ABSTRACT

ALTUNAY, MINE. Collaboration Policies: Access Control Management in SOA-based Dynamic Collaborations. (Under the direction of Gregory T. Byrd.)

Service-oriented architectures change the computing paradigm by providing easily accessible services and by promoting collaborations among the provided services. The services can be harnessed with other services to create more powerful services. Ideally, the end user expects to select from an existing service pool, mix-and-match services, and come up with original applications that are tailored to his unique needs.

A collaboration is a collection of services that harnessed together to achieve a common goal. During run-time, each service is expected to interact with multiple peer services. An interaction occurs in the form of a data exchange between two peer services. Although collaboration significantly helps tackling difficult problems, it also leads to the increased exposure of a service. First, the collaborations are often short-termed and dynamically built based on end-user's demands. Therefore, there may not be established trust relationships among peers. Second, during run time, a service becomes exposed to the all of the collaborative peers. The interactions within the collaboration are not isolated from one another. Instead, each interaction consecutively follows one another in order to propagate data among multiple parties. As a result, a service is not only exposed to the peers with which it directly interacts, but also exposed to other peers due to indirect interactions.

We approach the access management from a service owner's perspective. We first study the type of interactions that are present in a collaboration. Based on the identified interaction types, we discuss the security threats that can arise with each interaction type. Our access control model aims to mitigate these security threats. Our access control model is designed to

evaluate a collaboration context, and it recognizes the multitude of information present within a collaboration context: varying interactions, different peers engaged in these interactions, and the actions taken by each of these peers.

In order to express access requirements from a collaboration, we designed collaboration policies. A collaboration policy contains access rules that are specified to evaluate the collaboration context. A service owner can associate each access rule with a specific interaction type. As a result, different peers with different interaction types are applied against different access requirements. In other words, our access control model varies access requirements from a collaborative peer depending on the collaboration context.

We encompass our work inside a framework. We develop a system architecture where each service that is invited into the collaboration can use its own collaboration policy to reach a decision. These evaluations are carried out as peer-peer trust evaluations. Our framework provides a message infrastructure that is used to carry out these evaluations. Moreover, the results of the security evaluations are collected and are used to determine the feasibility of the collaboration. We determine a collaboration is feasible when each collaborative service is willing to join the collaboration as a result of its security evaluations.

Our work aims to provide a secure and autonomous computing environment, and it aims to promote collaboration among services. We do this by enabling service owner’s with necessary means to protect themselves, and by encompassing these decisions into a framework.

COLLABORATION POLICIES: ACCESS CONTROL MANAGEMENT IN SOA-BASED DYNAMIC COLLABORATIONS

by
MINE ALTUNAY

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

COMPUTER ENGINEERING

Raleigh, NC

2007

APPROVED BY

Eric Rotenberg	Douglas S. Reeves
Ralph A. Dean Co-Chair of Advisory Committee	Gregory T. Byrd Chair of Advisory Committee

BIOGRAPHY

Mine Altunay was born in Turkey, on September 9, 1979. She attended Bursa Science and Math High School, and later graduated from Bilkent University with a Bachelor of Science Degree in Electrical and Electronics Engineering, 2001. Immediately after graduation, she moved to the U.S.A. to purse graduate studies in Computer Engineering at North Carolina State University.

ACKNOWLEDGEMENTS

There are a few people, without whom I could not have completed this body of work. I would like to take this opportunity to express my deep gratitude and thanks to those. Dr. Greg Byrd and Doug Brown contributed immensely to my education, not only in academic terms, but also other aspects of life. I cannot thank them enough for their availability, accessibility and patience. My family was the strongest driving force behind me that enabled me to complete this work; without their support, I could not have possibly been where I am today. My mother, my aunt, Gulcan Ozer, my uncle, Kadir Ozer, and my sister are the people that I cannot thank enough. Sarah Timberlake was not only a friend, but she was my extended family here; she is no different than a sister in my heart.. Also, I like to thank my aunt for making me believe that I can achieve anything I want (and also I can graduate within 6 short months). I am very blessed that I have such great role models and have such a wonderful family.

There are several friends, family members, professors, and fellow graduate students that helped me in so many ways. I cannot list all the names in this restricted space, but I am thankful to all of them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1: Introduction

The service-oriented architecture [W3C04] provides a computational environment that is not constrained by geographical or organizational proximity. The computational environment contains various services that are drawn from different organizations and are provided for the end users. A service is the smallest building block of the service-oriented architecture, and each service provides a particular functionality. The end user has a variety of choices for his needs; he can select a single service, or combine multiple services in various ways.

The service-oriented architecture promotes collaboration among the services. Services are expected to cooperate and interact with each other, and they are harnessed together to create non-trivial applications. In order to fully realize service-oriented architectures, uniform patterns of interactions between services must be developed. To satisfy this need, bodies such as OASIS [OASIS] and W3C [W3C] have defined several standards. These standards define the interaction patterns among services at various layers: from the transport layer message exchange to the upper layer business execution logic.

Collaboration among services is achieved with the help of the Web-Services standards and service-oriented computing principles. Before the adoption of service-oriented architecture, each organization typically built applications with its own proprietary technologies. Interaction among different organizations and their applications were burdensome, if not impossible. Inter-organizational interaction usually required a significant amount of work and modifications to the existing applications. As a result, inter-organizational collaboration and interoperability were usually avoided or neglected.

The adoption of service-oriented architecture and the Web-Services standards has changed this situation. Applications that were once only accessed through proprietary technologies become services that are accessible via the Web-Services standards. The adoption of uniform practices results in interoperability across organizational domains. A service can still be implemented and built in domain-specific technologies; however, the Web-Service standards describe a uniform way of interacting with the specific service. As a result, a service can be exposed to the external world, beyond its own domain. The end user is no longer expected to be a member of the service's organization. Anyone who has the means to access the service and can bear the consequences of using the service (such as, paying a fee for the service) can become an end user.

This new computing paradigm triggered what we now call Web 2.0 [O'Reil05]. The term Web 2.0 points to the change from the earlier web (Web 1.0), where applications usually aim to disseminate static data and have one-way interaction with the end user. Web 2.0, on the other hand, distinguishes itself by allowing two way service-user and service-service interactions. Several organizations are already participating in Web 2.0 by providing their services. For example, Google, Yahoo!, and Flickr provide their services via open APIs. An end user can use the provided services in his own application, mix and match services, and custom-tailor his application to his own needs. The resulting end user application usually contains services drawn from different organizations, blending and mixing disparate services. This type of application is called a mash-up due to its generation process. For example, a simple mash-up application can combine Google Maps API with Yahoo shopping API to provide the end user with a more sophisticated shopping tool, which shows the goods and their locations with respect to the user’s location. The organizations that provide their APIs are content with this new usage scenario because their services are used in creative ways and even promoted to newer markets through the mash-up. Moreover, the end user’s creativity teaches them new usage scenarios for their services, hence, enhancing their services for the market trends in a faster manner.

The service-oriented architectures are instrumental not only for creating mash-ups, but for numerous other application domains, such as scientific applications in bioinformatics or physics domains. A single scientific application is challenging to be fully home-made, and can be realized through combining services from different organizations, companies, or different laboratories. Scientific applications benefit from the service-oriented architectures in the same way the mash-ups do; however, scientific applications are usually realized through more sophisticated technologies, workflow management tools [WFMC]. In our work, we focus more on the workflow management tools due to their wide-spread adoption for large collaborations and the established prior research on them. Although our work is independent of a specific collaboration technology, we show the details of our architecture on the basis of workflow management tools. (We discuss the workflow management tools in Chapter 2.)

Although different application domains benefit from the service-oriented architectures in different manners, their ultimate goal is the same: achieving collaboration among the services. The collaboration is a collection of services that work together to achieve a specific goal on behalf of the end user. The end user is regarded as the person or the entity that has triggered the collaboration. The collaboration of the services provides an advantage to tackle significant problems; however, it causes security issues, which are the main focus of this work.

In service-oriented architectures, a service has a large and a heterogeneous user pool. The service, in fact, is designed to reach end users that are beyond its organizational boundaries. This is significantly different from before the adoption of service-oriented architectures. Then each service or the application had a limited and pre-determined user list. The user list usually consisted only of the members of the application's own domain. Since the applications were rarely exposed outside of their own domains, limiting the user list to the members of the organization was viable.

The service-oriented architecture leads to increased exposure of services. A service is not only exposed to the end users outside of its home domain, but it also expected to interact with other services. The access control becomes more complicated because the end user and the other collaborative services belong to separate security domains. These domains do not share a pre-established framework for identification, authentication, and authorization. Furthermore, there may not be pre-established trust among these domains. Assuming existence of pre-established trust is not realistic because the services are combined in arbitrary ways and often on-demand. A collection of the services may include services owned by rival companies, or separated by corporate firewalls, or otherwise inhibited from working collaboratively.

Furthermore, a service that is involved in the collaboration is affected by the other collaborative services, even when these collaborative services are not directly interacting with the service. In other words, they can still indirectly interact with each other via other services. Security threats, such as viruses, Trojan Horses, or corrupted data, may travel to the service through the indirect interactions within the collaboration. As a result, it is insufficient to circumscribe the security evaluations to the collaborative services that are explicitly interacting with the service or to the end user. The service involved in the collaboration must assess the security threats introduced by several parties, including the end user and the collaborative services.

Managing access control for a single service within the collaboration is challenging. The security threats are introduced by multiple parties: the end user as well as the collaborative services. Moreover, each of these parties has a different interaction with the service throughout the collaboration. To address the security challenges, an access control mechanism must (1) have a model of the collaboration, (2) distinguish between the interactions occurring in the collaboration, and (3) evaluate the involved parties accordingly. These are non-existent in the current access control models, which are built for interactions between a single service and a single end user. They cannot assess and detect the security threats unique to a collaboration. Our work addresses this challenge. We identify and analyze the security threats associated with collaborations. We provide the necessary tools to eliminate the identified threats. Based on the characteristics of ad-hoc and dynamic collaborations, we have identified the following list of security requirements.

Peer-level mutual trust evaluations. Collaborations require several services to interact with each other. These interactions often lead to unconsidered security consequences, such as direct or indirect accesses to a service by its peer services. A service may have reservations about joining a proposed collaboration due to the unexamined interactions. In order to address these reservations, each service must be able to evaluate a proposed collaboration. Each service must evaluate the potential access requests that would occur throughout the collaboration and determine whether they constitute a security threat. This leads to peer-peer trust evaluations among the services and the end user. A collaboration framework that enables and incorporates the peer-level trust evaluations eliminates the unforeseen security violations, and reduces the reluctance towards service's participation into the collaborations.

Decentralized authorization framework. Each service participating in a collaboration may have domain-specific security policies and requirements that are confidential [KM03]. Thus, the collaboration framework should have a decentralized access control management such that each service must independently evaluate the collaboration and reach a decision over whether to join the collaboration. Moreover, the collaboration framework should not assume any prior knowledge about the security policies governing the collaborative services, because the collaborations are built dynamically, and services are not expected to reveal their security policies to other parties.

Context-based, collaboration-aware access control model. Classical identity-based models or the families of role-based (RBAC) [San96] and task-based (TBAC) [TS93] access control models assume that a service owner has prior knowledge of the user pool. This assumption is not adequate for today's highly dynamic, market-oriented web services paradigm, wherein the services are offered to anyone with the necessary credentials. Proposed access control models based on trust management [BFIK99] address this problem. However, trust management-based access control models still need to be incorporated with a high level abstraction that encompasses the requirements of multi-party collaborations. The new access control model must be designed for evaluating access requests based on the context of a collaboration.

Our work aims to promote dynamic, on-demand collaborations among services by addressing the access control issues. We aim to enable the services to protect themselves against the security threats that can occur within the collaborations. Each service is enabled to evaluate a proposed collaboration context and to make its own decision on whether to join. A service evaluates the proposed collaboration in terms of the access requests that are going to happen during the collaboration execution. When the service determines that these access requests are authorized and do not constitute a threat, it joins the collaboration. Otherwise, it declines the collaboration. Our work incorporates the services' responses into an access control management framework. Based on the services' responses, our framework determines whether the current collaboration is feasible for execution.

1.1 Contributions of the Work

Our work has two main contributions: it provides a service with necessary means to express and evaluate its trust requirements from a proposed collaboration *(collaboration policies)*, and it provides an access control management framework that takes these evaluations into consideration (Figure 1.1).

We developed an access control model (Chapter 3) tailored for collaborations. Our model can be used to evaluate arbitrary collaborations, and is independent of the technology that is used to create a collaboration. Our access control model views a collaboration as the collection of interactions among the services and the end user. The access control model interprets these interactions from the viewpoint of a specific service, which is protected by the access control model. The interpretation of these interactions, and hence the collaboration, is different for each service. For example, the same interaction may be interpreted and evaluated differently by two different services' access control models; this is

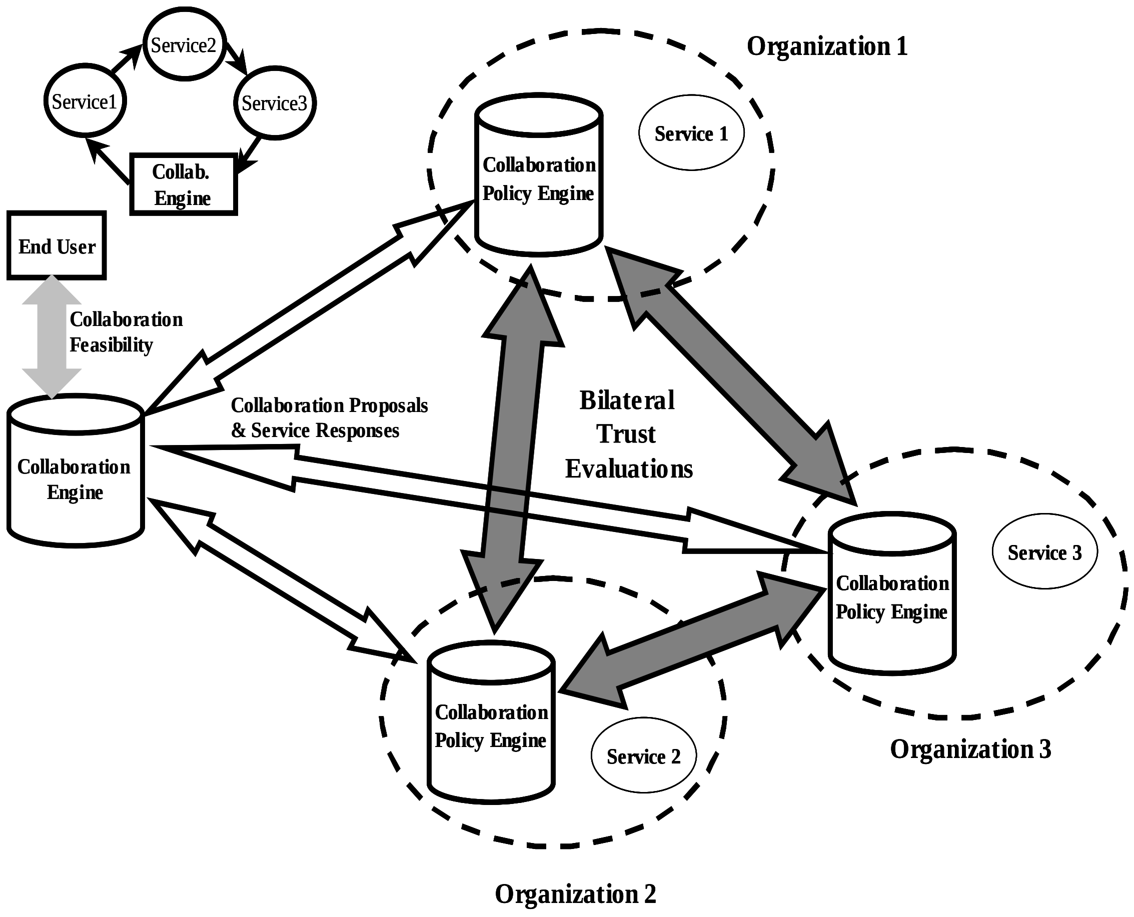

Service2
Service1
Service3
Collab. Engine
End User
Collaboration Feasibility
Organization 1
Service 1
Collaboration Policy Engine
Collaboration Proposals & Service Responses
Bilateral Trust Evaluations
Collaboration Engine
Service 3
Collaboration Policy Engine
Collaboration Policy Engine
Service 2
Organization 3
Organization 2

Figure 1.1 Our Framework. The contributions of our work: the collaboration policy engine at each service's security domain; the collaboration engine, a module that manages access control within the collaboration; the communication infrastructure among the autonomous modules.

due to the different roles played by each of these services within the collaboration. The model allows defining varying access requirements based on these interactions. The collaborative services or the end user that request access over a certain service can be applied to different access requirements based on the collaboration context and their interactions with the requested service. Moreover, our access control model deals with special cases that are likely to occur during collaboration, such as the delegation of rights and conflict of interest scenarios.

Based on our access control model, we developed a policy language: collaboration policies (Chapter 3 and Chapter 4). The collaboration policies enable services to express their trust requirements from the collaboration. A collaboration policy includes access control rules designed for different interaction types occurring within a collaboration. In order to ease the adoption of the collaboration policies, we built them as enhancements over an existing and widely-used access control language, XACML [XACML05]. We enhanced the XACML existing syntax in order to fit the collaboration policy syntax. We implemented the tools that can evaluate and enforce the collaboration policies. Likewise, these tools are built by using an open source XACML implementation provided by Sun [Sun05].

Our access control management framework (Chapter 5) views services as equal peers, and enables them to carry out their own trust evaluations with one another. Our framework defines uniform message patterns among the collaborative services and the end user so that the trust evaluations can be carried out. Our framework uses the result of the peer-peer trust evaluations in order to determine whether the collaboration is feasible for execution.

Our work contributes to the services and to the end users who want to build collaborations and make use of the service-oriented architecture. We enable the services to conduct their own security evaluations before joining a collaboration. Thus, our work increases the services' willingness to participate into the collaborations. Services that find the collaboration insecure refuse to join. On the other hand, we enable the end user to determine the feasibility of his proposed collaboration. The end user is saved from trying to execute a collaboration that was not agreed upon by all parties; thus, he is saved from run-time security failures.

In the rest of this paper, we first present the background of our research and the related literature in Chapter 2. We then discuss our access control model and policy model in Chapter 3. Chapter 3 also present the syntax and implementation details of collaboration policies. We continue with a special case that can occur within a collaboration: delegation of credentials. Chapter 4 discusses how we deal with delegation of credentials within a collaboration. We present our policy evaluation mechanism in Chapter 5. Our framework that encompasses our work is discussed in Chapter 6. In Chapter 7, we present the run time performance results of our framework and discuss the conclusion from the data collected. We conclude in Chapter 8.

Chapter 2:
Background and Related Work

In this chapter, we first present the underlying assumptions of our work and familiarize the reader with the necessary background material. Then, we present and discuss prior research that is closely related to our work.

2.1 Background and Assumptions

A collaboration can be realized via many technologies. Mash-up applications typically use AJAX [AJAX], which uses JavaScript as the glue code between the services. Scientific applications, on the other hand, employ workflow management tools that have more formal and sophisticated execution environments. The workflow tools, for example, provide graphical-user interfaces that allow for selection of services, and generate documents explaining the combination of the services and their execution. Both approaches embody the service-oriented architecture and employ the same Web-Services standards; however, they realize them through different technologies.

Our work addresses the service-oriented computing principles and the existing standards, and we do not limit ourselves to any specific technology. However, we focus more on the workflow management tools, due to their widespread adoption and the breadth of existing literature.

Workflow management tools model a collaboration as an ordered collection of tasks. A workflow task represents the smallest unit of work that must be accomplished. When all the tasks are accomplished, the collaboration reaches its goal. In a service-oriented workflow, a service is assigned to accomplish each task. The data flows among the workflow tasks such that once a service accomplishes its task, the service forwards its output to another service that is responsible for the next task. As a result, a collaboration can also be modeled as an ordered set of interactions among the services. For the rest of our paper, we refer to and illustrate a collaboration as a directed acyclic graph. (Our implementation currently does not support directed cyclic graphs; we leave that as future work.) Each node of the graph indicates a service. The arc between two nodes indicates the data exchanged between two services. The direction of the arc is same as the direction of the dataflow. When we refer to an interaction, we mean a specific data exchange between two services. An interaction is represented by an arc of the graph.

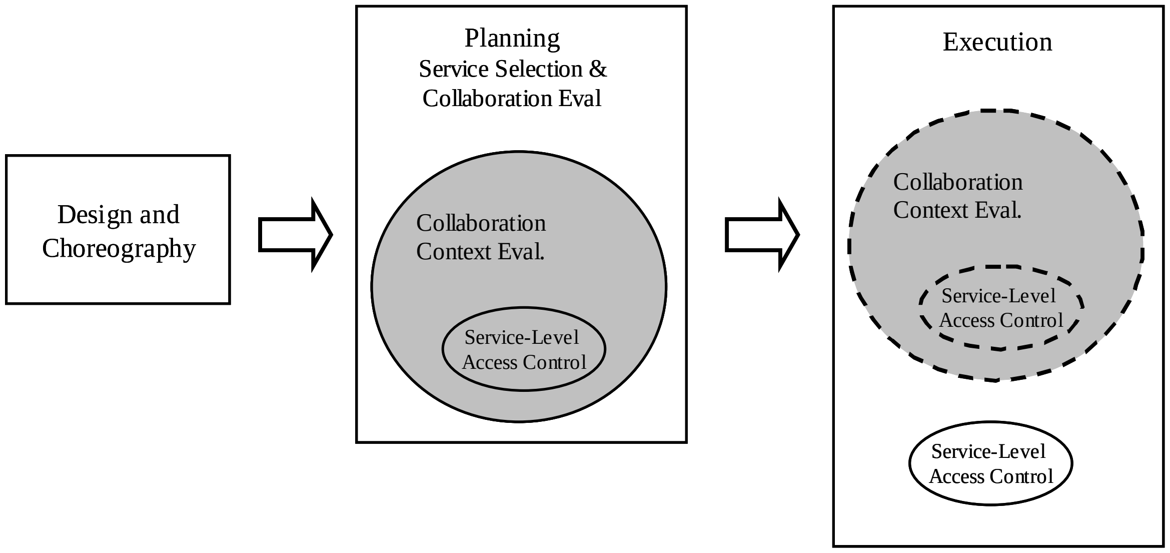

Design and Choreography
Planning
Service Selection & Collaboration Eval
Collaboration Context Eval.
Service-Level Access Control
Execution
Collaboration Context Eval.
Service-Level Access Control
Service-Level Access Control

Figure 2.1 The phases of workflow construction and execution. The collaboration context evaluation is performed in planning stage. It includes all peer-peer trust evaluations. Service-level access control indicates the access control checks that are done when a service receives a standalone "traditional" access request. The collaboration context evaluation includes the service-level access control checks in addition to the other peer-peer trust evaluations that are not part of the service-level access control checks. In execution phase, the result of the collaboration context evaluation can be verified; however, it does not have to be performed again, indicated by the dashed lines. In execution time, only service-level access control checks are performed.

Figure 2.1 shows a very high-level view of the stages involved in constructing and executing a workflow. The management of the stages is performed by a workflow engine. In the Design and Choreography phase, a complex business process or application is expressed in terms of interacting tasks. In the Planning phase, the services which meet the operational requirements of the design are chosen, and the interactions among the services – in other words, the required collaborations – are identified. In the Execution phase, services perform their assigned tasks, and messages are passed among them to carry out the workflow operations.

In many workflow environments, security evaluations are delayed until the execution stage. A service evaluates its security policies when it is invoked at run time. The requested service evaluates its invoker, which is the service that has accomplished the preceding task, and the requested service determines whether to grant access. There are multiple problems with this approach. First, if the requested service refuses access, it leads to the breakdown of the workflow at execution time, and eventually forces re-planning and re-execution. For large workflows with numerous services, re-executing the workflow until it successfully completes is unaffordable, not to mention very inefficient. The second problem is that the security evaluation does not inform the services about the collaboration context. An access request only contains information about the requestor service and the requested service: the other services that are present in the collaboration are not conveyed in the access request. Therefore, the requested service is not aware of the interactions that have led to the current request, nor the interactions that must occur afterwards. Left unevaluated, the remaining interactions present in the collaboration can cause security breaches, such as data propagated from or into un-trusted domains, Trojan horses, or conflict of interest scenarios.

As a solution, we incorporate the security evaluations into the planning stage (Figure 2.1). Our security evaluations do not involve the discovery and selection of suitable services. We assume that discovery and service selection has already been performed by the planning engine. After the planning engine finds suitable services that meet the functional requirements of the collaboration, our framework receives the name of the selected services and the collaboration graph as its inputs. Our work provides the means for conducting security evaluations among the selected services; therefore, it addresses the access control management for a group of selected services. In our framework, services are presented with the collaboration context during the planning stage. The collaboration context represents the interactions occurring in the collaboration and the services that are involved in these interactions. For a requested service, the collaboration context not only includes the explicit interaction between the requestor service and the requested service, but also it contains other interactions that leads to or succeed this explicit interaction. If the same service is involved in multiple separate interactions, the collaboration context contains all the interactions that can affect the security of a service. Each service receives a different collaboration context because each context is circumscribed to the interactions that affect the security of a specific service. In Chapter 6, we explain how we create the collaboration context and communicate this with the services.

Based on the collaboration context, the service can distinguish between its peer services, and it can evaluate them accordingly. Each service applies its collaboration context against its collaboration policy. During this evaluation, a service conducts peer-peer trust evaluations with the services within its context. The trust evaluations allow a service to determine whether to provide access to its peer or not. For example, a service may deny access to another service if one of the preceding interactions is deemed to be insecure. These trust evaluations are preliminary authorization checks among the services.

When a service discovers that it would not grant access to any of its peer services, it declines the collaboration. The planning stage completes when all services agree on participating in the collaboration. Only then does the execution stage start. As a result, the execution stage has a higher chance of successful completion.

During the execution stage, the collaboration context can be evaluated again in order to ensure that the collaboration has not been changed since the planning stage. The execution stage is beyond the scope of our work; however, this can be accomplished if each service stores a copy of the collaboration context from the planning stage, and compares it with the access requests received at run-time. The service-level access requests during the execution stage (Figure 2.1) refer to the accesses that are made during the execution stage. These run time access requests do not carry any additional information about the collaboration context. They are traditional access requests in the sense that they only carry information about the requestor service and the requested service. Note that the service-level access requests are also represented inside the collaboration context; therefore, they are already evaluated during the planning stage. We recognize that the collaboration context can contain information that cannot be captured from the run time access requests. As a result, ensuring that the collaboration has not been changed between planning and the execution could become challenging. We leave this as an open question and later discuss it among our future work.

The final problem with the current security evaluation is that it is only unidirectional: the security evaluation is only performed by the requested service to determine whether the requestor service is authorized or not. It does not evaluate the requestor service's trust in the requested service. In other words, there is not a bilateral trust evaluation between the requestor and the requested service. This is a problem in a collaboration because neither service has a prior knowledge of each other. In fact, the services are collaborating with each other only because the collaboration owner selected them. Therefore, the services may not have an established trust in each other.

To remedy this situation, we allow bilateral peer-peer trust evaluations among the services. At the planning stage, each service receives a different collaboration context. Each service evaluates the interactions in the context against its collaboration policy. These interactions are different with respect to each service. (We discuss the interaction types in the next section.) As a result, each service can evaluate all of its peer services, and may also be evaluated by other peers.

Our work focuses on the planning stage, and does not make any modifications to the remaining workflow stages. We assume that a suitable service for each task is earlier found during the service discovery stage. We focus solely on the security evaluations among the selected services. Once we ensure that the selected services agree to the proposed collaboration, they can be bound to their tasks and the execution stage can start. However, the implementation of the service binding and the execution stage is beyond the scope of our work. We only forward the name of the services that agree to join the collaboration to the execution stage. The actual service binding and the execution occur in this stage.

We assume that a service is the provision of any kind of facility to the public, such as computing power, storage, or simple remote code invocation. A service is not limited to its domain boundaries. We assume that the services are exposed over a network, and utilize the current Web-Service standards such as WSDL [WSDL1.1] and SOAP [SOAP]. Each service's collaboration policy is private, and is not divulged to other services, or to the workflow engine. Each service has access credentials that can be evaluated by its peers for authorization and authentication purposes. These credentials are assumed to belong to the actual service owner. We model a service's credentials similar to the proxy credentials [Wel03] defined by Welch *et al.* such that a service has the same rights and privileges as that of its owner. For example, a service may invoke other services that its owner is authorized for. Furthermore, due to the heterogeneity of the services involved in the collaboration, we do not assume that a service has prior knowledge of other peer services that are present in the collaboration.

2.2 Interaction Types

Within a collaboration, each service interacts with a number of peer services. An interaction involves the data exchange between two services. The interactions among the services are crucial for the security evaluations. Therefore, below we examine these interactions in two different categories: direct and indirect. Later, we refine these two categories with respect to the direction of the dataflow: upstream and downstream. As a result, we introduce four different types of interactions: direct-upstream, direct-downstream, indirect-upstream, and indirect-downstream.

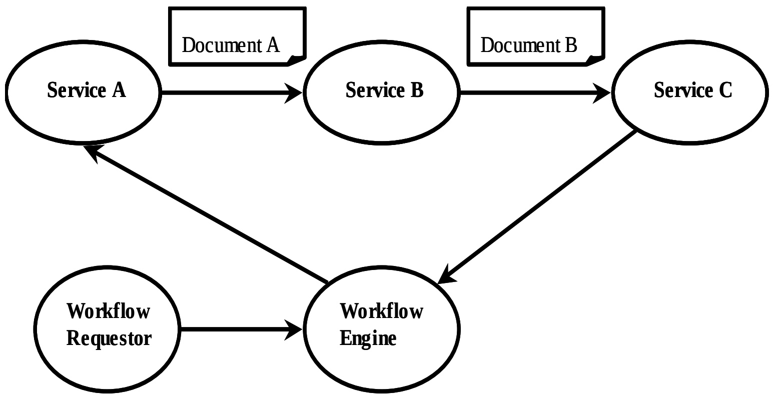


Figure 2.2 Collaboration scenario.

Direct interactions occur between the services that exchange data with each other without relaying the data through other services. Such services are called direct neighbors. For example, services A and B in Figure 2.2 have a direct interaction. Any direct interaction between services is a *bilateral relationship*, even when the dataflow seems to be one-sided. To illustrate this, refer to Figure 2.2, where each service accomplishes a unique task labeled by its name. The arcs between the services indicate the dataflow. The direct interaction between Service A and B is seemingly one-sided: Service A presents an input file to Service B for invocation and B determines if it trusts A for access. However, there are actually two relationships: (1) A determines that it trusts B and agrees to share a copy of its result file, and (2) B determines that it trusts A for invocation with the specified input file.

Both of Service A’s and Service B’s access requests involve risk. From A’s perspective, B could be a rival company with whom A is not willing to share its results; from B’s perspective, A could be a malicious user who sends a Trojan horse. Existing access control models such as TrustMaker [BFIK99], RBAC [San96] or password-based schemes, are geared towards assessing the trustworthiness of the requestor. The *reverse* trust evaluation –

i.e., the trustworthiness of the requested service from the requestor’s viewpoint – is not explicitly modeled. Instead, it is assumed that the invoking party implicitly makes a trust evaluation before launching its request. This implicit modeling does not work in a multi-party collaboration because a third party, the collaboration owner, selects the participating services and requests them to interact with each other. The selection of collaborating services does not necessarily equate to the existence of trust between the services. As we illustrate in above example, the collaboration owner has selected Service A and Service B to interact with each other. However, this does not guarantee that B does not possess any security threats to A, vice versa. As a result, the bilateral nature of direct interactions must be recognized, and interacting services must be allowed to perform bilateral authorization checks on each other [ABBD2-05].

Indirect interactions occur between services that exchange data with the help of intermediate services. The intermediate services relay the data between the interacting services. Two services with an indirect interaction are also called indirect neighbors. The Services A and C in Figure 2 are indirect neighbors. There are several reasons why indirect interactions must be carefully evaluated.

(1) Confidential documents or the results of a sensitive service are typically passed among several peers throughout a collaboration; thus even an indirect neighbor might have access to confidential data. Furthermore, partnership agreements and competition among businesses may prevent them from doing business with certain organizations. Even when such interactions are safe from a security standpoint, the higher-level business logic may forbid them. In addition, some peers in a workflow graph may introduce security threats to the other peers due to the security breaches within their own domains. An access request that

has traveled through an un-trusted security domain may infect the other peers that are on the same workflow path.

(2) Indirect neighbors can cause conflict of interest scenarios that cannot be caught by only inspecting the direct neighbors. An indirect neighbor involved with an access request may cause fraud and should not be allowed indirect access.

(3) A workflow chain may require or allow delegation of rights between services. Delegated rights may have to travel through several disparate security domains and may be handled by intermediate peers until utilized for an access request. A service that receives an access request with delegated credentials may want to evaluate the trustworthiness of the intermediate parties as well as the peer that launched the access request. Likewise, the riginal owner of the delegated rights may also put trust requirements on the intermediate parties in order to prevent abuse of its rights.

We further refine direct and indirect interactions with respect to the direction of the dataflow: *upstream and downstream* interactions. A service experiences an *upstream interaction* with another service when the first service is the recipient of the data exchange and the second service is the sender. For example, Service B has a direct-upstream interaction with Service A in Figure 2.2. On the other hand, when the data flows out of the first service into the second service, the first service has a *downstream interaction* with the second service. The Service A in Figure 2.2 has a direct-downstream interaction with the Service B. We refine both direct and indirect interactions with respect to the direction of the dataflow, resulting in four kinds of interactions: *direct-upstream, direct-downstream, indirect-upstream, indirect-downstream.*

Refining an interaction with respect its dataflow is important for a few reasons. First of all, the dataflow indicates the sender and the recipient services in an interaction. Although the two services participate in the same interaction type, the roles they play in these interactions are different. Typically, the sender service (Service A in Figure 2) is the requestor that invokes the requested service. The recipient (Service B) is the requested service that would accomplish the next task in the workflow. Second, due to the flow of the data, the security threats associated with the services are different. Service A, due to the downstream interaction, is concerned about revealing its output document to Service B. B is concerned about allowing A to invoke. Therefore, informing a service only about the interaction type such as direct or indirect is not sufficient. The service must also be informed about the direction of the dataflow because based on the direction of the dataflow, the service's access requirements from its interaction partner differ. Later, when we introduce our access control model, we discuss how different types of interactions can be evaluated.

2.3 Security Issues and the Related Work

We present the related work in four sections. First, we discuss the existing workflow authorization management frameworks. We then discuss the delegation of rights and the conflict of interest, respectively – their characteristics within a multi-party workflow and the shortcomings of the existing work to capture these characteristics. Conflict of interest and delegation of credentials are well-studied research problems in the literature; however, most of the existing work studies them either outside of the workflows, or within established homogeneous communities. We believe that these two research problems frequently occur in collaborations, and they have special characteristics due to the multiple autonomous security domains involved. We demonstrate that these characteristics cannot be captured within existing models. Finally, we discuss the implications of multi-party collaborations on the current business models and regulations. We present how current government regulations and legal contracts may hold each collaborator responsible for the consequences of their interactions with other parties.

2.3.1 Workflow Authorization Management

There are several workflow authorization frameworks proposed [AH96, HA99, Kno00]. These frameworks are designed to manage workflow authorization within a single security domain, such as within a large organization. The existing work focuses on selecting suitable services or human subjects that can perform the workflow tasks. A central workflow authority defines the access rules that must be satisfied by a candidate service or a human to perform a task. Since these frameworks have a single-domain model, they omit peer-to-peer trust relationships, which is one of the focuses of our work. Every workflow participant is a member of the same security domain, and there is established trust between the participants. Furthermore, there is a central security governing the entire security domain. The participants do not have separate policies to protect themselves; the participants are supposed to trust and follow the central security policy. The above frameworks target to assign the services to the tasks with respect to the central security policy. Another contribution of the above frameworks is to synchronize accesses to workflows tasks with respect to the workflow progression. In other words, no workflow participant can execute a task before the workflow reaches a certain state.

Other approaches by Bertino [BFA97], Tan [TCG04], and Hung [HK03] are similar to the above frameworks in that they use authorization constraints to define which services may be used for executing a workflow task. They rather focus on extending RBAC models to express the authorization constraints over the workflow tasks.

Kang [KPF01] recognizes the inter-organizational, distributed nature of new-generation workflows and adopts a multi-domain security model. However, Kang's model requires a pre-established trust relationship between the disparate security domains. Kang uses the RBAC model, and assumes that a central workflow engine can access each service's security policy so that the workflow engine can determine which services may interact with other services. This type of preparatory communication results in pre-established relationships among the security domains. In other words, before the collaboration is even conceived, the security domains communicate with each other in order to gain an understanding of one another's security policies. The main drawback of Kang's work is that it does not allow building dynamic workflows, where the workflow engine should not require prior knowledge about the internal security policies of participants.

Koshutanski [KM03] proposes an authorization framework for ad-hoc workflows. Based on the collaboration owner's request, a workflow engine dynamically selects suitable services to perform tasks. Koshutanski assumes that none of the services publicly announces their access control policies. Koshutanski's key contribution is an authorization mechanism between the collaboration owner and the services. Instead of sending access policies at the data-level (i.e., publicly exposing the policies), each service sends a mobile process to the collaboration owner. The mobile process must be executed in the owner's domain, and it determines whether the owner is authorized to access the service. The mobile code is assumed to access the credentials stored within the owner's domain so that an access decision can be reached. The reliance on mobile code introduces other security issues, such as how the workflow owner’s domain can verify the mobile code, and how the mobile code should retrieve all the required credentials. Koshutanski’s framework neglects the peer-level trust evaluations between services, and focuses on authorizing the owner to each service individually.

The WAS framework [KKHK03] adopts a multi-domain security model and targets grid-based computing environments. The WAS framework assumes pre-established trust relationships between the domains. The WAS engine functions as a trusted third party between the service owners and the collaboration owner. Each service owner informs the WAS engine about his service, and delegates the access rights over the service to the WAS engine. Upon building a workflow, the WAS server determines which services are participating in the workflow. The WAS engine delegates the access right associated with these services to the parties that would request access during the execution stage. WAS framework can function well in small grid communities, where prior trust and community-wide policies can be established. However, it is not well suited for ad-hoc distributed workflow models.

Shehab [SBG05] also addresses the security issues of multi-domain collaborations. This framework assumes that there are cross-domain role mappings, and each domain is aware which mappings are forbidden or authorized. In addition, formation of the cross-domain mappings is not within the scope of their work, and assumed to be handled priorly. Their approach is focused on tracking the history of an access request, which is a list of the domains that are visited until the access request. All the domains that are involved with an access request can be evaluated at access decision time. Their approach is similar to ours in that they allow a workflow participant to check the direct or indirect domains involved with an incoming access request. However, they differ in their assumption of cross-domain role-mappings. They state that domains must somehow consult each other and generate role-mappings. Our proposed framework does not require pre-defined mappings between domains, formation of which require exposure of policies between the domains and an established trust relationship. Instead, our approach focuses on expression and evaluation of collaboration policies such that, without public exposure, each domain can examine the history of an access request. As a result, their framework has a rather straightforward approach at detecting illegitimate access requests that traveled through unauthorized domains; each service has a complete list of forbidden mappings between domains and an access request that includes a forbidden link is denied access.

2.3.2 Conflict of Interest

In collaborations, conflict of interest usually occurs due to the high number of collaborative services, especially when the services perform sensitive tasks. A malicious service, which is assigned to perform multiple sensitive tasks, may deliberately modify the outcome of one of the tasks to provide benefit. Furthermore, a malicious service may deliberately provide corrupted information to its peers in order to affect the outcome of other sensitive tasks. Therefore, the conflicts may affect the collaboration owner, as well as the services. As a result, the detection and prevention of conflicts within workflows remains a significant research problem.

One of the early works in the conflict of interest area is that of Saltzer and Schoreder [SS75]. They argued that assigning multiple entities to specific tasks reduces the likelihood of fraud. This argument later led to a concept known as separation of duties. Most of the latest work [CW87, BFA99, San88, San90, HQ03, KS01, BE01] in this area studies the conflict of interest problem within a single security domain, such as a large organization. The services or humans perform the tasks, and they all belong to the same domain. The permissions to execute workflow tasks are centrally defined and regulated; thus, the complexity of the problem is reduced, and unauthorized accesses can be caught centrally. There are two reasons for the reduction in the complexity: first, the services belong to the same security domain and are assumed trust one another; second, a single central policy per collaboration is sufficient for preventing the conflicts. A service does not have its own policy for protecting from the conflicts. Rather, the service trusts the central policy that its successful enforcement would also protect the service's interest. The work in this area focuses on assigning services to workflow tasks such that the there would be no conflict of interest among the services and the collaboration owner. Typically, a central security policy dictates how to select services for each workflow task.

We, on the other hand, study the conflicts within a multi-domain model. Our multi-domain model assumes that each service has a distinctive view of a conflict and accordingly has a different policy to protect itself. These views and policies may not overlap or comply with that of other domains. This is radically different than single-domain approach, which assumes a common understanding of a conflict and accordingly requires a single policy. We, on the other hand, focus on the conflicts that may arise in a workflow participant's domain due to the unexamined direct or indirect interactions with other services. Such conflicts are not necessarily considered as violations by other services or by the collaboration owner. Note that above approaches have a central policy that defines what a conflict is from the collaboration owner's perspective (i.e. the large organization). When the services are selected according to the central policy, there should not be any conflicts at all.

We believe that the single-domain approach is fitting when there is an established trust relationship between the involved parties and they agree on what constitutes a conflict. Each party would be assured that its interests would also be protected by enforcement of the central policy. However, in the absence of an established trust relationship, this cannot be guaranteed. Each workflow participant must enforce its own collaboration policies to ensure that the proposed workflow does not violate their policies.

In particular, Sandhu [San88, San90] proposed that the roles that are allowed to execute tasks (called transactions in the original paper) must be specified in such a way that a user may be allowed to execute only a single task in a workflow. Bertino [BFA99] proposed a formal language that defines the access control constraints within a workflow context. Bertino's framework allows a security officer to define the conflicts between tasks and how these conflicts are reflected as constraints on the task execution rights given to the roles. (Roles here refers to the role types defined within the organization. Each role has assigned permissions to access the organization's resources. Services as well as human participants are assigned to specific roles.) During the execution of a specific workflow instance, the history of events in a workflow is captured, and the services' rights to execute tasks are dynamically adjusted based on the history of events and the workflow constraints. For example, a typical constraint is that a manager role that has accessed a task to create a loan application can no longer access another task that determines the outcome of the application in the same workflow instance.

Knorr [KS01] designed a workflow management tool that helps security officers to analyze the consistency of access constraints through graphical modeling. His Simple Separation of Duties Language (SSoD Language), which is used to express constraints, is similar to the constraint language proposed by Bertino, in that both languages express constraints based on task abstractions.

