Étude des architectures de sécurité orientées service

par

Alexandre Beaupré

Mémoire présenté au Département d'informatique
en vue de l'obtention du grade de maître ès sciences (M. Sc.)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, septembre 2012

Your file Votre référence
ISBN: 978-0-494-91659-9

Our file Notre référence
ISBN: 978-0-494-91659-9

Le 20 septembre 2012

le jury a accepté le mémoire de Monsieur Alexandre Beaupré dans sa version finale.

Membres du jury

Professeur Marc Frappier
Directeur de recherche
Département d'informatique

Professeur Sylvain Giroux
Membre
Département d'informatique

Professeur Luc Lavoie
Président rapporteur
Département d'informatique

Sommaire

L'adoption massive de l'architecture orientée service expose les systèmes d'information à de nouvelles formes d'attaque informatique. Paradoxalement, les besoins en matière de sécurité ne cessent d'augmenter et de se raffiner, notamment dans les systèmes de la santé où l'accès aux données des patients doit être géré à travers une multitude d'acteurs.

Dans ce contexte, l'équipe du GRIL de l'Université Sherbrooke, en partenariat avec l'Université de Paris-Est Créteil, s'est penchée sur les problèmes d'expression et de gestion de politiques d'accès et a lancé le projet de recherche EB^{3}SEC. Ce dernier, basé sur les spécifications formelles, vise à assurer la sécurité des processus d'affaires des entreprises.

L'objectif de ce travail est de définir et d'évaluer une architecture de référence permettant d'intégrer le projet EB^{3}SEC à une infrastructure de sécurité moderne orientée service. À cette fin, nous avons montré qu'EB^{3}SEC pouvait être employé à l'intérieur d'un modèle d'autorisation purement centralisé en ayant recours aux transactions distribuées afin de synchroniser l'état d'EB^{3}SEC avec celui des données de l'entreprise. Nos tests préliminaires nous ont également permis de suggérer que l'impact d'EB^{3}SEC sur la performance d'une infrastructure de sécurité orientée service était négligeable par rapport à la lourdeur de cette dernière.

Remerciements

Je remercie, en premier lieu, mon directeur de recherche, monsieur le Professeur Marc Frappier, de m'avoir offert la chance d'effectuer ma maîtrise au sein de son équipe, pour m'avoir conseillé tout au long de mes travaux et pour avoir été patient et compréhensif avec les longueurs qu'ont connues ces derniers.

Je remercie également l'équipe du GRIL de l'accueil chaleureux qu'ils m'ont réservé, de leur enthousiasme et de leur disponibilité pour venir me rencontrer à Montréal.

Je remercie aussi ma conjointe de m'avoir encouragé tout au long de ma rédaction.

Finalement, je remercie la Banque Nationale du Canada de m'avoir permis d'accorder mon horaire de travail afin de me permettre de réaliser ma maîtrise.

Table des matières

Liste des abréviations

BPEL : *Business Process Language*

EB^{3}SEC : Évenement B 3 Securité

CISQ : *Consortium for IT Software Quality*

CORBA : *Common Object Request Broker Architecture*

ESB : *Enterprise Service Bus*

HIP : *History information point*

HTTP : *Hyper Text Transfer Protocol*

HUP : *History update point*

IDL : *Interface Definition Language*

JAX-WS : *Java API for XML Web Service*

JEE : *Java Enterprise Edition*

JMS : *Java Messaging Service*

JNDI : *Java Naming Directory Interface*

JSF : *Java Server Face*

JVM : *Java Virtual Machine*

OASIS : *Organisation for the Advancement of Structured Information Standards*

PAP : *Policy Administration Point*

PEP : *Policy Enforcement Point*

PIP : *Policy Information Point*

REST : *Representational State Transfer*

RST : *Request Security Token*

RSTR : *Request Security Token Response*

SAML : *Security Assertion Markup Language*

SAMLP : *Security Assertion Markup Language Protocol*

SGBD : Service de Gestion de Base de Données

SOA : *Service Oriented Architecture*

SOAP : *Simple Object Access Protocol*

SSO : *Single Sign On*

STS : *Security Token Service*

TI : Technologies de l'Information

WADL : *Web Application Description Language*

WS : *Web Service*

WS-T : *Web Service Trust*

WS-TX : *Web Service Transaction*

WSDL : *Web Service Description Language*

WSS : *Web Service Security*

WSS4J : *Web Service Security for Java*

XA : Protocole de transactions distribuées à deux phases

XACML : XML *Authorisation Markup Language*

XML : *Extensible Markup Language*

XML-DSIG : XML *Digital Signature*

XML-ENC : XML *Encryption*

Liste des tableaux

Liste des codes

Liste des figures

Introduction

Contexte

La conception des systèmes d'information d'entreprise tend de plus en plus vers les architectures orientées service (SOA) dans le but de faciliter les échanges avec les partenaires et d'augmenter la flexibilité des processus d'affaires. Ce style architectural a pour conséquence d'ouvrir les systèmes d'information aux attaques informatiques en plus de les rendre vulnérables à de nouvelles formes d'attaque. Parallèlement, la gestion des accès à l'information est un enjeu grandissant, non seulement en importance, mais aussi en complexité. L'importance du thème du droit à la vie privée est un exemple parmi tant d'autres de ce phénomène.

C'est dans ce contexte que l'équipe du GRIL, de l'Université de Sherbrooke, conjointement avec l'Université Paris-Est Créteil, développe le projet EB^{3}SEC, basé sur des spécifications formelles et s'intéressant à la sécurité fonctionnelle. Cette dernière est définie comme les règles de sécurité spécifiées dans les processus d'affaires des entreprises. Ce projet permettrait, entre autres, de contrôler l'accès à l'information dans une entreprise.

Problématique

Alors que les architectures orientées service nécessitent la revue des mécanismes traditionnels de sécurité et l'exploration de nouveaux concepts de sécurité, le projet EB^{3}SEC présente lui-même la particularité de baser, si nécessaire, les décisions d'autorisation sur l'historique des actions réalisées par les utilisateurs du système. Cette dernière caractéristique requiert la mise en place d'une stratégie pour assurer la cohérence et la synchronisation de l'historique avec l'état du système qu'il vise à protéger.

Objectifs

Ce travail comporte deux objectifs. Premièrement, il vise à identifier la meilleure architecture de sécurité orientée service pour supporter un contrôle d'accès complètement séparé des applications, c'est-à-dire une architecture où la logique d'affaires n'a pas à contenir de logique d'autorisation. Une telle architecture présente l'avantage d'éviter d'avoir à modifier plusieurs applications lorsqu'une politique d'autorisation est ajoutée et, ainsi, ces politiques peuvent être administrées de façon centralisée pour l'ensemble de l'entreprise, dans notre cas, à l'aide du projet EB^{3}SEC. Bien entendu, l'architecture de sécurité proposée doit, d'abord, respecter les principes d'architectures orientées service, puis, assurer la confidentialité et l'intégrité des transactions, éviter au maximum de dégrader la disponibilité et l'accessibilité des systèmes et promouvoir leur évolutivité.

Deuxièmement, nous voulons proposer, dans le cadre de l'architecture précédemment identifiée, un mécanisme pour assurer la synchronisation entre l'état du contrôleur d'accès d'EB^{3}SEC et celui de l'état du système contrôlé. Ce mécanisme est requis afin que l'historique des actions des utilisateurs dans le système, consulté au besoin par EB^{3}SEC pour prendre des décisions d'autorisation, reflète exactement l'état actuel des processus de l'entreprise. Encore une fois, ce mécanisme doit être invisible aux développeurs d'applications et aisé à maintenir.

Méthodologie

Tout au long d'ouvrage, nous utilisons une approche non stricte pour présenter les différents concepts. Dans un premier temps, une architecture de référence, qui supporte un mécanisme de synchronisation entre l'historique des actions des utilisateurs et l'état du système au sein d'une architecture SOA, est proposée. Pour ce faire, les concepts de sécurité et les modèles d'autorisation ayant trait aux SOA sont d'abord passés en revue, puis les contraintes liées au modèle d'autorisation d'EB3SEC sont étudiées. Dans un deuxième temps, une brève implémentation du design résultant est réalisée afin d'évaluer sommairement notre travail et afin d'offrir un environnement de test à l'équipe du GRIL.

Résultats

Au travers ce travail, nous montrons que le projet EB^{3}SEC peut être intégré à l'intérieur d'une infrastructure moderne de sécurité d'architectures orientées service, mais que cette dernière doit être basée sur un modèle d'autorisation purement centralisé, notamment en raison de la complexité que représenterait un déploiement de EB^{3}SEC sur chacun des services de l'entreprise, de la particularité d'EB^{3}SEC de baser certaines de ses décisions sur l'historique des actions des utilisateurs et de la fine granularité des politiques d'accès qu'offre EB^{3}SEC. De plus, nous établissons que la synchronisation systématique d'EB^{3}SEC avec l'état du système nécessite le recours aux transactions distribuées.

Nous montrons également que, contrairement aux promesses faites par les fournisseurs de produits et d'outils SOA, l'implémentation et la sécurisation d'une architecture orientée service sont des tâches complexes et ardues qui aboutissent à un système peu performant. Ce dernier point est démontré lors de l'évaluation de notre preuve de concept. Nous calculons un temps moyen de 5 secondes pour sécuriser une requête lorsque le nombre de ces requêtes concurrentes atteint 40.

Finalement, nous montrons que l'utilisation de transactions distribuées impacte la performance du système, mais que cet impact n'est pas considérable dans une infrastructure moderne de sécurité d'architectures orientées service.

Structure du mémoire

Le chapitre 1 introduit les architectures orientées service en présentant les technologies employées pour les implémenter et les raisons qui motivent leur utilisation. Le chapitre 2 explique les enjeux de sécurité liés à ces architectures et passe en revue les différents concepts et mécanismes impliqués dans leur sécurisation. Le chapitre 3 étudie le projet EB^{3}SEC et propose une architecture de référence pour l'intégrer à une infrastructure de sécurité moderne SOA. Finalement, le chapitre 4 propose une preuve de concept afin d'évaluer à haut niveau la faisabilité et la performance de cette architecture de référence.

Chapitre 1 SOA

Les architectures orientées service, traduction française de SOA (*Service Oriented Architectures)*, sont un style d'architecture de solutions de technologies de l'information (TI) d'entreprise. Bien qu'elles datent du début des années 80, elles n'ont connu un essor important de leur popularité qu'au cours des dernières années. Ceci à un point tel qu'en 2007, Gartner prédisait que les SOA seraient utilisées dans plus de 50% des nouvelles applications opérationnelles critiques d'entreprise [6]. Micheal Rosen, directeur éditorialiste de la *SOA Institute,* qualifie les SOA comme «l'état actuel de l'art» dans l'architecture des applications TI [17]. Aujourd'hui, les chefs de file dans le domaine des solutions d'affaires TI, tels Microsoft, IBM, Oracle et SAP, ont tous adopté les SOA et y investissent des milliards de dollars en recherche et en développement. Le terme SOA est désormais omniprésent dans les stratégies de marketing des logiciels d'entreprise.

Ce chapitre propose, dans un premier temps, une définition sommaire des SOA en présentant ses caractéristiques, ainsi que les bénéfices recherchés par une implantation de cette architecture. Dans un deuxième temps, les services web et les services REST, qui constituent les technologies d'implémentation les plus populaires des SOA et sur lesquels se base la majorité des exemples de cet ouvrage, sont explorés.

1.1 Définition des SOA

Malgré l'engouement qu'elles suscitent, les SOA demeurent un concept abstrait et leur définition varie d'un auteur à l'autre. Dans leur article portant sur les répercussions des SOA sur le contenu des cours d'informatique en Australie, Teo et ses collaborateurs définissent les SOA en reprenant la définition de Channabasavaiah et al. : « SOA est une architecture d'application dans laquelle toutes les fonctions sont définies comme des

services indépendants possédant des interfaces bien définies qui peuvent être appelées dans des séquences définies pour former des processus d'entreprises » [8]. Le groupe OASIS, dans son modèle de référence pour l'architecture orientée service, défini un service en trois points : « Les services sont le mécanisme qui lie les besoins aux capacités. Ils combinent les trois idées suivantes : la capacité d'effectuer un travail pour un autre, la spécification du travail offert pour un autre et l'offre d'exécuter un travail pour un autre » [13].

Les services sont hiérarchisés selon la quantité de traitement d'affaires qu'ils réalisent [17]. Le niveau supérieur comprend les services qui réalisent un processus d'affaires alors que le niveau le plus bas comprend les services utilitaires et les services qui agissent comme façade pour exposer de façon standard les fonctionnalités offertes par les systèmes existants. La Figure 1, où les services sont représentés par des cercles, illustre cette hiérarchisation par un exemple simple où un processus d'achat en ligne est implémenté à l'aide d'une couche d'intégration SOA. Une fois les données de l'utilisateur récupérées par l'application web, celle-ci démarre le processus en appelant le service « Achat » qui orchestre le déroulement du processus à l'aide des services des couches inférieures.

Nous notons que SOA ne spécifie pas comment les services accèdent aux ressources du système d'information. Chaque service est analysé, conceptualisé et implémenté individuellement en suivant les bonnes pratiques déjà en place de l'organisation, telle la séparation du code en différentes couches.

1.2 Motivations

Les architectures orientées service ont pour but premier d'augmenter l'agilité des systèmes d'information d'une entreprise, c'est-à-dire d'augmenter leur capacité à s'adapter rapidement aux changements des besoins du marché. Cette agilité se traduit en processus d'affaires flexibles et elle est réalisable grâce au potentiel intégrateur des SOA. Cette intégration est assurée par deux caractéristiques.

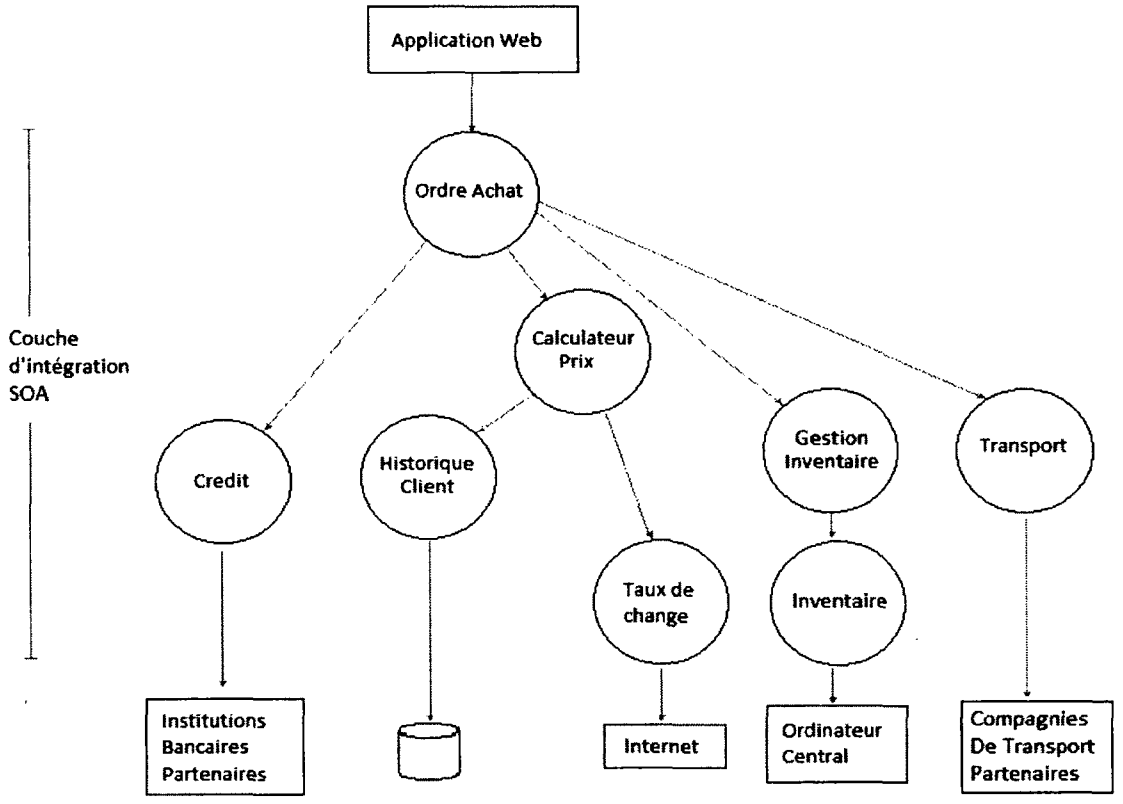

Figure 1 – Exemple d'une solution SOA d'entreprise

Premièrement, les SOA brisent les silos applicatifs en regroupant la logique d'affaires dispersée à travers les applications et en la représentant sous forme d'unités fonctionnelles réutilisables. Par exemple, si deux applications web offrent chacune la possibilité aux clients d'acheter un produit différent, plutôt que de coder à deux endroits la logique pour valider le solde d'un compte, les SOA proposent de regrouper cette fonctionnalité sous un seul service. Une fois généralisé, ce principe permet d'éviter la duplication des données et, du coup, assure aux clients une vue uniforme des données au travers l'entreprise et facilite leur maintenance.

Deuxièmement, dans les SOA, les services exposent leurs fonctionnalités sous forme d'interface et communiquent de façon standard et indépendamment de leur technologie d'implémentation. Ces caractéristiques augmentent l'interopérabilité entre les systèmes

d'information hétérogènes, aussi bien à l'interne d'une entreprise qu'avec ceux de ses partenaires. L'interopérabilité est l'habileté d'un système à travailler avec un autre sans difficulté.

1.3 Implémentation

Les architectures orientées service ont vu le jour avant l'invention de leur nom[1]. Par exemple, dès 1991, la première version du standard CORBA, « *Common Object Request Broker Architecture* », était publiée [2]. Le CORBA normalise les appels faits entre des objets possiblement distribués sur un réseau et possiblement écrit dans des langages différents. Pour ce faire le standard IDL « *Interface Definition Language* » est employé pour décrire comment accéder un objet. CORBA spécifie ensuite comment passer du contrat IDL à chaque langage supporté. Cette définition correspond à celle des SOA, où les objets sont les services.

