathrm{RBAC}}$ algorithms do not distinguish between the corresponding subjects. Furthermore, in SAAM$_{\mathrm{RBAC}}$, we do not take into account the relationship between users and their sessions.

RBAC96 treats permissions as "uninterpreted symbols", because such entities are very likely to be application- and context-specific. However, ANSI RBAC defines permissions to be object-operation pairs. It seems appropriate to regard a SAAM request $(s, o, a, c, i)$ and an RBAC request $(s, p, c, i)$ as equivalent, where $p = (o, a)$.

A response $(r, i, E, d)$ indicates the decision to a request $(s, p, c, i)$. For simplicity, we introduce the following conventions that will be used in the remainder of the chapter: we omit $c$ and $i$ from requests; we omit $r$, $E$ and $i$ from responses; we write $+(s, p)$ to denote a grant decision for request $(s, p)$, and $-(s, p)$ to denote a deny decision. More specifically, $+(s, p)$ means that there exists role $r \in s$ such that $(p, r) \in PA$ and $-(s, p)$ means that there does not exist such an $r$. We also write $X - Y$ to denote the set $\{x \in X : x \notin Y\}$.

[7]Note that static and dynamic separation of duty constraints can be enforced before or during the role activation in a session. The resulted role set is then used for representing the subject in SAAM$_{\mathrm{RBAC}}$. Hence, enforcing these constraints is orthogonal with SAAM$_{\mathrm{RBAC}}$ algorithms.

### 3.1.3 Inference rules

Using the notation from the previous section, we first note the following rules that can be applied to generate approximate responses.

**Rule$^+$** If $+(s, p)$ and $s' \supseteq s$, then request $(s', p)$ should be granted.

**Rule$^-$** If $-(s, p)$ and $s' \subseteq s$, then request $(s', p)$ should be denied.

**Rule$^+$** follows from the fact that if some permission $p$ is granted for the set of roles $s$, then there exists $r \in s$ such that $r$ is authorized for $p$, and $r \in s'$ for any $s' \supseteq s$. **Rule$^-$** follows from the fact that if $p$ is denied for the set of roles $s$, then there does not exist $r \in s$ such that $r$ is authorized for $p$; trivially, no subset of $s$ will be authorized for $p$.

In the following sub-sections, we first present naive algorithms and show that they are suboptimal in terms of success rate and space. Then, we define a canonical form of cache and describe algorithms that work over such cache.

### 3.1.4 Naive algorithms

We construct two relations $Cache^+ \subseteq 2^R \times P$ and $Cache^- \subseteq R \times P$ to generate approximate responses. The basic idea is to use primary deny responses to build $Cache^-$ and primary allow responses to build $Cache^+$.

**Cache construction** Whenever the SDP receives a deny response $-(s, p)$, the pair $(r, p)$ is added to $Cache^-$ for every role $r \in s$ (since we know that no role in $s$ can be authorized for $p$). In contrast, whenever the SDP receives an allow response $+(s, p)$, the pair $(s, p)$ is added to $Cache^+$.

**Request evaluation** Then to evaluate a request $(s, p)$, the SDP first checks whether $s$ contains a role $r$ such that $(r, p) \notin Cache^-$. (If not, no role in $s$ is authorized for $p$ and the SDP denies the request.) Then the SDP checks whether there exists $(s', p) \in Cache^+$ such that $s \supseteq s'$. If so, then the SDP allows the request and otherwise the SDP returns undecided. The algorithm to evaluate request $(s, p)$ is summarized below.

1. Let $s^+ = \{r \in s : (r, p) \notin Cache^-\}$
2. If $s^+ = \emptyset$, then deny (every role in $s$ was not authorized for $p$)
3. Else
   (a) If there exists $(s', p) \in Cache^+$ such that $s \supseteq s'$ then allow
   (b) Else undecided

**Proposition 1.** *An SDP that implements the above request evaluation algorithm is safe and consistent.*

*Proof.* Consider the response produced by the SDP for request $(s,p)$. If the SDP produces a deny response then for all $r \in s$, there exists $(r,p) \in \mathit{Cache}^-$. This means that the PDP must have generated a number of responses of the form $-(s_1,p),\dots,-(s_k,p)$, $k \geqslant 1$, such that for all $r \in s$, $r \in s_i$ for some $i$. Hence, the PDP would also deny $(s,p)$.

If the SDP produces an allow response then there exists $(s',p) \in \mathit{Cache}^+$ such that $s \supseteq s'$. Hence, the PDP would allow request $(s,p)$, since it must have allowed $(s',p)$. □

The naive algorithms, however, may return undecided responses for some requests that would be allowed by the PDP. Suppose that $(\{r_1,r_2,r_3\},p) \in \mathit{Cache}^+$ and $(r_3,p) \in \mathit{Cache}^-$. Now the evaluation of request $(\{r_1,r_2,r_4\},p)$ with the above algorithm returns undecided because $\{r_1,r_2,r_4\} \not\supseteq \{r_1,r_2,r_3\}$. However, $\{r_1,r_2,r_4\} \supset \{r_1,r_2\}$ and hence request $(\{r_1,r_2,r_4\},p)$ can safely be authorized. The optimized algorithms we present in the following sections correct this problem.

### 3.1.5 Cache compression

We have seen that the naive method of constructing the cache may not optimize the "hit rate" – the proportion of requests for which the SDP can provide a definitive answer. We now define the *canonical form* of the cache.

**Definition 1.** *Given a cache, $\mathit{Cache} = (\mathit{Cache}^+, \mathit{Cache}^-)$, we say Cache is in* canonical form *if the following conditions hold:*

*(1) if for all $(s,p) \in \mathit{Cache}^+$, there does not exist $r \in s$ such that $(r,p) \in \mathit{Cache}^-$;*

*(2) for all distinct $(s,p),(s',p) \in \mathit{Cache}^+$, $s \not\subseteq s'$ and $s \not\supseteq s'$.*

The first of the two requirements above ensures that all roles of a subject $s$ that are known *not* to be authorized for a permission are removed from $s$. The second requirement simply ensures that there is no redundancy in the cache: it makes no difference to the allow responses returned by the request evaluation algorithm; in other words, it minimizes the amount of storage required for $\mathit{Cache}^+$.

Cache compression improves the hit rate. In particular, we claim the following statements hold:

1. If $\mathit{Cache}^+$ satisfies property (2) but does not satisfy property (1) above, then the hit rate is not optimal.

2. If $\mathit{Cache}^+$ does not satisfy property (2) above, then the size of the cache is not minimal. That is, there exists a smaller cache that provides the same hit rate.

3. If the cache is in canonical form, then any smaller cache has a lower hit rate.

**Proof of Claim 1**

*Proof.* Suppose that $Cache^+$ does not satisfy property (1). Then there exists $(s, p) \in Cache^+$ such that $r \in s$ and $(r, p) \in Cache^-$. Now request $(s', p)$, where $s' \supseteq s - \{r\}$, is authorized since $r$ is not authorized for $p$. However, $s' \not\supseteq s$ and by assumption $Cache^+$ satisfies property (2) so there does not exist $s'' \subseteq s$ such that $(s'', p) \in Cache^+$. Hence, the SDP cannot resolve request $(s', p)$. Hence, the hit rate is not optimal. □

**Proof of Claim 2**

*Proof.* Suppose that $(s, p), (s', p) \in Cache^+$ and $s' \supseteq s$. Then $(s', p)$ is authorized and any request $(s'', p)$, where $s'' \supseteq s'$, is authorized because $s'' \supset s$. Hence $(s', p)$ may be omitted from $Cache^+$. □

**Proof of Claim 3**

*Proof.* Suppose now that $(s, p) \in Cache^+$ and there does not exist $(s', p) \in Cache^+$ such that $s' \supset s$ or $s' \subset s$. Then request $(s, p)$ is authorized when the SDP uses $Cache^+$ but is not authorized if we remove $(s, p)$ from $Cache^+$ (since, by assumption, there is no $s' \subset s$ such that $(s', p) \in Cache^+$, we cannot find an entry $(s', p) \in Cache^+$ such that $s \supseteq s'$). In other words, omitting $(s, p)$ from $Cache^+$ will decrease the hit rate. □

### 3.1.6 Optimized algorithms

We now present optimized algorithms that produce a canonical form of the cache in order to improve the likelihood of the evaluation algorithm returning an allow response. Henceforth, we write $Cache^- \subseteq 2^R \times P$, making it consistent with the representation of $Cache^+$. Naturally, the meaning of $(s, p) \in Cache^-$ is that all roles in $s$ are known not to be authorized for $p$. The full algorithm (**C**) for constructing compressed cache relations is shown in Figure 3.1(a). To satisfy property (1) of the canonical form definition, in line 3C, which handles negative primary responses, we delete any roles in $s$ from sets of roles that had previously been authorized for $p$ (that is, tuples in $Cache^+$). Analogously, in line 15C, which handles positive primary responses, we delete any roles from $s$ that are known not to be authorized for $p$. To satisfy property (2) of the canonical form definition, line 10C is used to prevent any superset of existing roles in $Cache^+$ from being added and line 14C is used to prune redundant tuples from $Cache^+$.

Figure 3.1(b) shows the decision algorithm (**D**) for generating an approximate response, which follows directly from rules **Rule**$^+$ and **Rule**$^-$ (Section 3.1.3) and the construction of the cache. Since $s$ may include roles that are known not to be authorized for $p$, we remove those roles first and then see whether the remaining roles are authorized for $p$. In other words, given request $(s, p)$, we first find $(s^-, p) \in Cache^-$ (line 3D) and compute those roles in $s$ that are not in $s^-$ (line 10D), namely $s - s^-$. If this set is empty, then we know that all roles in $s$ are in

```
Input: response $q$
1C: $AddResponse(q)$
2C: if $q = -(s,p)$ then
3C:    replace each $(s^+, p) \in Cache^+$ with $(s^+ - s, p)$
4C:    if $(s^-, p) \in Cache^-$ then
5C:       replace it with $(s \cup s^-, p)$
6C:    else
7C:       add $(s, p)$ to $Cache^-$
8C:    end if
9C: else //  we know that $q = +(s,p)$
10C:    if there exists $(s^+, p) \in Cache^+$ such that $s^+ \subseteq s$
        then
11C:       return
12C:    end if
13C:    find $(s^-, p) \in Cache^-$
14C:    delete all $(s^+, p) \in Cache^+$ such that $s - s^- \subseteq s^+$

15C:    add $(s - s^-, p)$ to $Cache^+$
16C: end if
```

(a) The cache construction algorithm

```
Input: request $(s, p)$
1D: $EvaluateRequest(s, p)$
2D: find $(s^-, p) \in Cache^-$
3D: $d \leftarrow s - s^-$
4D: if $d = \emptyset$ then
5D:    return deny
6D: else
7D:    for all $(s^+, p) \in Cache^+$ do
8D:       if $s^+ \subseteq d$ then
9D:          return allow
10D:      end if
11D:   end for
12D:   return undecided
13D: end if
```

(b) The decision algorithm

Figure 3.1: SAAM$_{\mathrm{RBAC}}$ optimized recycling algorithms

$s^-$; that is, $s \subseteq s^-$ and the request should be denied (by **Rule**$^-$). Otherwise, we need to check whether there is a tuple $(s^+, p) \in Cache^+$ such that $s^+ \subseteq (s - s^-)$.

The following example shows how the optimized algorithms work. Suppose $Cache^-$ and $Cache^+$ are empty and the following primary responses are obtained from the PDP:

$$-(\{r_1, r_2\}, p),\ +(\{r_2, r_3, r_4\}, p),\ +(\{r_4, r_5, r_6\}, p),\ -(\{r_4, r_7\}, p).$$

Table 3.1 illustrates how $Cache^-$ and $Cache^+$ develop as these responses are processed by the

| **Response** | *Cache*$^+$ | *Cache*$^-$ |
|---|---|---|
| $-(\{r_1, r_2\}, p)$ | | $(\{r_1, r_2\}, p)$ |
| $+(\{r_2, r_3, r_4\}, p)$ | $(\{r_3, r_4\}, p)$ | $(\{r_1, r_2\}, p)$ |
| $+(\{r_4, r_5, r_6\}, p)$ | $(\{r_3, r_4\}, p)$, $(\{r_4, r_5, r_6\}, p)$ | $(\{r_1, r_2\}, p)$ |
| $-(\{r_4, r_7\}, p)$ | $(\{r_3\}, p)$, $(\{r_5, r_6\}, p)$ | $(\{r_1, r_2, r_4, r_7\}, p)$ |

Table 3.1: Building *Cache*$^+$ and *Cache*$^-$ from primary responses

SDP. Notice how $r_4$ is removed from both tuples in *Cache*$^+$ once the primary deny response $-(\{r_4, r_7\}, p)$ is processed.

Note also that the final contents of *Cache*$^-$ and *Cache*$^+$ are independent of the order in which primary responses are received. If, for example, we reverse the order of the last two responses, we find that $r_4$ is added to *Cache*$^-$ a step earlier and that $r_4$ does not appear with $r_5$ and $r_6$ in a tuple in *Cache*$^+$.

Now suppose we wish to generate secondary responses for the following requests:
(1) $(\{r_3, r_4\}, p)$, (2) $(\{r_1, r_4, r_7\}, p)$, (3) $(\{r_1, r_5\}, p)$.

- The SDP returns an allow response for request (1) because $(\{r_3\}, p) \in$ *Cache*$^+$, even though permission $p$ is not assigned to $r_4$, as indicated by the content of *Cache*$^-$.
- The SDP returns a deny response for request (2) because $(\{r_1, r_2, r_4, r_7\}, p) \in$ *Cache*$^-$.
- The SDP returns an undecided response for request (3).

It is worth noting that although the SDP does not explicitly store primary responses, it will always return the same response as the PDP for any requests whose decisions have been included in the cache relations. More formally, we have the following result.

**Proposition 2.** *Suppose the PDP has produced a response for request $(s, p)$. Then an SDP that implements the construction and decision algorithms in Figure 3.1 will produce the same response as the PDP for request $(s, p)$.*

*Proof.* First note that lines 3C and 15C imply that if $(t^-, p) \in$ *Cache*$^-$ and $(t^+, p) \in$ *Cache*$^+$, then $t^- \cap t^+ = \emptyset$.

Given that the PDP has produced a response, there are two possibilities to consider. If the PDP produced an allow response for $(s, p)$, then $(s^+, p) \in$ *Cache*$^+$ for some $s^+ \subseteq s$, by construction of *Cache*$^+$. If there does not exist $(s^-, p) \in$ *Cache*$^-$ then we are done. Otherwise, consider $d = s - s^-$ (as computed in line 10D). We claim that $d \supseteq s^+$ and hence the SDP will return an allow response. To establish the above claim, consider $r \in s^+$. Then $r \in s$ since $s^+ \subseteq s$. Now $s^+ \cap s^- = \emptyset$ and $r \in s^+$. Hence, $r \notin s^-$ and $r \in s - s^- = d$.

Conversely, if there exists a primary deny response for $(s, p)$, then $(s^-, p) \in Cache^-$ for some $s^- \supseteq s$, by construction of $Cache^-$. Hence $s - s^- = \emptyset$ and the SDP will return a deny response (line 5D). □

**Lemma 1.** *An SDP that implements the construction and decision algorithms is safe and consistent.*

*Proof.* We need to show that if the SDP produces a conclusive (i.e., not undecided) secondary response for request $(s, p)$, then that response is the one that would be produced by the PDP.

Suppose that the SDP produces the response $-(s, p)$. Then there exists $(s^-, p) \in Cache^-$ such that $s \subseteq s^-$ (by line 5D). Moreover, for each $r \in s^-$, $r$ is not authorized for $p$, by construction of $Cache^-$. Hence, the PDP would return $-(s, p)$.

Suppose that the SDP produces the response $+(s, p)$. Then there exists $(s_1^+, p) \in Cache^+$ such that $s \supseteq s_1^+$, which implies the existence of a primary response $+(s_2^+, p)$ with $s_2^+ \supseteq s_1^+$. This implies the existence of $r \in s_2^+$ such that $r$ is authorized for $p$. Moreover, the construction of $Cache^-$ and $Cache^+$ implies that $r \in s_1^+$. Hence $r \in s$, since $s \supseteq s_1^+$ and $r \in s_1^+$, and the PDP would return $+(s, p)$. □

### 3.1.7 Discussion

We now briefly and informally discuss the expected behavior of the SDP algorithms. In Section 3.2, we describe the experimental work we undertook to evaluate the actual behavior.

Suppose $p$ is assigned to roles $r_1, \ldots, r_k$, and that there are $n$ users $u_1, \ldots, u_n$ with $u_i$ assigned to roles $s_i \subseteq R$. Now a user $u_i$ may request $p$ using a subject comprising any subset of $s_i$. In principle, therefore, $Cache^-$ may contain $(s^-, p)$, where $s^- \subseteq R - \{r_1, \ldots, r_k\}$, and $Cache^+$ may contain $(s^+, p)$, where $s^+ \subseteq s_i$ for some $i$.

#### Secondary response rate

Let us suppose that $(s_1^+, p), \ldots, (s_m^+, p) \in Cache^+$ and $(s^-, p) \in Cache^-$. Then the probability that our SDP can produce an approximate response (a "hit") is the probability of it returning either allow or deny. Clearly, the smaller $s_1^+, \ldots, s_m^+$ are, the greater the chance of an allow response, because allow responses require the subject to be a superset of an element in $Cache^+$. Conversely, the larger $s^-$ is, the greater the chance of a deny response are, because allow responses require the subject to be a subset of an element in $Cache^-$.

In short, the probability of a hit increases as the size of $s^-$ increases and the sizes of $s_i^+$ decrease. It can be seen from the construction algorithm that the effect of processing a primary response (whether it is an allow or deny response) is to either increase the size of $s^-$ or decrease the size of $s_i^+$ (or both). In other words, increasing the number of primary responses processed will increase the hit rate.

It is worth noting that it is advantageous to have negative primary responses in the cache, because these affect both $\mathit{Cache}^+$ and $\mathit{Cache}^-$. If there have only been allow primary responses, then $\mathit{Cache}^- = \emptyset$ and hits can only be obtained from secondary allow responses.

For a cache of fixed size, it is advantageous to have $s^-$ large and $s_i^+$ small. It is easy to see that $s^-$ will be large if the number of roles to which $p$ is assigned is small and there have been a large number of requests for $p$ that have been denied (by the PDP). One way to ensure small $s_i^+$ is by assigning each user to a small number of roles.

Alternatively, we are likely to get a hit if there is a significant amount of overlap between the sets of roles assigned to different users. This situation arises when each user is assigned to a significant fraction of the available roles or when some roles are more popular than others so that many users are assigned to those roles. In summary, we would expect the probability of a hit (the "hit rate") to increase when users are assigned to a small number of roles, or to a significant proportion of the roles available, or to a similar set of roles due to the uneven role assignment. We sought to confirm these expectations by experiment, the results of which are reported in Section 3.2.1.

**Performance considerations**

Clearly, the number of tuples in $\mathit{Cache}^-$ is bounded by the number of permissions $|P|$, while the number of tuples in $\mathit{Cache}^+$ is bounded by $|P|\, 2^{|R|}$. For a request $(s, p)$, a secondary deny response can be computed in time proportional to $|s| \log |R|$, as we simply need to determine whether $s$ is a subset of the roles contained in $s^-$. Therefore, the number of primary deny responses is unlikely to have a significant effect on performance. However, the time taken to compute a secondary allow response grows with the number of primary allow responses.

The time taken by the construction algorithm to process a primary response is proportional to the size of $\mathit{Cache}^+$. In the case of a deny response, it is necessary to check each tuple in $\mathit{Cache}^+$ and remove any roles that formed part of the denied request (line 3C). In the case of an allow response, we check to see whether each tuple has been made redundant by the new information (line 14C).

However, we note that the existence of redundant tuples in $\mathit{Cache}^+$ does not compromise the ability of the SDP to compute correct secondary responses, although it may degrade the response time. Therefore, we could periodically purge $\mathit{Cache}^+$ of redundant tuples, rather than delete them as new primary responses are added, thereby improving the processing time for primary allow responses.

In summary, it is easier to incorporate new primary allow responses into the cache rather than deny responses, but it is harder to produce secondary allow responses than deny responses. We investigate these aspects in Section 3.2.1.

### 3.1.8 Using the role hierarchy in SAAM$_{\mathbf{RBAC}}$

When flat RBAC is employed, the binding of a session to a set of roles is trivial: the session is associated with the roles activated by the user. However, in hierarchical RBAC, there are two possibilities, which we call *pre-request* and *post-request* session-to-role binding. We assume that a user initiates a session $s \subseteq R$ by selecting some subset of the roles to which she is assigned. The set of permissions for which the session is authorized is determined by the permissions assigned to the roles in $s$ and to any roles in $R$ that are junior to at least one role in $s$. It is this set of roles, therefore, that should be used to evaluate requests, not simply $s$. More formally, let $\downarrow s$ denote $\{r' \in R : \exists r \in s, r' \leqslant r\}$. Then the evaluation of a request originating from session $s$ requires the computation of the permissions for which the roles in $\downarrow s$ are authorized.

Pre-request binding occurs when the user authenticates. The user $u$ first activates a set of roles $s$ for which she is authorized: that is, for all $r \in s$, there exists $r' \geqslant r$ and $(u, r') \in UA$. The authentication service uses the role hierarchy to compute $\downarrow s$, which is then bound to each process associated with session $s$. This set of roles forms part of the application request that is passed to the PEP. Clearly, pre-request binding means that the computation of all roles associated with a session is performed once, which means that request evaluation should be quicker. Post-request binding occurs when the PDP evaluates an access request. In this case, the PDP has to compute $\downarrow s$ before querying the $PA$ relation.

In the case of pre-request binding, neither the PDP nor the SDP need be aware of the role hierarchy. Hence, we only need to consider what we should do if post-request binding is employed. So far, we have assumed that the SDP is unaware of the role hierarchy. As an optimization, let us now assume that the SDP is aware of the structure of the role hierarchy, and examine how the SAAM$_{\text{RBAC}}$ algorithms need to be modified.

We first note that the SDP could perform post-request binding in exactly the same way as the PDP would. However, we observe that it is not necessary to do this when checking $Cache^-$. To see this, suppose that $(s^-, p) \in Cache^-$ and request $(s, p)$ is received by the SDP. We can check whether $s \subseteq s^-$, as before. Moreover, if $s \subseteq s^-$, then no role belonging to $\downarrow s$ can be authorized for $p$ either (otherwise, some role in $s$, and hence $s^-$, would be authorized for $p$). Hence, it suffices to compute $\downarrow s$ only if the request is not denied. The revised decision algorithm is shown in Figure 3.2. Notice the use of $\downarrow d$, where $d = s - s^-$, in line 8D$'$. Also note that $\downarrow d$ can be computed in polynomial time.[8]

### 3.1.9 Handling policy changes

An enterprise authorization system must support changes to security policies. If the access control policy changes and the SDP is not updated accordingly, the SDP may make incorrect decisions. Policy changes in an RBAC system occur as a result of changes to one of $U$, $P$, $UA$,

[8]More specifically, it can be computed in time proportional to the total number of edges and vertices in the role hierarchy using a simple modification to a standard graph traversal algorithm.

```
Input: request $(s, p)$
1D′: $EvaluateRequest(s, p)$
2D′: find $(s^-, p) \in Cache^-$
3D′: $d \leftarrow s - s^-$
4D′: **if** $d = \emptyset$ **then**
5D′:    **return** deny
6D′: **else**
7D′:    **for all** $(s^+, p) \in Cache^+$ **do**
8D′:       **if** $s^+ \subseteq \downarrow d$ **then**
9D′:          **return** allow
10D′:      **end if**
11D′:   **end for**
12D′:   **return** undecided
13D′: **end if**
```

Figure 3.2: The decision algorithm in a hierarchical setting

$PA$, $R$, or $RH$. Changes to $U$, $P$, or $UA$ do not affect the cache construction and decision-making algorithms. Hence, we only consider changes to $PA$, $R$, and $RH$.

The first type of change involves modification of $PA$. In particular, we considered the following two basic cases.

- A permission $p$ is assigned to a role $r$, that is, $(p, r)$ is added to $PA$.

  If the cache is not updated, the SDP may return incorrect negative decisions for some requests for $p$. Specifically, if $(s, p) \in Cache^-$ and $r \in s$, then any request $(s', p)$ such that $s' \subseteq s$ and $r \in s'$ will be denied despite the fact that $r$ is now authorized for $p$. To avoid this situation, $r$ needs to be removed from $(s, p) \in Cache^-$. Moreover, $(\{r\}, p)$ should be added to $Cache^+$.

- A permission $p$ is revoked from a role $r$, that is, $(p, r)$ is removed from $PA$.

  If the cache is not updated, the SDP may make false positive decisions, because it may compute allow decisions to those requests that are denied by the PDP. To avoid this, we need to replace $(s, p) \in Cache^-$ (if it exists) with $(s \cup \{r\}, p)$, or add $(\{r\}, p)$ to $Cache^-$ otherwise. Moreover, we need to delete every $(s, p) \in Cache^+$ such that $r \in s$. This is because we cannot assume that any of the remaining roles in $s$ are authorized for $p$.

The full algorithm for updating the cache relations to deal with updates to $PA$ is shown in Figure 3.3(a). Comparing it with the cache construction algorithm (Figure 3.1(a)), we note that there are two main differences. First, if $p$ is revoked from $r$, it is not sufficient to remove $r$ from each tuple in the $Cache^+$; instead, all tuples in $Cache^+$ that contain $r$ need to be removed (line 3U$_{\text{PA}}$). Second, if $p$ is assigned to $r$, we add $(\{r\}, p)$ to $Cache^+$ (line 16U$_{\text{PA}}$) and also delete $r$ from the set of roles in $Cache^-$ (line 13U$_{\text{PA}}$), since we know that $r$ is authorized for $p$.

PA changes can be signaled to the SDP by passing "artificial" responses to it. For example, when $(p, r)$ is added to $PA$, the SDP can be sent response $+(\{r\}, p)$. These responses are "artificial" in the sense that they are not generated as a result of a genuine request. In order to distinguish them from normal primary responses, we call them *policy update responses*. When the SDP receives a policy update response, it will invoke the cache update algorithm (shown in Figure 3.3(a)), rather than the cache construction algorithm.

We note that adding $(\{r\}, p)$ to $Cache^+$ may not be necessary but it is a desirable optimization step for two reasons. First, having many tuples of the form $(\{r\}, p)$ in $Cache^+$ will lead to a higher hit rate since more sessions will be a strict superset of an entry in $Cache^+$. Second, it helps remove redundancy from $Cache^+$ as shown in line $15\text{U}_{\text{PA}}$.[9] In an extreme case, while all permissions are being added to $PA$ from scratch and the cache is updated using the cache update algorithm, $Cache^+$ will increasingly resemble $PA$. However, due to the limited size of cache storage and the large size of $PA$, it is unlikely that the SDP will eventually store the whole $PA$ in the cache. By using some cache replacement algorithm, e.g., the least-recently used (LRU) algorithm, SDP is able to keep a small but most-used portion of $PA$ in the cache.

The second type of changes that we considered involves modification of $R$, in particular, when a role $r$ is removed from $R$. Assuming that users cannot start a session that includes deleted role(s), keeping $r$ in the cache will not affect the correctness of the responses that the SDP makes, but will degrade the performance of the SDP. Therefore, it is still desirable to purge the cache of those roles.

The full algorithm for updating the cache relations to deal with updates to $R$ is shown in Figure 3.3(b). Unlike the previous cache update algorithm, this algorithm must consider all tuples containing $r$ in both $Cache^-$ and $Cache^+$. Therefore, this change may result in a large number of tuples being removed from the cache. Like the previous algorithm, all tuples in $Cache^+$ that contain $r$ need to be removed, because we can not assume that any of the remaining roles in the tuple are authorized for $p$.

Third, we consider those changes that involve modification of $RH$. No support for changes in $RH$ is needed if *pre-request* binding is used. When *post-request* binding is used, the SDP needs to be updated with the new $RH$ so that the computation of $\downarrow d$ is correct when a request is evaluated (line $8\text{D}'$ in Figure 3.2).

**Proposition 3.** *A safe and consistent SDP that implements the cache update algorithm is still safe and consistent after a policy change.*

*Proof.* We need to show that, after a policy change, if the SDP produces a secondary response for request $(s, p)$, then that response is the one that would be produced by the PDP after the same policy change.

First, we consider the case when $(p, r)$ is added to $PA$. For any request $(s, p)$ such that $r \notin s$, the policy change has no effect on the decisions returned by the PDP and SDP. We now consider the case when $r \in s$. Clearly, the SDP will return allow for all such requests, since

[9]Note line $15\text{U}_{\text{PA}}$ is used to remove redundancy from $Cache^+$: as for the construction algorithm, this step may be omitted and $Cache^+$ periodically purged of redundant tuples instead.

```
Input: policy update response $q$
1U$_{PA}$: $UpdateCache(q)$
2U$_{PA}$: if $q = -(\{r\}, p)$ then
3U$_{PA}$:     remove those $(s^+, p) \in Cache^+$ for which $r \in s^+$

4U$_{PA}$:     if $(s^-, p) \in Cache^-$ then
5U$_{PA}$:         replace it with $(s^- \cup \{r\}, p)$
6U$_{PA}$:     else
7U$_{PA}$:         add $(\{r\}, p)$ to $Cache^-$
8U$_{PA}$:     end if
9U$_{PA}$: end if
10U$_{PA}$: if $q = +(\{r\}, p)$ then
11U$_{PA}$:     find $(s^-, p) \in Cache^-$
12U$_{PA}$:     if $r \in s^-$ then
13U$_{PA}$:         replace it with $(s^- - \{r\}, p)$
14U$_{PA}$:     end if
15U$_{PA}$:     delete all $(s^+, p) \in Cache^+$ such that $r \in s^+$
16U$_{PA}$:     add $(\{r\}, p)$ to $Cache^+$
17U$_{PA}$: end if
```

(a) The cache update algorithm when *PA* is changed

```
Input: the role $r$ which is to be removed
1U$_R$: $UpdateCache(r)$
2U$_R$: for all $p$ in $Cache^-$ do
3U$_R$:     find $(s^-, p) \in Cache^-$
4U$_R$:     if $r \in s^-$ then
5U$_R$:         replace it with $(s^- - \{r\}, p)$
6U$_R$:     end if
7U$_R$: end for
8U$_R$: for all $p$ in $Cache^+$ do
9U$_R$:     delete all $(s^+, p) \in Cache^+$ such that $r \in s^+$
10U$_R$: end for
```

(b) The cache update algorithm when a role is removed from $R$

Figure 3.3: Cache update algorithms

$(\{r\}, p)$ was added to $Cache^+$ as a result of the cache update. Equally, the PDP will return allow for such requests since $(p, r) \in PA$ as a result of the policy change.

Second, we consider the case when $(p, r)$ is removed from $PA$. As above, we need only consider the case when $r \in s$. If the SDP returns an allow decision then there exists $(s^+, p) \in Cache^+$ such that $s^+ \subseteq s$ and $r \notin s^+$. Hence, there exists some role $r' \in s^+$ that is authorized for $p$. Since the only change to $PA$ was to remove the authorization for role $r$, we may infer that the PDP would also allow request $(s, p)$, since $r' \in s$. If the SDP returns a deny decision then $s \subseteq s^- \cup \{r\}$. In other words, no role in $s^- \cup \{r\}$ is authorized for $p$ and now that $(p, r)$ has been removed from $PA$, the PDP will also return deny.

Third, we consider the case when role $r$ is removed from $R$. No new session will contain role $r$ and *Cache*$^-$ and *Cache*$^+$ are able to decide fewer requests.

- Suppose first that $(s, p)$ was allowed by the SDP and the PDP before the removal of $r$.

  Now if $(s - \{r\}, p)$ is denied by the SDP after the removal of $r$, then $(s^-, p) \in Cache^-$ and $r' \in s^-$ for all $r' \in s - \{r\}$, which in turn implies that no role in $s - \{r\}$ is authorized for $p$ and the request would also be denied by the PDP.

