

# Infrastructure-based Mediation for Enforcement of Policies in Composed and Federated Applications

Gary Vecellio ([vecellio@mitre.org](mailto:vecellio@mitre.org)) & Bill Thomas ([bthomas@mitre.org](mailto:bthomas@mitre.org))

The MITRE Corporation  
7515 Colshire Drive  
McLean, VA 22102-7508  
(703) 883-6000

## 1. Introduction

By design, a federated “system of systems” application offers limited control over the components and services that it is built upon. However, as the application evolves, there will be changes required to the components and services that comprise the application. As these services are likely to support multiple federated applications, not all will evolve in the same manner. It is essential to limit the scope of change resulting from this application coupling. We see the separation of “policy enforcement” from core application logic as a mechanism to better decouple federated applications. Such separation can allow policy to evolve without impacting other applications or the core services, and allow the components to evolve and still be subject to the same policy.

While such separation can be done in an application-specific manner, the lack of control over the components and services of the federated system makes it likely that such ad-hoc approaches will be of limited value. We need a common, configurable approach for this policy enforcement. Our position is that as components and services increase in size, their widespread use will require the configurable declaration and enforcement of application specific policies. As these policies are on the application, i.e., the composition, rather than the components, we see the infrastructure as a logical place for this capability. We are investigating mechanisms to tailor existing infrastructure to better support these needs. In particular, we are investigating approaches to decouple policy enforcement from application core logic by performing the policy enforcement within configurable “mediators” that are activated on component/service method invocations.

Our work has a focus to address needs of “high confidence software,” where the ability to constrain behavior is essential in enabling efficient verification. Levying a policy across a system is one such approach for implementing behavioral constraints, and if applied in a consistent manner, can allow for improved efficiency in verification and re-verification. Policy Enforcement refers to general class of assurance mechanisms, but one that typically is realized within the application software and is tied to the underlying technology. Our approach centers on the idea that looser coupling can be achieved by adding generic property enforcement mechanisms to the composition and federation infrastructure. We support this through independent, configurable mechanisms including mediators, that intercept calls to a service to enforce compliance, and monitors, that perform periodic checks to ensure continued compliance.

The following sections present our approach to infrastructure-enabled policy enforcement, describe a hypothetical example application that features an “information guard” policy, and, finally, compares the capabilities of two composition/federation infrastructures in terms of their support for this approach.

## 2. Software Policy in Composed and Federated Systems

We view a software policy as a collection of software related properties and procedures. For example, a safety policy is the collection of procedures for building the software and properties the software must

enforce at runtime, and that, if not met, may have an impact on safety. The safety argument for the system thus relies on the enforcement of this policy. Policies can be divided into three classes:

- procedures applied in the development, verification, and qualification of the software (e.g., RTCA-DO-178B[7], MIL-STD-882D[6])
- capabilities provided by the system (e.g., the system must monitor and display the health of the system components)
- constraints on the behavior of the system (e.g., weapon release is never allowed if the status of the tracker is in an error state).

Our focus is on mechanisms to support this last category of policies – those that serve to constrain the behavior of the system. A variety of approaches can be used to place such constraints on the behavior of a software-based system. One way is through the addition of invariant checking and error handling code to the various components of the system. However, experience has found that constraining system behavior often introduces additional coupling between the parts. That is, constraining the behavior of a system often requires knowledge about the state of a number of, at times unrelated, components. Limiting this form of coupling is particularly important to our sponsors because these dependencies can greatly increase re-verification costs when systems undergo modifications or reconfigurations.

We view an application state as the collection of the values of the state variables of the components of the system, and the application state space as an n-dimensional space whose elements include all possible state variable value combinations. A policy may define a subset of the state space, identifying the states in which that policy is satisfied or not. Policy Enforcement within an application refers to ensuring that the application starts in a state that satisfies the policy, and does not transition to a state that violates the policy (i.e., a state outside the subset defined by the policy). That is, we constrain the application behavior to remain within the bounds of the application policy.

