PERFORMANCE CONSIDERATIONS FOR A CORBA-BASED APPLICATION AUTHORIZATION SERVICE[1]

Konstantin Beznosov, Luis Espinal, Yi Deng

The Center for Advanced Distributed Systems Engineering
Florida International University, Miami, FL, 33199
{beznosov,lespin03,deng}@cs.fiu.edu

Contact: Dr. Yi Deng deng@cs.fiu.edu
Words: 7660
Submission type: Proceedings

April 3, 2000

Abstract

Resource Access Decision (RAD) Service allows separation of authorization from application functionality in distributed application systems by providing a logically centralized authorization control mechanism. RAD has attractive features such as decoupling of authorization logic from application logic, simplicity, generality, flexibility, support for complex application level access control, and ease of policy administration in heterogeneous, distributed systems. However, there is a concern of performance penalty for obtaining authorization decisions from a possibly remote server on each application request. We describe our work in measuring run-time performance of a CORBA-based Application Authorization Service (CAAS), which is compliant with the OMG specification of Resource Access Decision Facility, and draw conclusions about performance considerations in implementation of RAD compliant authorization services. We identify factors, which affect overall run-time performance of the approach and suggest possible solutions.

Keywords

Authorization, security, application-level security, distributed systems, heterogeneous systems, software engineering, performance evaluation, CORBA, distributed object technology.

1. This work was supported in part by the NSF under grant No. HDR-9707076. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official polices or endorsements either expressed or implied by NSF.

PERFORMANCE CONSIDERATIONS FOR A CORBA-BASED APPLICATION AUTHORIZATION SERVICE[1]

1. INTRODUCTION

Security is an essential function and concern to every enterprise. In general, application systems embed security mechanisms [1, 2] such as access control within their own application logic. This embedding causes enterprise security administrators to configure access control logic on application-by-application basis [3]. This application-based multiple-point security control makes enterprise security administration tremendously difficult, costly and error prone. It makes security policies and control mechanisms harder to change, and makes application software difficult to develop, change and dynamically reconfigure [3, 4]. Suitable security architectures, which separate authorization logic from application logic by treating authorization as an independent generic service, and which supports complex application level access control, are necessary in order to enable enterprise system evolution. By using these architectures, changes in authorization logic should have minimal or no impact on application-specific logic and vice versa.

Beznosov et al. [3] suggested a conceptual architecture of an authorization service for CORBA-based application systems. This body of work served as foundation for Resource Access Decision Facility (RAD) specification from the OMG [5]. RAD approach promises to provide a common foundation for a security architecture, which separates authorization logic from applications, to support fine-grained, dynamic, complex policies, and to enable interoperability based on open standards. However, it is an open issue as how to design and implement a flexible, maintainable, extensible, and portable authorization server based on the logical design of RAD. Even more important question is what performance implications arise from using such an authorization server. Regardless of how attractive an approach for developing application systems is, if the resulting systems' performance decreases significantly, the approach would not be of much help to system developers. Hence, before studying the validity of all these promises by the approach, it is necessary to address the question of run-time system performance in the first place. One would expect middleware and communication overhead to affect run-time performance. However, we need to qualify and quantify overhead due to middleware and communication subsystem. This is because RAD architecture includes multiple components that can be located in the same process, in the same host or in different hosts in a network environment. In each of these configurations, middleware and communication subsystem will affect to different extend overall run-time performance. At the time this work is presented (4-2000), no work on architecture of authorization servers affecting performance has been reported that we can use to reason about RAD performance.

To address the above issues, we designed and implemented a CORBA-based Application Authorization Service (CAAS) according to the logical design of RAD to server as experimental test-bed. The focus of this research was to measure run-time performance of application systems that obtain authorization decisions from a possibly remote RAD server. We measure run-time performance of CAAS under various configurations, loads and server-side business logic delays using a basic performance model. By analyzing the performance of potential systems that use RAD approach for authorization decisions, we expect to obtain information that enable us to design and deploy appropriate RAD servers.

1. This work was supported in part by the NSF under grant No. HDR-9707076. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official polices or endorsements either expressed or implied by NSF.

The main contributions of our work are our performance measurements and conclusions we have drawn from them. Results from our experiments measure architecture of RAD-compliant authorization service affecting run-time performance. We identify factors affecting run-time performance of systems using CAAS and possible solutions for improving run-time performance of implementations of RAD. Moreover, we believe our performance results can be used to reason and measure run-time performance of authorization servers in general.

The organization of this paper is the following: Section 2 discusses background information and related work. Section 3 introduces RAD conceptual architecture, CAAS design requirements, design and implementation. Performance model, considerations, and measurement results are covered in Section 4. We discuss our findings in Section 5, and we draw conclusions along with status report and summary in Section 6.

2. BACKGROUND AND RELATED WORK

The idea of authorization decisions being separated from application logic is not new. An abstract model of a reference monitor [6] is a classical example of authorization decisions being made and enforced outside of applications. The industry achieved considerable results in regards to the control of access to resources of operating systems, databases and middleware [7-18]. However, fine-grain control of application resources is done traditionally in ad-hoc manner [19], and there is no automated means to ensure enterprise-wide consistency of such controls.

The research community has being working towards systematic ways to address the problem. There are three main research directions. They are policy agents, interface proxies and interceptors, as well as enterprise-wide authorization services.

The direction of policy agents [4] is motivated mainly by the goal of accommodating the existing body of products and technologies. The key property of the direction is centralized access control (AC) management via translation of AC rules into supported by local mechanisms languages, and distribution of the rules across systems. Approaches under this direction have a number of advantages: there is inherent fault tolerance; enterprise security is naturally compartmentalized without penalizing run-time performance; the architecture facilitates achieving nominal performance overhead; there is high degree of run time autonomy. The main challenge facing the approaches is the consistency of enforced global policy, and automation of mapping a global policy into various instances of AC mechanism languages and representations. The approaches also suffer from a number of inherent limitations. First, the granularity and expressiveness of AC policies in a policy domain can be only as good as the policies supported by the most coarse-grain and least expressive AC mechanism in that domain. Second, distribution of policy updates can be very slow. The direction of policy agents becomes irreplaceable, if other approaches fail in those circumstances when application systems are already deployed. The question if it is the best way to address the problem of application-level AC for newly developed systems remains open.