Bien que CORBA soit encore fréquemment utilisé, les SOA n'ont connu un essor important de leur popularité que grâce aux technologies des services web. À ce jour, les services web constituent l'implémentation de références des SOA. Pour cette raison, nous utilisons principalement ce type de service dans les exemples de cet ouvrage. Selon le groupe OASIS, "Un service web est un composant logiciel décrit par WSDL qui peut être accédé par des protocoles réseaux standards tels SOAP et HTTP ». Le langage WSDL, acronyme de « *Web Service Description Language* », est un langage basé sur XML qui permet de décrire un service. Il répond à trois questions : Qu'est-ce que fait le service? Comment utiliser le service? Et où est situé le service? [3]. Ainsi, l'accès à un document WSDL est suffisant pour être en mesure d'interagir avec le service décrit par ce dernier. La spécification WSDL a été élaborée et est maintenue par le groupe W3C. Elle constitue un standard dans l'industrie en raison de sa rigueur et de son indépendance à la plateforme technologique utilisée. Plusieurs outils permettent la génération automatique de code pour communiquer avec un service simplement à l'aide de son

[1] Selon Gartner, le terme SOA aurait été décrit pour la première fois dans un papier publié par Gartner en 1996 [9]

document WSDL. Typiquement, dans une entreprise, ces documents sont regroupés dans un ou plusieurs registres centraux pouvant être consultés par les développeurs et, possiblement, par les entreprises partenaires.

Nous divisons les services web en deux catégories : les services communiquant à l'aide du protocole SOAP (*Simple Object Access Protocol*) et les services REST (*REpresentational State Transfer*). Les deux sections suivantes présentent et comparent ces approches.

1.3.1 Services SOAP

SOAP est un protocole de communication basé sur XML et maintenu par le groupe W3C. La Figure 2 présente, de façon hiérarchique, les principaux concepts ayant trait à ce protocole. D'abord, le bloc « transport » représente le protocole applicatif de communication. Tout protocole non-propriétaire pouvant transporter un document XML peut être employé à ce niveau. Dans la quasi-totalité des cas, les protocoles HTTP et HTTPS sont retenus en raison de leur simplicité et de leur support déjà présent dans les entreprises. Toutefois, il arrive que des protocoles asynchrones ou propriétaires, tel JMS, soient employés. Le bloc XML indique que le message SOAP est encodé à l'aide du langage XML. Ceci a une grande importance, car le langage XML utilise les schémas XSD qui permettent de définir des métadonnées (bloc suivant) pour décrire les types d'objets échangés dans les messages. Le bloc « messagerie » représente, du point de vue des affaires, le document échangé lors de la communication. Finalement, les blocs « fiabilité », « sécurité » et « transaction » représentent des aspects pouvant être implémentés à l'aide des en-têtes des messages SOAP. Ces en-têtes constituent un mécanisme d'extension du protocole et sont représentés à la Figure 3 qui montre la structure d'un message SOAP. Le nombre d'en-têtes n'est pas fixe et ceux-ci peuvent être de n'importe quelle nature, pourvu qu'ils soient accompagnés de métadonnées. Si le récepteur ne reconnaît pas un en-tête, il n'est pas forcé de le traiter[2].

[2] Ceci n'est pas tout à fait exact. Selon la spécification «*SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)* », si un en-tête est accompagné de l'attribut

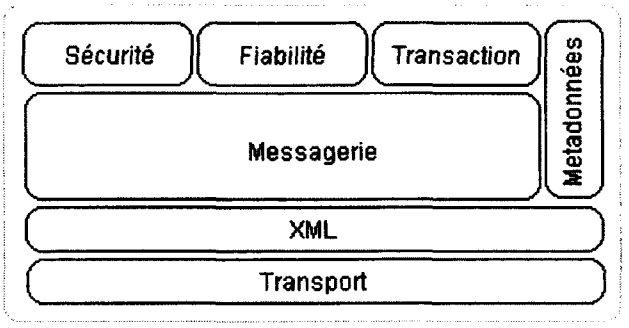

Figure 2 – Concepts du protocole SOAP

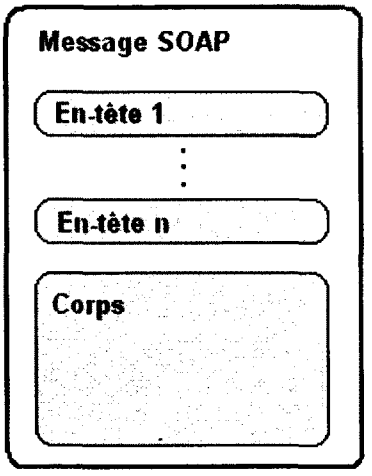

Figure 3 – Structure d'un message SOAP

La flexibilité offerte par les en-têtes des messages SOAP pose cependant un problème au niveau de l'interaction avec les services d'entreprises partenaires, car les deux parties doivent s'entendre sur les formats d'en-têtes utilisés. Par exemple, les deux parties doivent s'entendre sur le contenu de l'en-tête qui permet de communiquer l'identité d'un client. Ce problème est majeur, car un des buts premiers des SOA est de favoriser l'interopérabilité des systèmes et donc d'éviter qu'un processus de développement additionnel soit nécessaire pour chaque cas d'intégration. De nombreux standards ont été rédigés par différents groupes afin de pallier à ce problème, notamment par les groupes OASIS et W3C. Parmi ces standards, on compte des spécifications de chiffrement, de

---

« mustUnderstand="true" », un service doit traiter cet en-tête avant de traiter le corps de la requête. S'il n'est pas capable, il doit lancer une exception.

signature digitale, d'attachement de jetons de sécurité, etc. Le choix du ou des standards supportés par un service récepteur peut être communiqué à l'intérieur même du fichier WSDL à l'aide du standard WS-Policy [23], présenté à la section 2.2.3.

Nous présentons trois avantages du protocole SOAP. Premièrement, l'utilisation des schémas XML permet de décrire formellement la structure des opérations offertes par un service ainsi que le type de leurs paramètres. Ainsi, il est possible pour un consommateur de service de valider automatiquement son message avant de l'envoyer au récepteur. Deuxièmement, le protocole SOAP offre plusieurs points d'extension en ne restreignant pas les types d'éléments XML pouvant apparaître dans les en-têtes. Cette caractéristique permet l'implémentation de concepts complexes tels que les transactions et la sécurité. Troisièmement, et dernièrement, SOAP est indépendant du protocole de transport utilisé. En plus de permettre la réutilisation de l'infrastructure en place, ceci offre la possibilité d'une implémentation synchrone ou asynchrone des services.

En revanche, le langage XML est verbeux et coûteux à traiter en termes de performance. Il ouvre même la porte aux attaques de négation de service, notamment en permettant d'inclure des références à des documents externes sur internet. Un autre argument en défaveur des services SOAP est la difficulté à les tester. Malgré la disponibilité d'outils de tests unitaires graphiques gratuits, par exemple *SoapUI*, le support pour tester rapidement des cas d'utilisations complexes, tels que l'insertion d'un jeton de sécurité à l'intérieur d'un en-tête, est limité. Finalement, les mécanismes d'extension et autres standards sont tellement nombreux que le risque qu'un des partenaires n'implémente que partiellement une des spécifications ou simplement que son interprétation soit erronée est élevé. Du coup, le principe fondamental d'interopérabilité est mis en mal.

1.3.2 Services REST

REST est un style d'architecture[3] pour les systèmes hypermédias distribués, défini par Roy Fielding en 2000 dans sa thèse de doctorat, auquel se conforme l'architecture

[3] Dans ce même ouvrage, un style d'architecture est défini comme un ensemble de contraintes architecturales coordonnées.

HTTP/1.1 [5]. Selon Wikipédia, avec REST « une communication entre deux composants est considérée comme un transfert de la représentation de l'état d'une ressource, où une ressource est une information pouvant être nommée et variant dans le temps et où une représentation décrit une ressource dans un format donné, par exemple HTML » [16]. Une architecture REST comprend les contraintes suivantes : les contraintes associées à l'architecture client-serveur, les communications entre le client et le serveur ne doivent pas avoir d'état, les données à l'intérieur d'une réponse doivent être implicitement ou explicitement marquées comme pouvant être conservées en mémoire cache, les interfaces de communication entre les composants doivent être uniforme, les contraintes liées aux systèmes implémentés en couches et, optionnellement, les serveurs doivent pouvoir étendre les fonctionnalités des clients en permettant à ceux-ci de télécharger du code qu'ils peuvent ensuite exécuter localement.

La principale différence entre REST et SOAP vient du fait que le deuxième n'accède pas directement aux ressources (ou leur représentation), mais utilise plutôt des opérations (ou procédures) pour le faire. Ainsi, ces opérations décrites à l'aide de SOAP, n'utilisent le protocole HTTP qu'en tant qu'enveloppe statique de transport. En revanche, un service REST exploite pleinement le protocole HTTP. Quelques exemples de requêtes REST sont présentés à l'annexe A. Les services REST représentent généralement leurs ressources sous forme de document XML, mais rien ne les empêche d'utiliser d'autres représentations, pourvu que le type de ces dernières soit mentionné en en-tête.

REST ne pose pas de contraintes au niveau de l'implémentation, mais encourage la réutilisation des infrastructures en place, ce qui résulte, dans la quasi-totalité des cas, à une implémentation à l'aide du protocole HTTP. Or, il n'existe pas de spécification pour décrire un service REST, sinon qu'il doit se conformer aux principes des architectures REST. Ainsi, dans ce travail, nous considérons qu'un service REST est nécessairement implémenté à l'aide du protocole HTTP.

Les services REST offrent l'avantage d'être simple et rapide à développer, car ils sont basés sur des protocoles simples, maitrisés et, souvent, déjà en place. De plus, leur utilisation de mots clés tels « GET » et « PUT », afin d'indiquer si la requête implique une mise à jour de l'état du système, facilite leur filtrage au niveau des murs pare-feux d'une entreprise. Finalement, il est aisé de les tester puisqu'un simple fureteur est souvent suffisant.

Les services REST sont généralement employés pour les scénarios simples qui n'impliquent pas l'appel de plusieurs services en chaîne, et donc, qui ne requièrent pas de mécanismes de sécurité ou de transaction évolués. Au moment de la rédaction, les standards permettant de décrire l'interface d'un service REST, soient WSDL 2.0 et WADL, sont, dans le premier cas, encore très peu supportés et, dans le deuxième cas, non officiels. L'intégration d'un service de ce type, notamment pour prédire les types de données échangeables, demande un effort supplémentaire par rapport aux services SOAP. Pour ces raisons, les services REST sont souvent des services utilitaires à partir desquelles transigent des informations simples et ne réalisent que peu de logique d'affaire.

1.3.3 Conclusion SOAP et REST

Les services SOAP et REST présentent tous les deux des avantages et des désavantages et peuvent être employés de façon complémentaire dans une architecture orientée service. Ainsi, pour des cas d'utilisations complexes, tel l'implémentation d'un mécanisme de transactions distribuées entre les services, ou encore simplement pour assurer que les services exposés sont conformes à des standards officiels, les services SOAP sont préférables. Cependant, le développement de petits services utilitaires participant à des cas d'utilisation simple est plus rapide à effectuer et à tester avec les technologies REST. Dans cet ouvrage, nous utilisons les services SOAP afin de tirer profit des nombreux standards relatifs à la sécurité ayant déjà été écrits pour ce protocole.

Chapitre 2 Sécurité des SOA

Ce chapitre explore, toujours avec une approche non stricte, les différents concepts impliqués dans la sécurisation des infrastructures SOA. Il débute par la comparaison entre les vulnérabilités présentes dans les architectures web traditionnelles par rapport à celles présentes dans les SOA, puis les concepts de ce chapitre et leurs interrelations sont brièvement survolés. Ensuite, chacun de ces concepts, c'est-à-dire la sécurité XML, les filtres XML, les services de jetons de sécurité, la propagation d'identité et les architectures d'autorisation, sont détaillés afin de décrire les problèmes qu'ils résolvent ainsi que leurs avantages et inconvénients.

2.1 Nouveaux besoins des SOA en matière de sécurité

Les SOA sont plus à risque en matière de sécurité comparativement aux applications web classiques. Par application web classique, nous entendons une application web HTTP soutenue par une base de données. Afin de soutenir cette affirmation, cette section débute par la comparaison des deux architectures au point de vue de la sécurité, puis, détaille les vulnérabilités des SOA à l'aide de trois cas d'utilisation.

2.1.1 Comparaison des applications classiques web avec SOA

Les solutions de sécurité traditionnelles tirent profit des barrières présentes naturellement dans les systèmes d'information, telles les applications développées en silo, l'hétérogénéité des plateformes technologiques et les frontières des entreprises. Par exemple, dans la Figure 4, où une application web expose directement les fonctionnalités A et B (par exemple l'achat d'un ordre et la visualisation d'un ordre), si nous supposons que le canal de communication est sécurisé, que le serveur applicatif est en mesure d'authentifier et d'autoriser les requêtes et que ce dernier est digne de confiance pour voir

et traiter les données personnelles des clients, alors il ne suffit que de sécuriser le point d'entrée du serveur puisque les requêtes aux deux fonctionnalités passent nécessairement par ce point. Cet exemple illustre une frontière naturellement créée par le développement d'application en silo. Pour réutiliser une de ces fonctionnalités à l'intérieur d'une autre application, il faut soit placer son code dans une bibliothèque et maintenir cette dernière à deux endroits, ou soit créer une solution particulière d'intégration entre les deux applications.

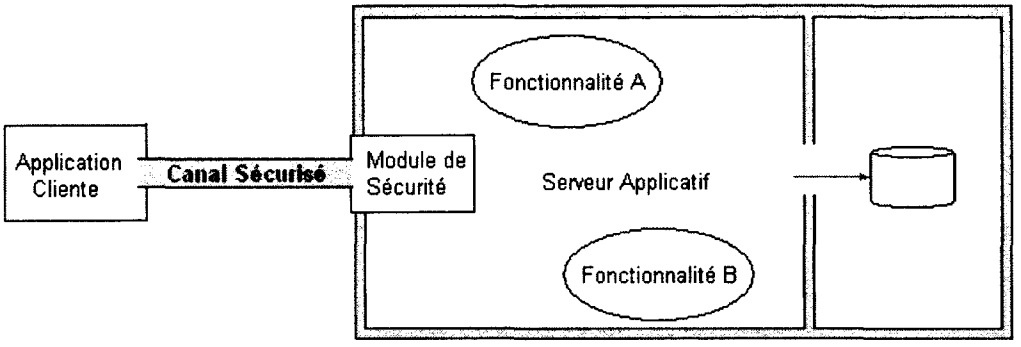

Figure 4 – Sécurité d'une application web traditionnelle (inspirée de [7])

Une fois transposée dans une architecture SOA, l'application classique de la Figure 4 pourrait ressembler à celle de la Figure 5. On note que les barrières ont disparu. Le système expose ses fonctionnalités sous forme de services, possiblement répartis sur plusieurs machines physiques, directement accessibles depuis les applications clientes et depuis les services de l'entreprise ou d'un partenaire. Même le fossé technologique entre les différentes générations de systèmes d'informations, ici illustré par un AS400, est réduit par le « service Z » qui agit en tant que façade en exposant de façon standardisée des fonctionnalités d'un serveur AS400.

2.1.2 Détails des nouveaux besoins

Afin d'illustrer les vulnérabilités des SOA en matière de sécurité, nous présentons trois cas d'utilisation pour l'appel d'un service, illustrés à la Figure 6. Les services sont représentés par les cercles et les utilisateurs par les rectangles.

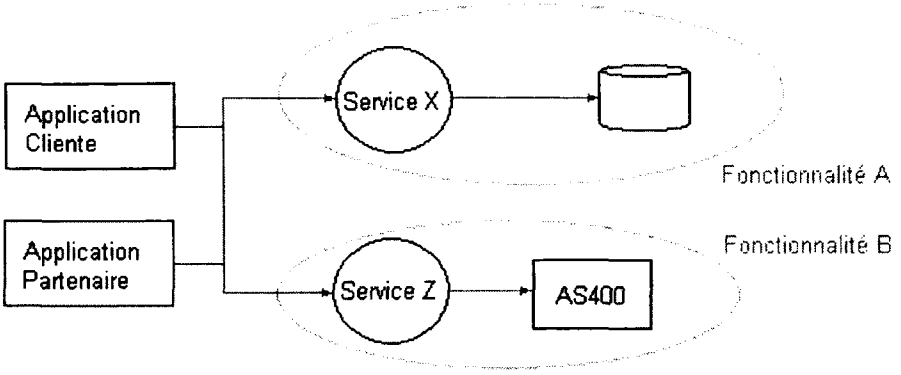

Figure 5 – Application web de la Figure 4 transposée dans une SOA

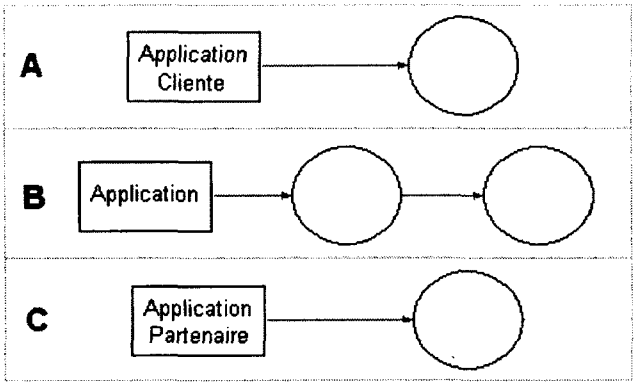

Figure 6 – Trois scénarios d'appel de service

Le premier cas, « A », représente le cas trivial où un utilisateur de l'entreprise appelle directement un service. La sécurisation de ce cas est pratiquement identique à celui des applications web classiques. L'utilisateur inclut son identité avec la requête et chiffre cette dernière. De son côté, le service déchiffre la requête, extrait l'identité de l'utilisateur, authentifie cette dernière et l'autorise.

Le scénario « B » illustre une composition de services, c'est-à-dire qu'un service est appelé par un autre service ou nœud SOA. Dans ce cas, la problématique est à deux niveaux. D'abord, comment le destinataire final authentifie-t-il et autorise-t-il la requête? Doit-il se fier à la sécurité appliquée par le premier service ? Si oui, comment peut-il être sûr que la requête provient effectivement de ce premier service ? Sinon, doit-il vérifier l'autorisation de la requête par rapport au service intermédiaire ou par rapport à l'utilisateur, initiateur de la requête ? Ensuite, au niveau de la confidentialité, comment

l'utilisateur peut-il s'assurer que les informations personnelles qu'il envoie dans sa requête ne sont visibles que par le deuxième service alors que le premier service à besoin de certaines informations de la requête afin la diriger vers le deuxième service ?
Le troisième scénario, « C », représente la caractéristique que présentent les SOA à faciliter les échanges avec les partenaires de l'entreprise. Dans ce cas, le problème est de savoir comment authentifier et autoriser les utilisateurs des partenaires, car ces derniers ne sont pas inclus dans le registre des utilisateurs de l'entreprise.

