NGC Project

End-to-End Security Policy Auditing and Enforcement in Service Oriented Architecture

Progress Report: January 2014

and

Related Research

Table of Contents

NGC Progress Report: January 2014

Introduction

In a service-oriented architecture (SOA) environment, services can collaborate to provide broader functionality and accomplish tasks quickly. A SOA service can dynamically select and invoke any service from a group of services to offload part of its functionality. This is very useful to build large systems with existing services and dynamically add services to support new features. One of the main problems with such a system is that from a client (user or service) perspective, it is very difficult to trust the service interaction lifecycle and assume that the services behave as expected and respect the client's policies.

Problem Statement

The SOA problem that we try to address here is as follows. A client interacts with a service, submits its request and gets a response back. The client expects certain levels of security guarantees. These can be expressed as policies about request data, e.g security and privacy policies that dictate how this data should be handled or the policies about quality of service, e.g. processing and response time should be within a certain limit. It is impossible to guarantee whether these constraints are respected and enforced without a monitoring agent. Moreover, if a service is compromised or misbehaves, the monitoring agent needs to discover the malicious activity and provide feedback to the client. There is need for novel techniques to monitor service activity, to discover and report service misbehavior.

Current Solution

We take a two-prong approach for ensuring end-to-end security in SOA. These approaches are:

1. End-to-End Security using Service Monitor
2. End-to-End Security using Active Bundles

The details of both approaches are provided below.

End-to-End Security using Service Monitor

Our current solution using a service monitor audits and detects malicious or compromised services based on a monitoring agent, trust management and policy enforcement. The main features of this solution are the following:

- An agent that can monitor all services in the system and report non-compliance of service behavior with respect to system and subject level policies.
- A trust management module as part of the monitoring agent that can track and manage trust levels of services in the system based on the service state, behavior and system context.
- A transparent mechanism to track, report and block service actions.

- A policy engine and enforcement component to manage and apply the global and subject (user, service) level policies.
- Detecting attacks, malicious service behavior and compromised services by monitoring any illegal service invocations from services inside a trusted domain.

Figure 1 below shows the components of the proposed end-to-end security framework and their interactions. The details of the different components and how they interact is described in the next section.