Approaches under another direction employ either interface proxies, as in views as objects [20], role classes [21], security meta objects [22, 23], or interception of intersystem communications, as in SafeBots [24, 25] and Legion system [26-28]. Access to an application system is controlled externally. The main advantage of the direction is that it does not require almost any changes to the application system. The reference monitor is implemented externally to the application system, and security developers can control its size. Another advantage is the ability to make all the decisions locally to an application system, which

facilitates performance scalability. However, there are a number of significant limitations. First, AC granularity cannot be finer than method and arguments granularity. Second, decisions have to be made either before or after an application system is in possession of control. Third, variables, whose values become available at some point after the method is invoked, cannot be used in authorization decisions. Fourth, since there are as many instances of controls as application systems, insuring consistency of enforced policies as well as coherency of data used for authorization decisions becomes a challenge.

Another direction in AC for distributed application systems is based on authorization services [3, 29-32]. Decisions provided by an instance of the service are enforced by an application system. Both an application system and an authorization server constitute a reference monitor [6], which requires an application system to be trusted to enforce AC decisions. The main advantages of the direction are inherent consistency and coherence of enforced authorization policies, ease of policy changes and updates, ability to change policies and their policy types without affecting application systems, relatively low cost of AC administration, ability to obtain authorization decisions just when they are needed, and potentially any level of granularity of the resources to be protected. However, in order to construct a successful architecture for a distributed authorization service, one must address several key problems. They are performance, fault tolerance, scalability, security of communicating authorization information, guarantee of authorization decisions being enforced, and common representation of information used for making the decisions.

We believe that contemporary information enterprises have to have application AC implemented in all three forms. Successful architectural solutions will employ a combination of proxies, interceptors, policy agents, and authorization services because solutions from all three groups complement each other. For systems with the existing AC mechanisms tightly integrated into applications, policy agents are the only choice. In those existing systems, where AC mechanisms are missing, weak, or have too coarse granularity, interceptors and proxies, combined with the ideas from policy agents and authorization services could cure the problem. New applications with requirements for fine-grain, complex or very dynamic AC policies or to be deployed in organizations of different types (e.g. military, government, finance, health care, telecommunications) and sizes, will be best constructed with the use of application authorization service.

Beznosov et al. [3] outlines a conceptual architecture of an authorization service for CORBA-based application systems, which served as foundation for Resource Access Decision Facility (RAD) specification from the OMG [5]. We continue research on RAD approach by investigating how RAD architecture affects run-time performance. CAAS design, which is based on RAD architecture is the topic of the next section.

3. CAAS DESIGN

By using current technologies, namely CORBA and Java, we have designed and developed a CORBA-Based Application Authorization Service (CAAS) compliant with OMG's RAD specification. CAAS design was flexible, configurable, extensible and portable, and served as a test-bed for our performance measurements. RAD conceptual architecture is described next.

3.1. RAD CONCEPTUAL ARCHITECTURE

We use the terms RAD service, RAD logical design or simply RAD to denote the Resource Access Decision Authorization Facility as outlined in the OMG's RAD specification [5]. Detailed RAD description can be found in [3, 5].

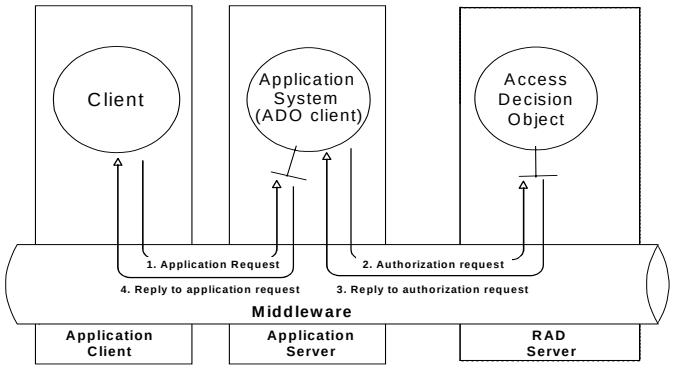


Figure 1: Interaction between Client, Application System and RAD Server

RAD architecture is designed to accommodate needs of organizations that differ in size, structure, and market sectors and to enable integration with existing infrastructures. RAD is not intended to replace middleware or other security environments, such as CORBA Security. The design rather assumes the existence of a capable middleware security, and compliments it with the capability of more sophisticated authorization, which is independent from the application logic.

RAD approach is a representative example of authorization services direction described in Section 2. As most authorization services, RAD provides authorization decisions to an application system (AS) (Figure 1). An application client or simply client sends an application request to AS (step 1). If the application request needs to be authorized, the AS sends one or more authorization requests to RAD (step 2). Each authorization request consists of client security credentials, name of the resource to be accessed, and name of the operation to be performed on the resource. Client security credentials are supposed to be obtained by AS from the middleware security infrastructure. AS is expected to compute the resource operation names as part of its application logic. For each authorization request, AS receives back a binary (yes/no) authorization decision (step 3). AS enforces the authorization decision(s) and returns back to the client (step 4). Interactions between a client, an AS and an instance of RAD are very common to most solutions based on authorization services. The main difference between RAD and other authorization services is in its "internal" architecture.

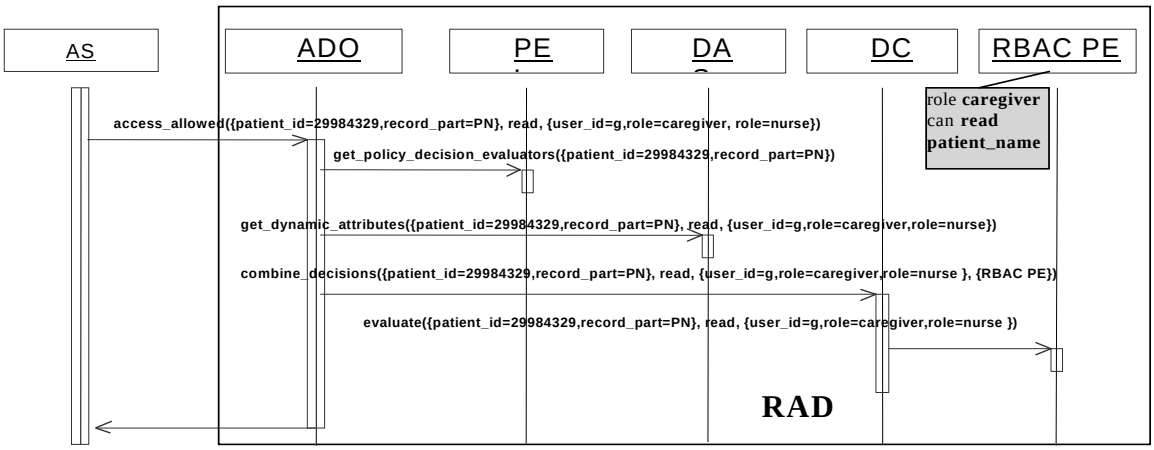


Figure 2: Sequence of events in an authorization decision request

RAD design aims to enable implementation of its components by various vendors due to the diversity