Finalement, indépendamment des scénarios d'utilisation des services, la sécurité des SOA est complexe à maintenir. En effet, les politiques en matière de sécurité peuvent varier d'un service à l'autre en fonction des données traitées, des groupes d'utilisateurs, etc. Ainsi, la sécurité de chaque service doit pouvoir être configurée individuellement.

2.2 Aperçu des concepts de sécurité SOA

Les solutions pour sécuriser les SOA sont nombreuses et dépendent des besoins de l'entreprise ainsi que du niveau de sécurité recherché. Cependant, elles ne réinventent pas la roue : pour la plupart, elles réutilisent les mécanismes classiques de défense ou bien elles étendent ceux-ci. Cette section débute par un survol des liens entre les différents concepts de la sécurité SOA, puis se poursuit par un exemple qui illustre l'interaction entre les composants.

2.2.1 Relations entre les concepts de sécurité SOA

La Figure 7 illustre les liens entre les différents concepts de la sécurité des SOA. Chacun de ces concepts est repris dans les sections suivantes de ce chapitre.

D'abord, les services web SOAP introduisent le concept de sécurité au niveau du message. Ce concept propose notamment d'attacher à l'intérieur des en-têtes SOAP des informations sur l'identité de l'utilisateur requérant ainsi que des informations de chiffrement afin de pouvoir assurer la confidentialité de certaines parties du message.

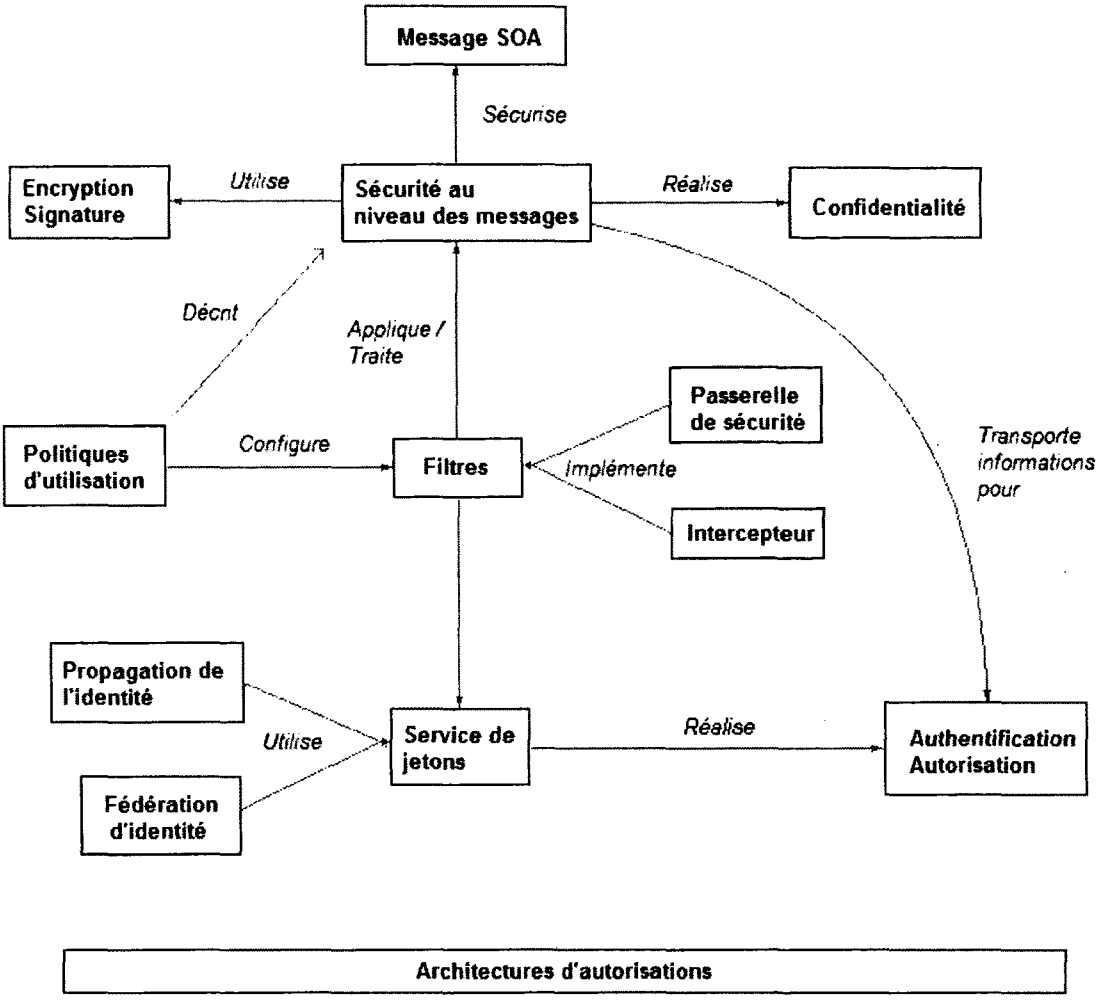

Figure 7 – Liens entre les concepts de sécurité dans les SOA

L'application et le traitement de la sécurité au niveau des messages sont réalisés par le filtrage des requêtes. Nous divisons les filtres en deux catégories, soit les passerelles de sécurité et les intercepteurs.

Les politiques d'utilisation décrivent comment un service peut être utilisé, notamment en précisant les mécanismes de sécurité qu'il supporte. Cette approche peut également être utilisée pour configurer automatiquement les filtres de sécurité au moment de leur conception ou de leur exécution.

L'identité d'un utilisateur peut être représentée à l'aide d'un jeton de sécurité, tel une chaîne de caractères unique, une combinaison nom d'utilisateur / mot de passe, une assertion signée, etc. Un service central de jetons de sécurité est alors employé, dans un premier temps, pour gérer le registre d'utilisateurs connectés au système et, dans un deuxième temps, pour créer, valider ou échanger des jetons pour ces utilisateurs connectés. Ce design offre une solution de connexion unique aux travers les différents systèmes hétérogènes d'une entreprise. Ce mécanisme repose sur une relation de confiance préalablement établie entre le service de jetons et les applications, tel le partage d'une clé. Les services de jetons de sécurité facilitent les fédérations d'identité entre les entreprises partenaires en établissant des cercles de confiance. Une fédération d'identités est le regroupement des différents comptes d'un même utilisateur auprès de plusieurs entreprises dans le but qu'il n'ait à s'authentifier qu'une seule fois pour avoir accès du même coup aux applications des deux entreprises partenaires.

Finalement, les différentes architectures d'autorisation (ou modèles d'autorisation) tirent profit de tous ces concepts.

2.2.2 Exemple d'interactions entre les composants de sécurité SOA

Le diagramme d'activité de la Figure 8 présente un exemple simple où un service A désire appeler un service B. Ces deux services possèdent chacun un intercepteur, préalablement configuré, qui a pour rôle de sécuriser les communications entrantes et sortantes. Un intercepteur, ou agent, est un objet auquel le service n'est pas couplé et dont il ignore même l'existence et qui, idéalement, peut être réutilisé devant et derrière d'autres services. Le flot d'exécution débute par la création et l'envoie du message par le service A vers le service B. Le message est intercepté par l'intercepteur A, un objet auquel le service n'est pas couplé et dont il ignore même l'existence et qui, idéalement, peut être déployé autour d'autres services. Cet intercepteur demande au fournisseur d'identité de lui fournir un jeton représentant l'identité de l'utilisateur et conforme aux politiques de sécurité du service B. Le service de jeton de sécurité authentifie le demandeur et vérifie s'il est autorisé à accéder au service B. Une fois ces deux opérations complétées, un jeton est créé puis retourné à l'intercepteur A. Celui-ci insère

le jeton à l'intérieur du message original puis le communique au service B à l'aide du protocole sécurisé HTTPS. Avant d'atteindre le service B, le message est une fois de plus intercepté, cette fois par l'intercepteur B. Celui-ci extrait d'abord le jeton du message, puis, selon le type de jeton, le valide lui-même ou demande au service de jetons de sécurité de le valider. Une fois le jeton validé, l'intercepteur relaie la requête au service B afin qu'elle soit traitée.

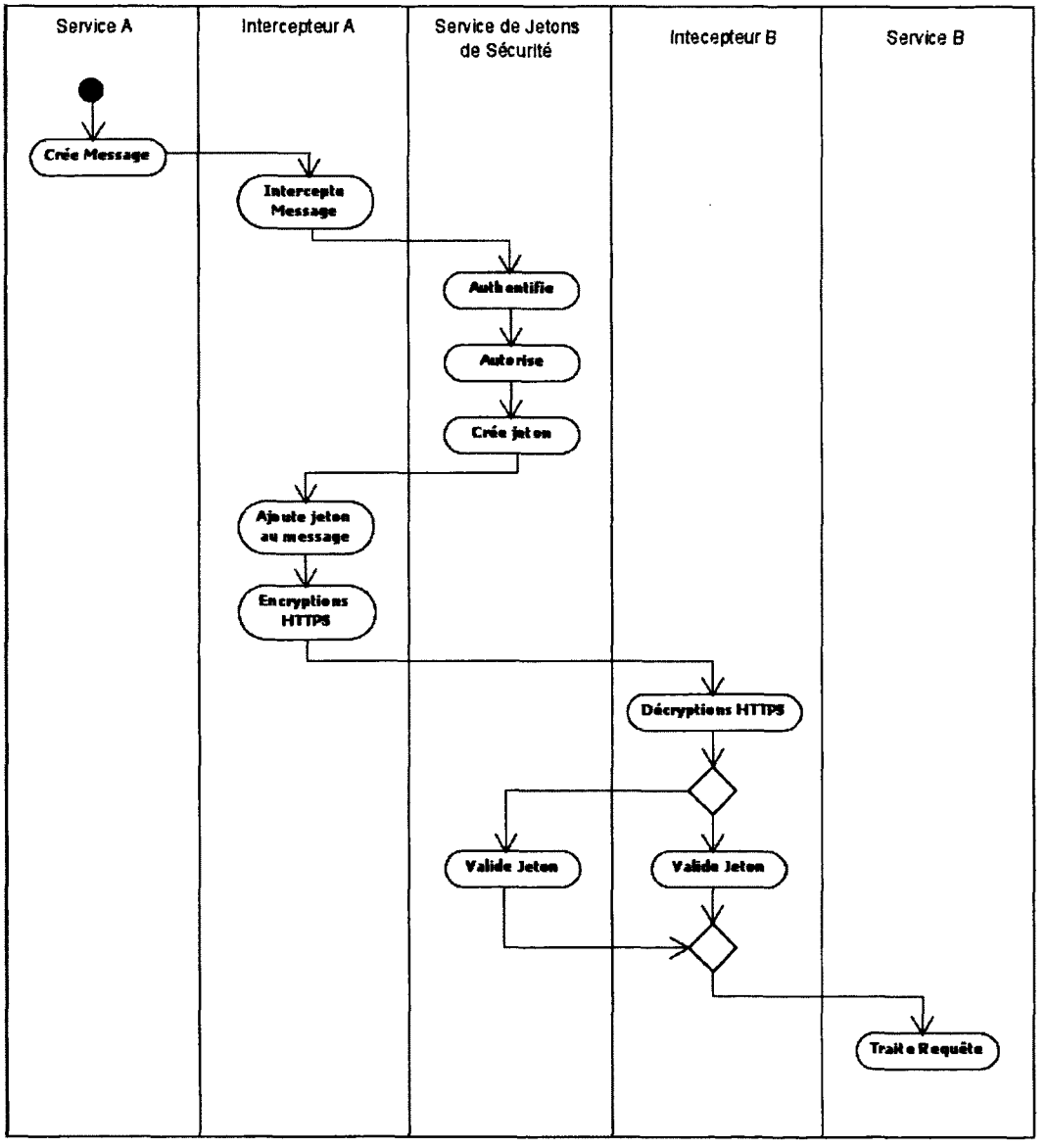

Figure 8 – Exemple d'un diagramme d'activité des composants de sécurité SOA

Nous notons encore une fois que cet exemple ne sert qu'à donner une idée générale de la sécurisation d'un système SOA et n'est qu'une possibilité parmi une infinité. Les

solutions varient selon les cas d'utilisation et selon les besoins. De plus, cet exemple ne contient pas la plupart des concepts présentés à la section précédente.

2.3 Sécurité au niveau des messages

La sécurité au niveau des messages est un outil pour sécuriser les SOA. Elle offre, entre autres, une solution aux sous-problèmes présentés par le cas d'utilisation « B » de la Figure 6, soit la propagation d'identité et la confidentialité lorsqu'une requête traverse plus d'un service. Le mécanisme consiste à l'inclusion d'informations de sécurité à l'intérieur même du message applicatif. Ainsi, la propagation d'identité est réalisée en incluant un jeton d'identification dans le message alors que la confidentialité des sections sensibles du message est réalisée à l'aide de métadonnées qui décrivent un potentiel chiffrement partiel du message (les sections chiffrées sont propagées d'un message à l'autre jusqu'à ce qu'elles atteignent le service qui sait les déchiffrer).

Le protocole SOAP tire profit des standards « *XML Digital Signature* » et « *XML Encryption* », du groupe W3C, qui spécifient respectivement une méthode pour signer numériquement des portions de documents XML et une méthode pour chiffrer des portions de document XML [24, 25]. Le groupe OASIS maintient la spécification « *WS-Security* » qui propose un en-tête SOAP pour regrouper et structurer ces informations de sécurité [12].

Le recours à la sécurité au niveau des messages ne garantit pas la sécurité des communications. Les informations de sécurité à inclure dans le message et les éléments à chiffrer ou à signer doivent être étudiées au préalable. Pour cette raison, le groupe WS-I, « *Web Service Interoperability Organization* », définit des profils de sécurité, les « *Web Service Security Profil*» 1.0 et 1.1, qui prescrivent des applications sécuritaires des standards W3C et OASIS mentionnés plus haut [20].

Au niveau de la performance, la sécurité au niveau des messages est coûteuse. D'abord, l'insertion d'un élément à l'intérieur d'un document XML nécessite généralement la

représentation et la manipulation de ce dernier sous la forme d'un arbre en mémoire. Dans le cas où les requêtes sont nombreuses et les documents XML volumineux, l'espace mémoire nécessité pour ces opérations devient rapidement important. De plus, le standard *XML-DSIG* considère que la signature d'un document XML doit être réalisée sur sa sémantique plutôt que strictement sur ses octets afin d'obtenir la même signature même si la syntaxe du document signé change d'un service à l'autre. Pour ce faire, une étape supplémentaire est nécessaire pour transformer le document dans une forme normale que la spécification qualifie de canonique. *XML-DSIG* propose donc un algorithme de « *canonicalization* », abrévié c14n, qui nécessite de nombreuses manipulations XML. Finalement, *XML-DSIG* permet que les informations de signatures (algorithmes, clés et éléments signés) soient placées après la signature elle-même ce qui implique, dans ce cas, que la signature ne peut être validée au fur et à mesure qu'elle est lue.

2.4 Filtres

Comme dans la majorité des systèmes distribués, la sécurité des architectures orientées service repose sur le filtrage des requêtes et des réponses échangées entre les clients et les fournisseurs de services. Les filtres de sécurité des solutions SOA étendent les mécanismes classiques de protection des applications web et peuvent être regroupés en deux catégories : les passerelles de sécurité et les intercepteurs. Cette section compare les deux approches.

2.4.1 Passerelles de sécurité

Les passerelles de sécurité, aussi appelées pare-feu XML et proxy XML dans les environnements SOAP, sont des composantes matérielles ou logicielles placées à l'entrée d'un réseau. Ce design, illustré à la Figure 9, offre une première couche de sécurité aux services et empêche physiquement les requêtes de l'extérieur d'atteindre les services.

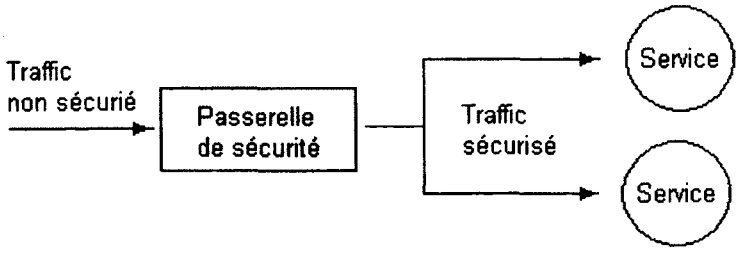

Figure 9 – Passerelle de sécurité [17]

Les passerelles de sécurité permettent d'alléger la charge de travail des services en effectuant à leur place les opérations redondantes et coûteuses en termes de performance, telles que le déchiffrement du message et la validation de sa structure. Ainsi, les services peuvent s'attendre à recevoir des requêtes d'un certain niveau de qualité. L'authentification et l'autorisation des requêtes destinées aux services peuvent aussi être réalisées par les passerelles de sécurité. Le caractère « point d'entrée unique» de ce modèle facilite la configuration et la gestion de ces opérations et, du coup, le support aux nombreux protocoles et jetons de sécurité requis par la nature hétérogène des SOA est amélioré. Nous notons que ce modèle peut aussi être utilisé à l'inverse, c'est-à-dire comme point de sortie unique. Dans ce cas, la passerelle sécurise les messages sortants et, encore une fois, libère les services de cette tâche.

L'utilisation exclusive de ce modèle de sécurité comporte cependant des désavantages. Dans un premier temps, les services sont laissés sans protection. Si un client, externe ou interne, parvient à contourner la passerelle, il contourne d'un seul coup tous les mécanismes de sécurité. Nous notons toutefois que ce risque peut être mitigé en restreignant l'accès aux services à quelques nœuds du réseau (avec un mur pare-feu par exemple) ou en imposant une authentification mutuelle entre la passerelle et les services. Dans un deuxième temps, les services ne gérant pas la sécurité, un contexte de sécurité ne peut leur être transmis pour, par exemple, propager l'identité de l'utilisateur à l'origine de la requête.

En pratique, les passerelles sont utilisées conjointement avec les intercepteurs et elles instancient le patron de conception SOA « service de garde de périmètre » [4], c'est-à-dire qu'elles sont situées devant un réseau privé de services, typiquement dans une zone démilitarisée, comme l'illustre la Figure 10. De plus, leur charge étant très élevée, des processeurs spécialisés en chiffrement / déchiffrement sont généralement employés. Les implémentations matérielles ont la cote sur les implémentations logicielles. La suite de produits *WebSphere Datapower* de IBM constitue la solution la plus populaire sur le marché à l'heure actuelle.