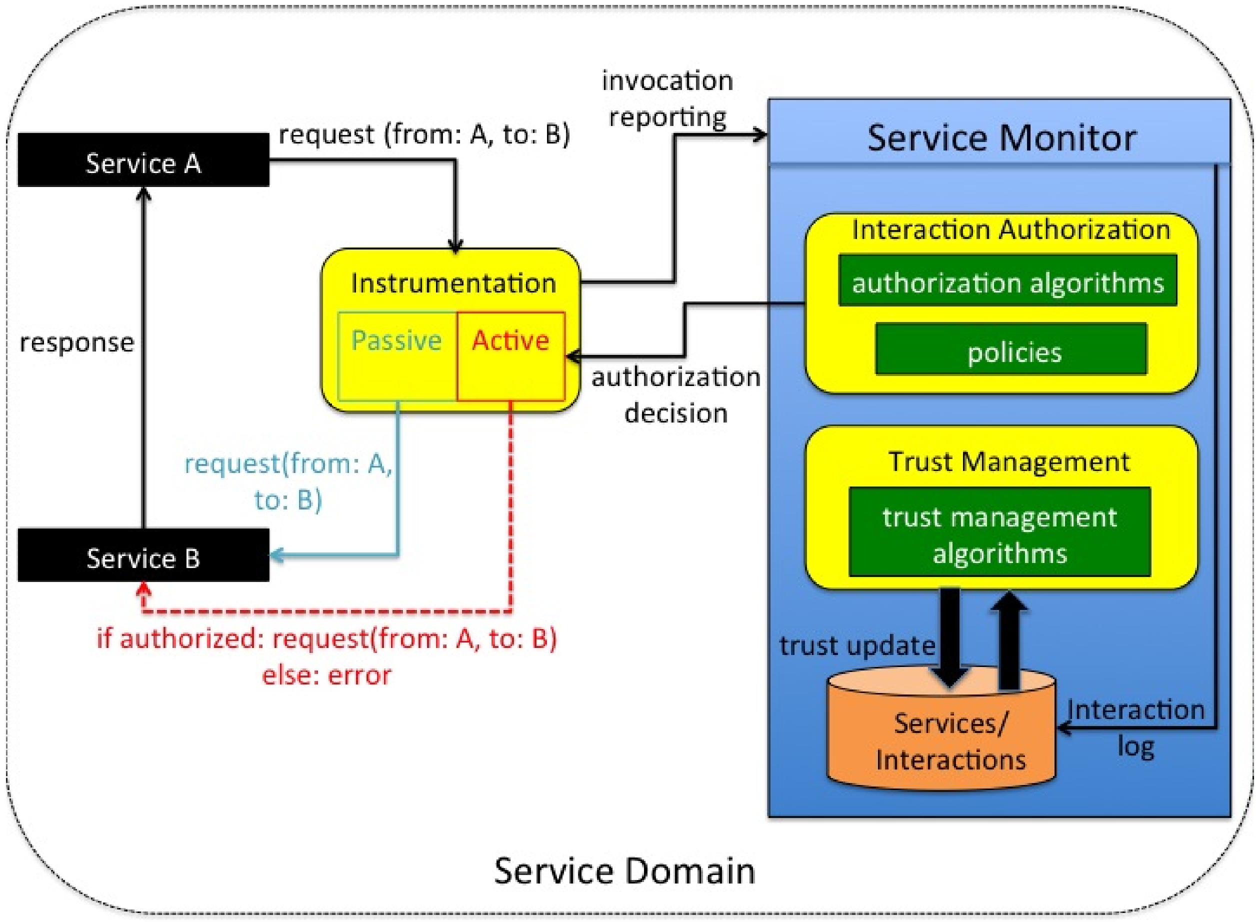

Fig. 1 System Architecture for End-to-End Security Solution with Service Monitor

REST Implementation Overview

We have implemented an end-to-end security framework with the service monitoring approach for REST-based services, using the node.js framework (http://nodejs.org/) for network applications. In the developed framework, end-to-end security is provided by tracking all service invocations (including inter-service communication), and either recording (passive monitoring) or blocking (active monitoring) illegal service invocations (i.e. those not complying with system-level policies). The capabilities of the current framework can be listed as follows:

- Invocations in an end-to-end chain of services for a request can be tracked and recorded.
- Multiple global (system-level) XACML policies can be specified for enforcement on service invocations.
- Service invocations can be actively monitored to block those not complying with global policies.
- Different trust algorithms can be integrated into the framework to evaluate the trustworthiness of services in the system.
- Trust values of services can be updated based on their interactions with other services.

At the heart of the framework is the service monitor module, which includes submodules for managing trust, tracking service invocations and making service invocation authorization decisions. The service monitor and its components are described in detail below.

Service Monitor

The service monitor is a module installed as part of the SOA system. The service monitor was developed as a set of services and a management console with the following features:

- Active and Passive Monitoring Modes
- Service Composition Patterns
- Trust Management
- Interaction Authorization

Active and Passive Monitoring Modes

The service monitor can either record and analyze all service interactions or it can intervene to manage interactions as configured. There are two modes of monitoring: passive and active. Details of each monitoring mode are described below. The implementation details for each mode are provided in the “Instrumentation Modules” section.

Passive Monitoring:

The passive monitoring mode depicted in the figure below involves the recording of all service interactions by the service monitor in the form of (caller, callee) pairs. In this mode, all service invocations are intercepted, but not blocked. The effect of the interception is only the analysis of service interactions and the update of trust levels of services based on the interactions. The recording and trust update operations following the interception are performed in an asynchronous manner transparent to the service invocation.

