Infrastructure Support for Predictable Policy Enforcement

Gary J. Vecellio
The MITRE Corporation
7515 Colshire Drive
McLean, VA 22102-7508
1+ 770 739-8598
vecellio@mitre.org

William M. Thomas
The MITRE Corporation
7515 Colshire Drive
McLean, VA 22102-7508
1+ 703 983-6159
bthomas@mitre.org

ABSTRACT

Component and service-based application infrastructures provide mechanisms for efficiently composing a system from a diverse collection of components and services. However, because of the lack of insight into the components and services within the application, integrating changes can be challenging. One class of change that we perceive as being both common and necessary is in the area of policy adherence (i.e., the constraints on a system's behavior that are imposed across the system). Unless the mechanisms that implement the policy are well isolated from the core application logic, any upgrade to the policy can have a ripple effect through the system. For systems that require robust certification, this ripple effect hampers the ability to rapidly deploy changes in policy. In this paper we highlight some patterns for separating policy adherence from application core logic, and discuss how these patterns can be mapped to commercially available infrastructures. By realizing these patterns as common infrastructure extensions, we allow applications to be developed in a manner consistent with the commercial infrastructure, provide the power of policy enforcement mechanisms to the system developers, and separate the policy enforcement logic from core application functionality.

1. INTRODUCTION

For any software system we would like to predict, or bound, the behavior of that system. That is, we would like to be able to predict that a predefined set of application specific policies (or properties) are satisfied by the system. These policies might be common to most types of applications (e.g., a role-based security policy) or they might be specific to an application type, or to all applications executing in a specific environment (e.g., a hardware-based encryption policy). In general, mechanisms for ensuring such policies can be built into the application itself. However, component and service composition creates challenges in this area since components and services are typically developed separately from the application being composed.

There are several aspects of policy adherence that present challenges in component and service-based applications. We would like to have some level of assurance before the application is deployed that a composition adheres to its policies. When policies do not depend on shared services, the composition's deployment configuration or the platform on which the composition executes, prediction of property adherence seems reasonable. However, policy prediction is more difficult if deployment configuration or the underlying platform can impact the policy. For example, predicting service response time becomes difficult when multiple, independently-developed applications assess the same service or are co-located in a single computer. In this case, some form of runtime property monitoring and enforcement is needed.

Policy enforcement mechanisms depend on the language(s) used to express the program logic and the policies. Many efforts to express policies have focused on the programming language level. Design by Contract™ (DBC) 1] uses pre and postcondition and invariant evaluation to enforce the policy that program modules must be used in a manner consistent with the developer's expectations. This policy is enforced by the programming language infrastructure on a module-by-module basis. Aspect Oriented Programming [2] can also be viewed as a policy enforcement mechanism. Software development tools extract common aspect from business logic modules and build modules that implement a policy enforcement aspect. Both of these mechanisms can be used to build components and services, but since they are programming language-based, they can not be use to monitor or enforce policies among independently developed components or services.

Commercial composition technologies have some composition-level mechanisms to enforce common aspects or policies. An example of commercial composition technology is the Java™ 2 Platform Enterprise Edition (J2EE™) [3]. On this platform, components are developed without regard to some common aspects, (e.g., security). At integration time, the container in which the components execute is configured to apply the application's security policy. When the application is deployed there is an opportunity to perform a site-specific configuration of the policy. For example, a site-specific configuration is used to map an application's generic security roles to site-specific roles. Post development configuration improves component reuse characteristics, because it allows the same component to be used in a variety of different security contexts.

Performing runtime property monitoring and enforcement can be accomplished in at least two ways. One approach is to develop an application-specific framework or infrastructure. This approach has been used successfully in the past, and it may be the most appropriate approach for some classes of applications. However, this approach can be expensive to implement and given the fixed budgets of many of our sponsors, we would like to find a more cost effective approach. The approach we have chosen to investigate is the extension of commercially available infrastructure technologies. That is, finding a common repeatable way to extend a set of commercial technologies to support policy enforcement. In the remaining sections of this paper we will show how a design pattern, the principles of DBC, Aspect Oriented Programming (AOP), and commercial infrastructure technology can be used to address property monitoring and enforcement.

2. BACKGROUND

We previously reported on our experiments with extending a J2EE Enterprise JavaBeans™ (EJB) container with services that offer the system developer flexible mechanisms to compose a system with well-constrained behavior [4, 5]. We view the container as an ideal place to offer such services, as it is designed to provide a separation of the application logic from the infrastructure. We augmented the JBoss open source EJB container to offer interceptor-based mediator services in addition to the standard JBoss container services. These mediators were configured to provide flexible constraint enforcement on method invocations, offering precondition and postcondition invocation-level checks on the state of the composition. Additionally, we extended the J2EE server with monitoring capabilities to support enforcement of invariants over the composition.