Botha [BE01] argued that identifying conflicts between tasks is not sufficient to detect all possible conflict scenarios. He proposed that in addition to specifying conflicting tasks, conflicting entities must also be specified. Conflicting entities would capture scenarios such as: John cannot execute a task that is conflicting with another task that has been performed by Jack because John and Jack are brothers.

Huang [HQ03] explicitly uses conflict of interest classes, where each conflict class identifies the conflicting tasks of a workflow. A workflow participant is allowed to perform a single task from each conflict class. At the planning stage, suitable services are selected with respect to the conflict of interest classes defined for a workflow. A web service that is functionally capable of executing two tasks of a workflow is selected for only one of these tasks, when the two tasks are in the same conflict class. Huang demonstrates how conflict classes within a workflow can be expressed in WS-Policy, and how the security policy of the workflow is conveyed so that the service discovery and selection would incorporate the conflict classes.

Huang's approach is closest to ours in that it assumes multiple security domains. Each service belongs to a different security domain. However, his approach omits the examination of peer-level conflicts, and adopts the view of a collaboration owner to prevent conflicts. In other words, Huang's work assumes that there is a single policy to detect the conflicts. This policy is defined by the collaboration owner, and it only reflects the view of a conflict from the collaboration owner's perspective. This approach neither enforces the individual policies of the services, nor protects them against conflicts among the services.

2.3.3 Delegation of Rights

Delegation of rights usually occurs within a multi-party collaboration. A peer service (the delegator) delegates its credentials (i.e. rights) to another service (the delegatee). The delegation occurs typically between peers that have established trust relationships. The delegation is performed in order to prevent an authorization failure and to complete the workflow execution. The delegatee lacks the necessary credentials to access another service, and the delegator provides its own credentials to prevent the access failure. It is possible that multiple intermediate services may relay the delegated credentials until they reach the delegatee.

Several delegation frameworks assume a single-domain security model and specify centrally enforced delegation policies [ZAC01, ZAC02, WK05]. This model is insufficient to meet the needs of a multi-domain model, which is the focus of our work.

First of all, each party (i.e. the delegator, the delegatee, the requested service) that is involved with a delegation may belong to disparate domains, and has a separate delegation policy. From a delegator's standpoint, the delegation policy must specify the access rules for prospective delegatees and the treatment of received rights by the delegatees, such as re-delegation depth and width. Conversely, from a service owner's standpoint, the delegation policy must specify whether access with delegated credentials are accepted, and the access rules over the delegatees. In other words, the service owner must determine whether the delegatees constitute a security threat against the service's domain. A delegator, on the other hand, is more interested in preventing the abuse of its rights.

In a multi-domain security model, these policies may not overlap, or may contradict with each other. For example, a delegatee that is evaluated as trustworthy by a delegator may be found to be un-trustworthy by a service owner. Likewise, a service owner may accept an access request from a delegatee that is abusing the delegated rights, and violating the policies of the delegator, as long as such violations do not cause a harm to the service's domain.

Unless disparate security domains have access to each other's policies, they have no means to know or enforce each other's policies. Furthermore, even when we assume they have access to each other's policies, even though this is an unrealistic assumption for ad-hoc distributed workflows, they may not have sufficient motivation to protect each other's best interest.

Zhang's policy language (RDM2000) [ZAC01, ZAC02] introduces a set of relations into RBAC96 and RDBM0. The can_delegate relation defines the delegating role and the conditions on the delegated role along with the maximum re-delegation depth. Similarly, can_revoke relation specifies who is eligible to revoke a delegated right. A central security officer must specify these relations and enforce them each time a request for delegation and revocation is made. Note that a delegator must send a request to the central security officer to delegate its rights to a delegatee, an approach called administrator-directed delegation [LN99].

Wainer [WK05] separates the object rights from delegation rights: the former indicate the access rights on an object, whereas the latter show the right to delegate the object rights to another entity. A central authority decides to accept delegations and controls re-delegations by checking three properties: (1) the delegator must have the right to delegate, (2) the delegatee must satisfy all the constraints in order to receive the delegated rights, and (3) the generic constraints must not be violated. (These generic constraints specify the additional organizational policies on the delegation.) This approach, like Zhang's, is administrator-directed.

Both Wainer and Zhang’s approaches adopt the single-domain model and specify a single delegation policy per domain. Since all the parties belong to same domain, a central security officer can enforce the policy and prevent any violations.

Other delegation frameworks with multi-domain security models [KFP01, PWFK+02] suffer from their dependence on pre-established trust relationships between the domains.

Kagal’s framework [KFP01] models two distinctive security domains: the delagator’s and the service owner’s. It assumes that a delegatee and a delegator reside in the same security domain, and are therefore controlled by the same central security officer. Kagal adopts an administrative directed approach. Each time a delegatee requires access to a remote service, the security officer in the delegatee’s domain examines the request, and sends an approval message to the security officer in the requested service’s domain. The security officer at the requested service’s domain must verify that the remote access request is indeed approved by the delegatee’s security officer.

A similar approach is presented in the Community Authorization Service (CAS) [PWFK+02], where a central CAS server delegates access rights to Virtual Organization (VO) [FKT01] members. Through these delegated rights, the members may access distributed resources of a VO. The VO resources may reside in any organization that is a member of the VO. Each time CAS server delegates access rights to a VO member, the delegated rights are inserted into the delegatee’s X.509 credentials as extensions. When an access request occurs, the security officer in the requested resource’s domain checks the delegatee’s X.509 credential, and grants access to the object if it complies with the delegated rights.

2.3.4 Business Regulations and Partnerships

The same rules governing brick and mortar businesses also govern the businesses that offer their services in cyberspace. Businesses pick their collaborators based on their partnership logic, rivalry, and government regulations. Although, currently, business functions such as selecting partners, contract building, and quality of service enforcement are not fully automated, the growing support technologies will soon turn these obstacles into added benefits of doing business on the Internet. Security and trust are two inseparable decision factors shaping businesses' everyday functions. The lack of necessary tools to incorporate security and trust into the collaboration decisions prevent building dynamic collaborations. Businesses are often reluctant to join a collaboration for fear of interacting with parties that are rivals or blacklisted organizations.

Government regulations, such as Health Insurance Portability and Accountability Act (HIPAA) [HIPAA03], put further restrictions on businesses' choice of partners. For example, HIPAA defines every organization that exchanges confidential patient reports with each other as partners and holds them responsible for each other's actions. Another example is licensing contracts that strictly define how to distribute and use a service [BP02]. A licensee that uses the licensed service must be careful about how the result of service is propagated and used in a collaboration chain. Some licenses may strictly forbid using their services in certain countries or for certain purposes. It is the licensee's responsibility to ensure that none of the rules of engagement are violated.

The fear of unknowingly breaking regulations that may lead to litigation or lost profit can prevent businesses from joining dynamic on-demand collaborations. Without dynamic heterogeneous collaborations, the true benefit of service oriented computing, which is to help discovering and connecting with previously unknown services, may never be realized.

Our collaboration policy-based framework allows businesses to define access control rules on their direct or indirect neighbors so that an illegal chain of interaction can be detected at the workflow planning stage. A service may put additional restrictions on its partners' identities based on the level of interaction required and the service being offered. This added assurance would promote the willingness of service owners to join collaborations.

Chapter 3: Collaboration Policies

A collaboration policy is used to determine whether or not to provide a service to a proposed collaboration. In order to accomplish this, the collaboration policy models the collaboration as a collection of collaborative peers that are interacting with each other to achieve a common goal. The collaboration policy [ABBD05] states the trust requirements sought from the collaborative peers. The collaborative peers and the interactions among these peers are evaluated against the stated trust requirements.

Definition 1: Collaboration Policy (CP) is a collection of access control rules, each of which represents the trust requirements sought from a collaborative peer included in a proposed collaboration. The peers are applied to the access rules with respect to their interactions inside the collaboration. A policy decision is “permit” or “deny”, where permit means joining the collaboration, and deny means declining the collaboration. Boolean logic operators are used to combine the results of access rules.

For the remainder of this chapter, we discuss our access control model and accompanying collaboration policy model, then present the syntax of collaboration policies, and finally discuss the implementation of the collaboration policies. The discussion of our access control model and the policy model aim to explain why we need a model specifically designed for services provided to the collaborations. The succeeding sections present the complete policy syntax and the implementation details.

3.1 Access Control Model

Before delving into our discussion, let us introduce some terminology. A subject is an entity that requests access. A subject can also be called a requestor. An object is a resource that is being requested. An action is an activity that is to be performed on the object. A collaborative peer is a service. A collaborative peer is being proposed to join the collaboration. The collaborative peer uses its collaboration policy to determine whether to accept the proposal. When the collaboration is built and enacted, the collaboration is realized by the interactions among the collaborative peers. During the collaboration, a collaborative peer can act both as a subject and an object. When the collaborative peer requests access to another peer (for example, invoking another peer, or sending data, or consuming output data of another peer) within the collaboration, the collaborative peer acts as a subject, while the requested peer acts as an object.

Our access control model is designed to evaluate an “access request” that represents an invitation to the collaboration. When the service “grants the access request”, the service joins the collaboration. When the service “denies the access request”, it declines the invitation. The “access request”, in our model, represents the collaboration and all the interactions contained within the collaboration. By granting access to the collaboration, all of the collaborative peers that must interact with the provided service are granted access. In other words, the “access request” represents all of the accesses that are going to be performed by the collaborative peers once the collaboration is built. We call this “access request” a *collaboration request*. This is not just a syntactical change; it reflects that a collaboration request differs from a traditional access request. A traditional access request represents a single interaction between a subject and an object, whereas a collaboration request indicates all the interactions that are contained within the proposed collaboration. In order to join the collaboration, the service must simultaneously accept all of the interactions within the collaboration request.

Upon joining the collaboration, a service will interact with various collaborative peers. Each of these interactions introduces a different security threat to the service. Hence, each of the peers that interact with the service must be applied to the security evaluations. Moreover, the access requirements sought from these peers may change with respect to their interaction types.

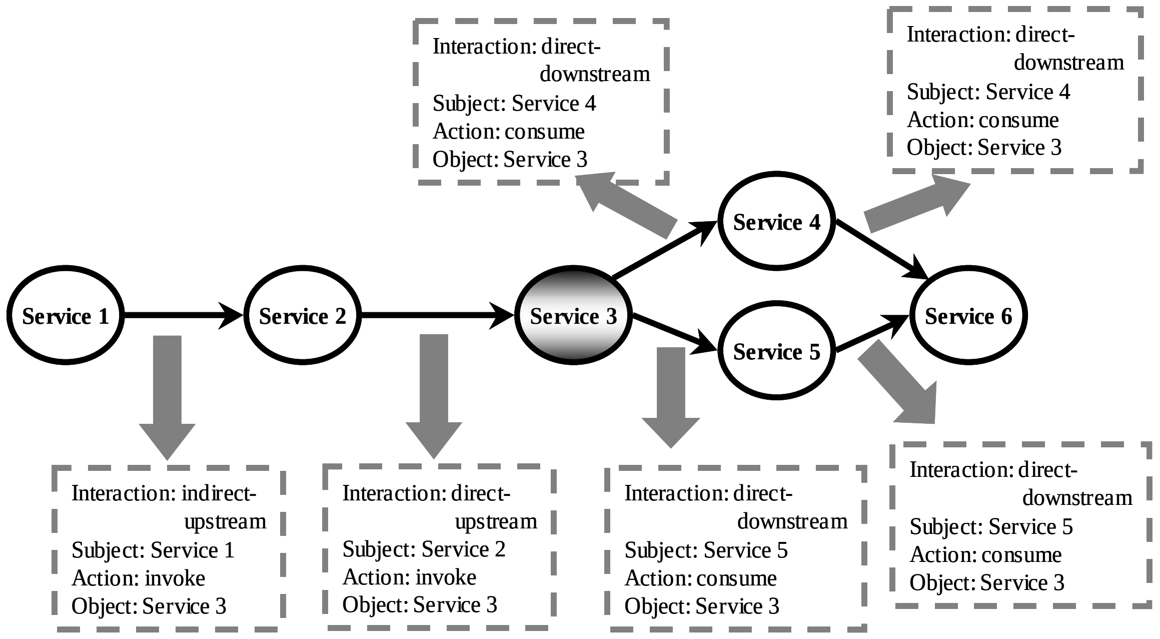
In order to accomplish this, a collaboration request consists of multiple subjects (i.e. collaborative peers), their interactions with the requested service, and the requested actions over the service. The interaction types are used as a differentiator among the subjects. A subject that is involved in a certain interaction type may be applied to different access requirements than another subject that is involved with a different interaction type. A key point of the collaboration request is that due to the varying interaction types, each subject maybe involved with a different action over the same service. For example, consider a collaboration involving three services (Services A, B, and C), where the Service A invokes Service B and passes its results to Service B. Later, Service B must send its results to Service C, and invoke it. From Service B's perspective, the interactions with Service A and Service C are different: Service A invokes Service B and passes input arguments, whereas, Service C consumes the output generated by Service B. The first interaction can be represented as an "invoke" request from "Service A" over "Service B", whereas the second interaction can be represented as a "consume" request from "Service C" over "Service B". Although both interactions are related to Service B, the specific actions taken in each of them are different, as well as, the subjects taking these actions. More importantly, Service B must approve both interactions simultaneously in order to join this collaboration.

Existing access control models are designed to evaluate what we call traditional access requests. A traditional access request represents a single interaction. Each request is typically evaluated alone and an individual policy decision is generated. The request consists of a subject, an object and an action entity. Recent research [ACDV+, XACML05] has introduced variations to this model such that the access request includes additional entities such as an environment entity. The environment entity is evaluated as part of the access request so that additional attributes of the request such as the date, the time, and the location of the request (when the requestors are mobile) can be evaluated. Therefore, the access request is defined as four-tuple of a subject, an object, an action, and an environment entity.

Furthermore, there has been other work that introduces multiple subject entities into a single request. The XACML [XACML05] framework, specifically, allows for this. The reasoning is that when multiple subjects pertain to the same access request, they must be evaluated simultaneously. The most common use-case scenarios are seen in the financial sector, where two or more subjects must be involved with a specific request to prevent any conflict of interest. The XACML model, however, does not allow for multiple action entities. Moreover, it only allows for multiple resource entities under certain circumstances: when the resources have a hierarchical tie such that accessing a higher level resource allows accessing a lower level one, then the multiple resources can be included in a single request. This situation is often observed when a request pertains to system directories or files.

A collaboration request, on the other hand, conveys information that is not conveyed in a traditional access request. The collaboration request represents all of the interactions that involve the service as part of the collaboration. In other words, the collaboration request presents a limited snapshot of the collaboration as it relates to the service being contributed.

The collaboration request includes 4 pieces of information: the subjects (i.e. the collaborative peers), the interaction types, the actions, and the object (i.e. the provided service). The collaboration request maintains the association among a subject, its interaction type with the object, and the associated action (Figure 3.1).



Collaboration Request for Service 3:

Interaction Types:	Subjects:	Object:	Actions:
indirect-upstream	Service 1	Service 3	invoke
direct-upstream	Service 2		invoke
direct-downstream	Service 4		consume
direct-downstream	Service 5		consume
indirect-downstream	Service 6		consume
indirect-downstream	Service 6		consume

Figure 3.1 The collaboration request for Service 3.

Each subject is distinguished from each other by their interaction types. The subjects that have the same interaction type with the service are applied to the same access requirements.

However, the subjects that have different interaction types are applied to different access requirements. Therefore, the access requirements sought from the subjects are not uniform.

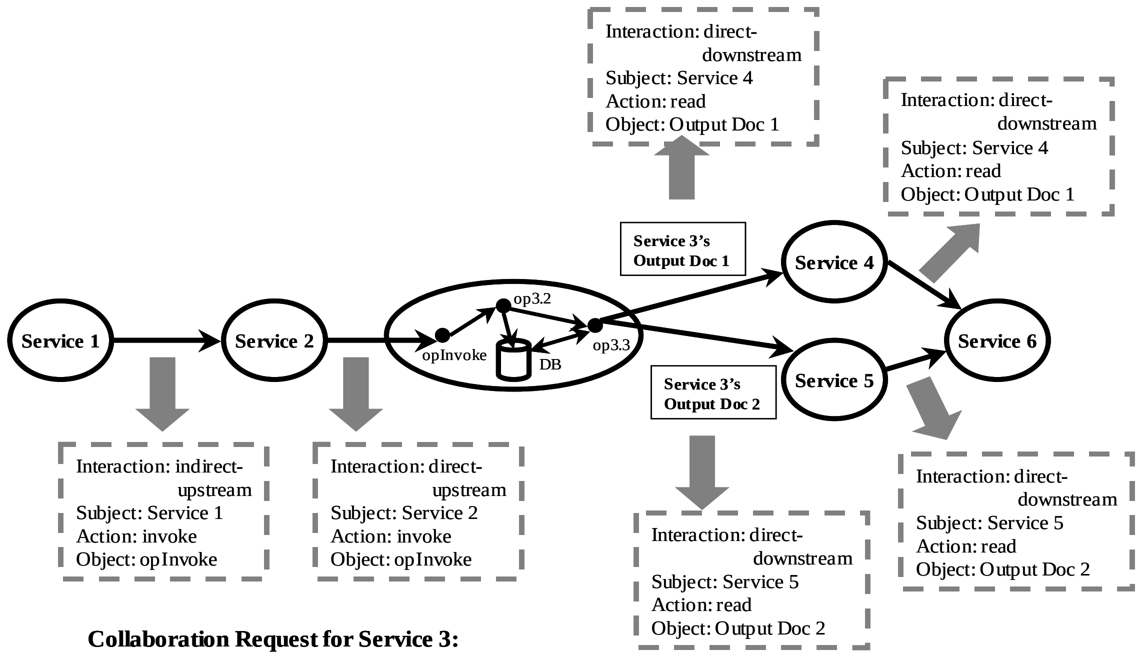
There are two actions defined in our model: *invoke* and *consume*. The invoke action is used by any collaborative peer that has an upstream interaction with the service. Any collaborative peer that has a downstream interaction with the service uses the consume action.

The collaboration request contains a single object: the service provided to the collaboration. However, as an exceptional case, when a service has a composite nature, we allow for defining multiple object elements within a single collaboration request (Figure 3.2). A service is generally regarded as synonymous with a single web service (or a single program implementation); however, in many cases, this is not true. Services are composite of several resources, including but not limited to multiple web services orchestrated together, databases, file systems and so on. The service, which is exposed to the external world, is realized through the interactions among all of its resources. Therefore, in practice, a service is rarely a single concrete object, but rather a composite abstraction. We call such services with multiple resources as *composite services*. Each resource of the composite service may have different access requirements. In such cases, the collaboration request can have multiple object entities, each of which represents a specific resource of the service. In other words, the collaboration request does not have a single object entity representing the service. As shown in Figure 3.2, Service C has multiple resources: it has three operations implemented as web services, a database, and two outcome documents. Each of the resources has different access requirements. The interaction between Service 3 and Service 2 can be represented as an "execute" request over the "opInvoke" by "Service 2", whereas the interaction between

Service 3 and Service 4 can be represented as a “read” request over the “Output Doc 1” by “Service 4”. As seen, the collaboration and its interactions affect each resources of Service C differently. In this case, the collaboration request can list each of these resources as a separate object entity. This is a fine-grained approach.

When the fine-grained approach is taken, it is also possible to further refine action entity of the collaboration request model. Rather than just using two action types, *invoke* and *consume*, more refined action types can be used. Our access control model does not restrict the refined action types. However, the actions must be meaningful over the resources included in the collaboration request. In Chapter 5, we show how this type of request can be prepared in detail.

A service must evaluate and accept the entire collaboration request, all of the subject and action entities (and the object entities, if it adopts the fine-grained approach) listed within the collaboration request, before committing to the collaboration. Once the service joins the collaboration, the service must grant access to all of its peers through their designated interactions. Consequently, the service would become exposed to the collaborative peers such that it would become impossible to isolate and protect the service from untrustworthy peers. Therefore, it is crucial that the collaboration request must be evaluated in a comprehensive manner. The entire collaboration request must be evaluated against the collaboration policy. Evaluating the parts of the collaboration request separately, such as evaluating each subject separately, can lead to undetected security breaches. The comprehensive evaluation approach eliminates security risks that cannot be detected by examining individual interactions or peers. (We later have a detailed discussion of how the lack of a comprehensive approach results in conflict of interest issues. The reader can refer to Chapter 2 for a detailed discussion of our comprehensive evaluation method.)



Collaboration Request for Service 3:

Interaction Types:	Subjects:	Objects:	Actions:
indirect-upstream	Service 1	opInvoke	invoke
direct-upstream	Service 2	opInvoke	invoke
direct-downstream	Service 4	Output Doc 1	read
direct-downstream	Service 5	Output Doc 2	read
indirect-downstream	Service 6	Output Doc 1	read
indirect-downstream	Service 6	Output Doc 2	read

Figure 3.2 Fine-grained collaboration request for Service 3.

3.2 Collaboration Policy Model

The nature of our access control model calls for a policy model that can tackle the multitude of information conveyed. We identified two main requirements of the policy model. We first discuss the requirements and then introduce our policy model. Finally, we present sample policies in Section 3.2.3.

3.2.1 Policy Model Requirements

The first requirement is that a collaboration policy must be able to distinguish among the collaborative peers with respect to their interactions with the service. The peers must be applied to different access requirements based on their interaction types. Moreover, the collaboration policy must combine the evaluation result of each collaborative peer to generate a final policy outcome. While combining the peers' evaluation results, the interaction types must be taken into account. For example, a collaboration policy must be able to state that access to a service is allowed as long as all the peers with a direct interaction type are trustworthy and all collaborative peers with an indirect interaction type do not belong to a rival company. As seen, the peers with indirect interaction types do not have to be trustworthy; however, they should not belong to a certain organization. The collaboration policy must be able to apply the peers to different access requirements based on their interaction types, and combine the results to reach a final decision.

The second requirement is that the collaboration policy must be easily integrated into an existing access control system. A collaboration policy is an upper layer access rule collection. It is called an "upper-layer" rule collection because collaboration policies are designed to co-exist with the access control policies that are traditionally designed to evaluate standalone access requests. The existing access control policies (also called underlying policies) handle the access requests that are standalone and are not part of any collaboration. It is imperative that: first, collaboration policies must not disrupt the existing access control system; second, collaboration policies must be easily augmented to the existing system; third, the collaboration policies may make use of existing policies whenever desirable. The third requirement aims to promote policy re-use among the collaboration policies and the existing policies. Although the collaboration policies are specifically designed with collaborations in mind, a policy writer must easily be able to refer to the existing policies in order to re-use them for collaboration decisions. An efficient method that provides policy re-use between underlying and collaboration policies is essential. As a result of achieving this affect, collaboration policies would be regarded as a complementary and an easy-to-adopt security feature.

3.2.2 Collaboration Policy Model

We model a collaboration policy as the smallest building block of the security system that makes access decisions for a service provided to the collaborations. There can be multiple such blocks; each collaboration policy manages access to a different service. In other words, for each service that can be offered to collaborations, there must be a specific collaboration policy. A service owner who offers multiple of his services to the collaboration must separately evaluate each service's collaboration policy.

Within a collaboration policy, an access rule is the smallest building block that states the access requirements sought from a collaborative peer. Each rule is designed to evaluate a specific interaction type. In order to distinguish between the peers, each rule is incorporated with a target interaction type. The target interaction can be one of the four distinctive interactions: upstream-direct, upstream-indirect, downstream-direct, and downstream-indirect. When desired, these interaction types can be further refined. (We discuss this in Section 1.3.1) In addition, each rule is designed for a specific action and object. The action is determined with respect to the target interaction of this rule. When the rule targets upstream interactions, the action is set to *invoke*, whereas, when the rule targets downstream interactions, the action is *consume*. The object is the service being provided to the collaboration.

A collaborative peer is evaluated against a specific rule when the peer possesses the designated interaction type. The peer must also possess the action and object entities that match the rule's target. In a proposed collaboration, there can be multiple separate peers that possess the target interaction type of a specific rule. Each peer is evaluated by the matching rule separately. The final result of the rule is determined in a deny-overrides manner. When a single peer fails the rule, the result of the rule becomes deny, even if all other matching peers satisfy the rule. For example, consider a rule that states that all peers with an upstream-indirect interaction must belong to a certain trusted organization. There are likely to be multiple peers in a given collaboration matching this rule. The rule result must become a deny decision when even a single peer belongs to another un-trusted organization, although all other peers belong to the specified trusted organization.

The result of each rule is combined with respect to a pre-defined combination logic. The name of the combination algorithm must be explicitly stated in each policy. The result of the combination algorithm constitutes the final decision over contributing service to the collaboration. It is possible that a rule's result may be a permit decision with obligations. The obligation refers to the future activities that must be performed by the subject. The rule result is contingent upon the subject satisfying the obligation. In such cases, the rule's obligations are propagated through the policy decision. We discuss obligations and how they are represented within a policy decision in Chapter 4.

Rule Types:

In order to meet our second requirement (the policy re-use between underlying and collaboration policies), we designed two rule types: *Local (L)* and *Underlying (U)*. A rule type conveys information about the manner in which the rule is evaluated. This information is used in addition to the access requirements stated inside the rule. The *Local (L)* rule type indicates that the rule is locally contained within the collaboration policy. In other words, all the access requirements associated with this rule are locally stored inside the rule; therefore, the rule does not make any references to external rules or policies.

The *Underlying (U)* rule type indicates that the access requirements associated with this rule are stored in an underlying policy (Figure 3.3). The *Underlying (U)* rule type is used to provide the re-use between underlying policies and collaboration policies. Instead of re-stating rules from underlying policies, the service owner simply creates a collaboration rule of *Underlying* type. The type Underlying rules do not have their own access requirements; they only refer to other policies. During the collaboration policy evaluation, their results are determined by the underlying policy decision (as we later show this in detail).

We designed two more rule types in order to deal with delegation of credentials: *Delegation-downstream* (D_D), and *Delegation-upstream* (D_U). The *Delegation-upstream* (D_U) rule type is used when a service is accessed with delegated credentials. The *Delegation-downstream* (D_D) rule is used when a service's credentials are delegated to other parties. Peers that have established trust relationships can join the same collaboration simultaneously. One of these peers may delegate its credentials to one of its trusted peers. In such cases, above rules types evaluate the access requests pertaining to the delegated credentials. We discuss the details of rule types, their syntaxes and implementation issues in succeeding sections.

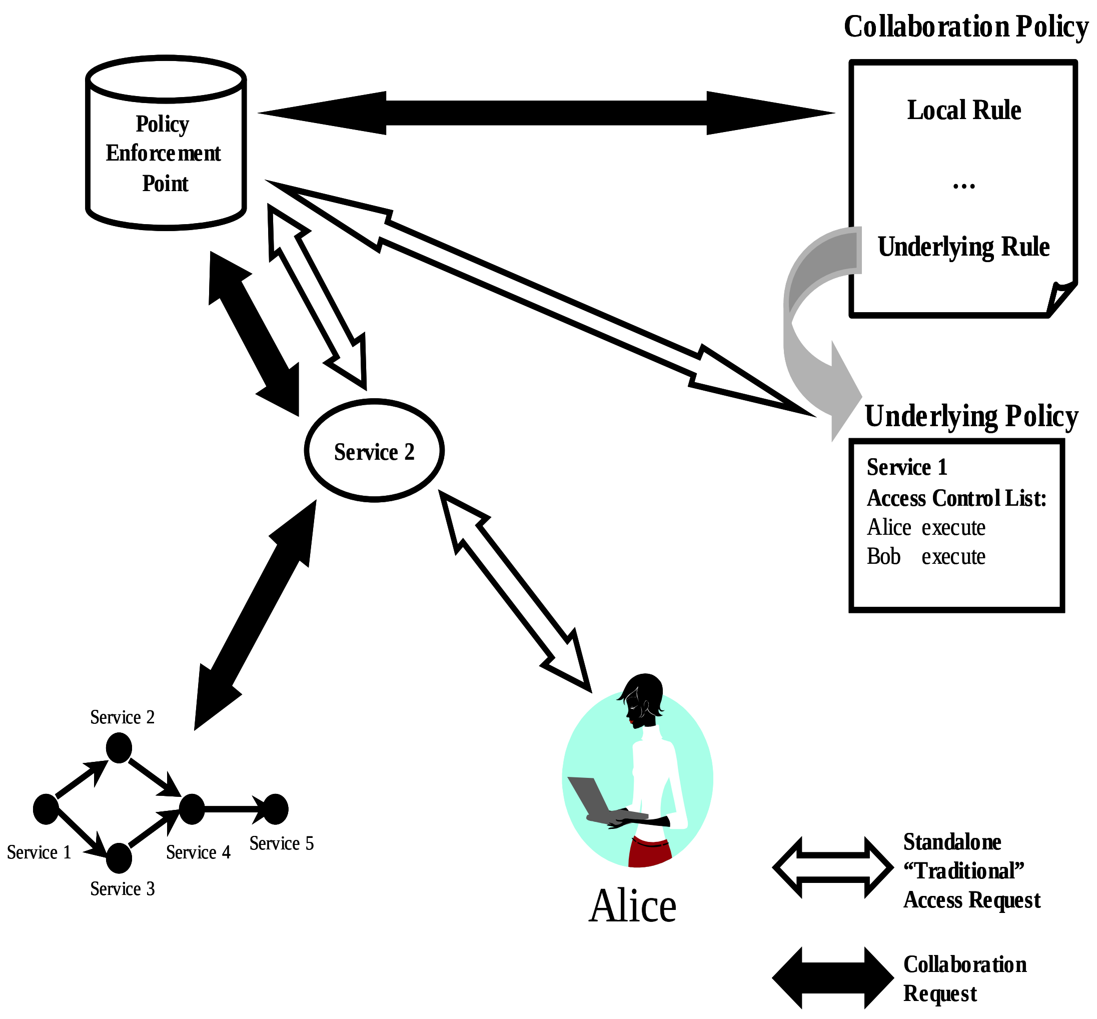


Figure 3.3 The Policy Model. The Underlying rules accomplish the policy re-use. The Local rules are tailored for collaborations.

Sample Policies:

Below we present sample collaboration policies in plain language in order to illustrate our policy and access control model better. The service below refers to the service that is provided to the collaboration. Each example stands on its own.

Example 1: The service can only be provided to a collaboration where: all collaborative peers that have an upstream-direct interaction with the service must be members of "Organization Y"; and all collaborative peers that have a downstream-direct interaction with the service must have credentials from the "Better Business Bureau".

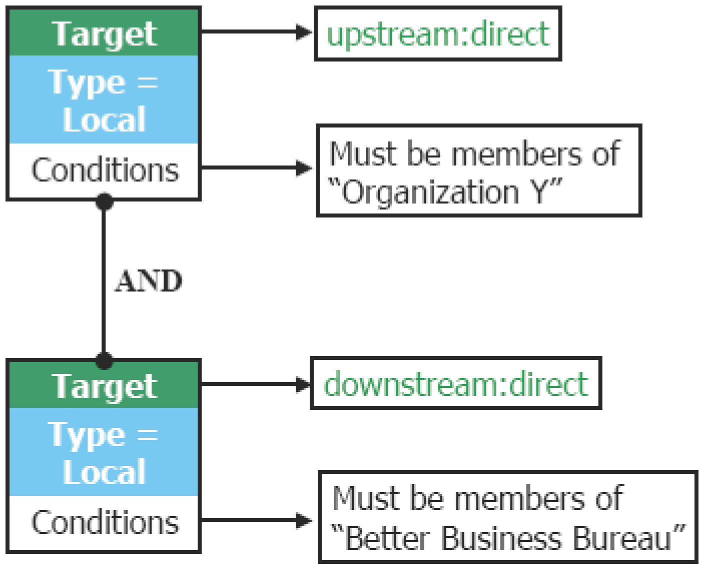


Figure 3.4 The collaboration policy stated in Example 1. The access rules are shown as individual blocks consisted of three bars: target, type and conditions elements. The connection among the rules indicates the logical combination of the rule results.

Example 2: The service can only be provided to a collaboration where: all collaborative peers with an upstream interaction (upstream-direct or upstream-indirect) with the service must have credentials from the "Better Business Bureau"; and all collaborative peers that have an upstream-direct interaction with the service must already have authorization according to the underlying policies.

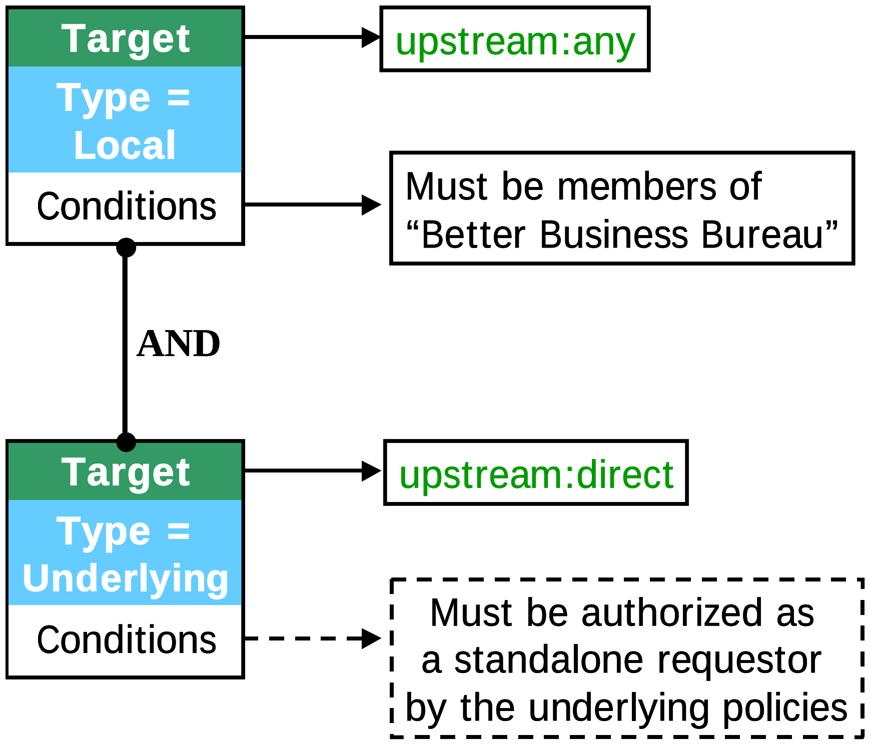

Target
Type = Local
Conditions
upstream:any
Must be members of "Better Business Bureau"
AND
Target
Type = Underlying
Conditions
upstream:direct
Must be authorized as a standalone requestor by the underlying policies

Figure 3.5 The collaboration policy stated in Example 2. The Underlying rule has no access conditions since it merely indicates that the matching collaborative peer must be authorized by the underlying policies (indicated by the dashed lines).

Example 3: The service can only be provided to a collaboration where: all collaborative peers that have an upstream-direct interaction or a downstream-direct interaction with the service must be authorized by the underlying policies; and all collaborative peers that have a downstream-indirect interaction or an upstream-indirect interaction with the service must have credentials from the "Better Business Bureau".

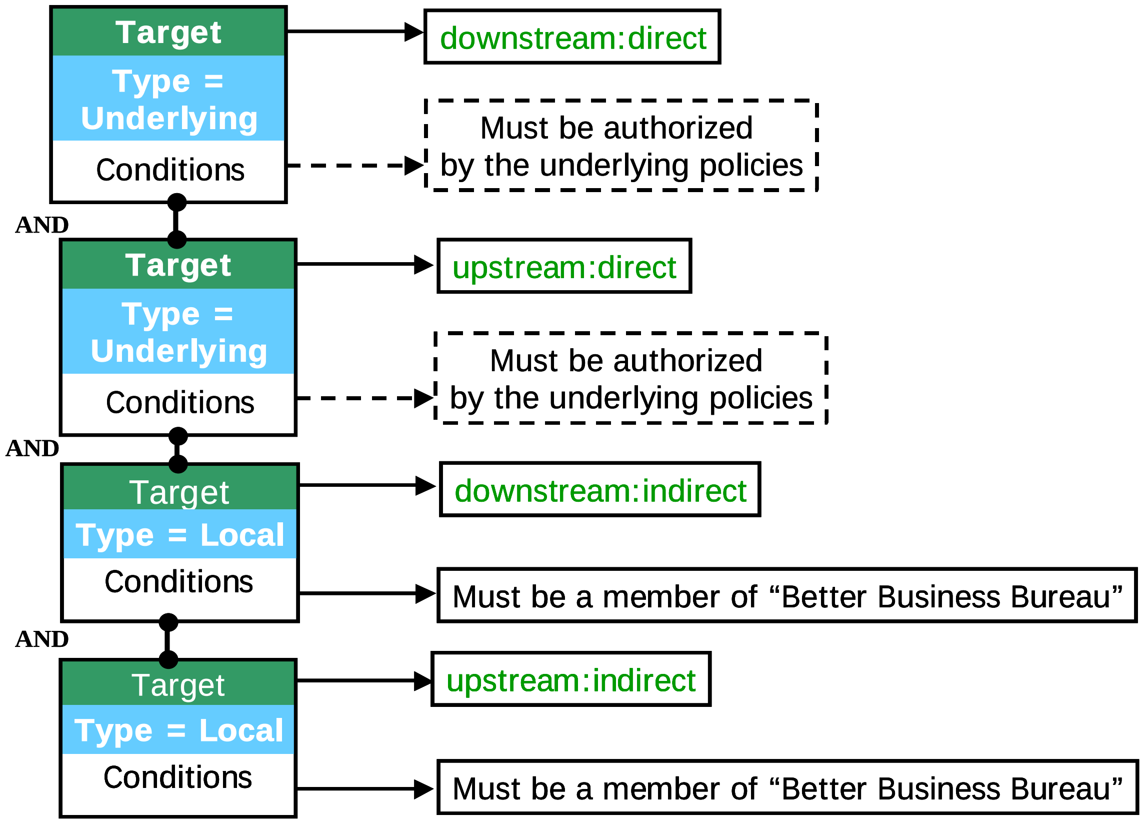


Figure 3.6 The collaboration policy stated in Example 3. The Underlying rules have no access conditions since they merely indicate that the matching collaborative peer must be authorized by the underlying policies (indicated by the dashed lines).

Example 4: The service can only be provided to a collaboration where: all collaborative peers that have an upstream-direct interaction with the service must already have authorization according to the underlying policies; all collaborative peers that have a downstream-direct interaction with the service must be members of “Organization Y”; all collaborative peers that have a downstream-indirect interaction or an upstream-indirect interaction with the service must have credentials from the “Better Business Bureau”.

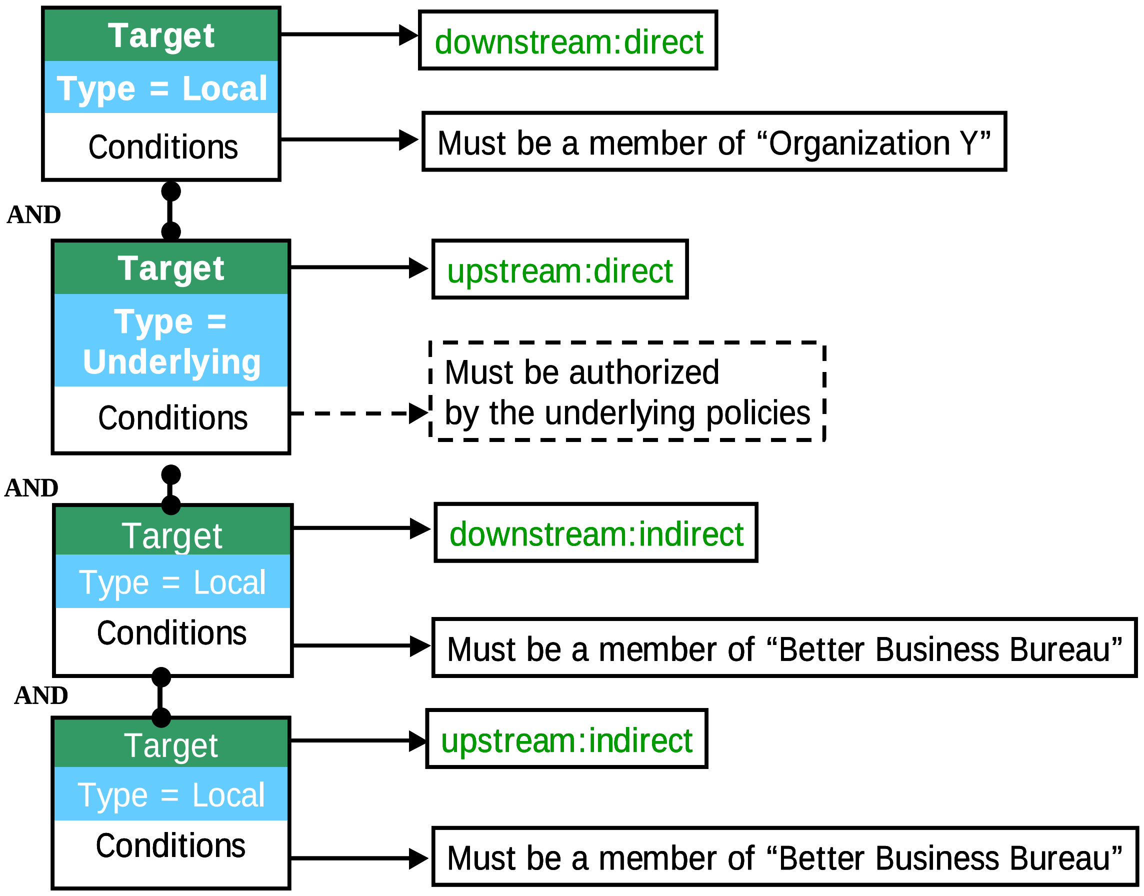


Figure 3.7 The collaboration policy stated in Example 4. The Underlying rule has no access conditions since it merely indicates that the matching collaborative peer must be authorized by the underlying policies (indicated by the dashed lines).

3.3 The Collaboration Policy Syntax

A collaboration policy consists of three elements: combination logic, maximum evaluation radius, and access rules. Below we present each element respectively. Note that whenever we refer to an element of our syntax, we represent them in italics.

3.3.1 Combination Logic

The combination logic element states the name of the algorithm that is used to combine the rule results. We provide a Boolean rule-combining algorithm in our implementation. This algorithm takes the Boolean operators and the access rules as its inputs, and combines the rule results accordingly. The policy writer must set the *CombinationLogic* element to the name of the rule-combining algorithm and provide the inputs. It is possible to define different algorithms in addition to the Boolean combining algorithm we provide. The policy writer can implement custom-made algorithms and point the *CombinationLogic* element to a specific algorithm name.

In fact, as we discuss in Section 1.5, our policy implementation is based on the XACML specification. We modified and enhanced the XACML specification as deemed necessary. The XACML specification also supports an element named *RuleCombiningAlgId*, which does the same job as the *CombinationLogic*. The XACML implementation provides a few rule-combining algorithms, such as deny-overrides, permit-overrides, first-matching-rule, etc. Since our implementation is based on that of XACML, the policy writer, in addition to the Boolean algorithm we provide, can also reference these rule-combining algorithms.

Definition 12: The CombinationLogic element equals to the name of the combination algorithm that combines the rule results and generates a policy decision.