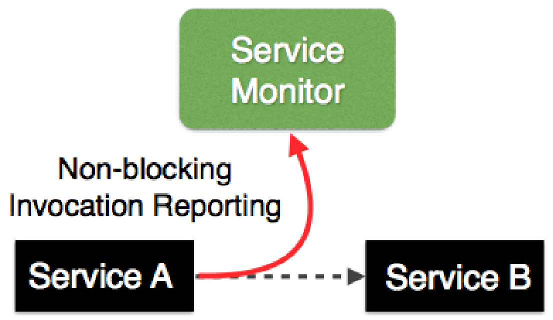

Fig. 2 Passive Monitoring

Active Monitoring:

The active monitoring mode depicted in the figure below involves interception of all service invocations to dynamically check their compliance with user-specified and/or global policies. The service monitor in this mode uses its interaction authorization component to decide whether to allow the service invocation. The service invocation is blocked until an authorization is received based on the policy associated with that specific call, and cancelled if the invocation does not comply with the policies.

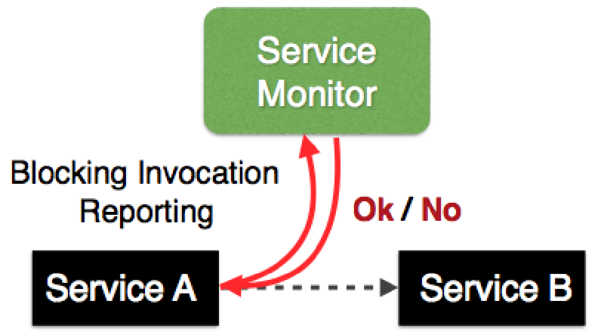

Fig. 3 Active Monitoring

Service Composition Scenarios/Patterns

The framework provides a set of fundamental SOA composition patterns. These include scenarios such as “proxy”, “facade” and “chains”. This set is not an exhaustive list and there are provisions to easily integrate new patterns into the system. The objective of using such patterns is to allow exploration of best practices in monitoring these service compositions as a whole.

Trust Management

The trust management module integrated with the service monitor provides the ability to conveniently integrate and evaluate different service trust management algorithms with the set of available scenarios. This module uses a database of essential information about services including service name, url, port, parameters and trust level. The trust level of a service at any point in time indicates how that service compares to other services in terms of compliance with policies, security requirements, service level agreement etc. This information can be used in service composition policies to restrict invocations to services with trust levels below a certain threshold.

The trust management module is responsible for updating the trust value of services based on various parameters like service state, service behavior and system context. In our current implementation, trust levels of services are updated based on their interactions with other services. Specifically, the trust level of a service *s1* is updated upon its invocation of another service *s2.* The interface implemented by the different trust management algorithms in the framework has the form:

```
trust_update(from, to)
```

where the `from` and `to` parameters include metadata about the caller service (from) and the callee (to). Algorithm implementations are added to the framework as self-contained modules, which expose this function. The current framework implements the trust algorithms described below, and additional algorithms from the rich literature in trust management can easily be integrated into the system by adding their implementation files to a specific directory holding all trust algorithm implementations. The algorithm used by the trust management module at any point in time can be selected dynamically from among the algorithms available in the framework.

Trust Algorithms

1. Simple Average: This algorithm sets the trust value of the caller to the average of the trust values of the caller and the callee.

```
from.trust_level = (from.trust_level + to.trust_level)/2
```

2. Weighted Moving Average: In this algorithm, the trust value of the caller is updated using a weighted average of the trust levels of the caller and the callee. The trust value of the caller is updated using the following formula, where *w* is a real number in the range [0,1].

```
from.trust_level = w * from.trust_level + (1-w) * to.trust_level
```

While a weight in the range [0, 0.5) attaches greater importance to the original trust level of the caller, a weight in the range (0.5, 1] has greater emphasis on the trust value of the callee. For the case of *weight = 0.5*, this is equivalent to the *simple average* algorithm.