in the requirements to AC policies, performance, scalability and other system properties from different government and commercial markets. A RAD server is composed of the following components: 1) an *AccessDecision* Object (ADO), 2) zero or more *PolicyEvaluator* (PE) objects, 3) a PolicyEvaluatorLocator (PEL), 4) a *DynamicAttributeService* (DAS), and 5) a *DecisionCombinator* (DC) object. RAD conceptual architecture is such that all of these components could be replaced dynamically by different implementations as long as they comply with the interface specifications. The components are only logically disjoint. In practice they can be co-located in the same process or host. This feature is provided to further the support for dynamic composition and re-configuration, as well as for high availability and fault tolerance of the services based on RAD architecture.

3.2. COMPUTATION OF AUTHORIZATION DECISIONS

An authorization decision is computed through a sequence of operations carried out by the RAD components. These sequence of operations or control flow is described in the next section. The ADO receives requests for authorization decisions from ASs (RAD clients in this context). Through DC, ADO delegates evaluation of policies governing access to resources to corresponding PEs, which encapsulate AC policies. Unlike most authorization services [29, 30, 33], RAD design does not impose any particular authorization language. Each policy evaluator can be administered using a different interface and AC rules written in a different language. Such a design is done to enable use of the existing policy engines (such as RACF [10]), that were not originally developed to be PEs. It also enables dynamic swapping of PEs. ADO delegates a DC the task of combining multiple results of evaluations made by PEs into a final decision. This is because there can be several PEs responsible for the same resource name. Each DC implements a particular combination policy. Its response is returned to the ADO and becomes the authorization decision, which the AS is expected to enforce. To obtain references to PEs and DCs, the ADO consults a PEL, which implements the logic of deciding what PEs and DCs are to be used to authorize a given access operation on a given resource name.

In order to support AC policies based on the factors whose value can change from request to request, ADO obtains dynamic privilege attributes of the application client from DAS before it passes the request to the corresponding DC and PEs. Dynamic attributes are privilege attributes, which can be determined only at the time when a request for an authorization decision takes place. Thus they are specific to the authorization request in question. Examples of such attributes are relationships between physicians and patients in a hospital [34]. DAS is an important architectural element, which distinguishes RAD from other authorization services and enables support of complex and dynamic AC policies with traditional access matrix [35].

To illustrate the work of RAD components, we provide an example of processing an authorization request in Figure 2. It shows the sequence of invocations among RAD components in a hypothetical case. For the sake of illustration, let us assume that there is an authorization policy, which contains a statement that a user can read patient name (PN) if the user is performing role caregiver. In this example, AS is requesting to authorize read access to patient_name of the medical record on patient with ID 29984329 for a user with user_id d, who activated role nurse, which is senior to role caregiver. ADO obtains a list of references to PEs and to a DC, which should be used for making authorization decision on resource with name {patient_id=29984329, record_part=PN}. The PEL returns a reference to the DC and a reference to one PE - RBAC PE. The DAS does not change the list of privilege attributes, which specifies that the user ID is g and the roles the user activated are caregiver and nurse. RBAC PE implements authorization based on roles [2]. According to the authorization rules, users acting as caregivers have access to names of all

patients. Thus the PE returns "yes" and DC returns back to ADO the same answer. The ADO authorizes the AS to access patient name for patient with ID 29984329.

When an AS requests an authorization decision for access to a particular resource, one or more PE objects that encapsulate policies controlling access to that resource may be invoked. For that particular authorization decision request, only one DC would combine PE results into an authorization decision. Also, in an authorization decision, DC consults each PE once (unless PEL returns duplicated object references to ADO). How DC and PE objects are associated with resource names depends on the access control mechanism or mechanisms implemented and maintained by security administrators.

3.3. DESIGN AND IMPLEMENTATION OF CAAS

For the sake of brevity we demonstrate the key design decisions of CAAS using the design of PE. Although ADO, PEL, DAS and DC are important components of CAAS, their designs share similarities with PE design, and it has the most complex design. A more detailed description of the design and implementation of all CAAS components can be found in [36].

PE has run-time and administrative interfaces, which we wanted to implement using the same class. However, IDL interfaces are implemented in Java using *implementation* classes, which makes it impossible to implement multiple IDL interfaces in Java via inheritance [37]. We resolved this issue by using a delegation pattern known as *Tie Approach* [38]. Tie approach means the use of delegation over inheritance when implementing IDL interfaces.[1] IDL interfaces are implemented with a tie class (*PolicyEvaluatorContext* in Figure 3), which provides functionality needed to operate with the ORB middleware. The tie class delegates implementation of operations to a *delegate* class, which in the case of PE is the *PolicyEvaluatorContext. PolicyEvaluatorContext* implements operations defined in *PolicyEvaluatorExt* and *PolicyEvaluatorAdminExt* IDL interfaces (see Figure 3). In summary, *Tie Approach* separates application-dependent logic from middleware logic by encapsulating middleware mechanisms in the tie class and application logic in the *delegate* class.

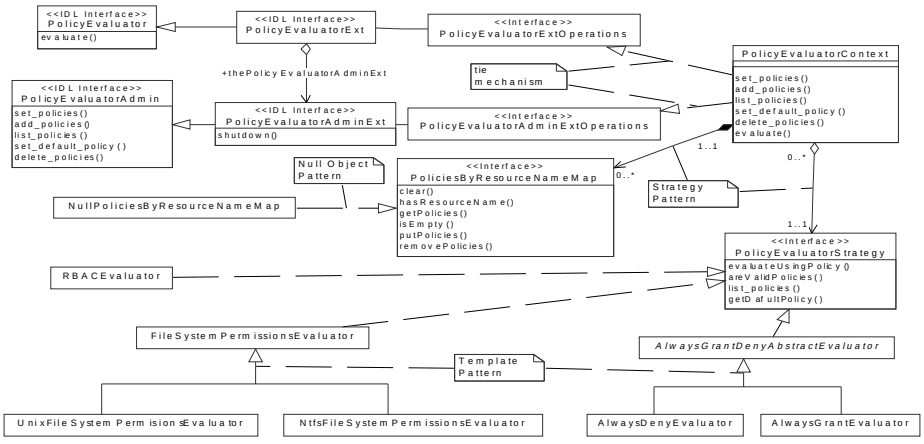


Figure 3: PolicyEvaluator Design

CAAS may use PE components implementing different policy evaluation mechanisms based on file system permissions, role-based access control (RBAC) [2], or even default evaluation policies which

1. . In principle, this is similar to the *Bridge* pattern described in [39].