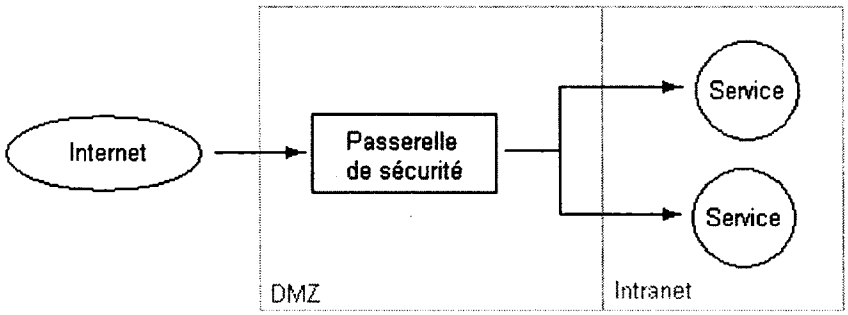

Figure 10 – Implémentation du patron ce conception SOA « Service de garde de périmètre » par une passerelle de sécurité.

2.4.2 Intercepteurs

Les intercepteurs, aussi appelés agents, implémentent un filtre logiciel placé immédiatement devant un service, tel qu'illustré à la Figure 11. Contrairement aux passerelles de sécurité, ils partagent le même espace mémoire que le processus système exécutant le service auquel ils sont associés. Les utilisations des intercepteurs sont multiples et ne se limitent pas à la sécurisation d'un service. Ils sont typiquement déployés en chaîne devant la ressource protégée et chaque maillon traite un aspect différent de la requête (sécurité, fiabilité, transaction, etc.).

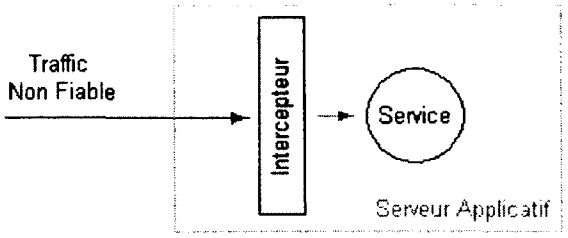

Figure 11 – Intercepteur [17]

Cette solution architecturale complète celle des passerelles de sécurité, en ce sens qu'elle règle le problème des services laissés sans protection contre un attaquant placé derrière la passerelle de sécurité. Toutefois, ce mécanisme présente deux inconvénients. D'abord, il nécessite la configuration et la maintenance d'au moins autant de filtres que de services. En conséquence, un changement au niveau des requis de sécurité, par exemple l'acceptation d'un nouveau type de jeton de sécurité, nécessitera un effort proportionnel à la taille de l'inventaire de services. Ensuite, les intercepteurs imposent une charge de travail additionnelle aux machines hôtes des services.

2.4.3 Configurations

Les paramètres de sécurité de chaque passerelle et de chaque intercepteur doivent être configurés. Typiquement, ceux-ci spécifient si les requêtes entrantes doivent être sécurisées au niveau du protocole de transport et si elles doivent inclure une sécurité au niveau du message. Dans les deux cas, des paramètres additionnels spécifient les types d'algorithme de chiffrement supportés, le besoin de vérifier une signature digitale, les clés ou certificats utilisés, etc. L'exécution et la maintenance de cette configuration est lourde à maintenir à l'échelle d'une entreprise. Cette section explore trois approches pour configurer les intercepteurs et les passerelles.

2.4.3.1 Configuration codée « en dure » dans chaque filtre

L'approche la plus utilisée pour sécuriser les services d'une entreprise est de figer dans le code de chaque service son mécanisme de sécurité [7]. En d'autres termes, chaque intercepteur est codé à la main. Ces implémentations peuvent ensuite être placées dans une bibliothèque pour être réutilisé par d'autres services.

Cette approche est simple et rapide à mettre en place, car les outils pour aider à sa réalisation sont nombreux et faciles d'utilisation. Elle ne requiert pas l'achat de produits commerciaux et peut être adaptée afin de réutiliser l'infrastructure de sécurité déjà en place. Pour ces raisons, la configuration codée « en dure » est idéale pour les systèmes d'informations dont l'inventaire de services est petit.

Toutefois, cette pratique nuit à l'évolution des systèmes, car la répartition des configurations à l'intérieur de chaque service rend leur mise à jour fastidieuse. Non seulement, par le besoin de reconfiguration manuelle de chaque service devant mettre à jour son mécanisme de sécurité, mais aussi par la besoin de reconfiguration manuelle des applications clientes à ces services.

2.4.3.2 WS-Policy, WS-SecurityPolicy et WS-PolicyAttachment

La spécification WS-Policy, rédigée par le groupe W3C, définit un standard pour exprimer les capacités ou les limitations d'un service sous forme de politiques [23]. Ce standard permet notamment d'exprimer les mécanismes de sécurité supportés par le service à l'aide de la spécification WS-SecurityPolicy, élaborée par le groupe OASIS. La spécification WS-PolicyAttachment du groupe W3C, encore au stade d'ébauche au moment de la rédaction, mais largement utilisée par les suites commerciales, propose à son tour un standard pour attacher les politiques WS-Policy à un document XML. Ainsi, ces politiques peuvent être incorporées à un document WSDL. L'annexe C présente un tel exemple. Nous notons que ces politiques ne sont qu'informatives et donc qu'elles ne garantissent pas que le service qu'elles décrivent leur est conforme.

Les politiques d'utilisation WS-Policy sont principalement utilisées par les outils de configuration automatique d'intercepteurs ou de passerelles de sécurité. Cette configuration peut être effectuée au moment de la conception ou au moment de l'exécution et elle peut servir soit à sécuriser une requête (côté client, à partir des politiques de sécurité du service cible) ou soit à valider une requête (côté serveur). Les politiques d'utilisation présentent deux avantages par rapport à l'approche « figée dans le

code ». Premièrement, elles augmentent l'interopérabilité d'un système en permettant aux partenaires ou aux développeurs de connaître automatiquement les mécanismes qu'un service supporte sans devoir ouvrir son code, et, deuxièmement, elles permettent de modifier les politiques de sécurité, encore une fois, sans toucher au code.

Toutefois, cette approche n'est pas parfaite. La création d'un interpréteur universel est impossible, car WS-Policy permet l'inclusion de politiques personnalisées. Ainsi, les implémentations commerciales d'interpréteur de politique WS-Policy restreignent généralement leur interprétation aux expressions de configuration de sécurité standards, prouvées sécuritaires. Ces restrictions varient d'un vendeur à l'autre et peuvent du même coup nuirent à l'interopérabilité entre deux systèmes.

2.4.3.3 Configurations centralisées

L'approche de centralisation des configurations n'est pas exclusive aux deux précédentes. Elle propose simplement de réunir les configurations des intercepteurs et des passerelles de sécurité dans un répertoire central. L'implémentation de ce répertoire peut être réalisée selon deux stratégies : soit à l'aide du registre de services, soit à l'aide d'une solution propriétaire. Dans le premier cas, les documents WSDL contenus dans le registre de services contiennent des politiques de sécurité à partir desquelles les intercepteurs et les passerelles de sécurité se configurent automatiquement. Dans le deuxième cas, les composantes communiquent avec le répertoire de configurations, à l'aide du protocole du vendeur, pour obtenir les informations de configurations. Cette dernière approche a cependant pour effet de lier la solution de sécurité à un vendeur spécifique.

2.5 Service de jetons de sécurité

Un service de jetons de sécurité, abrégé STS pour « *Security Token Service* », est le point central où sont gérés les jetons de sécurité. Un jeton de sécurité est simplement une représentation de l'identité d'un utilisateur. Ce jeton peut aussi bien être une combinaison du nom d'un utilisateur et de son mot de passe, un certificat X.509, une

chaîne unique de caractères, etc. La gestion des jetons de sécurité inclut leur création, validation, renouvellement, échange et destruction. Un service de jeton de sécurité est aussi fréquemment appelé un fournisseur d'identité.

En général, les STS sont employés dans les deux scénarios illustrés à la Figure 12, où un client (consommateur) désire consommer un service (producteur). Dans le premier scénario, le consommateur s'authentifie d'abord auprès d'un STS et lui demande un jeton de sécurité prouvant son identité. Le STS crée et retourne un jeton que le consommateur inclut à l'intérieur de son message vers le producteur. Dépendant du type de jeton, le producteur peut lui-même valider le jeton ou demander au STS de le faire.

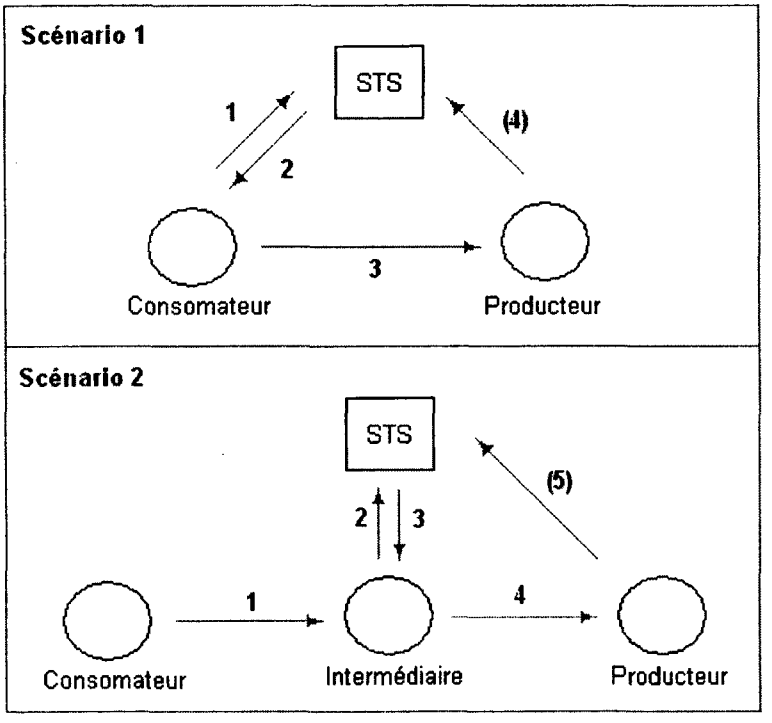

Figure 12 – Deux scénarios d'utilisation d'un STS

Dans le deuxième scénario, l'authentification du consommateur est réalisée par un intermédiaire, ce qui évite au client d'avoir à communiquer avec le STS. L'intermédiaire peut aussi bien être une passerelle de sécurité qu'un simple intercepteur. Ce mécanisme est aussi appelé « *Proxy Authentification* », car l'authentification est faite par l'intermédiaire, mais de la part du consommateur. Nous notons que, premièrement, le

consommateur doit tout de même fournir suffisamment d'informations à l'intermédiaire afin que ce dernier puisse l'identifier et que, deuxièmement, l'intermédiaire se porte garant de l'identité associée à la requête faite au producteur, c'est-à-dire que le producteur a confiance que l'intermédiaire ne tente pas de voler des jetons pour les associer à d'autres requêtes.

Les STS sont aussi utilisés pour implémenter des cas d'utilisation plus complexes, tels que fournir des jetons certifiant une autorisation ou fournir des attributs propres à un utilisateur et nécessaires à la prise de décision de son autorisation. Les jetons SAML (« *Security Assertion Markup Language* »), entre autres, supportent ces scénarios [11].

Plusieurs raisons motivent l'utilisation d'un STS. Premièrement, tel que discuté brièvement en début de chapitre, un STS est un point central d'administration des utilisateurs connectés au système. L'authentification des utilisateurs passe par le STS et ce dernier maintient un registre de toutes les sessions actives. Il contrôle, entre autres, la durée de vie de ces sessions. Du point de vue de l'utilisateur, cette option architecturale offre une solution de connexion unique pour accéder aux différents applications et services d'un système et, du point de vue des administrateurs, cette option facilite la maintenance. Deuxièmement, un STS protège le système contre les vols d'identités en étant le fournisseur unique d'identité et en signant les jetons. Ainsi, un jeton de sécurité ne peut être reproduit dans le but de réaliser une opération au nom d'une autre personne. Troisièmement, un STS libère les services du fardeau de la création du jeton de sécurité approprié pour communiquer avec un autre service. Cette stratégie est du même coup une alternative à la sécurisation automatique des requêtes à l'aide des politiques de sécurité, décrites dans la section précédente. Finalement, des protocoles tels que « *Security Assertion Markup Language Protocol* » (SAMLP) [11] et « *Web Service Trust* » (WS-T) [10] ont été établis pour standardiser la communication avec un STS. Cette standardisation facilite l'intégration d'une entreprise avec ses partenaires.

Un STS présente les désavantages de tout système centralisé, c'est-à-dire qu'il peut potentiellement constituer un point unique de défaillance et qu'il doit supporter une

charge importante. D'autres critiques affirment que les spécifications actuelles de STS sont trop vagues pour offrir un avantage réel au niveau de l'interopérabilité et qu'elles permettent des implémentations non-sécuritaires.

2.6 Propagation d'identité

La propagation d'identité est l'action de transmettre l'identité de l'utilisateur et une preuve de son authentification à chaque nœud impliqué dans le traitement d'une requête. Le but de cette propagation est d'éviter à l'utilisateur de devoir s'authentifier auprès de chaque service. Cette section présente quatre mécanismes non-exclusifs de propagation d'identité au sein d'une architecture SOA.

2.6.1 Confiance transitive

Dans un modèle de confiance transitive, chaque service d'une chaîne se fie à une assertion émise par le service précédent affirmant que l'utilisateur à l'origine la requête a été authentifié. Chaque service est donc responsable de générer une assertion contenant au moins l'identité de l'utilisateur, et ce, pour chacune de ses requêtes. Afin de protéger ces assertions, un lien de confiance est préalablement établi entre les services, typiquement par le partage de clés de chiffrement.

Ce modèle est simple à mettre en place, mais les services doivent assumer le coût de la création des assertions. De plus, si un service est compromis dans la chaîne, ce dernier peut faussement affirmer que l'identité d'un utilisateur a préalablement été authentifiée. En partant de ce dernier principe, Rosen propose que, dans ce modèle, plus on tend à s'éloigner de la source d'authentification, plus la confiance en celle-ci tend à descendre [17].

2.6.2 Propagation à l'aide d'un conteneur applicatif

Lorsque plusieurs services sont exposés à l'aide du même conteneur applicatif, tel un serveur Web, ils partagent un espace de mémoire commun dans lequel peuvent être conservé des contextes de sécurité à propos des utilisateurs authentifiés. Ces contextes de

sécurité assurent la propagation de l'identité des utilisateurs à l'intérieur du conteneur applicatif. Le scénario typique est illustré à la Figure 13 : le premier service contacté authentifie la requête ou valide l'assertion qui l'accompagne, puis, crée un contexte de sécurité contenant l'identité de l'utilisateur associé à la requête. Le deuxième service, contacté par le premier, n'a qu'à vérifier qu'un contexte d'information est, soit déjà attaché au fil d'exécution courant, soit récupérable dans la mémoire partagée à l'aide d'un identifiant passé à l'intérieur de la requête.

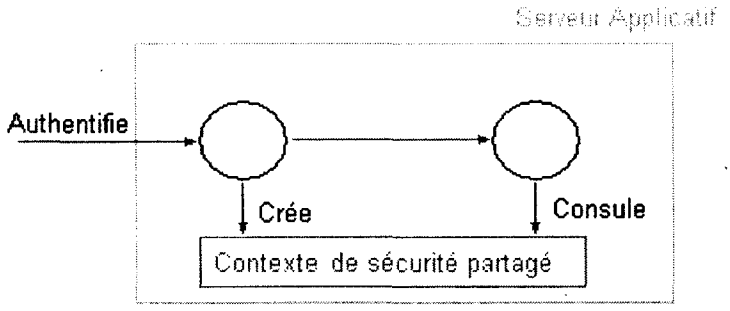

Figure 13 - Propagation de l'identité à l'aide d'un conteneur applicatif

La faiblesse de ce modèle est qu'il ne peut être utilisé pour propager une identité entre deux services que s'ils sont exposés dans le même conteneur. Néanmoins, l'absence d'appel externe rend ce modèle performant et il peut être utilisé pour optimiser la communication à l'intérieur de groupes de services.

2.6.3 Propagation à l'aide d'un service de jetons de sécurité

Le modèle de propagation d'identité à l'aide d'un STS est similaire au modèle de confiance transitive d'authentification et peut être réalisé à l'aide des scénarios d'utilisation d'un STS présentés à la Figure 12. Les services propagent donc les assertions créées par le STS. L'avantage de ce mécanisme, par rapport à celui de confiance transitive d'authentification, est que toutes les assertions sont signées (et datées) par une seule et même identité, ce qui diminue le nombre potentiel de services pouvant être compromis. De plus, la configuration et la gestion des clés associées à la création et à la validation des assertions sont simplifiées. Les désavantages de ce modèle sont simplement ceux d'utiliser un STS.

2.6.4 Propagation de l'identité aux services REST à partir du fureteur

L'identité de l'utilisateur peut être propagée entre des services REST à partir d'un fureteur en suivant le « *SAML Browser SSO Profil* » [11]. Ce mécanisme est basé sur le précédent, car il nécessite l'utilisation d'un STS. Le jeton SAML est inséré par le serveur dans le corps d'une requête HTTP à l'aide d'une soumission automatique d'un formulaire HTML. Le diagramme de séquence de la Figure 14 résume ce profil. D'abord, un utilisateur tente, via son fureteur, d'accéder à une ressource détenue par le service REST S1. Le service (ou plutôt son intercepteur) remarque que la requête n'est pas accompagnée d'une assertion et redirige donc l'utilisateur, via un code de retour HTTP 302, vers un fournisseur d'identité, abrégé IdP (« *Identity Provider* »). À son tour, l'IdP vérifie si l'utilisateur est déjà authentifié dans le système. Si ce n'est pas le cas, une page d'authentification est affichée. Une fois authentifié, l'utilisateur est de nouveau redirigé, via une soumission automatique de formulaire HTML, vers le service initial S1 avec une assertion SAML attaché à l'intérieur du corps de sa requête.