3.3.2 Maximum Evaluation Radius

Definition 13: The MaximumEvaluationRadius element has an integer value. This value indicates the maximum number of edges between a collaborative peer and the service within the collaboration; any peers beyond this distance are not applied to the collaboration policy.

In large collaborations, the number of collaborative peers that match a rule's designated interaction type increases significantly. Especially, when a rule is designed for the indirect-upstream or the indirect-downstream interaction types, the number of matching peers increases with the collaboration complexity. As a limiting measure, the service owner can set the *MaximumEvaluationRadius* element to an integer. This integer indicates the maximum number of edges between a collaborative peer and the service such that only the peers whose distances from the service are equal or smaller than this value are applied to the collaboration policy. There can be peers in the proposed collaboration that are beyond this set distance; these peers are exempted from the policy evaluation. If the *MaximumEvaluationRadius* element is not included in a policy, an effective radius of infinity is used.

3.3.3 Access Rule Syntax

Definition 2: An Access Rule (AR) = {Target, Type, Conditions}, is the minimum building block in a collaboration policy that communicates the access requirements sought from a collaborative peer based on the peer's interactions within the collaboration. Each rule has a target interaction type, and only evaluates the peers that possess the target interaction. The access rule evaluates to either "access" or "deny".

An access rule consists of three elements: *Target*, *Type*, and *Conditions*. The *Target* element determines which collaborative peers, and their corresponding interactions, must be evaluated by this rule. The unmatched peers are not applied to the rule.

Definition 3: Target = {PeerLocation, Object, Action}. Target element determines the collaborative peers and their interaction types that must be evaluated by the rule. PeerLocation indicates the specific interaction type, Object indicates the requested resource, and Action indicates the requested permission over the Object. The Object element can either be the requested service name, or a resource component of the requested service.

The *PeerLocation* element indicates the interaction type that a collaborative peer should possess in order to be evaluated against the rule. The *PeerLocation* element can be represented in two ways: either as a *direction:interaction* pair, or by the keyword *EndUser*. In the former case, the direction could either be *upstream (up)* or *downstream (down)*. The interaction element indicates the interaction type that must exist between the service and the collaborative peer (i.e. the subject). The interaction element is either one of the keywords *direct, indirect*, and *any*, or alternatively, the interaction element could be an integer. The keywords *direct/indirect* respectively state that only a requesting peer with a direct/indirect interaction can be evaluated against the rule. The keyword *any* states that any peer, regardless of its interaction type, must be applied to this rule. When the interaction element is an integer, it indicates the number of edges between the service and the collaborative peer (i.e. the subject) within the collaboration.

Instead of a *direction:interaction* pair, the keyword *EndUser* can be used. The *EndUser* indicates that the collaboration owner, which is the entity on whose behalf the collaboration is initiated, must be evaluated against this rule, regardless of the interaction type shared between the service and the collaboration owner. The evaluation of the collaboration owner is performed when the collaboration owner is present in the collaboration such that there is a service present in the collaboration that belongs to the collaboration owner. If the collaboration owner has no services in the collaboration, this evaluation must happen through the workflow engine. The service must notify the workflow engine about its desire to evaluate the collaboration owner, and the workflow engine urges the collaboration owner to send his credentials to the service. Our implementation currently does not support this second operation mode. However, it can be implemented as the future work.

Definition 4: PeerLocation, indicates the interaction type and the relative location of a requesting peer with respect to the author. It is represented either as a direction:interaction pair or by the keyword EndUser

Definition 5: The interaction indicates the interaction type of a collaborative peer. It is one of the keywords direct, indirect, and any, or it can also be specified as an integer. When specified as an integer, it indicates the umber of edges between the collaborative peer and the service within the collaboration.

Definition 6: Direction indicates the relative location of a collaborative peer with respect to the service. It is either upstream (up), or downstream (down).

Definition 7: A collaborative peer is in the Upstream Direction of the service when there exists a directed walk W between the peer and the service such that W= v_0, e_1, v_1, …, e_n ,V , where V represents the service; v represents collaborative peers; e represents the data exchanged between two services such that the data is sent from the service on the left side of e to the service on the right side of e. If a collaborative peer is a member of W, it is in the upstream direction of the service.

Definition 8: A collaborative peer is in the Downstream Direction of the service when there exists a directed walk W between the collaborative peer and the service such that W= V, e_1 , v_1 , …, e_n , v_n , where V represents the service; v represents collaborative peers; e represents the data exchanged between two services such that the data is sent from the service on the left side of e to the service on the right side of e. If a collaborative peer is a member of W, it is in the downstream direction of the service.

The *Conditions* element of an access rule states the access requirements sought from a collaborative peer. The access requirements are represented as a predicate. During rule evaluation, the matching collaborative peer is applied against the predicate. The result of the predicate is used to determine the rule’s result.

Definition 9: An Attribute is a characteristic of an entity, such as a subject, an object, or an action. Each attribute has a name and a value.

Definition 10: A Predicate, P: (F(e) = *v*), is a Boolean-valued function, where e denotes a variable, F denotes the predicate function that e must be applied to, and *v* denotes the desired outcome of F. When the result of F equals to *v*, the result of the predicate becomes true, otherwise becomes false. F could be any arbitrary function. The variable e typically represents attributes of entities such as subject, object, or action. It is also possible that e could be another predicate. When e is defined as another predicate, first, the value of e is calculated, and then this value passed as a variable to the F. This situation results in iterative evaluation of at least two predicates (or more, depending on the number of variables). It is usually used to express complicated access requirements.

Definition 11: The Conditions element states the access requirements sought from a collaborative peer. The Conditions element is represented as a predicate, whose value is either true or false. A true evaluation is associated with “access”, and false evaluation is associated with “deny”.

Each entity, a subject, an object, or an action entity, has several attributes. In order to distinguish among the attributes, one of the keywords Subject, Object, or Action is used. These keywords represent which entity in the collaboration request owns the attribute. The predicate function (F) takes advantage of these keywords when it has to check an attribute used by different entities.

Example 5:

```
AC: {
    ( Subject.X509OrgNameAttr = “OrganizationNameX”)
}
```

Above access condition (AC) states that the subject entity must have an attribute named *X509OrgNameAttr* with a value equal to “OrganizationNameX”. The variable e is the Subject.X509OrgNameAttribute. The predicate function F is trivial, it is the identity function, so it is not shown above explicitly. *v* is the desired outcome of F, which is OrganizationNameX. It is possible to define complex predicate functions (F) that take multiple variables as its input; here, we opt to show the simplest case for brevity. In the remaining examples, unless we explicitly indicate F, the reader can assume that F is trivial and it is equal to the identity function.

As it is observed in above example, we used an attribute that is created based on the subject’s X.509 credential. Of course, this is not mandatory, and only for illustrative purposes. An access condition can choose to use any attributes. However, the common practice is to use attributes that are already conveyed by the well-adopted credentials, tickets or keys. In other words, it is good practice to look for attributes that can easily be extracted and generated based on the existing technologies. Existing access control systems typically employ X.509 credentials [IETF99], Kerberos tickets [Kerberos], SAML tokens [OASIS05-2], or user name-password pairs. In our framework, any of these keys, tokens or credentials can be employed. A policy writer can refer to these credentials, keys or tokens in order to retrieve the attributes of a collaborative peer. We do not limit the policy writer for specifying any of these attributes. A policy, for example, may require checking the security domain of a requestor by retrieving the requestor's SAML token issued by a trusted server, or by retrieving the requestor's X.509 credential. In the current prototype, we tested with X.509 credentials; however, we plan to demonstrate our framework with different type of technologies, such as SAML tokens, in future.

Example 6:

```
AR₁: { { up: direct, Service C, invoke}, L,
            Conditions:  {
                    ( Subject.X509DistName = "Alice")
            }
   }
```

Above access rule (AR_1) targets the collaborative peers with an upstream direct interaction with Service C, and they request to invoke Service C. The Conditions element states that any collaborative peer applied against this rule must have an X509DistName attribute with a value equal to Alice.

Example 7:

```
AR₂: { { up: 2, Service C, invoke}, L,
            Conditions:  {
                    ( Subject.X509OrgNameAttribute = "Organization Y")
            }
   }
```

Above access rule (AR_2) targets collaborative peers that are in upstream direction of Service C, and are 2 edges away from Service C. Note that instead of an interaction type, the service owner states the exact distance between a collaborative peer and the Service C. Therefore, this rule does not apply to any peer that has a direct interaction, or has an indirect interaction with a distance bigger than 2 edges.

Since we conclude the basics of our syntax here, below we present sample collaboration policies. The policies below only include the basic rule type *Local (L)*. Later, once we discuss rule types in more detail, we present how our syntax is augmented with additional elements.

Example 8:

```
CP₁ :
        {
                CombinationLogic= “AND”

                AR { { down: direct, Service C, consume}, L,
                        Conditions:  {
                                ( Subject.X509OrgNameAttribute = “Organization Y”)
                        }
                }

                AR { { up: direct, Service C, invoke}, L,
                        Conditions:  {
                                ( F (Subject.X509DistName = “Alice”)
                        }
                }
        }
```

Above policy (CP_2) has two access rules (each indicated by AR). The rules are combined with a Boolean AND operator, as indicated in CombinationLogic element. The MaximumEvaluationRadius is not specified; therefore, any collaborative peers matching the above rules must be evaluated. The policy states that in order to contribute Service C to a collaboration: the direct downstream peers (note that there could be multiple direct downstream peers) must be members of Organization Y; the direct upstream peers, on the other hand, can only be an entity with the name Alice. Note that this policy has no requirements from the peers that have an indirect interaction with the Service C. Instead, the policy only expresses access requirements from peers with direct-upstream and direct-downstream interactions.

Example 9:

```
CP3 :
        {
                CombinationLogic= "AND"

                AR { { up: indirect, Service C, invoke}, L,
                        Conditions:  {
                                ( Subject.X509OrgNameAttribute = "Organization Y")
                        }
                }

                AR { { up: direct, Service C, invoke}, L,
                        Conditions:  {
                                ( Subject.X509DistName = "Alice")
                        }
                }
        }
```

Above policy (CP_3) has two access rules. The rules are combined with a Boolean AND operator. The MaxiumumEvaluationRadius is not specified. (Note how drastically the number of peers matching the first rule can increase with a large collaboration. In next example, we show how to remedy this situation.) The policy states that in order to contribute Service C to a collaboration: the upstream indirect neighbors must be members of Organization Y; the upstream direct neighbors must be entities with a name equal to Alice. This policy does not have any requirements from the downstream peers. This could be because the service owner does not think that the downstream peers constitute a security threat against the Service C.

Example 10:

```
CP4 :
        {
                CombinationLogic= "AND"
                MaximumEvaluationRadius= 3

                AR { { up: indirect, Service C, invoke}, L,
                        Conditions:  {
                                ( Subject.X509OrgNameAttribute = "Organization Y")
                        }
                }

                AR { { up: direct, Service C, invoke}, L,
                    Conditions:  {
                              ( Subject.X509DistName = “Alice”)
                    }
          }
}
```

Above policy (CP_4) has two access rules, combined with a Boolean AND operator. CP_4 differs from CP_3 of Example 5 due to its MaximumEvaluationRadius, set to 3. As a result, the collaborative peers that must be applied to the first rule is limited. Although the first rule targets any peer with an indirect interaction type, the MaximumEvaluationRadius exempts the peers that are located more than 3-edges away from the Service C. Therefore, only the peers with a distance of 2-edges or 3-edges are applied to the first rule, of course given that they must be in the upstream direction of Service C.

Example 11:

```
CP5 :
{
          CombinationLogic= “AND”
          MaximumEvaluationRadius= 3

          AR { { up: any, Service C, invoke}, L,
                    Conditions:  {
                              ( F (Subject.X509OrgNameAttribute = “Organization Y”)
                    }
          }

          AR { { up: direct, Service C, invoke}, L,
                    Conditions:  {
                              ( F (Subject.X509DistName = “Alice”)
                    }
          }
}
```

Above policy (CP_5) is almost identical to the CP_4 of Example 6. However, the first access rule of CP_5 uses the *any* keyword to indicate its target interactions, whereas the first rule of CP_4 uses the *indirect* keyword. As a result of this difference, the collaborative peers with a distance of 1-edge, 2-edges or 3-edges away from the Service C are applied to the first rule of CP_5. On the other hand, the peers with 2-edges or 3-edges away from the Service C are applied to the first rule of CP_4. Due to this difference, in CP_5, the upstream peers with direct interaction type are applied to both the first rule and the second rule. Thus, an authorized upstream peer with direct interaction must belong to the Organization Y, and also has a name attribute equal to Alice.

3.4 Access Rule Types

There are four types of access rules defined for collaboration policies: *Local (L), Underlying (U), Delegation-downstream* (D_D), and *Delegation-upstream* (D_U). In this chapter, we only discuss the *Underlying* and the *Local* rule types. The *Delegation-upstream* and *Delegation-downstream* rule types are discussed in the succeeding chapter due to their complexity.

A rule type communicates information about the manner in which a rule must be evaluated. This information is in addition to the access requirements stated within the rule. The *Local (L)* type indicates that the rule is locally contained within the collaboration policy. All access requirements associated with this rule are locally stored inside the rule; therefore, the rule does not make any references to external rules or policies. The *Underlying (U)* type, on the other hand, indicates that the rule refers to external policies during its evaluation. The access requirements associated with this rule are stored in an external policy, typically in underlying policies. The *Underlying (U)* rule type is used to provide the re-use between underlying policies and collaboration policies. If a rule lacks an explicit rule type, its type is defaulted to the *Local* (*L).* We discuss the details of rule types, their syntaxes and implementation issues in succeeding sections.

The *Local* rule type fully conforms to the policy syntax we introduced in the earlier sections. Its syntax is identical to the syntax presented in the Definition 2. The *Local* rule type can be regarded as a generic rule type with the simplest syntax, and it is most commonly used in collaboration policies. In this section, we rather focus on the Underlying rule type, which has a slightly different syntax.

3.4.1 Underlying (U) Rule Type

The *Underlying (U)* rule type allows a collaboration rule to refer to another policy. The referred policy is different from the collaboration policy that contains the rule. The referred policy does not have to be another collaboration policy; it can be an arbitrary access control policy. A rule of this type indicates that the access requirements associated with this rule are stored inside the referred policy, not inside the collaboration policy that contains the rule. Therefore, the rule's result is determined by the referred policy.

This rule type is designed to provide the re-use between the underlying policies and the collaboration policies. As we discussed in our policy model, we model a collaboration policy as an upper-layer access control policy. The other access control policies that are not designed to evaluate collaborations are modeled as lower-level policies. These lower-level policies already exist in almost every security system and they are designed to evaluate standalone access requests. We call these policies as *underlying policies*. A service owner can re-use some of the access requirements that are already stated in underlying policies in his collaboration policy. The resulting collaboration policy is a medley of access rules; some rules are stated only for evaluating collaborations, and some rules have access requirements taken from the underlying policy. This situation is observed when the underlying policies are necessary and sufficient to express some of the access requirements; thus, they must be included in the collaboration policies.

However, to achieve this effect, the service owner should not be enforced to re-iterate all of the access requirements taken from the underlying policy in his collaboration policies. First, this would not be an efficient solution. Second, the desired access requirements may be spread across multiple underlying policies. Third, the underlying policies may be difficult to convert into the collaboration policy syntax. An *Underlying (U)* type collaboration rule is designated to address this problem. That is why we call this rule type as *Underlying (U)*. (From now on, we call this rule type as type *U* for brevity.)

Definition 12: A type Underlying (U) collaboration rule has an empty Conditions element. A collaborative peer that matches the Target of this rule must be evaluated against an underlying policy. The policy decision returned from the underlying policy determines the result of this rule.

The *Target* element of a type *U* rule indicates which collaborative peers must be evaluated against this rule. Since a type *U* rule refers to an underlying policy for its access requirements, the matching collaborative peers are, in effect, evaluated by the underlying policy. The result returned from the underlying policy determines the outcome of the type *U* rule.

In order to evaluate type *U* rules, the collaboration policy engine must be given the location of the underlying policy engine that enforces the underlying policies. During the evaluation of a type *U* rule, the collaboration policy engine contacts the underlying policy engine. The collaboration policy engine creates a new access request. The new request complies with the request model expected by the underlying policy engine. Our collaboration policy engine, the prototype, is already configured to create requests complying with some of the well-known request models such as that of XACML model, and it can easily be configured for other request models. While creating the new access request, the collaboration policy module may eliminate some of the information that is not meaningful to the underlying policy engine, such as the interaction type of a collaborative peer. The new access request still contains the attributes of the collaborative peer. The result returned from the underlying policy engine is treated as though it is generated by the type *U* collaboration rule.

An advantage of our evaluation scheme is that the underlying policy engine carries out the actual evaluation of the new request; thus, the collaboration policy is isolated from the details of the lower-level security system. Consider that most underlying security systems have complicated legacy-like structures. It is possible that there may be multiple underlying policies, each located in different places, even built in different languages. As a result, the evaluation of the underlying policy could be a challenge in and of itself. However, since we delegate this responsibility to the underlying policy engine, the job of the collaboration policies significantly eases. Furthermore, our solution frees the service owner from replicating the underlying structure at an upper layer, which leads to the data pollution.

Example 12:

```
CP6 : {

        CombinationLogic= "AND"
        AR { {up:any, Service C, invoke}, U, Conditions: ∅ }
}
```

CP_6 states that any upstream peers with a direct or indirect interaction type must be evaluated against the underlying policy. In other words, each upstream collaborative peer, regardless of its interaction type, must be authorized as though it requested the Service C standalone, not part of a collaboration.

Example 13:

```
CP7 : {
        CombinationLogic= "AND"
        MaximumEvaluationRadius= "3"

        AR { { up: direct, Service C, execute }, U, Conditions: ∅ }

        AR { { up: indirect, Service C, execute}, L,
                Conditions:  {
                        (Subject.X509OrgNameAttr = "Organization Y")
                }
        }
}
```

CP_7 states that all upstream collaborative peers with a direct interaction type must be evaluated against the underlying policy. Therefore, these peers are subject to the access requirements that are expected of the standalone requestors. The upstream collaborative peers with an indirect interaction, on the other hand, must only be members of “Organization Y”.

3.5 The Collaboration Policy Implementation

Due to the difficulties involved with promoting and implementing a new access control language, we selected an existing language and enhanced its syntax and implementation to meet the requirements of our collaboration policies.

The XACML (eXtensible Access Control Markup Language) framework provides an XML-based meta-language to represent access control policies, an extendable policy engine to evaluate and enforce the policies, and a simple model of access requests and access decisions that can be easily exchanged over the wire. Sun’s implementation of the XACML framework [Sun05] provides an open source Java library, which makes it possible to realize and enhance Sun’s framework. A custom-built XACML engine, based on Sun’s Java libraries, can be exposed as a web service so that it can easily communicate with other services for authorization purposes. Since we focus on web services collaborating with each other in a dynamic manner, being able to expose a policy module as a web service and to exchange access requests and the policy decisions in a uniform XML-based message format is essential for us. Finally, our experience from our earlier work [ABBD2-05] (incorporating a custom-built XACML engine into the Globus Toolkit (GT) [FK97]) motivated us to adopt the XACML framework as our foundation.

In order to implement the collaboration policies, we have enhanced the existing XACML syntax and its implementation, when necessary. In the following sections, we discuss each of these enhancements and their implementation details.

3.5.1 Collaboration Request Model

A collaboration request consists of prospective collaborative peers, their interactions with the requested service, and their resulting actions over the service. A key point of a collaboration request is that it includes multiple collaborative peers, each acting in different interaction types with the service. As a result, each peer may be involved in a different action over the same service.

A typical XACML access request consists of a subject, a resource and an action. XACML, however, also recognizes the situations in which multiple subjects pertain to a single access request and each subject acts in different capacities; therefore, each subject must be evaluated accordingly. XACML uses an attribute, namely subject-category, to differentiate between these subjects. An XACML policy, for example, can include two separate access rules: each targets a different subject-category attribute, and each rule correspondingly has different access requirements on the matching subjects. Each access rule target must explicitly indicate its target subject-category, so that only the matching subjects would be evaluated by the rule. Since an XACML access request can only have a single action element and a single resource element, there are no action-category or resource-category attributes. (XACML allows for multiple resources under special circumstances, such as when there is a resource hierarchy, e.g. a hierarchical file system; however, it is a special case [MULT].)

An XACML access request with multiple subjects, which we call a composite XACML request, would be evaluated in the same manner as any other XACML request with a single subject: the access request is checked against each rule contained in the XACML policy and is evaluated against the rules that they match. However, once the composite XACML request matches a rule, the rule's access condition must only be evaluated against the intended subject-category. To achieve this, the rule's access condition must explicitly indicate the subject-category attribute. Otherwise, the rule's conditions would be applied to on unintended subject and may be incorrectly false. For example, a composite XACML request has two subjects: a manager and a loan approver. There are two categories for each subjects. In order to return a permit decision, the manager and the loan approver must be applied to two different rules simultaneously. Each rule has the intended subject-category expressed in its access condition element. During evaluation, the composite request matches both rules. The first rule, intended for the manager, must only check the attributes of the manager contained in the XACML request. If the rule tries to evaluate the unintended subject, the loan-approver, it may return a false result.

The subject-category attributes of XACML standard would have eased our job of expressing collaboration requests in the XACML access request model. However, XACML access request model allows a single action element to be defined per each access request. In a collaboration request, it is possible that different collaborative peers may request different actions over the same service due to their interaction types. Consider that a collaborative peer that possesses an upstream direct interaction type with the service may request an "invoke" action over the service, whereas, another collaborative peer that possesses a downstream direct interaction type with the same service may request a read action over the service's outcome.

In order to fit our collaboration request model into the XACML request model, we chose to create a separate XACML access request for each of the collaborative peers (Figure 3.8). The XACML requests are combined to form a single collaboration request. Each access request has a single subject, resource (corresponds to object in our terminology) and action element. Since XACML does not allow for an additional element for indicating interaction types, we incorporated the interaction as a sub-element of subject element. As discussed in Chapter 5, the interaction types are implemented as attributes of a subject element and passed into the XACML context. Since we create a separate XACML request for each collaborative peer, we did not have to use multiple subjects in a XACML request. Thus, we do not enforce using subject-categories in collaboration policies. As a result, we did not have to incorporate subject-categories into our policy syntax, and we relieved the service owner from incorporating subject-categories into his policies.

```
<Collaboration Request 1>

    <XACML Request 1>
            <Subject>
               <Interaction>
           <Resource>
           <Action>
    </XACML Request 1>

    <XACML Request 2>
            <Subject>
               <Interaction>
           <Resource>
           <Action>
    </XACML Request 2>
              …
```

Figure 3.8 A sample collaboration request implemented with XACML access requests.

The evaluation of a collaboration request is different from that of an XACML request. Below we first discuss the evaluation of an XACML request without any modifications. We later discuss how we implemented the evaluation of collaboration requests.

In XACML framework, an XACML request is first checked against all available policies. Only a single XACML policy must match the request. Within the matched policy, the request is then checked against all the access rules. Each access rule that matches the request evaluates the request. A rule’s result can be one of the permit, deny, inapplicable, or undetermined decisions. The results of evaluated rules are combined with respect to the rule-combining logic stated in the policy. The evaluation is performed by the Sun’s implementation of Policy Decision Point (PDP).

In order to evaluate a collaboration request, we initially thought of separately evaluating each XACML request contained within the collaboration request. We could use the existing Policy Decision Point (PDP) implementation of Sun. However, combining the results of XACML requests would be troublesome. First, collaboration policies may combine their rule results in non-trivial manners. The rule-combining logic is only presented in the collaboration policy and directly accessed by the PDP. Once an XACML request is evaluated, its result has not yet been combined with any other XACML requests. The only entry point to the Policy Decision Point (PDP) is to invoke it with an XACML request. We could have modified the PDP implementation such that we can combine the XACML results out of the PDP. However, this would not be an elegant solution. It would be inefficient because the PDP is designed to the combination if the requests were provided simultaneously.

Moreover, we realized that separately evaluating XACML requests might lead to the loss of information pertaining to the collaboration context. This would have led to undetected conflict of interest scenarios. Consider that a policy states that no two upstream-indirect peers may belong to the same organization in order to prevent any conflict of interest scenarios. In a collaboration with two such indirect-upstream peers, the conflict of interest may go undetected. Assume that these two peers belong to the same trusted organization. They satisfy all their matching rules, but fail the conflict of interest principle. When we combine their results out of the PDP, this may go undetected. Our current prototype does not support detecting conflict of interest scenarios; however, we leave this as future work, and want to design our existing implementation for easily incorporating this feature in future.

In order to evaluate a collaboration request against the collaboration policy, we modified the XACML framework as follows. We modified the PDP so that it evaluates all of the XACML requests before returning a policy decision. Each XACML request is iteratively evaluated against the policy rules. Each rule only evaluates a matching access request, or returns an inapplicable result if there is no matching XACML request found. Separate access requests that share the same interaction types can match the same access rule during their evaluations. For example, there are likely to be multiple peers that possess the indirect interaction with the requested service; hence, they each match against the same access rule during their evaluations. In these cases, we determine the rule result in a deny-overrides manner. In other words, if one matching access request fails to pass the access rule, even if all other matching requests satisfy the rule, we determine the rule result as deny. Once each access request is evaluated, we resume to regular XACML implementation, which already provides ways to combine rule results to produce a policy result.

Moreover, we enhanced the matching algorithm between a request and a collaboration policy. Existing XACML standard requires each policy to have a Target element. The Target element consists of three elements: subjects, resources, and actions elements. A subjects, element can iteratively contain multiple subject elements. Note that subjects and subject are two different element names. Likewise, the resources and the actions elements can have multiple resource and action elements. The lowest level elements (e.g. subject elements) have a Boolean OR relationship. An XACML request, for example, matching one of the subject elements contained within the subjects element is considered to be a match. In order for the policy to match the entire request, the resource and action elements of the request must also match with the policy Target. Furthermore, when we tested Sun's XACML implementation with an XACML access request that includes two different subjects, we realized that the request matches an XACML policy as long as one of the subjects matches the policy target, even when the other subject does not match the policy target.

Since we have multiple XACML requests embedded inside a collaboration request, it is crucial for us to ensure that not only one of the requests, but all of them separately match a collaboration policy. Otherwise, a collaboration request may end up matching a collaboration policy that is not fit to evaluate the collaboration request. In those cases, one of the collaborative peers and their interaction types contained in the collaboration request may not be covered within the collaboration policy. To prevent such situations, we slightly modified the XACML framework: we imposed a Boolean AND relationship such that unless all XACML requests contained in the collaboration request match the policy, we regarded it a mismatch between the collaboration request and the policy.

3.5.2 Collaboration Rules

Each collaboration rule has a *Target* element that defines which collaborative peers must be applied to the rule. Recall that *Target* element has three elements *PeerLocation, Object* and *Action*. Fortunately, XACML rule syntax allows for defining a target element for an XACML rule; the XACML standard defines the target element as a composite of subject, resource and action elements. We decided to use existing resource and action elements with no modification; therefore, they would respectively correspond to our *Object* and *Action* elements.

However, for the *PeerLocation* element, we had to enhance XACML standard. Each collaboration rule is required to define their target peer either with a *direction:interaction* pair (e.g., up:indirect, down:any), or with a *direction:distance* pair (e.g., up:2, down:3), or with the *EndUser* keyword. In order to implement these keywords, we modified the Sun's XACML implementation. The rule target and policy target matching mechanisms are modified to recognize these keywords and they match the incoming collaboration requests accordingly. In our implementation, the keyword *any*, when placed to indicate the direction, matches both directions. When *any* is used in the place of an interaction type, it matches all interaction types. The keyword *direct*, when compared with an integer, is treated as integer value of one. For example, an upstream collaborative peer with a direct interaction type would match with any rule *Target* element covering the upstream direction and the direct interaction type. Examples of matching rule Target elements are: *up:1, up:direct, up:any, any:any, any:1, any:direct.* For other keywords, we seek for an absolute match between the collaboration request (or the XACML requests contained within the collaboration request) and a rule or a policy *Target* element.

3.5.3 Collaboration Rule Types: Local and Underlying

Each collaboration rule must include a *Type* element. The lack of a rule type indicates that the rule type is *Local* (*L*). The XACML standard does not use any type information associated with an access rule. In order to introduce rule types, we enhanced the Sun's XACML implementation by adding the type information to a rule instance. We realized that the *Local* type rules have a similar syntax to that of XACML access rules, except that *Local* type rules have different *Target* and *Type* elements. Since we already covered how we incorporated the *Target* and the *Type* elements into XACML standard, we do not discuss *Local* type rules here separately. Instead, we present an example of *Local* type rule and focus on type *U* rules. The discussion of *Delegation-upstream* and *Delegation-Downstream* rules is presented in Chapter 4.

Example 14:

```
<Rule RuleId="LocalRule1" RuleType="urn:collaboration:L" Effect="Permit">
   <Target>
     <Subjects>
       <Subject>
         <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue
                 DataType="http://www.w3.org/2001/XMLSchema#string">
                     up:any</AttributeValue>
            <SubjectAttributeDesignator
                 DataType="http://www.w3.org/2001/XMLSchema#string"
                 AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
         </SubjectMatch>
       </Subject>
     </Subjects>
     <Resources>
        <AnyResource/>
     </Resources>
     <Actions>
       <AnyAction/>
     </Actions>
   </Target>

   <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
```

```
        <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
          <SubjectAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:subject:X509:DN"/>
        </Apply>
        <AttributeValue
            DataType="http://www.w3.org/2001/XMLSchema#string">Alice</AttributeValue>
   </Condition >
</Rule>
```

Above *Local* type rule targets toward upstream peers with all interaction types. The rule's *Target* element has a *subject* element that indicates that the peers in upstream direction with all interaction types match this rule (specifically the *SubjectMatch* element). The rule's access requirements are shown in the *Condition* element. The predicate in the *Condition* element has a single variable, a string-equal function. The input variable is conveyed by the *Apply* element inside the Condition element. Apply element states that the subject's attribute with the indicated AttributeId must be passed as a variable to the predicate function. This specific predicate function would find the value of the indicated attribute and return the value. If returned value is equal to the desired value (indicated as "Alice"), the predicate evaluates to true.

Type U Collaboration Rules

A type *Underlying (U)* collaboration rule indicates that any collaborative peer that matches this rule must be evaluated by the underlying policy engine. The decision returned from the underlying policy engine is treated as though it is the result of the type *U* rule. A type *U* rule has an empty *Conditions* element.

Before we set out to modify XACML standard, we explored if there was a way to introduce this rule type with the minimum amount of modifications. We realized that XACML standard adopts a flexible approach when it comes to introducing new attribute types or combining algorithms, or custom-made finder modules. For example, a service owner is allowed to define new attribute types and to include them in his policy. Likewise, he can define new combining algorithms for calculating policy results. Correspondingly, the Sun's XACML implementation leaves several points of entry for developers who like to enhance the existing implementation and introduce custom-designed types. One of such entry points is *Finder* modules. Finder modules come in three flavors: attribute finder, resource finder and policy finder. An attribute finder searches an incoming access request and retrieves the attribute values that are requested by the XACML policy. In case a developer knows ahead of time that an incoming access request would not include the specified attribute type, he can design a custom-made attribute finder module that searches alternative locations to grab the specified attribute value. For example, after introducing a new attribute type, only meaningful to the service owner, it is likely that the incoming access request cannot have the new attribute readily available. In this scenario, a developer can overwrite existing attribute finder module and provides alternative methods to retrieve the value of his custom-designed attribute.

The resource finder module deals with finding the resources that are included within an access request, but are not included in a policy. This situation is most likely to occur when there is a hierarchy of multiple resources. Finally, policy finder module allows for searching for policies in alternative ways. A Policy Decision Point (PDP) module embedded with a custom-designed policy finder module can retrieve a matching policy for the request. The policy finder module is most useful when policies are placed into the security system in an ad-hoc manner, or they are placed at alternative locations. XACML standard states that only a single matching policy for each request must be returned.

For our purposes, we need to provide a service owner with an easy and efficient way to write a collaboration rule such that, when included in a policy, this rule indicates that whichever access request matches the rule must be evaluated by the underlying policy engine. We expect that a service owner would like to re-use existing access conditions over some of his collaborative peers. For example, it is likely to define new access rules over an indirect neighbor or a neighbor involved in a delegation; however, it is also likely that existing underlying access control policies might still be sufficient to evaluate a direct neighbor. We do not purport to force a service owner to specify each and every access rule from scratch. Instead, we view the collaboration policy as an upper layer policy that can re-use the policies that are already defined and used at the lower level.

We first explored whether we can allow such rule re-use without introducing a special rule type. Policy finder module stands as the most promising solution since we desired a way of retrieving the underlying policy decision. As a first solution, we deliberated to write a new policy finder module that, in addition to the Sun's basic policy finder module, could point to the underlying policy or policies, if there are multiple of them. Whenever a collaborative peer is evaluated, the collaboration policy engine would first search for the collaboration policy by using the basic policy finder module. If no matching rules are found for the peer, the collaboration engine then would employ the new policy finder module to retrieve the matching underlying policy. For the collaborative peers that must be evaluated by the underlying policy, it is essential that the collaboration engine must not find any matching rule inside the collaboration policy.

The problems with this approach are twofold. First, we need to ensure that the collaboration policy does not match the incoming request. If the policy writer mistakenly covers the incoming access request in the collaboration policy's *Target* element or includes a rule that matches the peer, then the access request would match two policies: the collaboration policy and the underlying policy, which is an error in our framework, and in the

XACML framework, too. Since this method requires exclusion of type *U* rules from the collaboration policy, we thought this approach could be prone to mistakes.

Second problem is that once the underlying policy's result is returned, this result might still need to be combined with the result of the collaboration policy. Since the policy results are directly returned to the collaboration policy engine, combining the underlying policy result and the collaboration policy result would have had to occur outside of any policy context. In the case that the combining algorithm between two (or more policies, if there are multiple underlying policies or collaboration policies) policies is complicated, this would have been burdensome for the policy engine. Consider that a service owner states that he would like to evaluate his direct neighbors in either direction against the underlying policy as if they are standalone requestors. However, if the upstream direct neighbor fails to satisfy the underlying policy, the service owner is still willing to permit access if the direct neighbor receives delegated credentials from a two-hops away upstream neighbor, given of course that the indirect upstream neighbor has the authorized credentials and the direct neighbor is not member of a rival organization. Above scenario contains Delegation-upstream rule types; however, the discussion of the delegation is beyond the scope of our discussion right now. Instead, this example is to illustrate that the result of an underlying policy decision can be combined with the result of the collaboration policy in non-trivial manners.

As an alternative solution, we deliberated to employ a policy set such that it includes the collaboration policy and all underlying policies related to the service. By defining a policy-combining algorithm, we could have gotten rid of policy result combination problem. However, literally including all the underlying policies inside the policy set defeats our purpose of providing an easy method to re-use underlying policies. After all, we set ourselves for not forcing the service owner to repeat the underlying policies over and over again. Moreover, it is possible that there could be multiple underlying policies, or even multiple policy sets, that manage standalone access requests to a service. We certainly did not want to replicate a complicated underlying system at the collaboration policy level.

As our final solution, we retreat to using rule type *Underlying (U)*. This solution required a service owner to simply specify a rule with an appropriate *Target* element and label the rule type as *Underlying*, with an empty *Conditions* element. In order to evaluate a type *U* rule, we modified the Sun's implementation as follows. Inside the Rule class, that is used to evaluate and determine a rule result, we placed a software hook. When an access request is applied against a rule type *U*, the hook acts as a policy evaluation agent. The software hook contacts the underlying policy engine, creates a new access request based on the one being evaluated, and sends the new request to the underlying policy engine. The newly created access request has a message content that is compatible with the underlying policy engine's expectations. Recall that an access request inside a collaboration request has interaction types, distances and other attributes that are not understandable, nor desired by an underlying policy. The software hook acts as if it is a policy evaluation point (PEP) for the underlying policy module and makes sure that the newly created request fully complies with the underlying policy model. The result returned from the underlying module is treated as the result of type *U* rule.

Example 15:

```
<Rule RuleId="RuleU1" RuleType="urn:collaboration:U" Effect="Permit">
   <Target>
     <Subjects>
       <Subject>
         <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
           <AttributeValue
                DataType="http://www.w3.org/2001/XMLSchema#string">
                any:direct
           </AttributeValue>
```

```
        <SubjectAttributeDesignator
              DataType=http://www.w3.org/2001/XMLSchema#string
              AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
      </SubjectMatch>
    </Subject>
  </Subjects>
  <Resources>
    <AnyResource/>
  </Resources>
  <Actions>
    <AnyAction/>
  </Actions>
 </Target>
</Rule>
```

Above example illustrates a type *U* rule that requires every direct neighbor in each direction to be evaluated with respect to the underlying policies. In other words, each direct neighbor would be subject to the same access control checks as if their access request is not part of a collaboration, rather a standalone request. Note that this rule has no *Conditions* element. The specified permit effect of this rule is only taken when there is a permit decision returned by the underlying policy.

Chapter 4: Delegation of Credentials

In our framework, the delegation of credentials occurs when a peer discovers that it does not have the necessary credentials to join the collaboration. This peer must have failed to pass the access requirements of another peer, and hence, would not likely to join the collaboration. The peer with insufficient credentials can ask other collaborative peers to delegate their credentials. The act of delegation occurs when the peer with insufficient credentials (the delegatee) finds another peer (the delegator) who is willing to delegate its credentials. It is likely that the delegatee and the delegator have a pre-established trust relationship even before joining the collaboration. For example, they may belong to the same organization, or to organizations that are business partners. The delegation of credentials is a rare case; nevertheless, it can occur when such peers join the collaboration together. Therefore, we designed our framework to allow for it.

We designed two rule types that deal with the delegation of credentials: *Delegation-upstream (D_U)* and *Delegation-downstream (D_D).* The *Delegation-upstream (D_U)* rule type is used when a service allows itself to be accessed with delegated credentials. The *Delegation-downstream (D_D)* rule is used when a service's credentials are delegated to other parties. For the remainder of this chapter, we first discuss our motivation for designing D_U and D_D rule types. During this discussion, we introduce each rule's syntax. As we show later, the D_U and D_U type rules have a slightly different syntax than that of *Local* rules. Finally, we discuss the implementation details of two rule types.

4.1 Delegation-upstream (D_U) Rule Type

The *Delegation-upstream (D_U)* rule type is syntactically and semantically different from *Local* and *Underlying* rule types. This rule type is designed for cases in which a subject (i.e. a collaborative peer) may be allowed to access the object (i.e. the provided service) with delegated credentials, rather than using its own credentials.

It is possible that the delegated credentials can be relayed through intermediate parties. In such cases, the delegator forwards its credentials to an intermediate party, and the intermediate party (or parties) relays the credentials to the designated delegatee. It is possible that the intermediate parties can also use the received credentials for their own access purposes. However, as we discuss below, the intermediate parties and their usage of received credentials are beyond the scope of *Delegation-upstream (D_U)* rules. A malicious intermediate party may use the delegated credentials for unintended purposes (i.e. the purposes other than the original purpose for which the delegator agreed to delegate its credentials). However, type D_U rules are not designed to prevent such situations. A D_U rule is only designed to protect the requested object: whether or not accessing the object (i.e. the service) with delegated credentials poses a security threat to the requested object's security domain. (Later, we show that Delegation-downstream type rules can be used to prevent the abuse of credentials.) In order to protect the requested object, a type D_U rule states the access requirements sought from each of the parties that is involved with the delegated credentials: the delegator, the intermediate parties and the delegatee. These access requirements are specified from the viewpoint of the object's owner. Therefore, they do not aim to regulate the intermediate parties' or the delegatee's treatment of the delegated credentials. In fact, the owner of the requested object may not be even aware of the circumstances under which the delegator agreed to the delegation.

Our motivation for designing type D_U rule is to enable a service owner to place access requirements over the parties involved with the delegated credentials. The delegator exercises his discretion over whether or not to delegate its credentials to another party, and if so, under which circumstances should the delegation occur. The delegated credentials naturally provide access to a set of services. However, at the time of the delegation, the owner of these services may have no means to prevent an undesired delegation that has taken place between a delegator and a delegatee. A delegatee that poses a security risk to the service owner may be deemed as trustworthy by the delegator. The delegator's trust in the delegatee does not equate to the service owners' trust in the delegatee, especially when the delegatee and the service owner belong to different organizations. Moreover, the intermediate parties handle the credentials, and hence they can introduce security threats such as viruses, Trojan horses and so on. Likewise, the delegator's trust in the intermediate parties does not guarantee the service owner's trust in them. Instead, the service owner must be enabled to carry its own security evaluation over each of these parties and to reach a decision over whether or not to allow access with delegated credentials. A type D_U rule is used to eliminate such security risks by allowing the service to exercise access control over the parties involved with the delegated credentials.

Definition 13: The rule type Delegation-upstream (D_U) indicates that accessing the requested service with delegated credentials is allowed as long as the Conditions element of this rule evaluates to true. The Conditions element of this rule places access requirements over the parties involved with the delegated credentials: the owner of the delegated credentials (i.e. the delegator), any intermediate parties relaying the credentials, and the final recipient of the delegated credentials (i.e. the delegatee).

A type D_U rule's syntax differs from *Local* and *Underlying* rule types in two ways: First, the Conditions element of a D_U rule uses a special predicate, namely delegation-upstream predicate, Second, this rule type optionally includes an additional element, namely *DelegationDistance* (*DelDist*). (Note that for the remainder of this chapter, whenever we refer to the elements of the rule syntax, we show them in italics.)

The delegation-upstream predicate contains three inner-predicates. These inner-predicates are used to convey the access requirements sought from the delegatee, the delegator, and the intermediate peers. All three inner predicates must evaluate to true in order to return a true decision from the delegation-upstream predicate. If a service owner does not require placing access requirements over one of these parties, he can leave the corresponding inner predicate unspecified.

When specified, the *DelDist* element is located between the *Type* element and *Conditions* element of the rule, causing the access rule to have four elements instead of three.

Definition 14: The DelegationDistance, DelDist, element shows the maximum number of times the credentials may have been relayed until they reach the delegatee. A DelegationDistance of 1 indicates that the credentials have been relayed directly from the delegator to the delegatee. Any DelegationDistance bigger than one indicates the presence of intermediate parties between the delegator and the delegatee.

A collaborative peer with delegated credentials can only be allowed access when a D_U rule evaluates to true. If the collaboration policy does not include any D_U type rules targeting this peer, then the peer is denied access. Moreover, if the *DelDist* element is specified, the number of times that the credentials have been relayed must not exceed the value of the *DelDist* element. Otherwise, the collaborative peer is denied access. If the DelDist element is not specified, its value is assumed to be infinity.