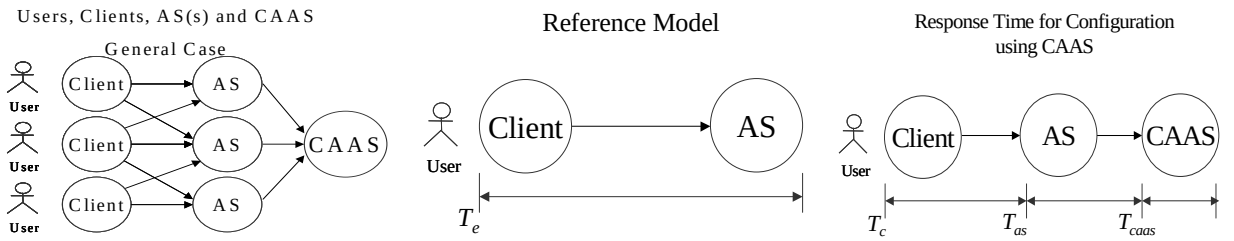
always grant or deny access. However, most of these PE instances may use the same mechanisms to associate resource names to access control policies. In this situation, using one implementation class per evaluation policy could introduce many related PE classes that differ only in their evaluation policy.

To address this issue, we used a solution based on *Strategy* pattern [40] in which *PolicyEvaluatorContext* implements functionality common to most other implementations of PE such as addition and removal of access control policies. Different policy evaluation mechanisms are delegated to an object implementing the *PolicyEvaluatorStrategy* Java interface as shown in Figure 3. Similarly, management of associations of resource names to access policies may vary between instances of PE. Consequently, *PolicyEvaluatorContext* delegates the implementation of such functionality to objects implementing the *PoliciesByResourceNameMap* Java interface. By using this interface, developers can implement associations using any form of storage suitable to their needs independently of the implementation of *PolicyEvaluatorStrategy.*

Implementations of *PolicyEvaluatorStrategy* interface are further refined using a design pattern known as *Template*. The rationale behind *Template* pattern (or *Template Method* pattern) is to define an outline or skeleton of an algorithm in a base (potentially abstract) class while leaving some steps to be defined in subclasses as shown in Figure 3 [39, 40]. By doing so, the design of PE allows extensions and modifications to policy evaluation mechanisms with relative ease as security requirements change during the system life cycle.

To know what access policies to evaluate given input parameters (from DC), a *PolicyEvaluatorContext* must maintain relationships or mappings between AC policies and resource names. These mappings are also implemented using *Strategy* pattern. That is, a class implementing such mappings implements the *PoliciesByResourceNameMap* interface shown Figure 3. Developers can use this interface to implement relationships using any form of storage suitable to their needs independently of the implementation of *PolicyEvaluatorStrategy.* The design includes a default *PoliciesByResourceNameMap* implementation based on the *Null Object* pattern [40], the *NullPoliciesByResourceNameMap* class. With *Null Object* pattern, developers can provide do-nothing versions of classes for which no particular implementation exists during execution. By using *Null Object* pattern, *PolicyEvaluatorContext* is relieved from testing for null values before accessing methods of *PolicyByResourceNameMap.*

We refer to [36] for detailed description of the design and implementation of RAD components.



(a) **(b)** **(c)**

Figure 4

4. PERFORMANCE MEASUREMENTS

For our performance measurements, we decided to measure run-time performance of CAAS in the presence of one client only, which sends requests to a single AS in a sequential manner as shown in Figure 4-c.

That is, the client waits until it receives reply from its previous request before it makes a new one. This allows to estimate a lower bound run-time performance of CAAS -- performance would be expected to degrade in the presence of multiple ASs sending concurrent authorization requests as shown in Figure 4-a. Since our immediate goal was to study only performance, we decided to leave performance scalability experiments for future research.

Another consideration was that CAAS flexibility allows developers and administrators to deploy its components on different hosts, as separate processes in the same system, or even as objects in the same address space. It was also possible to deploy CAAS and ASs on different hosts or as co-located processes. We expected such configurations to be important factors affecting run-time performance.

4.1. MEASUREMENTS MODEL

Our first step towards measuring performance was to build an experimental model and determine what aspect of AS-CAAS interaction would be as performance metric. We decided to measure response time T_c experienced by a client while making requests to AS as shown in Figure 4-c. We could have also measured response time at time T_{caas} when CAAS completes the processing of an authorization request and at time T_{as} when AS finish processing of an application request, which in turn contains time T_{caas}. We chose to measure response time perceived by clients since it includes response times at the other two points, and it was the main concern when authorization decisions are computed by CAAS. That is, our performance metric for CAAS is end-to-end response time that clients experience while interacting with AS.

We also needed to determine if absolute or relative response delay values should be used for the performance analysis. Since we did not use any standard benchmarks and our test bed is based on a prototype implementation, we measured response time relatively to a reference configuration or model that simulates traditional client-server systems where authorization logic is implemented within AS. That is, performance measurements are relative to overall response time T_e experienced by a user in the case of the reference model as shown in Figure 4-b. Moreover, by measuring relative response time, we minimized any overhead introduced by programming environments, which can affect absolute measurements. Using measurements T_c, and T_e, we calculated the response time increase percentage I of external access control for each configuration of CAAS with respect to embedded access control using the following formula:

$$I = \left(\frac{T_c}{T_e} - 1\right) \times 100 \quad (1)$$

The goals of conducting performance measurements on CAAS was to observe response time experienced by end users of ASs using CAAS to obtain authorization decisions relative to AS using embedded authorization. We expected overhead from ORB middleware and communication subsystem to be the most contributing factor in response time increase. Therefore, we needed to relative measure run time performance of CAAS under different configurations, which are discussed next.