  If, however, $(s - \{r\}, p)$ is allowed by the SDP after the removal of $r$, then there exists $(s^+, p) \in Cache^+$ such that $s - \{r\} \supseteq s^+$ and hence it would be allowed by the PDP.

  (There are also requests that may be allowed by the SDP before the removal of $r$, but cannot be decided after. However, these requests are irrelevant to the definitions of safety and consistency.)

- Suppose now that $(s, p)$ was denied by the SDP and the PDP before the removal of $r$.

  Then $(s - \{r\}, p)$ will be denied after $r$'s removal. Hence, any request that is denied by the SDP after $r$'s removal will be denied by the PDP.

□

An important question to answer is how to propagate update messages to SDPs. In Chapter 4, we provide a detailed discussion on the alternatives for propagating update messages and a solution for implementing well defined semantics for policy updates. For the sake of completeness, below we briefly describe our solution in the context of SAAM$_{\text{RBAC}}$.

Based on the fact that not all policy changes are at the same level of criticality, we divide policy changes into three types: critical, time-sensitive, and time-insensitive changes. By discriminating policy changes according to these types, system administrators can choose to achieve different consistency levels. In addition, system designers are able to provide different consistency techniques to achieve efficiency for each type. Our design allows a SAAM$_{\text{RBAC}}$ deployment to support any combination of the three types.

***Critical changes*** of authorization policies are those changes that need to be propagated urgently throughout the enterprise applications, requiring immediate updates on all SDPs. We assume that the policy is persistently located in a separate policy store. To avoid modifying existing authorization servers and maintain backward compatibility, we add a policy change manager (PCM) that monitors the policy, detects policy changes, and informs the SDPs about the changes. To support critical changes, SDPs would have to implement algorithms in Figure 3.3 and PCM would have to "push" changes to SDPs, which requires adding SDP-PCM communication channel. Support for two other types of policy changes is less intrusive, however.

***Time-sensitive changes*** in authorization policies are less urgent than critical ones but still need to be propagated within a known period of time. We suggest using time-to-live (TTL) approach for processing time-sensitive changes. Every primary response is assigned a TTL that determines how long the response should remain valid in the cache, e.g., one day, one hour, or one minute. The assignment can be performed by the SDP to achieve backward compatibility. Every SDP periodically purges from its cache those responses whose TTL elapses.

When the administrator makes a ***time-insensitive change***, the system guarantees that all SDPs will *eventually* become consistent with the change. A simple approach for supporting time-insensitive change is for system administrators to periodically flush SDPs caches.

### 3.1.10 Evidence construction and verification

Recall that a SAAM authorization response is defined as $(r, i, E, d)$, where $r$ is the response identifier, $i$ is the corresponding request identifier, $d$ is the response decision, and $E$ is the evidence containing information to verify the correctness of $d$. The previous sections omit $E$ for simplicity. In this section we discuss how to construct $E$ and use it to verify a response.

If a response is *primary*, then $E$ is empty because the response is generated by the PDP. The PDP computes access control decisions using access control policies directly. Therefore, all its decisions are assumed to be correct. To protect the authenticity and integrity of a primary response during its transit between the PDP and the PEP, the PDP could cryptographically sign the response. Then, a PEP may verify the primary response's authenticity and integrity by checking its signature.

In this section, we present algorithms for creating evidence and for using it to verify the correctness of *secondary responses*, which are generated by the SDP. To be consistent with the previous discussion, we write $+(s, p, E)$ to denote a secondary allow response for request $(s, p)$, and $-(s, p, E)$ to denote a secondary deny response. When the context is clear, we simply use $\pm(s, p, E)$ to denote a general secondary response and $\pm(s, p)$ to denote a general primary response.

Before we present the algorithms, let us first clarify what we mean by a secondary response being "correct". We say that a secondary response $\pm(s, p, E)$ is "correct", if the same decision would be made by the PDP. Therefore, to verify the correctness of a secondary response $\pm(s, p, E)$, the evidence part $E$ must contain enough information so that an entity external to the SDP (e.g., the PEP or a third-party auditor) can conclude that the PDP would make the same decision given the request and the policy at hand.

The first type of secondary response we consider is a *precise* response. Given a request $(s, p)$, the SDP generates a *precise* response $\pm(s, p, E)$, if the SDP finds in its cache a *primary* response $\pm(s, p)$ and the corresponding request $(s, p)$. In this case, $E = [\pm(s, p)]$. The correctness of the response can then be established through verifying that $\pm(s, p)$ has been produced by the PDP.

The other type of secondary response is *approximate* response. Since approximate responses are inferred from primary responses in the cache, the set of these primary responses are constructed by the SDP as the evidence. Each primary response $\pm(s, p)$ tells whether or not the roles of subject $s$ are assigned to $p$. By combining this information from all the primary responses in the evidence, an external party may determine whether or not the roles of the subject $s'$ in request $(s', p)$ are assigned to the same permission $p$, thus verifying the decision of the approximate response $\pm(s', p, E)$. The rest of this section describes the algorithms for evidence construction and verification for approximate responses.

### Evidence construction

Before it can construct an evidence, the SDP needs to associate the roles in both $Cache^-$ and $Cache^+$ to those primary responses that lead to the existence of those roles. Therefore, we introduce two relations: $Evidence^+$ and $Evidence^-$. For each $(s, p) \in Cache^-$, we use $Evidence^-$ to associate each individual role of $s$ to the corresponding primary deny responses. For each $(s, p) \in Cache^+$, we use $Evidence^+$ to associate $s$ to the corresponding primary allow responses. More specifically, $(r, -(s, p)) \in Evidence^-$ represents that there exists a primary deny response $-(s, p)$ and $r \in s$. On the other hand, $(s', +(s, p)) \in Evidence^+$ represents that there exists a primary allow response $+(s, p)$ and $s' \subseteq s$. Note that $r$ or $s'$ may be associated with multiple responses because they may appear in the roles of multiple subjects.

$Evidence^+$ and $Evidence^-$ are constructed when the SDP constructs $Cache^+$ and $Cache^-$. Figure 3.4(a) shows the revised cache construction algorithm ($\mathbf{C}''$) with the construction of both $Evidence^+$ and $Evidence^-$. For a deny response $-(s, p)$, $s$ is split into individual roles and each role leads to a new entry in $Evidence^-$ (lines 10C″–12C″). For an allow response $+(s, p)$, only one entry is added to $Evidence^+$ after those roles that are not assigned to $p$ have been removed from $s$ (line 21C″). Note that any change to the entry in $Cache^+$ (lines 3C″ and 18C″) is accompanied by a change to $Evidence^+$ (lines 4C″ and 19C″). Therefore, for each $(s, p)$ in $Cache^+$, there is at least one $(s, q)$ in $Evidence^+$, where $q$ is a primary allow response.

The evidence for an approximate response is constructed when the SDP infers an approximate decision. Figure 3.4(b) shows the revised decision algorithm ($\mathbf{D}''$) for generating $E$. The generated $E$ contains a set of primary responses. If the SDP makes a deny decision, $E$ only contains a set of primary deny responses. If the SDP makes an allow decision, $E$ contains a set of primary deny responses as well as a set of primary allow responses.

Note that computing evidence unavoidably adds additional overhead to both cache construction and decision processes. Therefore, it should be only enabled when necessary. For example, when a distributed version of SAAM is deployed and the cooperation between SDPs is enabled [WRB09], it is critical for a remote SDP to verify the secondary response sent by other SDPs.

### Evidence verification

Evidence verification follows directly from rules $\mathbf{Rule}^+$ and $\mathbf{Rule}^-$. For an approximate deny response $-(s, p, E)$, a third party verifies that none of the roles of $s$ are assigned to $p$. For an approximate allow response $+(s, p, E)$, the third party verifies that at least one role of the $s$ is assigned to $p$. The primary responses in the evidence $E$ provide such information for verification.

```
Input: response $q$
1C'': $AddResponse(q)$
2C'': if $q = -(s,p)$ then
3C'':   replace each $(s^+,p) \in Cache^+$ with $(s^+ - s, p)$
4C'':   replace each $(s^+, +(s',p)) \in Evidence^+$ with $(s^+ - s, +(s',p))$
5C'':   if $(s^-,p) \in Cache^-$ then
6C'':     replace it with $(s \cup s^-, p)$
7C'':   else
8C'':     add $(s,p)$ to $Cache^-$
9C'':   end if
10C'':   for all $r \in s$ do
11C'':     add $(r, -(s,p))$ to $Evidence^-$
12C'':   end for
13C'': else // we know that $q = +(s,p)$
14C'':   if there exists $(s^+,p) \in Cache^+$ such that $s^+ \subseteq s$ then
15C'':     return
16C'':   end if
17C'':   find $(s^-,p) \in Cache^-$
18C'':   delete all $(s^+,p) \in Cache^+$ such that $s - s^- \subseteq s^+$
19C'':   delete all $(s^+, +(s',p)) \in Evidence^+$ such that $s - s^- \subseteq s^+$
20C'':   add $(s - s^-, p)$ to $Cache^+$
21C'':   add $(s - s^-, +(s,p))$ to $Evidence^+$
22C'': end if
```

(a) The revised cache construction algorithm

```
Input: request $(s,p)$
1D'': $EvaluateRequest(s,p)$
2D'': $E = \emptyset$
3D'': find $(s^-,p) \in Cache^-$
4D'': $i \leftarrow s \cap s^-$
5D'': for all $r \in i$ do
6D'':   if $(r, -(s',p)) \in Evidence^-$ then
7D'':     add $-(s',p)$ to $E$
8D'':   end if
9D'': end for
10D'': $d \leftarrow s - s^-$
11D'': if $d = \emptyset$ then
12D'':   return -(s,p,E)
13D'': else
14D'':   for all $(s^+,p) \in Cache^+$ do
15D'':     if $s^+ \subseteq d$ then
16D'':       for all $(s^+, +(s',p)) \in Evidence^+$ do
17D'':         add $+(s',p)$ to $E$
18D'':       end for
19D'':       return +(s,p,E)
20D'':     end if
21D'':   end for
22D'':   return undecided
23D'': end if
```

(b) The revised decision algorithm

Figure 3.4: SAAM$_{\mathrm{RBAC}}$ revised algorithms for evidence construction

```
Input: response $\pm(s, p, E)$
1V: $VerifyEvidence(\pm(s, p, E))$
2V: $d = \emptyset$
3V: for all $-(s', p)$ in $E$ do
4V:     $d = d \cup s'$
5V: end for
6V: if $s \subseteq d$ and $\pm(s, p, E) = -(s, p, E)$ then
7V:     return true
8V: end if
9V: $s = s - d$
10V: for all $+(s', p)$ in $E$ do
11V:     $s' = s' - d$
12V:     if $s' \subseteq s$ and $\pm(s, p, E) = +(s, p, E)$ then
13V:         return true
14V:     end if
15V: end for
16V: return false
```

Figure 3.5: The evidence verification algorithm

Figure 3.5 presents the algorithm for verifying the evidence $E$ of an approximate response $\pm(s, p, E)$. If it is a deny response $-(s, p, E)$, then $E$ should only contain deny responses. Consider $E = \{-(s_1, p), \ldots, -(s_{k-1}, p)\}$. The algorithm merges all the roles of $s_i$, i.e., $s' = s_1 \cup \ldots s_{k-1}$ (line 4V). If $s'$ is a superset of $s$, the correctness of $-(s, p, E)$ is verified because all the roles of $s$ are not assigned to $p$.

If the response is an allow response $+(s, p, E)$, then $E$ should contain both deny responses and allow responses. Consider $E = \{-(s_1, p), \ldots, -(s_{k-1}, p), +(s_k, p), \ldots, +(s_n, p)\}$. The first step is still to merge all the roles of the subjects in deny responses, i.e., $d = s_1 \cup \ldots s_{k-1}$. The second step is to remove $d$ from $s$ (line 11V). If the resulted $s$ is a superset of any $s_{\{k,\ldots,n\}} - s'$, then the correctness of $+(s, p, E)$ is verified because some roles of $s$ are assigned to $p$.

In all other cases, the response fails the verification.

We note that each primary response in the evidence is signed by the PDP and the evidence contains both the response and its signature. The first step of any verification should be verifying the signature of each response to make sure that the response was generated by the PDP and has not been changed. For the simplification purpose, this additional step is not shown in Figure 3.5.

## Example

Let us use the example in Section 3.1.6, where the following primary responses are obtained from the PDP:

$$-(\{r_1, r_2\}, p), \ +(\{r_2, r_3, r_4\}, p), \ +(\{r_4, r_5, r_6\}, p), \ -(\{r_4, r_7\}, p).$$

| **Response** | *Evidence*$^+$ | *Evidence*$^-$ |
|---|---|---|
| $q_1 = -(\{r_1, r_2\}, p)$ | | $(r_1, q_1)$, $(r_2, q_1)$ |
| $q_2 = +(\{r_2, r_3, r_4\}, p)$ | $(\{r_3, r_4\}, q_2)$ | $(r_1, q_1)$, $(r_2, q_1)$ |
| $q_3 = +(\{r_4, r_5, r_6\}, p)$ | $(\{r_3, r_4\}, q_2)$, $(\{r_4, r_5, r_6\}, q_3)$ | $(r_1, q_1)$, $(r_2, q_1)$ |
| $q_4 = -(\{r_4, r_7\}, p)$ | $(\{r_3\}, q_2)$, $(\{r_5, r_6\}, q_3)$ | $(r_1, q_1)$, $(r_2, q_1)$, $(r_4, q_4)$, $(r_7, q_4)$ |

Table 3.2: Building *Evidence*$^+$ and *Evidence*$^-$ from primary responses

Table 3.2 illustrates the construction of *Evidence*$^-$ and *Evidence*$^+$ as these responses are processed by the SDP. In the resulting *Evidence*$^-$, each individual role is mapped to a deny response. On the other hand, in the resulting *Evidence*$^+$, each role set is mapped to an allow response, and each allow role set has a corresponding entry in *Cache*$^+$ (Table 3.1).

For the two decided requests:
(1) $(\{r_3, r_4\}, p)$, (2) $(\{r_1, r_4, r_7\}, p)$, the SDP returns the evidence $\{q_4, q_2\}$ and $\{q_4, q_1\}$ respectively. Using the verification algorithm, both evidences can be verified correctly.

### 3.1.11 Implementation considerations

To facilitate the integration with existing access control systems, the SDP should provide the same policy evaluation interface to its PEP as the PDP, thus enabling SAAM incremental deployment without any change to existing PEP or PDP components. Similarly, in systems that already employ authorization caching but do not use SAAM, the SDP can offer the same interface and protocol as the existing cache component.

SAAM may be deployed for a variety of performance-related reasons, depending on the specific application, geographic distribution, and network characteristics. These reasons will typically include one or more of the following: to reduce the overall load on the PDP; to minimize the delay in responding to the client; and to minimize the network traffic generated by the authorization service. We now discuss two alternative ways for managing the interactions between the PEP, the SDP, and the PDP. These strategies lead to different performance characteristics. Hence, different performance-related priorities can be realized by choosing different deployment strategies.

The first strategy is *concurrent authorization* by the SDP and the PDP. When the SDP receives an authorization request from the PEP, it forwards the request to the PDP. While waiting for a decision from the PDP, it also computes a decision locally. The SDP then returns to the PEP the first conclusive decision it receives or computes. The use of concurrent authorization reduces system response time but increases load on the PDP. Alternatively, we may use *sequential authorization.* The SDP only forwards the request to the PDP if it cannot decide the request. The use of sequential authorization reduces network traffic and load on the PDP, at the cost of increased response time as observed by the PEP. The evaluation of these two strategies is presented in Section 3.2.2.

## 3.2 Experimental evaluation

While the previous section described $\mathrm{SAAM_{RBAC}}$ algorithms and estimates their complexity, this section presents an experimental evaluation of those algorithms. We used both simulation and a prototype implementation for evaluation. The simulation enabled us to study the algorithms by hiding the complexity of underlying communication, while the prototype enabled us to study the system performance in a more dynamic and realistic environment.

### 3.2.1 Simulation-based evaluation

In the simulation-based evaluation, we studied three performance aspects of our algorithms: the achieved hit rate, the impact of policy changes on the hit rate, and the computational cost.

First, we studied the *hit rate*, which we define to be the ratio between the number of requests resolved by the SDP (regardless of the specific allow/deny decision) and the total number of requests received. A high hit rate has the effect of masking transient PDP failures, thus improving the overall authorization system's availability. It also reduces the load on the PDP, thus improving the system's scalability, and the authorization system response time.

Our informal analysis in Section 3.1.7 suggested that the hit rate is influenced by the following factors: (1) the *cache warmness* (the ratio between the number of authorization responses cached at the SDP and the number of possible requests); (2) the percentage of deny responses in the cache at a fixed cache warmness; (3) the characteristics of the RBAC policy, including the ratios between the numbers of users, permissions, and roles in the system; and (4) the popularity distribution of roles. Section 3.2.1 presents results of our experiments investigating the impact of these factors on the hit rate.

The second performance aspect we studied was the impact of *policy changes* on the hit rate. We wanted to understand how the algorithms for handling policy changes (Figure 3.1) affected the hit rate. Section 3.2.1 presents the experiment results.

The third performance aspect we investigated was the *computational cost* of the SDP algorithms. We measured two types of computational cost: the *inference time*—the time that the SDP takes to infer an approximate response (allow or deny) using its cache; and the *update time*—the time that the SDP takes to incorporate a new primary response in its cache. In particular, the lower the inference time, the more efficient the SDP is in accelerating the access control system. As cache warmness appears to be the main factor influencing performance, Section 3.2.1 presents the influence of cache warmness on the inference and update time.

### Experimental setup

To conduct the experiments, we implemented SAAM$_{\mathrm{RBAC}}$ recycling algorithms and integrated the implementation with the SAAM evaluation engine used in [CLB06]. Each run of the evaluation involved two stages.

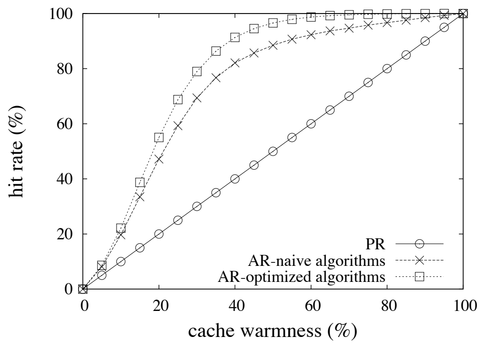
The first stage was to create the data input files that were required for the simulation. The engine first created an RBAC policy and assigned roles to both users ($UA$) and permissions ($PA$). Second, the engine created the warming set and testing set, which were simply lists of requests. Each request was made up of a subject and a permission. The *warming set* was a pseudo-random permutation of all possible requests, while the *testing set* was a random sampling of requests.

In the second stage, the simulation engine started operating by alternating between *warming* and *testing* modes. In the warming mode, the engine used a subset of the requests from the warming set, evaluated them using a simulated PDP, and sent the responses to the SAAM$_{\mathrm{RBAC}}$ SDP to build up the cache. During this phase, the evaluation engine also recorded the time required to add primary responses to the cache. Once the desired cache warmness was achieved, the engine calculated the average update time and then switched into the testing mode during which the SDP cache remained constant. We used this mode to evaluate the hit rate and the inference time at controlled, fixed levels of cache warmness. The engine submitted requests from the testing set, recorded the inference time. Once all the requests in the testing set had been submitted, the engine calculated the hit rate as the ratio of the testing requests resolved by the SDP to all test requests and the average inference time, and then switched back to the warming mode. These two modes were then repeated for different levels of cache warmness, from 0% to 100% in increments of 5%.

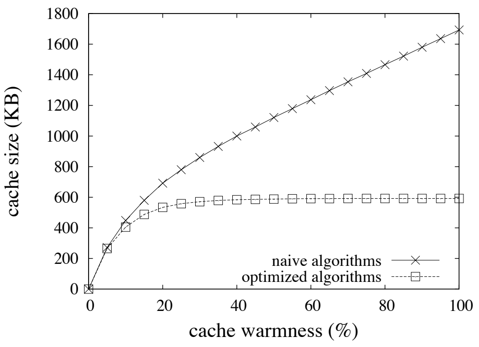
For all experiments we used a Linux machine with two Intel Xeon 2.33 GHz processors and 4 GB of memory. The evaluation framework ran on Sun's 1.5.0 Java Runtime Environment (JRE). Each experiment was run ten times and the average results are reported.

We assumed for simplicity that a user always activated all her roles. This assumption allowed us to describe the entire request space more easily because we could assume then that the request space was defined by the set of users and the set of permissions rather than the set of permissions and the set of all subsets of any set of roles for which some user was authorized. However, we do not believe that this assumption had a detrimental impact on our results. Indeed, our choice was likely to mean that the hit rate was lower than might be expected if users were to use subsets of their authorized roles. The reason for this is due to the fact that smaller role sets in subjects mean that (1) the likelihood of a negative response is increased, which increases the hit rate, and (2) the size of role sets in $Cache^{+}$ may be reduced, which means that the chance of a hit is also increased.

The reference RBAC policy used in our experiments contained 100 users, 3,000 permissions,



(a) Hit rate as a function of cache warmness



(b) Cache size as a function of cache warmness

Figure 3.6: Comparing optimized algorithm and naive algorithm of approximate recycling (AR), with precise recycling (PR), for an RBAC system with 100 users, 3,000 permissions, and 50 roles.

and 50 roles. Thus the overall size of the request space and the warming set was 300,000. The testing set contained 20,000 unique requests which were randomly selected from the request space. For simplicity, we only considered the flat RBAC model. Each assigned role was randomly selected from $R$. The probability of a given user being assigned to a given role was 0.1. Hence the number of roles assigned to a user was binomially distributed with mean 5 and variance 4.5, and the number of users to which a role was assigned was binomially distributed with mean 10 and variance 9. Similarly, the probability of a given permission being assigned to a given role was 0.04.

While the scale of the system we studied was limited by the computational resources available we believe that the values of these parameters are not important in themselves. We were interested in configuring a reasonably large system that would manifest a behavior asymptotically similar to possible real-world deployments. Additionally, we studied the impact of varying the number of users, roles per user, roles, and roles per permission as well as the popularity distribution of roles on system's performance. We note that, while the overall number of permissions in the system may influence the response time as a large number of permissions leads to less efficient memory use by the SDP, it will not influence the achieved hit rate.

### Evaluating hit rate

We first studied the hit rate for the reference RBAC configuration. Figure 3.6(a) presents the hit rate as a function of *cache warmness* for both approximate recycling and precise recycling with the reference policy. As expected, the hit rate of approximate recycling (AR in the figure) increased with cache warmness and was always higher than that of precise recycling (PR in the figure). In addition, the results demonstrate that optimized recycling algorithms achieved a better hit rate than naive recycling algorithms. The improvement was relatively small because it was only due to the increase in secondary allow responses.

Figure 3.6(b) compares the cache size of the naive and optimized approximate recycling algorithms. The results demonstrate that the optimized algorithms help reduce the cache size significantly. Specifically, using the optimized algorithms, the cache size stabilized at about 600KB after cache warmness reached about 20%. Using the naive algorithms, however, the cache size kept increasing with the cache warmness, and eventually reached about 1,700KB. The reason is that optimized algorithms maintain the cache in canonical form. In the rest of our evaluation, we used the optimized algorithms for all the experiments.

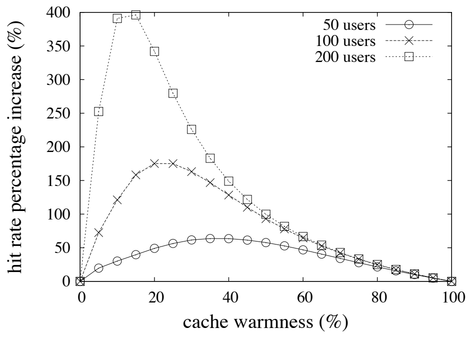
We then studied the impact of varying *the number of users* while the other configuration parameters were fixed. Figure 3.7(a) shows the *percentage increase* for the hit rate compared with precise recycling for an RBAC system that had 50, 100, and 200 users respectively. As expected, an increase in the number of users increased the chance that a role-permission pair was already cached thus leading to a higher hit rate. When averaged over the full range of cache warmness, the percentage increase was 36%, 80%, and 132% for 50, 100, and 200 users respectively.

For the experiments described in the rest of this section, we fixed the cache warmness and studied the impact of other system characteristics on the achieved hit rate. We choose to explore hit rate for relatively low cache warmness values as this is the region where we estimate the system is most likely to operate due to workload characteristics, limited storage space, or frequently changing access control policies.

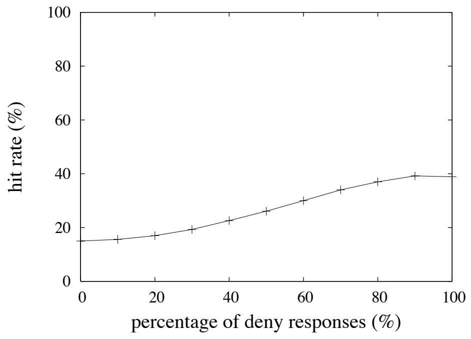
First, we studied the impact of *the percentage of deny responses in the cache.* In some systems, users may know what they are allowed to do and what not, or the user interface may even hide unauthorized actions from users. Hence, the cache may contain more primary allow responses than primary deny responses. To study this effect in the experiment, we engineered the warming set so that the PDP could generate a specified proportion of deny responses, which ranged from 0 to 100%. Figure 3.7(b) confirms our prediction that a higher proportion of deny responses leads to a higher hit rate. The intuition behind this result is that a negative primary response for a permission and a user means that the permission is not assigned to *any* of the user's roles. In contrast, a positive primary response only allows us to infer that the permission is assigned to at least one of the roles, but without the ability to infer exactly which role. Note that we only show the results for 15% cache warmness because the maximum cache warmness we could reach by using only allow responses was less than 20%.

Second, we studied the impact of *the total number of roles* on the hit rate by varying it from 10 to 100 (Figure 3.7(c)) and keeping constant the number of users and the mean number of roles a user/permission is assigned to. The results indicate that, as the number of roles increases, the hit rate decreases. This confirms our intuition that, as the number of roles increases, the overlap between the sets of roles each user is assigned to also decreases thus reducing the likelihood of a successful inference.

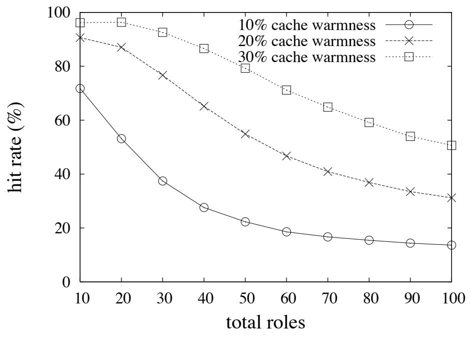
Third, we studied the impact of *the mean number of roles to which each user is assigned*



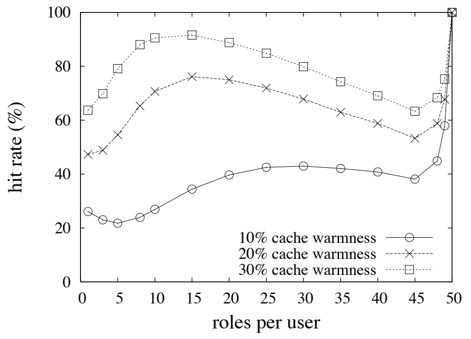
(a) Hit rate percentage increase (compared with precise recycling) as the SDP cache warmness varies, for 50, 100, and 200 users in the RBAC system.



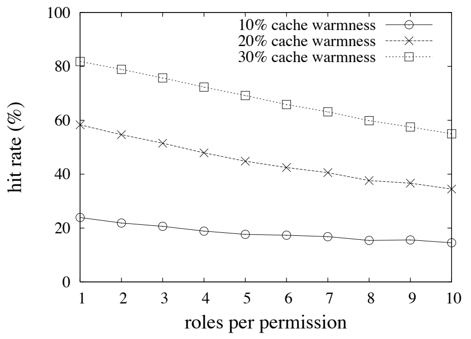
(b) Hit rate variation with the percentage of primary deny responses in the SDP cache for 15% cache warmness.



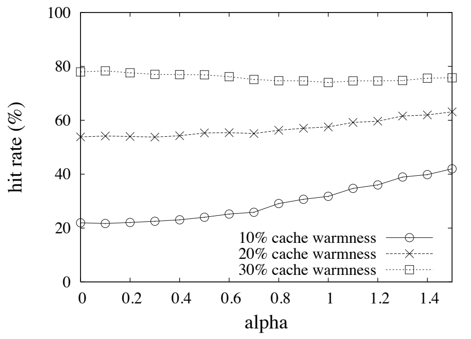
(c) Hit rate variation with the total number of roles in the RBAC system



(d) Hit rate variation with the mean number of roles per user



(e) Hit rate variation with the mean number of roles per permission



(f) Hit rate variation with the coefficient $\alpha$ in Zipf distribution.

Figure 3.7: The impact of various system characteristics on the hit rate.

by varying it from one to all the roles the system (50 roles) while keeping all other parameters constant. The results in Figure 3.7(d) suggest that the influence of this parameter on the hit rate is more complex. We now describe our understanding of these curves. The hit rate was low when each user was assigned to few (less than five) roles because there were few roles in each entry of $Cache^+$ and $Cache^-$ and hence the chances of making an approximate response were limited. As the number of roles per user increased, the size of the entries of the role sets in the cache increased and the chance of two users' role sets overlapping increased. While the overlap was still relatively low (when each user was assigned to less than ten roles), the deny responses dominated the content of the SDP cache. However, when the number of roles per user increased further, $Cache^+$ started increasing at the expense of $Cache^-$, leading to the decrease in the hit rate (as we predicted in Section 3.5). Moreover, for entries of the form $(s, p) \in Cache^+$, $s$ was likely to be large (since there were few deny responses to reduce their size). Since subjects contained all roles assigned to a user and users were assigned to a large number of roles, it became difficult to generate an allow secondary response for $(s, p)$, because $s$ was large and our approach requires a tuple $(s', p) \in Cache^+$ such that $s' \subseteq s$, and in such tuples $s'$ was also likely to be large. Less intuitive is the sharp increase to 100% in the hit rate on the right side of the graph. This increase was likely due to the fact that each user was assigned to (almost) all the roles in the system and, as a result, (almost) every user had the same set of roles. In practice, we would expect the number of roles to be a relatively small compared to the number of users (e.g., [SMJ01] find it to be around 3–4%) and that users will be allocated to a small fraction of those roles. Our experimental results suggest that the characteristics of real RBAC systems will not compromise the efficacy of our algorithms.

Fourth, we studied the impact of *the mean number of roles to which each permission was assigned*. Figure 3.7(e) confirms the results of our analytical analysis, which predicted that a larger number of roles per permission leads to a lower hit rate. This effect can be also attributed to the decrease of $Cache^-$.

Finally, we studied the impact of *role popularity distribution*. In all our previous experiments, roles were uniformly assigned to users and permissions so that all roles were equally "popular" in *UA* and *PA* relations. However, in reality some roles may be assigned to users or permissions more frequently than other roles. For example, in an enterprise most users are assigned an "employee" role while only a few are assigned a "manager" role. To model this type of highly uneven popularity, we used a Zipf distribution.

Zipf distributions have been widely used to model heterogeneous popularity distributions (e.g., web page popularity [BCF+99], web site popularity [AH02], and query term popularity [KLVW04].) A set of data obeys Zipf's law if the frequency of an item is inversely proportional to (some non-negative) power of its rank (determined by frequency of occurrence). More formally, suppose we have a frequency distribution $(x_1, f_1), \ldots, (x_n, f_n)$, where data item

$x_i$ occurs $f_i$ times and $f_1 \geqslant f_2 \geqslant \cdots \geqslant f_n$. Then the distribution obeys Zipf's law if

$$f_i \propto \frac{1}{i^\alpha}$$

for some $\alpha \geqslant 0$. Using English language as an example, the relative frequency of the most popular word "the" is 7%, and the relative frequencies of the next most popular words ("of" and "and") are 3.5% and 2.7%, respectively [FK67]. In other words, the most popular word occurs twice as often as the next most popular, and approximately three times as often as the third most popular word. The frequency of words approximately follows Zipf's law with $\alpha = 1$.