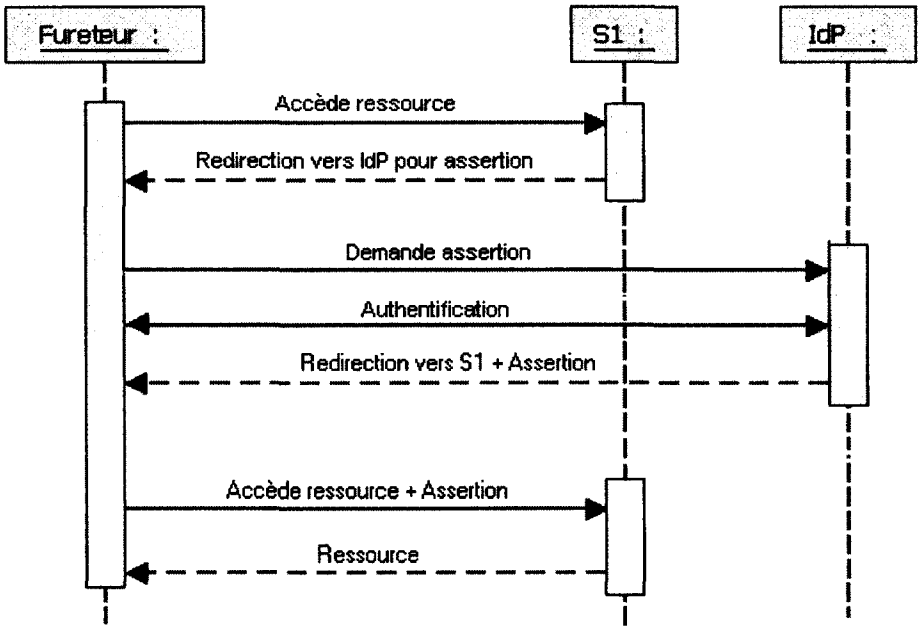

Figure 14 - Diagramme de séquence du « *SAML Browser SSO Profil* »

L'intérêt de ce mécanisme est qu'il offre un point où l'utilisateur peut s'authentifier auprès du système, c'est-à-dire saisir son nom et son mot de passe.

2.7 Modèles d'application et de gestion des politiques d'accès

La mise en vigueur d'une politique d'accès est l'action de stopper ou de laisser passer une requête selon si elle respecte ou non cette politique. Cette section présente les différents modèles architecturaux d'application et de gestion des politiques d'accès énumérés par Rosen, réalisables à l'aide des différentes combinaisons des mécanismes de sécurité présentés jusqu'ici [17].

Nous introduisons d'abord le modèle d'autorisation XACML (« *eXtensible Access Control Markup Language* ») [14], illustré à la Figure 15, afin de présenter les concepts en jeu lors de l'autorisation d'une requête. XACML est un langage XML pour exprimer des politiques d'accès et un protocole pour demander des permissions d'accès. Toutefois, nous ne présentons que le modèle conceptuel.

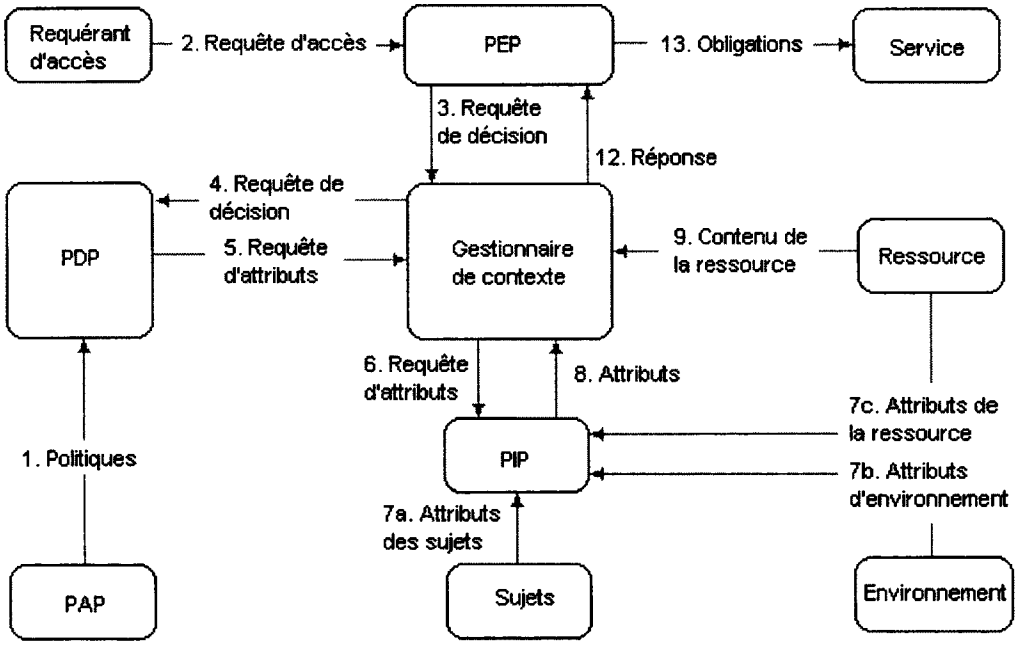

Figure 15 - Modèle XACML [14]

Le modèle XACML divise le processus d'autorisation en quatre composantes logiques principales : le point de mise en vigueur des politiques (PEP, pour « *Policy Enforcement Point* »), le point de décision des politiques (PDP), le point d'administration (ou le point

de configuration) des politiques (PAP) et le point d'information des politiques (PIP), responsable de récupérer les attributs reliés à la requête.

La séquence illustrée sur le modèle est la suivante :

1. Des politiques d'accès XACML sont configurées en avance à partir du PAP, généralement une interface graphique.
2. Le demandeur d'accès requiert l'accès à une ressource à un PEP.
3. Le PEP envoie la requête d'accès au gestionnaire du contexte dans le langage natif de ce dernier en incluant optionnellement des attributs à propos du sujet, de la ressource et de l'environnement.
4. Le gestionnaire du contexte construit une requête XACML pour demander au PDP de prendre une décision d'accès.
5. Le PDP demande au gestionnaire du contexte des attributs additionnels sur le sujet, la ressource et l'environnement.
6. Le gestionnaire de contexte relaie la demande d'attributs à un PIP.
7. Le PIP obtient les attributs demandés.
8. Le PIP retourne les attributs au gestionnaire de contexte.
9. Optionnellement, le gestionnaire du contexte inclut la ressource accédée à l'intérieur du contexte.
10. Le gestionnaire de contexte retourne les attributs au PDP et optionnellement la ressource. Le PDP évalue les politiques d'accès et prend une décision.
11. Le PDP retourne une décision d'autorisation en format XACML au gestionnaire de contexte.
12. Le gestionnaire de contexte traduit la décision d'autorisation dans le langage du PEP et lui retourne la réponse.
13. Le PEP applique la décision d'autorisation.

En omettant d'utiliser le langage XACML aux étapes 1-4-5-10, le modèle de XACML devient générique et permet de décrire la plupart des architectures d'autorisation. De plus, certaines composantes peuvent être réunies sous un même acteur. Ainsi, un intercepteur peut à la fois être PEP, PDP, PIP et gestionnaire de contexte.

2.7.1 Modèle purement centralisé avec politiques globales

Le modèle purement centralisé d'applications et de gestion des politiques d'accès, illustré à la Figure 16, délègue à un serveur central l'évaluation et la gestion des politiques. Les services envoient une demande de décision au serveur de politiques, puis ce dernier est responsable de collecter les attributs et les politiques relatifs à la requête et d'évaluer la permission d'accès.

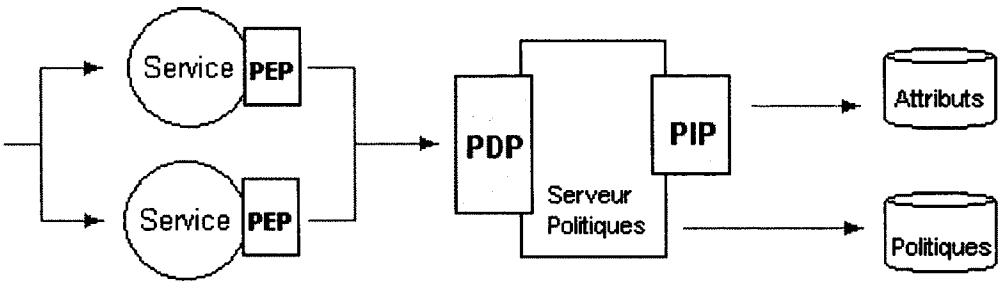

Figure 16 - Modèle d'autorisation purement centralisé avec politiques globales

Ce modèle offre d'abord l'avantage de cacher aux services et aux applications clientes les attributs et les politiques de sécurité, considérés comme des informations sensibles dans certains systèmes. De plus, le modèle centralisé réunit à un point la gestion des politiques de sécurité, facilitant la maintenance du même coup. Dernièrement, cette approche permet d'alléger la charge des services en leur retirant complètement le travail du calcul des autorisations. Cependant, comme nous avons déjà mentionné dans les sections précédentes, les systèmes centraux constituent un point de défaillance unique, demandent beaucoup de ressources matérielles et peuvent devenir un goulot d'étranglement. Encore une fois, ces défauts peuvent être mitigés à l'aide de la redondance, mais il faut alors parfois prévoir le coût des licences associées aux multiples instances. Le modèle purement centralisé avec politiques centralisées est utilisé dans les entreprises où les informations des utilisateurs sont très sensibles et ne doivent pas être vues par les services.

2.7.2 Modèle purement décentralisé avec propagation d'attributs

Le modèle purement décentralisé avec propagation d'attributs est à l'opposé du modèle précédent : alors que tous les services dépendent du serveur d'autorisation dans l'approche centralisée, les services sont autosuffisants dans l'architecture décentralisée. En effet, comme l'illustre la Figure 17, chaque service maintient ses propres politiques de sécurité et est responsable de calculer les autorisations et de les mettre en vigueur. L'astuce qui permet aux services de ne pas dépendre du répertoire global d'attributs est de déplacer cette dépendance du côté des applications clientes. Ainsi, ces dernières attachent à leurs requêtes les attributs de l'utilisateur pour qui elles agissent et ces attributs sont propagés au travers les services traversés. Afin d'assurer l'intégrité de ces attributs, une stratégie consiste à employer un service de jetons de sécurité pour fournir et signer ces attributs. Ce dernier signe et/ou chiffre les attributs à l'aide d'une clé qu'il partage avec tous les services.

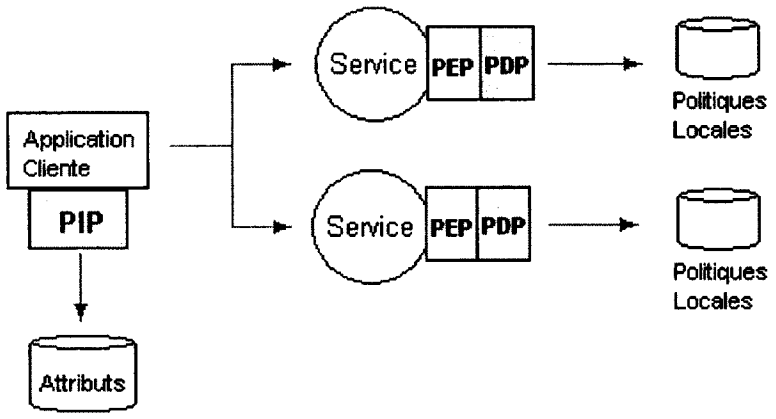

Figure 17 – Modèle d'autorisation purement décentralisé avec propagation d'attributs

Les avantages et les inconvénients du modèle décentralisé sont naturellement l'inverse du modèle centralisé. Le point de défaillance unique n'existe pas, mais les attributs peuvent être lus par les clients et par les services, la charge du calcul de l'autorisation repose sur les épaules des services et l'effort nécessaire pour la maintenance des politiques de sécurités et des clés est considérable. Ce modèle est particulièrement intéressant pour les systèmes où la disponibilité des services doit être très élevée.

2.7.3 Modèle décentralisé avec propagation d'identité

Le modèle d'autorisation décentralisé avec propagation d'identité, présenté à la Figure 18, est un modèle hybride entre les deux précédents. Les services, plutôt que de dépendre d'un serveur central d'autorisation, dépendent d'un répertoire central d'attributs. Ils sont responsables de récupérer ces attributs, de calculer l'autorisation et de la mettre en vigueur. Chaque service possède sa propre liste de politiques locales, possiblement déduite à leur démarrage ou à intervalle régulier à partir d'un répertoire globale centralisé de politiques. Une autre différence importante par rapport au modèle purement décentralisé est que, plutôt que de propager les attributs au travers les services depuis l'application cliente, c'est l'identité de l'utilisateur qui est propagée. Cette assertion d'identité peut être obtenue à l'aide d'un service de jetons de sécurité. L'avantage de propager l'identité, plutôt que les attributs, est que l'identité, au niveau de l'application cliente, n'est pas une information sensible.

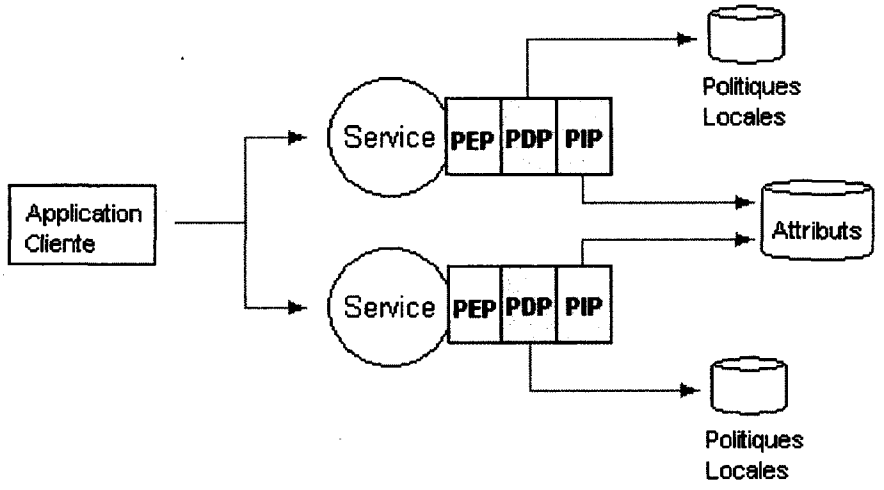

Figure 18 – Modèle d'autorisation décentralisé avec propagation d'identité

Cette architecture est un compromis entre les deux premières approches. Contrairement au modèle centralisé, le calcul d'autorisation ne peut devenir un goulot d'étranglement pour le système. Même si une composante centralisée demeure, soit le répertoire global d'attributs, le risque relié à sa défaillance peut être mitigé en configurant les services pour qu'ils conservent les attributs en mémoire pendant quelque temps. Ensuite, même si,

comme le modèle décentralisé, les services prennent la charge du calcul des autorisations, les attributs confidentiels sont cachés aux applications clientes. Bref, l'approche décentralisée avec propagation d'identité s'adresse aux entreprises où les attributs ne peuvent être vus des applications clientes et où la disponibilité des services est importante. Ce modèle est le plus populaire dans les produits commerciaux.

2.7.4 Modèle avec autorisations prédéterminées

Comme son nom l'indique, le modèle avec autorisations prédéterminées, illustré à la Figure 19, requiert d'une application cliente qu'elle possède préalablement une autorisation fournie par une autorité tierce avant d'accéder à un service. Dans ce cas, l'autorité tierce, typiquement un service de jeton de sécurité, a la responsabilité de récupérer les attributs (PIP), de calculer les autorisations (PDP) puis de fournir un jeton qui confirme la permission. Ce modèle peut notamment être utilisé pour implémenter une infrastructure de licences flottantes, où les licences sont représentées par les jetons et le STS limite leur distribution. Cette architecture nécessite la présence d'une infrastructure de clés publiques, où chaque client possède un certificat pour prouver son identité. Ceci permet au STS de lier le jeton de sécurité à l'identité de l'utilisateur. La spécification « *SAML profile for XACML* » supporte ce cas d'utilisation [14].

Nous notons qu'en refusant de fournir une autorisation, le STS agit aussi en tant que point de mise en vigueur des politiques d'accès (PEP). Les services, quant à eux, ne sont que des PEP.

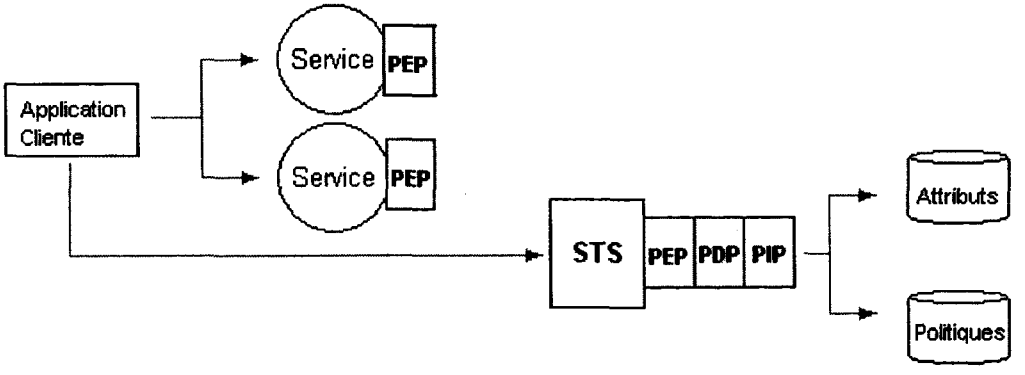

Figure 19 – Modèle de pré-autorisation

Le modèle d'autorisations prédéterminées possède aussi bien des caractéristiques du modèle centralisé que du modèle décentralisé. Dans le premier cas, le STS a le défaut de constituer un possible point de défaillance unique pour le système, mais la gestion des politiques est regroupée en un seul point, la charge au niveau des services est minime et les attributs des utilisateurs sont cachés aux applications et aux services. Dans le deuxième cas, les services sont autosuffisants car ils n'ont pas besoin de contacter le composant centralisé.

Nous croyons que ce modèle ne se prête pas aux scénarios où les politiques d'accès possèdent une fine granularité. Par exemple, dans le cas d'une architecture possédant des politiques de sécurité différentes pour chacun de ses services, il est pratiquement impossible pour un STS de savoir à l'avance tous les services qui peuvent être traversés par la requête d'un utilisateur pour laquelle il demande une autorisation. Si les politiques d'accès possèdent une forte granularité, ce modèle est recommandé, comme alternative au modèle centralisé, dans le cas où les attributs doivent être cachés aux services ou dans le cas où il y a un besoin pour la centralisation des politiques d'accès.