4.2. CAAS CONFIGURATIONS

CAAS configurations determine the boundaries crossed by messages send during computation of an authorization request. Figure 5 shows configurations considered during performance measurements. Messages can be sent between AS and CAAS (external messages) and among CAAS components (internal messages). Furthermore, messages can be sent between objects, processes, or systems as shown in Figure 5.

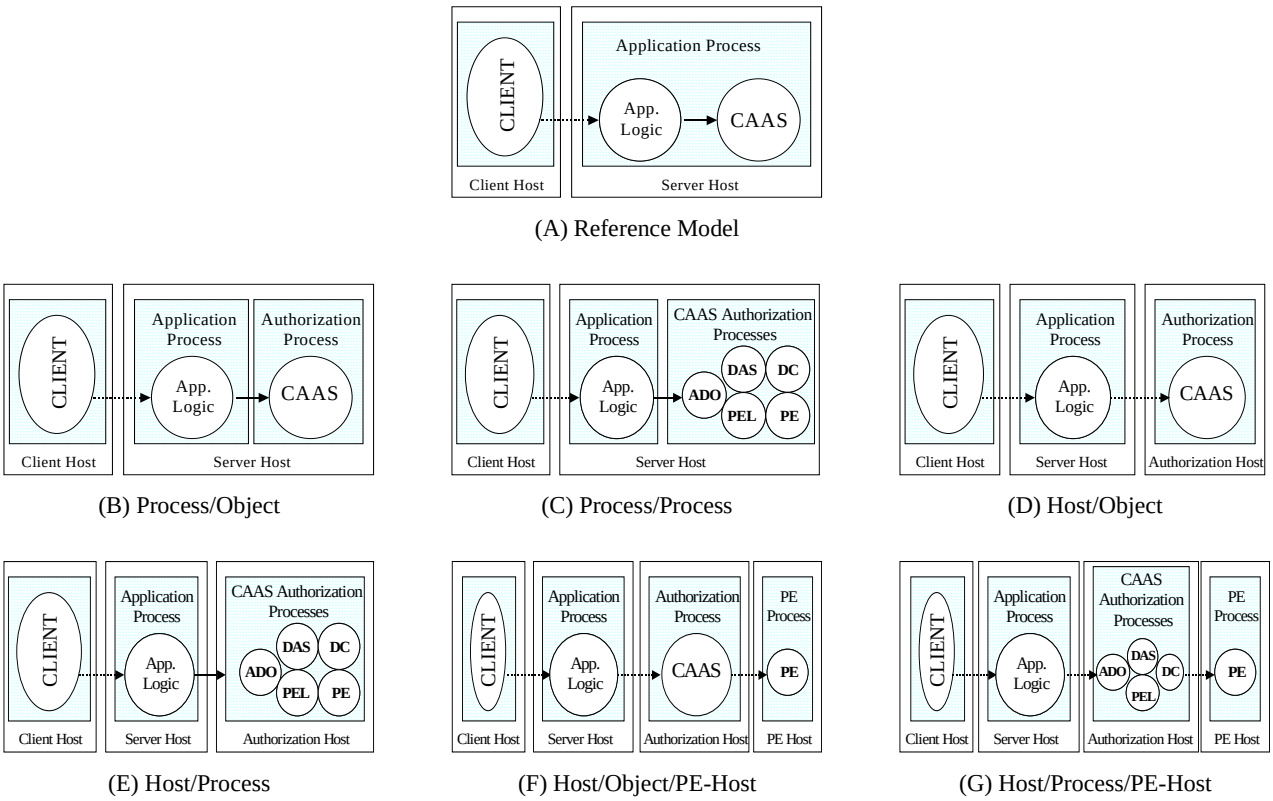


Figure 5: Reference model and some possible CAAS configurations

Messages cross *object* boundaries when components are co-located in the same address space and use direct method calls through JVM to communicate. Messages cross *process* boundaries when communicating components are co-located in the same host but run in their own processes. In this case, communication takes place through the ORB middleware, which is why we also call these boundaries as *middleware* boundaries. This form of communication, however, can take place using other mechanisms such as IPC [41, 42]. Finally, messages cross *host* boundaries when components reside on separate hosts; this involves middleware and communication subsystem overhead. CAAS can be deployed in any combination of configurations.

Even though code responsible for application and authorization logic can be highly coupled, we believe that code can be re-arranged into equivalent code in such a way that it will allow for every computer operation to identify whether it contributes to application or authorization performance overhead. Using this assumption, we simulate our Reference Model, which uses embedded authorization logic, by co-locating all CAAS components within the application process. This arrangement is illustrated in Figure 5-A.

With *Process/Object* configuration, AS and CAAS are co-located as independent processes in the same server host, and CAAS components are co-located within the same process as illustrated in Figure 5-B. Messages between AS and CAAS are transmitted via ORB middleware (process boundaries) whereas CAAS components communicate using native method calls using the JVM (object boundaries). Figure 5-C shows *Process/Process* configuration where CAAS components are deployed in their own process (process boundaries). In *Host/Object* configuration shown in Figure 5-D, CAAS components are co-located in the same process; however, AS and CAAS are on different hosts. That is, messages between AS and

CAAS are delivered through the ORB middleware and communication subsystem (host boundaries) while messages among CAAS components cross object boundaries.

In *Host/Process* shown in Figure 5-E , AS and CAAS are on different hosts, and CAAS components are on their own processes in the same host. Figure 5-F illustrates *Host/Object/PE-Host* configuration. This configuration is similar to *Host/Object* except that *PE* component runs in a different host. Communication among CAAS components incur object and host boundaries. Finally, in *Host/Process/PE-Host* configuration illustrated in Figure 5-G, *PE* is located in a host other than the authorization host while the other CAAS components run in different processes co-located in the authorization host. It is important to notice that when two components exchange messages through process boundaries, message passing involves middleware overhead and possibly context switch overhead at the host where the two reside. Host boundaries, on the other hand, do not involve such context switch overhead since the communicating components do not compete with each other for execution time.

This configurability allows developers and administrators to deploy CAAS in a way to obtain maximum performance (by avoiding ORB middleware and network overhead), or flexibility (by having any component in any system in the network). For example, administrators may deploy CAAS using *Host/Object* configuration to avoid middleware and network overhead. However, in an organization where one or more PE components are remotely located (perhaps in a different subnet), CAAS can be deployed using *Host/Object/PE-Host* or *Host/Process/PE-Host* configurations. *Host/Process* or *Process/Process* configurations can be used to deploy CAAS components developed by third parties, which are not enabled to run in the same address space with other components. In a real scenario, we expect to see most components be co-located in the same process or host while one or more components, possibly PE, be deployed in remote locations.