In our experiment, roles selected from the role set $R$ and assigned to users and permissions followed the Zipf distribution. In particular, the more popular roles were assigned to more users in $UA$ than the less popular roles. A role that appeared more frequently in $UA$ (that is, was more commonly assigned to users), however, was assigned to fewer permissions in $PA$. This simulated a scenario where, for example, the "employee" role is usually assigned to more users than the "manager" role but the "employee" role usually has fewer permissions than the "manager" role.

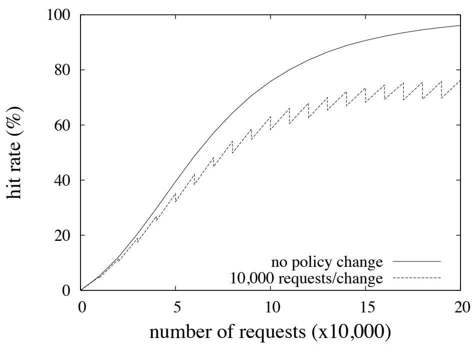
Note that by using a Zipf distribution and varying $\alpha$ for role assignment, we implicitly simulated the existence of a role hierarchy $RH$. A popular role in $UA$ simulated a junior role in $RH$ that had fewer permissions but was assigned to more users. In contrast, a less popular role in $UA$ simulated a senior role in $RH$ that had more permissions (as it inherited permissions from all its junior roles) but was assigned to fewer users. In addition, by varying $\alpha$, we also implicitly varied the shape of the $RH$ graph. When $\alpha$ is small the corresponding $RH$ graph has a wide and shallow shape. A large $\alpha$ makes the $RH$ graph narrow and deep.

Since the popularity distribution becomes less and less skewed with the decrease of $\alpha$, collapsing to a uniform distribution when $\alpha = 0$, we varied $\alpha$ between 0 and 1.5 in steps of 0.1. The results in Figure 3.7(f) show that, when $\alpha$ was lower than 1, the hit rate was almost the same as in the uniform distribution. When $\alpha$ was larger than 1, the hit rate began to increase along with $\alpha$. This is expected because the number of "overlapping" roles between users increased. This was also due to the increase of negative responses in the cache because more users were assigned fewer permissions. However, when the cache warmness increased, this improvement was less significant due to the already high hit rate.

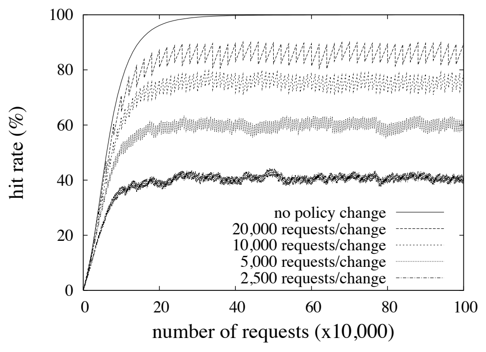
### Evaluating the impact of policy changes

We also studied the impact of policy changes on the hit rate. Since the hit rate depends on the cache warmness, and a policy change may result in removing one or more responses from SDP caches, we expected that frequent policy changes at a constant rate would unavoidably result in a reduced hit rate. This section quantifies this effect.

In the experiments, the simulation engine was responsible for firing a random policy change and sending the policy change message to both the PDP and SDP at pre-defined intervals, e.g.,



(a) Hit rate as a function of number of requests with and without changes to $R$.



(b) Hit rate as a function of number of requests at various frequencies of $R$ changes at a larger time scale.

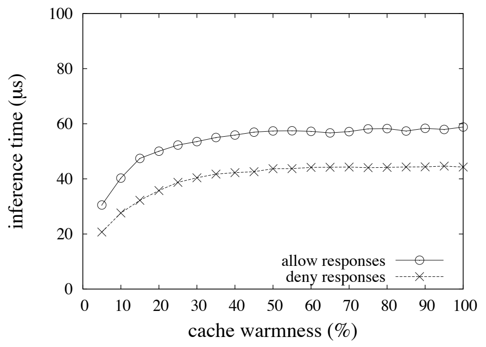
Figure 3.8: The impact of removing a role from $R$ on hit rate. In both figures, the order of the curves (from top to bottom) matches that of the legends.

after every 10,000 requests. The experiment switched from the warming mode to the testing mode once a policy change message was received. After measuring the hit rate right before and after each policy change, the experiment switched back to the warming mode.

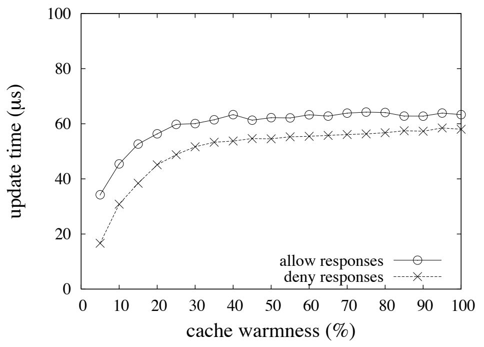
We studied three types of policy change operations: adding a tuple to the *PA* relation; deleting a tuple from the *PA* relation; and deleting a role from $R$. When adding a tuple, the warmness of the cache increases slightly (since a single tuple is added to $Cache^{+}$), so we would expect to see a slight increase in the hit rate. Our experiments confirmed this, although the hit rate never increased by more than 0.1%. Conversely, deleting a tuple from *PA* causes a reduction in the warmness of the cache, and is expected to result in a decrease in the hit rate. Again, our experiments confirmed this expectation, and the decrease in hit rate was negligible.

We now focus on the impact of deleting a role, as it is expected to have a more significant impact on the hit rate. We first studied how the hit rate was affected by an individual policy change, i.e., the removal of a single role from $R$. We expected that $\text{SAAM}_{\text{RBAC}}$ inference algorithms were sufficiently robust that an individual change would result in only minor degradation of the hit rate. In the experiment, the warming set contained 200,000 requests which were selected from the total request space with equal probability (with replacement). A randomly selected role was removed from $R$ every 10,000 requests and tuples containing that role in *UA* and *PA* were also deleted. Then the cache was updated accordingly. After the experiment switched back to the warming mode from the testing mode, the removed role was returned to $R$; *UA* and *PA* were also restored. Thus, the simulated system kept its policy characteristics. Any change in the hit rate was attributed to the reduced SDP cache size.

Figure 3.8(a) shows the hit rate as a function of the number of observed requests, with policy changes (lower curve) or without policy changes (upper curve). Because the hit rate was measured just before and after each policy change, every kink in the curve indicates a hit rate



(a) Inference time (the time to generate approximate responses) variation with cache warmness



(b) Update time (the time to add a primary response to the cache) variation with cache warmness

Figure 3.9: The impact of cache warmness on the inference and update time.

drop caused by a policy change. The results suggest that the hit rate drops were relatively small; the maximum hit rate drop was 6.2%, and the average was 4.0%. After each drop, the curve climbed again because the cache warmness increased with new requests.

Although the hit rate drop for each policy change was small, one can see that the cumulative effect of policy changes could be large. As Figure 3.8(a) shows, the hit rate decreased about 20% in total when the request number reached 200,000. This result led us to another question: Would the hit rate finally stabilize at some point?

To answer this question, we ran another experiment to study how the hit rate varied with continuous policy changes over a longer term. We used a larger number of requests (i.e., 1,000,000), and varied the frequency of policy changes from 2,500 to 20,000 requests per change.

Figure 3.8(b) shows the hit rate as a function of the number of observed requests, with each curve corresponding to a different frequency of random policy changes. Because of the continuous policy change, one cannot see a perfect asymptote of curves. However, the curves indicate that the hit rate stabilized after 200,000 requests. As we expected, the more frequent the policy changes were, the lower the stabilized hit rates were, since the responses were removed from the SDP caches more frequently. This result suggests that if $R$ is changed frequently, it is preferable to purge the cache periodically instead of immediately.

Figure 3.8(b) also shows that each curve has a knee. The steep increase in the hit rate before the knee implies that caching new responses improves the hit rate dramatically in this interval. Once the number of responses passes the knee, the benefit brought by caching further responses becomes negligible.

### Evaluating inference and update time

Figure 3.9(a) shows the inference time for allow and deny approximate responses as a function of cache warmness for our reference configuration. As expected, the computational overhead to infer allow responses was larger than that for deny responses. The inference time increased with cache warmness for two reasons: first, when more responses were cached, the SDP used more responses for inference leading to higher computational overheads. Second, larger cache sizes led to less efficient memory usage by the SDP (that is, SDP data did not fit in the host's cache anymore).

Figure 3.9(b) shows the time for updating the SDP cache using both allow and deny primary responses as a function of cache warmness. As expected, the update time also increased with cache warmness. Additionally, the SDP used more time to process allow than deny responses. The reason is that in the case of processing each allow response $+(s, p)$, the SDP had to purge redundant tuples, i.e., delete all $(s^+, p) \in \mathit{Cache}^+$ such that $s - s^- \subseteq s^+$, which involved an extra subset computation. This result suggests that, to improve the update time, the purge operation should be done in a periodical manner.

It is worth noting in Figure 3.9 that both the inference time and update time stabilized when cache warmness reached about 40%. This was because at about 40% warmness the SDP was able to resolve all possible requests so new responses provided no new information to the cache.

### 3.2.2 Prototype-based evaluation

We have also implemented a simplified SAAM$_{\text{RBAC}}$ prototype system to evaluate the performance of the overall authorization system. In particular, we studied the *response time* for two SAAM authorization schemes (described in Section 3.1.11): sequential authorization and concurrent authorization.

#### Experimental setup

The prototype system consisted of the implementations of PEP, SDP, and PDP. The PEP was process-collocated with the SDP, while the SDP communicated with the PDP using Java Remote Method Invocation (RMI). The PEP/SDP and PDP were located in two separate cluster nodes connected by a 1Gbps network. Each node was equipped with two Intel Xeon 2.33 GHz processors and 4 GB of memory, running Fedora Linux 2.6.24.3. Upon generating a random request at the PEP, the system attempted to resolve the request using one of the following two authorization schemes: *sequential authorization*, where a request was resolved first by the SDP and then by the PDP, or *concurrent authorization*, where a request was resolved by the SDP and the PDP concurrently.

For each authorization scheme, we ran experiments in the following two scenarios: (1) **Scenario I**, where the SDP and the PDP were collocated on the same local area network (LAN) and that the authorization policy of the PDP was relatively simple thus allowing the PDP to make authorization decisions swiftly; and (2) **Scenario II**, where the SDP was separated from the PDP by a wide area network (WAN) or/and the PDP had a complex authorization policy.

To model this scenario, we simulated additional 40ms delay added to each authorization request sent to the PDP.

### Evaluating response time

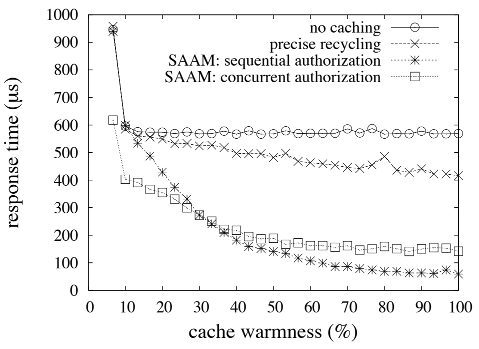
In our experiments, response time was measured as the time elapsed after the PEP generated a request until it received the response for that request. At the start of each experiment, the SDP caches were empty. The PEP uniformly selected a request from the request space, sent it to the SDP, and then recorded the response time for each request. After every 10,000 requests, the PEP calculated the mean response time and used it as an indicator of the response time for that period.

For both scenarios, we also ran experiments for the authorization system without SAAM. This included authorization without caching or only using precise recycling. Our purpose was to evaluate the gains in terms of response time by using SAAM. Figure 3.10 shows the results. It can be concluded that using SAAM helped to reduce the system response time in both scenarios and this reduction increased with cache warmness. Additionally, as we expected, the two SAAM authorization schemes showed different patterns in the two scenarios, which we explain below.

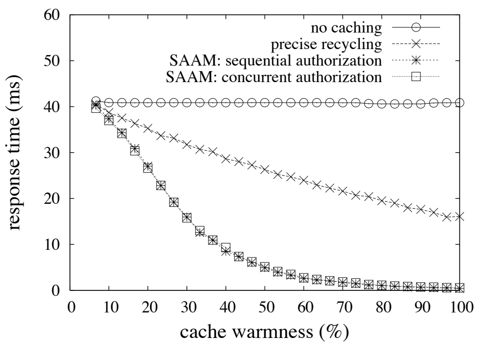
Figure 3.10(a) shows the result for Scenario I. The figure demonstrates that the response time for both authorization schemes decreased with cache warmness, while sequential authorization decreased more quickly. The reason was likely due to the lower cost of resolving requests at the SDP. When cache warmness increased, more requests were resolved by the SDP. Since the SDP was process-collocated with the PEP, getting responses through an interprocess call to the SDP was faster than getting responses through a network RMI call to the PDP.

More specifically, when cache warmness was small, i.e, less than 30%, concurrent authorization achieved shorter response time than sequential authorization. This was due to the extra time incurred by cache misses in sequential authorization. One unusual pattern in our result is that sequential authorization achieved lower response time as cache warmness exceeds 30%. This was possibly caused by the increased load at the SDP and additional thread management overhead in our concurrent authorization implementation. We should point out that, in an optimized implementation, concurrent authorization should at least achieve the same response time as the sequential authorization since concurrent authorization always uses first returned response.

Figure 3.10(b) shows the results for Scenario II. As expected, both response times decreased with cache warmness. More interestingly, the curves for concurrent and sequential $SAAM_{RBAC}$ authorizations almost overlapped each other. The reason is that in this scenario the extra time incurred by cache misses and thread management were small compared to the 40ms delay at the PDP. Therefore, their impact on the response time was trivial.



(a) Scenario I: the SDP and the PDP were collocated on the same LAN and that the authorization policy of the PDP was relatively simple.



(b) Scenario II: the SDP was separated from the PDP by a WAN or/and the PDP had a complex authorization policy.

Figure 3.10: Response time variation with cache warmness.

### 3.2.3 Evaluating hit rate with real-world data

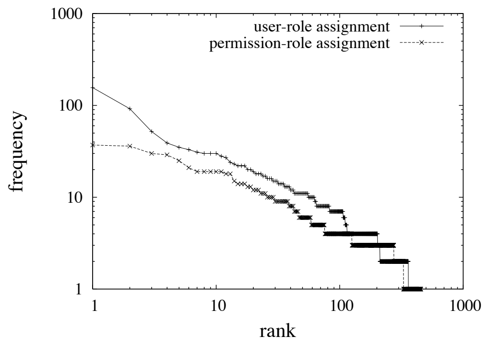
The results described in Section 3.2.1 help us understand the dependancy of hit rate on various factors. However, the results were based on synthetically generated policies and traces. In this section, we describe the experimental results using a real policy and a real trace. Our goal was to understand how the recycling algorithms would perform in selected systems.

#### Real-world policy

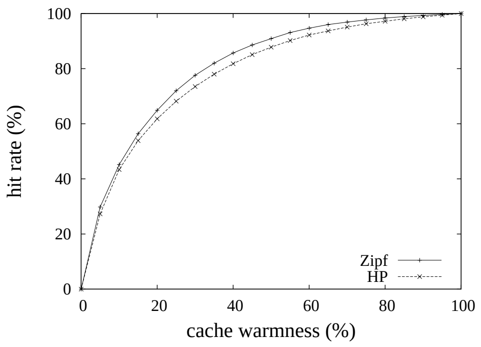
In the first experiment, we used a RBAC policy provided by Hewlett Packard (HP). The policy was extracted from some network access control rules used in HP to authenticate external users and provide them with limited HP network access based on their profiles. The policy contained 1164 subjects, 2044 permissions and 454 roles in total. Note that *UA* and *PA* relations of the policy were generated using the role minimization algorithm proposed in [EHM$^+$08]. Therefore, we expected that this policy represented an optimized policy deployment in an enterprise environment.

We previously used the Zipf distribution for role assignment in *UA* and *PA*, as we assumed that some roles are more popular than others. To verify this assumption, we studied the role assignment in the HP policy. Figure 3.11(a) shows the frequency of roles as a function of its rank in *UA* and *PA*, both axes in logarithmic scales. The results confirm that the role assignments exhibit a Zipf-like distribution, and the coefficients are 0.92 in *UA* and 0.74 in *PA*. We further studied the mean number of roles assigned to each user and mean number of roles assigned to each permission, which are 2.1 and 0.82 respectively.

Based on these findings, we used the Zipf role assignment to generate a synthetic policy that simulated the HP policy, and then ran experiments to compare their hit rates. In both experiments, the traces were generated following a uniform distribution. Figure 3.11(b) shows



(a) Role frequency vs. its rank in logarithmic scale



(b) Hit rate as a function of cache warmness for both HP policy and synthetic policy following the Zipf role assignment.

Figure 3.11: The experiments with the HP policy

the hit rate as a function of cache warmness for both policies. Both curves exhibit the similar shapes as what we have shown in Section 3.2.1. Hence, our simulation results on the hit rate are reasonable.

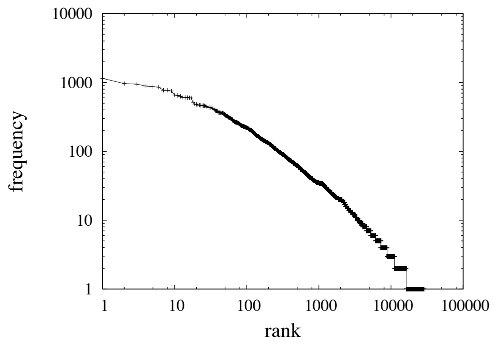
More interestingly, the results demonstrate that the two curves almost overlap each other. This further confirms that Zipf role assignment is able to simulate the role assignment in the real world at a certain level.

### Real-world trace

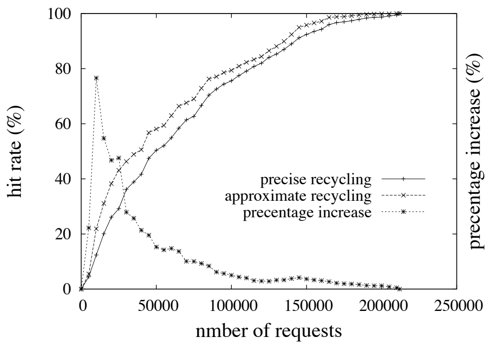
In the previous experiments, the warming set was randomly generated from the request space and all requests were unique. This enabled us to study the hit rate at a certain cache warmness level. In reality, some requests may be more popular than others since users may send those requests more frequently. To study how our recycling algorithm performs with such workload, we used a real-world trace.

The trace was gathered from the log of an online course management application used by a university. The trace recorded all the requests issued in about 3 months' time. After searching the whole trace, we identified 56 users, 6322 objects, 58 actions. Each user was assigned with one or two of the following four roles: course designer, instructor, teaching assistant, and student. For example, an instructor was usually also the course designer. Some teaching assistants might also be course designers. Depending on the assigned role(s), one could perform a number of actions within the application, such as posting new discussion topics, sending mails, and uploading course materials.

The trace contained 211,953 requests in total. Among them 28,734 requests were unique, as users tended to send repeated requests over time. We further studied the popularity distribution of requests. Figure 3.12(a) plots the frequency of requests as a function of its rank in the trace,



(a) Request frequency vs. its rank in logarithmic scale



(b) Hit rate and its percentage improvement as a function of the number of requests.

Figure 3.12: The experiments with the real trace

both axes in logarithmic scales. Interestingly, the results exhibited a Zipf-like distribution and the coefficient was about 1.11. In our context, each request was a combination of subject, object and action. Thereby, we found evidence that requests themselves may also follow a Zipf-like distribution, while the literature [BCF$^+$99] only suggests that objects usually follow Zipf-like distribution.

We used this trace to study the SDP hit rate. The policy was constructed according to the trace. The experiment was run as follows. We warmed the cache to a specific level using a portion of the requests in the trace, measured the hit rate by calculating the percentage of requests in the trace that could be resolved by the SDP using either precise recycling or approximate recycling. We repeated this process until all the requests in the trace had been used to warm the cache.

Figure 3.12(b) plots how the hit rate increases with the number of requests. It suggests that the improvement by approximate recycling is small. The reason is that many requests repeated themselves in the trace so that using precise recycling alone was already able to achieve a high hit rate. Although the absolute improvement is small, the maximum percentage increase can reach 76%.

It is important to note that this trace had several limitations which might restrict the applicability of the above results. First, the trace only contained those requests that had been allowed. Therefore, our recycling algorithm was only partially explored in the experiment. Second, the majority of the users were students so that most users were assigned only one student role. This unavoidably increased the similarity of role set of different users and thus the algorithm performed more efficiently. Third, the percentage of repeated requests was unusually high due to the small scale of the subject and object space, resulting in the high performance of precise recycling. Therefore, the pattern of this trace may not be typical in enterprise applications. We report the results here for the sake of completeness.

### 3.2.4 Discussion

The results of our experiments indicate that approximate recycling leads to higher SDP hit rates than precise recycling alone, thus improving the availability and scalability of the access control system. Compared with the naive algorithms, the optimized algorithms achieve a higher hit rate using a smaller cache. These results extend our understanding of the factors that influence the hit rate as follows:

- For cache warmness between 5% and 50%, the hit rate for approximate recycling is notably better than that of precise recycling.
- Larger numbers of users in the system having similar role memberships substantially improve the hit rate.
- A higher proportion of deny responses in the cache leads to a higher hit rate.
- As the number of roles increases, the overlap between the sets of roles each user is assigned to decreases thus reducing the likelihood of a successful inference based on cached responses.
- The hit rate is low when each user is assigned to few (1-3) roles because the SDP cache has little relevant information. With the increase of overlap in users' roles, the number of relevant entries increases, resulting in the increase of the hit rate. While the overlap is still relatively low (when each user has less than ten roles), the deny responses dominate the content of the SDP cache, resulting in a higher hit rate. However, when the number of roles per user increases further, $Cache^{+}$ starts increasing at the expense of $Cache^{-}$, leading to the decrease in the hit rate. When each user is assigned to (almost) all the roles in the system (almost) every user has the same set of roles, and the hit rate increases sharply to 100%.
- A larger number of roles per permission leads to a lower hit rate.
- Zipf's popularity distribution of roles leads to a higher hit rate when $\alpha$ is larger than 1, due to the increased overlap of roles assigned to users and permissions.

The volume of information available for inference, the percentage of deny responses, and the distribution of role assignment are the factors that are not controlled by the administrators of RBAC systems. Other factors that impact performance, however, e.g., the total number of roles, the number of roles per user, and roles per permission, might be engineered (e.g., by role engineering [VAG07]) by the designers of access control policies who might be able to tune these factors to achieve higher hit rates using the trends our experiments and evaluation revealed. Thus, we believe our evaluation results can be used to inform efficient $\text{SAAM}_{\text{RBAC}}$ deployment in real enterprise application systems, even though our experimental testbed was relatively small compared to large-scale systems deployed in organizations (e.g., [SMJ01]).

Results of our evaluation indicate that the impact of the update to $PA$ is trivial, as only a single permission is affected. In contrast, frequent policy changes to $R$ may have a large impact on the hit rate. Since the correctness of the response is not affected if the cache is not updated immediately, it is preferable to purge the cache periodically instead of immediately.

Our experiments with $\mathrm{SAAM}_{\mathrm{RBAC}}$ also demonstrate inference and update time well under 1ms, and we believe that response times can be further reduced by optimizing the implementation. We note that a low inference time is a key attribute for a real-world deployment as it directly affects the perceived performance of the access control system: an application request cannot be processed until the PEP obtains a response, either primary or secondary. Cache changes triggered by adding primary responses or policy changes, on the other hand, can be implemented in the background to hide their impact on perceived performance.

The evaluation results on response time further suggest the usefulness of SAAM techniques for reducing the response time of the overall access control system, especially in network-based deployments where network latencies are much larger or the PDP authorization logic is complex. The results with two authorization schemes indicate that concurrent authorization is only helpful when the PDP can make authorization decisions quickly. In other cases, sequential authorization is preferable because it can achieve both reduced response time and reduced load at the PDP.

An alternative to approximate recycling for RBAC systems is to replicate RBAC policy at each SDP. Run-time benefits of the proposed approach—compared to just replicating *PA* and *RH* relations at each SDP—depend on a number of factors. The first factor is the size of the policy (mainly the $PA$, since this is likely to be the largest) relative to the size of a PEP working-set (the set of all requests that come through the PEP). For a workload with good locality and a large *PA*, the proposed approach will require less space and may well be faster. Furthermore, if the *PA* is very large (say, larger than $10^9$ elements) then it may be too expensive to duplicate the hardware that supports the PDP to additionally support each SDP. The second factor is the ability of a PDP to predict the working set of a PEP. If the PDP is able to predict a PEP's future working set then providing the SDP with corresponding subsets of *PA* and *RH* will work better than authorization recycling (regardless of the relative sizes of the policy and the PEP working set). The third factor is the frequency of policy changes and the scope of these changes, i.e., how many elements in the *PA* they affect. The fourth factor is the relative benefits brought by one-time replication of the *PA* (or some subset of it)—as proposed by [TC09], for example—to the SDPs, as opposed to item-by-item caching of the responses.

Depending on the workload and policy characteristics, the most efficient solution may combine the proposed approach with the replication of some policy elements. For example, *RH* can be replicated to the SDPs, as suggested in Section 3.1.9. As a case in point, PEPs in the IBM Tivoli Access Manager [Kar03], which encodes *PA* in the form of access control lists, can operate in two modes. In "remote mode," a PEP sends authorization requests to the PDP. In "local mode," the PEP maintains a local replica of the authorization policy and performs all authorization decisions locally. Depending on the configuration, the policy local replica can be “pulled from” and/or “pushed” by the master authorization service database. “Overhead of policy replication” is mentioned in the technical documentation of the Access Manager [BAR+03], but no evaluation is reported.

## 3.3 Summary

In this chapter, we have presented $\text{SAAM}_{\text{RBAC}}$—the SAAM authorization recycling algorithms for RBAC systems. We define two inference rules, $\textbf{Rule}^{+}$ and $\textbf{Rule}^{-}$ that are specific to RBAC authorization semantics, and develop the recycling algorithms based on these rules. In particular, we have developed several algorithms, including the following three: the first caches authorization responses from the PDP and represents them as a compact data structure; the second uses this data structure to generate secondary (precise or approximate) responses; the third handles policy changes by updating the data structure. We show that the computational complexity of the algorithms is bounded by the cache size and the number of roles in the system.

We implement $\text{SAAM}_{\text{RBAC}}$ algorithms and evaluate their properties using an experimental testbed with 100 subjects, 3,000 permissions and 50 roles. Evaluation results demonstrate an average 80% increase (over the full range of cache warmness), compared to precise recycling, in the number of authorization requests that can be served without consulting access control policies stored remotely at the PDP. These results suggest that deploying $\text{SAAM}_{\text{RBAC}}$ improves the availability and scalability of RBAC systems, which in turn improves the performance of the enterprise application systems.

# Chapter 4

# Cooperative Secondary Authorization Recycling

The previous chapter presents SAAM$_{\mathrm{RBAC}}$—the authorization recycling algorithms for role-based access control (RBAC) systems. A local secondary decision point (SDP) can use SAAM$_{\mathrm{RBAC}}$ algorithms to resolve authorization requests not only by reusing cached authorizations but also by computing approximate authorizations from cached authorizations, even when the remote policy decision point (PDP) fails. This chapter describes a cooperative secondary authorization recycling (CSAR) mechanism which is, in essence, a distributed version of the secondary and approximate authorization model (SAAM). A CSAR system explores the cooperation among distributed SDPs which can further improve the availability and performance of access control systems.

In Section 4.1, we present the system design. Our design aims to meet five requirements: low overhead, ability to deal with malicious SDPs, consistency, configurability and backward compatibility. A discovery service is developed to facilitate the cooperation between SDPs. In particular, the discovery service helps one SDP find other SDPs that might be able to resolve a request. To achieve consistency, we propose the alternatives to propagate update messages and a solution to implement well-defined semantics for policy updates. To achieve higher hit rate, we describe an eager approach by using the responses in the evidence of an approximate response to warm the cache.

In Section 4.2, we present the evaluation on CSAR$_{\mathrm{BLP}}$, which is based on SAAM$_{\mathrm{BLP}}$ recycling algorithms. We use simulations and a prototype to evaluate CSAR's feasibility and benefits. Evaluation results show that by adding cooperation to SAAM, our approach further improves the availability and performance of authorization infrastructures. In Section 4.3, we also briefly discuss the evaluation results for CSAR$_{\mathrm{RBAC}}$.

Finally, Section 4.4 summarizes this chapter.

## 4.1 CSAR design

This section presents the design requirements for cooperative authorization recycling, the CSAR system architecture, and finally the detailed CSAR design.

### 4.1.1 Design requirements

The CSAR system aims to improve the availability and performance of access control infrastructures by sharing authorization information among cooperative SDPs. Each SDP resolves the requests from its own policy decision point (PEP) by locally making secondary authorization decisions, by involving other cooperative SDPs in the authorization process, and/or by passing the request to the PDP.

Since the system involves caching and cooperation, we consider the following design requirements:

- **Low overhead.** As each SDP participates in making authorizations for some non-local requests, its load is increased. The design should therefore minimize this additional computational and communication overhead.

- **Ability to deal with malicious SDPs.** As each PEP enforces responses that are possibly offered by non-local SDPs, the PEP should be prepared to deal with those SDPs that after being compromised become malicious. For example, it should verify the validity of each secondary response by tracing it back to a trusted source.

- **Consistency.** Brewer [Bre00] conjectures and Lynch et al. [GL02] prove that distributed systems cannot simultaneously provide the following three properties: availability, consistency, and network partition tolerance. We believe that availability and partition tolerance are essential properties that an access control system should offer. We thus relax consistency requirements in the following sense: with respect to an update action, various components of the system can be inconsistent for at most a user-configured finite time interval.

- **Configurability.** The system should be configurable to adapt to different performance objectives at various deployments. For example, a deployment with a set of latency-sensitive applications may require that requests are resolved in minimal time. A deployment with applications generating a high volume of authorization requests, on the other hand, should attempt to eagerly exploit caching and the inference of approximate authorizations to reduce load on the PDP, the bottleneck of the system.

- **Backward compatibility.** The system should be backward compatible so that minimal changes are required to existing infrastructures—i.e., PEPs and PDPs—in order to switch to CSAR.

### 4.1.2 System architecture

This section presents an overview of the system architecture and discusses our design decisions in addressing the configurability and backward compatibility requirements.