Example 16:

```
CP8: {
        CombinationLogic= “OR”
        MaximumEvaluationRadius= 3

        AR { {up:direct, Service C, invoke}, D_U , 2,
                Conditions:{
                        Delegation-Upstream{
                                (Subject.X509OrgName = “Organization Y”)
                                (Subject.X509DistName = “Alice”)
                                (Subject.X509OrgName = “Organization Y”)

                        }
                }
        }

        AR { {up:direct, Service C, invoke},  U, Conditions: ∅ }
}
```

CP_8 has two access rules: a type U (Underlying) rule and a type D_U rule. The type U rule states that any peer with the direct-upstream interaction must be evaluated against the underlying access control policies. The type D_U rule states that the delegatee must have a name equal to Alice (the second inner predicate), and the delegator and the intermediate parties involved in the delegation must be members of Organization Y (the first and third inner predicates). Moreover, the credentials must have been relayed only twice, meaning that there can only be a single intermediate party involved. Due to the combination logic above, the Boolean OR operator, a collaborative peer with the direct-upstream interaction has two options to satisfy the collaboration policy: the peer can either satisfy the U type rule by passing the access requirements specified for standalone service requestors, or the peer can use the delegated credentials in order to satisfy the type D_U rule.

Example 17:

```
CP_9: {
        CombinationLogic= “OR”
        MaximumEvaluationRadius= “3”

        AR { {up:direct, Service C, execute}, D_U , 2,
                Delegation-Upstream{
                        (Subject.X509OrgName = “Organization Y”)
                        (F_UP (Subject.X509DN) = “Permit”)
                        (Subject.X509OrgName = “Organization Y” )
        }
        OR
```

AR { {up:direct, Service C, execute}, U, Conditions: $\varnothing$ }
}

Example 2 is presented to show how a service owner can define specialized predicate functions and how varying inner predicates over delegation entities can be harnessed. CP_9 is almost identical to CP_8. The only difference between CP_9 and CP_8 is their second inner-predicates within their type D_U rules. Unlike CP_8, CP_9 uses a special predicate function, F_{UP}. CP_9 states that the delegator must be applied to F_{UP} and the outcome of F_{UP} must equal to permit. The predicate function F_{UP} indicates that the delegator entity must be evaluated against the underlying policy. F_{UP} function is a special function that serves the same purpose as Underlying rule types. Inside a type D_U rule, when a service owner wants an entity to be evaluated against the underlying policies, it employs F_{UP} function. This is because we cannot define a type Underlying rule inside a type D_U rule in a nested fashion. Allowing nested rules increases the complexity of our syntax, and the likelihood of mistakes made by a policy writer. As a solution, we defined a special predicate function (F_{UP}) that does not violate our syntax and also provides the desired functionality. F_{UP} predicate function is evaluated in an identical way as an Underlying rule type. The collaboration policy engine creates a new access request and routes it to the underlying policy engine, and finally retrieves the outcome of underlying policy. When this outcome is equal to Permit, F_{UP} would return a true result to the delegation-upstream predicate. (The implementation details of F_{UP} are explained in Section 3.1.5)

4.2 Delegation-downstream (D_D) Rule Type

The type *Delegation-downstream (D_D)* rules are evaluated when a collaborative peer wishes to obtain another peer's credentials. Type *Delegation-downstream (D_D)* rules are designed to regulate the delegation of credentials. A delegator evaluates its *Delegation-downstream (D_D)* type rules to determine whether to delegate its credentials. The *Delegation-downstream (D_D)* rule evaluates the delegatee, and any intermediate parties if they exist. If the rule evaluates to permit, the delegator delegates its credentials via the intermediate parties. The goal of the *(D_D)* rule is to prevent the abuse of credentials. It achieves this by allowing the delegation of credentials only to the trusted delegatees and intermediate parties.

Definition 16: The rule type Delegation-downstream (D_D) indicates that the downstream delegation of the service's credentials is allowed as long as the Conditions element of this rule evaluates to true. The Conditions element of this rule places access requirements over the parties involved with the delegation of the service's credentials: the final recipient of the delegated credentials (i.e. the delegatee) and the intermediate parties relaying the credentials.

The credentials must be propagated through the intermediate peers when the delegator and the delegatee do not have an edge between one another in the collaboration graph. It is of course possible that the delegator can send its credentials directly to the delegator, even when they do not share an edge in the collaboration graph. This can either happen outside of the collaboration context, or by adding a new edge between the delegator and the delegatee in the existing graph. We do not allow for either of these approaches. First, we are only interested in managing the delegation within the collaboration context; we have no control outside of the collaboration context. Second, adding a new edge to the collaboration graph would cause complications. The new edge causes new connections between the peers that were not connected before. Since this may happen when some peers have already finished evaluating their peers, these peers would have to re-evaluate the entire collaboration. Needless to say, due to the new connections, the number of peers that must be evaluated by a single peer increases significantly because the number of peers that are connected increases. Due to these reasons, we restrict a delegator to delegate its credentials only along the existing edges in the collaboration graph.

The delegator states the access requirements sought from the delegatee and the intermediate parties inside the D_D rule. Since the delegator itself is a service, the D_D rule governing the delegation of the delegator's credential must be present in its collaboration policy. Absence of a type D_D rule indicates that the credentials cannot be delegated to any other party.

The *Delegation-downstream* (D_D) rule type can be thought as the complement of the *Delegation-upstream* (D_U) rule type (Figure 4.1). The former is evaluated when the delegator's credentials are delegated to another party; therefore, it communicates the delegator's access requirements for the delegatee and the intermediate peers. The latter is evaluated when a service is accessed with delegated credentials; therefore, it communicates the requested service's access requirements for the delegatee, the delegator and the intermediate peers.

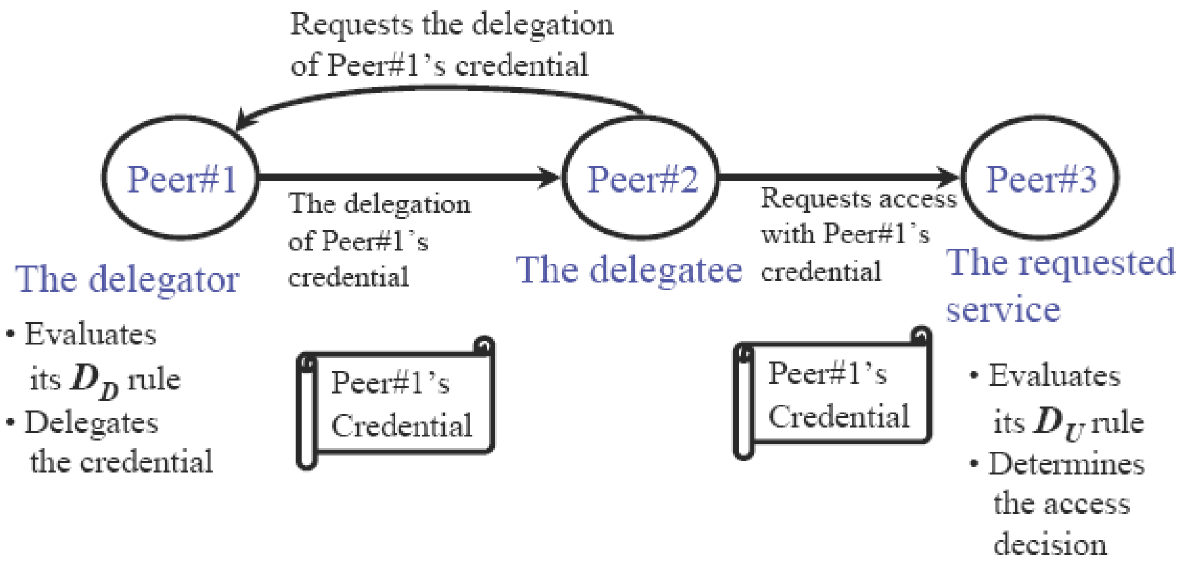

Requests the delegation of Peer#1's credential
Peer#1
The delegation of Peer#1's credential
Peer#2
Requests access with Peer#1's credential
Peer#3
The delegator
• Evaluates its D_D rule
• Delegates the credential
Peer#1's Credential
The delegatee
Peer#1's Credential
The requested service
• Evaluates its D_U rule
• Determines the access decision

Figure 4.1 Delegation of credentials scenario.

Having come from two different viewpoints, the access requirements included in a type D_D rule and a D_U rule can be completely different even when they pertain to the same service. The D_D rule aims to protect the delegated credential, hence the delegator, while the D_U rule aims to protect the service requested using the delegated credentials. As shown

Figure 4.1, the delegator and the requested service are two separate peers; they are likely to belong to separate organizations and have different policies.

There might be ideal situations, in which the D_D rule and the D_U rule are identical. In such cases, when the delegator allows delegation of its credentials, the requested service does not need to make any security evaluation of its own because the requested service would trust the delegator to have already deemed all parties involved in the delegation as secure and suitable. In a reverse example, the delegator may show some leeway in delegating its credential (such as not being explicit in which ways the credentials can be used), believing that eventually the requested service would detect all forbidden access requests and deny them. However, these types of situations are the exception, rather than the norm. As long as a delegator and the requested service belong to different security domains and have different security policies, it is highly unlikely that such a complete overlap would occur. Advocating otherwise may put an undue trust in a credential owner or in a service owner. We believe that employing separate D_D and D_U rules prevents such security breaches.

A type D_D rule syntactically differs from all other rule types. First, the *Conditions* element of the rule has a special predicate, namely Delegation-downstream predicate. Second, the rule can optionally include an additional element, namely the *DelegationDistance* (*DelDist*) element (defined in Definition 14). Third, the *Target* element of this rule must have the name of the service's credentials as its object and "delegate" as its action. When specified, the *DelDist* element is located between the *Type* element and *Conditions* element of the rule, causing the access rule to have four elements instead of three. If the DelDist element is left unspecified, it is treated as infinity.

The Delegation-downstream predicate contains two inner-predicates. These inner-predicates are used to convey the access requirements sought from the delegatee and the intermediate peers, respectively. Both inner predicates must evaluate to true in order to return a true decision from delegation-downstream predicate. If a service owner does not have any access requirements from one of these parties, he can leave the corresponding inner predicate unspecified.

The delegatee authorized by a D_D rule is the final recipient of the delegated credentials, and it is the only party authorized for using the credentials for access purposes. The intermediate parties are only authorized for relaying the credentials between the delegator and the delegatee. A type D_D rule does not express any additional requirements regarding the re-delegation of the service's credentials from the delegatee to other parties for access purposes. For the reasons related to the difficulties of enforcing re-delegation requirements and the architecture of our framework, we did not design D_D rule types to convey information about re-delegation of credentials. We discuss our reasons in Section 3.1.5 when we show the implementation of collaboration policies.

Example 18:

```
CP10:  {
        CombinationLogic= "AND"
        MaximumEvaluationRadius= 2

        AR { {down:any, ServiceCert, delegate}, DD , 1,
                Delegation-Downstream {
                        (Subject.X509DistName = "Alice")
                        (null)
                }
        }
}
```

CP_{10} states that any downstream collaborative peer can request the delegation of the ServiceCert. The delegation is granted as long as the downstream peer's name equals to "Alice". Note that the DelegationDistance (DelDist) element is set to 1; it indicates that there should not be any intermediate peers involved with this delegation. Therefore, the delegatee (i.e. the peer with name "Alice") is the final recipient of the *ServiceCert.*

Example 19:

```
CP11: {
        CombinationLogic= "None"
        MaximumEvaluationRadius= "2"

        AR { {down:any, ServiceCert, delegate}, DD , 5,
                Delegation-Downstream {
                        (Subject.X509DN = "Alice")
                        (Subject.X509OrgName = "Organization Y")
                }
        }
}
```

CP_{11} states that the downstream delegation of *ServiceCert* is allowed as long as a downstream collaborative peer with name Alice wishes to obtain the credentials. The DelegationDistance element indicates that the credentials can only be relayed 5 times before arriving at the delegatee. Moreover, the intermediate parties that relay the credentials must be members of Organization Y.

4.3 Implementation of Delegation-upstream (D_U) Rules

A type D_U rule differs from type *U* or *L* rule in that it can place access requirements over multiple collaborative peers that are involved with delegation. Each type *U* or type *L* rule is designed for a specific interaction type. A type D_U rule, on the other hand, can simultaneously target up to three specific interaction types that are involved in the delegation process. A delegatee is a collaborative peer that possesses a direct-upstream interaction type with the requested service. (We accept that, albeit a rare case, a peer with upstream-indirect interaction type can use delegated credentials during peer-peer evaluations. Although, our framework can handle these cases, we rather focus on the cases that involve a delegatee with the direct interaction type. Later in Chapter 5, we discuss why we care more about the delegatees with direct interactions.) A delegator and an intermediate peer possess an indirect-upstream interaction type with the requested service.

In order to state specific access requirements for the parties involved with the delegation, a type D_U rule employs a special predicate, delegation-upstream. Delegation-upstream predicate consists of three inner predicates that, respectively, check the conditions over a delegatee, a delegator and an intermediate peer(s).

In order to implement delegation-upstream predicate, we introduced a new function, namely delegation-upstream function, into Sun's XACML implementation. Since XACML standard welcomes contributing new functions or attribute types, our addition did not cause any disruption. Each inner-predicate consists of a predicate function, input variables, and a value that defines the desired outcome of the predicate function. A service owner is free to set any of these three items while defining each inner-predicate.

During the evaluation of a D_U rule, all three inner predicates are combined with a logical AND. The service owner does not have to explicitly set any combining algorithm to bind the inner-predicate results. The delegation-upstream function requires each inner-predicate to return a true outcome in order to return a true outcome for the delegation-upstream predicate.

The result of a type D_U rule is different from other rule results. The rule result contains the identities of the delegatee, the delegator, and the intermediate peers. This information is checked when the actual access is allowed at run-time. At run-time, if the peers involved in the delegation are different from the peers stated in the rule contract, the service refuses permitting access. A detailed discussion of rule contracts and policy obligations is presented in Chapter 5.

Finally, a type D_U rule has a third element, *DelegationDistance (DelDist),* placed between the *Type* and the *Conditions* elements. In case a D_U rule does not specify any values for the delegation distance, our implementation assumes infinity as the delegation distance. In order to implement *DelDist* element, we followed a similar approach to that of *Type* element's implementation: we modified the Rule class in Sun's implementation and added a new instance variable for showing the delegation distance. The delegation distance shows the number of times the credentials can be relayed. If the number of edges between the delegator and the delegatee exceeds this number, a deny result is returned from the D_U rule.

A peer that is involved with the delegation of credentials can also be subjected to type *L* and type *U* rules as part of the collaboration policy, as well as being evaluated by a D_U rule. The result of the collaboration policy depends on such a peer's ability to satisfy all of the matching rules. For example, a peer involved in a delegation may satisfy the conditions stated in a type D_U rule; however, if this peer fails to satisfy a matching type *L* rule, the outcome of the collaboration policy would be deny. (The complete overview of the policy evaluation is discussed in Chapter 5.)

Example 20:

```
<Rule RuleId="Delegation1" RuleType="urn:collaboration:Du" DelDist="2"
  Effect="Permit">
   <Target>
     <Subjects>
       <Subject>
         <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
           <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
                 up:any</AttributeValue>
           <SubjectAttributeDesignator
```

```
            DataType="http://www.w3.org/2001/XMLSchema#string"
            AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
      </SubjectMatch>
    </Subject>
  </Subjects>
  <Resources>
      <AnyResource/>     </Resources>
  <Actions>
    <Action>
      <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue
            DataType="http://www.w3.org/2001/XMLSchema#string">
                Request
        </AttributeValue>
        <ActionAttributeDesignator
            DataType="http://www.w3.org/2001/XMLSchema#string"
            AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
      </ActionMatch>
    </Action>
  </Actions>
</Target>
<Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:delegation-upstream">
         <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-
                        only">
                 <SubjectAttributeDesignator
                        DataType="http://www.w3.org/2001/XMLSchema#string"
                        AttributeId="urn:oasis:names:tc:xacml:1.0:subject:X509:ON"/>
              </Apply>
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
                   Organization Y
             </AttributeValue>
           </Apply>

           <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-
                        only">
                 <SubjectAttributeDesignator
                        DataType="http://www.w3.org/2001/XMLSchema#string"
                        AttributeId="urn:oasis:names:tc:xacml:1.0:subject:X509:DN"/>
              </Apply>
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
                   TrustedPeer
             </AttributeValue>
           </Apply>
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
```

```
                null
            </AttributeValue>    </Condition >
</Rule>
```

Above type D_U rule has a delegation distance of 2. Each inner-predicate is surrounded by an *Apply* element within the *Condition* element. The final inner predicate has a null value; thus, it is shown by a null-valued *AttributeValue* element. Within each *Apply* element (i.e. each inner predicate), there is a function, an input variable, and the desired value of the function outcome. Also, note that the *Condition* element has the predicate delegation-upstream. The D_U rule communicates that: the delegator must be a peer named TrustedPeer, which is proved by its X.509 credential; the delegatee must be a member of Organization Y; the intermediate peers could be any peers, shown by the null attribute value passed in the place of the third inner-predicate.

4.4 Implementation of Delegation-downstream (D_D) Rules

A type *Delegation-downstream* (*D_D)* rule is used to determine whether or not to delegate credentials to a downstream peer. An access request that has the requested credentials as its object and “delegate” as its action matches a type D_D rule. The collaborative peer who desires to receive the credentials, the delegatee, launches the access request. The service that receives the request, the delegator, evaluates the D_D rule.

A type D_D rule is very similar to a type D_U rule in terms of implementation purposes. A significant difference lies in the predicate employed by type D_D rules: delegation-downstream. The delegation-downstream predicate takes two inner-predicates as input. The first one expresses access requirements sought from a delegatee and the second one expresses access requirements sought from the intermediate peers involved. The inner predicates are bound with an implicit Boolean AND operator. As with delegation-upstream predicate, the order of the inputs passed into the delegation-downstream predicate is crucial. The service owner can select any predicate functions, attribute types and outcome values to form inner-predicates. Like delegation-upstream predicate, an inner-predicate can be left null, in case the evaluation of the corresponding peer is not necessary.

A type D_D rule has a third element *DelegationDistance (DelDist)*. *DelDist* element is implemented identically to that of the D_U rule.

The D_D rule does not convey any information about the re-delegation of the credentials. By re-delegation, we mean that the delegatee would further relay credentials to other parties. Programmatically, implementing this feature into D_D rule type would not be burdensome. We could have added another inner predicate that conveys the access requirements sought from peers that receives the credentials through re-delegation. However, we realized that enforcement of this feature at run-time would be complicated. As we will present in Chapter 6, our framework allows the delegation of credentials only once during a single collaboration; it is impossible to re-delegate the same credentials during the same collaboration. The evaluation of collaboration policies occurs at the planning stage in order to decide which peers must be allowed to join the collaboration. The type D_D rules are evaluated when a peer lacks the authorized credentials to join the collaboration. The framework allows a delegation between the delegator and the delegatee. However, this happens only once. If there were another peer who requires the re-delegation of the credentials from the delegatee, this peer would already be dismissed from the current collaboration due to the insufficient authorization. Thus, before the collaboration finishes the planning stage, all the peers joined the collaboration already have the required credentials to accomplish the tasks that they are assigned. Therefore, they would not be allowed for re-delegation. However, re-delegation of the credentials can occur in a subsequent collaboration. In that case, in order to enforce the D_D rule from the initial collaboration, we must record and evaluate the state information from the first collaboration. The delegatee who acts as a re-delegator in the second collaboration must contact the original owner of the credentials and notify him about the re-delegation.

Consequently, the original owner must re-evaluate D_D rule, specifically its third predicate over the re-delegation recipients, and reach a decision. Since we currently do not record such kind of state information across collaborations, we chose not to implement this feature. As a result, our current implementation works for the delegation cases in which a delegatee is the final receiver of the delegated credentials.

In order to ensure that the delegatee is the final recipient, we tie down each delegation to a specific collaboration graph. As we show in Chapter 6, each collaboration has a unique ID number. When the delegation of credentials occurs the delegator can embed this number into the credential, for example into the extensions field of an X.509 credential. During the delegation, the delegator can create a proxy credential [Vel03], embed the name of the delegatee, the collaboration ID, and its own name, and finally sign and delegate this credential. Since we use a collaboration ID per delegation, the revocation of delegated credentials is not required. We have not implemented this feature in our prototype. In other words, when the credentials are delegated, collaboration ID is not embedded into the delegated credentials. This is because our work so far focuses on the planning stage, not on the run-time creation of delegated credentials.

By tying a specific delegation collaboration ID, we can limit the abuse of the delegated credentials. For example, consider the case that a delegatee abuses the delegated credentials for accessing an unintended service, which results in financial charges to the delegator. By using the collaboration ID, the delegator can prove that the delegated credentials are abused because they are only valid within a specific collaboration context. We recognize that there can be other consequences of credential abuse that do not result in financial charges, such as accessing confidential files. Our framework does not address such situations currently.

Addressing such situations may require an additional mechanism that ensures that delegatee uses the delegated credentials only for intended purposes. This requires support within the domain of the requested object. In other words, the service that is accessed by the delegatee must honor the original intentions of the delegator and enforce them. However, in distributed environments, there is not an easy way of verifying that the requested object's domain honors the delegator's intentions, especially when there is no established trust between the parties, or one of the parties behaves maliciously. We recognize that this is a challenging problem and it has significant similarities to the Digital Rights Management issues (DRM) [DRM]. The DRM also deals with ensuring that parties who receive copyrighted material honors the intentions of the original party that gave them access to the material. We leave this as our future work.

Finally, the result of a D_D rule stores additional information that identifies the delegatee and the intermediate peers involved. This information stored in a special object called rule contract. Since collaboration policies are first evaluated during the planning stage, a delegator must store the results of its collaboration policies and the accompanying rule contracts until run-time. If the peers involved with the delegation at run-time differs from the authorized peers at planning time, the delegator refuses to delegate its credentials. (We discuss rule contracts in Chapter 5.)

Example 21:

```
<Rule RuleId="Delegation2" RuleType="urn:collaboration:Dd" DelDist="2"
  Effect="Permit">
   <Target>
     <Subjects>
       <Subject>
         <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
           <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
                   down:any</AttributeValue>
```

```
        <SubjectAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
       </SubjectMatch>
      </Subject>
     </Subjects>
     <Resources>
      <Resource>
       <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
                X509Credential
        </AttributeValue>
        <ResourceAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
       </ResourceMatch>
      </Resource>
    </Resources>
    <Actions>
      <Action>
       <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue
                DataType="http://www.w3.org/2001/XMLSchema#string">
                delegate
       </AttributeValue>
       <ActionAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
      </ActionMatch>
      </Action>
   </Actions>
  </Target>
  <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:delegation-downstream">
          <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-
                    only">
                <SubjectAttributeDesignator
                      DataType="http://www.w3.org/2001/XMLSchema#string"
                      AttributeId="urn:oasis:names:tc:xacml:1.0:subject:X509:DN"/>
              </Apply>
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
                   TrustedPeer
             </AttributeValue>
           </Apply>
            <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-
```

```
                    only">
                <SubjectAttributeDesignator
                      DataType="http://www.w3.org/2001/XMLSchema#string"
                      AttributeId="urn:oasis:names:tc:xacml:1.0:subject:X509:ON"/>
             </Apply>
             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
                  Organization Y
             </AttributeValue>
        </Apply>    </Condition >
</Rule>
```

Above type D_D rule indicates that the credentials can be delegated to a delegatee through a single intermediate party because the *DelDist* element’s value is 2. The delegatee must be a TrustedPeer as indicated by its X.509 distinguished name, and any intermediate peer must be a member of Organization Y.

Chapter 5:
Evaluation of Collaboration Policies

Two software modules perform evaluation of a collaboration policy: the Policy Enforcement Point (PEP) and the Policy Decision Point (PDP). The PEP plays the role of a gatekeeper; it receives the collaboration request, sends it to the PDP and enforces the decision returned from the PDP. The PDP, on the other hand, is responsible for generating a policy decision for each collaboration request. The PDP does not contact any requestor directly; it is the PEP's responsibility to interact with the requestors and with the external world. In fact, the PDP module has no point of contact outside of its security domain. In our framework, each collaborative peer has a PEP module and a PDP module. Once a policy decision is made, the PEP sends the result to the Collaboration Locator Module (CLM). If a policy decision is a permit with policy obligations, then the PEP further processes these obligations and sends the obligations along with the decision to the CLM. The policy obligations indicate that granting access at run-time is dependent upon fulfilling the stated policy obligations. Otherwise, the access will be denied.

In this chapter, we discuss the several steps involved with collaboration policy evaluation. We start with the preparation of collaboration requests (by PEP module). Later, we discuss the evaluation of a collaboration request (by the PDP module). Finally, we discuss the policy obligations and their preparation (by PEP module).

In order to avoid any confusion, for the rest of this chapter, we call a peer that desires to evaluate his collaborative peers an *authorizing peer*. The peers that are being evaluated by the authorizing peer are called *requesting peers*. During security evaluations, each peer plays both the role of an authorizing peer and the role of a requesting peer. In this chapter, we discuss the policy evaluation mechanisms from the perspective of an authorizing peer to simplify our discussion.

5.1 Preparation of Collaboration Requests

A collaboration policy evaluates the collaborative peers based on their interaction types. The peers that possess the interaction types indicated in the collaboration policy must be evaluated against the policy. However an authorizing peer does not know which one of the collaborative peers exhibits these interactions. Thus, the authorizing peer must ask the Collaboration Locator Module (CLM) to identify the corresponding peers for the specific collaboration. To achieve this, the authorizing peer's PEP module sends the interaction types required by its collaboration policy to the CLM. (How this is done is explained in detail in Chapter 6, in Section 6.1.) The CLM then consults the collaboration (the choreography of the collaboration, more specifically) and identifies the peers that correspond to the required interactions.

The CLM is responsible for sending a message to each of the requesting peers' PEP modules, and another message to the authorizing peer's PEP module to trigger the security evaluations. (The message content and the complete system architecture with order of the messages are discussed in Chapter 6.) The requesting peers' PEP modules are informed about the location of the authorizing peer's PEP and are notified to send their access requests. The authorizing peer's PEP is also informed about the identities of the requesting peers and their interaction types. The security evaluations start once each requesting peer's PEP receives a message from the CLM.

Each requesting peer's PEP sends an access request to the authorizing peer's PEP. A request consists of the requesting peer's credentials, the name of the collaboration and the name of the authorizing peer. These initial access requests do not conform to the XACML access request model, nor do they conform to collaboration request model. It is the job of the authorizing peer's PEP to convert these initial requests into the collaboration request format. We made this design choice for a few reasons. First, we want the initial access requests to have a uniform and simple message structure. Second, we want the message formats to be independent of any specific access control model. Each peer's PEP module converts these messages into whichever access control model its domain employs. As a result, the requesting peers require minimal knowledge about the access control model of the authorizing peer. Moreover, our framework can accommodate peers that have different access control models, other than our collaboration model.

The authorizing PEP receives an initial request from each of its requesting peers, and converts these requests into a collaboration request. A collaboration request consists of multiple XACML requests, each of which pertains to a specific requesting peer. (This is discussed in detail in Chapter 3, Section 3.5.1.) The PEP must first create an XACML request for each of the requesting peers, and later collect these requests into a collaboration request and send the collaboration request to the Policy Decision Point (PDP).

It is possible that a requesting peer has multiple separate interactions with the authorizing peer. In that case, each of the interactions is listed separately by the CLM, and the authorizing peer is informed of each of these interactions. The authorizing peer creates as many XACML requests as the number of separate interactions possessed by the requesting peer.

To convert the initial requests into the XACML format, the PEP first processes each received request. For each request, the PEP creates the attributes that are sought by the PDP, and extracts the corresponding values from the request. At this step, the PEP must be aware of which attributes are sought by the PDP. Although missing attributes can later be retrieved by the PDP (we show how this can be done in Chapter 3.5.3 – Type U Collaboration Rules), the PEP aims to reduce the burden on the PDP. It is highly likely that the attribute values are extracted from the credentials accompanying the request. For example, if the PDP requires an attribute showing the X.509 distinguished name of a subject, the PEP first creates the attribute and then extracts the value of the attribute from the X.509 credential. Once all attributes and their values are created, the XACML access request can be built. This step may include creating multiple attributes, each of which belongs to a different element of the collaboration request, such as the subject or action elements.

The PEP creates the *subject*, *resource* and *action* elements for each XACML request, and attaches the bags of corresponding attributes to each of these elements. In our current prototype, we defined a special *subject* attribute that shows the interaction type shared between a requesting peer and the authorizing peer. The PEP creates this attribute for each requesting peer and sets its value to each requesting peer’s interaction type. By examining this attribute, the PDP can determine which requesting peer matches which rules. If there are other attributes required by the collaboration policy, these attributes are also created by the PEP and attached to the *subject* element.

The *action* element has a single attribute; it shows the name of the action requested by the requesting peer, either *invoke* or *consume*. When a requesting peer has an upstream interaction with the authorizing peer, its action element is set to *invoke*, whereas, when a requesting peer has a downstream interaction with the authorizing peer, its action element is set to *consume*. Finally, the *resource* element is set to the name of the authorizing peer.

It is possible that the PEP can define the *resource* and *action* elements at a desired level of granularity. It is the job of the authorizing peer to determine the desired level of granularity and to configure the PEP accordingly. The PEP can be configured to adopt either a fine-grained or a coarse-grained approach. When the authorizing peer is a service that is composite of multiple resources (such as multiple orchestrated web services, databases, files, etc.), the fine-grained approach can be employed. In this approach, the PEP determines the interaction types of each requesting peer with the authorizing peer. Based on the interaction type, the PEP determines which resource of the service is accessed by the requesting peer. Consequently, the PEP sets the *resource* element to indicate the resource that gets accessed. When there are multiple resources of the service that are accessed by the same requesting peer, a separate XACML request with a different *resource* element is created. In other words, there may be multiple XACML requests correspond to the same requesting peer with different *resource* elements. Alternatively, when a coarse-grained approach is taken, the PEP sets the resource element of an XACML request to the authorizing peer's name (more specifically, to the URL of the service). Different collaborative peers each with different interaction types have the same *resource* element: the authorizing peer's name. In this approach, a single XACML request is created for each collaborative peer. For example, assume that the authorizing peer is a service that generates three disparate output documents. Each output document is accessed by a different peer with a downstream interaction. Furthermore, the access requirements for each document are different. In fine-grained approach, for each downstream peer, the PEP sets a different *resource* element indicating one of the output documents. In the coarse-grained approach, the PEP sets the *resource* element to the authorizing peer's name, and all downstream peers' XACML requests have the same *resource* element.

Likewise, the *action* element of an XACML request can be set at a desired level of granularity. Our framework does not restrict the action types that can be used within a policy when fine-grained approach is taken. However, the actions must be meaningful over the resources that are already used in XACML requests. Continuing with the above example, the action element of the XACML requests can be *read* action. Notice that in a coarse-grained approach, each XACML request has the same action element, *consume.*

Once all of the received requests are converted into the XACML format, the PEP starts converting them into the collaboration request format. Earlier we said that a collaboration request is the collection of all the XACML requests received from the requesting peers. However, when necessary, the XACML requests can be divided into groups, and for each group a separate collaboration request is created. As we discuss in the next section, a collaboration request may result in a permit decision with accompanying obligations, which are to be fulfilled by the requesting peers at run-time. An obligation pertains to a group of requesting peers. Therefore, if the PEP can, in advance, group the peers that are likely to be covered in the same obligation, the generation and process of obligations would be much easier. This is not a requirement; rather, it is an optimization performed by the PEP.

In our framework, there are two rule types that may cause obligations over a policy decision: type D_U rules and type D_D rules. For both rules, an obligation corresponds to the requesting peers that have consecutively direct interactions with one another. In other words, the peers that are covered in the same obligation constitute a directed walk that either originates from the authorizing service, or terminates at the authorizing service. In other words, these requesting peers are on the same path. For type D_U rules, an obligation covers peers that are on the same upstream path, where the path terminates at the authorizing peer. For type D_D rules, an obligation covers peers that are on the same downstream path, where the path originates from the authorizing peer. Since an obligation covers the peers that are on the same path, if we group and evaluate the peers on a path-by-path basis, the generation and processing of the obligation would be much easier. (Section 5.2.1 discusses the obligations and demonstrates how the PEP generates a policy obligation from a policy decision.)

As a result, in our prototype, we divided the XACML requests into groups based on their paths. The upstream neighbors that are located on the same path are grouped together. However, the downstream neighbors are kept in a single aggregate group. A single group of upstream peers is merged with the group of the downstream peers; the resulting group of XACML requests becomes a single collaboration request. In an iterative fashion, we create as many collaboration requests as the number of upstream paths terminating at the authorizing peer.

The reason for not separating downstream paths is due to our system architecture. A collaboration policy first evaluates its D_U rules (first round of our framework) and then, if the need arises, the policy evaluates its type D_D rules later (second round of our framework). Therefore, in the first evaluation of the policy, there is no need to group downstream peers because they would not be applied to any type D_D rules. Therefore, there cannot be any obligations generated over the downstream peers.

Finally, in a single collaboration request, we merge each group of upstream peers with all of the downstream peers because we do not want to cause any false-negative policy decisions. A single collaboration policy contains rules over requesting peers in any interaction types. If we were to evaluate each upstream group alone (without the downstream group), this might cause the policy to result in a false deny decision, due to the missing downstream requesting peers. Even though this is a small possibility, we choose to be overly cautious and include the downstream group each time we create a collaboration request.

Once the collaboration requests are prepared, they are sent to the PDP for evaluation. In order to join the proposed collaboration, each collaboration request must return a permit decision. Otherwise, the authorizing peer refuses to join the collaboration. In our prototype, the PDP evaluates each request sequentially; however, this can be optimized to evaluate all requests in parallel. The PDP returns a policy decision to the PEP only after it evaluates all of the collaboration requests. In the next section, we discuss how evaluation of a single collaboration request is accomplished. (Figure 5.2 shows the process diagram for evaluating collaboration requests.) Figure 5.1 shows the collaboration requests that are created by Service A’s PEP.

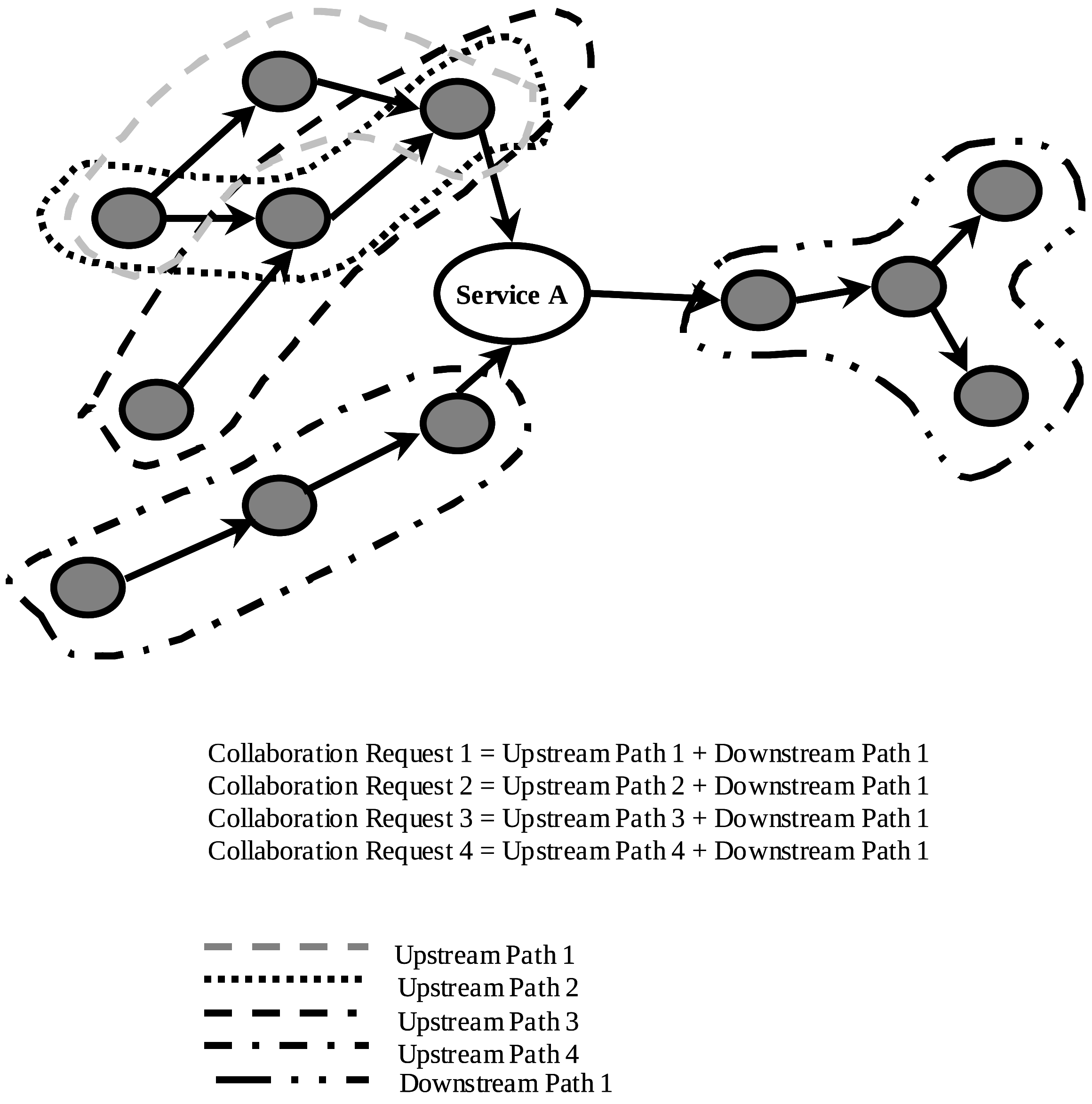

Service A
Collaboration Request 1 = Upstream Path 1 + Downstream Path 1
Collaboration Request 2 = Upstream Path 2 + Downstream Path 1
Collaboration Request 3 = Upstream Path 3 + Downstream Path 1
Collaboration Request 4 = Upstream Path 4 + Downstream Path 1
Upstream Path 1
Upstream Path 2
Upstream Path 3
Upstream Path 4
Downstream Path 1

Figure 5.1 The collaboration requests created by Service A's PEP. The requesting services are shown in the sub-collaboration graph that is sent to Service A by the CLM. Note that 4 collaboration requests are created for each upstream paths.

5.2 Evaluation of the Collaboration Request

There are three significant issues in the evaluation of a collaboration request: first, the composite structure of the collaboration request; second, the evaluation order of the collaboration rules; third, the policy obligations introduced by the type D_U and type D_D rules. We discuss them respectively in this section and the next sections.

Upon receiving a collaboration request, the PDP finds the matching collaboration policy for the collaboration request. To achieve this, the PDP must first find a matching policy for each XACML request contained inside the collaboration request. A matching policy must have a *Target* element that matches the *subject*, *resource* and *action* elements of an XACML request. When a policy simultaneously matches all the XACML requests, the policy is determined to match the collaboration request. Since each XACML request has a different subject and action element (and different resource elements if fine-grained approach is taken), it is crucial that the found policy simultaneously match all of the XACML requests.

The PDP starts evaluating the collaboration request against the policy. The evaluation order of the rule types is: first *Underlying(U)* and *Local(L)* rule types, second Delegation-upstream rule type, and third Delegation-downstream rule type are evaluated (Figure 5.2). Each rule only evaluates the XACML requests that are matching its *Target* element.

Since the collaboration request has multiple XACML requests inside, a rule must first determine which of the XACML requests it matches. A rule is determined to match an XACML request if the XACML request's *subject*, *resource* and *action* elements match the rule's *Target* element. If the rule does not match any of the XACML requests, its result remains as "inapplicable". When a rule matches more than one XACML request, each XACML request is evaluated separately. The rule's result is calculated in a deny-overrides manner: if a single XACML request fails the rule, then the rule result becomes a failure. To illustrate our logic, consider that a type *L* rule states that none of the peers (direct or indirect, upstream or downstream) can belong to a rival organization. Even when all peers except a single one come from trustworthy organizations, this rule result must evaluate to deny because one of the collaborative peers does not evaluate to true.

Once all the *L* and *U* type rules are evaluated, the policy result is computed with respect to the rule-combining algorithm stated in the policy. If a policy result is permit, the PDP stops evaluation of the collaboration request. Note that in this case, D_U rules are not evaluated at all. However, if the policy result is deny, then the collaboration request is marked as a *failing* request, and the evaluation of type D_U rules starts

At the end of the evaluation of type D_U rules, which we explain in detail in the next section, the policy decision for the *failing* request may be changed to permit with some obligations. If the *failing* request returns a deny decision from the evaluation of D_U rules, the PDP finalizes the decision for this specific request as deny. This result cannot be changed. Since each collaboration request pertaining to the same collaboration proposal must return a permit decision, the PDP terminates evaluating any more collaboration requests belonging to this collaboration. The PDP returns a deny decision to the PEP. The PEP conveys to the collaboration (to the CLM, more specifically) that the authorizing peer cannot join the proposed collaboration.

If all collaboration requests pertaining to the same collaboration return permit decisions, the PDP returns a permit result to the PEP. The PEP determines that the authorizing peer can join the collaboration. However, the PEP does not convey this result to the collaboration (the CLM module) right away. Instead, the PEP asks the PDP to start evaluating the D_D rules. All collaboration requests are evaluated once again by the PDP, but only against the type D_D rules this time. Once the PDP reaches a decision over the result of the D_D rules, it sends its result back to the PEP. Only then does the PEP sends its decision to the collaboration (to the

CLM module). Since all the collaboration requests have already returned a permit decision, the PEP determines that the authorizing peer can join the collaboration. Furthermore, if the evaluation of D_D rules allows any delegation of the authorizing peer's credentials, this is also conveyed to the CLM. We discuss the evaluation details of the type D_D rules in Section 5.2.2.

There are a few reasons for the evaluation order of our rule types. Type D_U rules allow access with delegated credentials. For a *failing* collaboration request that just returned a deny decision, evaluation of type D_U rules may switch the policy result to permit by considering the delegated credentials for access. For example, a requesting peer who cannot meet the access conditions imposed by type L or U rules, may satisfy a type D_U rule. Consequently, the requesting peer is allowed access, and the result of the policy is switched to a permit. However, this permit is dependent upon fulfilling the policy obligation: the requesting peer must use the delegated credentials for access, not its own credentials since they already failed the type L and U rules.

The reason for delaying the evaluation of D_U rules is that a D_U rule comes with obligations that must later be fulfilled. In order not to impose any unnecessary obligations, a collaboration request is first evaluated against the collaboration policy that is stripped of its D_U rules. If the result is deny, only then is the collaboration request evaluated against the type D_U rules. By delaying the evaluation of D_U rules after the evaluation of L and U type rules, we ensure that the final policy decision contains the minimum number of obligations.

The evaluation of type D_D rules occurs after the evaluation of type D_U rules. A type D_D rule determines whether to delegate the authorizing peer's credentials to a requesting peer. The result of this evaluation is stored separately, and it does not affect the authorizing peer's decision on joining the collaboration. The result of D_D rules only determines whether the authorizing peer delegates its credentials to a requesting peer upon joining the collaboration. The authorizing peer can still decide to join the collaboration even when it refuses delegation of its credentials.