4.3. TEST ENVIRONMENT

Our test environment is composed of 4 Gateway E-4200 400MHz Pentium III PC's running Windows NT Workstation 4.0 service pack 4. Each workstation has 128MB of physical memory, 139MB of swap space and its performance properties set to maximum boost for foreground applications. Also, each workstation is equipped with a Intel (R) PRO/100+ Management network adapter. These workstations interoperate on an 100Mb Ethernet with one hub, and connect to the rest of the domain through a 100Mb switch. Furthermore, during testing we used JDK 1.1.7 and Visibroker 3.3, and all java classes and jar Files were located in the local hard-drives.[1] Also, we use run Naming service on a 4 Gateway E-4200 400MHz Pentium III PC with same hardware configuration and RedHat Linux 6.1. We carry out performance measurements only when network utilization is less than 1% so that we eliminate or minimize any factors that are not of CAAS itself, which can affect our performance measurements.

4.4. EXPERIMENT PROCEDURE

Our experiment procedure consisted of one client, one AS, one Access Decision Object (ADO), one Policy Evaluator Locator (PEL), one Dynamic Attribute Service (DAS), one Decision Combinator (DC), and one Policy Evaluator (PE). The goal of the performance measurements was to estimate a worst case performance penalty experienced by clients when CAAS is used authorization requests. These perfor-

1. During our experiments, we noticed that our test clients experienced a response time increase of 40ms more per authorization request for a configuration with no external access control when our java classes were located in a remote drive than when these same classes were on a local disk.

mance penalty measurements are relative to performance experience by clients using the Reference Model shown in Figure 5-A. We estimated the response time T_c experienced by the client when external access control is implemented using the CAAS. Then, we estimated the response time T_e using embedded access control. Using these two measurements, we calculated the response time increase percentage *I* of external access control for each configuration of CAAS with respect to embedded access control using Equation 1 on page 9. DC objects used in the experiments implement logical-AND combination policy while PE objects implement simple policies that always grant access.

This procedure was repeated using configurations described in Section 4.2. Other parameters for our performance measurements were application processing or business logic time *B* and number of authorization requests *N* generated for each client request. Application processing time represents delays experienced by AS while servicing client requests and enforcing authorization decisions returned by ADO; it does not include processing time incurred by CAAS. Although we used one client during the experiment, in an actual system, a client request can trigger any number of authorization requests by AS. This was simulated using a variable number of authorization requests in our performance measurements.

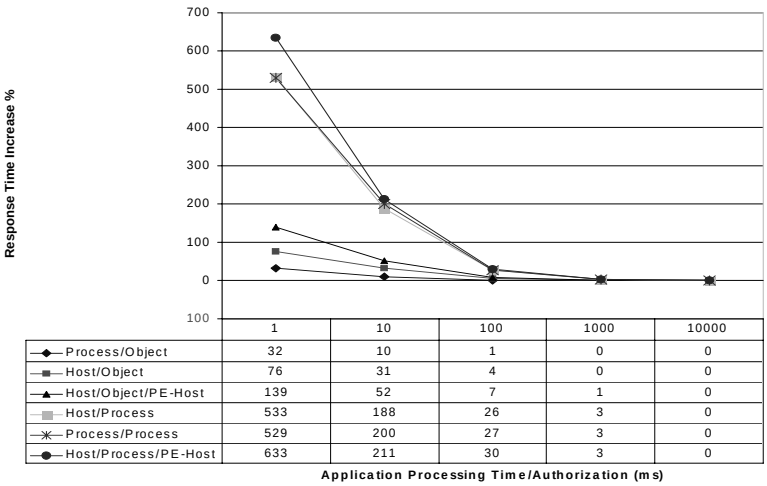
We concentrated on simple combination and evaluation policies in order to estimate a worst case performance penalty relative to the Reference Model. This is because more complex AC policies would most likely increase computation overhead without increasing significantly middleware and communication overhead due RAD components. In the Reference Model shown in Figure 5-A, this increase in computation overhead would mostly occur within embedded authorization logic. This would also be the case in *Host/Process/PE-Host* configuration shown in Figure 5-G. That is, complex AC policies would require an increase in computation overhead at the PE while communication between DC and PE remain unaltered. Similarly, complex combination policies would increase computation overhead within DC only. If computation overhead increases faster than communication and middleware overhead, response times of CAAS (T_c), and Reference Model (T_e,) would converge reducing relative response time increase *I* in Equation 1.

For our performance experiments we did not considered caching or encryption. Caching was not considered since it reduces communication overhead, may increase computation overhead, and therefore reduces relative response time increase *I*. Encryption, on the other hand, increases communication overhead, thus increasing *I*. Communication overhead, however, is something we cannot control. Moreover, encryption is application dependent. Different applications require different levels of encryptions. As a result, we decided not to consider encryption and estimate a worst case response time increase strictly in terms of middleware and communication due to RAD.

4.5. MEASUREMENT RESULTS

Measurements were carried out using CAAS configurations in Figure 5. The results from these measurements are illustrated in Figure 6. We measured response time increase as a function of application processing (business logic) time per authorization request. These measurements suggest that combined middleware and context switch overhead between two components co-located as separate processes in the same host can be as high as network overhead between the same components in different hosts. In Figure 6, when an authorization requests takes 10ms of application processing time (or when there are 100 authorization requests per second), response time increase for configuration *Host/Process* (Figure 5-E) differ from response time increase of *Process/Process* (Figure 5-C) by only 2%. When authorization requests take 1 second, differences between response times for these two configurations are negligible.

A more drastic difference is seen when comparing relative response increase of *Host/Process* (Figure 5-



	1	10	100	1000	10000
Process/Object	32	10	1	0	0
Host/Object	76	31	4	0	0
Host/Object/PE-Host	139	52	7	1	0
Host/Process	533	188	26	3	0
Process/Process	529	200	27	3	0
Host/Process/PE-Host	633	211	30	3	0

Figure 6: Response Time Increase for Various CAAS Configurations. Error size: 0.5±

E) or *Process/Process* (Figure 5-C) configuration with relative response time of *Host/Object/PE-Host* illustrated in Figure 5-F. When authorization requests take 10ms of application processing time (or when there are 100 authorization requests per second), relative response time increase of *Host/Object/PE-Host* is measured at 52% whereas relative response time increase of *Host/Process* is 188% (200% for *Process/Process*). Even though *Host/Object/PE-Host* has a PE in a different host, it performs better than *Host/Process* and *Process/Process* since the later configurations incur in much greater middleware and context-switch overhead.