Interaction Authorization

The service monitor has the ability to decide the fate of a pending service interaction when a service is under active monitoring. The decision is handed over to an “interaction authorization” algorithm. The interaction authorization algorithm in the current framework makes authorization decisions based on compliance with the global policies in the system.

The interface implemented by the interaction authorization algorithm has the form:

`authorize(from, to, cb)`

Implementations are added to the framework as self-contained modules which expose the above function. The `from` and `to` parameters provide caller and callee metadata. `cb` is the callback provided to return the authorization decision. This authorization decision is communicated by the service monitor to the service instrumentation.

Policy Enforcement

Policy Language:
XACML 3.0 specifications are followed to define policies in XML.

Policy Engine:
WSO2 Balana implementation of XACML 3.0 is used in the framework as the policy engine for making authorization decisions.

Creating and Testing a Policy:
A new policy is defined using XACML and undergoes unit testing using the *policy development and testing module* provided in the framework. An access control request in XML format is used to test all possible scenarios of the policy that generates the expected "Deny" and "Permit" outcomes.

Deploying a Policy:
Once tested, a new policy is integrated into the service monitor. The XML request format is converted to a JSON template. This template is populated with appropriate values by the interaction authorization algorithm when invoking the access controller.

Some examples of these policies are presented in the "Demo Overview" section below.

Instrumentation Modules

REST services are implemented as node.js applications. A node.js application uses the "`request`" module when invoking a REST service. This framework provides two instances of instrumentation of the "`request`" module:

- Non-blocking Instrumentation

  This instrumentation intercepts a REST call to a remote service and forwards interaction metadata to the service monitor application. It is important to note that the call to the service monitor does not block the regular operation of the external service invocation.

- Blocking Instrumentation

This instrumentation intercepts a REST call to a remote service, forwards interaction metadata to the service monitor and waits for the service monitor to authorize the interaction.

In both instrumentations, metadata forwarded to the service monitor includes invoker identity, target service identity, target operation, transport information (HTTP/HTTPS), and start and end times of the operation.

Scenarios/Service Compositions

All scenarios were developed using a set of node.js/express services. These services use the aforementioned blocking and non-blocking instrumented request modules. These services are grouped into scenarios and are independently tested to ensure regular operation of the scenario before being integrated into the service monitor management console.

Source

The source repository of this work is available at: https://code.google.com/p/end-to-end-soa/
Installation and setup instructions are provided in the README file available in the root directory of the source repository.

Demo Overview

The scenario developed for demonstration of the framework features involves a travel reservation service relying on three other types of services (airline, advertising and payment) to complete a service request. Figure 4 below shows a broad view of the scenario.