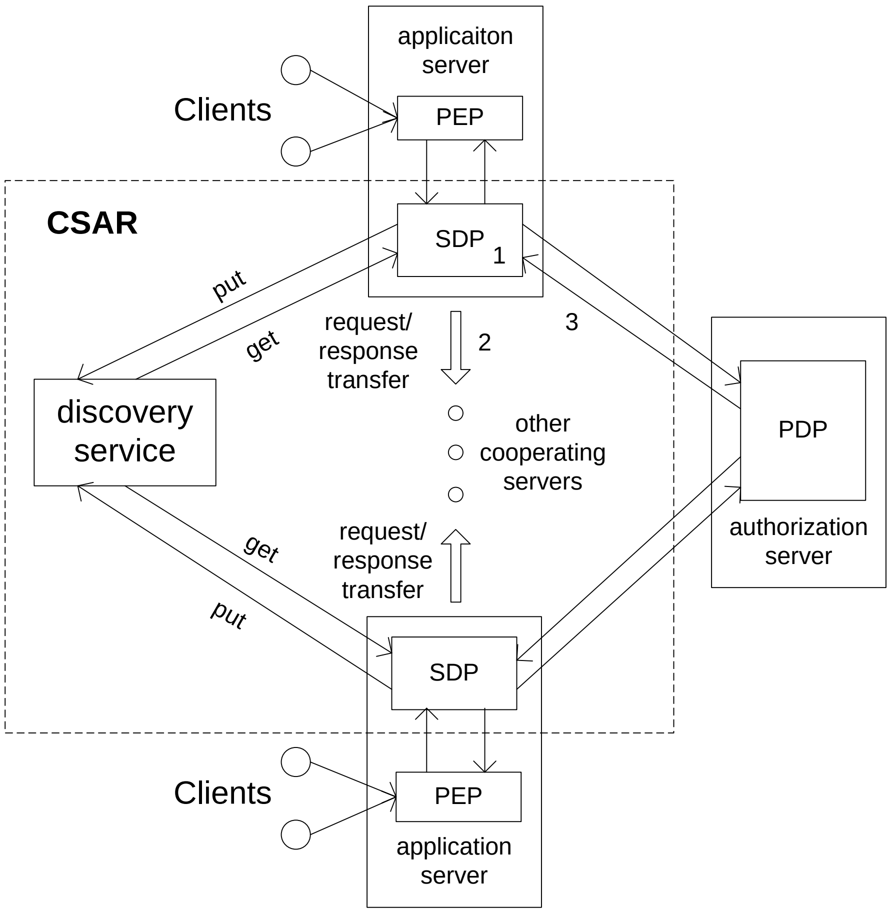


Figure 4.1: CSAR introduces cooperation between SDPs.

As illustrated by Figure 4.1, a CSAR deployment contains multiple PEPs, SDPs, and one PDP. Each SDP is host-collocated with its PEP at an application server. Both the PEP and SDP are either part of the application or of the underlying middleware. The PDP is located at the authorization server and provides authorization decisions to all applications. The PEPs mediate the application requests from clients, generate authorization requests from application requests, and enforce the authorization decisions made by either the PDP or SDPs.

For increased availability and lower load on the central PDP, our design exploits the cooperation between SDPs. Each SDP computes responses to requests from its PEP, and can participate in computing responses to requests from other SDPs. Thus, authorization requests and responses are transferred not only between the application server and the authorization server, but also between cooperating application servers.

CSAR is configurable to optimize the performance requirements of each individual deployment. Depending on the specific application, geographic distribution and network characteristics of each individual deployment, performance objectives can vary from reducing the overall load on the PDP, to minimizing client-perceived latency, and to minimizing the network traffic generated.

Configurability is achieved by controlling the degree of concurrency in the set of operations involved in resolving a request: (1) the local SDP can resolve the request using data cached locally; (2) the local SDP can forward the request to other cooperative SDPs to resolve it using their cached data; and (3) the local SDP can forward the request to the PDP. If the performance objective is to reduce latency, then the above three steps can be performed concurrently, and the SDP will use the first response received. If the objective is to reduce network traffic and/or the load at the central PDP, then the above three steps are performed sequentially (in an appropriate order).

CSAR is designed to be easily integrated with existing access control systems. Each SDP provides the same policy evaluation interface to its PEP as the PDP, thus the CSAR system can be deployed incrementally without requiring any change to existing PEP or PDP components. Similarly, in systems that already employ authorization caching but do not use CSAR, the SDP can offer the same interface and protocol as the legacy component.

### 4.1.3 Discovery service

One essential component enabling cooperative SDPs to share their authorizations is the discovery service (DS), which helps an SDP find other SDPs that might be able to resolve a request.

A naive approach to implementing the discovery functionality is request broadcasting: whenever an SDP receives a request from its PEP, it broadcasts the request to all other cooperating SDPs. All SDPs attempt to resolve the request, and the PEP enforces the response it receives first. This approach is straightforward and might be effective when the number of cooperating SDPs is small and the cost of broadcasting is low. However, it has two important drawbacks. First, it inevitably increases the load on all SDPs. Second, it causes high traffic overhead when SDPs are geographically distributed.

To overcome these two drawbacks, we introduced the DS to achieve a selective request distribution: an SDP in CSAR selectively sends requests only to those SDPs that are likely to be able to resolve them. This process is however specific to the underlying authorization recycling algorithms. Below we discuss it for $\mathrm{SAAM}_{\mathrm{BLP}}$ and $\mathrm{SAAM}_{\mathrm{RBAC}}$ separately.

- **$\mathbf{SAAM}_{\mathbf{BLP}}$.** As we illustrate in Section 2.6, the $\mathrm{SAAM}_{\mathrm{BLP}}$ inference algorithms use cached responses to infer information about the relative ordering on security labels associated with subjects and objects. Therefore, for an SDP to resolve a request, the SDP's cache must contain at least both the subject and object of the request. If either one is missing, there is no way for the SDP to infer the relationship between the subject and object, and thus fails to compute a secondary response. The role of the DS is to store and retrieve the mapping between subject/object and SDP addresses. In particular, the DS provides an interface with the following two functions: *put* and *get*. Given a subject or an object and the address of an SDP, the *put* function stores the mapping

$(subject/object, SDPaddress)$. A *put* operation can be interpreted as “this SDP knows something about the subject/object.” Given a subject and object pair, the *get* function returns a list of SDP addresses that are mapped to both the subject and object. The results returned by the *get* operation can be interpreted as “these SDPs know something about both the subject and object and thus might be able to resolve a request involving them.”

- **SAAM$_{\mathbf{RBAC}}$**. In this case, the SDP’s cache must contain at least one response for the requested permission, as the SAAM$_{\text{RBAC}}$ inference algorithms are performed between the responses for the same permission. If cache contains no primary response for a permission, then it is impossible for the SDP to infer an approximate response for that permission. The role of the DS is then to store and retrieve the mapping between permissions and SDP addresses. In particular, given a permission and the address of an SDP, the *put* function stores the mapping $(permission, SDPaddress)$. A *put* operation can be interpreted as “this SDP knows something about the permission.” Given a permission, the *get* function returns a list of SDP addresses that are mapped to that permission. The results returned by the *get* operation can be interpreted as “these SDPs know something about the permission and thus might be able to resolve a request for it.”

Using DS avoids broadcasting requests to all cooperating SDPs. Whenever an SDP receives a primary response to a request, it calls the *put* function to register itself in the DS as a suitable SDP for the subject/object or the permission of the request. When cooperation is required, the SDP calls the *get* function to retrieve from the DS a set of addresses of those SDPs that might be able to resolve the request.

Note that the DS is only logically centralized, but can have a scalable and resilient implementation. In fact, an ideal DS should be distributed and collocated with each SDP to provide high availability and low latency: each SDP can make a local *get* or *put* call to publish or discover cooperative SDPs, and the failure of one DS node will not affect others. Compared to the PDP, the DS is both simple—it only performs *put* and *get* operations—and general—it does not depend on the specifics of any particular security policy. As a result, a scalable and resilient implementation of DS is easier to achieve.

For instance, one can use a Bloom filter to achieve a distributed DS implementation, similar to the summary cache [FCAB00] approach. The Bloom filter must use a family of hashing functions. Each SDP builds a Bloom filter from the subjects or objects of cached requests, and sends the bit array plus the specification of the hash functions to the other SDPs. The bit array is the summary of the subjects/objects that this SDP has stored in its cache. Each SDP periodically broadcasts its summary to all cooperating SDPs. Using all summaries received, a specific SDP has a global image of the set of subjects/objects stored in each SDP’s cache, although the information could be outdated or partially wrong.

For a small-scale cooperation, a centralized DS implementation might be feasible where various approaches can be used to reduce its load and improve its scalability. The first approach is to reduce the number of *get* calls. For instance, SDPs can cache the results from the DS for a small period of time. This method can also contribute to reducing the latency. The second approach is to reduce the number of *put* calls. For example, SDPs can update the DS in batch mode instead of calling the DS for each primary response.

In Chapter 5, we show that we can also use a publish-subscribe channel to implement the DS. The benefits of using a publish-subscribe channel are that it provides a unified cooperation between SDPs and PDPs as well as *speculative* authorizations, where authorizations can be pre-computed and cached for future requests.

### 4.1.4 Adversary model

In our adversary model, an attacker can eavesdrop, spoof or replay any network traffic or compromise an application server host with its PEP(s) and SDP(s). The adversary can also compromise the client computer(s) and the DS. Therefore, there could be malicious clients, PEPs, SDPs and DS in the system.

As a CSAR system includes multiple distributed components, our design assumes different degrees of trust in them. The PDP is the ultimate authority for access control decisions and we assume that all PEPs trust[10] the decisions made by the PDP. We also assume that the policy change manger (introduced later in Section 4.1.6) is trusted because it is collocated and tightly integrated with the PDP. We further assume that each PEP trusts those decisions that it receives from its own SDP. However, an SDP does not necessarily trust other SDPs in the system.

### 4.1.5 Mitigating threats

Based on the adversary model presented in the previous section, we now describe how our design enables mitigation of the threats due to malicious DS and SDPs.

A malicious DS can return false or no SDP addresses, resulting in threats of three types: (1) the SDP sends requests to those SDPs that actually cannot resolve them, (2) all the requests are directed to a few targeted SDPs, (3) the SDP does not have addresses of any other SDP. In all three cases, a malicious DS impacts system performance through increased network traffic, or response delays, or computational load on SDPs, and thus can mount a denial-of-service (DoS) attack.

A malicious DS may also compromise system correctness when we consider policy changes. In Section 4.1.6, we describe three types of policy changes: critical, time-sensitive, and time-insensitive changes. In delivering critical changes, the DS is used to find those SDPs that should be contacted. In this case, if the DS returns an incorrect list of SDPs, some SDPs may receive unexpected policy changes or may not receive desired policy changes, which may result in those SDPs computing incorrect responses. However, we note that a malicious DS can have

[10]By "trust" we mean that if a trusted component turns to be malicious, it can compromise the security of the system.

an impact on system correctness only in the case of critical changes. For time-sensitive and time-insensitive changes, a malicious DS has no impact as it is not involved in propagating them.

To detect a malicious DS, an SDP can track how successful the remote SDPs whose addresses the DS provides are in resolving authorization requests. A benign DS, which always provides correct information, will have a relatively good track record, with just few SDPs unable to resolve requests. Even though colluding SDPs can worsen the track record of a DS, we don't believe such an attack to be of practical benefit to the adversary.

A malicious SDP could generate any response it wants, for example, denying all requests and thus launching a DoS attack. Therefore, when an SDP receives a secondary response from other SDPs, it verifies the authenticity and integrity of the primary responses used to infer that response as well as the correctness of the inference.

To protect the authenticity and integrity of a primary response while it is in transit between the PDP and the SDP, the PDP cryptographically signs the response. Then, an SDP can independently verify the primary response's authenticity and integrity by checking its signature, assuming it has access to the PDP's public key. Recall that each secondary response includes an evidence list that contains the primary responses used for inferring this response. If any primary response in the evidence cannot be verified, that secondary response is deemed to be incorrect.

To verify the correctness of a response, the SDP needs to use the knowledge of both the inference algorithm and evidence list. A secondary response is correct if the PDP would compute the same response. The verification algorithm depends on the inference algorithm. We have discussed the $\text{SAAM}_{\text{RBAC}}$ algorithms in Section 3.1.10. In the case of $\text{SAAM}_{\text{BLP}}$, it is simply the inverse of the inference algorithm.

Recall that the $\text{SAAM}_{\text{BLP}}$ inference algorithm searches the cached responses and identifies the relative ordering on security labels associated with the request's subjects and objects. In contrast, the verification algorithm goes through the evidence list of primary responses by reading every two consecutive responses and checking whether the correct ordering can be derived. To illustrate, consider the following example. A remote SDP returns a response $(r_4, i_4, [r_1, r_2, r_3], \mathit{allow})$ for request $(s_1, o_2, \mathit{read}, c_4, i_4)$, where $r_1$ is the primary *allow* response for $(s_1, o_1, \mathit{read}, c_1, i_1)$, $r_2$ is the primary *allow* response for $(s_2, o_1, \mathit{append}, c_2, i_2)$ and $r_3$ is the primary *allow* response for $(s_2, o_2, \mathit{read}, c_3, i_3)$. From these responses, the verification algorithm can determine that $\lambda(s_1) > \lambda(o_1) > \lambda(s_2) > \lambda(o_2)$. Therefore, $s_1$ should be allowed to read $o_2$, and thus $r_4$ is a correct response.

Verification of each approximate response unavoidably introduces additional computational cost, which depends on the length of the evidence list. A malicious SDP might use this property to attack the system. For example, a malicious SDP can always return responses with a long evidence list that is computationally expensive to verify. One way to defend against this attack is to set an upper bound to the time that the verification process can take. An SDP that always returns long evidence lists will be blacklisted.

We defined four execution scenarios, listed below, to help manage the computational cost caused by response verification. Based on the administration policy and deployment environment, the verification process can be configured differently to achieve various trade-offs between security and performance.

- **Total verification.** All responses are verified.
- **Allow verification.** Only 'allow' responses are verified. This configuration protects resources from unauthorized access but might be vulnerable to DoS attacks.
- **Random verification.** Responses are randomly selected for verification. This configuration can be used to detect malicious SDPs but cannot guarantee that the system is perfectly correct, since some false responses may have been generated before the detection.
- **Offline verification.** There is no real-time verification, but offline audits are performed.

### 4.1.6 Consistency

Similar to other distributed systems employing caching, CSAR needs to deal with cache consistency issues. In our system, SDP caches may become inconsistent when the access control policy changes at the PDP. In the previous chapter (Section 3.1.9), we briefly describe the consistency mechanisms. In this section, we elaborate on how consistency is achieved in CSAR.

We first state our assumptions relevant to the access control systems. We assume that the PDP makes decisions using an access control policy stored persistently in a policy store of the authorization server. In practice, the policy store can be a policy database or a collection of policy files. We further assume that security administrators deploy and update policies through the policy administration point (PAP), which is consistent with the XACML architecture [Com05]. To avoid modifying existing authorization servers and maintain backward compatibility, we further add a policy change manager (PCM), collocated with the policy store. The PCM monitors the policy store, detects policy changes, and informs the SDPs about the changes. The refined architecture of the authorization server is presented in Figure 4.2.

Based on the fact that not all policy changes are at the same level of criticality, we divide policy changes into three types: critical, time-sensitive, and time-insensitive changes. By discriminating policy changes according to these types, system administrators can choose to achieve different consistency levels. In addition, system designers are able to provide different consistency techniques to achieve efficiency for each type. Our design allows a CSAR deployment to support any combination of the three types. In the rest of this section, we define each type of policy change and discuss the consistency properties.

***Critical changes*** of authorization policies are those changes that need to be propagated urgently throughout the enterprise applications, requiring immediate updates on all SDPs. When an administrator makes a critical change, our approach requires that she also specifies a

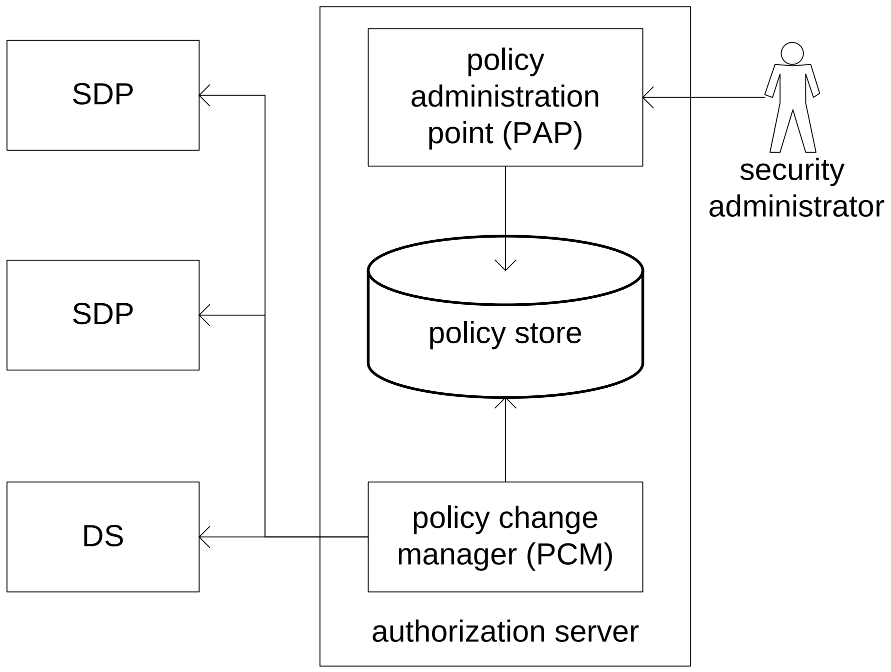


Figure 4.2: The architecture enabling the support for policy changes.

time period $t$ for the change. CSAR will attempt to make the policy change by contacting all SDPs involved, and must send the administrator within time period $t$ either a message that the change has been successfully performed or a list of SDPs that have not confirmed the change.

We developed a **selective-flush** approach to propagating critical policy changes. In this approach, only selected policy changes are propagated, only selected SDPs are updated, and only selected cache entries are flushed. We believe that this approach has the benefits of reducing server overhead and network traffic. In the following we sketch out the propagation process.

The PCM first determines which subjects and/or objects (a.k.a. entities) are affected by the policy change. Since most modern enterprise access control systems make decisions by comparing security attributes (e.g., roles, clearance, sensitivity, groups) of subjects and objects, the PCM maps the policy change to the entities whose security attributes are affected. For example, if permission $p$ has been revoked from role $r$, then the PCM determines all objects of $p$ (denoted by $O^p$) and all subjects assigned to $r$ (denoted by $S^r$).

The PCM then finds out which SDPs need to be notified of the policy change. Given the entities affected by the policy change, the PCM uses the discovery service (DS) to find those SDPs that might have responses for the affected entities in their caches. The PCM sends the DS a policy change message containing the affected entities, $(O^p, S^r)$. Upon receiving the message, the DS first replies back with a list of the SDPs that have cached the responses for the entities. Then it removes corresponding entries from its map to reflect the flushing. After the PCM gets the list of SDPs from the DS, it multicasts the policy change message to these affected SDPs.

When an SDP receives a policy change message, it flushes those cached responses that contain the entities and then acknowledges the results to the PCM. In the above example, with revoking permission $p$ from role $r$, the SDP would flush those responses from its cache that contain both objects in $O^p$ and subjects in $S^r$.

In order for the selective-flush approach to be practical, the PCM should have the ability to quickly identify the subjects or objects affected by the policy change. However, this procedure may not be trivial due to the complexities of modern access control systems. We have developed identification algorithms for the policies based on the BLP model, and will explore this issue for other access control models in future research.

***Time-sensitive changes*** in authorization policies are less urgent than critical ones but still need to be propagated within a known period of time. When an administrator makes a time-sensitive change, it is the PCM that computes the time period $t$ in which caches of all SDPs are guaranteed to become consistent with the change. As a result, even though the PDP starts making authorization decisions using the modified policy, the change comes into effect throughout the CSAR deployment only after time period $t$. Notice that this does not necessarily mean that the change itself will be reflected in the SDPs' caches by then, only that the caches will not use responses invalidated by the change.

CSAR employs a time-to-live (TTL) approach to process time-sensitive changes. Every primary response is assigned a TTL that determines how long the response should remain valid in the cache, e.g., one day or one hour. The assignment can be performed by either the SDP, the PDP itself, or a proxy, through which all responses from the PDP pass before arriving at the SDPs. The choice depends on the deployment environment and backward compatibility requirements. Every SDP periodically purges from its cache those responses whose TTL elapses.

The TTL value can also vary from response to response. Some responses (say, authorizing access to more valuable resources) can be assigned a smaller TTL than others. For example, for a BLP-based policy, the TTL for the responses concerning *top-secret* objects could be shorter than for *confidential* objects.

When the administrator makes a ***time-insensitive change***, the system guarantees that all SDPs will *eventually* become consistent with the change. No promises are given, however, about how long it will take. Support for time-insensitive changes is necessary because some systems may not be able to afford the cost of, or are just not willing to support, critical or time-sensitive changes. A simple approach to supporting time-insensitive change is for system administrators to periodically restart the machines hosting the SDPs.

### 4.1.7 Eager recycling

In previous sections, we explained how cooperation among SDPs is achieved by resolving requests by remote SDPs. In this section, we describe an eager approach to recycling past responses. The goal is to further reduce the overhead traffic and response time.

The essence of cooperation is SDPs helping each other in order to reduce the cache miss rate at each SDP. We considered two types of cache misses: *compulsory* misses, which are generated by a subject's first attempt to access an object, and *communication/consistency* misses, which occur when a cache holds a stale authorization. With cooperation, the SDP can avoid some of these misses by possibly getting authorizations from its cooperating SDPs.

Eager recycling can help further reduce the cache miss rate. As stated before, if an SDP succeeds in resolving a request from another SDP, it returns a secondary response, which includes an evidence component. The evidence contains a list of primary responses that have been used to infer the secondary response. In eager recycling, the receiving SDP incorporates those verified primary responses into its local cache as if it received them from the PDP. By including these responses, the SDP's cache increases faster and its chances of resolving future requests locally by inference also increases. Our evaluation results show that this approach can reduce the response time by a factor of two.

## 4.2 Evaluation of $CSAR_{BLP}$

We first evaluated $CSAR_{BLP}$, in which the underlying access control model is BLP and the SDP uses $SAAM_{BLP}$ algorithms [CLB06] for recycling authorizations. In evaluating $CSAR_{BLP}$, we wanted first to determine if our design works. Then we sought to estimate the achievable gains in terms of availability and performance, and determine how they depend on factors such as the number of cooperating SDPs and the frequency of policy changes.

We used both simulation and a prototype implementation to evaluate $CSAR_{BLP}$. The simulation enabled us to study availability by hiding the complexity of underlying communication, while the prototype enabled us to study both performance and availability in a more dynamic and realistic environment. Additionally, we have integrated our prototype with a real application to study the integration complexity and the impact of application performance.

We used a similar setup for both the simulation and prototype experiments. The PDP made access control decisions on either read or append requests using a BLP-based policy stored in an XML file. The BLP security lattice contained 4 security levels and 3 categories, 100 subjects and 100 objects, and uniformly assigned security labels to them. The total number of possible requests was 100x100x2=20,000. The policy was enforced by all the PEPs. Each SDP implemented the same inference algorithm. While the subjects were the same for each SDP, the objects could be different in order to simulate the request overlap.

### 4.2.1 Simulation-based evaluation

We used simulations to evaluate the benefits of cooperation to system availability and reducing load at the PDP. We used the cache hit rate as an indirect metric for these two characteristics. A request resolved without contacting the PDP was considered a cache hit. A high cache hit rate results in masking transient PDP failures (thus improving the availability of the access control system) and reducing the load on the PDP (thus improving the scalability of the system).

We studied the influence of the following factors on the hit rate of one cooperating SDP:

(a) the cache warmness at each SDP; (b) the number of cooperating SDPs; (c) the *overlap rate* between the resource spaces of two cooperating SDPs, defined as the ratio of the objects owned by both SDPs to the objects owned only by the studied SDP (The overlap rate served as a measure of similarity between the resources of two cooperating SDPs); (d) whether the inference for approximate responses was enabled or not; and (e) the popularity distribution of requests.

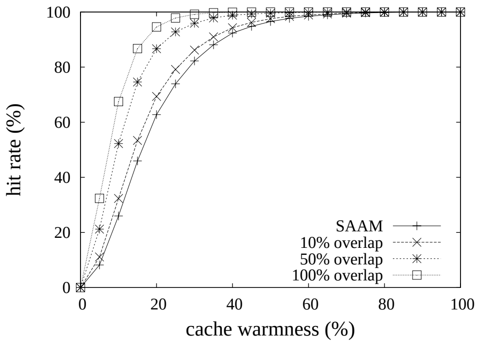
To conduct the experiments, we have modified the SAAM evaluation engine used in [CLB06] to support cooperation. Each run of the evaluation engine involved four stages. In the first stage, the engine generated subjects and objects for each SDP, created a BLP lattice and assigned security labels to both subjects and objects. To control the overlap rate (e.g., 10%) between SDPs, we first generated the object space for the SDP under study (e.g., obj0–99). For each of the other SDPs, we then uniformly selected the corresponding number of objects (e.g., 10) from the space of the SDP under study (e.g., obj5, obj23, etc.) and then generated the remaining objects sequentially (e.g., obj100–189).

Second, the engine created the *warming set* of requests for each SDP: that is the set containing all possible unique requests for that SDP. We also created a testing set for all SDPs, which comprised a sampling of requests uniformly selected from the request space of the SDP under study. In our experiment, the testing set contained 5,000 requests. Next, the simulation engine started operating by alternating between *warming* and *testing* modes. In the warming mode (stage three), the engine used a subset of the requests from each warming set, evaluated them using the simulated PDP, and sent the responses to the corresponding SDP to build up the cache. Once the desired cache warmness was achieved, the engine switched into testing mode (stage four) where the SDP cache was not updated anymore. We used this stage to evaluate the hit rate of each SDP at controlled, fixed levels of cache warmness. The engine submitted requests from the testing set to all SDPs. If any SDP could resolve a request, it was a cache hit. The engine calculated the hit rate as the ratio of the test requests resolved by any SDP to all test requests at the end of this phase. These last two stages were then repeated for different levels of cache warmness, from 0% to 100% in increments of 5%.

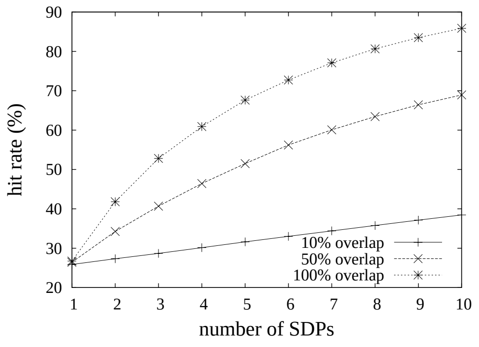
Simulation results were gathered on a commodity PC with a 2.8 GHz Intel Pentium 4 processor and 1 GB of RAM. The simulation framework was written in Java and ran on Sun's 1.5.0 JRE. Experiments used the same cache warmness for each SDP and the same overlap rate between the inspected SDP and every other cooperating SDP. We simulated three overlap rates: 10%, 50%, or 100%. Each experiment was run ten times and the average results are reported.

Figure 4.3 shows the results for requests that followed a uniform popularity distribution. Figure 4.3(a) shows the dependency of the hit rate on cache warmness and overlap rate. It compares the hit rate for the case of one SDP, representing SAAM (bottom curve), with the hit rate achieved by 5 cooperating SDPs. Figure 4.3(a) suggests that, when cache warmness was low (around 10%), the hit rate was still larger than 50% for overlap rates of 50% and up. In particular, when the overlap rate was 100%, CSAR achieved a hit rate of almost 70% at 10%

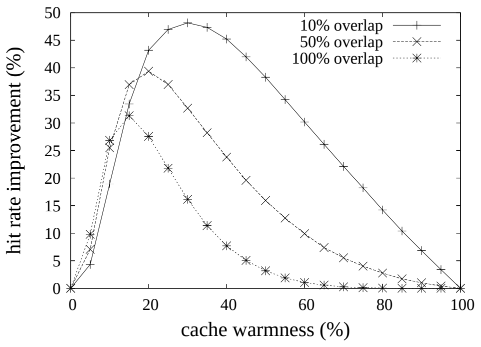
cache warmness. In reality, low cache warmness can be caused by the characteristics of the workload, by limited storage space, or by frequently changing access control policies. For a 10% overlap rate, however, CSAR outperformed SAAM by a mere 10%, which might not warrant the cost of CSAR's complexity.



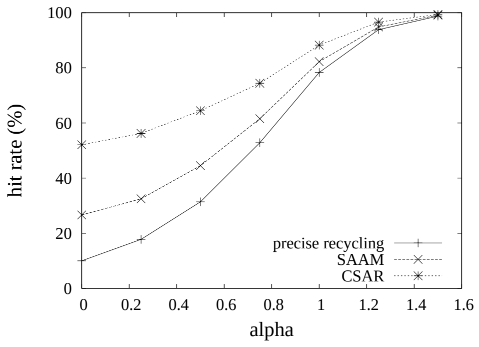
(a) Hit rate as a function of cache warmness for 5 SDPs compared to 1 SDP (i.e., SAAM).



(b) Hit rate as a function of number of SDPs at cache warmness of 10%.



(c) Hit rate improvement of approximate recycling over precise recycling as a function of cache warmness when cooperation is enabled (5 SDPs).



(d) Hit rate as a function of Zipf coefficient. When alpha is 0, it is a uniform distribution.

Figure 4.3: The impact of various parameters on hit rate. The requests for subfigures (a)—(c) follow a uniform popularity distribution.

Figure 4.3(b) demonstrates the impact of the number of cooperating SDPs on the hit rate under three overlap rates. In the experiment, we varied the number of SDPs from 1 to 10, while maintaining 10% cache warmness at each SDP. As expected, increasing the number of SDPs led to higher hit rates. At the same time, the results indicate that additional SDPs provided diminishing returns. For instance, when the overlap rate was 100%, the first cooperating SDP brought a 14% improvement in the hit rate, while the 10th SDP contributed only 2%. One can thus limit the number of cooperating SDPs to control the overhead traffic without losing the major benefits of cooperation. The results also suggest that in a large system with many SDPs, the impact of a single SDP's failure on the overall hit rate is negligible. On the other side, when the overlap rate is small, a large number of SDPs are still required to achieve a high hit rate.

Figure 4.3(c) shows the absolute hit rate improvement of approximate recycling over precise recycling when cooperation was enabled in both cases. We can observe from Figure 4.3(c) that the largest improvement occurred when the cache warmness was low. This was due to the strong inference ability of each SDP even at low cache warmness. When the overlap rate decreased, the tops of the curves shifted to the right, which implies that, for a smaller overlap rate, greater cache warmness is needed to achieve more improvement.

In addition, the peak in each curve decreased with the overlap rate. This lowering of the peak appears to be caused by the reduced room for improvement left to the approximate recycling. When the overlap rate increased, the hit rate of precise recycling was already high due to the benefit brought by cooperation.

To study how the request distribution affects hit rate, we also simulated the requests that follow a Zipf object popularity distribution. In the experiment, we varied the coefficient for $\alpha$ between 0 and 1.5. In the case of Zipf, the distribution of items becomes less and less skewed with the decrease of $\alpha$, reaching a completely uniform distribution at $\alpha = 0$. We expect real-world distributions of requests to be somewhere in the above range. We fixed all other parameters—the cache warmness at 10%, the overlap rate at 50%, the number of SDPs at 5—and varied only $\alpha$.

Figure 4.3(d) shows the hit rate as a function of the $\alpha$ for precise recycling, SAAM and CSAR. It suggests that the hit rate increases along with the $\alpha$ in all three cases. This is expected because requests repeat more often when alpha increases. When alpha was 1.5, all recycling schemes achieved a nearly 100% hit rate. It is also clear that the hit rate improvement due to cooperation only was reduced with the increase of $\alpha$. The reason appears to be two-fold. First, with requests following Zipf distribution, the hit rate in the local cache of each SDP was already high, so that there was less room for improvement through cooperation. Second, unpopular requests had a low probability to be cached by any SDP. Therefore, the requests that could not be resolved locally were unlikely to be resolved by other SDPs either.

**Summary:** The simulation results suggest that combining approximate recycling and cooperation can help SDPs to achieve high hit rates, even when the cache warmness is low. This improvement in hit rate increases with SDPs' resource overlap rate and the number of cooperating SDPs. We also demonstrate that when the distribution of requests is less skewed, the improvement in hit rate is more significant.