This approach for policy enforcement centers around performing actions in response to pre and post-condition evaluations. The actions that are performed can involve data transformation (e.g., tagging or filtering the data), or alteration of the control flow (e.g., raising an exception or redirecting the call,) and which are activated based on the evaluation of a condition. The conditions are represented by an evaluation point (e.g., method invocation or method return, timer expiration) and an expression involving application state, and which can involve aspects of data, control flow or quality of service. Thus policy enforcement involves establishing the policies on the system, evaluating the conditions at the defined evaluation points, and performing the appropriate actions based on the evaluation of the conditions.

Software compositions and federations result from the aggregation of independently developed functionality (i.e., components and services). Compositional frameworks and federation technologies support the deployment and execution of component- and service-based systems. These technologies have been increasingly adopted in the e-business domain, and the reported successes with them has increased interest in their use in other domains. We have investigated how such technologies may apply to “high-confidence software,” software in which it is essential to be able to constrain and verify application behavior. We see the separation of policy from the core application logic as being a general approach for placing such constraints on an application in a manner that the system components can better evolve independently. As such separation is not directly offered by these frameworks, we have investigated how they can be extended to provide such support.

Our experimentation with the capabilities offered by two framework technologies suggests that augmentations to the frameworks can provide much of the capabilities required to separate policy enforcement from the application. We see several potential benefits to this separation. First, as a means for separation of concerns, it offers the potential for productivity improvements by allowing developers to focus on their area of expertise (i.e., either application core logic or policy enforcement). For evolving applications, it can ease the re-verification effort associated with the modification of applications or components (as well as application policies). Finally, by making the policy explicit and decoupled from the application, we believe that it will be easier to evaluate a set of policies for potential policy conflicts.

### 3. Infrastructure-Enabled Policy Enforcement

While capabilities for this type of policy enforcement have been developed and integrated into applications, they have typically been applied at lower levels in the application, and at a smaller scale. With the rise of component-based applications and federated systems, the components of the system are at a large grain, and provide less visibility into their internals. It is well accepted that some form of infrastructure or framework support is essential for effectively integrating component and service based applications. These infrastructures offer some support for policy enforcement, but that support has often been limited to development-time mechanisms. Furthermore, since verification of the policy depends on both the application and its underlying infrastructure, the verification of ad-hoc approaches for enforcement can introduce unwanted coupling of the application to the infrastructure. This can result in significant re-verification effort when making even small changes to the application or the application policies.

In cases where policies are relatively unchanging, such re-verification needs may not be such a concern. However, for applications that are intended to be more dynamic, the increased re-verification effort due to the infrastructure coupling can become prohibitive. One approach toward decoupling the policy from the application is by wrapping sets of components or services with specific policy enforcement wrappers. As wrappers are an application-specific technique (i.e., not performed by the infrastructure), it is a useful technique for slowly changing applications that use a fairly fixed set of components. However, many of such applications are changed frequently, if not extensively, and some may need to be configured at deployment-time and /or run-time. Both of these factors minimize the effectiveness of wrapping.