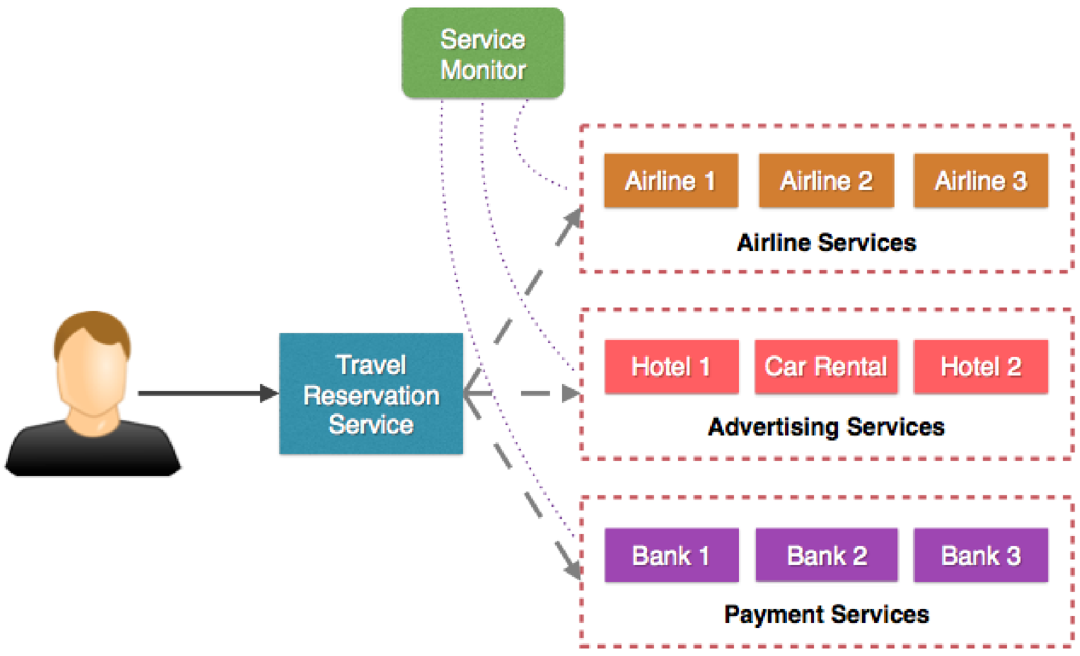

Fig. 4 Travel Reservation Scenario

Figure 5 below provides a sequence diagram for the flow of actions in the system upon a user's invocation of the travel reservation service:

1. The user submits a query to the travel reservation service (TRS) for a flight reservation with specific origin and destination locations and departure and return dates.
2. TRS prepares a request for an airline service, which is intercepted by the service monitor.
3. The service monitor uses its interaction authorization module to decide whether to authorize the invocation of the airline service, using the system level policies (the "no plain text transport policy" described below is relevant here). The trust level of TRS is updated based on the service interaction.
4. If the interaction with the airline service is authorized, the invocation proceeds (asynchronously).
5. TRS prepares a request for an advertising service, which is intercepted by the service monitor.
6. The service monitor uses its interaction authorization module to decide whether to authorize the invocation of the advertising service (the "remote untrusted ad-block policy" is relevant here in addition to the no plain text transport). The trust level of TRS is updated, this time based on its interaction with the advertising service.
7. If the interaction with the advertising service is authorized, the invocation proceeds.
8. TRS presents a response for the query of the user upon receiving responses from both services invoked.
9. The user selects a ticket and submits a payment processing request to TRS.
10. TRS prepares a request for a payment service and the interception-authorization steps above are repeated for the interaction with the payment service (the "block credit card

transmission policy" is relevant here in addition to the no plain text transport).
11. The service request is completed with TRS presenting the response to the user.