2.7.5 Résumé des modèles d'autorisation

Les principaux avantages et inconvénients de chaque modèle sont résumés au Tableau 1.

	Avantages	Inconvénients
Modèle purement centralisé	Confidentialité des attributs et des politiques de sécurité Facile à administrer Allège le traitement des services	Point central de défaillance Possibilité d'être un goulot d'étranglement
Modèle purement décentralisé	Services autonomes Haute disponibilité	Non-confidentialité des attributs et des politiques de sécurités Lourdeurs administratives
Modèle compromis	Services presque autonomes Confidentialité des attributs	Non-confidentialité des politiques de sécurité
Modèle de pré-autorisation	Autonomie des services Confidentialité des attributs et des politiques de sécurité Supporte une infrastructure de licences flottantes	Point de défaillance unique Possibilité d'être un goulot d'étranglement Non-support des politiques de sécurités de fines granularités Lourdeurs administratives

Tableau 1 – Résumé des modèles d'autorisation.

Chapitre 3 Étude du cas EB^{3}SEC

Ce chapitre introduit le projet EB^{3}SEC, développé conjointement par l'Université de Sherbrooke et l'Université Paris-Est Créteil, et tente de lui proposer une architecture de sécurité SOA. Il débute par une brève description du projet et une présentation de ses contraintes technologiques. Puis, il propose un modèle XACML adapté à ses particularités. Une d'elles, la gestion de l'historique des actions des utilisateurs, est ensuite étudiée en détail et différentes approches sont proposées pour l'implémenter. Finalement, une architecture d'autorisation SOA, adaptée à l'intégration du projet EB^{3}SEC et tenant compte des discussions de ce chapitre et du précédent, est proposée.

3.1 Description du projet EB^{3}SEC

Le projet EB^{3}SEC est basé sur les spécifications formelles et s'intéresse à la sécurité des systèmes d'information, dans un contexte SOA, à un niveau plus élevé que celui étudié jusqu'à présent. Ce niveau est celui de la sécurité fonctionnelle, définie comme les règles de sécurité spécifiées dans les processus d'affaires des entreprises. Le projet propose, dans un premier temps, une méthode de spécification des règles de sécurité au niveau des attributs d'une entité, au niveau des services atomiques et au niveau des processus d'affaires et, dans un deuxième temps, le projet propose des algorithmes de synthèse automatique de noyau de sécurité pour traduire une politique de sécurité en un programme exécutable.

EB^{3}SEC divise les contraintes de contrôle d'accès en deux catégories : les contraintes statiques qui ne dépendent pas de l'état du système ou de son évolution et les contraintes dynamiques qui, à l'inverse, dépendent de l'état du système. Parmi les contraintes dynamiques, on compte les permissions avec contraintes (permissions qui ne sont valides

que sous certaines conditions), les interdictions avec contraintes (interdictions qui ne sont valides que sous certaines conditions), les contraintes de type séparation des devoirs résolues dynamiquement (une personne ne peut poser une action *b* sur une ressource *r* si elle a déjà effectué une action *a* sur cette ressource *r*) et les contraintes d'obligation (lorsqu'une personne effectue une action *a*, l'action *b* doit également être faite). Ces contraintes dynamiques servent principalement à spécifier les règles de sécurité au niveau des processus d'affaires.

Les deux premiers niveaux de sécurité, celui des attributs d'une entité et celui des services atomiques, sont aisément transposables dans le modèle XACML en faisant correspondre les services aux ressources accédées. En revanche, celui des processus d'affaires est un concept absent du chapitre précédent qui nécessite une infrastructure pour conserver et consulter rapidement les actions effectuées par les acteurs du système. De plus, la mise à jour systématique de cet historique pour chaque action des utilisateurs et la synchronisation de son état avec celle du système d'information demande une revue du modèle XACML. La Figure 20 résume les sources d'information consultées par le projet EB^{3}SEC afin d'autoriser une requête, soit les répertoires des utilisateurs, des attributs, des politiques de sécurité et des historiques.

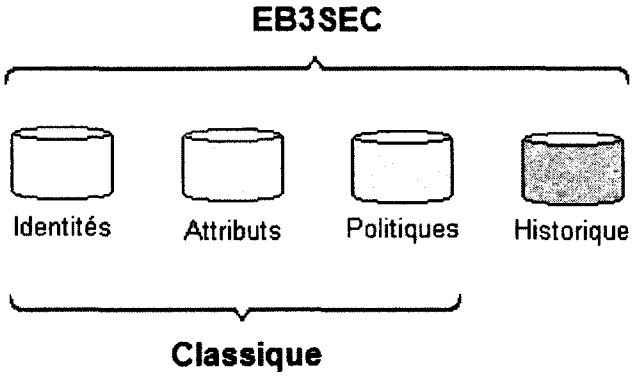

Figure 20 – Informations consultées pour l'autorisation dans le projet EB^{3}SEC

La synchronisation entre l'historique des actions posées par les utilisateurs et entre l'état du système est primordiale pour assurer la sécurité ou encore l'accessibilité des

fonctionnalités, et ainsi assurer l'intégrité des processus. Les mises à jour du système et du noyau de sécurité doivent donc être effectuées simultanément, d'où le besoin pour un mécanisme de transaction. Selon l'architecture choisie, le recours aux transactions distribuées peut être nécessaire.

3.2 Modèle XACML adapté

Le modèle de XACML ne peut être appliqué tel quel au projet EB^{3}SEC, car il ne comprend pas la notion d'historique. Bien que cette notion s'apparente au concept d'attribut, en ce sens qu'elle qualifie une entité (ici l'état des processus auxquels participent les utilisateurs) et qu'elle est consultée avant de prendre une décision d'autorisation, les attributs et les données d'historique se distinguent sur trois points. Premièrement, les données d'historique, constamment mises à jour et possiblement archivées régulièrement, sont plus dynamiques que les attributs, compliquant ainsi leur conservation en mémoire cache et leur propagation au travers les compositions de services. Deuxièmement, puisque les données d'historique peuvent contenir des informations sensibles telles des numéros de compte, on considère qu'elles ne devraient jamais être accessibles aux utilisateurs. Troisièmement, le conteneur (ou gestionnaire) de ces dernières, par exemple un système de gestion de base de données, doit pouvoir participer à une transaction distribuée afin d'assurer la synchronisation de l'historique avec l'état du système. En raison de ces distinctions, nous proposons une séparation logique entre les attributs et les données d'historique. Ainsi, nous ajoutons deux entités au modèle XACML, soit le point de consultation de l'historique, HIP (« *History Information Point* »), et le point de mise à jour de l'historique, HUP (« *History Update Point* »), tel qu'illustré dans le modèle de la Figure 21, qui simplifie les étapes de récupération des attributs. Le HIP est consulté, au besoin, par le gestionnaire de contexte après ou en même temps que le PIP, puis, l'historique est retourné avec les attributs au PDP. Le cas du HUP est plus complexe, car le gestionnaire de contexte doit débuter une transaction et propager celle-ci au service via le PEP en même temps qu'il retourne la décision d'autorisation. Cette transaction est utilisée pour mettre à jour l'historique et l'état du système de façon atomique. Nous notons qu'il peut être nécessaire pour le

gestionnaire de contexte de débuter la transaction dès l'étape 10 dans le cas où les consultations de l'historique doivent également être synchronisées avec ses mises à jour.

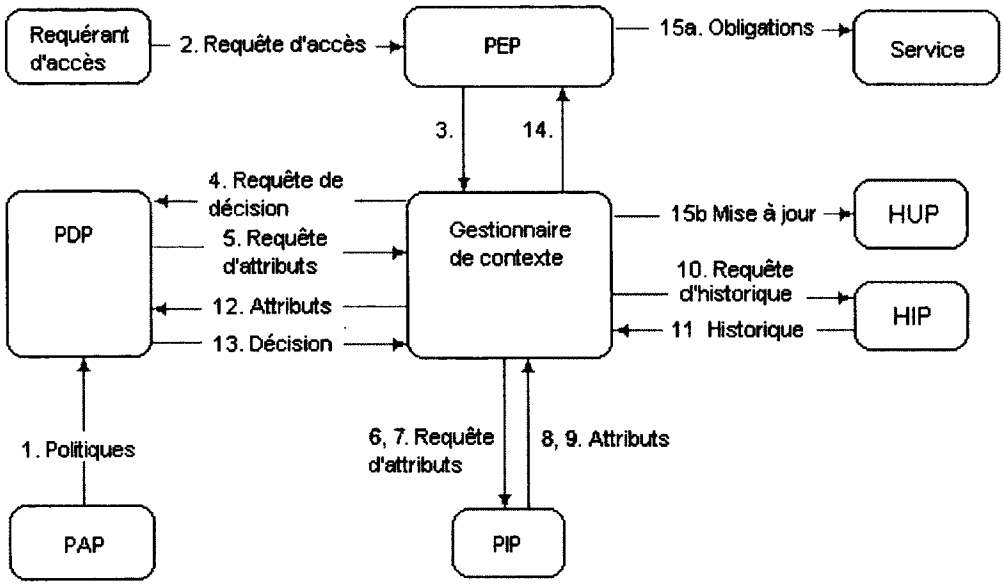

Figure 21 – Modèle XACML adapté au projet EB^{3}SEC

3.3 Gestion de l'historique

La gestion de l'historique peut se résumer en la conservation de fichiers d'audits sur des disques durs. Cependant, dans le cas du projet EB^{3}SEC, l'architecture doit permettre la recherche et la mise à jour rapides de cet historique, en plus de permettre sa participation à des transactions. Dans cette section nous étudions, dans un premier temps, les stratégies de mise à jour de l'historique en proposant une approche implicite et une approche explicite, puis dans un second temps, nous proposons des stratégies d'emplacement de l'historique, soit une stratégie locale et une stratégie centralisée.

3.3.1 Mise à jour implicite

La mise à jour implicite de l'historique est une approche simple à la gestion d'un historique. L'idée est d'intercepter les requêtes d'accès à une ressource et de noter l'utilisateur à l'origine de cette requête ainsi que la ressource demandée. Par exemple, le service de gestion de la base de données (SGBD) d'*Oracle* offre la possibilité de

configurer des règles d'audits sur des tables ou des procédures stockées. À l'aide de ces règles, une rangée est insérée automatiquement dans une table d'audit chaque fois qu'un de ces objets est consulté ou exécuté. Dans ce contexte, les tables d'audit deviennent l'historique du système. Dans les systèmes où tous les accès aux bases de données sont effectués via des procédures stockées, la mise à jour de l'historique peut être codée manuellement à l'intérieur de ces procédures stockées. Cette dernière technique, quoique rarement applicable, offre une grande flexibilité. Ces deux stratégies sont illustrées par la Figure 22.

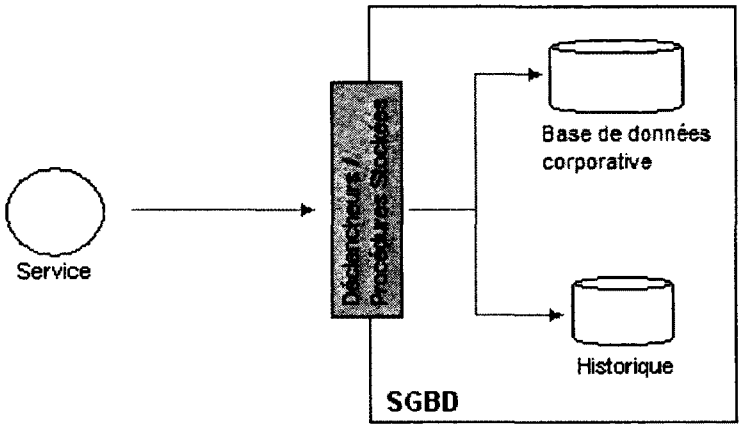

Figure 22 – Mise à jour implicite de l'historique au niveau du SGBD

L'avantage principal de l'approche implicite est sa simplicité d'implantation. Elle ne requiert pas la mise en place d'un service séparé de gestion d'historique, elle peut être implémentée à l'aide de quelques lignes de code dans le cas de politiques d'audits et les développeurs de services (internes et partenaires) n'ont pas à savoir qu'elle existe. De plus, dans le cas où un SGBD est utilisé, l'approche implicite assure la synchronisation entre l'état du système et l'état de l'historique puisque ces deux états sont gérés simultanément et localement par le SGBD, spécialiste des transactions.

Toutefois, la mise à jour implicite de l'historique présente des limites. Premièrement, seules les actions interceptées par le composant « historien » peuvent être archivées. Si une méthode d'un service ne traverse pas le composant « historien », aucune trace de cette méthode ne peut être conservée dans la base de données d'historique.

Deuxièmement, il est aisé d'oublier que cette mise à jour systématique « cachée » peut causer un impact de performance dans les systèmes hautement transactionnels. Finalement, le composant « historien » doit se débrouiller avec la requête reçue, c'est –à-dire que si, par exemple, elle ne permet pas d'identifier l'action qui a réellement été effectuée du point de vue des affaires, il peut être impossible de mettre l'historique à jour.

3.3.2 Mise à jour explicite

À l'inverse de la gestion implicite de l'historique, l'approche explicite implique que les services sont responsables de mettre à jour l'historique, soit par un appel à une base de données, soit par un appel à un autre service ou soit simplement par une écriture dans un fichier. La gestion explicite est illustrée à la Figure 23. Un service met à jour le contenu de deux bases de données, puis il demande explicitement au service d'historique, noté « H », de conserver la trace de ces mises à jour.

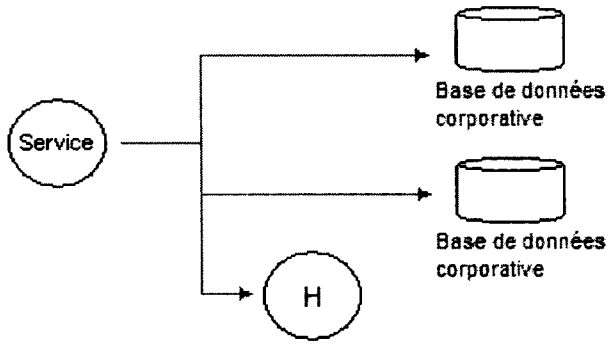

Figure 23 – Appel explicite à un service de mise à jour d'historique

À petite échelle, l'implantation d'une gestion d'historique explicite est simple et rapide à mettre en place. Cependant, la maintenance de cette logique répartie dans chaque service peut devenir laborieuse au fur et à mesure que le nombre de services croît. Au niveau de la sécurité, l'approche explicite présente aussi un désavantage par rapport à l'approche implicite, car le service de mise à jour de l'historique peut être accédé par plusieurs services, plaçant ainsi l'historique lui-même à la merci des programmeurs mal intentionnés. Finalement, chaque service est responsable de coordonner ses mises à jour entre les données de l'entreprise et l'historique, c'est-à-dire que chaque service doit gérer des transactions ou demander à une tierce partie de le faire (comme dans le standard WS-

TX d'OASIS). Dans le cas où le service d'historique n'est pas un SGBD qui héberge également les données de l'entreprise, les services doivent avoir recours aux transactions distribuées, coûteuses en termes de performance, difficiles à configurer et parfois simplement non supportées.

Malgré ces inconvénients, la gestion explicite de l'historique demeure plus flexible que la gestion implicite. En effet, contrairement à l'approche implicite, les services ont la possibilité de mettre à jour l'historique autant de fois que nécessaire pour une même requête, et ce, avec des informations qui ne sont pas utilisées dans les requêtes effectuées aux ressources externes.

3.3.3 Emplacement local

Lorsque l'historique est local, chaque service possède son propre historique, tel qu'illustré à la Figure 24. Les mises à jour peuvent être effectuées explicitement par le service, ou implicitement par un autre composant.

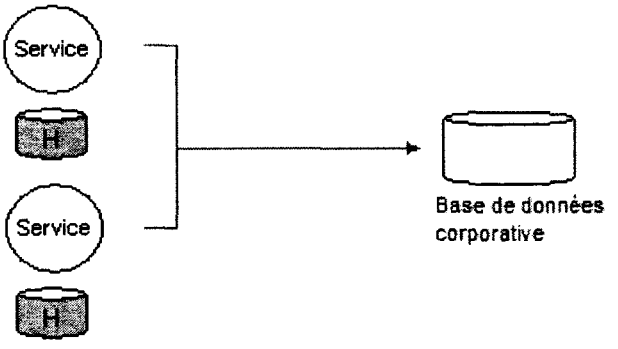

Figure 24 – Gestion locale de l'historique

Dans cette approche, lorsque le nombre de services augmente dans le système d'une entreprise, ceux-ci ne subissent que très peu de pertes de performance liées à la gestion de l'historique puisque les historiques ne sont mis à jour que par une seule une ressource. De la même façon, l'historique ne représente pas un point de défaillance unique. De plus, dans le cas où l'historique est embarqué dans le service, la synchronisation entre la base de données corporative est simple, car le recours aux transactions distribuées n'est pas nécessaire.

Toutefois, dans cette approche, le retraçage de l'historique d'un processus traversant plusieurs services est difficile. Ainsi, l'implémentation d'une politique de sécurité d'un processus basée sur l'historique de ce dernier est difficile. Également, la maintenance de plusieurs historiques demande beaucoup d'efforts. Nous notons qu'en raison de la nature sensible des données de l'historique, une attention particulière doit être portée à ces dernières.

3.3.4 Emplacement centralisé

Contrairement à la gestion locale d'historique, la gestion centralisée regroupe les historiques de toutes les opérations effectuées dans un système à un seul endroit. Encore une fois, cette gestion centralisée peut être implicite ou explicite (comme à la Figure 25). Les avantages et les désavantages d'une gestion centralisée sont essentiellement l'inverse de ceux d'une gestion locale, c'est-à-dire qu'elle permet d'avoir une vue globale sur les processus et facilite la maintenance augmentant du même coup la sécurité des données, mais elle comporte les désavantages d'un système centralisé, soit la présence d'un goulot d'étranglement, d'un point de défaillance unique et la nécessité d'avoir recours aux transactions distribuées afin d'assurer la synchronisation entre l'état de la base de données et celle de l'historique. Nous notons que les transactions distribuées ne sont pas nécessaires dans le cas où l'historique et la base de données sont tous deux gérés par le même SGBD.