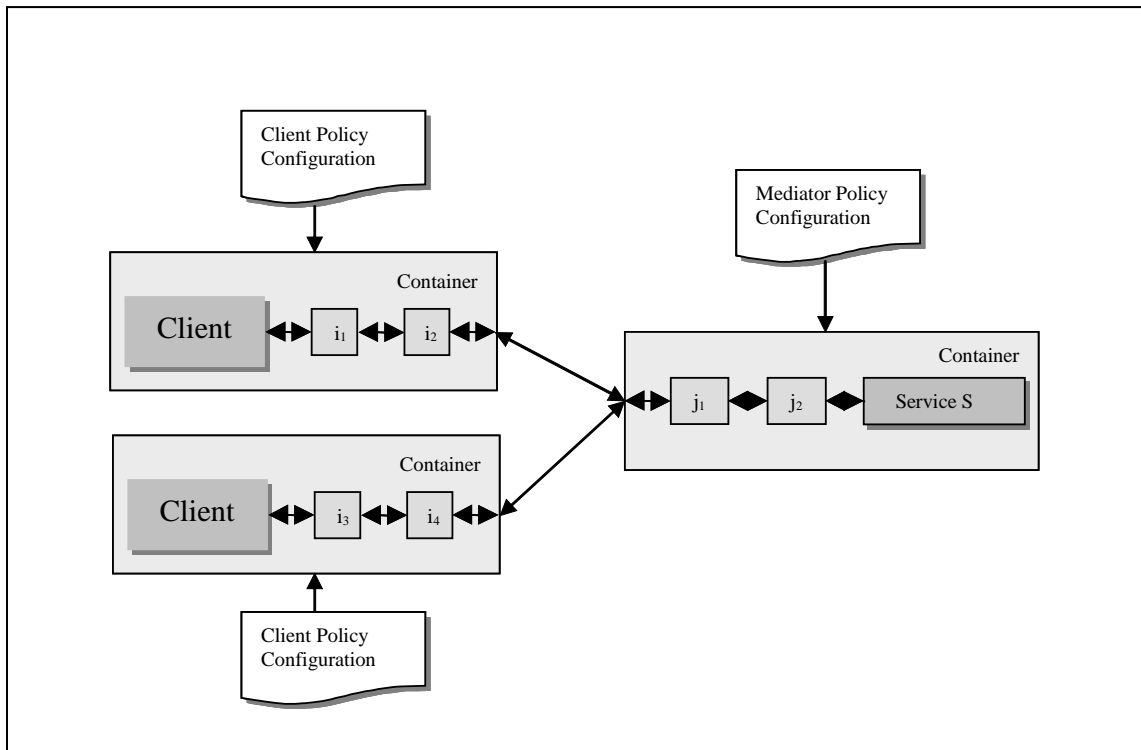
To address these challenges, we are investigating policy enforcement mechanisms that can be, from the component and service viewpoint, transparently applied, and remain separate from the application. Thus, as we identified in [9, 10, 11], there is a need for additional deployment-time and run-time infrastructure support to allow policy enforcement to be decoupled from the application in this manner. As such, we have been investigating how commercial infrastructure technology can be extended to support policy enforcement as a consistent, tailorable, and portable approach that can build on and evolve with both the applications and the infrastructure.

Our approach to decouple policy enforcement from core application logic is based on an interceptor approach, where the client and service operate independently and transparently from the policy enforcement. Policies are enforced by intercepting call to a service and performing actions to ensure that the policy is being satisfied. From a control flow perspective, our approach is based on the ability to:

- intercept any call to a service and perform deployment-time configurable actions
- configure the interception to perform different actions based on characteristics of the method being called and the data associated with the call (e.g., parameters, metadata supplied with the call)
- append information (metadata) to the call to allow downstream actions

To illustrate our approach, we have developed an example where multiple applications make use of a common Track Server. As shown in figure 1, two applications make use of a track service S, application 1 consists of the client C1 and S, and application 2 consists of C2 and S. C1, C2 and S are deployed in containers that specify an interceptor chain. These applications must satisfy an information guard policy that depends on both the Server S and the clients C1 and C2. This policy is enforced on the communication between the clients and the servers via interceptors on both the client ( $i_1$ - $i_4$ ) and the server side ( $j_1$  &  $j_2$ ).

In this example, the service S responds to requests for tracks (e.g., a collection of sensed and predicted position information). The clients C1 and C2 make requests to the service for lists of tracks. The policy determines whether all tracks should be made available to the clients, or what subset of the tracks should be made available. In this example, the “environment” of the clients determines whether the client should have visibility to the complete list of tracks; i.e., local users are authorized to see all tracks, remote users see a filtered set (e.g., not showing special forces positions). We want the clients to make the request in the same manner, and the server to respond to requests in the same manner, regardless as to whether the client is local or remote. We do this by having the interceptors on the client side (e.g.,  $i_1$  and  $i_2$ ) encode identity information and attach that as metadata to the service call. The server side interceptors ( $j_1$  and  $j_2$ ) decode



**Figure 1. Infrastructure-enabled mediators for policy enforcement. Container configuration files determine the interceptor chain for each container. Interceptors in the client ( $i_1$ - $i_4$ ) and the service ( $j_1$ - $j_2$ ) containers are transparently activated on calls to and responses from the service S.**

the identity information from the call, and pass the request on to the service. If the identity information indicates a local user (e.g., from the same subnet as the server), the full set of information is returned. Otherwise, the server side interceptor filters the data, removing all restricted tracks. The filtered data is then returned to the client.