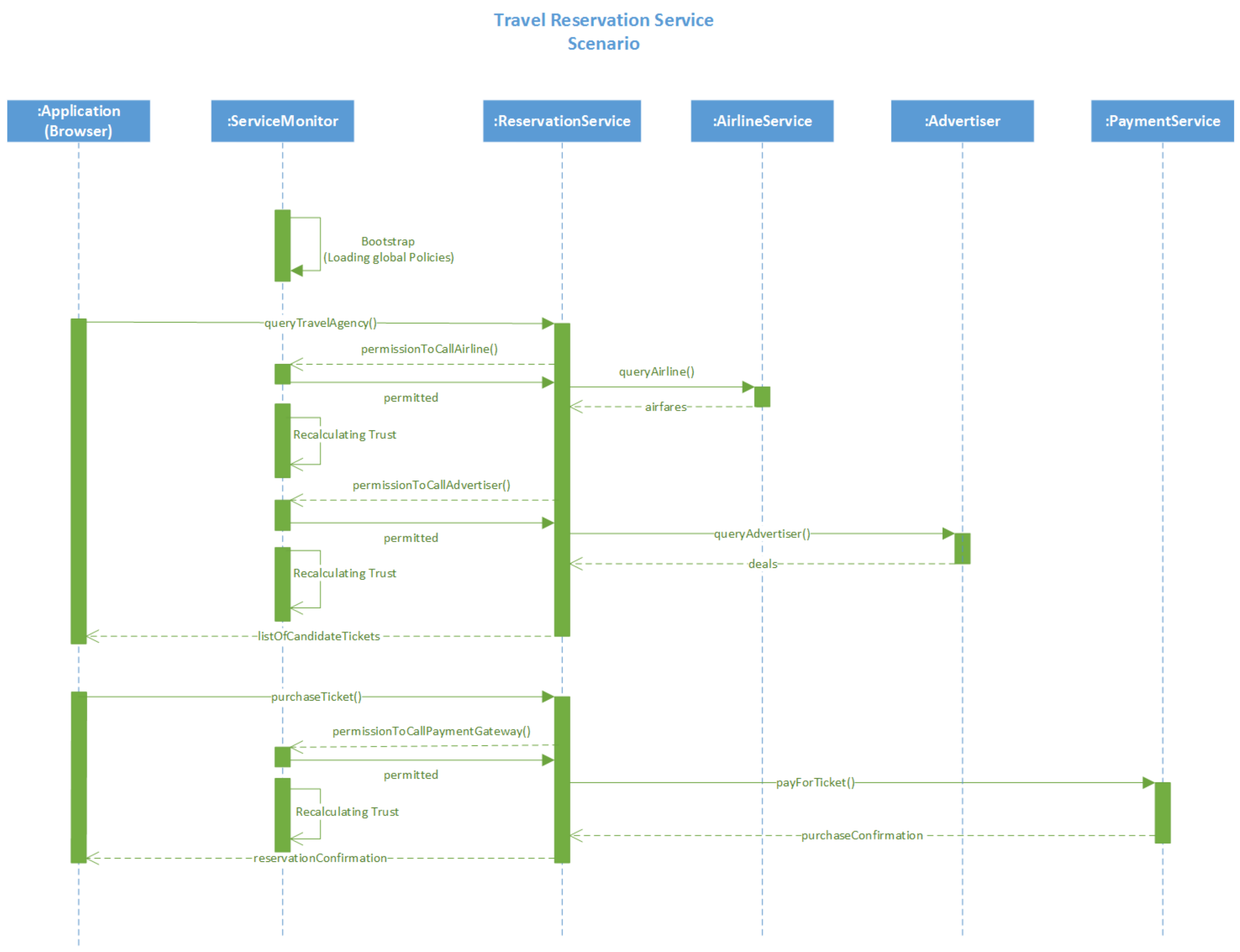

Fig. 5 UML Flow for Ticket Reservation Scenario

Possible Policies

- [Privacy] Service may not communicate with "advertising" services which carry a trust level below a certain threshold. This acts as a remote "ad-block" for the user where he/she wishes not to receive advertisements from services who are not trusted.

- [Privacy] Service may not leak the "address" information of the user to "advertising" services. This policy would prevent leakage of identity information of the user.

- [Confidentiality] Service may not interact with any service over plain text.

- [Confidentiality] Service may not transmit user's credit card information to any external service below a certain trust level.

Policy Implementation Details

The current framework contains implementations of the three interaction authorization policies detailed below. Any policy that can be specified using XACML can easily be integrated into the framework.

Policy 1: Remote Untrusted Ad-Block

Deny	
Resource	any
Subject	any
Action	READ
Env: http://endtoendsoa.cs.purdue.edu/policy/trust_level	< 5
Env: http://endtoendsoa.cs.purdue.edu/policy/operation	substr("advertisement")

This policy blocks invocations of services that include advertisements if those services have a trust level lower than 5. The operation value ('advertisement') is inferred from the url used to access the service (callee).

Policy 2: No Plain Text Transport

Deny	
Resource	any
Subject	any
Action	READ
Env: http://endtoendsoa.cs.purdue.edu/policy/transport	plain

This policy blocks invocations to services that do not use the "https" protocol.

http://endtoendsoa.cs.purdue.edu/policy/transport attribute value will be set to “plain” if data is transmitted over plain text. We infer the value of the transport protocol from the url used to access the resource.