The reason for delaying the evaluation of D_D rules is that an authorizing peer first must determine whether it joins the collaboration or not, which means having satisfied L, U and D_U rules. If the collaboration request already fails the L, U and D_U rules, then the peer refuses the collaboration proposal. Therefore, the peer does not need to consider whether it is willing to delegate its credentials to a collaborative peer, which results in skipping evaluation of D_D rules.

For the rest of this chapter, we follow the progress of the policy evaluation. We first discuss how type D_U rules are evaluated and their corresponding obligations are generated. Then, we discuss how type D_D rules are evaluated and their corresponding obligations are created.

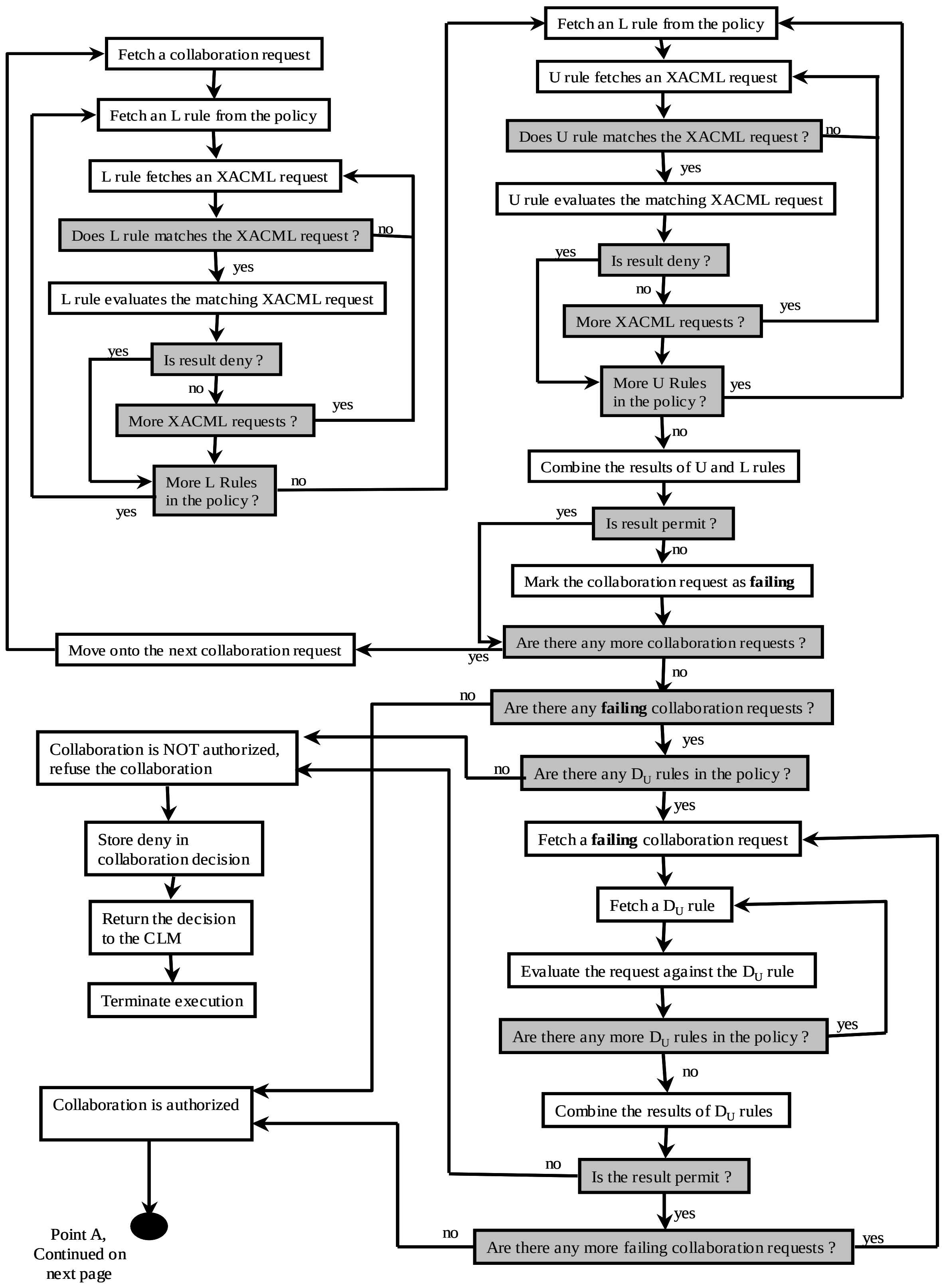

Fetch a collaboration request
Fetch an L rule from the policy
L rule fetches an XACML request
Does L rule matches the XACML request ?
no
yes
L rule evaluates the matching XACML request
Is result deny ?
yes
no
More XACML requests ?
yes
More L Rules in the policy ?
no
yes
Fetch an L rule from the policy
U rule fetches an XACML request
Does U rule matches the XACML request ?
no
yes
U rule evaluates the matching XACML request
Is result deny ?
yes
no
More XACML requests ?
yes
More U Rules in the policy ?
yes
no
Combine the results of U and L rules
Is result permit ?
yes
no
Mark the collaboration request as failing
Are there any more collaboration requests ?
yes
no
Move onto the next collaboration request
Are there any failing collaboration requests ?
no
yes
Are there any D_U rules in the policy ?
no
yes
Collaboration is NOT authorized, refuse the collaboration
Store deny in collaboration decision
Return the decision to the CLM
Terminate execution
Fetch a failing collaboration request
Fetch a D_U rule
Evaluate the request against the D_U rule
Are there any more D_U rules in the policy ?
yes
no
Combine the results of D_U rules
Is the result permit ?
no
yes
Are there any more failing collaboration requests ?
no
yes
Collaboration is authorized
Point A, Continued on next page

Figure 5.2 The evaluation of a collaboration proposal. The shaded boxes indicate the decision points.

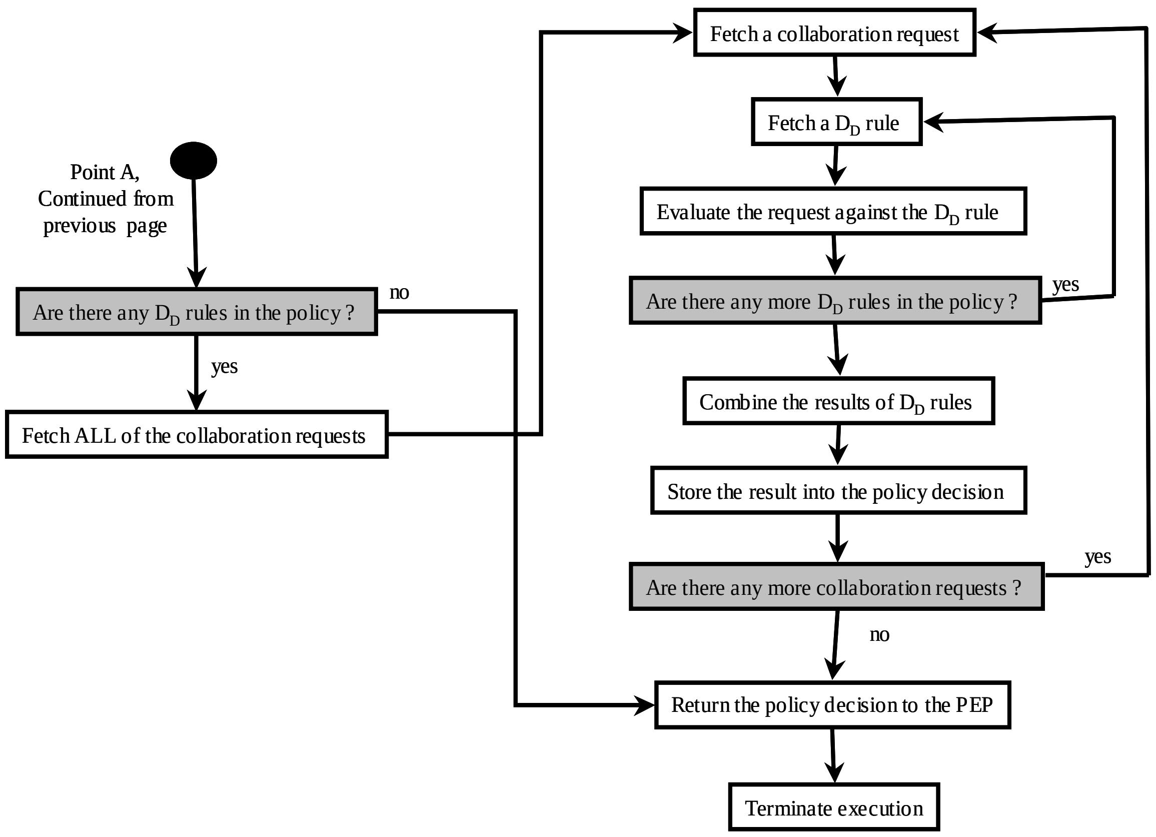

Point A, Continued from previous page
Are there any D_D rules in the policy ?
no
yes
Fetch ALL of the collaboration requests
Fetch a collaboration request
Fetch a D_D rule
Evaluate the request against the D_D rule
Are there any more D_D rules in the policy ?
yes
Combine the results of D_D rules
Store the result into the policy decision
Are there any more collaboration requests ?
yes
no
Return the policy decision to the PEP
Terminate execution

Figure 5.2 (continued).

5.2.1 Evaluation of Type D_U Rules

Each *failing* collaboration request is evaluated by the type D_U rules. The PDP first determines whether there are any D_U rules in the policy that can switch the policy outcome from deny to permit. Then, the PDP evaluates the *failing* collaboration request against the D_U rule(s) in order to see if the current request can indeed satisfy the D_U rule(s). A *failing* collaboration request that cannot satisfy the D_U rule(s) would return a deny decision to the PEP. If there are no D_U rules found in the policy, a deny decision over the collaboration request is returned to the PEP. The deny decision for a single collaboration request terminates the entire evaluation process. (See Figure 5.2)

A crucial point is that not all of the type D_U rules are evaluated: only the D_U rules that can change the policy result from deny to permit are evaluated. This is due to the obligations caused by the D_U rules. If a D_U rule is not absolutely necessary to switch the policy decision, then it should not be evaluated; thus, it cannot cause any unnecessary obligation over the policy decision. The selection of D_U rules that are necessary to change the policy outcome is discussed below.

Selection of Du rules

The rules of a collaboration policy are combined with Boolean operators: AND and OR operators. Once all rules reach a decision, their results are combined with respect to the rule-combining algorithm stated in the policy.

In order to select the D_U rules that can switch the policy decision, we perform the rule-combining algorithm. Since we have the results of L and U rules from the earlier evaluation, we substitute them into the combining logic. If there are any type D_D rules within the policy, they are skipped during this combination process.

When the combining operation between two rules is an AND operator, we compute their combined result as follows:

1. If both rules are of type D_U, return a set of both rule names as the combination result.
2. If only one of the rules is of type non-D_U and has a result of Deny, return Deny.
3. If only one of the rules is of type non-D_U and has a result of Permit, return a single set with D_U rule's name.
4. If none of the rules are of type D_U, return the logical combination of their results.

When the combining operation between two rules is an OR operator, we compute their combined result as follows:

1. If both rules are of type D_U, return two separate result sets; each set has one D_U rule's name
2. If only one of the rules is of type non-D_U and has a result of Deny, return a single set with D_U rule's name.
3. If only one of the rules is of type non-D_U and has a result of Permit, return Permit.
4. If none of the rules are of type D_U, return the logical combination of rule results.

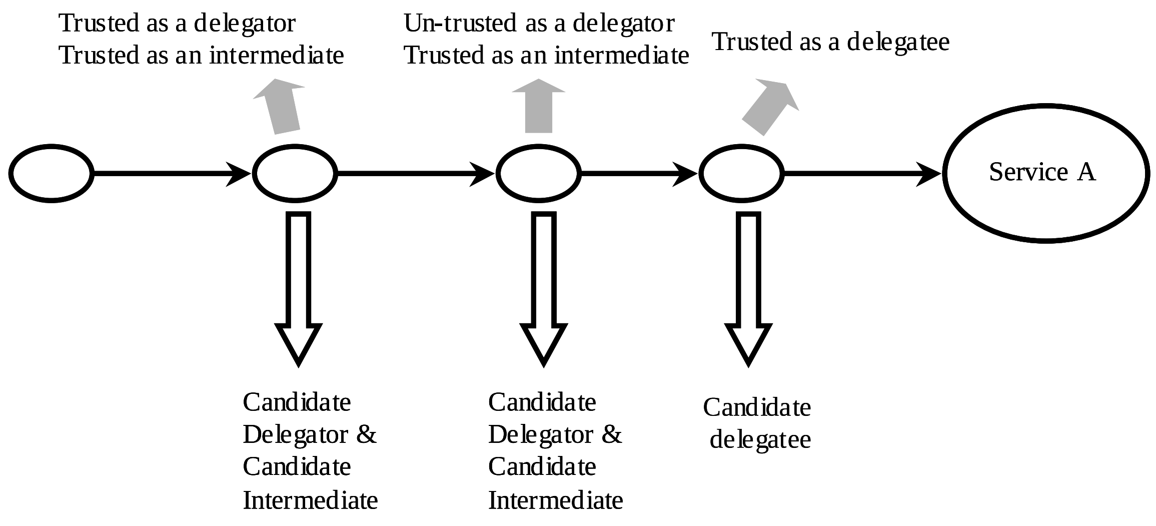
The rule-combining algorithm proceeds until all rules are exhausted. The combination result for D_U rules eventually becomes sets of D_U rule names. When these sets are combined with other rule results, they are treated as type D_U rules, in the manner explained above. Once the rule-combining algorithm is finished, the outcome may be a permit or a deny, or sets of D_U rules. Within each set, the rules maintain an AND relationship. Among the sets, there is an OR relationship: any set is capable of satisfying the policy alone. If the outcome of the combining process does not contain any D_U rules, we conclude that there is no D_U rule that can change the policy outcome from deny to permit. Hence, the execution of D_U rules is terminated, and deny decision for the specific collaboration request is returned to the PEP (which leads to the refusal of the collaboration proposal). Otherwise, we conclude that the D_U rules that are contained in the combining outcome are the necessary D_U rules that may switch the policy decision from deny to permit. The evaluation of these selected D_U rules then start.

Evaluation of Selected D_U rules

Since collaboration policies are first evaluated during the workflow planning stage, it is likely that the actual act of delegation between the delegator and the delegatee has not yet occurred. The act of delegation from the delegator to the requesting peer (i.e. the delegatee) may occur later, after the planning stage or at run-time. To take advantage of this, our framework adopts a preemptive approach with type D_U rules: even before the act of delegation occurs between the delegatee and the delegator (in fact, even before the delegator agrees to delegate its credentials), the PDP can assess a *potential* delegation. The PDP does this evaluation when a *failing* collaboration request is evaluated against the D_U rules.

The PEP sends a *failing* collaboration request back to the PDP for an evaluation against the D_U rules. This second evaluation is done in a preemptive manner such that type D_U rules assess the possibility of a *future* delegation that has not occurred yet. A type D_U rule assess three questions in a preemptive evaluation: if there is to be a future delegation taking place, (1) is there a suitable delegator in the collaboration (the peer must satisfy the second inner predicate of the rule); (2) are there suitable intermediate parties that can relay the credentials between the delegator and the delegatee (the peer must satisfy the third inner predicate of the rule); (3) does the delegatee satisfy the access requirements expected from the delegatees (the peer must satisfy the first inner predicate of the rule).

To evaluate a selected D_U rule, the PDP selects the suitable peers based on their interaction types (Figure 5.3). A peer that has a direct upstream interaction with the protected service is marked as a potential delegatee; a peer that has an indirect upstream direction with the protected peer is marked both as a potential delegator and as a potential intermediate party. The marked peers are located on the same path so that the delegation, if it occurs later, can occur among the peers that are already connected to each other by the existing collaboration connections. It is, of course, possible to seek delegation among the peers that are not connected by the collaboration; however, this would increase the number of potential delegation peers, increasing the evaluation complexity. It also may cause complications in creating new connections along which the delegation can take place. These are the motivating reasons for our decision in evaluating peers along an existing upstream path. Since when we prepared the collaboration requests, we already grouped the peers based on their paths, each collaboration request already contains peers along a single upstream path, making it easier to evaluate D_U rules.


Trusted as a delegator
Trusted as an intermediate
Un-trusted as a delegator
Trusted as an intermediate
Trusted as a delegatee
Service A
Candidate Delegator & Candidate Intermediate
Candidate Delegator & Candidate Intermediate
Candidate delegatee

Selected D_U rules: D_{U1}

DelDist = 2
Resulting Rule Contract for :
Delegator = up:3
Delegatee = up:1

Figure 5.3 The evaluation of a failing collaboration request against a D_u rule.

In our implementation, we start evaluating potential delegators with the smallest distance from the protected service for performance reasons. If a delegator cannot play the role of an intermediate party, the rule evaluation terminates immediately because no delegation chain beyond that peer can be built.

The evaluation of a D_U rule proceeds as follows: a potential delegatee is applied to the first inner-predicate of the D_U rule; a potential delegator is applied to the second inner predicate; a potential intermediate party is applied to the third inner-predicate of the D_U rule. Since a potential delegator can also be treated as a potential intermediate party, it is applied to both second and third inner predicates.

If the potential delegatee does not satisfy the first inner-predicate, the rule evaluation is terminated and a deny result is returned for this rule. If the delegatee satisfies its predicate, the evaluation continues with the potential delegator. If the potential delegator satisfies the second inner-predicate, it is marked and is recorded within the rule result (more specifically, the delegator's identity is stored inside the *rule contract*, discussed in next section.). The delegator is also evaluated against the third inner-predicate. If the potential delegator fails the third predicate, the rule evaluation terminates. The rule result is set to permit only if has the delegator satisfied the second inner-predicate; otherwise, if the delegator failed the second inner-predicate, the rule result is set to deny. If the potential delegator satisfies the third inner-predicate (with or without satisfying the second predicate), the rule evaluation continues with another potential delegator that is one-edge away from the current potential delegator. The new delegator is treated in the same manner. The evaluation continues until all potential delegators within the delegation distance are exhausted. A D_U rule can find multiple potential delegators in a collaboration request. Each of these delegators is marked and stored in the rule result separately.

Although it is not required in our framework, the PDP can adopt a different evaluation approach with D_U rules. In this approach, the evaluation of a D_U rule occurs after the actual act of delegation between the delegator and the delegatee; this is different than our preemptive approach, where the evaluation of type D_U rules occurs before the delegation actually takes place. In this alternative approach, the delegatee first obtains the delegated credentials and attaches them to its access request. Upon receiving the request with delegated credentials, the PEP examines the delegated credentials and extracts information to set attribute values. The attributes of the delegator, the intermediate peers and the delegatee are extracted from the delegated credentials. Consider, for example, a delegated X.509 credential. The PEP can extract information from the delegated credential, such as who is the original owner of the credential, if there are any intermediate parties, and the requesting peer's identity. The PEP creates a separate XACML request for the delegator, the delegatee and the intermediate parties, in the exact manner of the pre-emptive approach. The PEP finally creates a collaboration request including all the parties involved in the delegation, and sends it to the PDP for evaluation. The matching D_U rule would evaluate the peers as they correspond to the specific inner predicates, and a final rule result is returned. Since there is no need to seek for *potential* delegators, the D_U rule only evaluates a single delegator.

For a failing collaboration request, it is imperative that the request must satisfy all selected D_U rules. Otherwise, the policy result remains as deny. In our prototype, a failing collaboration request is evaluated against all D_U rules in the above fashion until the selected D_U rules are exhausted.

When all D_U rules are satisfied, the processing of their results can start, and subsequently the policy obligations are generated.

The Policy Obligations

A policy obligation indicates that the permission stated in a policy decision is contingent on mandatory actions that must later be taken by the collaborative peers. These mandatory actions are conveyed by the policy obligation. If the collaborative peers do not take these actions, the permission decision is revoked. There could be multiple obligations accompanying a single policy decision. Obligations are handled in XACML framework also. However, in XACML approach, the obligations are specified as part of the policy; they are not derived from the evaluation context. For example, a policy writer can specify a policy such that upon satisfying the policy, the requestor must accomplish the obligations stated in the policy. The PEP of XACML framework is responsible for ensuring that the obligations are honored. In our framework, obligations are not specified as part of a collaboration policy. Instead, the obligations are dynamically derived from the collaboration context. Upon satisfying the same collaboration policy, two collaboration requests can have different obligations.

In our framework, policy obligations can only be generated due to type D_U rules and type D_D rules. For both rules, the obligations state the identities of the delegator, the delegatee and the intermediate peers. A type D_U rule or a type D_D rule's result is only valid for a specific delegation instance; if there are to be changes in the delegation instance at run-time, the rule result becomes obsolete. For example, if one of the intermediate peers is replaced by a different peer, or the delegated credentials are different from the credentials that are evaluated, the rule result becomes deny. Policy obligations record the state information related to a specific delegation instance.

A D_U rule result contains a special object, namely a rule contract. A rule contract is used to identify the delegators that succeeded in satisfying the D_U rule. The contents of a rule contract is shown below:

Rule Contract: {Collaboration Request Path ID, (Location of the first delegator, allowed re-delegation distance), (Location of the second delegator, allowed re-delegation distance), *}. The star indicates that the location and re-delegation distance pairs are repeated for each suitable delegator.

Semantically, a rule contract states that the D_U rule would grant the promised access only if one of the delegators agrees to delegate its credentials to the delegatee. The identities of delegatee and the intermediate peers are not separately stored because the path ID along which the delegation must take place is included in the rule contract. A delegatee has a direct interaction with the service, and the peers in between are identified as intermediate parties along a specific path.

For example, Rule Contract 1 = {path:1, (3, unbounded), (5, unbounded)} means either a delegator located 3-hops away or another delegator located 5-hops away must be willing to delegate to the delegatee which is one-hop away from the service. The delegators located 3-hops away and 5-hops away have unbounded re-delegation distances. When a specific re-delegation distance is defined, the delegatee that has received the credentials in a manner that exceeds the delegation limit is refused access, even when the credentials are properly delegated. The delegation distance is indicated by the *DelegationDistance (DelDist)* element of the rule.

A policy obligation object has the same content as a rule contract. However, it differs from the rule contracts semantically.

Policy Obligation: {Collaboration Request Path ID, (Location of the delegator, allowed re-delegation distance)*}

A policy obligation object enumerates the delegators that must be willing to delegate their credentials. If even one of the stated delegators refuses to partake in the delegation, the obligation is said to be unsatisfied. A rule contract, on the other hand, enumerates the delegators, at least one of which must agree to delegate their credentials. In other words, the list of delegators inside a rule contract maintains an OR relationship, whereas, the delegators inside a policy obligation maintain an AND relationship. For example, a policy obligation including two peers, such as up:3 and up:5, means that both peers must simultaneously be willing to delegate their credentials. A rule contract that has the same content means that as long as one of the two peers agrees to delegate their credentials, the D_U rule is satisfied.

The semantic difference arises from the different usages of rule contracts and policy obligations; a rule contract enumerates the suitable delegators for a specific rule, whereas, a policy obligation enumerates the suitable delegators for an entire policy. In order to generate a policy obligation, the results of the requisite D_U rules and their rule contracts must be combined and be processed according to the policy's rule combining logic. A policy obligation must be created such that it states all of the requisite delegators that can and must satisfy simultaneously all of the selected D_U rules. In other words, a policy obligation is a combination of rule contracts with respect to the combining logic specified by the policy.

Generation of Policy Obligations

The following algorithm is applied to generate the policy obligations for a single collaboration request that has been evaluated against a group of selected D_U rules. Recall that when we select the D_U rules, we perform the rule-combining algorithm among the rule results. As the outcome of combining process, we obtain sets of D_U rule names. Within each set, the rules maintain an AND relationship. Among the sets, there is an OR relationship; meaning that any set is capable of satisfying the policy alone. Below, we show how we process these rule sets in order to generate the policy obligations.

For a set of D_U rules, combined with AND operator

a. Create a temporary policy obligation object
b. Compute the dot product of two rule contracts as follows
    i. For each element of the first rule contract, except the path ID element
        1. Concatenate the element with each element of the second rule contract. (Resulting element would have two separate delegators and their re-delegation distance.)
        2. Store the resulting elements inside the temporary policy obligation object.
        3. If the element of the first rule contract *covers* the element of the second rule contract, apply *pruning* algorithm and move onto Step (e), no more processing is required for this element of the first contract.
c. Delete both rule contracts from step 1-a

d. Select a new rule contract and move onto Step b, only this time compute the dot product of the temporary policy obligation object with the new rule contract. Repeat this step until all rule contracts are processed.

e. Create the final policy obligation objects

    i. Create a separate policy obligation object for each element of the temporary policy obligation object

The above algorithm determines which delegators must be willing to delegate their credentials in order to satisfy a group of D_U rules. For D_U rules combined with an AND operator, these delegators must agree simultaneously. By selecting a single element from each rule contract and concatenating with each element of the other rule contract, we determine all possible combinations that can satisfy both of the D_U rules. The resulting elements in the temporary obligation object maintains an OR relationship. Each of them can satisfy both D_U rule simultaneously. That is why we create a separate policy obligation for each of these elements (Step e-i).

The covering and pruning algorithms are employed in cases where two separate rule contracts contain the same delegator. In such cases, instead of concatenating the delegator with another delegator, it is sufficient to list the same delegator only once because it can alone satisfy both rules. Covering algorithm is used to discover these cases between two rule contracts.

Covering algorithm:

- To determine whether an element of a rule contract covers another element of another rule contract:
    - If there is a single delegator contained in the first element and that delegator is identical to the delegator contained in the second element:
        - The first element is said to cover the second element.
        - If the delegator in second element has a re-delegation value smaller than that of first element, the re-delegation value in the first element is changed to that value.
    - If the first element contains multiple delegators (this element has already been concatenated with another rule contract element):
        - If any of the delegators contained in the first element equals to the delegator contained in the second element
            - The first element is said to cover the second element.
            - If the delegator in the second element has a re-delegation value smaller than that of first element, the re-delegation value in the first element is changed to that value.

When an element of the rule contract (the first element) is said to cover another element (the second element), the following pruning algorithm is applied:

Pruning Algorithm

- Instead of concatenating the first and second elements, place the first element into the temporary obligation object alone.

- In temporary obligation object, check each previous element, except the one added in the previous step.
- If any of the previous elements contains the first element, remove that element.

The pruning algorithm removes any redundant elements in the temporary obligation object. The pruning process checks all the previously concatenated elements. If a previous element has the delegator that is discovered to satisfy both rules, we delete that previous element. The deleted element is apparently created before we discover that the delegator it contains can satisfy both D_U rules simultaneously. As a result, the deleted element has the delegator that has the ability to satisfy both D_U rules and, in addition, it has another delegator due to the earlier concatenation. Since only a single delegator is sufficient to satisfy both rules, the process of concatenation was completely unnecessary; therefore, pruning this element is viable.

Example 22:

Assume we have a collaboration policy that has four D_U rules: (R1, R2, R3, R4). The collaboration request fails the first policy evaluation; therefore, we select the D_U rules that can change the policy decision. After the selection process, we end up with two sets of rules: (R1, R2, R3); (R4). In order to convert policy decision to permit, either R1, R2 and R3 simultaneously must be satisfied, or R4 alone must be satisfied by the failing collaboration request.

Upon evaluating the D_U rules against the collaboration request, we obtain the following rule contracts:

Rule Contract for R1 (RC1): {path: 2, (3, unbounded), (5, unbounded)}
Rule Contract for R2 (RC2): {path: 2, (2, unbounded), (5, unbounded)}
Rule Contract for R3 (RC3): {path: 2, (3, 6), (6, 6)}
Rule Contract for R4 (RC4): {path: 2, (2, 6), (3, 6)}

We start with the first set. Our algorithm calls for computing the dot product between R1 and R2.The resulting temporary policy obligation is shown below.

```
Temporary Policy Obligation  = {path:2,
                                 (3, unbounded)(2, unbounded),
                                 (3, unbounded)(5, unbounded),
                                 (5, unbounded)(2, unbounded),
                                 (5, unbounded)(5, unbounded)}
```

Each row of the temporary policy obligation shows a single concatenated element, except first row, which shows the path ID. While computing the final element, row 5, the covering algorithm discovers that the delegator 5-edges away, (5, unbounded) element, can satisfy both R1 and R2 rules alone. Consequently, the pruning algorithm is invoked. The pruning algorithm determines that all earlier elements containing this delegator, i.e. the (5, unbounded) element, must be erased because they have unnecessary additional delegators. After the pruning, the policy obligation is shown below:

```
Temporary Policy Obligation= {path:2,
                               (3, unbounded)(2, unbounded),
                               (5, unbounded)}
```

The algorithm continues with computing the dot product of Temporary Policy Obligation Object with R3

```
Temporary Policy Obligation = {path:2,
                                (3, unbounded)(2, unbounded)(3, 6),
                                (3,unbounded)(2,unbounded)(6, 6),
                                (5, unbounded)(3, 6),
                                (5, unbounded)(6, 6)}
```

The above temporary obligation object shows the result of computing te dot product without any pruning, for illustrative purposes. During the algorithm execution, while computing the second row, the covering algorithm discovers that the element (3, unbounded)(2, unbounded) covers the element (3, 6). The delegator located 3-edges away is already listed in the element (3, unbounded)(2, unbounded) and in the element (3, 6). Therefore, the first element can satisfy the second rule as well. However, the re-delegation limit of the second element (which is 6) is smaller than that of the first element (which is unbounded); thus, the first element becomes (3, 6)(2, unbounded). Consequently, the pruning algorithm is invoked, but since there are no previous elements containing the (3, unbounded)(2, unbounded) element, no pruning is performed. Finally, the concatenation process for the (3, unbounded)(2, unbounded) element is terminated (stated in the step b-i-3 in the obligation generation algorithm) because it has been discovered that the element (3, unbounded)(2, unbounded) is alone sufficient to satisfy both participants of the dot product. Thus, it should not further be concatenated with any additional elements. The algorithm continues with element (5, unbounded), at row 4.

After the pruning:

Temporary Policy Obligation: {path:2,
(3, 6)(2, unbounded),
(5, unbounded)(3, 6),
(5, unbounded)(6, 6)}

Since all the rules are exhausted in this set, the final policy obligations below are generated:
(3, 6)(2, unbounded)(5, unbounded)(3, 6)Policy Obligation 3= {path:2, (5, unbounded)(6, 6)}

Each of above policy obligations alone stands to satisfy the collaboration policy. The first obligation requires both of the delegator 3 edges away and the delegator 2 edges away to delegate their credentials to the delegatee. The second obligation requires both the delegator 5 edges away and the delegator 3 edges away to delegate. Finally, the third obligation requires both the delegator 5 edges away and the delegator 6 edges away to delegate.

After applying the same procedure for the second set of D_U rules, (R4), we obtain the following policy obligations. Note that since there was only a single rule in this set, we skipped generating the dot products between the rule contracts. We generated a separate policy obligation for each delegator inside the rule R4's contract

Policy Obligation 4 = {path:2, (2, 6)}
Policy Obligation 5 = {path:2, (3, 6)}

The fourth obligation requires only a single delegator: the delegator 2 edges away; the fifth obligation also requires a single delegator: the delegator 3 edges away. □

Having processed the rule contracts, we obtain multiple policy obligation objects. In order to ensure that there is no redundancy among these objects, we apply the covering algorithm among policy obligation objects one more time. (Covering algorithm is presented above.) If the two obligations are found to cover one another, we remove the redundant obligation objects as follows:

Policy Obligation Pruning Algorithm:

1. For two policy obligation objects, such that the first obligation is found to cover the second obligation:
    a. Remove the first obligation object
    b. Replace the second element's re-delegation distance with that of the first element if the first element's re-delegation distance is smaller.

The above pruning algorithm differs from the earlier pruning algorithm we presented. The above algorithm prunes the redundant policy obligations, whereas the earlier algorithm prunes the redundant element across the rule contracts. Each policy obligation stands on its own to satisfy the collaboration policy. Hence, separate policy obligations maintain an OR relationship among each other. A collaboration request has to satisfy only a single obligation. As a result, when we discover two obligations such that the first one covers the second one, we remove the first one because the first obligation has more delegators listed than the second obligation. To reduce the number of required delegators, we remove the first obligation.

Example 23: (Continuing from Example 1)
At the end of Example 1, we obtained the following obligations:

(3, 6)(2, unbounded)(5, unbounded)(3, 6)Policy Obligation 3 = {path: 2, (5, unbounded)(6, 6)}

Policy Obligation 4 = {path: 2, (2, 6)}

Policy Obligation 5 = {path: 2, (3, 6)}

The covering algorithm discovers that the first obligation covers the fourth and the fifth obligations; therefore, the first obligation can be removed. The second obligation covers the fifth obligation; hence, the second obligation is removed. The final policy obligations are listed below:

Policy Obligation 3= {path: 2, (5, unbounded)(6, 6)}
Policy Obligation 4 = {path: 2, (2, 6)}
Policy Obligation 5 = {path: 2, (3, 6)}. □

The resulting policy obligation objects are attached to the policy decision. The permit decision along with the obligations is returned to the PEP. This concludes the evaluation of D_U rules.

5.2.2 Evaluation of Type D_D Rules

The evaluation of type D_D rules occurs last, after the evaluation of type *L*, *U* and D_U rules. The evaluation of D_D rules occurs only when the PEP determines that the authorizing peer can join the collaboration. The result of the D_D rules does not change the decision over whether the authorizing peer can join the collaboration. The D_D rules only determines, after joining the collaboration, whether the authorizing peer should allow downstream delegation of its credentials to other collaborative peers. The authorizing peer's decision to delegate its credentials affects the requesting peers. A requesting peer, which discovered that its own credentials are not sufficient to join the collaboration, may request the delegation of credentials so that it can join the collaboration. In such cases, the authorizing peer's decision whether or not to delegate affects the requesting peer's ability to join the collaboration.

All of the collaboration requests are evaluated against the D_D rules included in the collaboration policy. To determine whether the collaboration request matches any D_D rule, each of the XACML requests contained within a collaboration request is examined. If an XACML request has a *subject* element with a downstream interaction with the authorizing peer, and the XACML request's *resource* element indicates the authorizing peer's credentials, and the XACML request's *action* element indicates the "delegate" action, then the XACML request should match a D_D rule. Of course, if there is no D_D rule in the collaboration policy, then it is concluded that the XACML request does not match any of the rules. This means that the delegation of requested credentials is not allowed under any circumstances. Once a matching D_D rule is found, the collaboration request, more specifically the matching XACML requests contained in the collaboration request, is evaluated against the matching rules.

For the remainder of this section, we call the requesting peer (i.e. subject of the matching XACML request) that requests the authorizing peer's credentials the delegatee, and we call the authorizing peer the delegator. If there are any peers that must relay the credentials from the delegator to the delegatee, they are called intermediate peers. The PDP retrieves the interaction type between the delegator and the delegatee, and determines the number of edges (i.e. the distance) between the delegator and the delegatee. If the distance exceeds the delegation distance set by the D_D rule, the XACML request returns a deny decision. Otherwise, the D_D rule starts evaluation.

A D_D rule has two inner predicates, each of which states the access requirements sought from a delegatee and the intermediate parties. The delegatee is evaluated against the access requirements given by the first inner-predicate of the D_D rule. The intermediate peers are evaluated against the access requirements given by the second inner-predicate of the D_D rule. In order to return a permit decision from the D_D rule, both inner-predicates must evaluate to true. The XACML request that matches the D_D rule has the delegatee as its subject. Therefore, the attributes of the delegatee can be retrieved from its XACML request, and they are applied to the first inner predicate of the D_D rule.

If there are any intermediate peers, however, their XACML requests (hence their attributes) must be retrieved separately. The PDP first identifies the intermediate peers that must be involved in relaying the credentials. To accomplish this, the PDP looks up the interaction attribute of the delegatee. This attribute not only indicates the interaction type but also the upstream path ID that the delegatee belongs to and the number of hops between the delegatee and the authorizing peer (i.e. the delegator). The PDP retrieves the sub-collaboration graph that is sent by the CLM, and retrieves the upstream path that contains the delegatee. The peers between the delegatee and the delegator are identified as intermediate peers. The PDP then searches for the XACML requests that belong to the intermediate peers. The collaboration request must contain the intermediate peer's XACML requests, because each collaboration request contains all downstream peers, and the delegatee and the intermediate peers are downstream peers. Recall that while preparing the collaboration requests, we grouped the peers based on their upstream paths; however, we left a single group of downstream peers. Once the XACML requests belonging to the intermediate peers are found, their attributes are evaluated against the second inner-predicate of the D_D rule. If an intermediate XACML request cannot be found, it is evaluated to deny.

Unlike type D_U rules, the type D_D rules are not evaluated in a preemptive manner. The type D_D rules are evaluated only when there is a matching XACML request inside the collaboration request. Therefore, there is no need to seek for *potential* delegatees or intermediate peers. The result of each D_D rule is stored in a rule contract object. The rule contract only contains a single delegatee, the intermediate peers between the delegatee and the authorizing peer, and the name of the credentials that are being delegated.

When there are multiple D_D rules that manage delegation of the requested credentials (they each match the same delegatee's XACML request), each of the rules is evaluated separately. The rule results are combined with respect to the rule combining logic of the policy. When the combined result is permit, it is determined that the delegation of the credentials is allowed.

When multiple D_D rules match the same XACML request, the rule contract with the smallest delegation distance replaces the other rule contracts. In other words, only the rule contract with the smallest delegation distance is used to generate a policy obligation object. The other rule contracts are discarded. All rule contracts have the same delegatee and the same credentials that are being delegated; however, the rule contracts only have different delegation distances. The rule contract with the smallest delegation distance becomes a policy obligation. The policy obligation object contains the identities of the authorized delegatee, the intermediate parties and the delegated credentials.

Each XACML request that matches a type D_D rule is evaluated separately, in the manner described above. A separate policy obligation for each matching XACML request is generated. Once all XACML requests are exhausted, the evaluation of a single collaboration request concludes.

Since a single collaboration request already contains the entire downstream peers, it is sufficient to only evaluate a single one of them. The resulting policy obligations are sent to the PEP. This concludes the evaluation of the proposed collaboration.

Chapter 6: The System Architecture

Our architecture consists of four modules: Collaboration Locator Module (CLM), Authorization Management Module (AMM), Policy Enforcement Point (PEP), and Policy Decision Point (PDP) (Figure 6.1).

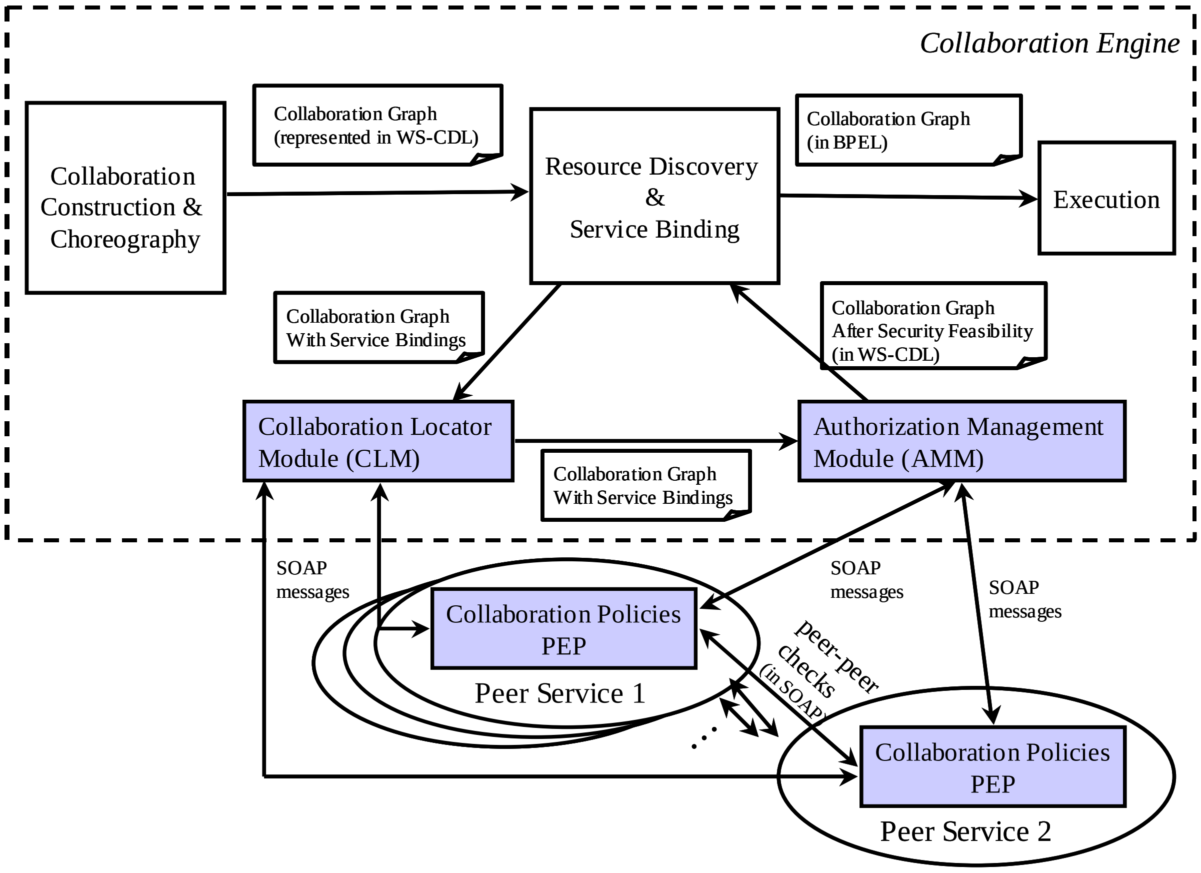

Collaboration Engine
Collaboration Construction & Choreography
Collaboration Graph (represented in WS-CDL)
Resource Discovery & Service Binding
Collaboration Graph (in BPEL)
Execution
Collaboration Graph With Service Bindings
Collaboration Graph After Security Feasibility (in WS-CDL)
Collaboration Locator Module (CLM)
Authorization Management Module (AMM)
Collaboration Graph With Service Bindings
SOAP messages
SOAP messages
SOAP messages
Collaboration Policies PEP
Peer Service 1
peer-peer checks (in SOAP)
Collaboration Policies PEP
Peer Service 2

Figure 6.1 The system architecture. The shaded boxes correspond to our contributions.

The CLM and AMM modules are designed to be incorporated into the collaboration management framework. The collaboration management framework deals with various aspects of collaboration management, such as the design and the choreography of the collaboration, the discovery of the services that partake in the collaboration, the security, and the execution of the collaboration. In practice, each of these aspects is individually built as separate autonomous modules in order to simplify the architectural design, and the collaboration management engine is regarded as the composition of these modules. For example, existing BPEL [OASIS05] engines can execute a collaboration and deal with fault recovery. Likewise, existing choreography editors enable describing and building collaborations. We loosely describe the collaboration management framework as the collection of all these managerial aspects.