The key points are that the identity encoding and filtering policy is completely separable from the client and service implementations, the policies are configurable, and the policy enforcement implementation is enabled by the infrastructure, but is still independent of the container implementation.

#### 4. Infrastructure Capabilities for Policy Enforcement

Some considerations when understanding how the infrastructure supports our model for policy enforcement include:

- the points at which the condition evaluations can be activated,
- the level of access to the “state” of the application,
- the actions that can be performed based on the condition evaluation
- the mechanisms to establish the policies on the system

In the following sections we compare and contrast our ability to extend J2EE [2] and JAX-RPC [3] technologies in each of these areas. For our J2EE experimentation, we investigated augmentations to the JBoss EJB container and server [4]. Our extensions include definition of new container types and modifications to deployment descriptor and server configuration files. These extensions support our development of mediators, which control communications among components, and monitors, which periodically ensure the state correctness properties outside the context of component communications. For JAX-RPC, we experimented with augmentation to the Apache AXIS framework, primarily involving

adding SOAP handlers to evaluate certain types of conditions and perform selected actions on method invocations calls. These handlers allow us to develop mediators similar to those described above to control communications on method calls and responses. The JAX-RPC framework does not offer support for the development of a similar monitor capability as we have experimented with in the J2EE framework, so under JAX-RPC, all our application monitoring has been limited to capabilities based on method calls and responses.

#### **4.1 Establishing policies on the system**

The JBoss server and container are based on the Java™ Management Extensions (JMX) framework. The container is built on a JBoss framework that includes the Interceptor Interface. Interceptors are selected by specifying them in an XML-based deployment descriptor file, and are instantiated and chained together by the container factory. The last interceptor in the chain is the container itself, which invokes the business method of the EJB. Our primary augmentations to establish policies on the system are in the development of container types with a tailored deployment descriptor that specifies the insertion points of our additional application-specific interceptors. More recent releases of JBoss (e.g., JBoss 3.0) include configurable containers for both the server and the clients. Thus similar interceptors can be deployed on the client side as well as on the server side, and are established in the same manner as described above. While our experiments in this area have been relatively straightforward (i.e., the deployment descriptor simply specifies the augmented interceptor chain for the container, and the logic of the policy enforcement is in the interceptor implementation), the JBoss environment does not preclude a more complex and capable deployment time and run-time configuration of policies. However, any such solution will likely be highly coupled to the JBoss environment.

In the JAX-RPC framework, our mediators are realized as additional SOAP handlers that are activated on method calls and/or responses. These SOAP message handlers serve as either client or server side interceptors. On the server side, the SOAP message handlers are installed and configured based on the policy that is included in the service's deployment descriptor. Client side handlers are handled similarly. This is done in a manner comparable to that in the JBoss J2EE environment, but is done in a standards-supported manner, and thus can be portable across implementations.

#### **4.2 Condition Evaluation Points**

The JBoss Container Framework offers a rich set of events upon which the interceptor chain is activated. These include EJB installation and startup as well as method calls and responses. There is a single interceptor chain for the container, with the event that is being processed being passed in succession to each interceptor in the chain. Thus actions can be readily tailored to be specific to all such events of the container or a wide variety of subsets of these events. For example, the interceptors can be made specific to the startup of and EJB, or to a call to a single method of that EJB, or to a call to any one of collection of methods of the EJB. However, this ability to tailor the interceptors to be specific to certain events is done within the logic of the interceptor chain rather than being an inherent part of the chain. Thus reconfiguration of interceptors to apply to a different set of events can be done in a straightforward manner only if one has designed such configurability into the mechanisms for interceptor deployment. Otherwise, such a change may require changes to the individual interceptors rather than just the deployment descriptors.