### 4.2.2 Prototype-based evaluation

This section describes the design of our prototype and the results of our experiments. The prototype system consisted of the implementations of PEP, SDP, DS, PDP, and a test driver, all of which communicated with each other using Java Remote Method Invocation (RMI). Each PEP received randomly generated requests from the test driver and called its local SDP for authorizations. Upon an authorization request from its PEP, each SDP attempted to resolve this request either sequentially or concurrently. Each SDP maintained a dynamic pool of worker threads that concurrently queried other SDPs. The DS used a customized hash map that supported assigning multiple values (SDP addresses) to a single key (subject/object).

We implemented the PAP and the PCM according to the design described in Section 4.1.6. To simplify the prototype, the two components were process-collocated with the PDP. Additionally, we implemented the selective-flush approach for propagating policy changes. To support response verification, we generated a 1024-bit RSA key pair for the PDP. Each SDP had a copy of the PDP's public key. After the PDP generated a primary response, it signed the response by computing a SHA1 digest of the response and signing the digest with its private key. In the following, we present and discuss the results of evaluating the performance of CSAR in terms of response time, the impact of policy changes on hit rate, and the integration of CSAR with a real application.

**Evaluating response time**

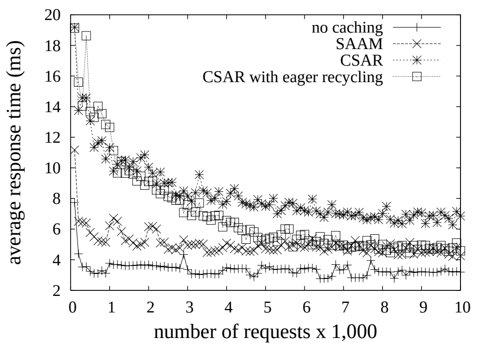
First we compared the client-perceived response time of CSAR with that of the other two authorization schemes: without caching and SAAM (without cooperation). We studied three variations of CSAR: sequential authorization, concurrent authorization and eager recycling. We also evaluated the impact of response verification on response time in the case of sequential authorization. We ran experiments in the following three scenarios, which varied in terms of the network latency among SDPs, and between SDPs and the PDP:

(a) **LAN-LAN.** SDPs and the PDP were all deployed in the same local area network (LAN), where the round-trip time (RTT) was less then 1ms.

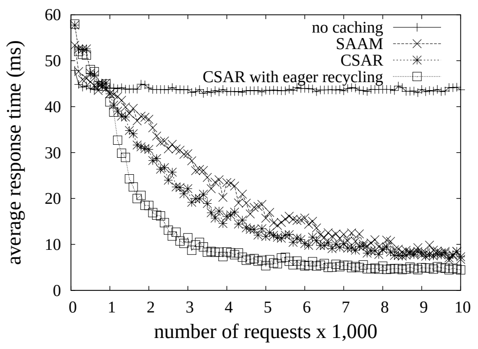
(b) **LAN-WAN.** SDPs were deployed in the same LAN, which was separated from the PDP by a wide area network (WAN). To simulate network delays between SDPs and the PDP, we added a 40ms delay to each authorization request sent to the PDP.

(c) **WAN-WAN.** All SDPs and the PDP were separated from each other by a WAN. Again, we introduced a 40ms delay to simulate delays that possibly occur in both the remote PDP and remote SDPs.

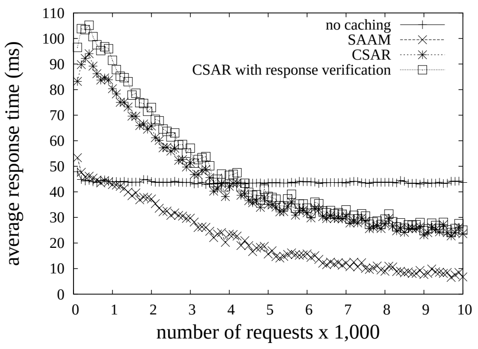
In the experiments we did not intend to test every combination of scenarios and authorization schemes, but to test those most plausibly encountered ones in the real world. For example, concurrent authorization and response verification were only enabled in the WAN-WAN scenario when SDPs were remotely located. Using concurrent authorization in this scenario can help to reduce the high cost of cache misses on remote SDPs due to communication costs. In addition, since the requests in such a scenario are obtained from remote SDPs that might be located in a different administrative domain, response verification is highly desirable.

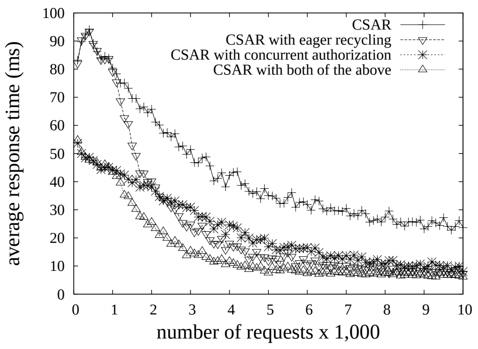


(a) LAN-LAN: SDPs and the PDP are located in the same LAN.



(b) LAN-WAN: SDPs are located in the same LAN while the PDP is located in a WAN.



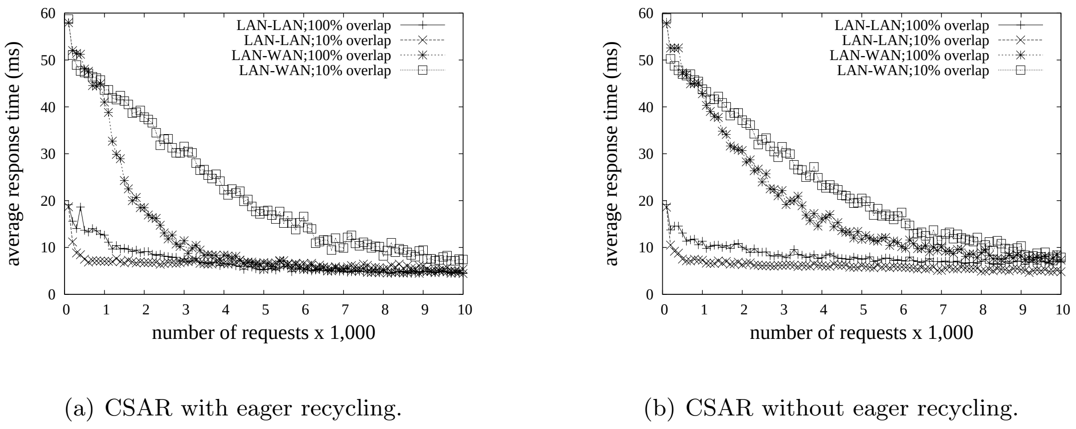


(c) WAN-WAN: SDPs and the PDP are separated by a WAN.

Figure 4.4: Response time as a function of the number of requests observed by SDPs. The requests follow a uniform distribution.

The experimental system consisted of a PDP, a DS, and four PEP processes collocated with their SDPs. Note that although the system contained only one DS instance, this DS simulated an idealized implementation of a distributed DS where each DS had up-to-date global state. This DS instance could be deemed to be local to each SDP because the latency between the DS and the SDPs was less than 1ms and the DS was not overloaded. Each two collocated PEPs and SDPs shared a commodity PC with a 2.8 GHz Intel Pentium 4 processor and 1 GB of RAM. The DS and the PDP ran on one of the two machines, while the test driver ran on the other. The two machines were connected by a 100 Mbps LAN. In all experiments, we made sure that both machines were not overloaded so that they were not the bottlenecks of the system and did not cause additional delays.

At the start of each experiment, the SDP caches were empty. The test driver maintained one thread per PEP, simulating one client per PEP. Each thread sent randomly generated requests to its PEP sequentially. The test driver recorded the response time for each request. After every 100 requests, the test driver calculated the mean response time and used it as an indicator of the



(a) CSAR with eager recycling.

(b) CSAR without eager recycling.

Figure 4.5: Response time comparison between overlap rate of 10% and 100%. The requests follow a uniform distribution.

response time for that period. We ran the experiment when the cache size was 10,000 requests for each SDP at the end of each run, which was one-half the total number of possible requests. Figure 4.4 shows the plotted results for 100% overlap rate. For the sake of better readability, we present the results for the WAN-WAN scenario in two graphs. The following conclusions regarding each authorization scheme can be directly drawn from Figure 4.4:

(i) In the **no-caching** scheme, SDPs were not deployed and PEPs sent authorization requests directly to the PDP. In the LAN-LAN scenario, this scheme achieved best performance since it involved the least number of RMI calls in our implementation. In the other two scenarios, however, all of the average response times were slightly higher than 40ms because all requests had to be resolved by the remote PDP.

(ii) In the **non-cooperative caching (SAAM)** scheme, SDPs were deployed and available only to their own PEPs. When a request was received, each SDP first tried to resolve the request locally and then by the PDP. For the LAN-LAN scenario, this method did not help reduce the latency because in our prototype SDP was implemented as a separate process and each SDP authorization involved an RMI call. In the other two scenarios, response times decreased consistently with the number of requests because more requests were resolved locally. Note that the network distance between SDPs does not affect the results in this and the previous scenario, since either no caching or no cooperation was involved.

(iii) In the **CSAR** scheme, SDPs were deployed and cooperation was enabled. When a request was received, each SDP resolved the request *sequentially*. For the LAN-LAN scenario, the response time was the worst because this scenario involved most RMI calls in our implementation. For the LAN-WAN scenario, using cooperation helped to slightly reduce the response time compared with the SAAM method, because resolving requests by other

SDPs is cheaper than by the remote PDP. However, this improvement continuously decreases, because more and more requests can be resolved locally. For the WAN-WAN scenario, using CSAR was worse than using just SAAM due to the high cost of cache misses on remote SDPs.

(iv) In **CSAR with the response verification** scheme, each response returned from remote SDPs was verified. Figure 4.4(c) shows that the impact of response verification on response time was small: response time increased by less than 5ms on average. When the local cache increased, this overhead became smaller since more requests could be resolved by the local SDP; thus, less verification was involved. Note that the time for response verification was independent of the testing scenario, which means that the 5ms verification overhead applied to the other two scenarios. This is why we did not show verification time in the graphs for the other scenarios.

(v) In **CSAR with the eager recycling** scheme, the primary responses from the evidence lists of secondary responses were incorporated into each SDP's local cache. As expected, eager recycling helped to reduce the response time in all three scenarios, and the effect was especially significant when the PDP or SDPs were remote, since more requests can quickly be resolved locally. The maximum observed improvement in response time over SAAM was by a factor of two. The results also demonstrate that the response time was reduced only after some time. This is because the evidence lists became useful for eager recycling only after the remote SDPs have cached a number of requests.

(vi) In **CSAR with the concurrent authorization** scheme, each SDP resolved the requests *concurrently*. Figure 4.4(c) demonstrates that the response time was significantly reduced in the beginning and decreased consistently. The drawback of concurrent authorization, however, is that it increases the overhead traffic and causes extra load on each SDP and the PDP. It could be a subject of future research to study and try to reduce this overhead.

(vii) In **CSAR with both eager recycling and concurrent authorization** scheme, both eager recycling and concurrent authorization were enabled. Figure 4.4(c) shows that this method achieved the best performance among those tested.

The above conclusions were drawn for 100% overlap rate. For comparison, we also ran the experiments using 10% overlap rate in LAN-LAN and LAN-WAN scenarios. Figure 4.5 compares the response times for the CSAR with and without eager recycling. In the LAN-WAN scenario, the small overlap rate led to increased response time for both schemes because more requests had to resort to the PDP, and the eager recycling scheme experienced more increases. On the other hand, in the LAN-LAN scenario, the response time was reduced in the beginning with the small overlap rate due to the reduced number of RMI calls, since the SDP sent most requests to the PDP directly rather than first to other SDPs which could not help.

**Summary:** The above results demonstrate that although using CSAR with sequential authorization may generate higher response times, adding eager recycling and/or concurrent authorization helps to reduce the response time. Eager recycling is responsible for the effective increase of cache warmness, while concurrent authorization enables SDPs to use the fastest authorization path in the system.

### Evaluating the effects of policy changes

We also used the prototype to study CSAR's behavior in the presence of policy changes. As we show in the previous chapter (Section 3.2.1), continual policy changes at a constant rate result in a reduced hit rate. We expected that using cooperation between SDPs would compensate for the decrease in hit rate; we wanted to understand by how much.

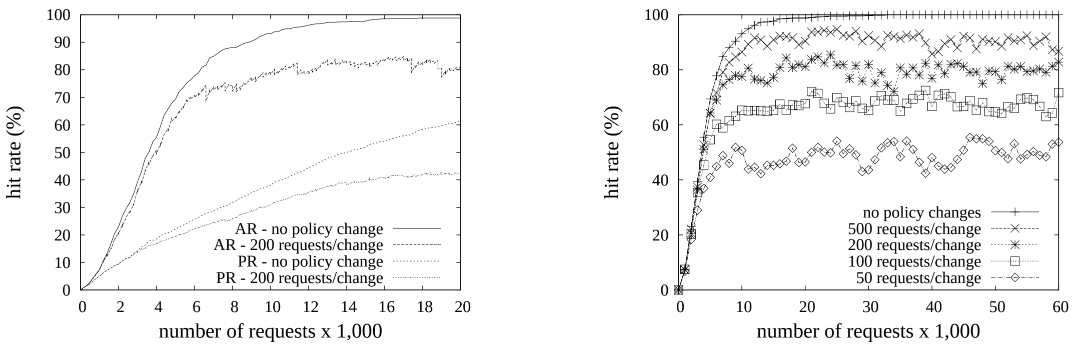
In all our experiments for policy changes, the overlap rate between SDPs was 100% and the requests were randomly generated. The test driver maintained a separate thread responsible for firing a random policy change and sending the policy change message to the PCM at pre-defined intervals, e.g., after every 100 requests. To measure the hit rate at run-time, we employed a method similar to the one used during the simulation experiments. Each request sent by the test driver was associated with one of two modes: *warming* and *testing*, used for warming the SDP caches or testing the cumulative hit rate respectively. Each experiment switched from the warming mode to the testing mode when a policy change message was received. After measuring the hit rate right before and after each policy change, the experiment switched back to the warming mode.

Section 3.2.1 presents $SAAM_{RBAC}$ policy change results using one SDP. Similarly, we first studied $SAAM_{BLP}$ policy change results with one SDP. In particular, we studied how the hit rate was affected by an individual policy change, i.e., the change of the security label for a single subject or object. As with $SAAM_{RBAC}$, we also expected that $SAAM_{BLP}$ inference algorithms were sufficiently robust so that an individual change would result in only minor degradation of the hit rate.

Figure 4.6 demonstrates that $SAAM_{BLP}$ resulted in similar policy change results as $SAAM_{RBAC}$ (Figure 3.8). We summarize the results as follows:

- Figure 4.6(a) shows how the hit rate drops with every policy change for both approximate recycling (the top two curves) and precise recycling. In the experiment, the test driver sent 20,000 requests in total. A randomly generated policy change message was sent to the PDP every 200 requests. Figure 4.6(a) indicates that the hit-rate drops are small for both approximate recycling and precise recycling. For approximate recycling, the largest single hit-rate drop was 5%, and most of the other drops were around 1%.

- Figure 4.6(a) also indicates that the curve for the approximate recycling with policy change is more ragged than it is for precise recycling. This result suggests, not surprisingly, that approximate recycling is more sensitive to policy changes. The reason is that approximate



(a) Hit-rate drops with every policy change for both approximate recycling (AR) and precise recycling (PR).

(b) Hit rate as a function of number of requests at various frequencies of policy change.

Figure 4.6: The impact of policy changes on hit rate with a single SDP. The requests follow a uniform popularity distribution.

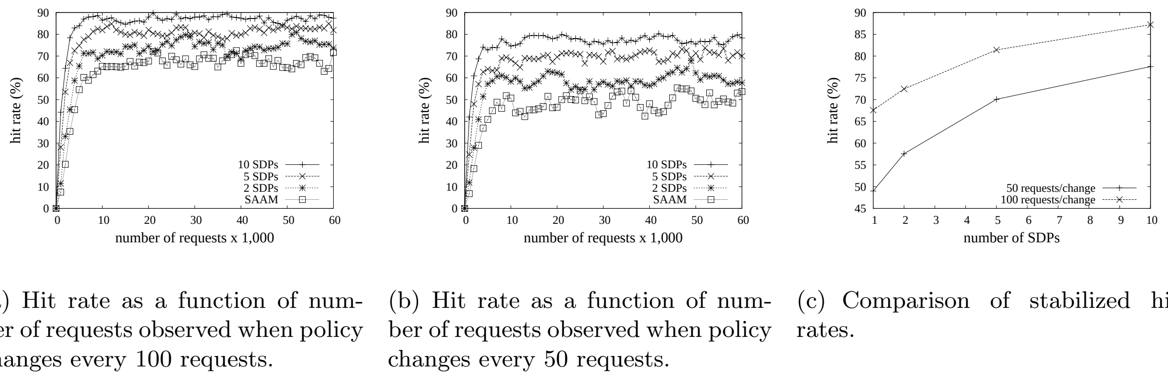
recycling employs an inference algorithm based on a directed acyclic graph. A policy change could partition the graph or just increase its diameter, resulting in a greater reduction in the hit rate.

- Figure 4.6(b) shows that the hit rate stabilized after a number of requests. In the experiment, we used a larger number of requests (60,000), and measured the hit rate after every 1,000 requests. We varied the frequency of policy changes from 50 to 500 requests per change. As expected, the more frequent the policy changes were, the lower the stabilized hit rates were, since the responses were removed from the SDP caches more frequently.

We then studied how the hit rate under continuous policy changes could benefit from cooperation. In these experiments, we varied the number of SDPs from 1 to 10. Figure 4.7(a) and Figure 4.7(b) show hit rates versus the number of requests observed when the policy changed every 50 and 100 requests. Figure 4.7(c) compares the eventual stabilized hit rate for the two frequencies of policy changes. As we expected, cooperation between SDPs improved the hit rate.

Note that when the number of SDPs increased, the curves after the knee became smoother. This trend was a direct result of the impact of cooperation on the hit rate: cooperation between SDPs compensates for the hit-rate drops caused by the policy changes at each SDP.

**Summary:** Our results show that the impact of a single policy change on the hit rate is small, while the cumulative impact can be large. Constant policy changes finally lead to a stabilized hit rate, which depends on the frequency of the policy change. In any case, cooperation helps to reduce this impact.



(a) Hit rate as a function of number of requests observed when policy changes every 100 requests.

(b) Hit rate as a function of number of requests observed when policy changes every 50 requests.

(c) Comparison of stabilized hit rates.

Figure 4.7: The impact of SDP cooperation on hit rate when policy changes. The requests follow a uniform popularity distribution. The overlap rate between SDPs is 100%.

### Integration with TPC-W

In this section, we describe the work on integrating CSAR with TPC-W [TPC02], an industry-standard e-commerce benchmark application that models an online bookstore such as Amazon.com. Our primary goal was to understand the complexity of integrating CSAR with real applications, and the secondary goal was to study the impact of policy enforcement on application performance.

A TPC-W deployment consisted of a front-end application server and a database server. We used Apache Jakarta Tomcat 5.5.4 as the application server and MySQL 5.0 as the database server. The Java code run by the application server to generate the web pages and interface with the database was derived from the code freely available from the University of Wisconsin PHARM project [Pha03], whose code implements both the servlets for the business logic and a Java remote browser emulator (RBE) for driving the experiment.

Figure 4.8 shows the system architecture that integrates the CSAR prototype with the TPC-W on a single application server. To enforce the access control policy, we added the PEP, the SDP, the PDP and a policy file to the original TPC-W architecture. The PEP was implemented as a servlet filter that contained about 100 lines of code. The PEP dynamically intercepted application requests and used them to generate authorization requests which only included the information about the subject, object and access right. We assured that every application request was intercepted by the PEP, then the authorization request was sent to its SDP for decision. The PDP and the SDP were reused from our prototype implementation with only minor changes to the system configuration file.

We modeled three user roles (or security labels) in the access control policy: visitor, customer and administrator. Each role could access a number of application objects. For example, the visitor could only browse books; the customer could not only browse books but also buy books; the administrator could only access the administration pages for managing books.

The experiment used 10 emulated browsers (EBs) to simulate 10 concurrent users and each

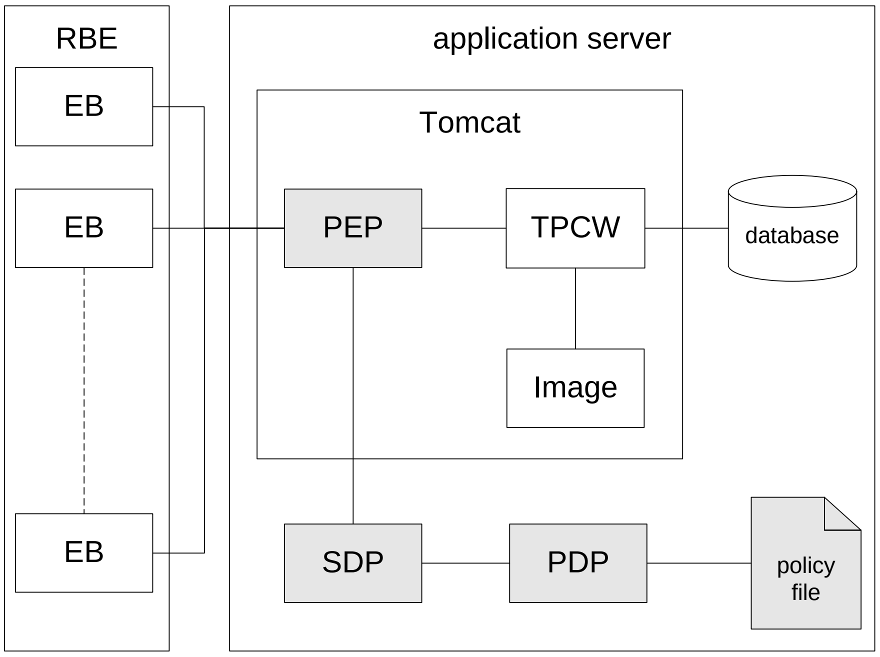
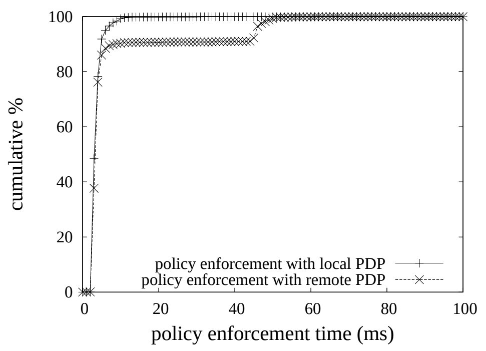


Figure 4.8: Adding CSAR-based policy enforcement to TPC-W.

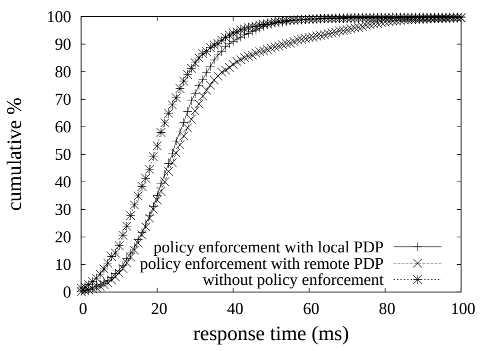
EB followed a browsing mix behavior defined by the TPC-W specification. We used two PCs in the experiments: one was used for running EBs while the other was used as both application server and database server. Each experiment lasted 30 minutes. We simulated both remote and local PDPs as defined in previous sections, and studied the increase in response times introduced by policy enforcement.

First, we were interested to understand how much time was used for policy enforcement in our setup. We measure the policy enforcement time as the time between the event that the PEP receives a request from the client and the event that the PEP receives a response from the SDP. Figure 4.9(a) shows the cumulative distribution of policy enforcement time. For the local PDP, unsurprisingly, almost all policy enforcement times were small, i.e., less than 10ms. For the remote PDP, 10% of the response times were between 40ms and 50ms, which means that these requests were resolved by remote PDPs. The fact that the 90% of enforcement times were less than 10ms even when the PDP was remote, implies a high hit rate on the SDP, which we believe was due to the access pattern and object space of the TPC-W application. In this experiment, each emulated session lasted 15 minutes and client thinking time was 7 seconds, as specified by the TPC-W standard. This means that in each session each user accessed about 128 pages (a.k.a. objects). On the other hand, the TPC-W application consists of only 14 unique pages. Combining these two facts, each page would have been accessed around 9 times in an average session. This behavior unavoidably led to a high cache hit rate.

Second, we were interested to understand how the policy enforcement affected the overall TPC-W response time. Figure 4.9(b) plots the cumulative distribution of overall TPC-W



(a) Cumulative distribution of response time



(b) Cumulative distribution of policy enforcement time

Figure 4.9: The impact of policy enforcement on response time.

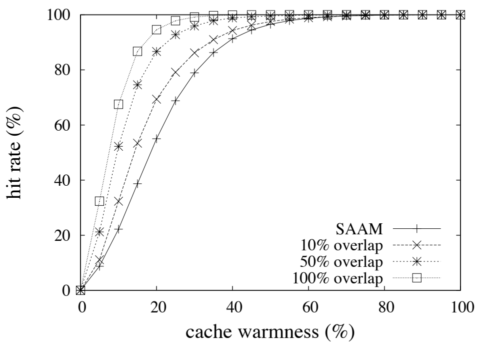
response times in three scenarios: without policy enforcement, and with policy enforcement by either a local PDP or remote PDP. Compared with the scenario without policy enforcement, the following observations can be made: (1) when the PDP was remote, the average response time increased by 20% since 10% policy enforcement time was between 0 and 10ms, and another 10% was between 40ms and 50ms; (2) when the PDP was local, its curve before 20ms matched the curve of remote PDPs, while its curve after 50ms matched the curve of no policy enforcement. The reason was that most policy enforcement times were less than 10ms.

**Summary:** Our experience shows that the prototype can be easily integrated with the Java-based TPC-W application. The PEP can be simply implemented as a Java servlet filter and the other CSAR components require few changes. The results show that the overhead of policy enforcement is highly dependent on the application's requests pattern and object space. In the case of TPC-W, a single SDP achieves a 90% hit rate, thus the impact on the overall application performance is small. Based on this result, we decided not to run the experiment with cooperation, as the improvement space for cooperation was only 10%.

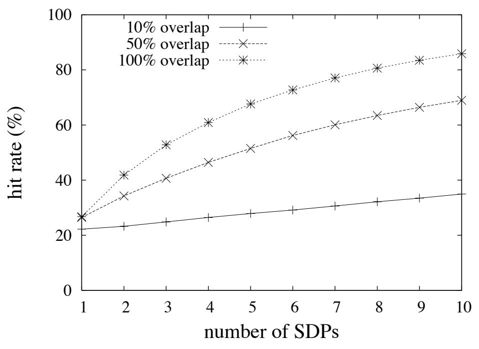
## 4.3 Evaluation of CSAR$_{\mathbf{RBAC}}$

In this section, we briefly describe the evaluation results for CSAR$_{\text{RBAC}}$, when the underlying access control model is RBAC.

We used the same reference RBAC policy used in the previous chapter. The policy contained 100 users, 3,000 permissions, and 50 roles. Thus the overall size of the request space and the warming set was 300,000. The testing set contained 20,000 unique requests which were randomly selected from the request space. Each assigned role was randomly selected from $R$. The probability of a given user being assigned to a given role was 0.1. Hence the number of roles assigned to a user was binomially distributed with mean 5 and variance 4.5, and the number



(a) Hit rate as a function of cache warmness for 5 SDPs compared to 1 SDP (i.e., SAAM).



(b) Hit rate as a function of number of SDPs at cache warmness of 10%.

Figure 4.10: The impact of various parameters on hit rate

of users to which a role was assigned was binomially distributed with mean 10 and variance 9. Similarly, the probability of a given permission being assigned to a given role was 0.04.

Figure 4.10 shows the simulation results on how the hit rate was affected by cache warmness, number of SDPs, and the overlap rate. Comparing them with the results depicted in Figure 4.3(a) and 4.3(b), we observed similar patterns. This is because the hit rates of $\text{SAAM}_{\text{BLP}}$ recycling algorithms and $\text{SAAM}_{\text{RBAC}}$ recycling algorithm exhibit similar patterns and the cooperation technique is independent from the underlying recycling algorithm.

Furthermore, although we only show the $\text{CSAR}_{\text{RBAC}}$ hit rate results, we expect similar results on performance as $\text{CSAR}_{\text{BLP}}$. Response time is significantly determined by the SDP hit rate: high hit rate results more responses resolved locally, thus reducing the response time. As Figure 4.10 shows the similar pattern in the hit rate increase as $\text{CSAR}_{\text{BLP}}$, we also expect to see similar pattern in the response time decrease.

## 4.4 Summary

This chapter defines CSAR system requirements, and presents a detailed design that meets these requirements. Built on SAAM and the concept of SDP, the CSAR system explores the potential cooperation between SDPs through a discovery service. We have introduced a response verification mechanism that does not require cooperating SDPs to trust each other. Cache consistency is managed by dividing all of the policy changes into three categories and employing efficient consistency techniques for each category.

A decision on whether to deploy CSAR depends on a full cost-benefit analysis informed by application- and business-specific factors, for example, the precise characteristics of the application workload and deployment environment, an evaluation of the impact of system failures on business continuity, and an evaluation of the complexity associated costs of the access control system. To inform this analysis, we have evaluated CSAR's application-independent benefits:

higher system availability by masking network and PDP failures through caching, lower response time for the access control subsystem, and increased scalability by reducing the PDP load; and costs: computational and generated traffic overhead.

The results of our CSAR evaluation suggest that even with small caches (or low cache warmness), our cooperative authorization solution can offer significant benefits. Specifically, by recycling secondary authorizations between SDPs, the hit rate can reach 70% even when only 10% of all possible authorization decisions are cached at each SDP (assuming that there is no bogus SDP). This high hit rate results in more requests being resolved by the local and cooperating SDPs, thus increasing availability of the authorization infrastructure and reducing the load on the authorization server. In addition, depending on the deployment scenario, request processing time is reduced by up to a factor of two, compared with solutions that do not cooperate.

Chapter 5

# Authorization Using the Publish-Subscribe Architecture

In the previous two chapters, we introduce authorization recycling approaches to addressing the problems of fragility and poor performance caused by the tight coupling between policy enforcement points (PEPs) and policy decision points (PDPs). A secondary decision point (SDP), which is collocated with the PEP, can resolve authorization requests not only by reusing cached authorizations but also by computing approximate authorizations from cached authorizations. In addition, each SDP may further share their capability of resolving authorization requests with other SDPs by cooperation. The communication between SDPs and PDPs, however, is still based on the point-to-point architecture. When the cache hit rate is low, e.g., due to the small number of cached responses or the frequent policy changes, the authorization system is still fragile.

In this chapter, we examine the use of a publish-subscribe architecture for delivering authorization requests and responses between applications and authorization servers, which further enables applications to reduce their dependence on authorization servers. Our analysis shows that using the publish-subscribe architecture helps achieve partial replication and reduces system administration overhead. We also evaluate the system performance, and study the dependence of performance on various factors.

Section 5.1 describes the background on the publish-subscribe architecture. Unlike in a point-to-point architecture, where PEPs are configured to send their requests to specific PDPs, a publish-subscribe architecture enables PEPs to send their requests without knowing which PDP will receive them. Similarly, the PDPs show interest in requests without knowing which PEPs generate them. Therefore, the coupling between PEPs and PDPs is further reduced.

Section 5.2 describes a general design based on the publish-subscribe architecture for the system based on the request-response model and secondary and approximate authorization model (SAAM). Our design is independent of the underlying access control policies. Our analysis shows that using publish-subscribe helps improve system availability and reduce system administration overhead. Our study also shows that using pub-sub with SAAM helps achieve cooperative authorization recycling as well as speculative authorizations.

Section 5.3 presents the evaluation results. We study the availability analytically and the results confirm that the publish-subscribe architecture improves the availability of the authorization infrastructure. We also develop a prototype and use it to study the performance under different design schemes.