Policy 3: Block Credit Card Transmission

Deny	
Resource	any
Subject	any
Action	WRITE
Env: http://endtoendsoa.cs.purdue.edu/policy/trust_level	< 10
Env: http://endtoendsoa.cs.purdue.edu/policy/data	Includes(Credit Card #)

This policy blocks the posting of a user’s credit card number to a payment service with a trust level less than 10.

End-to-end Security using Active Bundles (AB)

This solution is based on the use of Active Bundles as data carrying agents. The user interacts with a service by sending it an Active Bundle which contains data about user request and the policies associated with the data. An Active Bundle (AB) is a data protection mechanism, which can be used to protect data at various stages throughout its lifecycle. The active bundle is a robust and an extensible scheme that can be used to disseminate data securely across multiple domains. An Active Bundle bundles together, as illustrated in fig. below, sensitive data, metadata, and a Virtual Machine (VM) specific to the bundle.

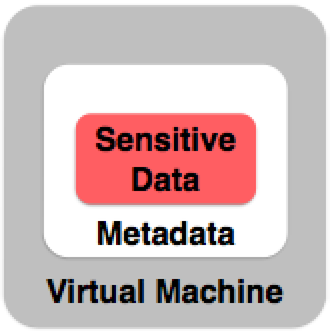

Fig. 6 Structure of Active Bundle

Structure of an Active Bundle

It includes the following components:

- Sensitive data: It is the digital content that needs to be protected from privacy violations, data leaks, unauthorized dissemination, etc. The digital content can include documents, pieces of code, images, audio, video files etc. This content can have several items, each with a different security/privacy level and an applicable policy to ascertain its distribution and usage.
- Metadata: It describes the active bundle and its policies. This can include information such as AB identifier, information about its creator and owner, creation time, lifecycle etc. It also includes policies that govern AB interaction and usage of its data, such as access control policies, privacy policies, dissemination policies etc.
- Virtual Machine (VM): The AB's VM is a specific purpose VM used to operate AB, protect its content and enforce policies (for example, disclosing to a service only the portion of sensitive data that it requires to provide its service).

Active Bundle API

Active Bundle has a generic API to support different functionalities and operations. This is implemented using Apache Thrift framework. The API provides the following public functions for communication with AB:

- **`getSLA()`**

Active Bundle promises to provide data/service with certain guarantees which is represented as AB Service Level Agreement (SLA). This information is defined by the user when creating the AB. This information is public and anybody can get it by calling this function.

- **`authenticateChallenge()`**
  This is used by services for authentication with AB. Service gets a challenge by calling this function. AB internally generates a random UUID which is the token returned as challenge.
- **`authenticateResponse(token, signedToken, serviceCert)`**
  Service signs the token which it gets from `authenticateChallenge()` with its certificate and calls this function passing as the original token, signed token and its certificate as parameters. Service certificate should be signed by a trusted Certificate Authority (CA) so that AB can trust this service. AB verifies that service certificate is valid, has not expired and is signed by the CA certificate or the signing chain leads to the CA. Then AB verifies if the token is signed by the service's certificate, creates a new session for the service and returns a secret session key to securely exchange messages with the service.
- **`getValue(sessionKey, dataKey)`**
  Service calls this function to get data from AB. Service uses the session key as the argument to function. AB verifies the session key and returns the data requested by the service.