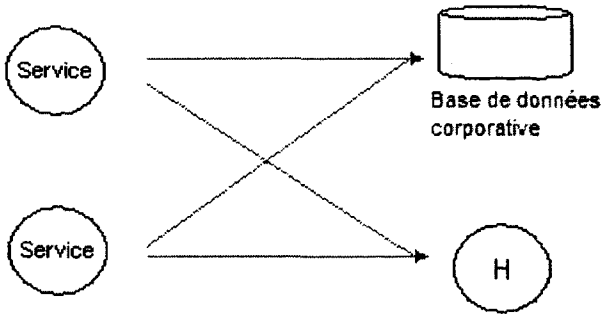

Figure 25 – Gestion centralisée explicite de l'historique

3.4 Architecture d'autorisation du projet EB^{3}SEC

Cette section propose une architecture d'autorisation au projet EB^{3}SEC, d'abord, en étudiant la faisabilité des modèles de gestion de l'historique, puis en étudiant la faisabilité des modèles d'application et de gestion des politiques de sécurité. Ces modèles sont comparés dans le contexte du projet EB^{3}SEC, c'est-à-dire avec le besoin de conserver un historique détaillé et facilement consultable pour chaque processus, avec le besoin de protéger les données d'historiques et avec le besoin d'assurer la synchronisation entre l'historique et l'état du système.

3.4.1 Modèles de gestion de l'historique

De manière générale, l'approche de mise à jour implicite de l'historique est préférable à l'approche explicite, car elle présente une maintenance moins coûteuse et, dans le cas où elle est réalisée par le SGBD, elle permet d'éviter l'implémentation d'un mécanisme complexe de synchronisation entre l'état de l'historique et celui des données d'entreprise. Cependant, le projet EB^{3}SEC exprime des politiques de sécurité à un niveau de granularité très fin et, selon la politique, peut nécessiter de nombreux détails à propos de la requête à autoriser et de son contexte. Le mécanisme de mise à jour de l'historique doit donc permettre de récupérer les paramètres de la requête et de les archiver. À moins d'avoir un système complètement basé sur des procédures stockées (ce qui ne peut être pris comme hypothèse), cette dernière contrainte ne peut être réalisée simplement à l'aide des déclencheurs ou des règles d'audit d'un SGBD pour deux raisons principales : premièrement, ces techniques s'appliquent sur des tables accédées à partir desquelles il est difficile, voir impossible, de comprendre l'action initiale posée par l'utilisateur, et deuxièmement, les comptes utilisateurs de SGBD utilisés par les services sont généralement des comptes génériques à partir desquels il est impossible de retrouver l'utilisateur réel. Pour ces raisons, le mécanisme de mise à jour retenu est une approche explicite à l'extérieur du SGBD. L'architecture comprendra donc un mécanisme de synchronisation.

Finalement, puisque EB^{3}SEC s'intéresse à la sécurité des processus, l'historique ne peut être conservé localement, car cette approche complique la reconstruction de l'historique d'un processus. Pour cette raison nous privilégions une approche centralisée.

3.4.2 Modèles d'autorisation

D'entrée de jeux, nous éliminons le modèle d'autorisation purement décentralisé. Comme mentionnée à la section précédente, une décentralisation de l'historique complique l'accès à un historique cohérent des processus impliquant plusieurs services. De plus, le modèle décentralisé a pour but de rendre les services indépendants en matière de sécurité, ce qui implique que le partage de l'historique doit être fait par propagation à l'intérieur des requêtes, de même que pour les attributs. Puisque les données d'historique sont sensibles, elles ne doivent pas transiter par les applications clientes.

Le projet EB^{3}SEC peut théoriquement employer un modèle avec autorisations prédéterminées. Dans ce cas, EB^{3}SEC devient une extension du STS et s'occupe aussi de la consultation de l'historique, comme illustré à la Figure 26. En pratique, ce modèle est peu utilisé. Premièrement, les jetons d'autorisation ne supportent pas nécessairement la propagation d'identité, ce qui peut nécessiter un jeton d'authentification additionnel par requête. Deuxièmement, contrairement aux jetons d'authentification, le même jeton d'autorisation ne peut être réutilisé dans les deux requêtes consécutives du scénario de la Figure 6-B, à moins que les politiques d'autorisations du système soient de très forte granularité ou encore que le jeton exprime plusieurs politiques d'autorisations (par exemple à l'aide de XACML). Dans ce dernier cas, la taille des jetons peut devenir problématique. Un compromis à ce problème est de laisser le soin au STS de calculer l'ensemble des autorisations nécessaires afin que le client puisse accomplir le processus pour lequel il demande à être autorisé. Cependant, un tel calcul est complexe à effectuer et à maintenir. Finalement, en général, les STS commerciaux n'offrent pas beaucoup de point d'extension et, donc, EB^{3}SEC devrait implémenter son propre STS, ce qui est en dehors de la portée du projet.

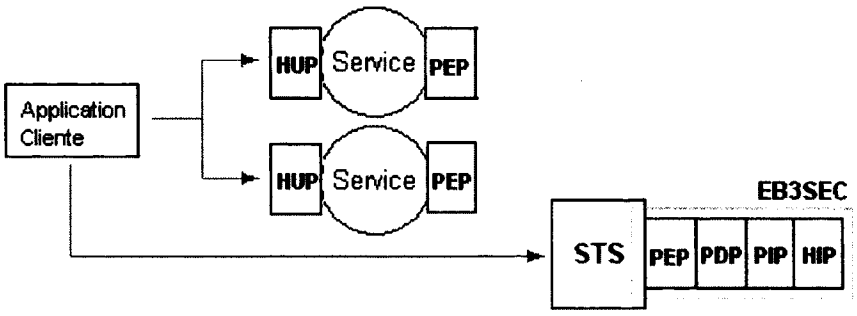

Figure 26 – EB^{3}SEC dans un modèle de pré-autorisation

Le modèle d'autorisation centralisé est tout à fait indiqué pour le projet EB^{3}SEC. Dans ce contexte, EB^{3}SEC devient le serveur central de prise de décision et se charge, pour chaque demande d'autorisation, de récupérer les attributs et de consulter l'historique, assurant du même coup la confidentialité de toutes les informations d'autorisation. Dans cette approche, représentée à la Figure 27, les services peuvent communiquer avec EB^{3}SEC dans un protocole propriétaire, contrairement au modèle avec autorisations prédéterminées qui requiert la médiation d'un STS.

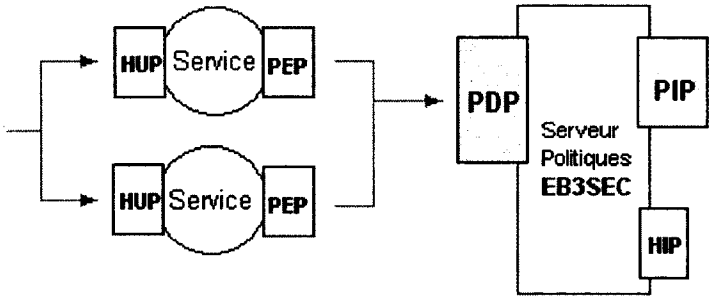

Figure 27 – EB^{3}SEC dans un modèle d'autorisation centralisée

Finalement, la combinaison du modèle d'autorisation décentralisé avec propagation d'identité (modèle compromis) à EB^{3}SEC, illustré à la Figure 28, est aussi faisable. Il impose cependant la contrainte importante que le noyau décisionnel de EB^{3}SEC soit déployé sur chaque service. Cette caractéristique permet un important gain de performance par rapport au modèle centralisé, mais limite le choix du langage

d'implémentation de EB^{3}SEC à un langage pouvant être appelé depuis celui utilisé pour coder les services, ce qui va à l'encontre des technologies retenues pour le projet EB^{3}SEC.

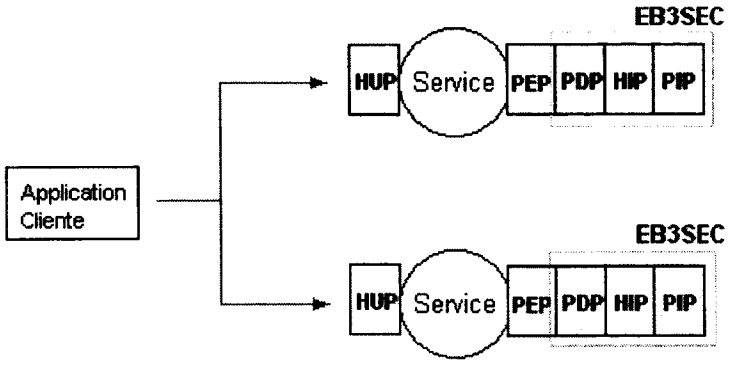

Figure 28 – EB^{3}SEC dans un modèle d'autorisation compromis

Bref, au stade du projet EB^{3}SEC au moment de l'écriture de cet ouvrage, le seul modèle d'autorisation possible pour EB^{3}SEC est le modèle centralisé.

3.5 Architecture complète de référence pour EB^{3}SEC

La dernière étape de l'étude du projet EB^{3}SEC consiste à intégrer son architecture d'autorisation à une architecture de sécurité d'entreprise. Cette dernière doit assurer la confidentialité, l'intégrité des communications ainsi que la disponibilité, l'accessibilité et l'évolutivité du système. Elle doit aussi permettre la sécurisation des trois scénarios présentés à la Figure 6 ainsi que la sécurisation de l'ensemble des processus. Elle doit offrir un mécanisme de connexion unique aux utilisateurs et de propagation d'identité aux travers les services, le tout en promouvant l'interopérabilité (entre les services de l'entreprise et avec ceux des partenaires), une maintenance simple et un temps de réponse acceptable pour les utilisateurs. En intégrant les différents composants décrits dans le Chapitre 2, nous proposons l'architecture de référence illustrée à la Figure 29.

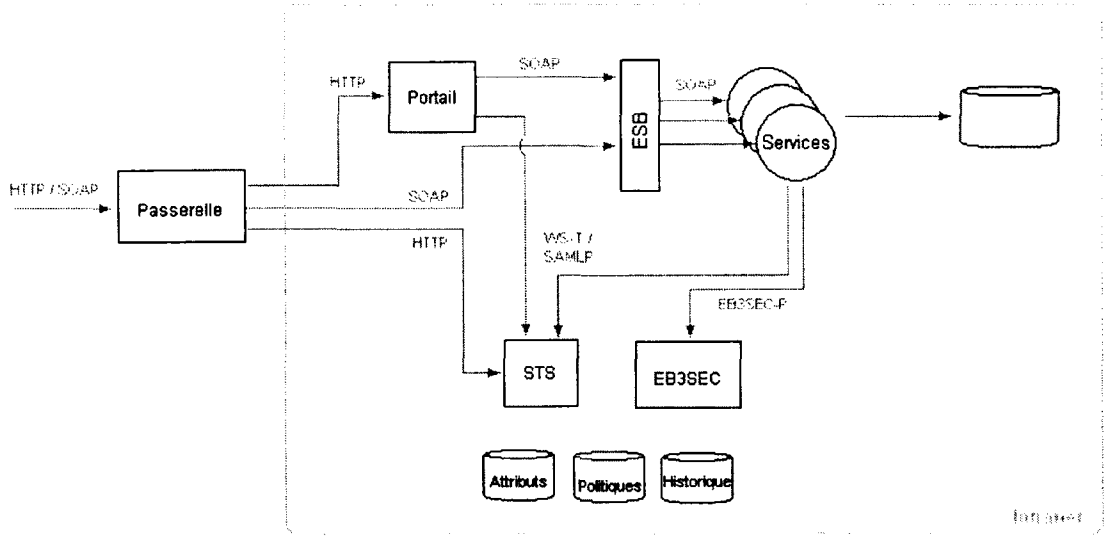

Figure 29 – Architecture de référence pour le projet EB^{3}SEC

Tous les composants sont déployés à l'intérieur d'une zone sécurisée à l'aide de murs pare-feux afin de limiter les connexions aux clients autorisés. Dans ce cas, seule la passerelle de sécurité est autorisée à franchir cette zone. Elle implémente donc le patron de conception « service de garde de périmètre ». L'avantage de ce patron est de pouvoir filtrer les requêtes incomplètes ou invalides dès leur arrivée et ainsi libérer les autres composants de cette tâche. Bien que ce ne soit pas illustré dans l'image par souci de lisibilité, la passerelle peut authentifier et/ou autoriser les requêtes. Habituellement, seule l'authentification des requêtes SOAP est effectuée, et ce, par la validation de la signature d'un jeton SAML inclus dans la requête.

Le STS permet d'offrir un mécanisme de connexion unique et de propagation des identités. Si un client tente d'accéder à son portail et qu'il n'est pas authentifié, celui-ci est redirigé vers une page d'authentification hébergée par le STS. Une fois les informations d'authentification saisies, le STS crée une session pour l'utilisateur et lui remet le numéro d'identification unique de cette session via un cookie. Ce cookie peut ensuite être présenté aux applications web qui le valideront auprès du STS. À l'aide de ce cookie, ces mêmes applications web peuvent demander au STS un jeton « complet » pour l'utilisateur, par exemple un jeton SAML, afin de pouvoir propager l'identité aux

services. Nous mentionnons que les identités des utilisateurs sont généralement conservées et maintenues à l'aide d'un répertoire LDAP avec lequel le STS communique.

Bien qu'un « *Enterprise Service Bus* » (ESB) ne soit pas obligatoire dans une architecture orientée service, sa popularité est grande auprès des entreprises qui possèdent un large inventaire de services. Un ESB permet, entre autres, d'offrir un point unique d'accès aux services, des mécanismes de routage ou de balancement de charge automatique vers les différentes versions d'un même service et des outils de transformation ou d'augmentation des messages. La raison de sa présence dans notre architecture de référence est simplement de montrer que du point de vue de la sécurité, un ESB peut être considéré comme un simple service, c'est-à-dire qu'on peut le configurer (ou non) afin qu'il valide (ou propage) un jeton SAML ou afin qu'il autorise une requête. [15]

Finalement, des intercepteurs sont configurés sur chacun des services. Ils sont responsables d'effectuer, dans l'ordre, les opérations suivantes pour chaque requête : au besoin, déchiffrer le message, valider la signature du jeton SAML, demander à EB^{3}SEC d'autoriser la requête, démarrer une transaction distribuée, faire une tentative de mise à jour de l'historique, déléguer le traitement de la requête au service et, dépendant du mécanisme de transaction utilisé, possiblement tenter de commettre la transaction. De la même façon, des intercepteurs sont aussi configurés sur les services afin de sécuriser les messages et afin de propager l'identité de l'utilisateur en demandant un jeton SAML au STS.

Chapitre 4 Réalisation

La dernière portion de cet ouvrage présente une implémentation de l'architecture de référence du projet EB^{3}SEC proposée à la section 3.5. Le but est, dans un premier temps, d'offrir une infrastructure au département du GRIL de l'Université de Sherbrooke avec laquelle le projet EB^{3}SEC peut être testé et, dans un deuxième temps, d'évaluer à haut niveau notre architecture de référence. En raison du manque de tests, de la gestion incomplète des erreurs et du petit nombre de cas d'utilisation supportés, la solution développée n'est pas de niveau industriel.

Le chapitre débute par la description des contraintes et de leurs impacts sur le design retenu. Ensuite, les technologies et les outils employés sont brièvement détaillés, puis le chapitre se termine par une brève discussion sur le résultat final ainsi que sur les problèmes rencontrés.

4.1 Contraintes et design

L'architecture de référence du projet EB^{3}SEC comprend plusieurs composants optionnels destinés aux entreprises de grande taille et que, faute de temps ou de budget, nous choisissons de ne pas inclure à notre design. D'abord, les passerelles de sécurité sont généralement des produits matériels spécialisés très coûteux dont le principal avantage est leur vitesse d'exécution pour valider ou transformer les requêtes, ce qui n'est pas requis dans cette réalisation. Toutefois, nous pensons qu'une passerelle peut être considérée comme un simple service intermédiaire et que son addition future dans notre solution n'aurait qu'un impact mineur. De même, les ESB dans les architectures SOA agissent comme proxy aux services et offre des capacités avancées de routage, de répartition de charge, de transformation de message, de conversion de protocole et de transparence de la localisation réelle des services. Toutes ces facilités ne sont utiles qu'aux systèmes complexes d'entreprises comprenant un inventaire important de services

et qui ont des besoins d'intégration complexes et variés. Pour ces raisons et dans le but de restreindre la portée du projet, nous choisissons de ne pas inclure ces deux composants à notre solution finale.

La seconde contrainte à l'implémentation de notre architecture de référence est que le projet EB^{3}SEC n'est pas terminé au moment de la rédaction. Ainsi, nous ne savons pas si la communication avec le gestionnaire de contexte d'EB^{3}SEC doit être effectuée à l'aide d'un protocole propriétaire ou à l'aide d'un protocole standard, tel que le protocole XACML. De même, nous ne connaissons pas le modèle de données sous lequel l'historique des actions des utilisateurs est conservé, ni si celui-ci est exposé aux services. Pour ces raisons, nous avons choisi d'inclure un API constitué d'un ensemble d'interfaces qui pourront être implémentées lorsque ces questions seront réglées. Pour l'instant, nous choisissons d'interagir avec une base de données pour simuler l'interaction avec EB^{3}SEC.

La troisième et dernière contrainte à notre implémentation est la complexité des transactions distribuées. La plupart des plateformes modernes offrent des mécanismes de transactions distribuées implémentant la norme XA. Par exemple, un serveur conforme à la spécification JEE doit offrir un gestionnaire de transactions distribuées aux applications qu'il héberge et ce dernier doit permettre la propagation des transactions au travers d'autres instances de ce serveur. Chaque plateforme possède ses propres mécanismes internes pour réaliser cette propagation. Afin de permettre une propagation entre deux plateformes hétérogènes, on doit recourir à des protocoles de haut niveau, telle la spécification WS-TX, qui dépend du protocole SOAP, lui-même indépendant de la technologie d'implémentation. Cependant, les produits gratuits qui implémentent la spécification WS-TX sont rares et difficiles à configurer. Ainsi, dans le cadre de ce travail, nous nous limitons à des transactions XA gérées par une seule plateforme technologique.

Le design résultant de l'application de ces contraintes à notre architecture de référence est présenté à la Figure 30. Afin d'accélérer et de simplifier le développement, tous les

composants sont exécutés sur la même machine physique. Les services sont regroupés sur une instance de conteneur web distincte de celle du service de jeton de sécurités, ceci dans le but d'accélérer le redémarrage des conteneurs. Pour cette preuve de concept, l'utilisation de la base de données et son contenu sont gardés minimaux.