Finally, we summarize this chapter in Section 5.4.

## 5.1 The publish-subscribe architecture

Pub-sub is an asynchronous messaging paradigm that has been widely studied and applied to distributed applications, enabling loosely coupled interaction between entities whose location and behavior may vary throughout the lifetime of the system [EFGK03]. Generally, publishers that send messages "publish" them as events, while subscribers that wish to receive certain events "subscribe" to those events. An event notification service (ENS) mediates the communications between publishers and subscribers, thereby fully decoupling publishers from subscribers. An entity may be a publisher as well as a subscriber, thereby being able to send messages and receive events within the system.

Due to its flexibility in decoupling publishers and subscribers, pub-sub has been used to support a wide range of applications, such as Internet games [BRS02], mobile agents [PLZ03], user and software monitoring [HR98], mobile systems [CJ02], groupware [DB92], collaborative software engineering [SNvdH03], and WWW updates [RPS06], among others.

### 5.1.1 The event notification service

An ENS is the core component of any pub-sub system. The ENS is responsible for: (1) receiving events from publishers, (2) receiving subscriptions from event subscribers, and (3) matching each event to subscriptions and routing the event, in the form of notifications, to the interested subscribers.

The ENS is only logically centralized but can have a distributed implementation to achieve scalability and fault-tolerance. It may consist of a set of dedicated brokers forming a network (e.g., Siena [CRW01] and Scribe [RKCD01]). Its clients (i.e., publishers or subscribers) connect to an arbitrary broker to subscribe or publish. This broker is called *border broker* and hides the distributed nature of the ENS. Brokers are responsible for subscription routing and event forwarding. In order to deliver an event to the interested clients, subscriptions need to be routed and stored by all brokers (or by a subset of brokers, using various optimization techniques, e.g., [CRW01]). Subscriptions are stored in routing tables, which are used for event forwarding. A broker performs forwarding by evaluating the predicates of the stored subscriptions over the name-value pairs of the incoming event. This process is referred to as *matching.* The result of matching is a subset of the broker's neighbors (local destinations) that should have the event forwarded to them. Each local destination corresponds to an output path that associates a set of subscriptions. By performing a matching operation at every broker, the event will eventually be delivered to the subscriber that issued the relevant subscription.

The brokers in a distributed ENS implementation can be organized into topologies of two types. One is the tree topology, where brokers are organized as a tree or an acyclic graph (e.g., Siena [CRW01]). The resilience to node failure of broker overlays can be achieved by employing self-replication techniques like those introduced by Jaeger et al. [JPMH07] and Baldoni et al. [BBQV07]. Another is the peer-to-peer topology (P2P), where nodes are organized as structured overlay networks (e.g., Hermes [PB02]). These systems, also known as distributed hash tables (DHTs), usually partition content space and map each partition to a broker using a hash function. Subscriptions and publications are routed to their corresponding brokers using DHT-based routing techniques such as Chord [SMLN$^+$03] and Tapestry [ZHS$^+$04]. Fault tolerance is achieved inherently by the P2P overlay.

### 5.1.2 Subscription schemes

Subscribers are usually interested in particular, but not in all, events. The different ways of specifying the events of interest have led to two subscription schemes. The first is called *topic-based* scheme, and has been implemented by many industrial strength solutions (e.g., TIBCO Message Service [TIB99] and Java Message Service [Mic01]). In this scheme, each message belongs to one of a fixed set of topics. A subscription targets a topic, and the subscriber receives all events that are associated with that topic. Topic-based systems are similar to the earlier group communication and event-notification systems.

The second is called *content-based* scheme and has attracted notable interest (Gryphon [BCM$^+$99], Siena [CRW00], Elvin [SAB$^+$00], and Jedi [CDNF01].) This scheme is not constrained to the notion that each message must belong to a particular topic. Instead, the message delivery decisions are based on a predicate issued by the subscriber. The advantage of a content-based scheme is flexibility: it provides the subscriber with the ability to specify just the information it needs without having to learn a set of topic names and their content before subscribing. In this chapter, we develop our authorization system based on the content-based scheme.

In content-based schemes, each subscription is usually modeled as a set of predicates. Each predicate consists of attribute names and attribute constraints. An attribute constraint consists of an operator and an attribute value. Each event is modeled as a set of attribute name-value pairs. A sample subscription can be represented as: $\{income > 5000; age > 18\}$ and a matching event has the following form: $\{income = 6000; age = 20\}$.

## 5.2 System design

This section presents the design of an authorization system based on a pub-sub architecture for the request-response authorization model. We first describe the system requirements. We then present the system architecture and the expected benefits of our design. Finally, we discuss other important design issues, such as subscription mechanisms, security consideration, and system consistency.

### 5.2.1 Design requirements

Our design aims to use a pub-sub channel to replace the point-to-point communication between PEPs and PDPs in existing authorization systems. We consider the following requirements in our design:

- **Generic design.** The system design should be generic so that it does not depend on any specific ENS technology or the underlying access control policies.
- **Low overhead.** Using an ENS to mediate the communication between PEPs and PDPs adds to the communication latency, which may, in turn, degrade application performance. In particular, the ENS needs propagate subscriptions, locate the potential subscribers for each event, and route the event to them. Therefore, it is important to reduce the overhead introduced by ENS operations. In particular, we discuss various subscription schemes that can be used to reduce the ENS overhead in Section 5.2.3.
- **Ability to deal with adversaries.** Using an ENS also adds additional threats to the system. For instance, as each PEP enforces responses that are possibly offered by unknown PDPs and the communications are mediated by a possibly untrusted ENS, the PEP should expect to receive responses that have been modified by an adversary. Therefore, the PEP may have to verify the validity of each response by tracing it back to a trusted PDP. We discuss security considerations in Section 5.2.4.
- **Consistency.** Since multiple PDPs may participate in resolving a request, different PDPs may return inconsistent responses, e.g., due to the delayed policy updates on some PDPs. In this case, the PEP should have a strategy to deal with the inconsistency between responses. We discuss consistency considerations in Section 5.2.5.

### 5.2.2 System architecture

Our architecture of the pub-sub authorization system is illustrated in Figure 5.1. The system consists of multiple PEPs, PDPs and a logically-centralized event notification service (ENS). Both PEPs and PDPs can be subscribers and publishers. In particular, a PEP publishes authorization requests it generates from application requests, while subscribing to the responses for these requests. In contrast, a PDP subscribes to the authorization requests that it can resolve, while publishing the responses. The ENS is responsible for delivering the requests to the corresponding PDPs (that can resolve these requests) and for delivering the responses to the corresponding PEPs (that wait for these responses).

To meet our first requirement on general system design, we view the ENS as a black box that provides three basic operations: *subscribe*, *unsubscribe* and *publish*. In this chapter, we focus on answering the question of how the PEPs and PDPs can use an ENS with these limited operations. We assume that the underlying ENS is scalable and robust so that it will not

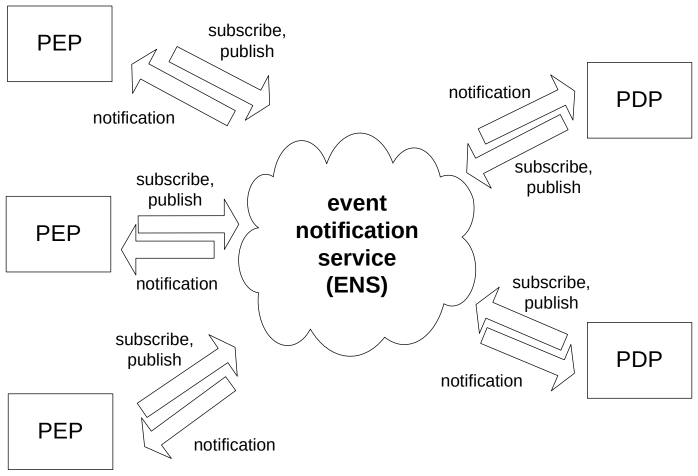


Figure 5.1: Publish-subscribe architecture for delivering authorization requests and responses.

become the system performance bottleneck or single point of failure. Past work focussed on implementing a scalable and robust ENS has been discussed in Section 5.1.1.

To prevent the PDP from missing any request sent by PEPs, when the system bootstraps, each PDP subscribes to all the requests that it can resolve, e.g., by making a *subscribe*() call to the ENS for each resolvable request. Therefore, PDP subscriptions can be viewed as a one-time process. Section 5.2.3 further discusses different PDP subscription strategies that can be used to reduce the number of posted subscriptions.

Now consider that a PEP intercepts an application request from the user and generates an authorization request. Figure 5.2 shows the sequence diagram of how an authorization request is resolved in our system. In particular, the following steps are included (also shown in the figure):

(i) The PEP first subscribes to the response to that request by making a *subscribe*() call to the ENS. The purpose is for this PEP to receive the corresponding response in the future. Section 5.2.3 provides a discussion on the different subscription schemes that the PEP can use in this step. Using some schemes, e.g., the session-based scheme, this step is not necessary for every request, thus reducing the ENS overhead.

(ii) The PEP then sends out the request by making a *publish*() call to the ENS and expects that the ENS will deliver a response back within a certain time frame.

(iii) After the ENS receives the request, it forwards it to those PDPs that can resolve it by matching the request to the subscriptions posted by PDPs.

(iv) After receiving the request from the ENS, each PDP computes a response that either allows or denies the request.

(v) Each PDP sends the response back by making a *publish*() call to the ENS.

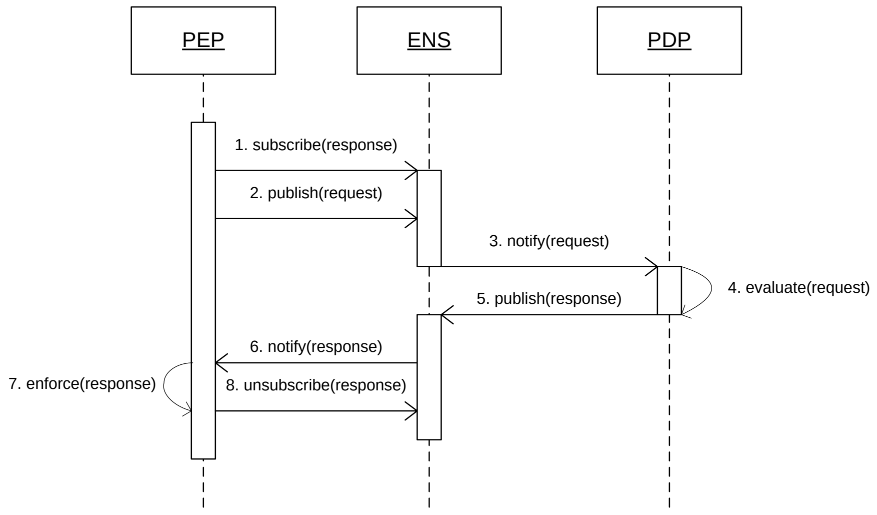


Figure 5.2: Basic sequence diagram in resolving an authorization request.

(vi) The ENS forwards the response to the PEP that is waiting for that response by matching the response over the subscriptions posted by the PEP.

(vii) The PEP enforces the authorization decision in the returned response, either allowing user's request to proceed to the requested resource or denying the request. Since a request may be delivered to multiple PDPs and each PDP may compute and return a response, the PEP may have to select one response for enforcement. In Section 5.2.5, we discuss the strategy that the PEP can use to maintain a monotonic-read consistency when different PDPs return inconsistent responses.

(viii) The PEP unsubscribes from the response to that request by making a *unsubscribe*() call to the ENS. This reduces the number of subscriptions maintained by the ENS. Similar to step 1, this step may be not required for each request in some schemes that we propose in Section 5.2.3.

Although the above eight-step process appears heavy and incurs additional overheads compared to the request-response communication, we expect that it has the benefits of supporting a number of desirable features. Below we present three expected benefits: increased availability, reduced management overhead and improved software development process. Section 5.2.6 presents additional benefits when integrating the pub-sub with SDPs. Through integration, the pub-sub provides a unified framework for cooperative recycling of authorization decisions between SDPs and PDPs. In addition, the integration facilitates speculative computing of authorizations. Both will help improve system performance and availability.

- **Increased availability.** In the point-to-point architecture, a PEP is generally configured to send each request to only one PDP. In existing enterprise application systems, however, the same resources often reside at multiple locations, on multiple machines, and within a variety of applications [MT00, HAB$^+$05]. Consequently, multiple PDPs may have to be set up to resolve access requests for the overlapping sets of resources. Our solution exploits this situation: using the pub-sub architecture, multiple PDPs may show their interest on the overlapping sets of requests. Therefore, a request from the PEP may reach all the PDPs that are able to resolve it. Even though some of the PDPs may fail, the chances that at least one will provide a response on time are higher. In other words, our design achieves a certain level of PDP collaboration where a collection of PDPs appear as a single large more reliable PDP "cloud" to PEPs.

- **Reduced management overhead.** We expect that decoupling PEPs and PDPs using the ENS will reduce the human costs of operating and administering authorization infrastructures. Consider the previous example of a failed PDP and the PEPs that depend on that PDP. After the PDP is brought back and possibly relocated, as long as it subscribes again to the ENS, none of the PEPs need to be re-configured. The ENS ensures that the request will be delivered to the relocated PDP.

- **Improved software development process.** Using an event-driven, standards-based pub-sub channel to provide a comprehensive communication framework between PEPs and PDPs also improves the software development cycle. In particular, the integration of PEPs and PDPs is faster and less expensive using a pub-sub channel than using a point-to-point architecture. Research by Gartner Group [Gar02] shows that the use of a common integrated information infrastructure, such as a pub-sub channel, can reduce the number of hours by between 25% and 43% (to build interfaces between applications), depending on the complexity of the interface that is being built.

### 5.2.3 Subscription/unsubscription schemes

Our architecture uses a two-way pub-sub communication channel to emulate synchronous point-to-point communication. For this purpose, the PEP needs to publish a request to the ENS once it receives that request and the PDP needs to publish a response back once it computes that response. In other words, the PEP/PDP needs to publish immediately whatever they receive or compute.

On the other side, we have more flexibility in designing subscription/unsubscription (or "(un)subscription" for short) schemes. An (un)subscription scheme determines how a client issues (un)subscriptions to the ENS. Our preliminary experiments identified two factors that have significant impact on the system performance. In this section, we discuss these two factors and different (un)subscription schemes that affect them.

### Performance factors

One factor is the frequency of (un)subscription calls posted to the ENS. As we mentioned in Section 5.1.1, in a typical ENS implementation [RRH06], subscription is the process that sets the routing path for events and builds the routing table at each ENS node, while unsubscription is the process that removes obsolete subscriptions from the routing tables. Furthermore, to prevent inconsistency in the routing tables, (un)subscription calls are commonly serialized using mutual exclusion, which makes frequent (un)subscription calls prohibitively expensive. Therefore, it is important to reduce the frequency of subscriptions/unsubscriptions.

The other performance factor is the number of outstanding subscriptions stored at the ENS, which affects the ENS performance in matching an event to a set of interested subscriptions. Previous research (e.g., Aguilera et al. [ASS$^+$99]) on matching algorithms suggests that a large number of outstanding subscriptions posts a challenge to the ENS: usually the algorithm that the ENS uses to find a matching subscription for an event has linear or sub-linear time complexity with respect to the number of outstanding subscriptions [ASS$^+$99]. Therefore, it is desirable to reduce the number of outstanding subscriptions.

As a summary, we have two performance-related goals in designing (un)subscription schemes: (1) reducing the frequency of (un)subscription calls, and/or (2) reducing the number of outstanding subscriptions. As the ENS emulates a two-way communication channel in our design, below we consider PEP and PDP subscriptions separately.

### PEP schemes

A PEP subscribes to responses, indicating that it is waiting for the decisions to some of its authorization requests. A PEP can subscribe to each individual response or to a general attribute (e.g., the subject) that matches a number of responses (e.g., for the same subject). In the following, we describe four PEP (un)subscription schemes which have different impact on the performance factors, thus affecting the system performance.

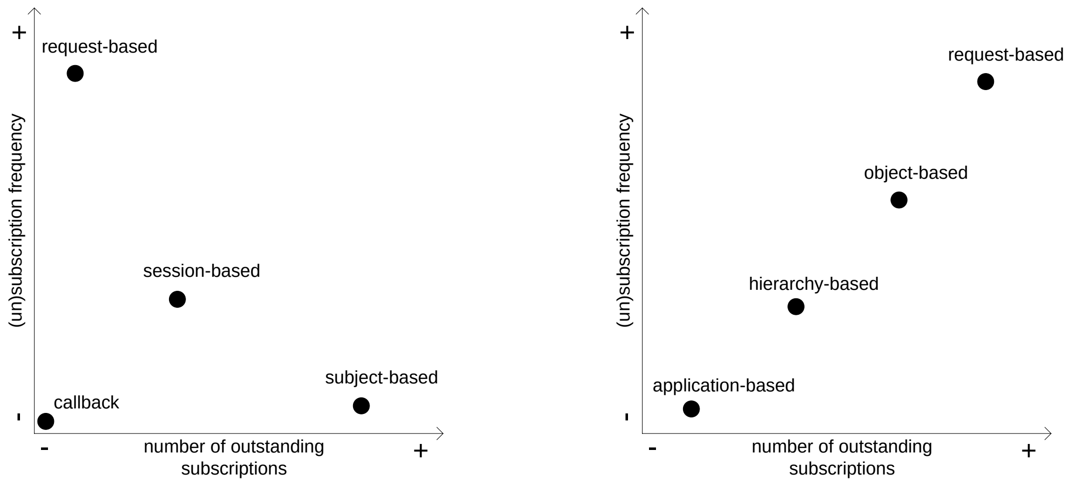
**Request-based scheme.** In this scheme, the PEP submits a new subscription to the ENS for each request it generates, by passing the request as the parameter of the *subscribe*() operation. Once the PEP receives the response, it then unsubscribes from that response. The benefit of this approach lies in a low number of outstanding subscriptions, which equals to the number of concurrent requests launched by the PEP. However, this scheme can also lead to an extremely high (un)subscription frequency as each request will result in an (un)subscription call. Therefore, it is suitable for applications where the incoming request rate is low.

**Subject-based scheme.** In this scheme, the PEP subscribes to each subject, instead of each request. Particularly, when the PEP detects a request that is issued by a subject never seen before, the PEP makes a *subscription*() call to the ENS by specifying the subject as the parameter. With this scheme, only one subscription call is required for the first request of each subject, as subscribing to a subject enables the PEP to receive the response for any request this subject may make in the future. No unsubscription is required in this scheme. As more and more subjects have interacted with the system, the frequency of subscription calls will gradually become smaller. Eventually, no subscription call will be made because the PEP has subscribed to all the subjects in the system. On the other side, this scheme has the drawback of increasing the number of outstanding subscriptions, which will eventually be equal to the total number of subjects for each PEP. Hence, this scheme is useful when the number of subjects in the system is small.

**Session-based scheme.** In this scheme, the PEP only subscribes to those active subjects that are currently having a *session* with the system. Similar to the previous scheme, the PEP makes a subscription call when a subject starts a new session with the system. Differently, when the subject logs out of the system or its session expires, the PEP unsubscribes from the responses for that subject. Next time when the same subject logs into the system, the PEP subscribes again. Compared with the subject-based subscription scheme, this scheme reduces the number of outstanding subscriptions, which is equal to the number of active subjects or sessions, while increasing the (un)subscription frequency. Compared with the request-based subscription scheme, this scheme reduces the (un)subscription frequency while leading to a larger number of outstanding subscriptions. Therefore, this scheme is especially useful in the following two scenarios: (1) although the number of subjects in the system is large, the number of active subjects at each moment is relatively small, or (2) although subjects issue requests very frequently, only a small portion of these requests is issued as the first request by a new or returned subject.

**Callback scheme.** In this scheme, instead of returning responses through the ENS, the PDP returns them directly to the PEP. Therefore, the PEP does not need to issue subscriptions to receive responses. To preserve the decoupling property of the pub-sub architecture, each authorization request contains the call-back reference for the PEP that publishes it. Each PDP then can make a remote call to return the response without locally storing the PEP address in advance. Although this scheme removes the PEP (un)subscription overhead and the response matching overhead, it posts additional requirements to the infrastructure. For example, special rules have to be configured in the firewall to enable the call from the PDP to reach the PEP. Therefore, this scheme is suitable for those systems, for which performance overhead is not affordable and there is supporting infrastructure available.

Figure 5.3(a) summarizes the trade-offs between these schemes. It shows that the request-based scheme incurs high (un)subscription frequency but has a low number of outstanding subscriptions. Session-based and subject-based schemes help reduce (un)subscription frequency but at the cost of an increased number of outstanding subscriptions. Ideally, the callback scheme



(a) PEP schemes.

(b) PDP schemes.

Figure 5.3: Trade-offs bettween (un)subscription schemes.

can be deployed to fully remove the need for (un)subscription. We report the evaluation results of these schemes in Section 5.3.2.

### PDP schemes

By subscribing, a PDP communicates its ability to resolve certain requests. This ability is determined by the authorization request space the PDP is responsible for. For simplicity, we define a request as a tuple $(s, o, a)$, where $s$ is the subject, $o$ is the object, and $a$ is the access right. Therefore, the request space is determined by the subject space, object space and access right space.

As we mentioned before, PDP subscription is a one-time process: the PDP subscribes when the system boots up. After that, any subsequent (un)subscriptions are due to the change of request space, e.g., new users added to the system. Below we describe four PDP (un)subscription schemes, which have different impacts on the number of outstanding subscriptions and (un)subscription frequency.

**Request-based scheme.** In this scheme, the PDP subscribes to all the request tuples $(s, o, a)$ it can resolve. However, this approach may lead to a large number of outstanding subscriptions registered in the ENS when the request space of each PDP or the number of PDPs is large. In addition, every change to the request space will result in an (un)subscription call to the ENS, which may lead to high (un)subscription frequency.

**Object-based scheme.** In this scheme, instead of subscribing to each possible authorization request, the PDP only subscribes to each possible object for which it is "responsible". This approach reduces the number of outstanding subscriptions by reducing the number of dimensions of the subscription space from three ($s$, $o$ and $a$) to one ($o$), but it may incur unnecessary overhead. For example, a PDP that subscribes to ($o = o_1$) may receive requests issued by subjects $s_1$ and $s_2$, but the PDP can only make a decision for the request by $s_1$ because $s_2$ is not in its subject space. With this scheme, only changes to the object space will result in an (un)subscription call.

**Hierarchy-based scheme.** This scheme exploits the hierarchical structure of the namespace in many distributed systems. For example, the protected object namespace in IBM Tivoli Policy Director [Kar01] is organized in a hierarchical structure. Each object is represented as a string with a syntax, and the structure is similar to absolute URIs but without the scheme, machine, and query components. The slash character ('/') is used to delimit, from left to right, hierarchical substrings of the object's name. In this case, each PDP can subscribe to only the directory string instead of subscribing to each individual object string, which, we expect, will further reduce the number of outstanding subscriptions significantly. Furthermore, any change within the directory will not result in an (un)subscription call.

**Application-based scheme.** This scheme is based on the assumption that each PDP is only responsible for resolving the requests from certain applications. Assuming that each authorization request contains an extra attribute to indicate the application this request belongs to, the PDP only needs to subscribe to those application identities to receive that request. Additionaly, any request space change within the application will not result in an (un)subscription call. However, this practicality of this approach depends on the structure of enterprise application systems.

Figure 5.3(b) illustrates the trade-offs between these schemes. To summarize, by applying specific knowledge about the request space to the subscription scheme, one can reduce the number of outstanding PDP subscriptions as well as (un)subscription frequency at the same time.

### 5.2.4 Security considerations

Decoupling PEPs and PDPs by the ENS introduces additional threats to the system. In this section, we describe our adversary model and discuss options for mitigating threats.

#### Adversary model

In our approach, the PDPs that are set up by system administrators are the ultimate authority for access control decisions. For the purpose of discussion, we refer to them as "legitimate" PDPs. We assume that PEPs trust the decisions made by legitimate PDPs. An adversary can eavesdrop, spoof or replay any network traffic. Thus, any decision made by a legitimate PDP may be modified.

In addition, adversaries can set up their own PDPs. Masquerading as legitimate PDPs, these "malicious" PDPs can listen to the requests coming through the ENS and publish false, spoofed, responses. Malicious PDPs can also attempt denial-of-service (DoS) attacks, e.g., by flooding the ENS—and consequently the PEPs—with replayed responses, which increases the ENS computation overhead or even overloads it or the subscribed PEPs. Therefore, actions of malicious PDPs may worsen system performance or jeopardize its correctness.

We also assume that PEPs set up by system administrators only send legitimate requests to the ENS. Attackers however may setup their own "malicious" PEPs and attempt to spam or flood the pub-sub network with replayed or spurious requests, which may overload the ENS and PDPs, consequently degrading system performance.

In addition to publishing bogus information, both malicious PDPs and PEPs can post spurious subscriptions to the ENS. If the ENS accepts these subscriptions, the matching time will increase and thus the system performance would be further reduced, as can be seen from our experiments in Section 5.3.2.

An adversary can also set up malicious ENS nodes that could sabotage the operation of the ENS in a number of ways. A malicious ENS node may selectively drop messages or inject a large number of spurious events/subscriptions, thus launching a DoS attack. In addition, it can modify the content of messages, allowing unauthorized access.

To summarize, an adversary can (1) compromise system correctness by publishing or injecting in the network links spoofed responses and/or tampering with genuine ones, or (2) degrade system performance by flooding the ENS with spurious or replayed requests and/or responses.

### Mitigating threats

In traditional request-response authorization systems, each request is sent from a PEP to a pre-specified PDP. The communication between the PEP and PDP is usually protected through (mutually) authenticated and encrypted channels, as in the case of the IBM Tivoli Access Manager [BAR+03], which protects with SSL the integrity and authenticity of requests/responses transported between PEPs and PDPs.

In our approach, the communication between PEPs and PDPs is mediated by the ENS. We first assume that the ENS is a trusted broker network located in a single administrative domain or across multiple domains with mutual trust; each ENS broker node is trusted to perform the routing as designed and will not attempt to modify the messages it routes. To achieve the same level of security as the request-response authorization system, one can still employ authenticated and encrypted channels (e.g., SSL) between the ENS nodes and PEPs/PDPs. Using such channels ensures that only authenticated PEPs/PDPs can publish and subscribe using the ENS, thus preventing it from being overloaded by messages sent from malicious PEPs and PDPs. At the same time, legitimate PEPs and PDPs can assume that messages are not modified while in transit.

To maintain the trustworthiness of a distributed broker network, it is important to prevent malicious nodes from joining it. One technique is to use mutual authentication between interacting ENS nodes. To reduce the number of costly authentication handshakes, the ENS node should support persistent or "keep-alive" connections, as for example in the case of IBM WebSphere Message Broker [IBM08].

We next relax the assumption of a trusted ENS: the ENS may span multiple administrative domains of unknown trustworthiness, and some ENS nodes can be malicious. In this case, it is difficult to prevent the DoS attack. One technique to alleviate this negative impact on performance is to allow each ENS node to actively monitor the behavior of its neighbors, and hope that eventually the trusted node is able to detect the "abnormal" behavior of malicious neighbors and blacklist them from future routing, as proposed by Srivatsa and Liu [SL05].

Even with malicious ENS nodes, it is still possible to ensure system correctness, i.e., the authenticity and integrity of requests/responses. One approach is to apply end-to-end protection mechanisms: each response/request is signed and verified at PDPs/PEPs. For example, after a PDP computes a response, it signs the response using its private key and attaches the signature to the message. The PEP that receives the response can then verify the digital signature. The benefit of this approach is that it requires no changes to the ENS. A signature provides two pieces of evidence. First, the response has not been modified by the untrusted ENS since it was signed, thus protecting the response integrity. Second, the response is indeed originated from a trusted PDP, thus protecting its authenticity. Furthermore, a nonce or sequence number can be added in each message to prevent replay attacks.

Note that we disregard requirements for the confidentiality of requests, responses, or the authorization policies, which can be inferred, at least partially, from the first two. Those deployments—such as in military environments—for which confidentiality of any of the three is important, can use conventional methods for end-to-end confidentiality protection, such as message encryption.

### 5.2.5 Consistency considerations

The system contains multiple PDPs that may resolve the same request based on their own access control policies. In some cases, e.g., when policy updates are made on PDPs that are responsible for overlapping sets of resources, different PDPs may return inconsistent decisions. This section discusses these cases and describes the mechanism to maintain a certain level of consistency.

We assume that each PDP makes decisions using the policy stored persistently in its local policy store. In practice, the policy store can be a policy database or a collection of policy files. We also assume that security administrators deploy and update policies through a centralized policy administration point (PAP), as in the XACML architecture [Com05], and the PAP is able to push the changes to all relevant policy stores, as in Tivoli access manager[BAR$^+$03].

We assume that the policies at each PDP will be eventually consistent [Vog09]. That is, in the absence of new updates, all the overlapped policies at each PDP converge toward identical copies of each other. Eventual consistency allows for updates to be propagated to all PDPs asynchronously so that a call for policy update may return before the update has been applied at all the PDPs. The policies of different PDPs however may be temporally inconsistent with each other. If there are no failures during an update, then there is a bound on the update propagation times. However, under certain failure scenarios (e.g., server outages or network partitions), updates may not arrive at all relevant PDPs for an extended period of time.

To PEPs, eventual consistency implies that a PEP may receive inconsistent decisions, because some PDPs may return a decision based on a policy that does not have the latest update applied. For example, consider a system with one PEP and two PDPs. $\text{PDP}_1$ receives a policy update at time $t_1$ and $\text{PDP}_2$ receives the same update at time $t_2$, where $t_2 > t_1$. If some requests are allowed before the policy update but denied after the policy update, the PEP may receive inconsistent responses for these requests between $t_1$ and $t_2$.

One approach to dealing with this situation is to require monotonic-read consistency [TS01] at each PEP. That is, if a PEP has enforced a response based on a version of policy, any successive responses enforced at this PEP should be based on the same or a more recent version of the policy. In other words, monotonic-read consistency guarantees that if a PEP is aware of a policy update at time $t$, it will never enforce the decision by an older version of policy at a later time.

To achieve monotonic-read consistency, each response includes the PAP generated time-stamp of the last policy update that the PDP received. The PEP records the policy update time-stamp it has used for the returned responses. When the PEP receives a new response for a request, it checks whether its time-stamp is at least as large as the latest time-stamp it has recorded. If it is, the PEP will enforce that response. Otherwise, it discards the response and waits for another response that comes with a same or larger time-stamp (or waits until a time-out is reached, in which case it will enforce a default decision.)[11]

### 5.2.6 Integrating pub-sub with authorization recycling

So far we have presented the design of using a pub-sub channel to replace point-to-point communication in the request-response authorization system. Now we present our design that integrates pub-sub with a SAAM-based authorization recycling system. SAAM [CLB06] achieves authorization recycling by adding a secondary decision point (SDP). Our design explores how SDPs and the pub-sub channel can work together.

Figure 5.4 shows the system architecture after adding pub-sub to the authorization recycling system. In this design, the SDP is responsible for the interaction (i.e., subscription and publishing) with the ENS. By using the pub-sub channel, the dependency of an SDP on a particular

[11]Note that using time-stamp requires time synchronization across the PDPs, which may not be trivial for WAN environments. Other techniques to achieve monotonic-read consistency have been presented in [TS01].

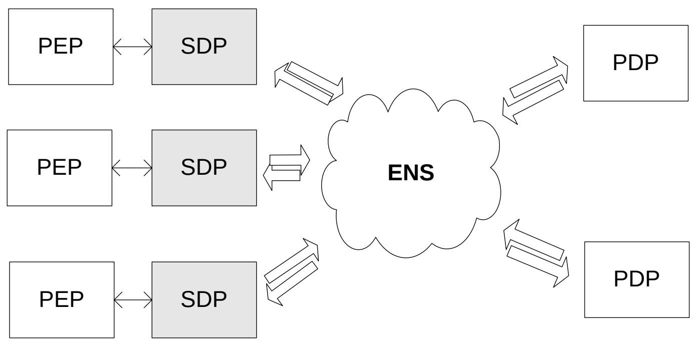


Figure 5.4: Pub-sub for the authorization recycling system.