The SOAP message handlers in JAX-RPC are designed to apply to messages and/or message responses. They operate similarly to the interceptor chain described above, as the ordering of the handlers is defined in a configuration file, and the message is passed from handlers to handler in accordance with that configuration. As with the JBoss interceptor framework, the SOAP handlers can be widely tailored to handle a variety of messages or only a very specific subset. However, there is more flexibility in the configuration of the SOAP handlers to have them apply to different messages without involving the handler logic. This flexibility offers more power in reconfiguring policy, but at the same time can increase the difficulty in identifying and resolving potential conflicts among sets of handlers.

### 4.3 Mediator Access to Application/Infrastructure State

The type of application state information that is available to the mediators can have a great effect on what policies can be enforced by the infrastructure. We are examining three broad classes of such information: data information, control flow information, and quality of service information. The ability of the infrastructure to provide access to such data can enable the development of increasingly more powerful policy enforcement mediators.

Data information can include simple information being passed as parameters with a call, visible application/environment state data, as well as internal state data. Both the JBoss interceptors and the JAX-RPC SOAP handlers have access to method call parameters. As such, policies involving the data that is being supplied or returned can be evaluated within a policy. In our “information guard” example, we want to restrict access to certain data coming from a server, while allowing unrestricted access to other data from that server. We implement this guard policy by having the client container attach to the call some form of “credentials” of the client, e.g., the client’s IP address. These credentials are then evaluated by the server container. If the credentials satisfy the policy for full access (i.e., it is from a local address), all data from the server can be returned. If it does not, the data being returned is filtered to remove the limited access data.

While both frameworks have access to passed parameters, there are differences in their ability to access other data. Using the Java reflection API, the JBoss container can get reflective access to non-public attributes and methods. With JAX-RPC we are limited to public attributes and methods. While this may reduce the power of the mediators, it is not clear in practice to what extent this would present difficulties. In general, we want decoupled, portable interceptors, and thus do not want to rely on internal information.

Control flow information is another type of information upon which we may want to design policies. For example, we may want to ensure that prior to invoking a critical piece for software, the application has recently (e.g., within 5 seconds) performed a call to check the status of all its dependent components. In general such information is not directly available to the mediators in either the J2EE or JAX-RPC environment. While we can perform and access some form of logging with either framework, we are limited in the ability to verify control flow. For such cases it may be more appropriate to tailor the policy so that the mediator actually performs the status checks rather than attempting to verify that they had been previously made, as is described in [9, 10].

The third category of information upon which we want to build policies involves “Quality of Service” (QoS) information, i.e., information regarding the behavior of the service. An example of this type of policy is one where we want to redirect certain calls to a “high-priority” interface if the predicted response time exceeds a certain threshold. While we can not directly ensure that response time will be within a specified limit, we can monitor response time, and if the trend predicts that response time is likely to exceed the threshold, we can redirect the privileged calls to the high-priority interface without any changes to the client or the server. This QoS is a characteristic of the running component or service, and can change over time due to the demands placed upon it. In the JBoss J2EE environment, powerful, flexible monitors with a high level of access to the server internals can be developed to monitor application performance. The JAX-RPC environment does not provide access to the service internals, and we are limited to monitoring at service call and response times.

### 4.4 Mediator-Allowable Actions

Different infrastructures offer varying support for different types of actions. The primary actions that we are interested in involve data transformation (e.g., tagging or filtering the data), and alteration of the control flow (e.g., aborting a call and raising an exception or redirecting the call). Both the frameworks (J2EE and JAX-RPC) offer support for a wide variety of actions to occur within the mediator, including strong support for altering control flow and transforming data. As with the previously discussed differences with visibility into application state, the J2EE environment offers a greater capability for access to intra-server actions,

which offering more power, but also with a cost of limiting portability. The JAX-RPC approach offers the potential for greater portability, as it limits the use of similar intra-server actions.