Scenario: Online Shopping Portal

User invokes the online shopping service. User searches for items and selects items of interest. User specifies the order through this service, which communicates with an inventory service to find out if the selected items are available in the specified sizes and colors. The order and user's shipping details are sent to another service which applies tax, shipping charges and calculates the total amount due for the order. User submits the order by specifying a payment option. The selected mode of payment is verified through the respective payment gateway service. Once approved the order goes through another service that generates an estimate of shipping time and a tracking number to keep track of the order's status and location.
The main issue in this scenario is that there are multiple services each with a separate functionality, thus each service requires different data to provide its service. For e.g shipping service only needs the address of the user, payment service only needs payment credentials, inventory service needs the information about the items in the order and the online shopping service needs to authenticate user. No single service needs all the information. The research problem is how to selectively disclose information, minimize the unnecessary disclosure and ensure security and privacy of the information is protected. Our current solution uses Active Bundles to achieve this.

Infrastructure Overview

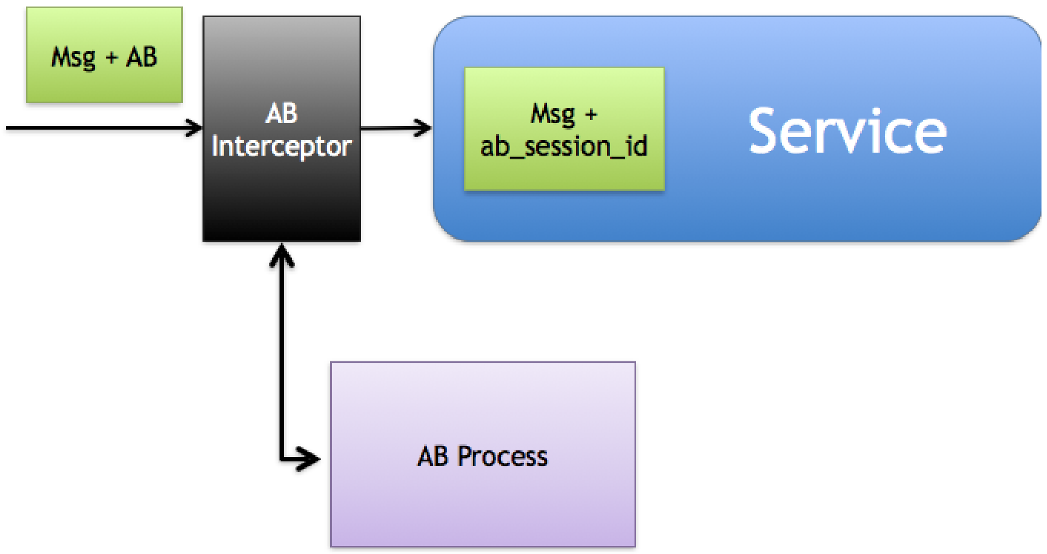

Fig. 7 AB-Service Interaction

The services are setup using Apache Axis2 Server and interact using SOAP messages. The server has an AB Interceptor component that starts the AB process and makes the service aware of its presence. An AB is included in a SOAP header as shown below:

```
<soap:Envelope>
  <soap:Header>
    <ab:ActiveBundle xmlns:ab="http://absoa.cs.purdue.edu/ns/">
            [BASE 64 Encoded ActiveBundle.jar]
    </<ab:ActiveBundle>
  </soap:Header>
  <soap:Body>
    ...
  </soap:Body>
</soap:Envelope>
```

The Active Bundle is created along with its data and packaged as an executable jar file. This is read by a service client program and Base64 encoded. The result is added as a header element called 'ActiveBundle' under the namespace 'http://absoa.cs.purdue.edu/ns/'.

The AB Interceptor is placed before the SOAP message reaches the intended service. This is implemented as an Apache Axis2 module with a single handler on the inflow of the service. Upon invocation, this handler performs the following steps:

- Extract the Base64 encoded active bundle jar file from the 'ab:ActiveBundle' header
- Decode the active bundle (and optionally verify the signature of the active bundle jar)
- Store active bundle jar in a temporary directory in the file system
- Generate a random port number
- Start the active bundle as an independent process and provide the above randomly generated port number

- Set a pointer to the process instance and the port number in the Axis2 MessageContext (This is required to access the running active bundle at the service implementation code, and to manage the active bundle process)