Our work focuses only on the access control aspects of collaboration management. Rather than adopting a holistic approach, we circumscribe our work only to the access control management due to the variety and the complexity of distinct managerial aspects. As a result, we designed our modules, the CLM and the AMM, as standalone architectural modules that only tend to the access control aspects of the collaboration framework. They are built such that they can be plugged into an existing collaboration management framework, so long as the collaboration engine can create and handle the input/output messages expected by the AMM and CLM modules. The AMM and the CLM modules are built as standalone web services that can exchange SOAP messages. Any collaboration engine that has the ability to contact a web service and to generate/process input/output variables could easily employ the AMM and CLM modules.

The CLM and AMM modules require a collaboration choreography document. The choreography document describes the collaboration as an ordered collection of the interactions, where each interaction is a peer-to-peer data exchange between two services. In our work, we used the Web-Services Choreography Description Language (WS-CDL) [W3C05] to create choreography documents. The WS-CDL standard offers a means by which the rules of participation within a collaboration can be clearly defined and jointly agreed to.

The CLM and AMM modules require the choreography document for accomplishing the security evaluations over the collaboration. Upon completing the security evaluations, the CLM and AMM modules determine the feasibility of the collaboration from a security standpoint.

The Policy Enforcement Point (PEP) and Policy Decision Point (PDP) modules are designed to evaluate and prevent security threats against a service that is joining the collaboration. The PEP and PDP modules do not focus on the overall security of the collaboration. Instead, they aim to protect the service from the security threats. They are designed with the objective in mind that they can be easily plugged into a service's existing security system. It is expected that each service would already belong to an existing security domain, and the service has an access control system. The existing access control system is likely to be geared towards evaluating standalone access requests, which are not part of any collaboration. It is important for us to design the PEP and PDP modules such that they can be incorporated into the existing security system with no disruption.

As a result, we designed and built the PEP as a standalone web service that can interact with external world via SOAP messages, and it can interact with the existing security system (as explained in Chapter 1). The PDP module, unlike the PEP, is not exposed to external world; it can only be contacted within the security domain it belongs to. The PEP is designed as a gatekeeper among the service's home domain, the collaboration management engine, and the other collaborative peers; the PEP receives collaboration proposals, contacts other peers' PEPs for examining their requests, invokes the PDP to evaluate the collaboration proposal, and sends the final decision back to the collaboration management engine, more specifically to the AMM.

Each collaborative peer (i.e. service) joining the collaboration must have separate PEP and PDP modules installed in its home organization. Since the PEP is designed as a web service, the peer should have the ability of running web services via a SOAP engine, such as Axis.

In this chapter, we first present the mechanics of each module, and then we discuss the interaction patterns among the modules.

6.1 The Collaboration Locator Module

The Collaboration Locator Module (CLM) is responsible for processing the collaboration choreography document. The choreography document, written in Web Services-Choreography Description Language (WS-CDL) [W3C05] – an XML-based choreography language, describes a collaboration as a composition of peer-to-peer interactions that take place using a jointly agreed set of ordering and constraint rules. In our framework, the peers correspond to the services, and interactions correspond to programmatic data exchanges between the services. The CLM's main responsibility is to process the choreography document and examine the interactions among the services from a security standpoint. Below we briefly discuss the WS-CDL standard in to order to familiarize the reader.

The WS-CDL notation has various elements to describe collaborations such as *interaction, participantTypes, roleTypes* and more. (The element names taken from WS-CDL notation are shown in italics.) Since the CLM's job is to examine interactions between the services, it focuses on *roleTypes* and *interaction* elements. A *roleType* element conveys a collection of behaviors that must be exposed by a participant (i.e. a service). Each *roleType* element is assigned to a specific service. A service has multiple operations, where each operation is a separately invocable programming method. The service's operations are specified in a unique WSDL document. A *roleType* element specifies a behavior by referencing the assigned service's WSDL document. A behavior is explicitly linked to one of the operations listed in the service's WSDL document.

The *interaction* elements are the basic building blocks of WS-CDL documents. An *interaction* element shows the data exchange between two *roleTypes*. An interaction element conveys three important pieces of information for our purposes: two service operations, each of which implements one of the two roles, the data exchanged by each operations, and the direction of data flow. An interaction element also contains other information, such as fault recovery and synchronization issues.

For each service that plays a role in the choreography, the CLM is responsible for generating a sub-collaboration graph. The sub-collaboration graph informs the service about what type of interactions it would engage in upon joining the collaboration. A sub-collaboration graph only includes the parts of the collaboration that are related to the service: the name of the other services that interact with the service, the interaction types, and the data exchanged. In other words, for a specific service, the sub-collaboration graph only includes peers that interact with the specified service. The remaining parts of the collaboration are left outside of the sub-collaboration graph.

Unless there is a limit over the interaction types to be included in a sub-collaboration graph, the sub-collaboration graph would become an approximation of or an identical copy of the entire collaboration graph, because each service eventually has an indirect interaction with another service. To prevent this situation, we expect the services to set their limits on the interaction types to be included in their sub-collaboration graphs. For example, a service may set its limit to examine the peers that have a direct interaction; or the service may request to examine other services that have indirect interactions, but less than 5 edges away. In the latter case, the sub-collaboration graph only includes the services that are at most 5 edges away. We call this limit the evaluation radius, the discussion of which is presented in Section 6.1.3 in detail. Each service must specify an evaluation radius for their upstream and downstream directions. The values of the upstream and downstream radiuses may be different.

The CLM builds a sub-collaboration graph for each service that plays a role in the collaboration. The sub-collaboration graph is generated in two separate phases: one for the upstream direction and one for the downstream direction. In both phases, the same generation algorithm is used. The final sub-collaboration graph is composed of results from both directions (Figure 6.2).

In order to generate a sub-collaboration graph for a specific service, the CLM follows the following steps. Let us call this service the owner of the sub-collaboration graph, to avoid confusion. The CLM selects a direction and obtains the evaluation radius in that direction. The CLM checks each *roleType* element of the choreography document, and marks the roles that are assigned to the owner service. It is possible that an owner service is capable of tackling multiple different roles in the collaboration (through different service operations); thus the owner service may be assigned to multiple roles. In such cases, the owner service has multiple graphs generated for each role. Each sub-collaboration graph has a unique identification number and only pertains to a single *roleType*.

For each role assigned to the service, the CLM identifies all the *activities* involved with the *roleType*. The WS-CDL defines various types of *activities* that must be performed by a specific role. In our implementation, we are only interested in *activities* that require an interaction between two services: the *interaction activity* type in WS-CDL notation. If an *activity* is solely accomplished by the service itself without engaging in any interactions with another peer, then the CLM does not analyze this activity and does not include it in the sub-collaboration graph. Instead, the CLM marks the *activities* that are interactions between two peers. This type of activities is described by the *interaction* element of the WS-CDL syntax. Thus, the CLM searches for the *interaction* elements that pertain to the specific *roleType*. Some of the information conveyed by an *interaction* element are: the names of the services involved, the names of the services' operations that would handle the exchange (referring to the operation types in WSDL), the direction of the data flow, and exchanged data types.

The CLM selects the interactions based on their data flow direction. If the sub-collaboration graph is generated for the upstream direction, then the CLM is interested in interactions in which the data flows into the owner service. If the sub-collaboration graph is generated in the downstream direction, the CLM is interested in interactions in which the data flows out of the owner service. By processing the selected *interaction* elements, the CLM identifies the services that are exchanging data with the owner service. We call these services interaction partners.

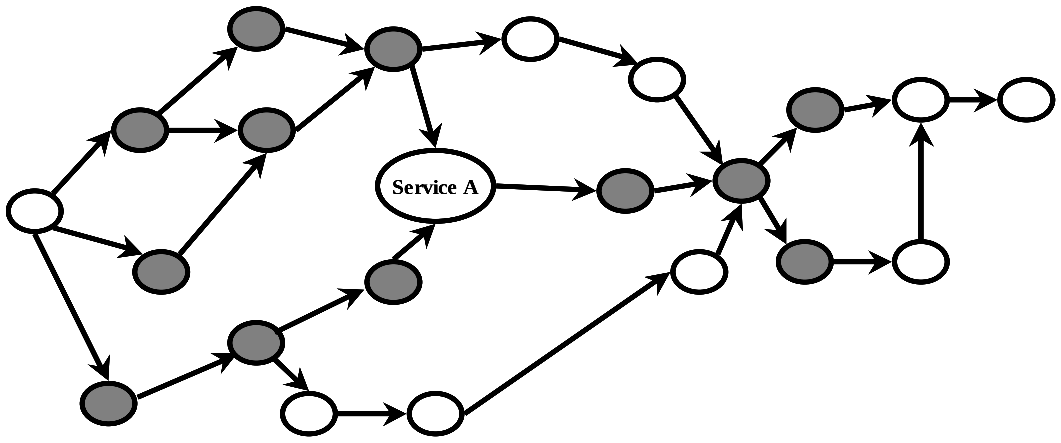


Figure 6.2. The sub-collaboration graph for Service A. Only shaded nodes are included in the sub-collaboration graph of Service A.

It is possible that the owner service may have multiple interactions with the same interaction partner. In such cases, the CLM examines whether the interactions are duplicates of each other, or are different. This is done from the perspective of the owner service. If two interactions have the same data exchange and use the same operation of the owner service, they are determined to be duplicates. If two interactions have a different data exchange or use different operations of the owner service, they are determined to be different. When determining duplicity, we do not distinguish between the operations of the interaction partner. Even if the operations of the interaction partner are different, when the same operation and data is used by the owner service, we conclude that two interactions are duplicates of each other. When two interactions are found to be duplicates of each other, only one of them is included in the sub-collaboration graph. If the interactions are different, each of the interactions is listed separately. Thus, the same interacting partner appears as many times as the number of different interactions in the same graph. For example, in Figure 6.3, Service C has two original interactions with Service B; therefore, Service B appears twice in the sub-collaboration graph of Service C.

The reason for not distinguishing between the operations of an interaction partner is that each operation has the same credentials, which are inherited from the service, and thus the operations are indistinguishable to the external world. From Service C’s perspective, in Figure 6.3, all of Service B’s operations are equal in the sense that they have the same credentials. On the other hand, C may have fine-grained access control rules associated with each of its own operations, such as B is allowed access to an insensitive operation, while it is not allowed for a sensitive operation. This kind of fine-grained policy is not the norm; nevertheless our access control model allows them as exceptional cases. Therefore, when we generate the sub-collaboration graphs, we inform the owner service about which of its operations is engaged in an interaction with another service.

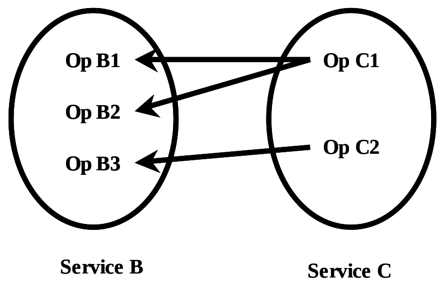


Figure 6.3 The multiple interactions between two services. From Service C’s perspective 2 original interactions: OpC1—OpB1 and OpC2—OpB3. From Service B’s perspective 3 original interactions: OpB1—OpC1 and OpB2—OpC1 and OpB3—OpC2

Once an original interaction is found (non-duplicate), the CLM lists the following data in the sub-collaboration graph: the interacting partner’s name, its interaction type, and the name of the owner service’s operation. For each separate interaction, a unique path ID is assigned.

The path ID is later used to figure out which services are consecutively connected to each other in the sub-collaboration graph.

Once the interaction partners are identified, the CLM repeats the above steps recursively for each interaction partner: the CLM searches for the *roleType* elements that the interaction partner is assigned to, processes the *interaction* elements pertaining to these *roleTypes*, and finds the interaction partners of the interaction partner. This recursive process continues until it reaches the evaluation radius of the owner service, or until it finds a service that does not have any *interaction* elements.

During the recursive execution, there are a few differences. First, the path ID generated by the owner service is passed onto the interaction partner and later to its own partners. Thus, the services that are on the same path are identified. Second, if the interaction partner has multiple interactions with another service (let's call it the third service for brevity), only one of these interactions is listed in the sub-collaboration graph, even when the interactions are different (Figure 6.4). As explained above, two interactions between two services can differ when either the data exchange or the set of service operations is different from each other. However, representing these interactions multiple times in the sub-collaboration graph does not help the owner of the sub-collaboration graph. From the owner's perspective, it has an indirect interaction with the third service through its interaction partner. The owner service has no control or knowledge over the domains of neither its interacting partners nor the third service. Thus, the owner service cannot possibly distinguish these multiple interactions, which are going on between the interacting partner and the third service. Thus, from the owner service's perspective, the multiple interactions are indistinguishable in the security threat they pose against the owner service. This is the reason why the sub-collaboration graph includes only one of these interactions. Of course, programmatically, we can easily list the multiple interactions; however, we are not convinced if this significantly boosts the owner service's security evaluations.

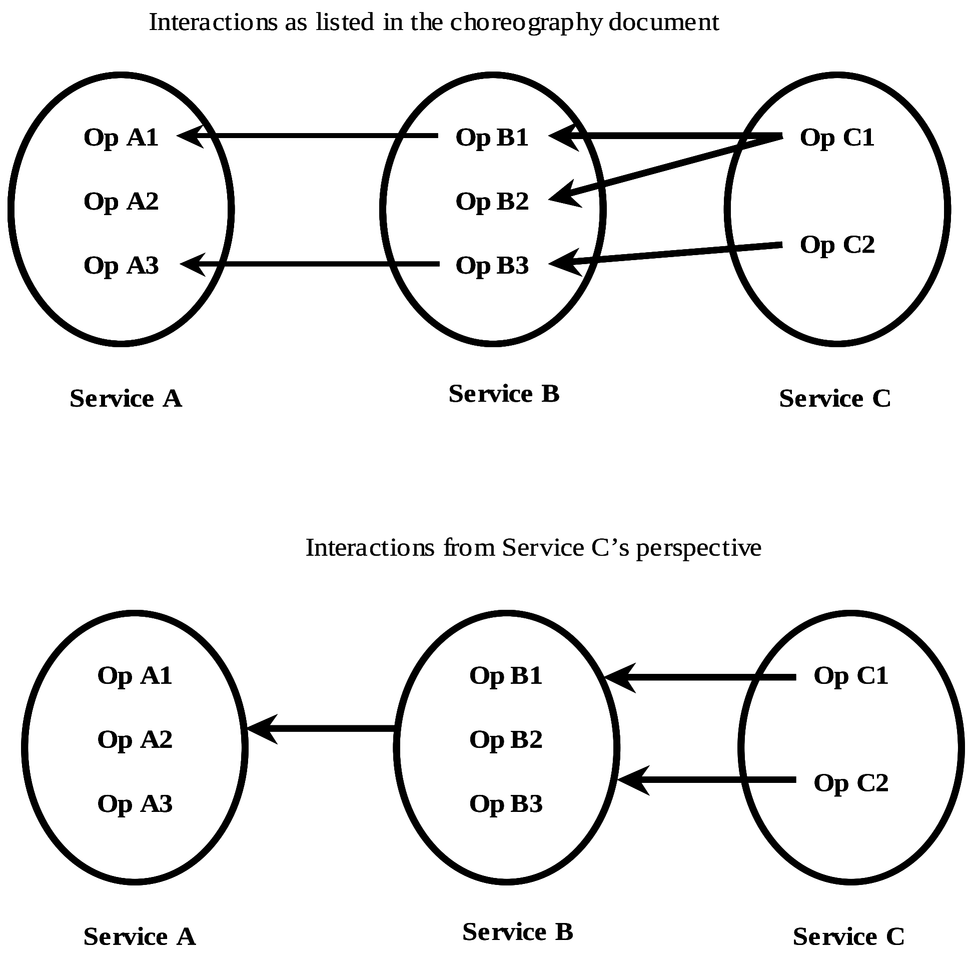


Figure 6.4 Multiple interactions. From Service C's perspective, it has two interactions with Service B; Service B has a single interaction with Service A. Service C is not aware of the details of the interaction between Service A and Service B such as the operation names.

In order to complete the sub-collaboration graph, the above process is repeated in both directions separately, upstream and downstream. The CLM generates multiple sub-collaboration graphs if the owner service plays multiple roles in the collaboration. Each sub-collaboration graph shows all the services that are within the owner service's evaluation radius and interacts with the owner service. Each sub-collaboration graph is assigned a unique identification number. The resulting sub-collaboration graphs are sent to the owner service (discussed in Section 6.5, message#4).

6.2 The Authorization Management Module

The Authorization Management Module (AMM) has two main responsibilities: collecting services' decisions over the collaboration proposal and determining whether the collaboration is feasible under these decisions.

Each service that is proposed to join the collaboration must evaluate its security policies and respond back to the AMM with its policy decision. The decision could either be permit or deny. The deny decision indicates that, for security reasons, the service refuses to join the collaboration. The permit decision comes in two flavors: either with policy obligations, or without any obligations. The AMM is responsible for ensuring that all services have turned in their policy decisions before it starts analyzing the decisions. If one of the policy decisions does not arrive before the designated time-out (10 seconds in current implementation), the AMM considers the missing decision as deny.

If a policy decision is deny, the AMM determines that the current collaboration is infeasible as it is. The AMM records the deny decision and the sender service's name. Although it is beyond the scope of our current work, it is possible that the AMM would make some adjustments to the current collaboration in order to redeem it to a feasible status, such as replacing the refusing service with another one, or replacing another service that fails to meet the sender service's access requirements.

If a policy decision is permit with no obligations, the AMM simply records the message and the service name, and continues with the remaining policy decisions. If the message is permit with policy obligations, the AMM starts evaluating the feasibility of the carried obligations.

A policy obligation states that only when the obligation is satisfied, the promised permit decision is granted. In our framework, a policy obligation indicates that the delegation of credentials between two services is necessary. (See Chapter 4.)

A policy obligation is only satisfied when the delegator agrees to delegate its credentials to the delegatee through the designated intermediate services. Otherwise, the obligation fails and the requested service revokes its permit-with-obligation decision, leading to the failure of the current collaboration.

For each policy decision with obligations, the AMM extracts the set of obligations. A policy decision can have multiple obligations. Each obligation belongs to a specific sub-collaboration and a path within that sub-collaboration (indicated by sub-collaboration ID and the path IDs). (Recall that each sub-collaboration graph is generated for a specific role.) When multiple obligations are present, each of the obligations must be satisfied. If there are multiple obligations that have the same path ID and the same sub-collaboration ID, they are considered to be alternatives of each other; it is sufficient to satisfy only one of them. In other words, for each path of a sub-collaboration graph, there must be at least one feasible obligation.

Recall that each policy obligation can have multiple elements, where each element identifies a potential delegator and the acceptable re-delegation distance for that delegator. When there are multiple elements, hence multiple delegators, in an obligation, all of the delegators must be willing to delegate their credentials. If even one of the delegators refuses to delegate its credentials, the obligation fails.

The AMM is responsible for notifying the delegators about the delegation and collecting their responses. To accomplish this, the AMM fetches a delegator from the obligation. The AMM retrieves the sub-collaboration graph that belongs to the owner of this policy obligation, i.e. the service who sent the policy decision and the obligation. By consulting the sub-collaboration graph, the AMM identifies the delegator. Moreover, by using the specific path ID and the allowed re-delegation distance, the AMM identifies the intermediate services that must also partake in the delegation, as well as the delegatee. The AMM sends messages to the delegator, the intermediate peers and the delegatee in order to inform them about the delegation. The message contents are discussed in Section 6.5.

For each delegator within the obligation, the AMM repeats the above steps. Once all parties are informed, the AMM starts waiting for the responses. (The message exchanges between the delegator, the intermediate parties and the delegatee is explained in Section 6.5.)

Each delegator must return a reply stating whether or not it would like to delegate its credentials. This reply is either permit or deny, and it cannot have any further obligations. The AMM waits until all delegators return a response. Even when a single delegator refuses the delegation, the obligation is concluded as infeasible.

As we stated earlier, each policy decision can carry multiple obligations. The AMM is responsible for assessing each obligation. In order to call a policy decision as satisfiable, at least a single obligation for each path ID must be feasible. Upon assessing the obligations, the AMM determines whether or not the policy decision is satisfiable.

Once all policy decisions are examined in the above manner described, the AMM determines whether the collaboration is ready for execution stage as it is. To start the execution stage, all the services that play a role in the collaboration must return a permit decision. If the permit decision is accompanied with the obligations, the obligations must found to be feasible.

6.3 The Policy Enforcement Point

Each service that plays a role in the collaboration must have the Policy Enforcement Point and the Policy Decision Point modules installed. The Policy Enforcement Point (PEP) functions as a gatekeeper for security evaluations. The PEP exchanges messages with with other service's PEPs, the CLM, the AMM, and its accompanying PDP module. The PEP's main objective is to protect its own domain and to evaluate the security threats associated with joining the collaboration. To accomplish its objective, the PEP performs several duties: (1) informing the CLM about the scope of the security evaluations required by, (2) collecting access requests from other services and consolidating them into collaboration requests, (3) having the PDP evaluate the collaboration request, (4) preparing the policy decision and the policy obligations, and (5) sending access requests to other PEPs when the mutual evaluation is needed.

Several of the above duties have already been discussed in different parts of this thesis. A detailed discussion of (2), (3) and (4) is presented in Chapter 2. The duty listed in (5) is not discussed in this section because it is related to the discussion of message contents and interaction patterns; it can be found in Section 6.5. As a result, in this section, we solely focus on the duty listed in (1). In order to get a holistic view of the PEP, and how the above duties are performed in an order, the reader can refer to Section 6.5 that presents all operations performed by the PEP, without discussing their inner mechanics.

6.3.1 The Scope of a Collaboration Policy: The Calculation of Evaluation Radiuses

The service protected by the PEP must have a collaboration policy. Each collaboration policy has a scope that must be calculated by the PEP. The interaction types that must be evaluated by a collaboration policy constitute the scope of the collaboration policy. Recall that each rule of the collaboration policy targets a different interaction type. The accumulation of all target interactions within a policy constitutes the scope of the policy. For example, the scope of a collaboration policy that only has rules targeting direct interactions is the direct interactions. Another policy having rules that targets the direct interactions and the indirect interactions within a 3-hop radius has a scope of direct and indirect interactions within a distance of 3 hops.

In order to determine the scope of a policy, the PEP must identify which interaction types are required by each rule. In addition, an aggregate target must be calculated for the entire policy such that it includes all of the rule targets. Such an aggregate target must be conveyed to the CLM in order for the CLM to identify the other services that possess the indicated interaction types. Consequently, the CLM would generate the corresponding sub-collaboration graphs for the service protected by this PEP.

We call such an aggregate target the "evaluation radius" of a collaboration policy. The evaluation radius points to the peer services that must be evaluated by the policy. An evaluation radius shows which interaction types, thus which collaborative peers corresponding to these interactions, must be evaluated. An evaluation radius is either represented by a keyword or an integer value. When it is a keyword, it is one of the *direct* or *all* keywords; the former shows that only the peers with a direct interaction is needed for the evaluation, while the latter shows that all the peers that have an indirect or direct interaction must be evaluated. When it is specified as an integer, it shows the maximum number of hops allowed between a collaborative peer and the service protected by the PEP in a specific direction. Any peer located beyond this distance is not required for any security evaluations. The PEP generates a separate evaluation radius for each direction, upstream and downstream. The radiuses in two directions can have different values.

For each direction, the PEP consults the collaboration policy and collects all the required interaction types in that direction. To achieve this, for a specified direction, the PEP collects the *Target* elements of each rule such that the rule's target interaction must be in the same direction as the specified direction.

Recall that the *Target* element of a collaboration rule has the *PeerLocation* element embedded inside, and the *PeerLocation* element has a *direction:interaction* pair. The interaction can be a keyword, such as *direct*, *indirect*, *EndRequestor*, *all*, or it could be an integer showing the maximum number of hops between a requestor and the service. The integer form is most frequently used by the type D_U or D_D rules in order to limit their delegation distances. In our prototype, the target interactions of the D_U or D_D rules are limited with respect to their specified delegation distances regardless of their *Target* elements. For example, if the delegation distance is set to 3, no matter what the rule's *Target* element conveys, the target interactions for this rule are the direct interaction and the indirect interactions within a 3-hop radius.

Once the PEP collects all the required interaction types, it calculates the evaluation radius for the entire policy. The rules' target interactions are combined in an accumulative manner;

the result of the combination covers the target interactions of all the rules. When the target interactions are represented as integers, the largest integer becomes the evaluation radius. When the target interactions are represented as keywords, an *indirect* target is assumed larger than a *direct* target, and an *indirect* target is assumed equal to an *all* target. For example, combination of a *direct* interaction target with an *indirect* interaction target points to *all* interactions. Finally, when a target interaction represented with a keyword is combined with another target interaction represented with an integer, the result is calculated as follows: if the first keyword is *all* or *indirect*, it is returned as the evaluation radius, if the keyword is *direct*, its value is substituted with the value of 1, and the largest integer of the two is returned as the evaluation radius.

It is possible that a policy's evaluation radius can quickly run up to large distances, or result in evaluation of *all* the collaborative peers. To be able to limit the evaluation radius, we use the *MaximumEvaluationRadius* element. When the *MaximumEvaluationRadius* element is left unspecified, it is treated as infinity; the evaluation radius calculated from the policy is used with no adjustment. However, when it is specified as an integer, it is compared against the evaluation radius calculated from the policy. The smaller of the two is chosen as the final evaluation radius.

For example, if the calculated radius indicates the evaluation of *all* interaction types (i.e. direct and indirect interactions together), and if the *MaximumEvaluationRadius* is set to an integer, say 5, then the evaluation radius is adjusted to 5. Only the collaborative peers within a distance of 5 hops are required to be evaluated by the policy. The *MaximumEvaluationRadius* is directionless, meaning that the same value is used for limiting the evaluation radiuses in both directions.

6.4 Policy Decision Point

The Policy Decision Point (PDP) has a single responsibility: evaluating the collaboration requests. The PDP is not exposed to the external world and can only be contacted by the PEP. The PDP module is not built as a standalone web service; it is built as a software module that can only accept connections from the PEP. The PDP receives the collaboration request from the PEP and returns the policy decision back to the PEP. For a detailed discussion of PDP, we refer the reader to Chapter 5.

6.5 The Interaction Patterns Among the Modules

Our framework regulates the interactions that can occur among the modules; it specifies the content and the order of the exchanged messages. Our framework is composed of two separate rounds of interactions among the modules. Each round has a different message collection and a different message order. On an abstract level, each round has a different goal to achieve, and once completed, gives us a chance to review and recover from unexpected failures. Thus, the separate rounds allow us to capture and process the state information of the collaboration, and to determine the course of upcoming interactions. Even though our approach introduces two more layers of abstraction, it helps us design the framework in an efficient and error-free way.

6.5.1 Round-One

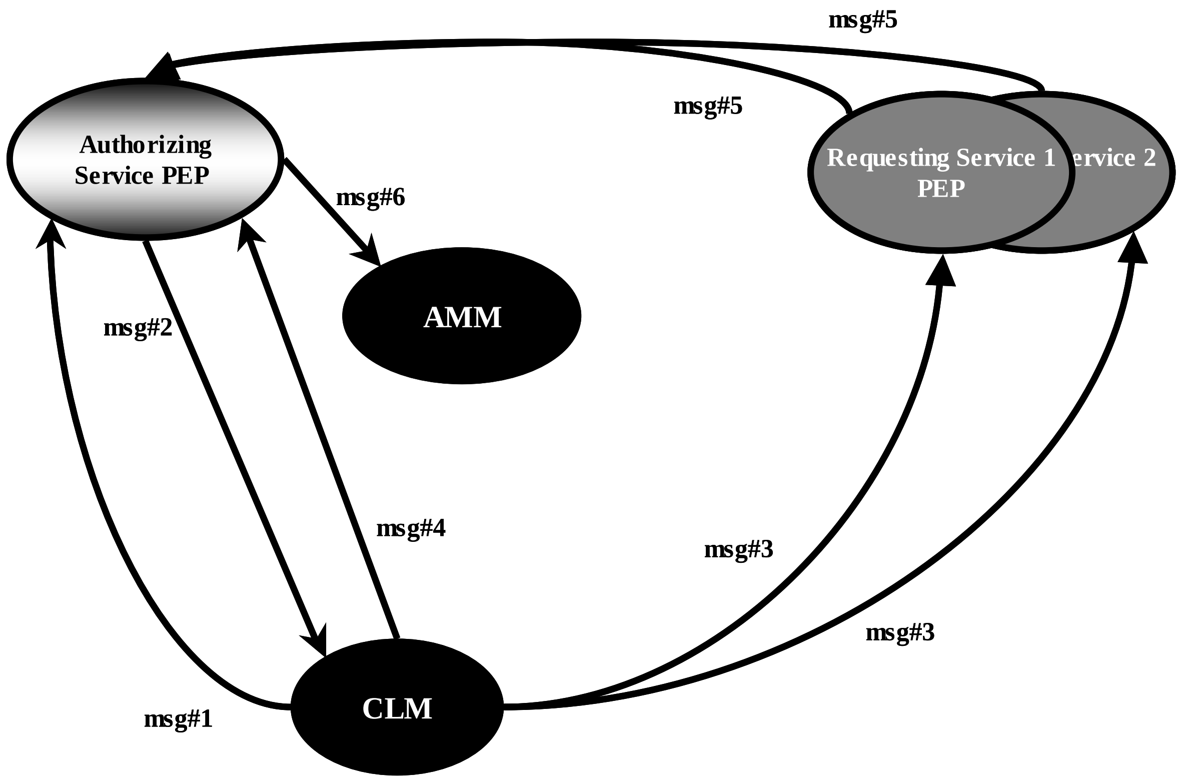


Figure 6.5 Round One of our framework.

The first round of interactions (Figure 6.5) informs the collaborative peers (the services) about the collaboration proposal, and collects their decisions on the proposal. At the end of this round, each collaborative peer sends the result of its security evaluation to the AMM. The AMM, then, initiates the second round of interactions (if necessary).

The first round of interactions is initiated when the collaboration management engine invokes the CLM with a choreography document and the list of services that are tentatively assigned to each *roleTypes* elements. The choreography document specifies the roles that are played in the collaboration, the ordered interactions between the roles, the services assigned to the roles and so on.

By consulting the choreography document, the CLM identifies the services that are assigned to a role. For each service, the CLM contacts the service’s PEP and sends a collaboration proposal message (message#1). This message has the name of the collaboration, the requested service’s name and the URL of the AMM. The URL of the AMM is passed explicitly so that the service’s PEP can return its policy decision directly to the AMM, which collects the results from all the PEPs and determines the course of upcoming interactions. The service name indicates the URL of the service that is tentatively assigned to a role. It is a tentative assignment because the service has not yet confirmed that it will play the role. In case, a service’s PEP manages access to multiple separate services (the PEP’s security domain owns multiple services), the name of the requested service would allow the PEP to distinguish among the services. The final piece of information is the name of the proposed collaboration. The collaboration name is used for keeping track of the messages exchanged for a specific collaboration. In case the PEP receives multiple collaboration proposals for the same service, or if it receives multiple separate collaboration proposals from the same CLM, the collaboration name serves as an identifier.

For the rest of this chapter, we call a service that desires to evaluate other services an authorizing peer, whereas we call the services that are being evaluated the requesting peers. Thus, each PEP that receives the message#1 plays the role of an authorizing peer. We later show how an authorizing peer interacts with the requesting peers, as well as how an authorizing peer plays the role of a requesting peer to other authorizing peers.

Upon receiving message#1, each PEP fetches the collaboration policy that manages access to the requested service. By consulting the policy, each authorizing peer’s PEP generates two evaluation radiuses, one in each direction. (Refer to Section 6.3.1 for more detail in evaluation radiuses.) Upon calculating the evaluation radius, each authorizing peer's PEP creates a message (message#2) that consists of the evaluation radius, the collaboration name, and the requested service name. This message is sent back to the CLM.

For each message (message#2) received, the CLM marks the sender as an authorizing peer. Moreover, from each message, the CLM extracts the evaluation radiuses, consults the collaboration choreography, and identifies the peers that possess the requested interaction types. Each identified peer would play the role of a requesting peer to the marked authorizing peer. The CLM creates a sub-collaboration graph for each authorizing peer. In case the authorizing peer plays multiple roles in the collaboration, a separate sub-collaboration graph with a unique ID is created for each role. Since each role has a different set of behaviors, hence different interactions, it is likely that the sub-collaboration graphs generated per role differ from each other.

In order to notify the requesting peers, the CLM sends a message (message#3) to each of them. This message conveys the name of the collaboration, the URL of the authorizing peer's PEP, the name of the authorizing peer (i.e. the service URL), the name of the requesting peer (i.e. the service URL), and the sub-collaboration ID. It is possible that a requesting peer can appear in two different sub-collaboration graphs. This means that the authorizing peer interacts with the same collaborative peer while playing two different roles.

Iteratively, for each authorizing peer, the CLM repeats the above steps: identifies all the requesting peers and notifies them via separate messages. The CLM also informs each authorizing peer about the list of requesting peers (message#4). The CLM prepares a sub-collaboration graph for each authorizing service. Based on this graph, the CLM prepares the message#4 that consists of: the collaboration name, the name of the authorizing peer, the names of the requesting peers, their interaction types and their sub-collaboration IDs. For each separate sub-collaboration graph, the message#4 repeats the names of the requesting peers, their interaction types and their sub-collaboration IDs. As a result, each authorizing peer knows how many requesting peers they have and what type of interaction each of them possesses. This information is recorded by the authorizing peers and by the CLM, and it is later used for verification.

Once an authorizing peer receives message#4 from the CLM, it starts waiting for the access requests from each of its requesting peers. Since the CLM first sends the message#3 to the requesting peers, and then sends the message#4 to an authorizing peer, it is possible that some requesting peers might already send their access requests to the authorizing peer even before the message#4 reaches the authorizing peer. In that case, the authorizing peer's PEP simply stores the received access requests and later compares them against the message#4 to identify which collaboration and sub-collaboration they belong to.

All of the requesting peers must send their access requests (message#5) to the authorizing peer's PEP. An access request includes the requesting peer's credentials, the collaboration name, the sub-collaboration ID, name of the requesting peer, and the name of the authorizing peer.

If a requesting peer fails to send its access request to the authorizing peer, the authorizing peer waits until the time-out expires (10 seconds in our implementation). The authorizing peer's PEP creates an empty access request for each missing request.

If the information conveyed in an access request cannot be validated against the information sent by the CLM (message#4), the access request is considered invalid and it is treated as a missing access request.

The CLM repeats the above steps for all of the authorizing peers: notifies the corresponding requesting peers and sends the sub-collaboration graphs. Since each peer that is assigned to a role is treated as an authorizing peer, every peer plays the role of an authorizing peer and the requesting peer sometime during round one. It is likely that a peer may play both roles simultaneously. While waiting for messages from its requesting peers, the authorizing peer may also receive a notification from the CLM to send its access request to another authorizing peer.

Once an authorizing peer receives all of the access requests from its requesting peers, it starts evaluating these. (Refer to Chapter 5 for the evaluation of requests.) The resulting policy decision is returned to the AMM (message#6). The message#6 contains the collaboration name, the name of the authorizing peer, the sub-collaboration IDs and the policy decisions associated with each sub-collaboration ID. The policy decision is represented either as *permit* or *deny*. If the policy decision has any obligations, the obligations are immediately listed after the policy decision.

The AMM is responsible for collecting policy decisions from each authorizing peer. Once the policy decisions are collected, the first round of the interactions is concluded.

6.5.2 Round-Two

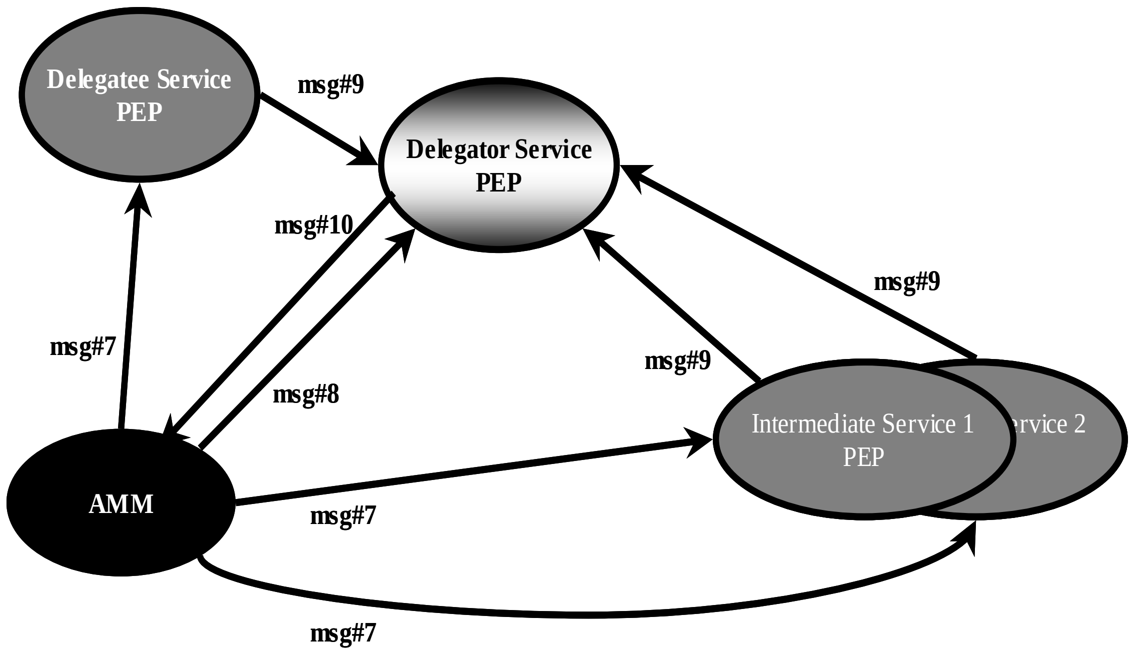


Figure 6.6 Round Two of our framework.

The second round of the interactions (Figure 6.6) aims to determine whether the collaboration is feasible from the security standpoint. The peers' decisions on joining the collaboration and their obligations to do so are evaluated in this round. The outcome of this round must be a conclusion over whether the choreographed collaboration is feasible for execution or not. When an overall agreement among the peers cannot be reached, the outcome of this round becomes negative, which signals to the collaboration management engine that the existing collaboration cannot be performed as it is. Even though it is beyond the scope of our work, succeeding this round, the necessary adjustments, such as replacing some services assigned to the roles, must be made by AMM in order to create a feasible and an overall-agreeable collaboration. Our work in this round is circumscribed to analyzing the peers' policy decision and measuring the feasibility of their obligations to join the collaboration.

The second round of the interactions is started by the AMM when each peer's policy decision (message#6) is collected. For each received message, AMM checks the policy decision, and if present, the policy obligations. For messages with deny policy decisions, the AMM takes no action. It simply records the result for future adjustments. For messages with permit decision, if there are no obligations present, the AMM takes no action and the message is stored away.

For the messages with a permit decision along with policy obligations, the AMM has to examine the feasibility of the obligations. For each obligation, the AMM takes the following actions. The AMM identifies the delegator, delegatee and intermediate peers. In order to notify the delegatee and the intermediate peers about the delegation, the AMM creates a message for each of them (message#7): the name of the collaboration, the URL of the delegator's PEP, the name of the delegatee or the intermediate peer (whichever one suits the specific message), and the name of the delegator. The collaboration name once again serves as an identifier. The location of the delegator's PEP is necessary because the delegatee and the intermediate peers must send their delegation requests to the delegator's PEP. The name of the delegatee/intermediate peer indicates the service that receives/relays the delegated credentials at execution time. The name of the delegator indicates the service that owns the credentials that are to be delegated.

The AMM sends a different message to the delegator's PEP (message#8). This message conveys the collaboration name, the name of the delegator (in case the PEP manages multiple services), the name of the delegatee and the intermediate peers, and their interaction types with the delegator. This message, similar to the message#4 in the first round, is recorded by the delegator's PEP to validate the delegation requests that would be coming from the delegatee and the intermediate peers.

Each intermediate peer and the delegatee prepare delegation requests (message#9) to be sent to the delegator's PEP. The message#9 includes the collaboration name, the credentials of the delegatee/the intermediate peer, the name of the delegatee/the intermediate peer, and name of the delegator. These requests are evaluated by the delegator to determine whether or not to delegate the requested credentials.

The delegator's PEP waits until all the delegation requests are received. The delegator's PEP handles the missing or late requests in the same manner as in round one. Once all the requests are collected, the evaluation starts. The policy decision over the delegation is stated either as a permit or as a deny. We do not allow for defining obligations over a delegation decision, since it drastically increases the complexity involved. The delegation decision from the delegator's PEP to the AMM (message#10) contains the collaboration name, the delegator service's name, and the delegation decision.

For each obligation contained within a policy decision, the AMM repeats the above steps iteratively. When at least an obligation is found to be satisfiable for each path ID, the policy decision is determined to be feasible. The AMM continues with checking for the remaining policy decisions and their accompanying obligations until all policy decisions are exhausted.

Once each policy decision and its obligations have been checked, the AMM determines the final result of the collaboration. The result in terms of a success or a failure is sent back to the collaboration management engine. When all policy decisions are permits and the obligations are satiable, the collaboration is marked as a success. This final message concludes the second round of the interactions.

6.6 The Security Analysis of Our Framework

We discuss the security of our framework in two different aspects. First, we examine the message-level security, and discuss whether the messaging infrastructure introduces additional threats. Second, we analyze the framework under two different threat scenarios: a malicious peer service, and a malicious collaboration owner. In each scenario, we discuss if the malicious entities can introduce additional threats to the other peers in the collaboration.

The messages between the architectural modules can reveal information about the collaboration graph and policy decisions of the collaborative peers. By eavesdropping on these messages, a malicious party can learn about the collaboration, and the relationships between the peer services. For example, in round two of our framework, the AMM checks a delegator service's willingness to delegate its credentials to a delegatee service. A third party eavesdropping to the messages can determine the trust relationship between the delegator and the delegatee. In order to prevent such security threats, we employ message-level security. Each message exchanged between the modules must be encrypted by the recipient peer's public key. Our prototype does not implement the message-level security; however, we leave it as our future work. In the current implementation, we used X.509 credentials for the collaborative peers. Therefore, adding encryption to a message by the recipient peer's public key is not a big challenge for us. However, our framework is not only tied down to X.509 credentials; SAML attributes, Kerberos tickets or any other attribute certificates can be employed. In such cases, we must ensure that each peer learns each other's public key before the peer-peer evaluations start.

Furthermore, our framework must provide means for enforcing non-repudiation. A message sent by a peer service must be traced back to the sender of the message undoubtedly. For example, a service who accepted to join the collaboration at planning time, but refuses participation at run time, can be held responsible for its decision, if our framework has the means to prove that the service initially agreed to the collaboration. Such a mechanism helps preventing disputes that can occur at run time. To achieve this, we must incorporate signatures into the message-level security. Each peer must sign its messages with its private key before sending the message. Our prototype does not support this feature yet; we leave it as our future work.