## 5. Summary/Conclusions

Our experimentation with infrastructure enabled enforcement of application-specific policies leads us to believe that it offers a number of a potential advantages, including more effective component and service reuse, reduced recertification costs, and an improved understanding of properties of a composition. While our focus to date has been primarily on how this approach can support policy enforcement on policies represented as method-level pre- and post-conditions, we are also investigating monitoring approaches for enforcing application level invariants relating to behavioral characteristics of service-based federated systems.

In part, our work is similar to the goals of Aspect-Oriented Programming [5], as we are developing mechanisms to integrate policy enforcement aspects with core business logic. However, our approach is at the framework level rather than the language level. Related to our work is the research on “meta programming” mechanisms for distributed object computing [8]. Composition mechanisms, such as smart proxies and interceptors, provide 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. Also similar to our work is the Object Infrastructure Framework (OIF) is an approach for achieving non-functional “ilities” through the use of “injectors” attached to communications [1]. In this framework, meta-information is “attached” to CORBA method invocations, identifying a sequence of injectors that are to be processed for that communication. With the OIF, support for a number of diverse “ilities” can be developed, (e.g., reliability, maintainability, quality of service and security) and in a consistent manner, integrated into applications.

We have experimented with technologies for composition (EJB) and federation (JAX-RPC), and have demonstrated interceptor patterns that can be applied to these infrastructures. While the differences in the infrastructure capabilities does impact the potential power of the mediators, both technologies still offer strong support for enforcing application policies. We are encouraged to see the level of JAX-RPC support for the interceptor pattern at the technology specification level. Our current plans for this year include extending our work to include enterprise service bus technologies (e.g., SonicXQ [12]). This technology can allow us to more seamlessly apply the interceptor pattern to applications that integrate components deployed on a variety of platforms, including those outside the Java language. Our initial investigation of the SonicXQ technology indicates that it can be similarly extended to support the approach described in this paper. However, as enterprise service bus technologies are relatively new and are not built to standards, our extensions to the SonicXQ infrastructure are likely to be product specific.

It is clear that frameworks for component and service based development offer great power for composing systems. However, as the system components evolve independently, this development presents significant challenges to limit the effect of unintended application-level coupling. We believe that such coupling can be reduced or more effectively controlled by separation of policy enforcement from core application logic. As these policies are on the application, i.e., the composition, rather than the individual components, the infrastructure is a logical place for this capability, and is essential for growth in composed and federated systems.

## References

- [1] Filman, R., et. al. "Achieving Ilities by Controlling Communications." *Communications of the ACM*, vol. 45, no. 1, January, 2002.
- [2] Java 2 Platform , Enterprise Edition, Version 1.4, <http://java.sun.com/j2ee>.
- [3] JAX-RPC, Java™ API for XML-based RPC (JAX-RPC) Version 1.0, JSR-101 Java Community Process, <http://java.sun.com/xml/jaxrpc>.
- [4] J2EE on JBoss. The JBoss Group. <http://www.jboss.org>.
- [5] Kiczales, G. et al., "Aspect Oriented Programming," *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, Finland, 1997.
- [6] MIL-STD-882D, Department of Defense Standards Practice for System Safety, February 2000.
- [7] RTCA-DO-178B, *Software Considerations in Airborne Systems and Equipment Certification*, Radio Technical Commission for Aeronautics, Inc., 1992.
- [8] Schmidt, D., *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*, Wiley, 2000.
- [9] Vecellio, G., W. Thomas, 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. <http://www.sei.cmu.edu/pacc/CBSE5/Vecellio-cbse5-final.pdf>.
- [10] Vecellio, G., W. Thomas, R. Sanders, "Container Services for High Confidence Software," *Seventh International Workshop on Component-Oriented Programming*, June 10, 2002. [http://research.microsoft.com/~cszypers/Events/WCOP2002/09\\_Vecellio.pdf](http://research.microsoft.com/~cszypers/Events/WCOP2002/09_Vecellio.pdf).
- [11] Vecellio, G., W. Thomas, "Infrastructure Support for Predictable Policy Enforcement," *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly*, (to appear) May, 2003.
- [12] SonicXQ™, Sonic Software. <http://www.sonicsoftware.com/products/sonicxq>.