Our results suggest that response time of a configuration using CAAS increases between 3% and 30% when authorization requests take 100ms or more of application processing time or when there are 10 or less authorization requests per second. In the presence of an external PE, response time seems to increase from 16% (*Host/Object/PE-Host* with respect to *Host/Object*) to over 80% (*Host/Process/PE-Host* vs. *Process/Process*) as illustrated in Figure 6.

5. PERFORMANCE CONSIDERATIONS

Our measurement results are consistent with our expectations that remote calls would be the most contributing factor in run-time performance of CAAS. More important, they suggest that for small business logic delays (1ms-1sec) or for large numbers of authorization requests, context-switch overhead affects response time as much as communication subsystem overhead in the presence of one client and one AS. Our measurements identify worst-case scenarios occurring with small business logic delays and/or large numbers of authorization requests per second of application processing. In such cases, response time of a configuration using CAAS can increase by 500% relative to systems using embedded authorization logic since greater context switch overhead is incurred within CAAS.

Since we measured performance in the presence of one client and one AS, these results provide a lower bound in response time increase for configurations using CAAS for authorization requests. When multiple clients and ASs are present, response time will most likely increase exponentially when the number of clients exceeds certain capacity level. Investigating response time increase as a function of the number and nature of external CAAS components warrants further research.

Messaging overhead can be minimized by co-locating components in the same process or by using smart ORBs, which provides “smart” binding by detecting if two communicating objects are in the same address space or host.[1] However, in situations where some or all of CAAS components are remotely deployed, middleware overhead contributes the most to increase in response time. Deployment and implementation of RAD-based authorization services such as CAAS should take into consideration the interactions among components. That is, optimizations just mentioned should be applied to components that have a high rate of interaction. For example, evaluations of policies that require more than one PE can be optimized by co-locating corresponding PEs with the appropriate DC in the same process or by using smart ORBs. Further research is needed for investigating possible optimizations at the middleware level.

Although concurrent requests were not part of our performance measurements, it is an aspect that warrants further research. Concurrency is not a trivial issue to handle in component-based systems. Safety preservation, the insurance that all objects in a system maintain valid states in the presence of concurrent access, requires the avoidance of read/write and write/write conflicts [44]. To address this issue, we decided in the current implementation of CAAS to use fully synchronized methods. Although this property does not guarantee the system to be free of liveliness failures such as deadlocks and resource starvation, it does guarantee consistency of values at the object level. Also, synchronized object instances are ready to be used in concurrent settings [44]. However, this introduces unnecessary synchronization, which can affect overall run time performance because calls to synchronized methods are more expensive, than to unsynchronized ones. Also, synchronized operations on CAAS components are of a coarse granularity, which can cause threads to block and unblock unnecessarily. The current version CAAS is a straightforward implementation of the RAD logical design. We plan to re-design CAAS in order to achieve efficient concurrent access and extend the scope of our performance measurements to include multiple clients and ASs.

The original design of RAD service defines an operation, the multiple_access_allowed() operation [5] for requesting multiple authorization requests. However, this operation is only defined in the ADO interface. Other components do not have operations defined for multiple authorization requests. Preliminary results from performance measurements suggest that with the current design benefits of multiple_access_allowed() are substantial only when most or all of RAD components are co-located in the same host. We believe multiple authorization requests methods can be implemented in PEL, DAS, DC and PL objects. Such an enhancement will improve overall performance, we believe. Other issues, which warrant further research include scalability, replication, caching, fault-tolerance, and the possibility of implementing RAD using middleware other than CORBA. Measuring the performance of such implementations as well as performance over high-speed networks or over the Internet-like environments warrants further study.

6. STATUS AND CONCLUSION

Resource Access Decision (RAD) Service enables separation of authorization logic from application logic. This decoupling allows development and maintenance of applications with fine-grain access control requirements independently from particular access control policies, factors used in authorization decisions and from particular access control models. As of February 2000, we have implemented a CORBA-Based

1. Visibroker 3.3 for Java, the orb utilized for our experiments, provides “smart” binding for objects in the same address space. However, for objects in different process even if they are in the same host, Visibroker orb uses IIOP [43].

Application Authorization Service (CAAS) compliant with the OMG specification of RAD Facility, and its run-time performance has been measured under different parameters (configuration, load and server-side delays). The design and implementation of components available in CAAS is covered with more detail in our technical report [36]. More information and documentation on CAAS current status, including the report, is available at http://cadse.cs.fiu.edu/.

The main contributions of our work are the performance measurements and the conclusions drawn from them. These measurements estimate a worst case response time increase for systems using CAAS for authorization decisions. Our performance results confirm quantitatively that middleware and context switch overhead can be more expensive that network overhead. Implementation of RAD servers can improve performance by co-locating as many components as possible in the same address space. In situations when this is not possible, utilization of smart ORBs that skip middleware layer for objects located in the same host have the potential to significantly improve response time. These optimizations should focus on components with high rate of interaction such as DC and PE components. Our measurements results are not only relevant to CORBA-based systems using RAD approach. Relative performance results for *Process/Object* configuration shown in Figure 5-B can be used to estimate response time increase of authorization servers co-located with application systems. Similarly, response time increase for authorization servers remote to application systems can be estimated from measurement results for *Host/Object* illustrated in Figure 5-B.

The authors would like to thank CADSE's Suresh R. Chegireddy, for his help with CAAS implementation and Bangalore Guruprakash, Manish Mahajan, Nathan N. Vuong for helpful comments and corrections on the first draft of the paper.

REFERENCES

[1] T. Y. C. Woo and S. S. Lam, "Authentication for Distributed Systems," *Computer*, vol. 25, pp. 39-52, 1992.

[2] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, "Role-Based Access Control Models," *IEEE Computer*, vol. 29, 1996.

[3] K. Beznosov, Y. Deng, B. Blakley, C. Burt, and J. Barkley, "A Resource Access Decision Service for CORBA-based Distributed Systems," presented at Annual Computer Security Applications Conference, Phoenix, Arizona, USA, 1999.

[4] J. Hale, P. Galiasso, M. Papa, and S. Shenoi, "Security Policy Coordination for Heterogeneous Information Systems," presented at Annual Computer Security Applications Conference, Phoenix, Arizona, USA, 1999.