Related to our work is the research on "meta programming" mechanisms for distributed object computing [6]. Composition mechanisms, such as smart proxies and interceptors, provide additional control over CORBA communications. The smart proxies are client-side extensions that offer capabilities beyond those offered by the generated proxy, including application-specific needs. The interceptor, which can operate on both the client and server side, is an extension to the ORB that intercepts communication between client and server and integrates application-specific behavior into method invocation requests without involving the client or server core logic. These capabilities are described as being essential to allow for efficient upgrade of distributed object computing-based applications.

Filman describes an Object Infrastructure Framework (OIF) that is an approach to achieving non-functional "ilities" through the use of "injectors" attached to communications [7]. In this framework, meta-information is "attached" to CORBA method invocations, identifying a sequence of injectors that are to be processed for that communication. Using this approach, support for a number of "ilities" can be developed, including reliability, maintainability, quality of service and security; and in a consistent manner, such diverse "ilities" can be integrated into applications.

Our research is related to these efforts, but with more focus on how to realize application-specific policies for "high integrity" software, and on how to support the patterns of mediators and monitors in diverse sets of infrastructure-based commercial standards. We found that we could develop fairly powerful constraint checking services with mediators and monitors. Our initial investigation focused on how these mediators and monitors could be configured to offer a variety of services applicable to "High-Confidence Software" (HCS), allowing for a development model that provides a separation of the constraint checking for policy enforcement from the core logic of the system. The assurance case, or argument, as to whether the HCS was guaranteed to satisfy its policy constraints, can be directly tied to these mediators and monitors, and thus can limit the scope of any "recertification" activity. At the same time, we found limitations in the approach with respect to portability to other EJB container implementations, due to the non-standard internals of the container implementations. The interceptor interface is not in the EJB standard; therefore, our extensions to the JBoss container can not be assumed to be portable to EJB containers from other vendors. Additionally, as it is an internal interface, future versions of the container may change in a manner that could invalidate our augmentations, requiring some level of rework on our infrastructure augmentations to enable them to work with a new version JBoss.

In general, the coupling introduced by having the assurance argument extend further into the infrastructure (i.e., beyond the bounds of the mediators and monitors and into the container) increases the likelihood of recertification as the infrastructure evolves. Since we view commercially available infrastructure as essential in component and service-based compositions, such coupling may limit the achievable benefits of our approach. So the question arises as to what can be done to reduce such coupling. Web Services frameworks offer standards-supported composition mechanisms for loosely-coupled federations. Building on such a framework offers the potential for further decoupling of policy enforcement from infrastructure. We are investigating whether the patterns that we've experimented with in the J2EE framework can be tailored to work in a Web Services setting, and work in a manner that reduces the assurance argument coupling that we see as problematic in an evolving J2EE server world.

The next version of the J2EE standard includes Web Service compatible interfaces for J2EE components. This will allow applications to be built by federating loosely-coupled services that reside on a variety of platforms. Rather than using J2EE-specific packaging and transport mechanisms, Web Services use a more general standards-based mechanism. There are a number of different J2EE standards that allow J2EE components to be accessed from Web Service clients. Java™ API for XML (JAX)-RPC [8] is one of the standards for Web Service enabling J2EE components. Like other J2EE standards, JAX-RPC specifies the notion of a container, and also specifies that pluggable interceptors (SOAP handlers) be part of the container. Later in this paper we will discuss the application of our work to the JAX_RPC infrastructure.

3. POLICY ENFORCEMENT

As mentioned earlier, it can be challenging to predict whether a system composed of a collection of components and services will satisfy a predefined set of application-specific policies (or properties). For example, suppose we have a service (S) that has the requirement to build and supply the current picture of the battlespace. A battlespace picture includes information such as the location of blue and red forces, weather conditions, supply levels, etc. This service is composed with other software to form an application. Also suppose two independently developed applications use this service as their source for battlespace information, and that each application can modify the picture stored in the battlespace service. One application (A1) is responsible for situational awareness (e.g., displaying and modifying the locations of blue and red forces in a specific geographic area). The other application (A2) is responsible for situational analysis (e.g., determining the best way to attack an opposing force. Assume also that there is a strong economic incentive to leave the battlespace service unchanged, as it is part of a common infrastructure. Finally, assume in the deployed configuration both A1 and A2 use the same instance of S.