PDP is reduced.

For each request, the PEP first tries to get a response from its own SDP. If the SDP is unable to resolve the request, the SDP then sends the request to the ENS. The returned response is added to the cache maintained by the SDP. Allowing the SDP to communicate with the ENS allows the SDP to be deployed incrementally without requiring any change to existing PEP or PDP components. Note that the discussion of security analysis and consistency mechanisms in the previous sections also applies here.

We expect that using pub-sub in conjunction with an authorization recycling system will bring the same benefits as using the pub-sub channel in the request-response authorization architecture. These benefits include improving system availability, reducing management cost and achieving better software integration. In particular, the availability improvement is significant when the SDP cache "warmness" is low and a large number of requests still needs to be resolved by the remote PDP.

Other than the aforementioned benefits, we also expect the following specific benefits:

- **SDP cooperation.** The use of pub-sub also enables cooperation between SDPs. We previously demonstrated the benefits of SDP cooperation to serve the requests from all PEPs. In the pub-sub system, the ENS can be viewed as a discovery service: The ENS helps an SDP not only find PDPs, but also find other SDPs. For this purpose, after the SDP caches a response, it subscribes to the corresponding request with the ENS. This way, the SDP informs other SDPs that it can resolve a request using its cache, thus achieving cooperation between SDPs.

- **Consistency.** Using pub-sub can also help SDPs maintain cache consistency. If an SDP is unaware of a policy update at the PDP, it may return incorrect authorization decisions. We previously proposed a policy change manager (PCM) that monitors the policy used by the PDP [WRB09]. Once the PCM detects a *critical* policy update, it informs the SDPs immediately about the updates through remote RPCs. The pub-sub architecture provides an elegant way for propagating critical policy update messages: the SDP can subscribe to the policy update messages published by the PDP; after removing the affected cache entries, the SDP then publishes a result message to the ENS indicating whether the update was successful or not.

- **Speculative authorization.** Integrating pub-sub with authorization recycling also enables *speculative authorization* [Bez05], where PDPs can predict future requests, precompute responses to them, and push these responses back to the SDPs through the ENS. Even if not all SDPs need all speculative authorizations produced by the PDPs, those authorizations that turn out to be needed will be readily available with virtually no latency observed by PEPs, ultimately improving end-users' experience. This is where the borderline between SDPs and PDPs starts to blur as PDPs effectively "push" the policy, encoded in the form of authorizations, to the SDPs.

## 5.3 Evaluation

The previous section presented our design of an authorization system based on a pub-sub architecture. This section presents an evaluation of this design. Our evaluation sought to understand the contribution of pub-sub towards improvement in availability. We also studied the system performance using various subscription schemes and under different deployment conditions.

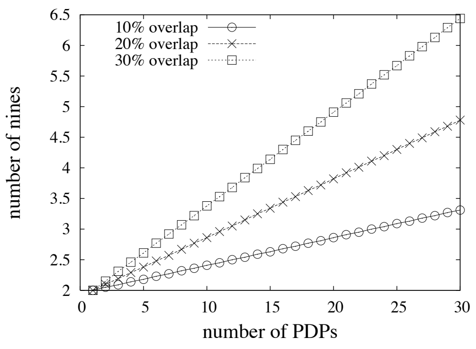
### 5.3.1 Evaluating availability

Availability measures the probability of receiving a response after a request is launched by the PEP. We expected that the use of the pub-sub architecture leads to increased availability in the presence of PDP failures because one request can possibly reach multiple PDPs, thus achieving a certain level of PDP replication. We aimed to quantify this effect through analytical analysis.

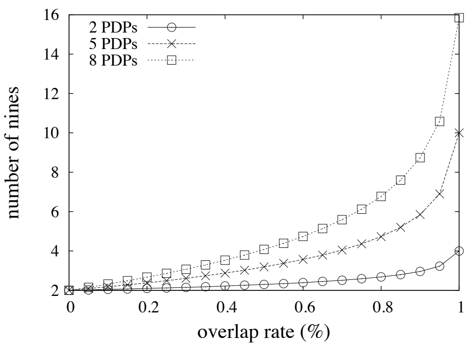
#### Evaluating the number of nines

Availability $p$ is usually reported as a number of "nines", which is the number of '9' digits after the decimal point. For example, .99999 availability is 5 nines. This translates to a down time of approximately 5 minutes per year. The number of "nines" can be generalized to include decimals that do not contain all '9' digits, using the following formula: $nines = \lfloor \log_{10} \frac{1}{(1-p)} \rfloor$. Therefore, we also used the "number of nines" as a metric of availability in the analytical analysis.

In the request-response authorization system based on point-to-point communications, a PEP sends each request only to its own PDP, while each request can possibly reach multiple PDPs in a pub-sub authorization system. Therefore, the pub-sub system can be viewed as using "PDP replication" to improve system availability. We studied the influence of the following



(a) The number of nines as a function of the number of PDPs.



(b) The number of nines as a function of overlap rate.

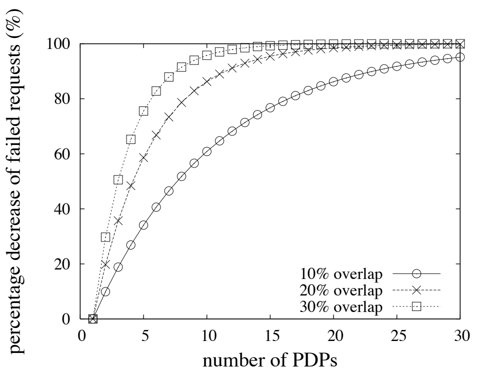
Figure 5.5: Analytical results on availability: number of nines.

two factors on availability: (a) the number of PDPs and (b) the *overlap rate* between the resource spaces of two PDPs, defined as the ratio of the objects maintained by both PDPs to the objects maintained only by the studied PDP. The overlap rate served as a measure of similarity between the resources of two PDPs. When any of these two factors increases, we expect increased availability, as one PDP will receive greater help from other PDPs in resolving a request.

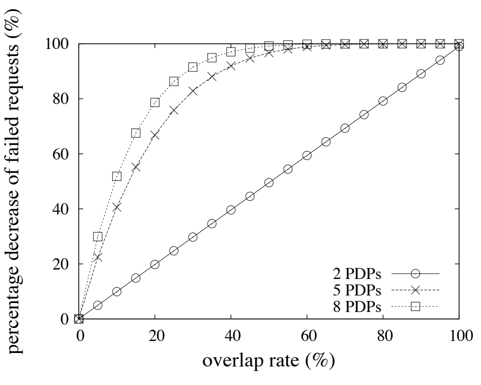
We derived an analytical model to study the effect of these two factors. Consider a system that contains $m$ PDPs, and the availability of each PDP is $p$. To simplify the analysis, we assumed that the overlap rate between the resources of every pair of PDPs is the same, i.e. equal to $o$. We also assumed that PDP's availability is independent and identically distributed. In the traditional request-response system, since each request is only sent to one PDP, the overall availability remains $p$. In the pub-sub authorization system, the system fails only after all the PDPs that can resolve that request fail. Therefore, the overall availability is $1-(1-p)\cdot(1-p\cdot o)^{(m-1)}$. In the following analysis, we studied the overall achievable availability in terms of number of nines when the availability of each PDP was .99.

Figure 5.5(a) shows the number of nines as a function of the number of PDPs with systems containing three overlap rates: 0.1, 0.2, and 0.3. The results demonstrate that the number of nines increases linearly with the number of PDPs, and the slope of the line depends on the overlap rate. The larger the overlap rate, the steeper the slope.

Figure 5.5(b) illustrates the number of nines as a function of the overlap rate with systems containing three different numbers of PDPs: 2, 5, and 8. The results demonstrate that the number of nines increases with the overlap rate in an increasing rate. Additionally, the larger the number of PDPs, the faster the increase.



(a) Percentage decrease of failed requests as a function of the number of PDPs.



(b) Percentage decrease of failed requests as a function of overlap rate.

Figure 5.6: Analytical results on availability: percentage decrease of failed requests.

**Evaluating the percentage decrease of failed requests**

We also used analytical analysis to study how our design can help reduce the number of failed requests. We estimated the *percentage decrease of failed requests* noted as $f$ in the following. Consider a system that contains $m$ PDPs and the availability of each PDP is $p$. A PEP generates $r$ requests in total. In the request-response authorization system, the number of failed requests is $f_r = r \cdot (1 - p)$. In the pub-sub authorization system, the number of failed requests is $f_p = r \cdot (1 - p) \cdot (1 - p \cdot o)^{(m-1)}$. Therefore, the percentage decrease of failed requests is $f = (f_r - f_p)/f_r$, which is $1 - (1 - p \cdot o)^{(m-1)}$. $f$ measures how other PDPs have helped resolve those failed requests in the request-response authorization system. Any increase of $f$ indicates an improvement in the availability as more requests have been successfully resolved. As in the previous analysis, we also studied the case when $p$ was 99%.

First, we studied the impact of the number of PDPs on $f$. Figure 5.6(a) shows the results. As expected, when the number of PDPs increases, $f$ also increases, which means that the PEP observes fewer failed requests. The reason is that the possibility to receive a response from other PDPs increases. In addition, the increase of $f$ is at a diminishing rate. One can thus limit the number of involved PDPs to control the overhead traffic without losing the major benefits of pub-sub.

Second, we studied the impact of the overlap rate between PDPs on $f$. Figure 5.6(b) shows the results. As expected, $f$ increases with the overlap rate. When the overlap rate is 100%, $f$ is close to 100%, which means that almost all requests have been resolved. More interestingly, the increase rate is diminishing for large overlaps. When the number of PDPs is two, $f$ increases linearly with the overlap rate, as $f$ is a linear function of the overlap rate, i.e., $f = p \cdot o$.

### Discussion

The above analytical analysis confirms that the use of the pub-sub model leads to higher availability. In addition, the results demonstrate that the availability increases with the number of PDPs and their resource overlap rates.

However, we note that the increase of availability does not come without cost. In particular, a pub-sub system uses multiple PDPs to achieve partial (computational) replication. That is, the same task (i.e., computing an authorization decision) might be executed multiple times on different PDPs, depending on the subscriptions posted by each PDP. Since each PDP now needs to resolve requests not only for its own PEPs but also for other PEPs, the load of each PDP is increased.

We quantify this additional load to each PDP as follows. Consider a system that contains $m$ PDPs and each PDP receives $n$ requests in a request-response system. Let us consider the load of $\text{PDP}_i$. In the pub-sub system, since each other PDP has an overlap rate $o$ with $\text{PDP}_i$, the total number of extra requests that $\text{PDP}_i$ needs to process is $n \cdot (m-1) \cdot o$. Therefore, the extra load increases linearly with the overlap rate and the number of PDPs.

We identified two possible approaches to reducing the additional computational load on each PDP in the deployment. We assume that each PEP has a default PDP as in the request-response system. First, the PEP adds a "destination" constraint (or attribute) in each published request which specifies the receiver of this request, usually its default PDP. Hence, every request is only delivered to its default PDP by the ENS. When the PEP detects that the number of outstanding requests begins increasing, the PEP infers that its default PDP has failed. Then the PDP removes the destination constraint from future requests and thus these requests will reach multiple PDPs. By sending requests to multiple PDPs only when the failure of the default PDP is detected, the PDP load is reduced.

An alternative approach is to give each PDP the ability to decide at runtime whether it wants to help other PDPs or not. This requires each request to indicate its sender. When a PDP is busy, it only resolves the requests coming from its "own" PEPs. Only when the PDP has idle computation cycles does it start to resolve the requests from other PEPs.

## 5.3.2 Evaluating performance

Along with studying the system availability using an analytical analysis, we developed a prototype to evaluate system performance and its dependency on various factors. In this section, we first describe the implementation of our prototype system, followed by an explanation of the experimental setup. We then present the performance evaluation results. Finally, we discuss the results.

### System prototype

Our prototype contained three main components: the ENS, the PEP, and the PDP. The PEP published requests to the ENS and listened to the responses, the PDP resolved requests and published responses back, and the ENS delivered requests/responses between PEPs/PDPs.

For the ENS, we used Siena [CRW01], a content-based ENS developed in Java. We chose Siena for two reasons: (1) it provides the necessary functionality to study feasibility and performance; (2) its code base is well maintained. In Siena, an event is published as a set of attribute-value pairs. Attribute names are simply strings, and values are from a predefined set of primitive types, for which a fixed set of operators is defined. Figure 5.7 shows how a PEP publishes a request in Siena.

```
Notification e = new Notification();
e.putAttribute("subject", "Sean");
e.putAttribute("object", "/etc/passwd");
e.putAttribute("access", "read");
siena.publish(e);
```

Figure 5.7: PEP publishes a request in Siena

The subscriber subscribes to events by specifying filters using the subscription language. The filters define constraints, usually in the form of name-value pairs of attributes and basic comparison operators ($=, <, \leq, >, \geq$), which identify valid events. Figure 5.8 shows how a PDP subscribes to the request for an object in Siena.

```
Filter f = new Filter();
f.addConstraint("object", "/etc/passwd");
siena.subscribe(f);
```

Figure 5.8: PDP subscribes to a request in Siena

The PEP was organized in three threads. The first thread was responsible for publishing requests and posting the subscriptions for the corresponding responses to the ENS. The second thread was responsible for receiving the response notifications from the ENS. Once a response was received for a request, the response was put into a queue. The third thread was responsible for getting the response from the queue and then unsubscribing from that response. Separating subscription, notification, and unsubscription into different threads was to avoid deadlocks [Car09] and improve code performance.

To study the effect of PEP subscriptions, we implemented the four subscription schemes discussed in Section 5.2.3: request-based subscription, subject-based subscription, session-based subscription and callback. For the callback scheme, the notification thread was replaced by a UDP server thread which was responsible for receiving returned responses. Each request sent by the PEP included the IP address and port number of the UDP server. After the PDP computed a response, the PDP returned the response by sending it as a UDP message to the

PEP directly.

The PDP implementation contained a policy evaluation engine that resolved requests based on a role-based access control (RBAC) policy. The policy contained two XML files: one defined user-to-role assignment (UA) and the other defined permission-to-role assignment (PA).

### Evaluation methodology

We used two metrics to evaluate performance: response time and maximum throughput. The response time was measured as the time elapsed from the moment when the PEP sent a request to the moment when it received the response for that request. The system maximum throughput was measured as the maximum number of requests that could be processed by the system per second.

**Workload generator** In order to simulate subjects' behaviors, we developed a workload generator to generate authorization requests with certain properties. A request is a tuple consisting of a subject, an object, and an access right. For each request, the object and access right were randomly selected from the object and access right sets. We focused on the subject generation in each request. In particular, the generator generated subjects according to the following parameters: (1) $s$, the total number of subjects, (2) $q$, the number of requests generated per second, (3) $n$, the percentage of requests in $q$ that were issued by active subjects, and (4) $a$, the number of active subjects in the system.

The generator maintained two sets of subjects: *active* and *inactive* subject sets. The subjects in the active set were currently in the system. The subjects in the inactive set either had not entered the system yet or had left the system. The algorithm used to generate a subject for a request was as follows. In the beginning, the active set was null while the inactive set contained all the subjects in the system. When the generator started to generate requests, each request was either issued by an active or inactive subject, which was randomly selected from the corresponding set. The sequence however was determined by $n$. For example, if $n$ was 80%, then the generator generated four requests by active subjects and then generated one request by an inactive subject. Once an inactive subject had been selected to issue a request, this subject was removed from the inactive set and moved to the active set. When the size of the active set reached $a$, this size would not increase anymore. To enforce this, whenever a new active subject was added, the oldest subject (that had been in the active set longest time) was removed and added to the inactive set. This can be interpreted as the oldest subject having closed the session in the system. These steps were repeated until a desirable number of requests were generated.

The request trace generated by this algorithm particularly enabled us to study the impact of two factors on the session-based subscription scheme. First, by controlling the number of active subjects, we fixed the maximum number of outstanding PEP subscriptions. Second, by controlling the percentage of requests issued by active subjects, we fixed the subscription frequency.

**Experimental setup** The experiments were run in a computer cluster. Each node in the cluster was equipped with two Intel Xeon 2.33 GHz processors and 4 GB of memory, running Fedora Linux 2.6.24.3. Each node connected to other nodes through a 1Gbps network. The round-trip-time (RTT) was around 0.1ms. We used the Linux Netem package to emulate the properties of a wide area network (WAN). In particular, we used 40ms RTT time to simulate locations in two adjacent states.

At the start of each experiment, the ENS server first started. Then the PDP booted up and subscribed to all the objects that appeared in the policy. The workload generator then started to generate authorization requests and send them to the PEP. The time between two consecutive requests was derived from an exponential distribution as used in TPC-W [TPC02]. The mean time interval $\mu$ was usually a few milliseconds. We used $\mu$ to calculate the incoming request rate (requests per second) $q$ as $1,000/\mu$. After the PEP received a request, it subscribed to the response to that request using one of the four subscription schemes, and then published the request to the ENS. After the PDP received a request, it computed the decision and then posted the response back to the ENS. Our experimental results show that the PDP computed each response in less than 0.1ms.

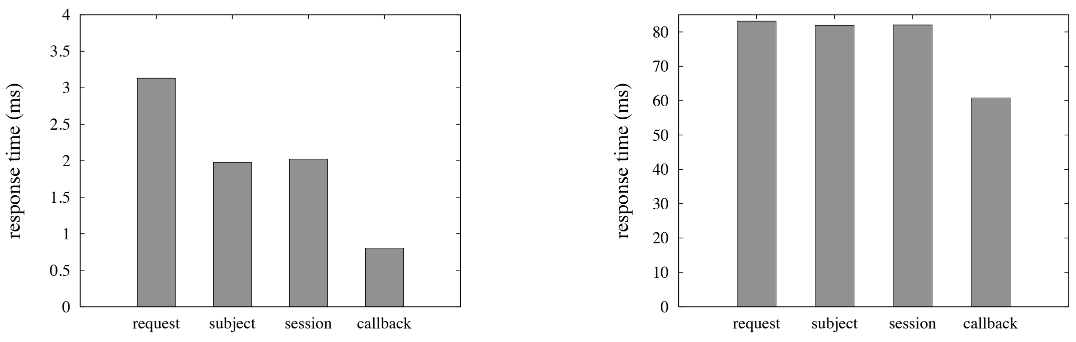
In the *basic setup*, we used one PEP, one PDP, and one ENS server, and each of them was located on its own cluster node. The workload generator generated 100 requests per second. On average, 80 out of these requests were issued by active subjects. The policy contained 500 subjects, 500 objects, and 3 access rights. The number of active users was 100. The workload generator generated 20,000 requests in total, and the response time was measured as the average of the response times of the last 5,000 requests.

Each experiment was run ten times and the average response time results are reported. The observed standard deviations for all experiments were very small, i.e., less than 0.05ms for all of them.

### Experimental results

We now describe the experimental results. Our previous analysis in Section 5.2.3 suggests that the ENS performance should be mainly affected by the number of subscriptions and subscription frequency. This is in turn affected by the subscription scheme used and other factors like the request rate, number of subjects, etc. By exploring the dependency of performance on these factors, the results shed light on the trade-offs one should consider in deploying a pub-sub authorization system.

In the following, we first describe the results that compare the response time of four subscription schemes in the basic setup. Then we present the results on how the performance was affected by varying the number of outstanding subscriptions. We then describe the results specific to the session-based scheme, which show the impact of the subscription frequency and



(a) LAN scenario. (b) WAN scenario.

Figure 5.9: Response time comparison for the four PEP subscription schemes: request-based, subject-based, session-base, and callback.

number for active subjects. We finally present the results for the system that contained multiple PEPs or multiple PDPs.

**Basic response time comparison** In comparing the response times of different schemes in the basic setup, we studied the response time in both LAN and WAN scenarios.

Figure 5.9 illustrates the response time for each PEP subscription scheme. The results show that different schemes achieved different response times. In our system, there are three factors that contribute to the response time: the network latency, the time used by the PDP to resolve a request, and the time used by the ENS to handle requests and responses. Since the network latency and the time used by the PDP were the same for all the four schemes, any difference in the response time was caused by the time used by the ENS, which in turn was decided by the number of subscriptions at the ENS and the subscription frequency posted by the PEP.

The following observations can be made from Figure 5.9:

- The callback scheme led to the shortest response time. The reason is likely that each response was directly returned to the PEP, bypassing the ENS. In addition, since the PEP did not post any subscription requests to the ENS, the ENS load was also reduced. Figure 5.9(b) also demonstrates that using the callback scheme saved one-way network delay (20ms). This saving would be more significant when the number of ENS nodes needed to route the message increases.

- The request-based subscription scheme led to the longest response time. This was due to the increased ENS overhead by frequent (un)subscription. However, we can not conclude yet that the other two subscription schemes, i.e., subject-based or session-based schemes, will always achieve lower response time, because their response times depend on other factors, e.g., the number of outstanding subscriptions. We studied this impact in the next two experiments.

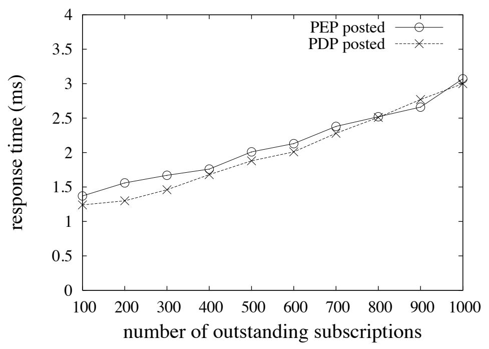


Figure 5.10: Response time as a function of the number of subscriptions.

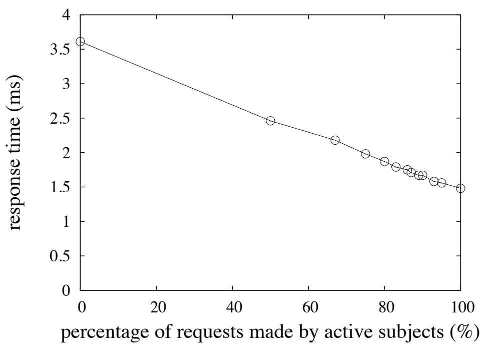
- The session-based scheme achieved similar response time to the subject-based subscription scheme. The reason is that, although the number of subscriptions was reduced in the session-based scheme, the subscription frequency was increased compared to the subject-based scheme. In particular, in order to maintain a small number of subscriptions, the session-based scheme required the PEP to post (un)subscriptions to the ENS at a constant frequency.

- Figure 5.9(b) demonstrates that the overhead added by the ENS was trivial compared to the network latency. When the network latency was high (or the time for the PDP to resolve a request was high due to the complex authorization logic), the ENS overhead was negligible.

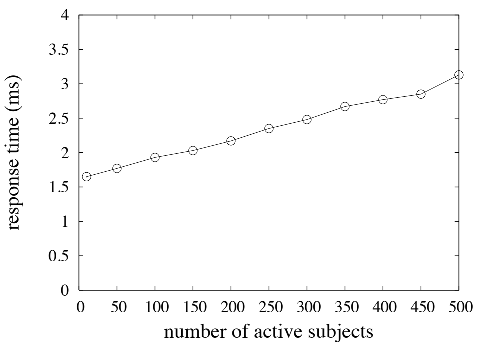
**The impact of the number of subscriptions** Next, we studied the extent to which the system performance depends on the number of outstanding subscriptions registered at the ENS. In the first experiment, we studied PDP subscriptions. We varied the number of objects from 100 to 1,000, thus varying the number of subscriptions posted by the PDP. We only ran the experiment using the session-based scheme as the results are also applicable to other subscription schemes. In the second experiment, we studied PEP subscriptions. We ran experiment using the subject-based scheme. We varied the number of subjects from 100 to 1,000, thus varying the number of subscriptions posted by the PEP.

Figure 5.10 shows the response time as a function of the number of outstanding subscriptions made by the PEP or the PDP. The result demonstrates that the response time increased almost linearly with the number of subscriptions. Therefore, it is important to keep the number of outstanding subscriptions small. For PDPs, this can be achieved by using the hierarchy-based or application-based subscription schemes. For PEPs, this can be achieved by subscribing to active subjects, as in the session-based scheme.

Note that this result depends on the matching algorithm used by a specific ENS implementation. In our case, this result is specific to Siena. Some other ENS implementa-



(a) The impact of (un)subscription frequency.



(b) The impact of the number of outstanding subscriptions.

Figure 5.11: Results for the session-based subscription scheme.

tions [ASS$^+$99, CCC$^+$01] use an optimized matching algorithm, where the response time may increase sublinearly with the number of subscriptions. An interesting direction for future work would be studying how our system would perform if these algorithms are used in Siena.

The result also advocates that it is important to regularly clean up obsolete subscriptions registered in the ENS to reduce the number of outstanding subscriptions. A subscription becomes obsolete when the corresponding party is no longer interested in the event matching that subscription. For example, in the request-based subscription scheme, if the PEP fails to receive the response for a request within a certain time frame, the PEP will enforce a default decision and will no longer need the response for that request. In this case, if the PEP is not triggered to issue an unsubscription call to the ENS, the number of subscriptions would accumulate in the system and lead to increased response time.

**Session-based subscriptions** Next, we focused on the session-based subscription scheme. Unlike other schemes, the performance of this scheme is affected by both the (un)subscription frequency and the number of outstanding subscriptions (or active subjects).

In the first experiment, we varied the (un)subscription frequency by varying the percentage of the requests that were made by active subjects. When this number increased, the subscription frequency decreased. Specifically, we varied it from 0 to 100% while fixing the number of active subjects at 100. When it was 0%, all requests were issued by the inactive subjects, thereby every request incurred an (un)subscription call to the ENS. When it was 100%, all requests were issued by the active subjects, thereby no subscription was posted. Figure 5.11(a) shows the result. As expected, the response time decreased almost linearly with the percentage of requests made by active subjects, due to the decrease in (un)subscription frequency. This result further confirms that it is important to reduce the (un)subscription frequency.

In the second experiment, we fixed the percentage of the requests made by active subjects at 80% while varing the number of active subjects from 10 to 500. Figure 5.11(b) shows

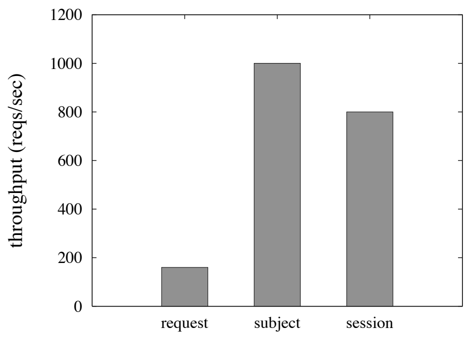


Figure 5.12: The impact of concurrent PEP subscriptions on the throughput.

that the response time increased with the number of active subjects. The result is consistent with Figure 5.10, as the number of active subjects determined the number of outstanding subscriptions posted by the PEP.

**The impact of concurrent PEP subscriptions** In the next experiment, we used four PEPs to study the system throughput under multiple concurrent requests. We estimated the maximum throughput in the following way. In each run of the experiment, the rate $q$ at which the PEP generated requests was fixed. We monitored the number of outstanding requests that were waiting for a response. If this number monotonically increased for five consecutive measurements, then we concluded that the ENS was saturated and the experiment was aborted. We iteratively varied $q$ across different experimental runs to identify the minimum value of $q$. The system maximum throughput was then measure as the sum of $q$ of four PEPs.

Figure 5.12 compares the throughput for the three subscription schemes that involved PEP subscriptions. In particular, the request-based subscription scheme introduced maximum overhead due to the frequent subscriptions and unsubscriptions. The subject-based scheme performed slightly better than the session-based scheme even though the number of outstanding subscriptions was lower in the session-based scheme. This result suggests that it is important to keep the frequency of subscriptions low to improve system throughput, especially when the system contains multiple concurrent requests. In this case, any subscription call to the ENS has to wait until another subscription call finishes because updates to the ENS routing table are mutually exclusive.

**The impact of duplicate PDP subscriptions** In the next experiment, we used four PDPs to study the system behavior under duplicate subscriptions. In the experiment, each PDP posted 1,000 subscriptions. Therefore, the total number of outstanding PDP subscriptions was always 4,000. By varying the overlap rate between PDPs, we studied how the overlap rate affected the response time. A larger overlap rate led to more duplicate subscriptions posted by

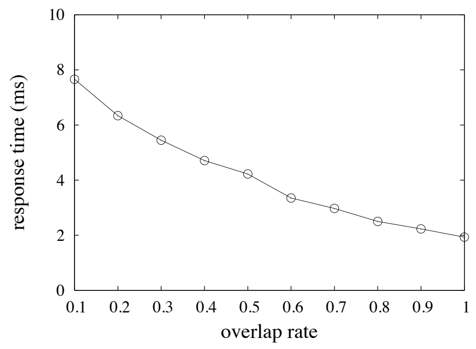


Figure 5.13: Response time as a function of overlap rate, when 4 PDPs exist.

different PDPs. We only tested the session-based subscription scheme since the results would also apply to other PEP subscription schemes.

Figure 5.13 shows the response time as a function of the overlap rate. The results demonstrate that the response time decreased with the overlap rate. The likely reason is that subscriptions with identical predicates were coalesced into a single subscription in the ENS routing table(s). Therefore, the size of the routing tables at the ENS node was reduced, thus saving the matching time.

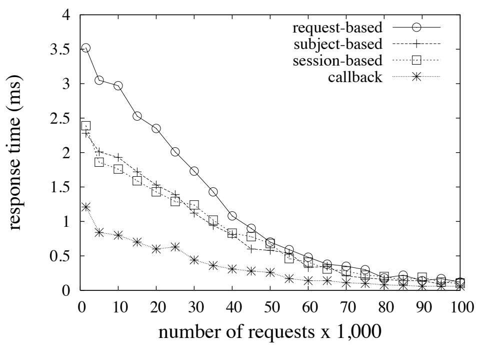
**Evaluating pub-sub with authorization recycling** To study the integration of pub-sub with authorization recycling, we also added an SDP component to the prototyped system. The SDP implemented the $\text{SAAM}_{\text{RBAC}}$ authorization recycling algorithms described in [WCBR09]. When the PEP received a request, it first sent the request to the SDP. Only when the SDP failed to resolve this request was the request sent to the ENS.

We studied the performance of the SAAM-enabled pub-sub system. In the experiment, the PEP sent 100,000 randomly generated requests in total. Figure 5.14 shows the results. As expected, adding the SDP helps to reduce the response time significantly. In other words, the impact of the pub-sub's overhead on the performance is only significant when the cache warmness is low. As the cache warmness increases, more requests can be resolved by the local SDP, thus reducing the time to obtain responses and reducing the load on the ENS.

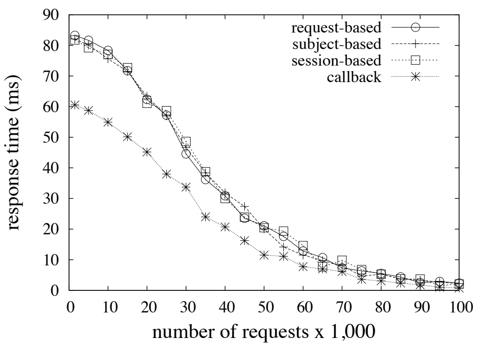
### Discussion

From the experimental results, we can draw the following conclusions about the four PEP subscription schemes:

- Request-based subscription. This scheme is not affected by the number of subjects in the system, since each subscription is removed once the response is received. However, this scheme generates the highest overhead at the ENS due to the frequent subscriptions and unsubscriptions, which results in significantly-reduced system throughput. Therefore, this



(a) LAN scenario.



(b) WAN scenario.

Figure 5.14: Response time comparison with approximate recycling

scheme is only suitable for those scenarios where the number of concurrent requests and the request rate are low.