[5] OMG, "Resource Access Decision Facility," Object Management Group OMG document number: corbamed/99-05-04, May 1999.

[6] J. Anderson, "Computer Security Technology Planning Study," Air Force Electronic Systems Division ESD-TR-73-51, Vols. I and II, 1972.

[7] J. H. Saltzer, "Protection and the Control of Information Sharing in Multics," in *Communications of the ACM*, vol. 17, 1974, pp. 388-402.

[8] B. J. Walker, R. A. Kemmerer, and G. J. Popek, "Specification and Verification of the UCLA Unix Security Kernel," *Communications of the ACM*, vol. 23, pp. 118, 1980.

[9] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn, "A Retrospective on the VAX VMM Security Kernel," *IEEE Transactions on Software Engineering*, vol. 17, pp. 1147-1165, 1991.

[10] M. Benantar, R. Guski, and K. M. Troidle, “Access control systems: From host-centric to network-centric computing,” *IBM Systems Journal*, vol. 35, pp. 94-112, 1996.

[11] M. J. McInerney, *Windows NT Security*: Prentice Hall, 1999.

[12] CA, “CA-ACF2 for OS/390,” : Computer Associates, 1998.

[13] CA, “CA-Top Secret for OS/390,” : Computer Associates International, 1998.

[14] IBM, *Resource Access Control Facility (RACF). General Information*: IBM Red Books, 1976.

[15] OMG, “Security Service Specification,” in *CORBAservices: Common Object Services Specification*: Object Management Group, 1996.

[16] OSF, “Authentication and Security Services,” : Open Software Foundation, 1996.

[17] W. Rubin and M. Brain, *Understanding DCOM*: P T R Prentice Hall, 1999.

[18] C. Lai, L. Gong, L. Koved, A. Nadalin, and R. Schemers, “User Authentication And Authorization In The Java Platform,” presented at Annual Computer Security Applications Conference, Phoenix, Arizona, USA, 1999.

[19] W. Wilson and K. Beznosov, “CORBAmed Security White Paper,” Object Management Group corbamed/97-11-03, November 1997.

[20] B. Hailpern and H. Ossher, “Extending Objects to Support Multiple Interfaces and Access Control,” *IEEE Transactions on Software Engineering*, vol. 16, pp. 1247-1257, 1990.

[21] J. Barkley, “Implementing Role-based Access Control Using Object Technology,” presented at The First ACM Workshop on Role-Based Access Control, Fairfax, Virginia, USA, 1995.

[22] T. Riechmann and F. J. Hauck, “Meta Objects for Access Control: Extending Capability-based Security,” presented at New Security Paradigms Workshop, Langdale, Cumbria, UK, 1997.

[23] T. Riechmann and F. J. Hauck, “Meta Objects for Access Control: A Formal Model for Role-based Principals,” presented at New Security Paradigms Workshop, Charlottesville, VA USA, 1998.

[24] R. Filman and T. Linden, “SafeBots: a Paradigm for Software Security Controls,” presented at New Security Paradigms Workshop, Lake Arrowhead, CA USA, 1996.

[25] R. Filman and T. Linden, “Communicating Security Agents,” presented at The Fifth Workshop on Enabling Technologies: Infrastructure for Collabarative Enterprises, Stanford, CA, USA, 1996.

[26] W. A. Wulf, C. Wang, and D. Kienzle, “A New Model of Security for Distributed Systems,” presented at New Security Paradigms Workshop, Lake Arrowhead, CA USA, 1996.

[27] A. S. Grimshaw, M. J. Lewis, A. J. Ferrari, and J. F. Karpovich, “Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems,” Department of Computer Science, University of Virginia CS-98-12, 1998.

[28] A. S. Grimshaw and W. A. Wulf, “The Legion Vision of a Worldwide Virtual Computer,” in *Communications of the ACM*, vol. 40, 1997, pp. 39-45.

[29] *V. Varadharajan and C. C. a. J. Pato,* “Authorization in Enterprise-wide Distributed System: A Practical Design and Application,” presented at 14th Annual Computer Security Applications Conference, 1998.

[30] T. Y. C. Woo and S. S. Lam, “Designing a Distributed Authorization Service,” University of Texas at Austin, Computer Sciences Department TR93-29, September 1993.

[31] T. Y. C. Woo and S. S. Lam, “Designing a Distributed Authorization Service,” presented at IEEE INFOCOM, San Francisco, 1998.

[32] M. E. Zurko, R. Simon, and T. Sanfilippo, “A User-Centered, Modular Authorization Service Built on an RBAC Foundation,” presented at Annual Computer Security Applications Conference, Phoenix, Arizona, 1998.

[33] R. Simon and M. E. Zurko, “Adage: An Architecture for Distributed Authorization,” OSF Research Institute, Cambridge 1997.

[34] J. Barkley, K. Beznosov, and J. Uppal, “Supporting Relationships in Access Control Using Role Based Access Control,” presented at ACM Role-based Access Control Workshop, Fairfax, Virginia, USA, 1999.

[35] B. W. Lampson, “Protection,” presented at 5th Princeton Conference on Information Sciences and Systems, Princeton, 1971.

[36] L. Espinal, K. Beznosov, and Y. Deng, “Design and Implementation of Resource Access Decision Server,” Center for Advanced Distributed Systems Engineering (CADSE) - Florida International University, Miami technical report 2000-01, January 2000.

[37] OMG, “IDL to Java Language Mapping,” Object Management Group, technical report OMG document number: formal/99-07-53, 1999.

[38] D. Pedrick, J. Weedon, J. Goldberg, and E. Bleifield, *Programming with VisiBroker: A Developer’s Guide to Visibroker for Java*. New York: Wiley Computer Publishing, 1998.

[39] E. Gamma, R. Helm, J. Vlissides, and R. E. Johnson, *Design Patterns: Elements of Reusable Object-Oriented Design*. Reading, MA: Addison-Wesley, 1994.

[40] M. Grand, *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*, vol. 1. New York: Wiley Computer Publishing, 1998.

[41] G. Nutt, *Operating Systems: A Modern Perspective*: Addison-Wesley, 1997.

[42] W. R. Stevens, *Advanced Programming in the UNIX Environment*: Addison-Wesley, 1993.

[43] *Visibroker 3.3. for Java Programmer's Guide*. Scotts Valley, CA: Inprise Corporation, 1998.

[44] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*. Reading, MA: Addison-Wesley, 1996.