Further suppose that A1 and A2 each have a set of policies P1 and P2 to which that they must adhere. Assume A1 has a policy $p_1 \in P1$ that states it must update the battlespace picture with new information every second, and assume A2 has a policy $p_2 \in P2$ that says the application must never display classified information. A problem can arise if application A1 is deployed after application A2, and A1 updates S with classified information that would cause a violation of p_2. Conversely, if A2 is deployed after A1, it could overload S, causing a violation of p_1. In general terms policy violations can arise if any policy $p \in P1$ depends on some characteristic of S, and A2 can affect that characteristic, or if any policy $p \in P2$ depends on some characteristic of S, and A1 can affect that characteristic.

Using current practices this is a solvable problem, but it could require modification of the previously built applications. For example, A2 could be modified to ignore the new type of information A1 is writing into S. But this means A1 is levying requirements on A2, an application that might not be under the control of the A1 developers. So, the question arises, is there a way for policies to be enforced and to predict that a composition will be valid without levying requirements on outside applications. The rest of this section will explore a solution to this problem.

The interceptor pattern is a general pattern for mediating communication between components [6]. This pattern can be extended to support configurable, application-level policy enforcement. Building on the previous example, suppose we define the set of policies U as the union of $p \in P1$ policies that depend on characteristics of S that can be altered by A2 and the $p \in P2$ policies that depend on characteristics of S that can be altered by A1. That is, U is the set of policies that depend on the modifiable characteristics of the S, and thus are threatened by the composition or federation of applications A1 and A2. In our example U contains p_1 and p_2. Figure 1 shows how p_1 and p_2 can be enforced in infrastructure technology. We use the term interceptor to refer to a module or component that intercepts the method level calls between the client of a service and the supplier of a service. Interceptors can exist in either the client or the service context, receive control before and after service invocation, and can append to or remove information from the call stack. A policy can be enforced by an interceptor if, and only if, the policy is expressible in terms of the state and operations visible in the composition. In our example, p_1 is enforceable assuming the calls to methods that update S can be selectively redirected to an externally hidden high priority update interface of S, and that this redirection can not be overridden by another mechanism. Likewise, p_2 is enforceable assuming classified information returned from S can be removed from the return stack before it is available to the client. If S did not offer an externally hidden high priority update interface, p_1 would not be enforceable without modifications to S.

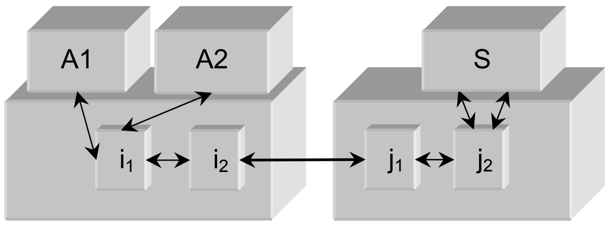


Figure 1 Generic Interceptor example. Calls to S are routed through a set of interceptor pairs (i_1, j_1, i_2, j_2). The interceptors enforce their associated policies (p_1, p_2).

To continue this example, assume a logistics application (A3) using the common S is developed. A3 has a policy (p_3) that the application should never display position-related information about Special Forces units. Since this policy does not interfere with the policies currently in place, as defined in U, and does not modify the state or require additional services of S, it can be implemented in interceptor pair i_3, j_3, which can be placed in the interceptor stack without impacting the behavior exhibited by the pairs i_1, j_1 and i_2, j_2. Interceptor j_3 filters the information from S before it reaches the client. This means previously developed clients can be used in A3 and also that the semantics of S's interface can be modified for clients without modifying S.

Of course, not all policies can be implemented in additional interceptors. Assume a missile tracking application (A4) has a policy (p_4) that states the application must update the battlespace picture every second. But p_1 and p_4 are conflicting policies (i.e., S might not be able to process updates from both A1 and A5 within the one second interval). Also, applications might introduce changes to the state or information contained in S that might cause the violation of previously established policies. Assume a threat tracking application (A5) is developed and that it updates threat information in S by adding references to the unit detecting the threat. In doing so, S might add positional information about Special Forces units to the information in S. This modification can cause the violation of p_3 which prohibits A3 from displaying positional information about Special Forces units. This violation could be corrected by modifying j_3 to filter out Special Forces positional information. A benefit of this approach is that re-implementing j_3 is likely to be easier and present fewer re-certification issues than re-implementing parts of the core application logic.