Our framework must be robust enough to mitigate security threats that are introduced by malicious parties. We have two distinct scenarios: a malicious peer service and a malicious collaboration owner. In the first scenario, we discuss whether having a malicious peer in the collaboration would introduce additional security threats. Since we plan to incorporate encryption and signature techniques at the message level, the malicious peer cannot forge or corrupt other services' messages. Another security threat is whether a malicious party can learn about the other peer's collaboration policies by examining the collaboration context and the peer's decision to join the collaboration. Even though the messages are encrypted, a malicious peer can compare the sub-collaboration graphs that it received from the CLM. This threat becomes more pronounced when AMM tries to redeem an infeasible collaboration. The AMM replaces some peers, creates new sub-collaboration graphs and informs the affected peers about the changes. A malicious peer can detect the replaced peers in its own sub-collaboration graph. It is crucial that each service's policies are confidential and are not revealed to any other parties. We plan to examine the severity of this threat as our future work; however, our initial take is that it is an unlikely threat. First, if the malicious peer's sub-collaboration graph is not affected by the changes, it would not receive any information about the replaced services. Second, even when the malicious peer is informed about the changes, it may take a high number of changes for the malicious peer to detect the relationships between other services. For example, consider a sub-collaboration graph consisting of ten peers, and a peer service refuses access; therefore, this service is replaced by another service. This replacement may be because either the replaced peer did not allow access to the peers in this specific sub-collaboration graph, or it did not allow access to another peer, which is included in the replaced peer's sub-collaboration graph, but not in the sub-collaboration graph of malicious peer. Furthermore, even within the malicious peer's sub-collaboration graph, the malicious peer cannot know which other peers are found untrustworthy by the replaced peer. It is equally likely that any of the ten peers may have failed the replaced peer's collaboration policy.

In above example, we assume the peer who refuses access declines the collaboration. However, this is a quite straightforward approach. We plan to incorporate more sophisticated methods for redeeming an infeasible collaboration graph. For example, instead of replacing the peer who refuses, we might replace the peer who fails to meet the access requirements of the authorizing peer. This would require some sort of feedback between the AMM and authorizing services; however, the complexity may pay off when we want to keep a specific service in the collaboration, which accomplishes a sensitive role in the collaboration graph.

In the second threat scenario, we assume a malicious collaboration owner. We are curious to examine whether the malicious owner can gain knowledge of trust relationships between the peer services. We regard this as an unlikely security threat because the owner is not informed about the collaboration graph until the collaboration is deemed feasible and starts execution. In other words, the end user is not informed about the collaboration graph until a feasible graph is found. At the beginning of the planning stage, the owner describes its requirements from the collaboration to the planning engine. The services are selected according to these requirements; however, the end user is only informed when a feasible graph is found. In a slightly modified threat scenario, the owner may have its own service that partakes in the collaboration graph. The owner’s service is informed about its sub-collaboration graph. In this case, the threat scenario becomes identical to that of a malicious peer service since we treat all collaborative services equally. Another security threat due to the collaboration owner is that by owning multiple distinct collaborations, the owner may gain knowledge of the peer services’ collaboration policies. This threat is not really effective unless the owner has his own service partaking in the collaboration. As we discussed above, for each collaboration, the owner is only informed about the feasible collaboration. Even when we assume that the owner’s service partakes in the collaboration, this is still an unlikely threat. First of all, the planning engine may select completely different services for different collaborations. Therefore, the owner’s sub-collaboration graph includes different peers each time. It may take a significantly high number of different collaborations to deduce any knowledge from the graphs. We leave this aspect as our future work.

Chapter 7:
Deployment and Measurements

In order to observe our prototype in an active computing environment, we deployed it using the Virtual Computing Laboratory (VCL) of NCSU. We tested our prototype with various choreographies and collaboration policies. In this chapter, we first explain the deployment process and how we collected our performance data. Later, we present the results and discuss them.

7.1 Deployment

The VCL uses image files in order to serve applications to end users. A user selects an image file that contains some applications, and the image is loaded to an available machine in the computing farm. The end user does not know the hardware features of the machine that he image is loaded onto. In order to deploy our prototype into the VCL test bed, we first reserved a base Windows XP machine, and built our prototype from scratch: we copied our source code, the Tomcat and the Axis engines, modified SunXACML libraries and so on, and compiled the system. Once we ensured that the services are performing correctly, we created an image file. This image file later deployed over the VCL machines we would use. Since we do not know the hardware specifics of the machines that our image has been loaded onto, we do not discuss it. Any available machine in the computing farm could have been selected.

Due to the high-demand for the VCL resources, we tested our prototype over a small group of machines: 7 machines. One of the machines was deployed with both the Choreography Locator Module (CLM) and Authorization Management Module (AMM). This machine functioned as both the CLM and the AMM. The remaining machines were setup as peer services that were invited to the collaboration. Each service had the Policy Enforcement Point (PEP) and the Policy Decision Point (PDP) installed.

The PEP, CLM and AMM modules were exposed as web services. All services were deployed into a Tomcat server (5.0.28). They performed the SOAP protocol via Axis engine deployed in the Tomcat server.

Once we reserved a computing node in VCL, we were informed of the IP address of the machine. Before we started a collaboration, we manually input the IP addresses of the peer services into the choreography document so that the CLM and AMM modules would contact the peer services. Once we completed a test run, we logged onto each machine and collected the performance measurements. We used the Remote Desktop Protocol in order to connect to a service node.

7.2 Performance Measurements

We measured the wall-clock time spent executing our framework. We used Java's System.TimeMilliSeconds() method. For CLM and AMM modules, we measured the difference between the time the CLM starts parsing a choreography document and the time the AMM sends its feedback to the collaboration owner about the feasibility of the collaboration. For a service node, we measured the difference between the time it received a collaboration request from the CLM and the time it sent a policy response to the AMM. If a service node participates in the second-round of framework as a delegator (this happens when another peer service requests the delegation of credentials), we measured the difference between the time the service received the delegation request from the AMM and the time it sent its response back to the AMM. For a peer service that participates in the second round of our framework as a delegatee, we measured the difference between the time the delegatee service received a notification from the AMM and the time it sent its request to the delegator.

In order to ensure that each node has enough time to receive messages from their peers, we imposed a waiting period of 10 seconds. Each node waits for 10 seconds from the time it receives the sub-collaboration graph from the CLM to the time it starts the policy evaluation. This was done to ensure that each node receives all the credentials from its peers. Furthermore, we thought that in a network environment where nodes are not uniformly distanced from each other, each service might start policy evaluation at different times. Therefore, their execution times may be different. To prevent this, we assigned a large enough waiting period so that each node started processing close to the same time. Since we focus on measuring the policy processing time, we did not want our results be affected by network delays. We used another 10-second waiting period when a delegator peer receives a delegation request from the AMM. The delegator peer waits for 10 seconds before it starts evaluating the delegation request. In the meantime, the delegatee and intermediate peers must send their credentials to the delegator. Due to these waits, our total execution times are around 10-30 seconds. However, a very small fraction of these times are spent on actual execution. When we present our data below, we subtract the waiting periods. For each service, we present only the time that is spent for processing. For the CLM+AMM node, however, we present the entire execution including the waiting periods. We later show for each test case how much of the total execution time is spent on actual processing vs. waiting.

We tested our prototype with various collaboration choreographies. We started with the simplest choreography and gradually increased the complexity. For a fixed choreography, we changed the collaboration policies as well. Each service is assigned the same collaboration policy. Since the services' interactions are different, their policy decisions are different. The MaximumEvaluationRadius element of a collaboration policy limits the number of interactions that must be evaluated by the policy. This element shows the maximum number of edges between an authorizing peer and a requesting peer allowable. The requesting peers that are beyond this value are not evaluated by the collaboration policy even when their interaction types match the policy. For each test run, we set this value to a different number, such as 1, 2, or 3. We simply call this value the radius for the remainder of this chapter.

During a test case, for each radius value, the measurements are repeated five times. We did not use the data from the first measurement because it was affected by the startup cost of Java compiler. We present the average of the 4 measurements in each test case. Although we present the average values in the following section, we observed that the standard deviation was typically around 10% to 15% of the average values. The highest standard deviation was 26% of the average value. All measurements are in milliseconds.

Our test cases are geared towards understanding the differences in executing different rule types. Due to the small cluster on which we can run our tests, we obtain fairly small data sets. This restricted us from trying our prototype with larger collaboration with complex choreographies. Moreover, the randomness of the hardware that runs our tests made it difficult for us to interpret our results. Hence, our results are an introductory analysis of our prototype; we will not claim to build a formal analysis based on these results. Rather, we use the current results to gain an insight towards where our future work must be focused.

Below we first introduce the collaboration policies we used. Later, we introduce each choreography graph, and present the data collected over a collaboration graph with varying collaboration policies.

7.2.1 Collaboration Policy 1 (L Policy)

Collaboration Policy 1 consists of only Local type rules. It has two rules, combined with a logical AND. Both rules target any of the interaction types that are present in the collaboration. In other words, the rules have a target of any:any. To limit the number of peers that match one of the rules, we use the MaximumEvaluationRadius element. Therefore, when the radius is set to 1, a rule matches only direct-upstream and direct-downstream interaction types. When the radius is 2, a rule matches indirect-upstream, indirect-downstream interactions types that are 2-edges away, in addition to the direct-upstream, direct-downstream interactions. For the remainder of this chapter, we refer to Collaboration Policy 1 as L Policy.

7.2.2 Collaboration Policy 2 (U Policy)

Collaboration Policy 2 consists of only Underlying type rules. It has two rules combined with a Logical AND. Both rules targeted any of the interaction types present in the collaboration, via the keyword any:any. The evaluation of an Underlying rule requires referring to the Underlying Policy, which is part of the Underlying security system for standalone access requests. We specified the Underlying Policy as a separate XACML policy. It only has a single rule. Each service node is provided with a copy of the Underlying Policy in addition to its Collaboration Policy 2. We refer to Collaboration Policy 2 as U Policy from now on.

7.2.3 Collaboration Policy 3 (L+U Policy)

Collaboration Policy 3 includes two rules: one is of type Local; the other one is of type Underlying. The Local rule targets indirect-upstream and indirect-downstream interactions, whereas the Underlying rule targets direct-upstream and direct-downstream interactions.

Both rules are combined with a logical AND. We think Collaboration Policy 3 is a likely choice for the policy writers. A policy writer may want to re-use existing access requirements for a direct-interaction, whereas he may choose to use weaker or collaboration-tailored access requirements for the indirect interaction types. The Underlying Policy remained the same.

7.2.4 Collaboration Policy 4 (L+U+D Policy)

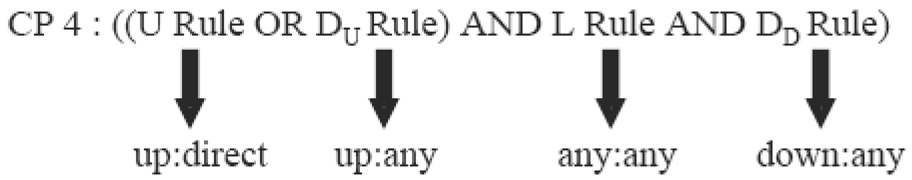


Figure 7. 1 Collaboration Policy 4.

Collaboration Policy 4 is the most complex of all policies. It contains 4 types of rules: Local, Underlying, Delegation-upstream, and Delegation-downstream. The first rule is of type Underlying, and it targets the direct-upstream interactions only. The second rule is of Delegation-upstream and it targets all of the upstream interactions within the DelegationDistance. The third rule is of Local type and it targets all of the interactions, via *any:any* keyword. The fourth rule is of Delegation-downstream type and it targets all of the downstream interactions within the Delegation Distance.

A peer with direct-upstream interaction must satisfy either the Underlying Rule or the Delegation-upstream rule. Otherwise the peer fails the policy. The Delegation-upstream rule allows the upstream-direct peer to use delegated credentials for access. Therefore, an upstream-direct peer must either be authorized against the Underlying Policy, or the peer must receive delegated credentials from an indirect-upstream peer.

7.2.5 The Collaboration Graph 1: The Simplest Case

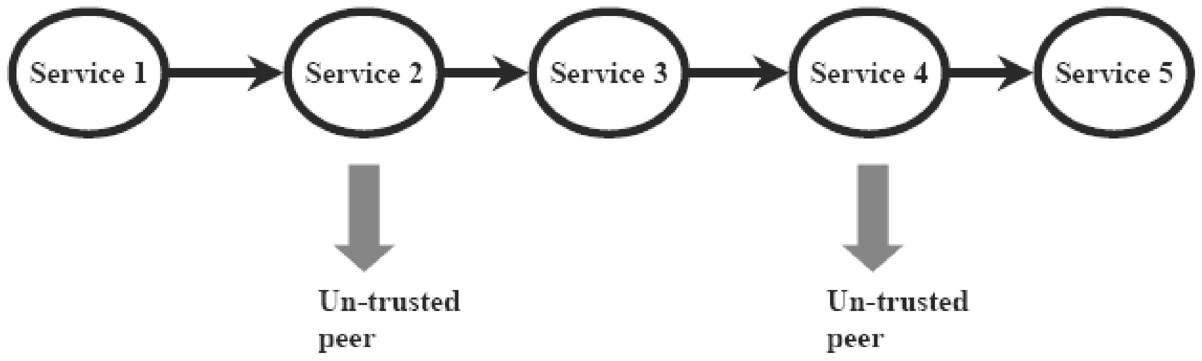


Figure 7.2 Collaboration Graph 1.

We used 6 nodes for our choreography: 5 peer services and a single CLM+AMM node. Each service is connected to one another via a single interaction. The services are named by their appearance order in the choreography, Service 1 through Service 5 (Figure 7.2). We test Graph 1 with all four collaboration policies. The main purpose of our test runs in this section (Cases 1 through 4) is to observe how different rule types affect a service's execution time. In the succeeding section, we change our collaboration graph so that we can observe the affect of the collaboration graph over the execution times.

Case 1: Collaboration Policy 1 (L Policy)

We setup the rules and the peer's credentials such that Service 2 and Service 4 fails the L Policy, whereas, other services are authorized successfully. Although each service has the same collaboration policy, the result of each policy differs with respect to the specific interactions that its service involves in. Table 7.1 shows the policy decisions for each service along with the name of the peers that are evaluated.

Table 7. 1 Services' policy decisions with L Policy over Graph 1.

	Service 1	Service 2	Service 3	Service 4	Service 5
Policy Decision Radius = 1	Deny	Permit	Deny	Permit	Deny
Evaluated Peers	2	1, 3	2, 3	3, 5	4
Policy Decision Radius = 2	Deny	Deny	Deny	Deny	Deny
Evaluated Peers	2, 3	1, 2, 3	1, 2, 4, 5	2, 3, 5	3, 4
Policy Decision Radius = 3	Deny	Deny	Deny	Deny	Deny
Evaluated Peers	2, 3, 4	1, 3, 4, 5	1, 2, 4, 5	1, 2, 3, 5	2, 3, 4

In this test run, we aim to collect the execution time spent for L Policy. Since L Policy is the simplest policy type we specified, we will use the results from this section later as a reference point. Moreover, we aim to observe the affect of increasing radius over the execution times.

Table 7.2 The service execution times with L Policy over Graph 1.

	Service 1	Service 2	Service 3	Service 4	Service 5	CLM+AMM
Radius 1	414	582	504.25	582.25	398.5	10992
Radius 2	656	812.5	840	781.25	629.25	11207
Radius 3	707	949.25	851.25	945.5	707	11316.25

For each service, the time it takes to evaluate the policy and return a response to the CLM increases with the radius. This is expected because the bigger radius means evaluating more collaborative peers. For Service 3, the increase from radius 2 to 3 does not make a big difference because the radius 2 already covers all the peers that are present in the collaboration. When we compare the differences in performance, we notice that evaluating more peers does not linearly increase the response time (Table 7.3). In order to understand the effects of adding more peers, we will later test with different choreographies.

Table 7.3 The difference in execution times with changing radiuses. The first row shows the difference between the radius of 1 and radius of 2; the second row shows the difference between the radius of 2 and radius of 3. All results are in milliseconds.

Service 1		Service 2		Service 3		Service 4		Service 5	
Δ Time	Δ Peers	Δ Time	Δ Peers	Δ Time	Δ Peers	Δ Time	Δ Peers	Δ Time	Δ Peers
242	+1	230.5	+1	335.75	+2	199	+1	230.75	+1
51	+1	136.75	+1	11.25	0	164.25	+1	77.75	+1

Case 2: Collaboration Policy 2 (U Policy)

In order to see whether the rule types affect the performance, we kept the same choreography; however, we used U Policy this time. The service's policy decisions and the number of peers evaluated by each service are presented in Table 7.4. The execution times are presented in Table 7.5.

Table 7. 4 The policy decision for each service with U Policy over Graph 1.

	Service 1	Service 2	Service 3	Service 4	Service 5
Policy Decision Radius = 1	Deny	Permit	Deny	Permit	Deny
Evaluated Peers	2	1, 3	2, 3	3, 5	4
Policy Decision Radius = 2	Deny	Deny	Deny	Deny	Deny
Evaluated Peers	2, 3	1, 2, 3	1, 2, 4, 5	2, 3, 5	3, 4
Policy Decision Radius = 3	Deny	Deny	Deny	Deny	Deny
Evaluated Peers	2, 3, 4	1, 3, 4, 5	1, 2, 4, 5	1, 2, 3, 5	2, 3, 4

Table 7.5 Service execution times for U Policy over Graph 1.

	Service 1	Service 2	Service 3	Service 4	Service 5	CLM+AMM
Radius 1	453.25	656.5	527.5	636.75	453	11000.75
Radius 2	597.5	874.75	831.75	773.75	617.5	11175.75
Radius 3	808.5	1000.25	957.5	945.5	828.25	11359.5

As with L Policy, the policy evaluation time increases with increasing radius (Table 7.5). Moreover, the overhead of accessing Underlying Policy also shows up in the results. The difference between the results of L Policy and the results of U Policy is calculated as follows: for a fixed radius, the execution time of L Policy is subtracted from the execution time of U Policy. This is repeated for each radius and for each service (Figure 7.3). The time differences are sometimes negative, meaning that U Policy is evaluated faster than L Policy

However, for the majority, the evaluation of U Policy takes longer than that of L Policy. This overlaps with our expectations because Underlying type rules requires evaluation of the Underlying Policy, causing an additional layer of policy evaluation. The anomaly observed in radius 2 might be due to a change in the VCL, such as the nodes were only dedicated to run our test scenario. As we discussed before, we do not know which machine our image is loaded onto. Moreover, it may be a server that simulates multiple images concurrently. Therefore, at a specific time, we might have gotten a machine solely dedicated to our image.

We also aim to understand how increasing the radius (i.e. increasing the number of peers evaluated) affects the execution time. However, as shown in Table 7.6, we did not observe a regular repeating pattern. We decide that the obtained results are not conclusive enough to make a decision in this issue.

Table 7.6 The difference between the execution time for changing radiuses with U Policy. The first row shows the difference between the radius of 1 and the radius of 2; the second row shows the difference between the radius of 2 and radius of 3. All results are in milliseconds.

Service 1		Service 2		Service 3		Service 4		Service 5	
Δ Time	Δ Peers	Δ Time	Δ Peers	Δ Time	Δ Peers	Δ Time	Δ Peers	Δ Time	Δ Peers
144.25	+1	218.25	+1	304.25	+2	137	+1	164.5	+1
211	+1	125.5	+1	125.75	0	171.75	+1	210.75	+1

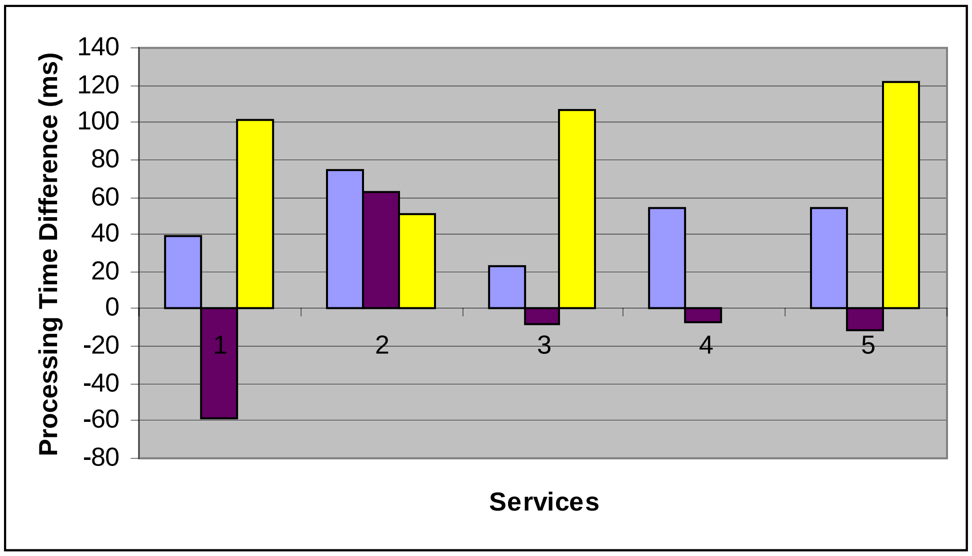


Figure 7.3 The difference in execution times with respect to L Policy and U Policy. The services are listed on the x-axis. Each service has a cluster of three bars. Each bar shows the difference in execution times for a fixed radius: first bar is for the radius 1; second bar is for the radius 2 and so on. A bar shows the difference that is calculated by subtracting the result of L Policy from the result of U Policy.

Case 3: L+U Policy

We tested the same choreography with L+U Policy. Our purpose is to see the affect of combined L and U type rules over the execution time. The services' policy decisions are presented in Table 7.7. The execution times are presented in Table 7.8.

Table 7. 7 The policy decision for each service with L+U Policy over Graph 1.

	Service 1	Service 2	Service 3	Service 4	Service 5
Policy Decision Radius = 1	Deny	Permit	Deny	Permit	Deny
Evaluated Peers	2	1, 3	2, 3	3, 5	4
Policy Decision Radius = 2	Deny	Deny	Deny	Deny	Deny
Evaluated Peers	2, 3	1, 2, 3	1, 2, 4, 5	2, 3, 5	3, 4
Policy Decision Radius = 3	Deny	Deny	Deny	Deny	Deny
Evaluated Peers	2, 3, 4	1, 3, 4, 5	1, 2, 4, 5	1, 2, 3, 5	2, 3, 4

Table 7.8 Service execution times with L+U Policy over Graph 1.

	Service 1	Service 2	Service 3	Service 4	Service 5	CLM+AMM
Radius 1	425.75	617.25	542.75	632.75	452.75	10972.75
Radius 2	691.5	789	839.75	773.75	706.75	11179.5
Radius 3	761.5	972.25	867.25	945.5	769.25	11316.75

We compare the results that are obtained from L+U Policy with that of U Policy and of L Policy. We set the results of L Policy as our reference point and subtract them from the results of U Policy and L+U Policy separately. The Figure 7.4 shows the differences in execution times for a fixed radius of 3. The results show that L+U Policy has faster execution time than U Policy; however, L+U Policy performs slower than L Policy. This correlates with our expectations. In L+U Policy, the Underlying rule is only used for the direct interaction types, whereas in U Policy, all peers are evaluated against the Underlying rules.

Moreover, this shows that L+U policy has no additional overhead due to combined L and U type rules.

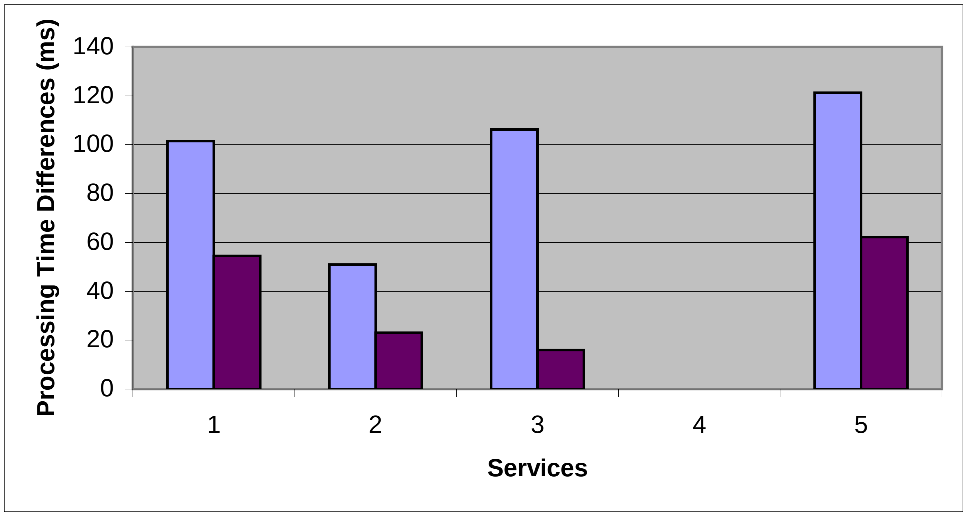


Figure 7.4 The comparison of results for the radius of 3. The services are listed on the x-axis. Each service has two bars. The first bar shows the difference between L Policy and U Policy; second bar shows the difference between L Policy and L+U Policy. All results are measured for radius 3.

Case 4: L+U+D Policy

We finally tested the Graph 1 with L+U+D Policy. We earlier set up the service's credentials such that Service 2 and Service 4 cannot satisfy the Underlying Rule. However, they both can satisfy the Delegation-upstream Rule. Therefore, for example, Service 3 evaluates its Delegation-upstream rule, and returns a permit decision with obligation. The obligation requires Service 1 to delegate its credentials to Service 2 at run-time. Note that Service 5 also returns a permit decision with obligation. Service 5 would require Service 3 to delegate its credential to service 4. In the second round of our framework, the AMM would seek the feasibility of this obligation. It would inform Service 1/Service 3 about the delegation and ask Service 1/Service 3 to evaluate its collaboration policy once more. Service

1/Service3 would evaluate its Delegation-downstream rule and determine whether or not delegating its credential. We setup service credentials such that Service 4 can satisfy the Delegation-downstream rule, whereas Service 2 cannot. Therefore, Service 1 does not delegate its credential, whereas Service 3 accepts to delegate it. Service 1' rejection of delegation would result in failure to meet Service 3's obligation. The AMM would determine that the current collaboration is infeasible for execution.

All services are setup such that their credentials can satisfy the Local Rule successfully. Therefore, the Underlying rule type and the Delegation-upstream rule dictate a policy decision. We present the policy decisions for each service in Table for radiuses 2 and 3.

We start taking measurements from a radius of 2 because Delegation-upstream rule is ineffective for a radius of 1, which basically means no delegation at all. The DelegationDistance is set according to the evaluation radius. When MaximumEvaluationRadius is 2, the Delegation-upstream rule has a DelegationDistance of 2. When the MaximumEvaluationRadius is 3, the DelegationDistance is set to 3. We repeat the measurements for radiuses of 2 and 3.

For this test case, we want to observe the effect of Delegation-upstream rule (i.e. obligation processing) over the execution time. In order to do that, we compare our results from L+U+D Policy with L+U Policy. Moreover, we want to observe the time spent in the second-round of our framework.

Table 7. 9 The policy decision for each service with L+U+D Policy over Graph 1.

	Service 1	Service 2	Service 3	Service 4	Service 5
Policy Decision Radius = 2	Permit	Permit	Permit Obligation: Delegator: 1 Delegatee: 3	Permit	Permit Obligation: Delegator: 3 Delegatee: 4
Policy Decision Radius = 3	Permit	Permit	Permit Obligation: Delegator: 1 Delegatee: 3	Permit	Permit Obligation: Delegator: 1 Delegatee: 3

Table 7.10 Service execution times for L+U+D Policy.

	Service 1	Service 2	Service 3	Service 4	Service 5	CLM+AMM
Radius 2						Total Time: 31484.25
1st Round	648.5	812.5	867	785	644.5	
2nd Round	54.75	4	105.5	7.5	0	
Radius 3						Total Time: 31605.25
1st Round	746	976.5	918	972.75	765.5	
2nd Round	55	7.75	109.5	11.75	0	

As seen from Table 7.10 and Figure 7.5, all service's performed similar to Case 3, which tested L+U Policy. In the first-round, the difference is mainly expected in the results of Service 3 and Service 5 because they evaluate the Delegation-upstream rules in addition to the other rules. As seen in Table 7.9, only Service 3 and Service 5 evaluate the obligations. However, our results show that the evaluation of Delegation-upstream rule does not affect the execution time significantly. Note that the AMM receives two obligations to determine their feasibility, from Service 3 and Service 5. The AMM evaluates the obligations sequentially.

This affects the overall performance of AMM. Due to the waiting periods at each delegator, the AMM's overall execution gets delayed by 20 seconds. We should modify our code to evaluate the obligations in a parallel manner to reduce this time.

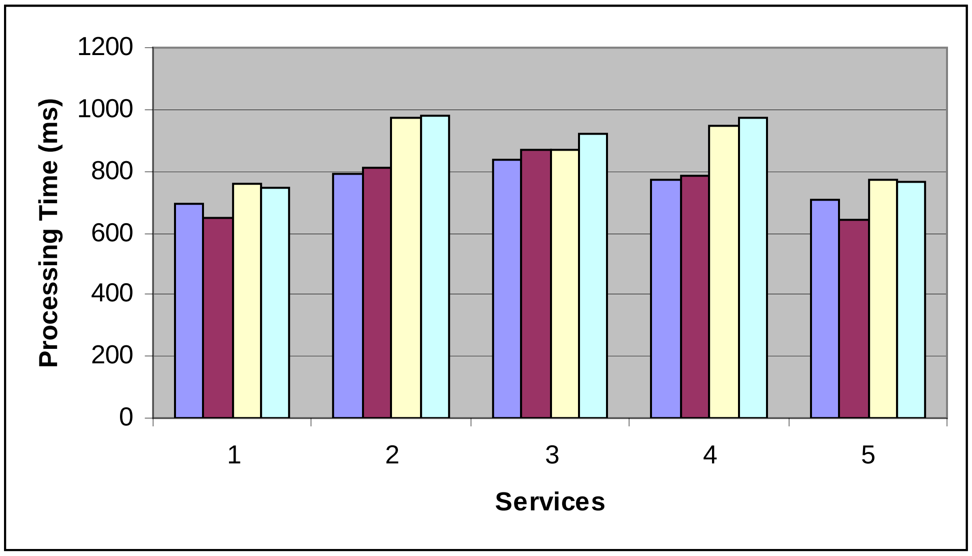


Figure 7.5 Execution time comparison between L+U Policy and L+U+D Policy. Each service has a cluster of 4 bars. First bar shows the execution time of Policy 3 with a radius of 2; second bar shows the execution time of Policy 4 with a radius of 2; third bar shows the execution time of Policy 3 with a radius of 3; fourth bar shows the execution time of Policy 4 with a radius of 3.

For both radiuses, the second-round is only dominated by the delegator service's evaluation speed (Services 1 and Service 3). The time for preparing and sending the delegation request to the respective delegators were almost negligible: less than 12 ms in all cases. Service 3 responded slower than Service 1. This may be because the permit decision takes longer than the deny decision to evaluate.

7.2.6 Collaboration Graph 2: 1-Branch

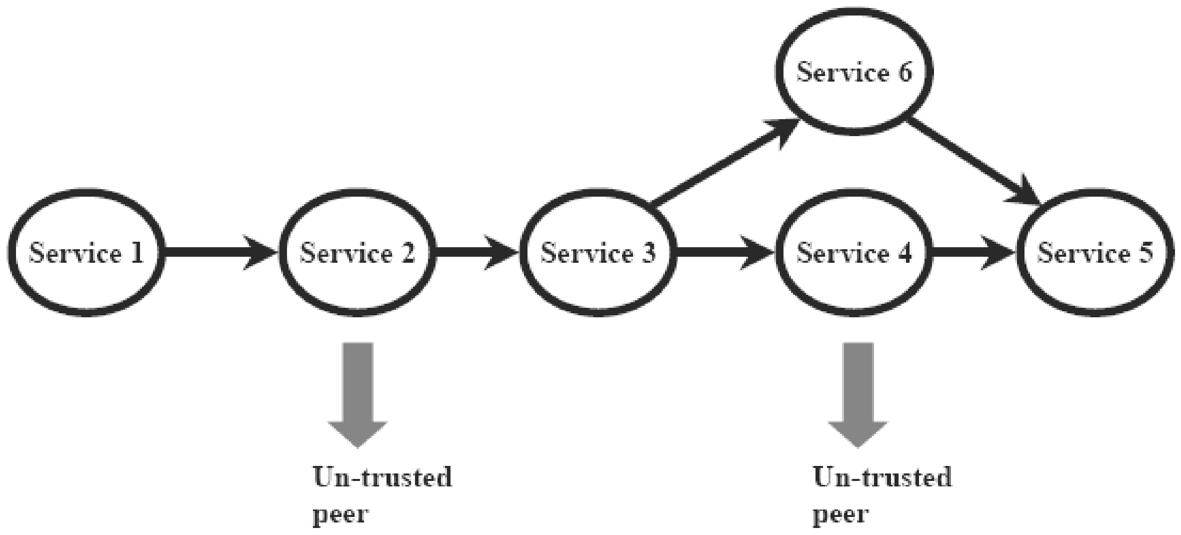


Figure 7.6 The Collaboration Graph 2.

We altered the collaboration graph as follows. We added a branch after Service 3. This increased the total node number to 6. The added node is called Service 6. At run time, Service 3 either interacts with Service 6 or Service 4. However, at planning we have no knowledge of which interaction is going to occur; therefore, we included both interactions in the collaboration graph and treated them as though they both are going to be performed. This was a cautious approach; nevertheless, it was necessary since our focus is on security.

We repeated the L+U Policy and L+U+D Policy with the new choreography. For the remainder, we change our choreography three more times, and each time we repeat L+U Policy and L+U+D Policy. For each test run, we only use a radius of 2 and 3, since radius of 1 is meaningless for L+U+D Policy. For the remainder of this chapter, we focus on understanding the affects of different collaboration graphs.

Case 5: L+U Policy

Case 5 tests L+U policy over Graph 2. Due to the additional branch in Graph 2, each service must conduct a higher a number of peer-peer evaluations than they did over Graph 1. We aim to observe the effect of higher number of peer-peer evaluations in execution times. Therefore, we compare L+U Policy over Graph 1 and Graph 2.

Table 7. 11 The policy decision for each service with L+U Policy over Graph 2.

	Service 1	Service 2	Service 3	Service 4	Service 5	Service 6
Policy Decision Radius = 2	Deny	Deny	Deny	Deny	Deny	Deny
Evaluated Peers	2, 3	1, 3, 4, 6	1, 2, 4, 5, 6	2, 3, 5	3, 4, 6	2, 3, 5
Policy Decision Radius = 3	Deny	Deny	Deny	Deny	Deny	Deny
Evaluated Peers	2, 3, 4, 6	1, 3, 4, 5, 6	1, 2, 4, 5, 6	1, 2, 3, 5	2, 3, 4, 6	1, 2, 3, 5

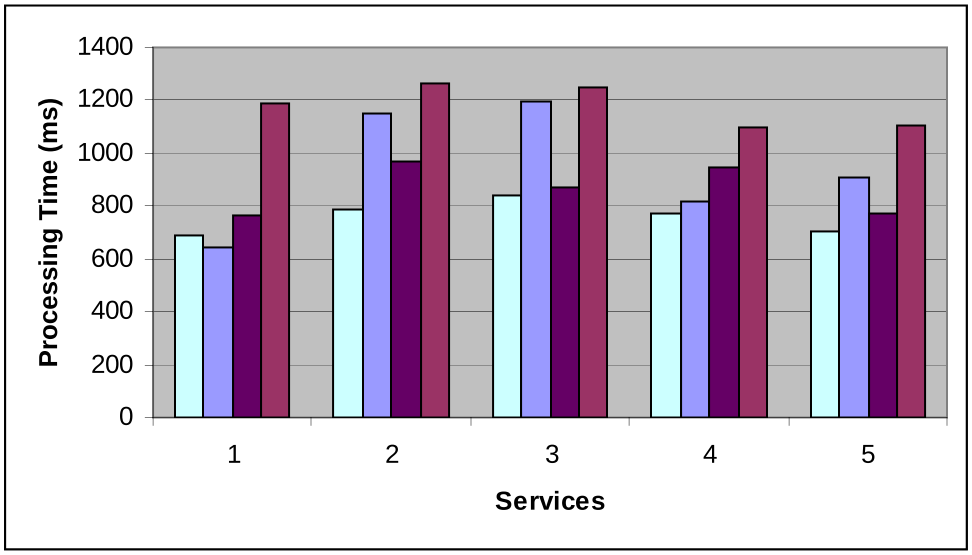


Figure 7.7 Service execution times for Policy 3 over Graph 1 and Graph 2. The results from Service 6 is not shown below because Service 6 does not exist in Graph 1. For each service, first bar shows the execution time for radius 2 over Graph 1; second bar shows execution time for radius 2 over Graph 2; third bar shows execution time for radius 3 over Graph 1; fourth bar shows execution time for radius 3 over Graph 2.

For Radius 2, we expect the results to be similar to that of Graph 1. When the evaluation radius is set to 2, Service 1 and Service 4 have no increase in the number peer-peer evaluations due to the added branch. Therefore, their performances should remain unaffected, whereas all other services must execute slightly slower than Case 3 because they all conduct a higher peer-peer evaluation. The results overlap with our expectations, as shown in Figure 7.7. For radius 3, we expect an increase in the execution time of all peers because all of them evaluate a higher number of peers than that of Case 3 and that of radius 2. The results again support our expectations.

Case 6: L+U+D Policy

Case 6 is a test run of L+U+D Policy over Graph 2. In Case 6, we aim to observe how much evaluation of Delegation-upstream rules affects the execution time. In order to this, we compare results of L+U+D Policy with the L+U Policy over the Graph 2. Each services policy decisions are presented in Table 7.12.

Table 7. 12 The policy decision for each service with L+U+D Policy over Graph 2.

	Service 1	Service 2	Service 3	Service 4	Service 5	Service 6
Policy Decision Radius = 2	Permit	Permit	Permit Obligation: Delegator: 1 Delegatee: 3	Permit	Permit Obligation: Delegator: 3 Delegatee: 4	Permit
Policy Decision Radius = 3	Permit	Permit	Permit Obligation: Delegator: 1 Delegatee: 3	Permit	Permit Obligation: Delegator: 1 Delegatee: 3	Permit

Service 1, Service 2, Service 4, and Service 6 are already returning permit results; they do not evaluate their Delegation-upstream rules. Therefore, their execution times must be identical to that of L+U Policy. Only Service 3 and Service 5 evaluate their Delegation-upstream rules. However, we do not expect a significant difference in the results of Service 3 and Service 5 because from the earlier test runs, we observed that evaluating Delegation-upstream rule does not severely impact the execution time.

Table 7.13 Service execution times with L+U+D Policy over Graph 2.

	Service 1	Service 2	Service 3	Service 4	Service 5	Service 6	CLM+AMM
Radius 2							Total Time: 31820.25
1st Round	668	1105.5	1173	883	941.5	879	
2nd Round	50.5	7.75	120	5	0	0	
Radius 3							Total Time: 32136.75
1st Round	1109.5	1461.25	1304.5	1167.75	1199.25	1086	
2nd Round	50.5	7.75	97.25	3.75	0	0	

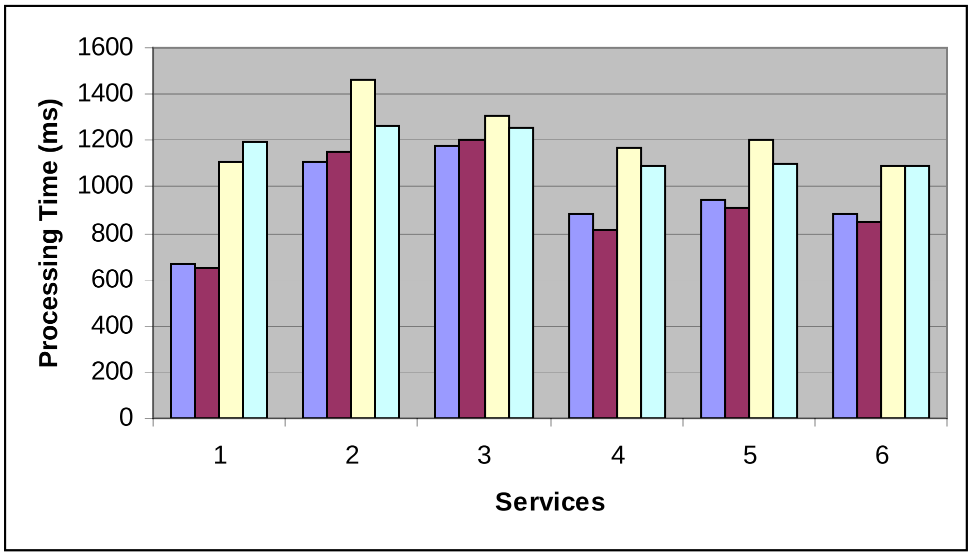


Figure 7.8 Service execution times for L+U Policy and L+U+D Policy over Collaboration Graph 2. The x-axis shows Services 1 through 6 and the CLM+AMM. For each service, first bar shows the execution time for radius 2 with L+U Policy; second bar shows execution time for radius 2 with L+U+D Policy; third bar shows execution time for radius 3 with L+U Policy; fourth bar shows execution time for radius 3 with L+U+D Policy.

The results obtained overlap with our expectations. As seen from Figure 7.8, Service 1, 2, 4 and 6 obtains almost identical results to that of L+U Policy. Service 5 and Service 3 are not severely impacted by the evaluation of Delegation-upstream rules.

For the second-round measurements, we obtain results almost identical to that of Policy L+U+D over Graph 1. The delegatees spend negligible time to prepare and send their delegation requests, whereas the delegators dominate the time spent in round-two. For both radiuses, permit decision from Service 3 takes twice as much as deny decision from Service 1. Service 1 and Service 3 sequentially spends 20 seconds in waiting periods before they start evaluating the delegation requests. This causes an additional 20 sec delay, on top of 10 second spent in the first-round, in the execution time of CLM+AMM.

Our conclusion from Case 6 is that adding a branch slows the overall execution time down, and the evaluation of Delegation-upstream rules does not have a visible affect on the execution times.