- Subject-based subscription. This scheme largely reduces the (un)subscription overhead when compared to the request-based subscription scheme. However, this is at the cost of a large number of subscriptions registered in the ENS, as the PEP may eventually subscribe to all subjects. Therefore, the scheme is useful for those systems with a small number of subjects or for those applications with good locality in terms of subjects.

- Session-based subscription. This scheme reduces the number of outstanding subscriptions, compared to the subject-based subscription scheme, and reduces the subscription frequency, compared to the request-based subscription scheme. However, when the system contains multiple concurrent requests, it is desirable to keep the subscription frequency low to achieve high throughput.

- Callback. In all the experiments, this scheme achieves best performance because it only partially uses the ENS. However, this scheme may need additional infrastructure support in a real deployment to deliver messages from the PDP to the PEP and does not support cooperative authorization recycling or speculative authorizations.

Finally, our results also confirm that by deploying SDPs, the impact of the ENS on system performance was reduced. Therefore, it is desirable to add SDPs to the pub-sub authorization system to improve system performance.

In our experiments, we used one ENS node. A real deployment may contain multiple distributed ENS nodes. We believe that our experimental results apply to those environments as well, since the factors we studied, e.g., the number of subscriptions, affect the performance of any individual node in a distributed ENS in a similar manner. We may evaluate the system scalability with a large number of PEPs and PDPs, which basically leads to more outstanding subscriptions and concurrent requests. However, this type of evaluation would be very specific to the ENS implementation used. To avoid the performance bottleneck in a distributed ENS, a load-balancing technique should be deployed to evenly distribute subscriptions to every node. In other words, the responsibility of matching requests/responses to PDPs/PEPs is distributed evenly among ENS nodes.

## 5.4 Summary

In this chapter, we present a design that uses the publish-subscribe architecture to improve the robustness and manageability of access control systems. By deploying an ENS between applications and authorization servers, our approach enables multiple PDPs to form a reliable PDP cloud to serve all PEPs. We also expect that using a standard ENS can reduce the overhead of authorization system development and management.

We define system requirements, and present a detailed design that meets these requirements. Our design is generic so that it is independent of the implementation of a specific ENS as well as the underlying access control policies. To reduce the impact of the ENS on system performance, we propose various subscription schemes that can be used in different scenarios. In addition, we discuss strategies to enforce security and consistency in our design. We evaluate our design using an analytical analysis, which confirms the improvement in availability. We also build a prototype system and evaluate the system performance under different scenarios.

Along with the traditional authorization infrastructure based on the request-response model, we also explore how the publish-subscribe channel can be integrated with the authorization recycling system and how it can support cooperative authorization recycling as well as speculative authorizations. The integration is expected to further improve authorization system availability and performance.

# Chapter 6

# Conclusion

As enterprise application systems become increasingly large and complex, their authorization infrastructures face new challenges. Conventional request-response authorization architectures that use centralized policy decision points (PDPs) become fragile and scale poorly. One state-of-the-practice approach to improving overall system availability and reducing the authorization processing delays observed by the client is to recycle previous authorizations at the policy enforcement point (PEP). These solutions, however, only employ precise authorization recycling: a cached authorization is reused only if the authorization request in question exactly matches the original request for which the previous authorization was made. As a result, the number of requests that can be resolved by the cache is low and the application still depends much on the remote authorization service.

This dissertation introduces three approaches to achieve the goals of improving the availability and performance of enterprise access control systems. In Chapter 3, we first propose SAAM$_{\text{RBAC}}$, which extends the precise authorization recycling approach by enabling the inference of approximate authorizations for role-based access control (RBAC) systems. We define inference rules specific to RBAC authorization semantics and develop recycling algorithms based on these rules. These algorithms can be integrated with existing RBAC systems and be used to cache authorization decisions and to infer approximate decisions from cached data. Our evaluation results demonstrate significant increase in the number of authorization requests that can be served without consulting the original decision point, as compared to precise recycling. Meanwhile, the time used to infer approximate responses is low. These results suggest that deploying SAAM$_{\text{RBAC}}$ improves the availability and scalability of RBAC systems, and in turn the performance of entire enterprise application systems.

In Chapter 4, we propose a cooperative secondary and approximate recycling (CSAR) approach to further improving the availability and performance of authorization solutions. This approach explores the cooperation between secondary decision points (SDPs), which is especially practical in distributed systems involving cooperating parties or replicated services, due to the high overlap in their user/resource spaces and the need for consistent policy enforcement. We analyze CSAR's design requirements, and propose a concrete architecture that meets these requirements. The evaluation results suggest that even with small caches (or low cache warmness), our cooperative authorization solution can still offer significant benefits.

SAAM and CSAR can be viewed as client-side (or PEP-side) techniques to improve system availability and performance. In Chapter 5, we further propose the use of a publish-subscribe architecture, which can be viewed a combination of client-side and server-side (PDP-side) techniques. We present and evaluate an authorization system that uses an event notification service (ENS) to replace the existing point-to-point communication architecture. Unlike in a point-to-point architecture, where PEPs are configured to send their requests to specific PDPs, a pub-sub architecture enables PEPs to send their requests without knowing which PDP will receive them. Similarly, the PDPs show interest in requests without knowing which PEPs generate them. By using the pub-sub architecture, the coupling between specific PEPs and PDPs is removed; remote PDPs form a reliable PDP cloud that serves the PEP client populations. As a result, system availability is improved and system administration is simplified.

Finally, we also show that these approaches can be integrated together in a synergistic way to provide even higher availability and performance. Using authorization recycling, the overhead of using an ENS between PEPs and PDPs is reduced. Furthermore, the pub-sub architecture also helps authorization recycling. First, the ENS can work as a discovery service for an SDP to find PDPs or other SDPs to resolve a request, thereby unifying the SDP cooperation with PDP cooperation. Second, the PDP can push policy update messages to SDPs through the ENS to maintain cache consistency. Third, the ENS also facilitates speculative authorizations, by which the PDP pre-computes authorizations and pushes them to SDPs for recycling.

## 6.1 Future work

Although we demonstrate that our approaches improve the availability and performance of enterprise authorization systems, there are several areas for future research.

Recall that SAAM defines an authorization request as a tuple $(s, o, a, c, i)$, where $s$ is the subject, $o$ is the object, $a$ is the access right, $c$ is the contextual information relevant to the request, and $i$ is the request identifier. Our SAAM$_{\text{RBAC}}$ recycling algorithms have ignored the contextual information. This is an important issue to consider in our future research, as contextual information has become increasingly important in making decisions in enterprise environments. Contextual information may come from a variety of sources, such as user profiles, the time of a day, IP addresses, and geographic locations. For example, a transaction that accesses sensitive information may be allowed only if the user is connecting from inside the corporate network. We plan to explore SAAM algorithms that are able to recycle those decisions that are made with contextual information.

The SAAM framework of making secondary and approximate access control decisions is independent of the specifics of the application and access control policy. For each class of access control policy, however, specific algorithms for inferring approximate responses—generated according to the particular access control model—need to be designed. This dissertation describes SAAM$_{\text{RBAC}}$, and previously SAAM$_{\text{BLP}}$ was also proposed [CLB06]. One avenue for future work would be to explore SAAM recycling algorithms for other types of access control policy. The challenge lies in examining the semantics for each type of access control policy and using them to develop efficient inference algorithms. Beznosov [Bez05] also points to a number of future work areas in this direction. For example, it would be interesting to explore how the proposed recycling approach is effective in environments where authorizations depend on access history, as in Chinese-Wall [BN89] policies. In addition, it is challenging to design a "stateless" SDP to support the polices based on *consumable rights*, or more generally *mutable attributes* [PS04], of subjects.

Additionally, the current SAAM recycling framework only considers the SDP and the PDP that make decisions, but neglects the PEP that finally enforces those decisions. Considering the PEP would suggest further research directions. For example, the PEP may implement a deny-by-default (DBD) enforcement scheme, where the PEP denies those requests for which a deny or undecided decision is returned. In this case, deny and undecided responses do not make a difference to the PEP. Therefore, the SDP only needs to compute allow responses, which may improve the performance of authorization recycling algorithms and reduce the burden of SDPs. A future research direction is to consider PEP, SDP and PDP as a unified access control system and to study different optimization choices for recycling algorithms.

For the approach using the publish-subscribe architecture, we have proposed general PEP subscription mechanisms, which are independent of the underlying authorization policy. The design of policy-aware subscription mechanisms could be another direction for future work. In particular, when the underlying policy model is known, we can study the policy semantics to provide a better subscription scheme. For example, in RBAC, the PEP can subscribe to the activated roles of a subject instead of the subject identities. By subscribing to roles, the number of subscriptions can be significantly reduced as multiple users may have similar roles. In addition, a returned response for one request may be reused for other outstanding requests due to their similar subscriptions. Hence, we expect that using a policy-aware subscription scheme will further reduce the ENS overhead in matching responses to requests and reduce system response time.

Furthermore, our current design of the pub-sub system is based on a generic ENS. In some circumstances, however, an access-control-specific ENS may be desirable. Consider that PEPs subscribe to the activated roles of a subject. If an allow response is returned for the request issued by subject $\{r_1\}$, this response can also be used for the request for the same permission but issued by subject $\{r_1, r_2\}$.[12] Similarly, a deny response returned for subject $\{r_1, r_2\}$ can be used for subject $\{r_1\}$. An access-control-specific ENS should be able to apply these rules in routing responses to the corresponding PEPs so that one response can be reused for multiple outstanding requests. Hence, we expect that an access-control-specific ENS is able to route the authorization request/response between PEPs and PDPs more efficiently, thus improving system performance.

[12]Note that this logic may not hold if mutual exclusion is used in an environment. For example, access is granted if the subject has $\{r_1\}$, or if the subject has $\{r_2\}$, but is denied if the subject has both $\{r_1\}$ and $\{r_2\}$ simultaneously.

# Bibliography

[AFM05] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: scaling file servers via cooperative caching. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 129–142, Berkeley, CA, USA, 2005. USENIX Association.

[AH02] L.A. Adamic and B.A. Huberman. Zipf's law and the Internet. *Glottometrics*, 3(1):143–50, 2002.

[All05] J. Allen. Governing for Enterprise Security. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005.

[And72] James Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Vols. I and II, Air Force Electronic Systems Division, 1972.

[ANS04] ANSI. ANSI INCITS 359-2004 for role based access control, 2004.

[ASA01] Mark Astley, Daniel C. Sturman, and Gul A. Agha. Customizable middleware for modular distributed software. *Communications of the ACM (CACM)*, 44(5):99–107, 2001.

[ASS$^+$99] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching events in a content-based subscription system. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 53–61, 1999.

[BAR$^+$03] Axel Bücker, Jesper Antonius, Dieter Riexinger, Frank Sommer, and Atsushi Sumida. *Enterprise Business Portals II with IBM Tivoli Access Manager.* IBM Redbooks, ibm.com/redbooks, March 23 2003.

[BBQV07] Roberto Baldoni, Roberto Beraldi, Leonardo Querzoni, and Antonino Virgillito. Efficient publish/subscribe through a self-organizing broker overlay and its application to SIENA. *The Computer Journal*, pages 444–459, 2007.

[BCF$^+$99] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the Conference on Computer Communications (INFOCOM'99)*, pages 126–134, New York, NY, USA, 1999. IEEE Computer Society.

[BCM+99] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R.E. Strom, and D.C. Sturman. Efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99)*, pages 262–272, 1999.

[BCR+09] Lujo Bauer, Lorrie Faith Cranor, Robert W. Reeder, Michael K. Reiter, and Kami Vaniea. Real life challenges in access-control management. In *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, pages 899–908, New York, NY, USA, 2009. ACM.

[BDB+99] Konstantin Beznosov, Yi Deng, Bob Blakley, Carol Burt, and John Barkley. A resource access decision service for CORBA-based distributed systems. In *Annual Computer Security Applications Conference (ACSAC'99)*, pages 310–319, Phoenix, Arizona, USA, 1999.

[BDS00] D. Balfanz, D. Dean, and M. Spreitzer. A security infrastructure for distributed Java applications. In *IEEE SYMPOSIUM ON SECURITY AND PRIVACY*, pages 15–26. IEEE Computer Society, 2000.

[Bez98] Konstantin Beznosov. Issues in the security architecture of the computerized patient record enterprise. In *Second Workshop on Distributed Object Computing Security*, Baltimore, Maryland, USA, 1998.

[Bez00] Konstantin Beznosov. *Engineering Access Control for Distributed Enterprise Applications*. Ph.D. dissertation, Florida International University, 2000.

[Bez05] Konstantin Beznosov. Flooding and recycling authorizations. In *Proceedings of the New Security Paradigms Workshop (NSPW'05)*, pages 67–72, Lake Arrowhead, CA, USA, 20-23 September 2005. ACM Press.

[BGR05] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Distributed proving in access-control systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 81–95, Oakland, CA, 2005. IEEE Computer Society.

[BH05] Dieter Buehler and Thomas Hurek. Performance tuning of portal access control. http://www.ibm.com/developerworks/websphere/library/techarticles/0508_buehler/0508_buehler.html, 2005.

[BL73a] D.E. Bell and L. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Volume I, Mitre Corporation, Bedford, Massachusetts, 1973.

[BL73b] D.E. Bell and L. LaPadula. Secure computer systems: A mathematical model. Technical Report MTR-2547, Volume II, Mitre Corporation, Bedford, Massachusetts, 1973.

[BN89] D. Brewer and M. Nash. The Chinese Wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, California, 1989. IEEE Computer Society Press.

[Bre00] Eric A. Brewer. Towards robust distributed systems. In *ACM Symposium on Principles of Distributed Computing (PODC'00)*, Portland, Oregon, 2000. Invited talk.

[BRS02] Ashwin R. Bharambe, Sanjay Rao, and Srinivasan Seshan. Mercury: a scalable publish-subscribe system for internet games. In *NetGames '02: Proceedings of the 1st workshop on Network and System Support for Games*, pages 3–9, 2002.

[BSF02] Lujo Bauer, Michael A. Schneider, and Edward W. Felten. A general and flexible access-control system for the web. In *Proceedings of the 11th USENIX Security Symposium*, pages 93–108, Berkeley, CA, USA, 2002. USENIX Association.

[BZP05] Kevin Borders, Xin Zhao, and Atul Prakash. CPOL: high-performance policy evaluation. In *Proceedings of the 12th ACM conference on Computer and Communications Security (CCS'05)*, pages 147–157, New York, NY, USA, 2005. ACM Press.

[Car09] Antonio Carzaniga. personal communication, 2009.

[CCC$^+$01] Alexis Campailla, Sagar Chaki, Edmund Clarke, Somesh Jha, and Helmut Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 443–452, Washington, DC, USA, 2001. IEEE Computer Society.

[CDNF01] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9):827–850, Sep 2001.

[CEE$^+$01] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, and Ronald L. Rivest. Certificate chain discovery in spki/sdsi. *Journal of Computer Security*, 9(4):285–322, 2001.

[Cha05] David Chadwick. Authorisation in grid computing. *Information Security Technical Report*, 10(1):33 – 40, 2005.

[CJ02] Gianpaolo Cugola and H.-Arno Jacobsen. Using publish/subscribe middleware for mobile systems. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):25–33, 2002.

[CKK02] Yan Chen, Randy H. Katz, and John Kubiatowicz. Dynamic replica placement for scalable content delivery. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 306–318, London, UK, 2002. Springer-Verlag.

[CLB06] Jason Crampton, Wing Leung, and Konstantin Beznosov. Secondary and approximate authorizations model and its application to Bell-LaPadula policies. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT'06)*, pages 111–120, Lake Tahoe, CA, USA, June 7–9 2006. ACM Press.

[Com05] XACML Technical Committee. OASIS eXtensible Access Control Markup Language (XACML) version 2.0. OASIS Standard, 1 February 2005.

[CRW00] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *PODC'00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 219–227, 2000.

[CRW01] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[CZO+08] David Chadwick, Gansen Zhao, Sassa Otenko, Romain Laborde, Linying Su, and Tuan Anh Nguyen. Permis: a modular authorization infrastructure. *Concurrency and Computation: Practice and Experience*, 20(11):1341–1357, 2008.

[Dav00] T.H. Davenport. The future of enterprise system-enabled organizations. *Information Systems Frontiers*, 2(2):163–180, 2000.

[Dav01] Brian D. Davison. A web caching primer. *IEEE Internet Computing*, 5(4):38–45, 2001.

[DB92] Paul Dourish and Sara Bly. Portholes: supporting awareness in a distributed work group. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 541–547, 1992.

[DBC+00] D. Durham, J. Boyle, R. Cohen, S. Herzog, R. Rajan, and A. Sastry. The COPS (common open policy service) protocol. *IETF RFC2748*, 2000.

[DK06] Linda DeMichiel and Michael Keith. JSR-220: Enterprise JavaBeans specification, version 3.0: EJB core contracts and requirements. Specification v.3.0 Final Release, Java Community Program, May 2006.

[Edd99] Guy Eddon. The COM+ security model gets you out of the security programming business. *Microsoft Systems Journal*, 1999(11), 1999.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[EHM$^+$08] Alina Ene, William Horne, Nikola Milosavljevic, Prasad Rao, Robert Schreiber, and Robert E. Tarjan. Fast exact and heuristic methods for role minimization problems. In *SACMAT '08: Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 1–10, New York, NY, USA, 2008. ACM.

[Ent99] Entrust. GetAccess design and administration guide, September 20 1999.

[FCAB00] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[FH02] S. Farrell and R. Housley. An internet attribute certificate profile for authorization, 2002.

[FK67] W.N. Francis and H. Kucera. Computational analysis of present-day American English. Providence, RI: Brown University Press, 1967.

[FK92] D. Ferraiolo and R. Kuhn. Role-based access controls. In *Proceedings of the 15th NIST-NCSC National Computer Security Conference*, pages 554–563, Baltimore, MD, USA, 1992. National Institute of Standards and Technology/National Computer Security Center.

[Gar02] Gartner. Cutting implementation costs by application integration. press release, 2002.

[GB99] R. Grimm and B. Bershad. Providing policy-neutral and transparent access control in extensible systems. *Lecture Notes in Computer Science*, pages 317–338, 1999.

[GC89] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 202–210, New York, NY, USA, 1989. ACM Press.

[GH95] Frederic Gittler and Anne C. Hopkins. The DCE security service. *Hewlett-Packard Journal*, 46(6):41–48, 1995.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[HAB$^+$05] Alon Y. Halevy, Naveen Ashish, Dina Bitton, Michael Carey, Denise Draper, Jeff Pollock, Arnon Rosenthal, and Vishal Sikka. Enterprise information integration: successes, challenges and controversies. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 778–787, New York, NY, USA, 2005. ACM.

[HGPS99] John Hale, Pablo Galiasso, Mauricio Papa, and Sujeet Shenoi. Security policy coordination for heterogeneous information systems. In *Annual Computer Security Applications Conference*, pages 219–228, Phoenix, Arizona, USA, 1999. IEEE Computer Society.

[HR98] David M. Hilbert and David F. Redmiles. An approach to large-scale collection of application usage data over the internet. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 136–145, 1998.

[IBM08] IBM. Websphere Message Broker 6.1 Information Center: Publish/subscribe security. http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r1m0/index.jsp, 2008.

[Int08] Internet2. Shibboleth System. http://shibboleth.internet2.edu/, 2008.

[JH97] Dean Povey John and John Harrison. A distributed internet cache. In *In Proceedings of the 20th Australian Computer Science Conference*, pages 5–7, 1997.

[Jim01] Trevor Jim. Sd3: A trust management system with certified evaluation. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 106, Washington, DC, USA, 2001. IEEE Computer Society.

[Joh96] B.W. Johnson. *Fault-tolerant computer system design*, chapter An introduction to the design and analysis of fault-tolerant systems, pages 1–87. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[JPMH07] Michael A. Jaeger, Helge Parzyjegla, Gero Mühl, and Klaus Herrmann. Self-organizing broker topologies for publish/subscribe systems. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 543–550, New York, NY, USA, 2007. ACM.

[Kai98] P. Kaijser. A review of the sesame development. *Lecture Notes in Computer Science*, 1438:1–8, 1998.

[Kar01] Gunter Karjoth. The authorization service of Tivoli policy director. In *Annual Computer Security Applications Conference (ACSAC)*, pages 319–328, New Orleans, Louisiana, 2001. IEEE.

[Kar03] G. Karjoth. Access control with IBM Tivoli Access Manager. *ACM Transactions on Information and Systems Security*, 6(2):232–57, 2003.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, Jyvaskyla, Finl, 1997.

[KLVW04] Alexander Klemm, Christoph Lindemann, Mary K. Vernon, and Oliver P. Waldhorst. Characterizing the query behavior in peer-to-peer file sharing systems. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 55–67, New York, NY, USA, 2004. ACM.

[KLW05] Zbigniew Kalbarczyk, Ravishankar K. Lyer, and Long Wang. Application fault tolerance with Armor middleware. *IEEE Internet Computing*, 9(2):28–38, 2005.

[KS07] Angelos D. Keromytis and Jonathan M. Smith. Requirements for scalable access control and security management architectures. *ACM Transactions on Internet Technology (TOIT)*, 7(2), May 2007.

[KT86] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. In *ACM '86: Proceedings of 1986 ACM Fall joint computer conference*, pages 1150–1158, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[Lam71] Butler W. Lampson. Protection. In *5th Princeton Conference on Information Sciences and Systems*, page 437, New York, NY, USA, 1971. ACM Press.

[LCL04] W. K. Lin, D. M. Chiu, and Y. B. Lee. Erasure code replication revisited. In *P2P '04: Proceedings of the Fourth International Conference on Peer-to-Peer Computing*, pages 90–97, Washington, DC, USA, 2004. IEEE Computer Society.

[LGK$^+$99] Charlie Lai, Li Gong, Larry Koved, Anthony Nadalin, and Roland Schemers. User authentication and authorization in the java platform. In *Annual Computer Security Applications Conference*, pages 285–290, Phoenix, Arizona, USA, 1999. IEEE Computer Society.

[LPL$^+$03] Markus Lorch, Seth Proctor, Rebekah Lepro, Dennis Kafura, and Sumit Shah. First experiences using xacml for access control in distributed systems. In *XMLSEC '03: Proceedings of the 2003 ACM workshop on XML security*, pages 25–37, New York, NY, USA, 2003. ACM.

[Maz04] Paul J. Mazzuca. Access control in a distributed decentralized network: an XML approach to network security using XACML and SAML. Technical report, Dartmouth College, Computer Science, Spring 2004.

[MD99] R. Mc Dougall. Availability-What It Means, Why It's Important, and How to Improve It. *Sun Blueprints Online*, 1999.

[Mer05] Mercury. Diagnostics for SAP solutions: a performance lifecycle white paper. white paper, September 2005.

[MH06] John Markoff and Saul Hansell. Google's not-so-very-secret weapon. International Hearald Tribune, June 13, 2006, 2006.

[Mic01] S. Microsystems. Java Message Service Specification, 2001.

[MMDV02] J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy. *Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication*, 2002.

[MT00] M. L. Markus and C. Tanis. *The enterprise systems experience-from adoption to success*, pages 173–207. Pinnaflex Education Resources, Inc, Cincinnati, OH, 2000.

[Mul93] Sape Mullender, editor. *Distributed systems (2nd Ed.)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[ND97] V. Nicomette and Y. Deswarte. An authorization scheme for distributed object systems. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy (S&P'97)*, pages 21–30, Oakland, CA, 1997. IEEE Computer Society.

[Net00] Netegrity. Siteminder concepts guide. Technical report, Netegrity, 2000.

[NWO88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the sprite network file system. *ACM Trans. Comput. Syst.*, 6(1):134–154, 1988.

[OMG02] OMG. Common object services specification, security service specification v1.8, 2002.

[Ora08a] Oracle. Modernizing access control with authorization service. white paper, November 2008.

[Ora08b] Oracle. Oracle entitlements server: Programming security for web services. Technical report, Oracle, September 2008.

[Ora09] Oracle. Maximizing portal application performance. white paper, April 2009.

[PB02] Peter R. Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 611–618, Washington, DC, USA, 2002. IEEE Computer Society.

[Pha03] Pharm. Java TPC-W implementation distribution. http://www.ece.wisc.edu/ pharm/tpcw.shtml, 2003.

[PLZ03] A. Padovitz, S.W. Loke, and A. Zaslavsky. Using the publish-subscribe communication genre for mobile agents. In *Proceedings of the First German Conference on Multiagent System Technologies*, Sep 2003.

[Pow94] David Powell. Distributed fault tolerance—lessons learnt from delta-4. In *Papers of the workshop on Hardware and software architectures for fault tolerance : experiences and perspectives*, pages 199–217, London, UK, 1994. Springer-Verlag.

[PS04] Jaehong Park and Ravi Sandhu. The UCONabc usage control model. *ACM Transactions on Information and System Security*, 7(1):128–174, 2004.

[RKCD01] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In *NGC '01: Proceedings of 3rd International Workshop on Networked Group Communication*, pages 30–43, London, UK, 2001. Springer-Verlag.

[RN00] Tatyana Ryutov and Clifford Neuman. Generic authorization and access control application program interface: C-bindings. Internet Draft draft-ietf-cat-gaa-bind-03, Internet Engineering Task Force, March 9 2000.

[RPS06] Venugopalan Ramasubramanian, Ryan Peterson, and Emin Gün Sirer. Corona: a high performance publish-subscribe system for the world wide web. In *NSDI'06: Proceedings of Networked System Design and Implementation*, pages 2–2, 2006.

[RRH06] Costin Raiciu, David S. Rosenblum, and Mark Handley. Revisiting content-based publish/subscribe. In *ICDCSW '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems Workshops*, page 19, Washington, DC, USA, 2006. IEEE Computer Society.

[SAB$^+$00] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content Based Routing with Elvin4. In *Proceedings of AUUG2K*, June 2000.

[San93] Ravi S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, 1993.

[Sar02] P. Sarbanes. Sarbanes-Oxley Act of 2002. In *The Public Company Accounting Reform and Investor Protection Act. Washington DC: US Congress*, 2002.

[SCFY96] Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[Sec99] Securant. Unified access management: A model for integrated web security. Technical report, Securant Technologies, June 25 1999.

[SL05] Mudhakar Srivatsa and Ling Liu. Securing publish-subscribe overlay services with eventguard. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 289–298, New York, NY, USA, 2005. ACM.

[Smi82] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[SMJ01] Andreas Schaad, Jonathan Moffett, and Jeremy Jacob. The role-based access control system of a european bank: a case study and discussion. In *SACMAT '01: Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 3–9, New York, NY, USA, 2001. ACM.

[SMLN$^+$03] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, 2003.

[SNvdH03] Anita Sarma, Zahra Noroozi, and André van der Hoek. Palantír: raising awareness among configuration management workspaces. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 444–454, 2003.

[SPLS$^+$06] Wolfgang Schroder-Preikschat, Daniel Lohmann, Fabian Scheler, Wasif Gilani, and Olaf Spinczyk. Static and dynamic weaving in system software with AspectC++. In *HICSS '06: Proceedings of the 39th Annual Hawaii International Conference on System Sciences*, page 214.1, Washington, DC, USA, 2006. IEEE Computer Society.

[SPMF03] J. Schonwalder, A. Pras, and J.-P. Martin-Flatin. On the future of internet management technologies. *Communications Magazine, IEEE*, 41(10):90–97, Oct 2003.

[SS75] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(6):1278–1308, 1975.

[SSL$^+$99] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *Proceedings of the 8th USENIX Security Symposium*, pages 123–140, Washington, D.C., 1999. USENIX Association.

[Sto04] Geoffrey H. Stowe. A secure network node approach to the policy decision point in distributed access control. Technical report, Dartmouth College, Computer Science, June 2004.

[Str07] Paul Strong. How Ebay scales with networks and the challenges. In *the 16th ACM/IEEE International Symposium on High-Performance Distributed Computing (HPDC'07)*, Monterey, CA, USA, 2007. ACM Press. Invited talk.

[SV01] Pierangela Samarati and Sabrina De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In *FOSAD '00: Revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design*, pages 137–196, London, UK, 2001. Springer-Verlag.

[SV04] Diane M. Strong and Olga Volkoff. A roadmap for enterprise system implementation. *Computer*, 37(6):22–29, 2004.

[TC09] Mahesh V. Tripunitara and Bogdan Carbunar. Efficient access enforcement in distributed role-based access control (RBAC) deployments. In *SACMAT '09: Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 155–164, Stresa, Italy, June 3-5 2009. ACM.

[TDV99] Renu Tewari, Michael Dahlin, and Harrick Vin. Design considerations for distributed caching on the Internet. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, page 273, Washington, DC, USA, 1999.

[TEM03] Mary R. Thompson, Abdelilah Essiari, and Srilekha Mudumbai. Certificate-based authorization policy in a pki environment. *ACM Trans. Inf. Syst. Secur.*, 6(4):566–588, 2003.

[TIB99] I. TIBCO. TIB/Rendezvous White Paper. *Palo Alto, California*, 1999.

[Til05] Henk C.A. van Tilborg. *Encyclopedia of Cryptography and Security.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[TPC02] TPC. TPC-W: Transactional web benchmark version 1.8. http://www.tpc.org/tpcw/, 2002.

[TS01] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[VAG07] Jaideep Vaidya, Vijay Atluri, and Qi Guo. The role mining problem: Finding a minimal descriptive set of roles. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 175–184, Sophia Antipolis, France, June20-22 2007. ACM Press.

[Vog04] Werner Vogels. How wrong can you be? Getting lost on the road to massive scalability. In *the 5th International Middleware Conference*, Toronto, Canada, October 20 2004. ACM Press. Keynote address.

[Vog09] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[VV00] John Viega and Jeffrey Voas. The Pros and Cons of Unix and Windows security policies. *IT Professional*, 2(5):40–45, 2000.

[WBR06] Qiang Wei, Konstantin Beznosov, and Matei Ripeanu. Cooperative secondary authorization recycling. poster at the *USENIX Security Symposium*, July-August 2006.

[WC98] D. Wessels and K. Claffy. Internet cache protocol (ICP), version 2, 1998.

[WCBR08] Qiang Wei, Jason Crampton, Konstantin Beznosov, and Matei Ripeanu. Authorization recycling in RBAC systems. In *Proceedings of the thirteenth ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 63–72, Estes Park, Colorado, USA, June 11–13 2008. ACM.

[WCBR09] Qiang Wei, Jason Crampton, Konstantin Beznosov, and Matei Ripeanu. Authorization recycling in hierarchical RBAC systems. Under review, 32 pages, 2009.

[WRB07] Qiang Wei, Matei Ripeanu, and Konstantin Beznosov. Cooperative secondary authorization recycling. In *Proceedings of the 16th ACM/IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, pages 65–74, Monterey Bay, CA, June 27-29 2007. ACM Press.

[WRB08] Qiang Wei, Matei Ripeanu, and Konstantin Beznosov. Authorization using the publish-subscribe model. In *Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 53–62, Sydney, Australia, December 10–12 2008. IEEE Computer Society.

[WRB09] Qiang Wei, Matei Ripeanu, and Konstantin Beznosov. Cooperative secondary authorization recycling. *IEEE Transactions on Parallel and Distributed Systems*, 20(2):275–288, 2009.

[XBH06] Zhiyong Xu, Laxmi Bhuyan, and Yiming Hu. Tulip: A new hash based cooperative web caching architecture. *The Journal of Supercomputing*, 35(3):301–320, 2006.

[YB97] Joseph W. Yoder and Jeffrey Barcalow. Archictectural patterns for enabling application security. In *Pattern Languages of Programming*, Monticello, Illinois, USA, 1997.

[YPG99] R. Yavatkar, D. Pendarakis, and R. Guerin. A framework for policy-based admission control. IETF RFC 2753, 1999.

[ZHS$^+$04] BY Zhao, L. Huang, J. Stribling, SC Rhea, AD Joseph, and JD Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on selected areas in communications*, 22(1):41–53, 2004.