In this section we described how application level policies can be enforced across a federation of services by infrastructure-level interceptors. We believe enforcing policies at the infrastructure-level has some advantages. First, it supports the principle of separation of concerns; components and services are relieved of the responsibility of policy implementation and can concentrate on business logic. Second, components and services that are free from policy concerns make better reuse candidates; components and services free of application-specific policy constraints can be more easily reused in other contexts. In the previous examples the same client and service software is used in each application. Third, infrastructure-level policy enforcement can make some aspects of certification easier. For example, the problem of certifying an application moves from the component and service level to the infrastructure-level. This also can decrease re-certification costs associated with different versions of an application. Finally, we see infrastructure-level policy enforcement as essential support for runtime compositions and federations.

While infrastructure-level enforcement of policies is useful, in its generic form it suffers from a common infrastructure problem, namely that it is very hard to get multiple programs and developers to single up on a specific infrastructure. However, acceptance of the approach can be eased if the general pattern can be readily mapped to a variety of well-accepted infrastructures. We have previously described infrastructure extensions for policy enforcement in component-based systems, focusing on the J2EE framework. In the following section we will show how infrastructure-level policy enforcement maps to a standards-based Web Service infrastructure.

4. INFRASTRUCTURE SUPPORT

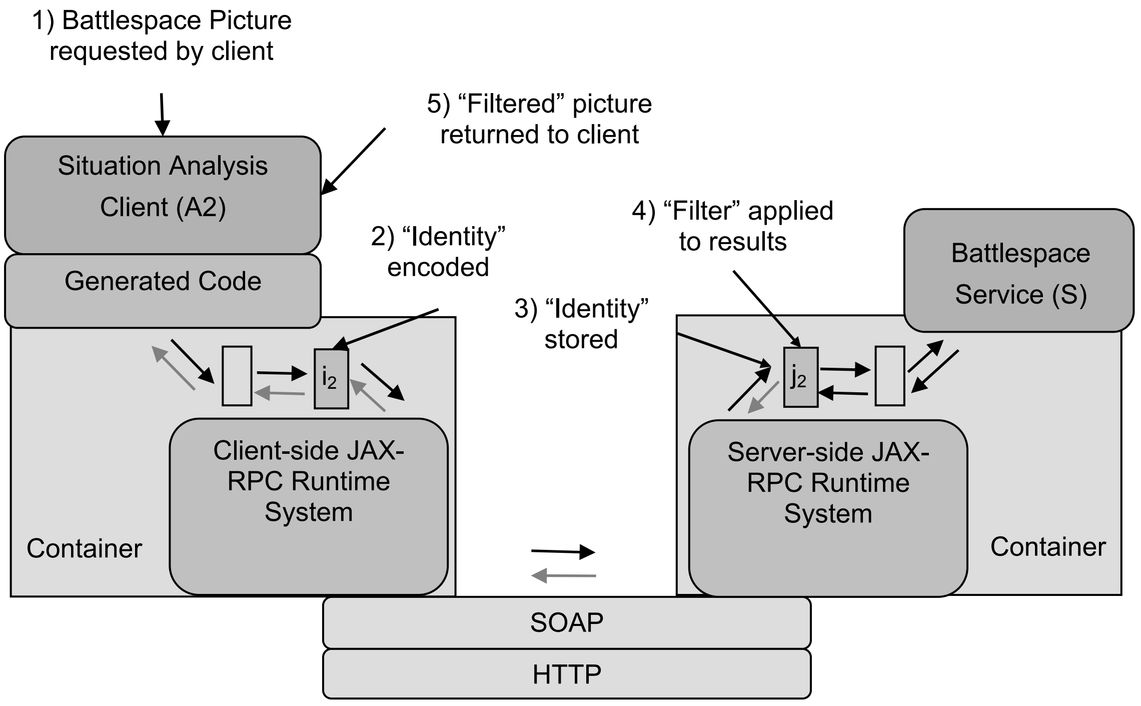
Continuing with the example from the previous section, assume A2 is a J2EE-based application that has an application level policy (p_2) that states the application must never display classified information. This application uses a J2EE-based Web Service (S) that supplies A2 the current picture of the battlespace. Figure 2 shows how this policy can be implemented in a JAX-RPC infrastructure.

In this instantiation of the policy enforcement pattern, JAX-RPC SOAP message handlers serve as the client and server side interceptors. The SOAP message handlers are installed and configured based on the policy that is included in the service's deployment descriptor. A situational analysis client application (A2) requests information from the battlespace service. Without involving the client, the JAX-RPC client-side interceptor (i2), encodes the client's identify into the SOAP message header. This message is then sent across the communication infrastructure to the JAX-RPC server-side interceptor (j2) where, without involving the service, the client's identity is stored. When the SOAP message handler receives control after the battlespace service has processed the information request, if as determined by the stored identify, the client is the A2 application, the interceptor filters the information before returning it to the client.