7.2.7 Collaboration Graph 3: Double Branching Effect

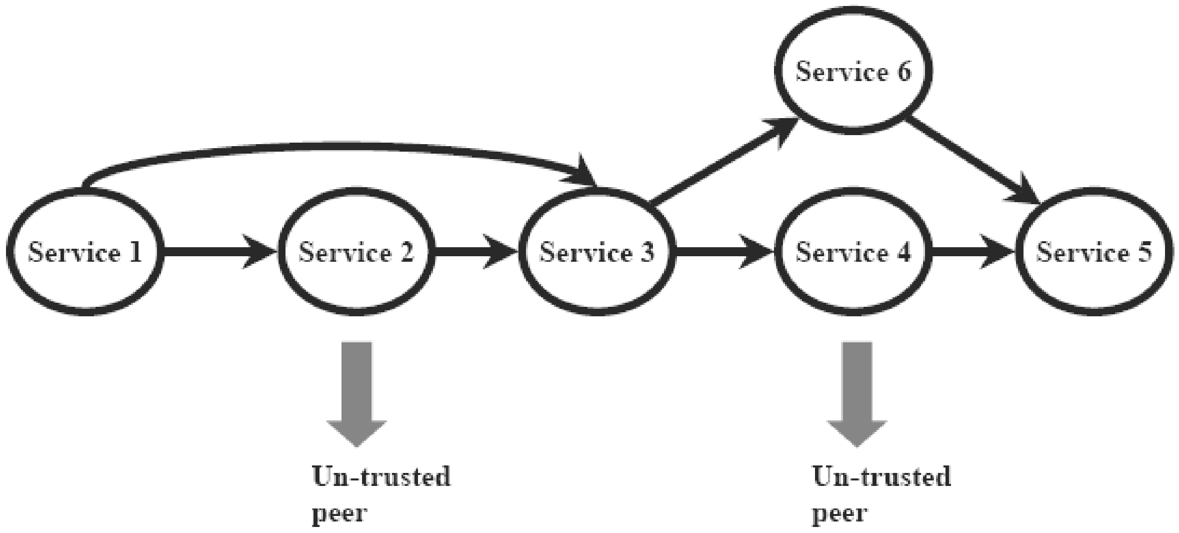


Figure 7.9 The Collaboration Graph 3.

We changed our choreography by adding another branch. The branch is added to the node of Service 1. As a result, Service 1 and Service 3 are connected with a direct interaction. The added branch creates new connections between services that are previously not connected or their connections were beyond the evaluation radiuses. Such as service 1 and Service 5 are now connected when the evaluation radius is set to 3. This should increase the evaluation time for the services because they should evaluate more peers than in earlier cases. We analyzed the new choreography with L+U Policy and L+U+D Policy.

Case 7: L+U Policy

In Case 7, we test L+U Policy over Graph 3. We compare our results with that of L+U policy over Graph 2. We aim to see how much the additional branch affects the execution time. The service policy decisions are presented in Table 7.14.

Table 7. 14 The policy decision for each service with L+U Policy over Graph 3.

	Service 1	Service 2	Service 3	Service 4	Service 5	Service 6
Policy Decision Radius = 2	Deny	Deny	Deny	Deny	Deny	Deny
Evaluated Peers	2, 3, 4, 6	1, 3, 4, 6	1, 2, 4, 5, 6	1, 2, 3, 5	3, 4, 6	1, 2, 3, 5
Policy Decision Radius = 3	Deny	Deny	Deny	Deny	Deny	Deny
Evaluated Peers	2, 3, 4, 5, 6	1, 3, 4, 5, 6	1, 2, 4, 5, 6	1, 2, 3, 5	1, 2, 3, 4, 6	1, 2, 3, 5

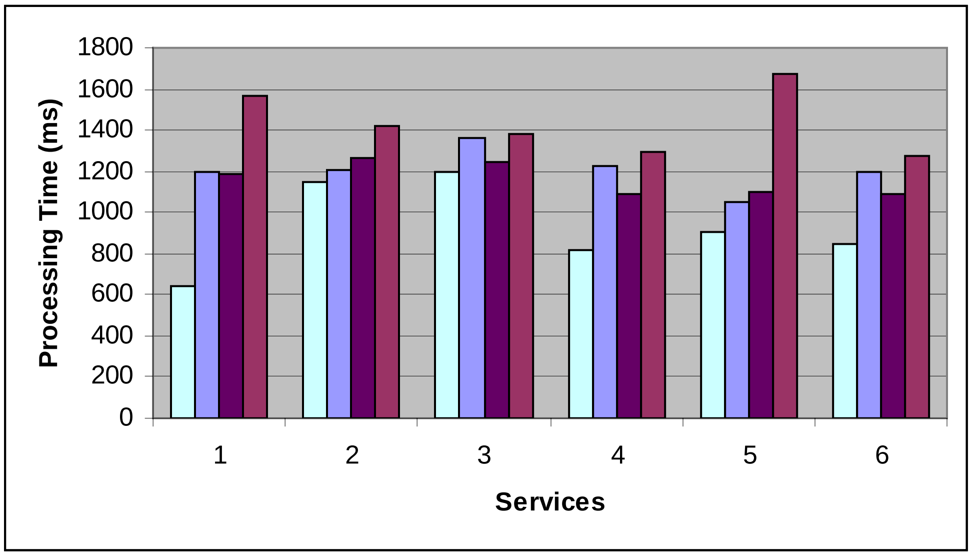


Figure 7.10 Service execution times for L+U Policy over Graph 2 and Graph 3. The x-axis shows Services 1 through 6. For each service, first bar shows the execution time for radius 2 over Graph 2; second bar shows execution time for radius 2 over Graph 3; third bar shows execution time for radius 3 over Graph 2; fourth bar shows execution time for radius 3 over Graph 3.

All services should be affected by the second branch because it increases the number of peer-peer evaluations for each service (Table 7.14). For Radius 3, the worst-case scenario occurs and each service evaluates all other services present in the collaboration. This causes cause slower execution times than that of Graph 1 (Figure 7.10).

Case 8: L+U+D Policy

Case 8 tests L+U+D Policy over Graph 3, and compares the results with results of L+U Policy over the same graph, Graph 3.

Table 7. 15 Service policy decisions with L+U+D Policy over Graph 3.

	Service 1	Service 2	Service 3	Service 4	Service 5	Service 6
Policy Decision Radius = 2	Permit	Permit	Permit Obligation: Delegator: 1 Delegatee: 3	Permit	Permit Obligation: Delegator: 3 Delegatee: 4	Permit
Policy Decision Radius = 3	Permit	Permit	Permit Obligation: Delegator: 1 Delegatee: 3	Permit	Permit Obligation: Delegator: 1 Delegatee: 3 Obligation: Delegator: 1 Delegatee: 4	Permit

In this test run, we are interested in comparing L+U+D Policy results with L+U Policy results over the same graph because L+U+D Policy has a high number of obligations when the radius is set to 3 (3 obligations), (Table 7.15). We want to see how increase in obligations affects the execution time.

Table 7.16 Service execution times with L+U+D Policy over Collaboration Graph 3.

	Service 1	Service 2	Service 3	Service 4	Service 5	Service 6	CLM+AMM
Radius 2							Total Time: 32019.5
1st Round	1199.25	1206.75	1261.75	1202.75	949	1199	
2nd Round	50.75	12	136.5	7.5	0	0	
Radius 3							Total Time: 22251.75
1st Round	1585.75	1386.75	1367	1332	1785.25	1308.5	
2nd Round	50.75	3.75	0	0	0	0	

Since Services 1, 2, 4 and 6 do not evaluate their Delegation-upstream rule, their execution times must be similar to their results from L+U Policy over Graph 3. This is supported by the data collected. The Services 3 and 5 evaluate Delegation-upstream rule in addition to the Local and Underlying rules. However, as before, the evaluation of Delegation-upstream rule does not affect their performance severely (Figure 7.11).

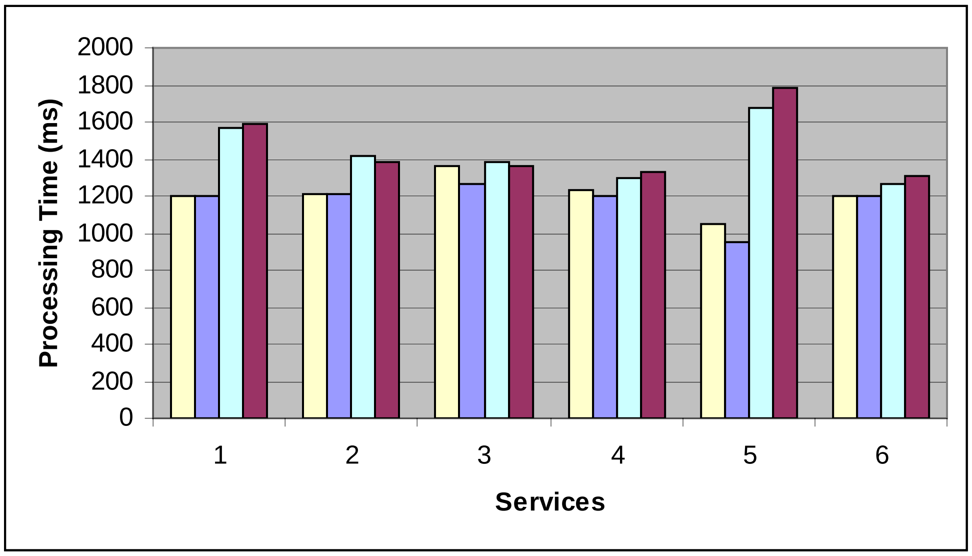


Figure 7.11 Service execution times for Policy 3 and Policy 4 over Collaboration Graph 3. The x-axis shows Services 1 through 6 and the CLM+AMM. For each service, first bar shows the execution time for radius 2 with Policy 3; second bar shows execution time for radius 2 with Policy 4; third bar shows execution time for radius 3 with Policy 3; fourth bar shows execution time for radius 3 with Policy 4.

Note that in the second-round of radius 3, we have only a single obligation that is being evaluated. When radius is set 2, the AMM first checks the feasibility of Service 5's obligation. This requires Service 4 to send a delegation request to Service 3. As before, Service 3 grants this request. The AMM then evaluates the feasibility of Service 3's obligations, which requires Service 1 to delegate to Service 2. However, when we set our radius to 3, the second-round results change. The Service 5's obligation is never evaluated. This is because the AMM first receives the Service 3's obligation. The AMM, therefore, first checks the feasibility of Service 3's obligation. Once Service 1 declines the delegation request, the AMM determines that the collaboration is infeasible as it is. It terminates the second-round before checking the feasibility of Service 5's obligation. Therefore, the time spent for the waiting period is only 10 seconds, instead of 20. This is why for radius 3, the CLM+AMM has a 10 seconds faster execution time than the execution time of radius 2.

7.2.8 *Collaboration Graph 4: Same Service Multiple Appearance*

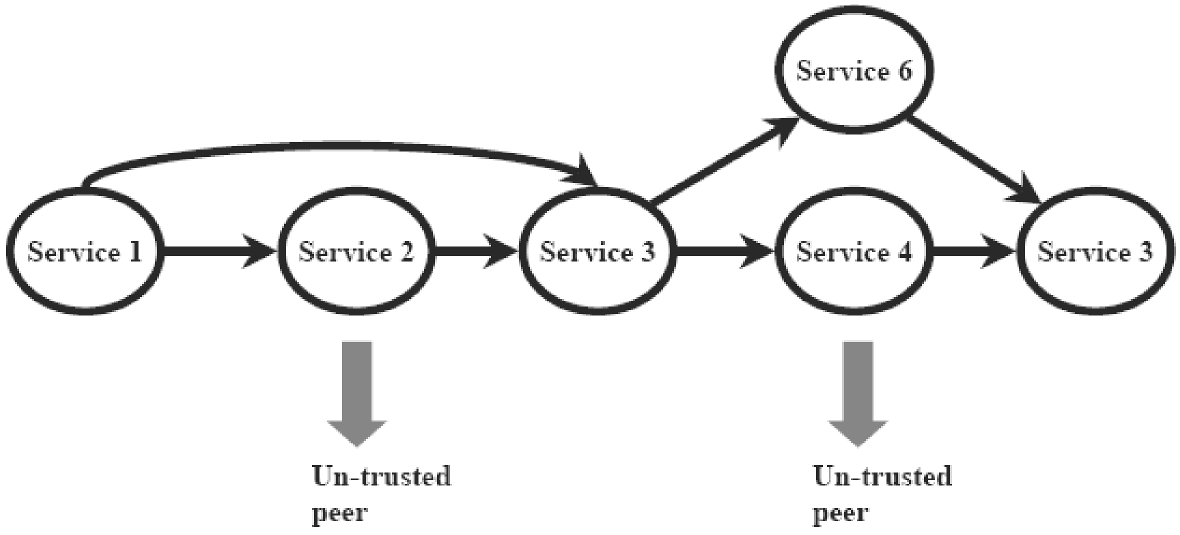


Figure 7.12 Collaboration Graph 4.

We changed the choreography such that Service 3 takes over the responsibility of Service 5, and appears twice in the choreography. Even though Service 3 and Service 5 accomplish different tasks, we assumed that Service 3 had the functionality to accomplish both tasks. This assumption is rooted in the fact that services usually have multiple operations advertised in their WSDL files, and each operation can be utilized for different functionalities. Therefore, a service may be selected to join a collaboration with two different operations. As a result, we wanted to observe the effect of having a service multiple times in a single collaboration. Below, we only present the results pertaining to Service 3, and skip other services' results because their results must be identical to that of Graph 3.

Case 9: L+U Policy

Although we expected a drastic change in Service 3's performance, it remains almost the same. This is because Service 3 evaluates two separate sub-collaboration graphs (a sub-collaboration graph for each appearance of Service 3); however, after evaluating the first sub-collaboration graph, Service 3 returns a deny decision and terminates its evaluation. Therefore, the results collected in this round are not different than that of Case 7, where we tested L+U Policy over Graph 2.

Case 10: L+U+D Policy

Service 3 should evaluate a permit decision with 3 obligations: the first obligation is due to the first appearance of Service 3 in Graph 4; the second and third obligations are due to the second appearance of Service 3 in Graph 4 (formerly Service 5). Service 3 evaluates the three obligations sequentially. This is unlike the previous case, where Service 5 evaluated two obligations in parallel with Service 3. We are interested in how evaluating 3 Delegation-upstream rules in sequential affects the execution time of Service 3.

Table 7.17 Service 3's policy decision with L+U+D Policy over Graph 4.

	Service 3
Policy Decision Radius = 2	Permit Obligation: Delegator: 1 Delegatee: 3 Obligation: Delegator: 3 Delegatee: 4
Policy Decision Radius = 3	Permit Obligation: Delegator: 1 Delegatee: 3 Obligation: Delegator: 1 Delegatee: 3 Obligation: Delegator: 1 Delegatee: 4

Table 7.18 Service execution times with L+U+D Policy over Graph 4.

	Service 3
Radius 2	
1st Round	1617.25
2nd Round	0
Radius 3	
1st Round	2187.25
2nd Round	0

For radius 2, the execution time increased by 355.5 ms (28%); for radius 3, the execution time increased by 820.25ms (60%), (Figure 7.13). For radius 2, Service 3 had one additional obligation, whereas for radius 3, Service 3 had two additional obligations. The increase in number of obligations is reflected proportionally in the execution time. This is expected because our algorithm evaluates the obligations in a sequential manner. However, in our earlier test runs, we have not observed a significant change in execution times due to the obligations. In order to understand if this case is an anomaly due to a change in our execution environment, we conduct one more test case.

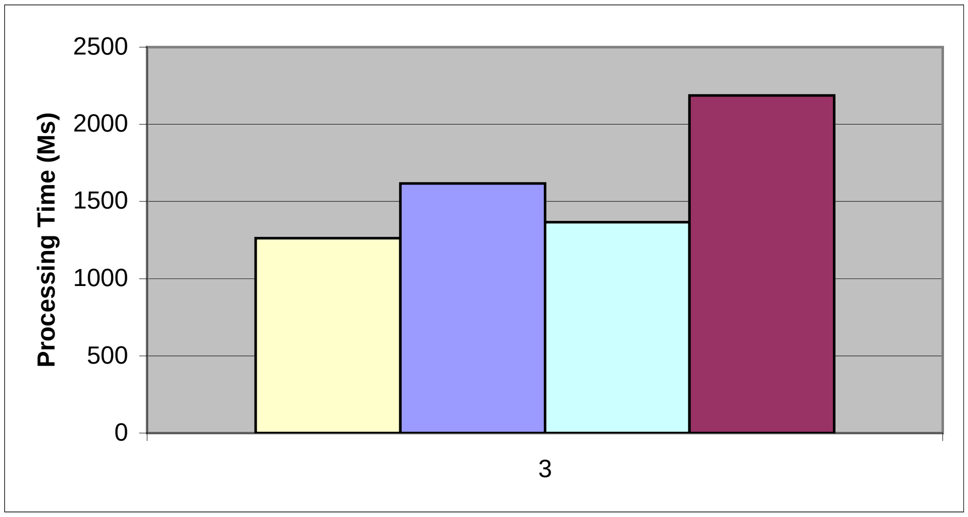


Figure 7.13 Service execution times for L+U+D Policy over Collaboration Graph 4 and Collaboration Graph 3. The x-axis shows Service 3 only. First bar shows the execution time for radius 2 over Graph 3; second bar shows execution time for radius 2 over Graph 4; third bar shows execution time for radius 3 over Graph 3; fourth bar shows execution time for radius 3 over Graph 4.

7.2.9 Collaboration Graph 5: Multiple Interactions

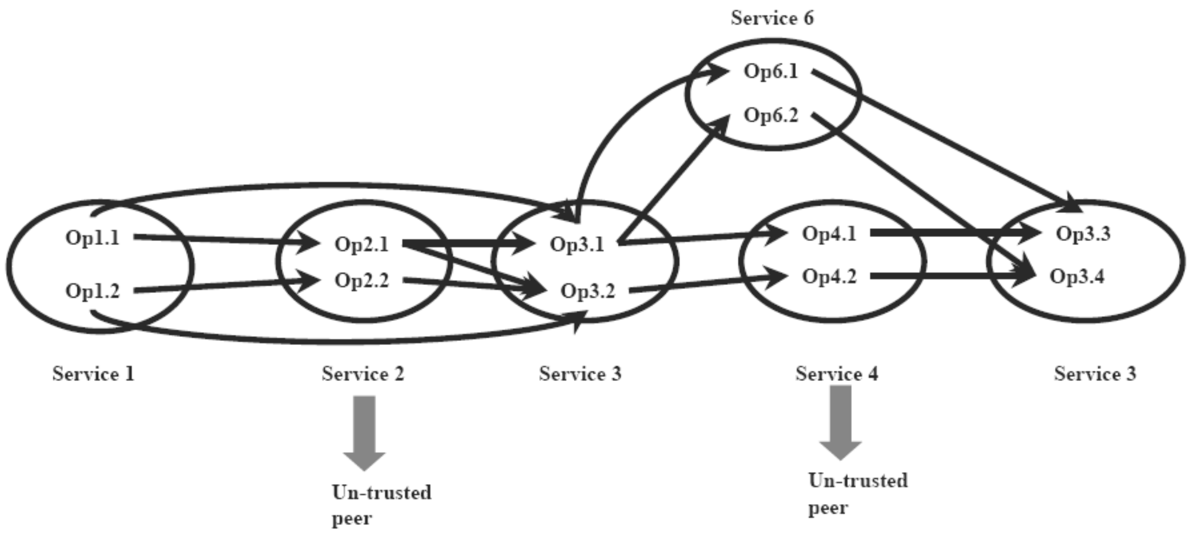


Figure 7.14 The Collaboration Graph 5.

Graph 5 includes multiple interactions between the same two services. This use case happens when a service has multiple operations defined in its WSDL. Such a service can have multiple interactions with another service. We also maintained that Service 3 appears twice in the collaboration graph.

A service identifies an interaction as original when the service's operation interacts with another service that the operation has not interacted before. For example, between Service 2 and Service 3, there appears to be 3 interactions. However, from Service 3's perspective, there are only two original interactions: op3.1 interacts with op2.1; op3.2 interacts with op2.2. The interaction between op3.2 and op2.1 is not original for security purposes because all operations of Service 2 inherit the same credential from Service 2. op3.2 already listed Service 2 in its list of interaction partners due to op2.2, therefore, listing Service 2 again due to the interaction with op2.1 is not going to benefit the security evaluations. (For a full explanation of this issue, reader may refer to Chapter 6 – Discussion of the CLM.) Likewise, from Service 2's perspective, there are two original interactions with Service 3. As a result of multiple interactions, all services have twice as much peer-peer evaluation as the previous choreographies. We observe the affect of increased interactions over the execution time.

Case 11: L+U Policy

Table 7. 19 The policy decision for each service with L+U Policy over Graph 5.

	Service 1	Service 2	Service 3	Service 4	Service 6
Policy Decision Radius = 2	Deny	Deny	Deny	Deny	Deny
Evaluated Peers	2, 3, 4, 6	1, 3, 4, 6	1, 2, 4, 6	1, 2, 3, 5	1, 2, 3
Policy Decision Radius = 3	Deny	Deny	Deny	Deny	Deny
Evaluated Peers	2, 3, 4, 6	1, 3, 4, 6	1, 2, 4, 6	1, 2, 3	1, 2, 3

Due to the increased peer-peer evaluations, all services' execution times should increase significantly. We compare results of L+U policy with Case 9, where L+U Policy is tested over Graph 4. As Figure 7.15 shows, the execution times almost doubles. This overlaps well with our expectations. Note that for Service 3, over the Graph 5, the change from radius 2 to radius 3 does not make a big impact over the execution time. This is because Service 3's first appearance in the graph already evaluates all of its peers with a radius of 2. Also note that second appearance of Service 3 (formerly Service 5) is not evaluated at all because after evaluating its first appearance Service 3 returns a deny decision to the AMM, and terminates its policy evaluation.

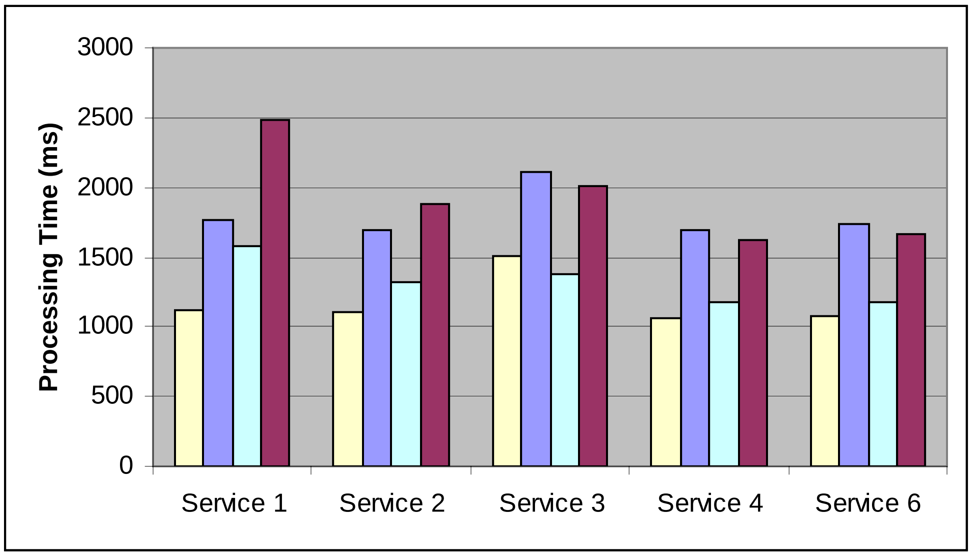


Figure 7.15 Service execution times for L+U Policy over Collaboration Graph 5 and Collaboration Graph 4. Service 5 is not shown because it is not included in Graph 5. For each service, first bar shows the execution time for radius 2 over Graph 4; second bar shows execution time for radius 2 over Graph 5; third bar shows execution time for radius 3 over Graph 4; fourth bar shows execution time for radius 3 over Graph 5.

Case 12: L+U+D Policy

In this case, the number of obligations that are evaluated by Service 3 is significantly higher than that of the previous cases (Table 7.21). We are mainly interested in how this increase affects the execution time of Service 3.

Table 7. 20 Service policy decisions with L+U+D Policy over Graph 5.

	Service 1	Service 2	Service 3	Service 4	Service 6
Policy Decision Radius = 2	Permit	Permit	Permit Obligation: Delegator: 1 Delegatee: 3 — Obligation: Delegator: 3 Delegatee: 4 Obligation: Delegator: 1 Delegatee: 3 — Obligation: Delegator: 3 Delegatee: 4	Permit	Permit
Policy Decision Radius = 3	Permit	Permit	Permit Obligation: Delegator: 1 Delegatee: 3 — Obligation: Delegator: 3 Delegatee: 4 Obligation: Delegator: 1 Delegatee: 3 — Obligation: Delegator: 3 Delegatee: 4 Obligation: Delegator: 3 Delegatee: 4 — Obligation: Delegator: 3 Delegatee: 4	Permit	Permit

Table 7.21 Service execution times with L+U+D Policy over Graph 5.

	Service 1	Service 2	Service 3	Service 4	Service 6	CLM+AMM
Radius 2						
1st Round	1613.25	1492.25	2496.25	1523.75	1562.5	22980.5
2nd Round	54.75	3.75	0	0	0	
Radius 3						
1st Round	2418	1898.25	3656.25	1629	1730.5	24160
2nd Round	46.75	4	0	0	0	

We compare the current results with that of Case 10, where we use L+U+D Policy over Graph 4. The only difference between Graph 5 and Graph 4 is the multiple interactions. Over Graph 5, Service 3 has 4 obligations with radius 2. Over Graph 5, Service 3 has 6 obligations with radius 3. Over Graph 4, Service 3 has 2 obligations with radius 2. Over Graph 4, Service

3 has 3 obligations with radius 3. As seen in Figure 7.16, for radius 2, the Service 3's execution time is increased by 50%, whereas, for radius 3, the execution time is increased by 67%. These results make us realize that a significant increase in the number of obligations affects the execution time significantly as well.

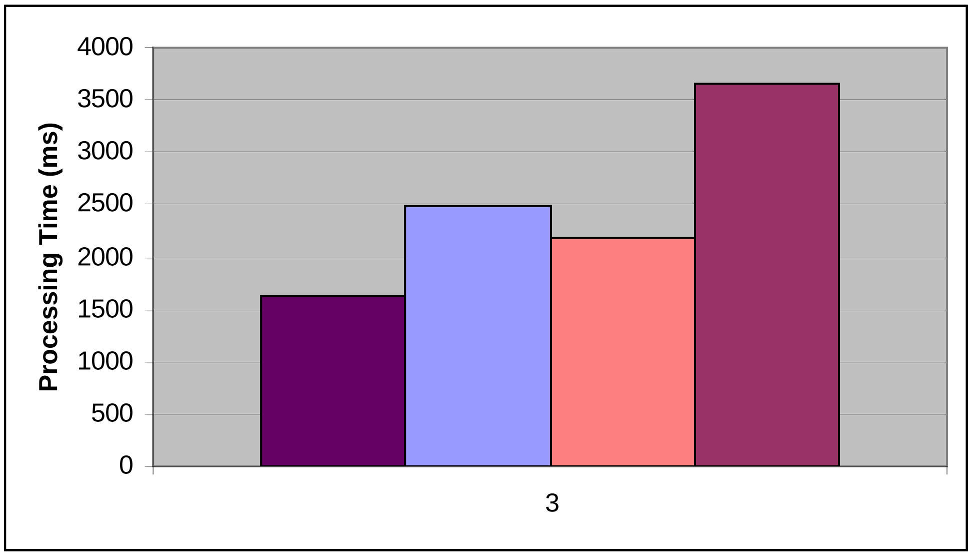


Figure 7.16 Service 3 execution times for L+U+D Policy over Graph 4 and Graph 5. For each service, first bar shows the execution time for radius 2 over Graph 4; second bar shows execution time for radius 2 over Graph 5; third bar shows execution time for radius 3 over Graph 4; fourth bar shows execution time for radius 3 over Graph 5.

7.3 Performance Conclusion

Our tests showed us that there are two significant variables dictating a service's execution time: the number of peer-peer evaluations and the Delegation-upstream rules (i.e. the obligations). The number of peer-peer evaluations I all cases increased the execution time significantly. Our data does not allow us to determine the nature of this increase whether it is linear or non-linear with the number of peer-peer evaluations. We must conduct experiments over a larger cluster so that the number of service nodes can increase significantly.

We also understood that Delegation-upstream rules could be effective over the execution time. This effect does not show up for small number of obligations such as one or 2. However, when we increase the number of obligations to 6, we realized that the processing of the obligations almost took up half of the execution time.

Finally, we observed that utilizing different type of rules combined in a policy did not cause any additional overhead. For U type rules, evaluating an additional policy clearly affects the execution time. In order to get a realistic view, we should conduct more experiments such that different Underlying security mechanisms are used.

Overall, we concluded that we must conduct tests with more complicated collaborations that span tens if not hundreds of service nodes. This would give a glimpse of real-life collaboration scenarios that occurs in scientific application domain. However, our initial results are promising in that they are almost insignificant compared with the actual execution time expected from a collaboration. In scientific application domain, the collaborations are expected to run at least for hours. Therefore, the overhead we introduced into the planning stage is miniscule. Moreover, this overhead is quite desirable when considering that an access failure arises hours after a collaboration starts executing, forcing the end user to repeat the entire planning and execution stages.

Chapter 8:
Conclusions and Future Work

Dynamic collaborations provide the ideal computing environments, in which otherwise unsolvable tasks can be tackled by combining available services from different domains. The users can harness services on-demand, based on their needs, and avoid issues, such as software writing and maintenance, low performance, and storage, which can all be handled by individual services. The service-oriented architecture provides the essential infrastructure that allows services to collaborate with each other. The emerging Web-Services standards bring openness and ease of collaboration into the computing field.

The reflections of this change are felt at various levels, including our personal lives. We, as end users, have started to take it granted that, for example, an online apartment search engine must use Google Maps in order to show us not only the features of a rental property, but also its location in the city. The lack of such combined services drives the end user away with frustration although even a naïve end user can copy a property's address and past it into the Google Maps search bar in a few seconds. This usage trend indicates us that connectedness and collaboration among the services are becoming the norm.

However, the highly appreciated benefits of collaboration come with the cost of more complicated security and trust problems. Our work focuses on these issues from an access control standpoint. We have two prime research questions: How can a service owner determine it is safe to join a collaboration ?; How can an end user determine if a collaboration is feasible for execution from security perspective ?

What makes collaborations different from traditional one-one relationships is that a collaboration requires multiple interactions among all of its participants. It is impossible to isolate a participant from a certain group of other participants because, by definition, a collaboration is built so that the participants can interact and share with each other. Even when two participants are not explicitly interacting, which is to say there is no direct data exchange between them, they in fact interact indirectly through other participants that have direct connections with them. This taught us that a service owner who joins the collaboration must consider the entire collaboration context, not just one or two interactions that his service is explicitly involved in.

This realization motivated us to understand the nature of interactions within a collaboration. We developed a model for interactions and classified them with respect to the different security threats that they introduce to a service. This naturally lent itself to developing an access control model and policy model that allows for specifying unique access requirements for different interaction types. We believe that our model is the first one that is tailored for dynamic collaborations.

A natural extension of our findings is to understand how this knowledge can be used in a framework so that collaborations can be built dynamically and execute without security failures, which becomes our second prime research question. As a solution we developed an access control framework that can be integrated into a collaboration engine. Service owners are informed about the collaboration context and they perform their own peer-peer trust evaluations. By moving the peer-peer trust evaluations into the planning stage, our framework mitigates the security failures. In addition, our framework increases services' willingness to join a collaboration because they have full control over how their interactions (hence their data) are propagated across the collaboration.

We believe that our work is practical not only in scientific application domain, but also in the e-commerce domain. An emerging problem in this domain is the intellectual property and confidentiality issues, and the conflict of interest scenarios that often occur between rival companies. In such scenarios access control is geared towards ensuring that business practices are translated to the virtual domain, such as honoring partnership agreements, licenses, or refraining from interactions with a rival company. Our access control model can express and enforce these concerns. In fact, as our future work, we plan to show how conflict of interest scenarios can be detected in a collaboration. We will demonstrate that such detection cannot be done without analyzing the collaboration context.

8.1 Future Work

We are interested in understanding the overhead of our security evaluations over the execution time. Our initial findings show that the number of peer-peer evaluations increases the execution time proportionally. Parallel execution techniques must be researched in this area. Moreover, large collaborations can have slower execution times due to large number of obligations that must be processed. Our initial experiments showed that when the number of obligations that must be processed by a service increases significantly, the processing time of the obligations becomes a significant overhead. Finally, we should experiment with large collaborations that span tens of services, scattered across the network. This would give us a realistic idea about the overhead.

Another future work area is redeeming an infeasible collaboration. Currently, we only determine whether a collaboration is ready for execution. We do not make any suggestions over what changes must be made in order to redeem the collaboration. This includes giving feedback to the planning engine so that different services can be assigned to the tasks. An important question is if a service owner refuses participation due to an un-trusted peer, should we replace the service refusing to partake or the un-trusted peer. The answer is dependent upon the collaboration: if one these services are absolutely necessary for the collaboration, then the other one can be replaced; alternatively, if one of the services has may interactions with other peers and they are all authorized, replacing the peer with less interactions would be more meaningful. We should understand what other variants are important in making this decision. Once this issue is studied, we can develop methods for optimizing the time spent in redeeming an infeasible collaboration graph. The redemption process may require multiple steps in which a different service may be replaced. In order to minimize this process, adequate optimization methods must be studied.

Another future work area is to determine the feedback given to the service owners. Currently, we give no feedback apart from the collaboration is ready for execution or not. We must research on this subject to understand if giving more detailed feedback is beneficial.

The final research area is to determine how we can leverage our framework to detect and eliminate conflict of interest scenarios in collaborations. Currently, the conflict of interest is widely studied in homogenous collaborations where all participants belong to the same security domain. The conflicts can be detected and prevented by a central policy. In heterogeneous, dynamic collaborations this cannot be achieved; each service has a different confidential policy. Our framework already allows each service owner to evaluate the collaboration context to determine access. We are curious to see whether a service's collaboration policy can express access requirements such that these requirements detect and prevent the conflicts within a collaboration. As a result, the collaboration policies not only protects the service from unauthorized accesses during the collaboration, but also detects possible conflict scenarios.

REFERENCES

[ABBD05] M. Altunay, D.E. Brown, G.T. Byrd, and R.A. Dean "Trust-Based Secure Workflow Path Construction." *Intl. Conf. on Service-Oriented Computing*, pp. 382 395, December 2005.

[ABBD2-05] M. Altunay, D. Brown, G. Byrd, R. Dean "Evaluation of Mutual Trust during Matchmaking." *6th IEEE Intl. Conf. On Peer-to-Peer Computing P2P 2005,* Konstanz, Germany.

[ACDV+] C. A. Ardagna, M. Cremonini, E. Damiani, S.D. di Vimercati, and P. Samarati "Supporting Location-based Conditions in Access Control Policies" *2006 ACM Symposium on information, Computer and Communications Security* (*ASIACCS '06),* (2006).

[AH96] V. Atluri and W-K. Huang "An Authorization Model for Workflows." *Fifth European Symposium on Research in Computer Security*, pp. 44-64, September 1996.

[AJAX] G. Murray "Asynchronous JavaScript Technology and XML (AJAX) with the Java Platform." http://java.sun.com/developer/technicalArticles/J2EE/AJAX/

[BE01] R. Botha, J. Eloff "Separation of Duties for Access Control Enforcement in Workflow Environments." *IBM Systems Journal*, Vol. 40, No. 3 (2001).

[BFA97] E. Bertino, E. Ferrari, and V. Atluri "A Flexible Model Supporting the Specification and Enforcement of Role-Based Authorizations in Workflow Management Systems." *2nd ACM Workshop on Role-Based Access Control*, 1997.

[BFA99] E. Bertino, E. Ferrari, and V. Atluri "The Specification and Enforcement of Authorization Constraints in Workflow Management Systems." *ACM Transactions on Information and System Security*, 2(1):65-104, February 1999.

[BFIK99] M. Blaze, J. Feigenbaum, J. Ioannadis, A.D. Keromytis "The Role of Trust Management in Distributed Systems Security." *In Secure Internet Programming: the Security Issues for Mobile and Distributed Objects*. Springer-Verlag (1999) 185-210

[BP02] D. Baumer and J.C. Poindexter *Cyberlaw & E-Commerce*. McGraw-Hill, New York, USA, 2002.

[CW87] D.D. Clark and D.R. Wilson "A Comparison of Commercial and Military Computer Security Policies" *IEEE Symp. on Security and Privacy* (April 1987), pp.

184–194.

[DRM] R. Safavi-Naini and M. Yung *Digital Rights Management: Technologies, Issues, Challenges and Systems.* LNCS 3919, Springer-Verlag. 2006.

[FK97] I. Foster, C. Kesselman "Globus: A Metacomputing Infrastructure Toolkit." *Intl J. Supercomputer Applications* (1997) 11(2):115-128.

[FKT01] I. Foster, C. Kesselman, and S. Tuecke "The Anatomy of the Grid: Enabling Scalable Virtual Organizations." *International J. Supercomputer Applications*, 15(3), 2001.

[HA99] W-K Huang and V. Atluri "SecureFlow: A Secure Web-enabled Workflow Management System." *4th ACM Workshop on Role-based Access Control,* October 1999.

[HIPAA03] Standards for Privacy of Individually Identifiable Health Information (HPR). 45 CFR 164.C. Federal Register (2003) 68(34):8334 – 8381.

[HK03] P.C.K. Hung and K. Karlapalem "A secure Workflow Model." *Australasian Information Security Workshop Conference*, 2003, pp. 33-41.

[HQ03] P.C.K. Hung and G-S.Qiu "Specifying Conflict of Interest Assertions in WS-Policy with Chinese Wall Security Policy." *ACM SIGecom Exchange* Vol. 4, No. 1 (May 2003), pp. 11-19.

[IETF99] R. Housley, W. Ford, W. Polk, D. Solo "Internet X.509 Public Key Infrastructure Certificate and CRL Profile", 1999, http://www.ietf.org/rfc/rfc2459.txt

[Kerberos] Kerberos: The Network Authentication Protocol, http://web.mit.edu/Kerberos/

[Kno00] K. Knorr "Dynamic access control through Petri net workflows." *16th Annual Conference on Computer Security Applications (ACSAC'00),* pp. 159-167, December 2000.

[KFP01] L. Kagal, T. Finin, and Y. Peng "A Delegation Based Model for Distributed Trust." *Intl Joint Conf. on Artificial Intelligence (IJCAI 2001), Workshop on Autonomy, Delegation and Control* (Aug 2001).

[KKHK03] S-H. Kim, J. Kim, S-J. Hong, and S. Kim "Workflow-based Authorization Service in Grid." *Fourth International Workshop on Grid Computing (GRID'03)*, 2003, pp. 94-100.

[KM03] H. Koshutanski and F. Massacci "An Access Control Framework for Business Processes for Web Services." *ACM Workshop on XML Security*, 2003, pp. 15-24.

[KPF01] M.H. Kang, J.S. Park, and J.N. Froscher "Access-Control Mechanisms for Inter-Organizational Workflow." *Sixth ACM Symposium on Access Control Models and Technologies*, 2001, pp. 66-74.

[KS01] K. Knorr and H. Stormer "Modeling and Analyzing Separation of Duties in Workflow Environments." *16th Intl. Conf. on Information Security (IFIP/Sec)*, (June 2001), pp. 199-212

[LN99] J. Linn and M. Nystrom "Attribute Certification: An Enabling Technology for Delegation and Role-Based Controls in Distributed Environments." *ACM Workshop on RBAC* (1999), pp. 121-130.

[MULT] Multiple Resource Profile of XACML v2.0 http://docs.oasis- open.org/xacml/2.0/access_control-xacml-2.0-mult-profile-spec-os.pdf

[OASIS] Organization for the Advancement of Structured Information Standards. http://www.oasis-open.org/home/index.php

[OASIS05] Organization for the Advancement of Structured Information Standards. *Web Services Business Process Execution Language Version 2.0*. Document identifier wsbpel-specification-draft-01, 01 Sep 2005, http://www.oasis-open.org/committees/download.php/-16944/wsbpel-specification-draft.241_proposal.doc

[OASIS05-2] Organization for the Advancement of Structured Information Standards. "SAML V2.0", http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security#samlv20

[O'Reil05] T. O'Reilly. "What Is Web 2.0 Design Patterns and Business Models for the Next Generation of Software." http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html, (2005)

[PWFK+02] L. Pearlman, V. Welch, I. Foster, C. Kesselman, S. Tuecke "A Community Authorization Service for Group Collaboration." *IEEE 3rd Intl. Workshop on Policies for Distributed Systems and Networks* (2002).

[San88] R. Sandhu "Transaction Control Expressions for Separation of Duties." *4th Aerospace Computer Security Conf.* (Dec 1988), pp. 282–286.

[San90] R. Sandhu "Separation of Duties in Computerized Information Systems." *IFIP WG11.3 Workshop on Database Security* (Sep 1990).

[San96] R. Sandhu "Role-Based Access Control Models." *IEEE Computer* (1996) 29(2):34-47

[Sun05] Sun's XACML Implementation. http://sunxacml.sourceforge.net/

[SBG05] M. Shehab, E. Bertino, and A. Ghafoor "Secure Collaboration in Mediator-Free Environments." *ACM Conf. on Comp. and Communications Security*, Nov. 2005, pp. 58-67.

[SOAP] Simple Object Access Protocol (SOAP). http://www.w3.org/TR/wsdl

[SS75] J.H. Saltzer and M.D. Schroeder "The Protection of Information in Computer Systems." *Proc. of the IEEE,* Vol. 63, No. 9 (1975), pp. 1278–1308.

[TCG04] K. Tan, J. Crampton, and C.A. Gunter "The Consistency of Task-Based Authorization Constraints in Workflow Systems." *17th IEEE Computer Security Foundations Workshop (CSFW'04)*, 2004, pp. 155-169.

[TS93] R.K. Thomas, R. Sandhu "Towards a Task-based Paradigm for Flexible and Adaptable Access Control in Distributed Applications." *ACM SIGSAC New Security Paradigms Workshop* (1992-93) 138-142.

[Wel03] V. Welch, et al., "Security for Grid Services." *12th Intl. Symp. on High Performance Distributed Computing*, 2003.

[W3C] Word Wide Web Consortium (W3C). http://www.w3.org/

[W3C04] World Wide Web Consortium (W3C). "Web Services Architecture." http://-www.w3.org/TR/2004/NOTE-ws-arch-20040211, 2004.

[W3C05] World Wide Web Consortium. *Web Services Choreography Description Language Version 1.0*, 9 Nov 2005, http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/

[WFMC] Workflow Management Coalition (WfMC). http://www.wfmc.org/

[WK05] J. Wainer and A. Kumar "A Fine-Grained User-to-User Delegation Method in RBAC." *ACM Symp. on Access Control Models and Technologies (SACMAT 2005)*

(June 2005).

[WSDL1.1] Web Services Description Language (WSDL) 1.1 http://www.w3.org/TR/wsdl

[XACML05] Organization for the Advancement of Structured Information Standards. *Extensible Access Control Markup Language*. Document identifier oasis-access_control-xacml-2.0-core-spec-os, 01 Feb 2005, http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf

[ZAC01] L. Zhang, G-J. Ahn, B-T Chu "A Rule-Based Framework for Role-Based Delegation." *ACM Symp. on Access Control Models and Technologies (SACMAT 2001)* (May 2001).

[ZAC02] L. Zhang, G-J. Ahn, B-T Chu "A Role-Based Delegation Framework for Health Care Information Systems." *ACM Symp. on Access Control Models and Technologies (SACMAT 2002)* (June 2002).