In this example the interceptor pattern separates the information access policy concerns from the core business logic of the battlespace server. The pattern allows the information access policy to be modified without changing the client or service components. For example, assume the information access policy is changed to allow classified information to be sent to clients, but only if the subnet the client demonstrates it is in the same building as the service. Since the implementation of this policy change is isolated to the SOAP handlers, it can be modified without touching the individual components. Even significant changes to the mechanism for establishing credentials can be realized without involving the service core logic, and, with little or no impact on the clients. In addition, the details that tailor the interceptor for a particular use (e.g., identifying the actual subnets that are allowed to receive classified information) can be specified at deployment or even at runtime. The other examples in the previous section are similarly implemented.

5. INFRASTRUCTURE DIFFERENCES

Our previous J2EE work utilized the JBoss J2EE server. The JBoss EJB container has a configurable method level interceptor stack on both the client and service side. This stack



1) Battlespace Picture requested by client
5) "Filtered" picture returned to client
Situation Analysis Client (A2)
Generated Code
2) "Identity" encoded
i2
Client-side JAX-RPC Runtime System
Container
3) "Identity" stored
4) "Filter" applied to results
Battlespace Service (S)
j2
Server-side JAX-RPC Runtime System
Container
SOAP
HTTP

Figure 2 Enforcement of an information exchange policy (p_2) in interceptors i_2 and j_2. The definition of "Identity" is flexible, in this case identity would identify the user as a situational analysis client. "Filtering" could be any operation (e.g., using XSLT to transform the returned data) to eliminate "sensitive" information.

allowed us to plug in interceptors that received control, and had access to, the actual parameters of each method invocation and return. The main differences noted when these patterns are applied to these two technologies (J2EE vs. JAX-RPC) is interceptor portability across implementations, and mediator access to the method invocation actual parameters. The EJB standard does not prescribe that implementations support interceptor plug-ins. The JBoss developers made this design tradeoff themselves. Other J2EE servers may or may not support a similar concept. Conversely, the JAX-RPC standard specifies support for SOAP handlers. As would be expected, the EJB specification does not say anything about how actual parameters are passed, and whether or how they can be accessed by an interceptor. JAX-RPC does specify the way actual parameters are passed, and thus a general mechanism for parameter access can be developed. This means that mechanisms developed for a JBoss EJB container environment offer little guarantee of portability to other J2EE environments. On the other hand, the JAX-RPC technology offers the ability to create more portable interceptors.

6. CONCLUSIONS/FUTURE DIRECTIONS

Enforcing application-level policies at the infrastructurelevel offers potential advantages in term of component and service reuse, recertification costs, and predicting properties of a composition. This approach appears to be a useful mechanism for enforcing method-level preconditions, postconditions, and invariants; applications level invariants; and information access policies. In addition, this approach has other potential uses, like monitoring the behavioral characteristics (e.g., invocation latency) of applications based on Web Service technologies.

We have experimented with J2EE-based technologies (EJB, JAX-RPC) and have interceptor patterns that can be applied to these infrastructures. We are encouraged to see JAX-RPC supporting the interceptor pattern at the technology specification level. Our current plans for this year include extending our work to include enterprise service bus technologies like SonicXQ [9]. If successful, this technology will allow us to apply the interceptor pattern to applications that integrate components deployed on a variety of platforms, including those outside the Java language.

7. REFERENCES

[1] Meyer, B., “Applying design by contract," IEEE Computer, 25(10): 40-51, October 1992.

[2] Kiczales G. et al., “Aspect Oriented Programming,” Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP), Finland, 1997.

[3] Java™ 2 Platform, Enterprise Edition (J2EETM) http://java.sun.com/j2ee.

[4] Vecellio, G., W. Thomas, and R. Sanders, “Containers for Predictable Behavior of Component-based Software”, Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly, May 19-20, 2002.

[5] Vecellio, G., Thomas, W., Sanders, R., “Container Services for High Confidence Software”, Seventh International Workshop on Component-Oriented Programming, June 10, 2002..

[6] Schmidt, D., et al., Pattern-Oriented Software Architecture Volume 2 Patterns for Concurrent and Networked Objects, Wiley, 2000.

[7] Filman, R., et al., “Inserting Ilities by Controlling Communications.” Communications of the ACM, vol. 45, no. 1, January 2002.

[8] JAX-RPC, Java™ API for XML-based RPC (JAX-RPC) Version 1.0, JSR-101 Java Community Process. http://java.sun.com/xml/jaxrpc.

[9] SonicXQ™, Sonic Software..

[10] Schwartz, M., and Task Force on Bias-Free Language. Guidelines for Bias-Free Writing. Indiana University Press, Bloomington IN, 1995.