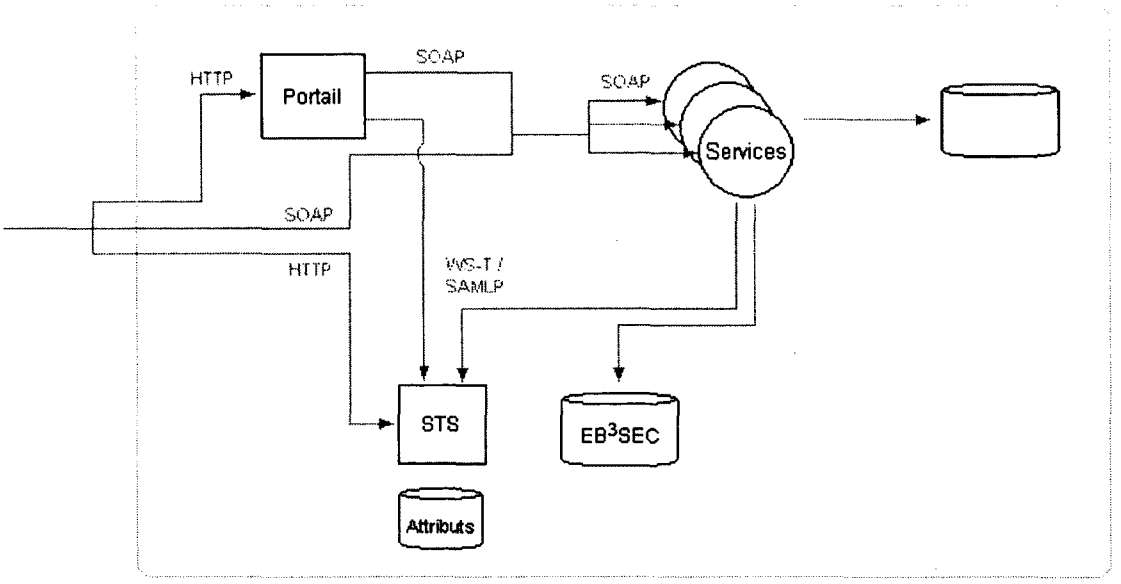

Figure 30 – Design de l'implémentation de l'architecture de référence d'EB^{3}SEC

Tableau 2 – Éléments exclus du design de la preuve de concept

Éléments Exclus	Explication
Passerelle de sécurité	Les passerelles de sécurité sont des produits coûteux agissant à titre de proxy. Leur addition éventuelle ne compromettrait pas notre architecture.
ESB	Les ESB agissent également à titre de proxy et ne sont requis qu'afin d'accélérer ou de faciliter l'intégration entre les différents systèmes d'une entreprise. Leur utilisation n'est pas pertinente et ajouterait de la lourdeur à notre solution. Toutefois, leur addition éventuelle ne compromettrait pas notre architecture.
WS-AtomicTransaction	Les transactions au niveau des services web sont complexes et leur support est limité au niveau des conteneurs applicatifs gratuits. Puisqu'elles ne sont pas essentielles à notre développement, nous ne les utilisons pas.

4.2 Implémentation

Cette section débute par l'énumération des produits et technologies retenues pour l'implémentation de notre design, puis, les mécanismes utilisés pour intégrer ces composants sont présentés.

4.2.1 Technologies

Tout le développement de notre preuve de concept est basé sur la plateforme JEE en raison du nombre important de projets et d'outils gratuits qu'elle comprend, de sa portabilité et de notre expérience avec cette dernière. Ainsi, nous exposons notre interface web et nos services SOAP à l'aide du serveur applicatif JBoss 7, développé par la fondation RedHat et entièrement écrit en Java. Bien que les serveurs applicatifs gratuits soient nombreux, JBoss est un produit stable et, contrairement au populaire conteneur de « *servlet* » Tomcat, il est conforme à la norme JEE6. Cette conformité implique notamment la présence d'un engin de services SOAP JAX-WS, d'un gestionnaire de transaction, d'un engin JSF 2.0 et d'un conteneur JNDI à l'intérieur duquel les sources de données JDBC peuvent être rapidement configurées et exposées. De plus, JBoss 7 comprend la bibliothèque WSS4J, utilisée pour appliquer et traiter la sécurité XML.

Au moment de la rédaction de ce travail, le seul fournisseur d'identité, ou STS, disponible gratuitement, offrant une solution de SSO et supportant les jetons SAML est le produit à code ouvert OpenSSO, initialement développé par la compagnie Sun. Cependant, depuis l'acquisition de cette dernière par *Oracle*, le support au produit OpenSSO est gardé minimal, car Oracle possède déjà une suite de produits de gestion d'identités, soit la suite « *Oracle Identity Management* ». Ainsi, nous utilisons plutôt le produit OpenAM, une branche d'OpenSSO développée et maintenue par la compagnie norvégienne RockForge.

Finalement, en raison de sa simplicité et pour sa portabilité, nous utilisons la base de données Derby, codée entièrement en Java, pour effectuer et tester nos transactions XA.

Le Tableau 3 résume les technologies que nous avons considérées et essayées avant de les exclure.

Tableau 3 – Technologies d'implémentation explorées, mais exclues de la solution

Technologie	Explication
Langage de programmation	
C#	Bien que C# semble offrir une solide implémentation des spécifications nécessaires, ses outils sont payants et notre expérience avec ce langage est limitée.
Serveur applicatif	
Tomcat	Tomcat n'est pas une implémentation complète de la spécification JEE. Ainsi, à la base, il n'offre pas de mécanisme pour exposer des services web ni pour gérer des transactions.
Glassfish 3	Bien que le serveur Glassfish 3 implémente entièrement la norme JEE et qu'il présente un temps de démarrage impressionnant, nous avons remarqué des problèmes de stabilité, d'incohérence et de non-fonctionnalité lors de nos tests préliminaires.
JBoss 4,5,6	Les versions 4 et 5 de JBoss ne supportaient pas la totalité des fonctionnalités dont nous avions besoin pour implémenter notre preuve de concept, notamment au niveau des services web. La version 6, quant à elle, présentait un temps de démarrage trop grand.
Solution de service de jetons de sécurité et de gestion d'identité	
OpenSSO	Le support de la part d'Oracle est insuffisant pour ce produit. De plus, nous croyons que ce produit pourrait être laissé de côté par Oracle puisque la compagnie possède déjà une suite complète de produits de gestion d'identité, soit la suite « *Oracle Identity Management* ».

4.2.2 Intégration

D'abord, l'interface web exposée par notre portail est sécurisée à l'aide d'agents fournis avec le projet OpenAM. Ces derniers respectent le principe d'intercepteur, présenté à la section 2.4.2, et leur configuration est centralisée à l'intérieur d'OpenAM. Ils implémentent les filtres JEE et vérifient l'authentification des utilisateurs du portail.

Pour ce faire, un agent vérifie si l'utilisateur est authentifié en regardant s'il possède un cookie SSO et valide ce jeton auprès d'OpenAM. Si la vérification échoue, l'utilisateur est redirigé vers la page d'authentification d'OpenAM, où il doit saisir son identité. Une fois authentifié, OpenAM crée pour cet utilisateur une session identifiée par une chaîne de caractères unique et un cookie de cette valeur. Le cookie est retourné à l'utilisateur à l'intérieur d'une réponse http de redirection vers l'adresse originale, pour tenter une nouvelle fois de franchir l'agent. Si l'authentification réussit, le portail exécute la suite de la requête en faisant appel aux services SOAP.

La sécurisation des requêtes SOAP est assurée à l'aide d'intercepteurs JAX-WS, développés dans le cadre de ce projet à l'aide du projet WSS4J. Plutôt que de créer une extension des intercepteurs propre à JBoss, nous basons notre développement sur la spécification JAX-WS afin d'augmenter la portabilité de notre solution. La séquence des interactions entre les composants prenant part à la sécurisation d'une requête, depuis un client SOAP à un service SOAP, est illustrée à la Figure 31. Elle débute par l'interception de la requête du client par notre intercepteur de requêtes « sortantes ». Celui-ci demande à OpenAM, à l'aide de l'implémentation du protocole WS-T fourni par OpenAM, de lui créer un jeton SAML afin de certifier l'identité de l'utilisateur[4] et son authentification. Cette étape de requête de jeton de sécurité est appelée RST (« *Request Security Token* ») et le retour est appelé RSTR (« *Request Security Token Response* »). Ce jeton est inséré dans la requête originale puis, avant de la relayer au service destinataire, au besoin, l'intercepteur de la requête « sortante » signe et / ou chiffre cette dernière. À son tour, l'intercepteur de requêtes « entrantes » intercepte la requête et vérifie sa sécurité en déchiffrant d'abord les portions du message qui lui sont destinées, en validant au besoin la signature du message et celle du jeton SAML et en validant

[4] Dans ce scénario, l'intercepteur de requêtes « sortantes » se porte garant de l'identité de l'utilisateur associée à la requête puisque, dans ce cas, le jeton SAML certifie uniquement que l'identité qu'il contient est authentifiée et non que cette dernière est l'auteur de la requête présente. Ceci est le scénario SAML « *Sender Vouches* » [9], qui correspond au scénario 2 de la Figure 12.

l'autorisation de l'utilisateur auprès d'EB^{3}SEC. Une fois ces conditions vérifiées, l'intercepteur démarre une transaction, relaie la requête à l'implémentation du service, met à jour l'historique de EB^{3}SEC, commet la transaction et, optionnellement, sécurise la réponse.

Le standard WS-Addressing est utilisé pour nommer de façon uniforme les services et leurs opérations dans les WSDL. Il est configuré à l'aide des fonctionnalités offertes par JAX-WS. Cette nomenclature est automatiquement ajoutée dans les en-têtes des requêtes SOAP et est consommée par un intercepteur additionnel afin de fournir cette information au connecteur d'autorisation EB^{3}SEC.

Le répertoire des utilisateurs, leurs permissions ainsi que leurs attributs sont conservés à l'intérieur d'un répertoire LDAP (« *Lightweight Directory Access Protocol* »). Afin de simplifier le travail, le répertoire LDAP de Sun, intégré à l'outil OpenAM, est utilisé.

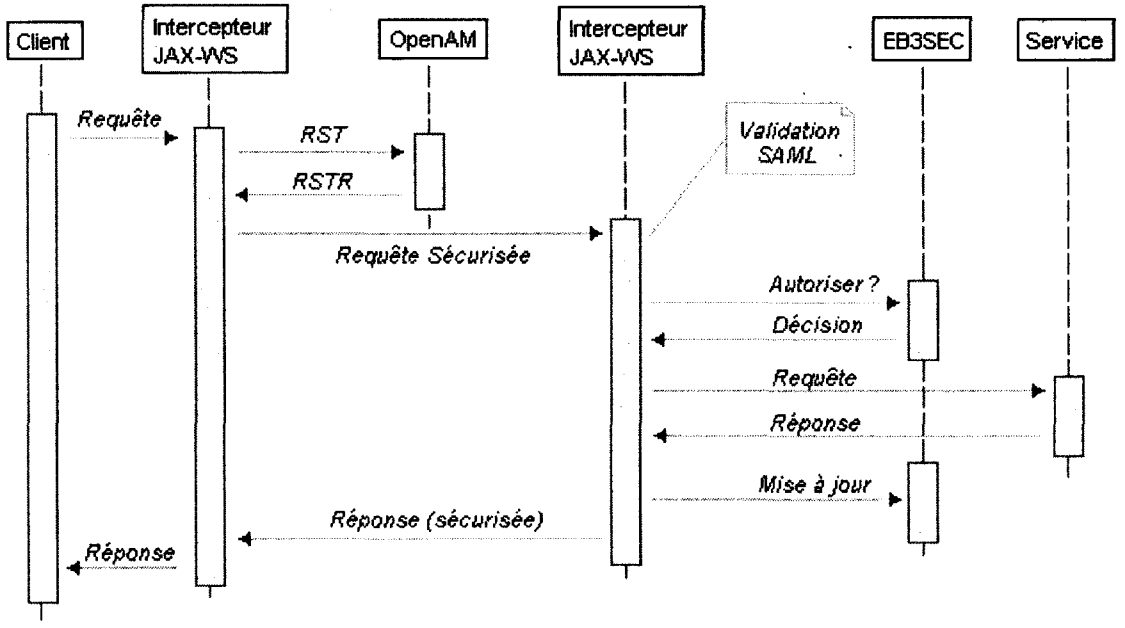

Figure 31 – Séquences des interactions lors de la sécurisation d'une requête SOAP

4.3 Évaluation

Nous évaluons notre solution autour de quatre des cinq attributs de qualité d'un logiciel qui ont été priorisés par le consortium CISQ (*Consortium for IT software quality*), soit la sécurité, la fiabilité, la maintenabilité et l'efficacité (ou performance). Nous ignorons le cinquième attribut, la taille du code, car notre réalisation est de petite taille.

4.3.1 Sécurité

Dans notre cas, la sécurité correspond aux besoins fonctionnels de base. De ce point de vue, la solution permet effectivement l'application des politiques de sécurité d'EB^{3}SEC aux trois scénarios présentés à la Figure 6. D'abord, l'utilisation d'un fournisseur central d'identités sous la forme de jetons SAML permet, dans un premier temps, de propager l'identité de l'utilisateur dans une chaîne de services et, dans un deuxième temps, d'offrir un mécanisme d'intégration standard aux partenaires externes de l'entreprise. Ensuite, la mise en place d'intercepteur capable d'appliquer et de traiter la sécurité au niveau des messages assure la confidentialité et l'intégrité de ces messages jusqu'au destinataire final. Finalement, la création d'une interface pour un connecteur éventuel à EB^{3}SEC permet de simuler l'intégration à ce dernier.

Nous notons toutefois que l'accès à OpenAM via le protocole HTTPS (plutôt que HTTP) n'a pas été configuré, et donc, que les communications entre les utilisateurs et OpenAM, lors de l'authentification, ne sont pas sécurisées, ce qui constitue une lacune de sécurité importante. Finalement, OpenAM consulte un LDAP afin d'authentifier les utilisateurs. L'intégration de ces deux composantes à EB^{3}SEC pourrait être problématique si l'implémentation d'EB^{3}SEC possède son propre mécanisme pour conserver le répertoire des utilisateurs.

4.3.2 Fiabilité

En termes de fiabilité, le code développé est simple, respecte les principes de programmation orientée objet et est basé sur les standards et les outils JEE. Cependant, ce code a été développé rapidement et les gestions d'erreurs sont généralement incomplètes, sinon absentes. Dans le même ordre d'idées, les tests ont été gardés

minimaux et les cas d'utilisation supportés sont limités. Par exemple, une seule version des jetons SAML est supportée et le protocole HTTPS n'a pas été configuré. Néanmoins, ces faiblesses pourraient toutes être corrigées par une revue attentive du code et par sa mise à jour. Par opposition, le filtre d'OpenAM ne supporte pas encore officiellement la version 7 de JBoss, ce qui peut potentiellement freiner l'adoption de notre solution par une entreprise. Ensuite, le recours fréquent aux transactions distribuées (chaque fois qu'un service met à jour l'état du système) est discutable, car le verrouillage d'une ressource centralisée (l'historique) pour pratiquement chaque requête pendant tout le temps d'exécution d'un service est susceptible de causer une dégradation de la performance globale du système. Finalement, les transactions à deux temps (« *two phase commit* ») XA, retenues pour implémenter les transactions entre les services et l'historique, ne garantissent pas toujours la synchronisation des deux systèmes.

4.3.3 Maintenabilité

Bien que la solution soit entièrement écrite en Java, qu'elle soit petite et qu'elle soit basée sur les plus récents standards JEE et WS, nous croyons qu'elle est difficile à maintenir. Cette difficulté s'explique par la complexité et la lourdeur des outils disponibles pour implémenter un service web et ses standards. D'abord, notre projet compte plus d'une dizaine de fichiers et d'écrans de configurations. La plupart des modifications à ces fichiers, comme au code source, demandent un redémarrage du serveur applicatif. Dès lors, les séances d'essais-erreurs deviennent longues et ardues. De plus, les outils utilisés pour sécuriser les requêtes SOAP sont eux-mêmes basés sur plusieurs autres outils à code source libre dont le support est très limité et la documentation, pour la majorité, est incomplète, voir inexistante. Cette réalité nous oblige à recourir à un débogueur pour trouver et corriger les problèmes.

4.3.4 Efficacité

Afin d'évaluer la performance de la solution, nous avons mesuré le temps d'exécution moyen de 100 requêtes pour passer au travers un scénario équivalent au cas B de la Figure 6. Ce scénario implémente un cas d'utilisation où un utilisateur place, à partir de l'interface web, une réservation pour un livre en soumettant son numéro de client et le

numéro du livre à louer. L'interface web effectue d'abord une requête de location de livre au service « *BookService* ». Ce dernier consulte d'abord le service « *ClientService* » pour savoir si le numéro de client fourni est valide, valide lui-même le numéro du livre et insère la transaction dans la base de données corporative. Ce scénario comprend deux lectures et une écriture dans la base de données corporative. Chacune des deux requêtes à un service est sécurisée à l'aide de la séquence d'interactions présentée à la Figure 31, se traduisant notamment par deux lectures et deux écritures supplémentaires dans la base de données d'historique.

Nous avons calculé le temps d'exécution moyen de 100 requêtes ainsi que le temps passé à l'intérieur de ces requêtes à effectuer des demandes de jetons de sécurité. Nous avons répété cet exercice en faisant varier, de 1 à 20, le nombre de ces requêtes exécutées en parallèle. Nous avons effectué ces jeux de tests avec et sans la présence de transactions XA, afin d'évaluer l'impact de ces dernières. Pour exécuter ces tests, nous avons eu recours à l'outil « ab » (*Apache Benchmark*), disponible avec le serveur web « httpd » d'*Apache*. Les résultats ainsi que la configuration du serveur et de l'ordinateur utilisé sont présentés aux

Tableau 4, Tableau 5 et Tableau 6. Nous notons que tous les composants exécutaient sur la même machine et que ces requêtes ne résultaient qu'en une simple page HTML.

Tableau 4 – Configurations du serveur de test

Processeur	Intel® Core™ 2 Duo CPU T7500 @ 2.20 GHz
Mémoire vive	2 Gigaoctets
Système d'exploitation	Microsoft Windows XP Professionel
Machine virtuelle Java	Sun JDK 1.6.0_26
Serveur Applicatif	RedHat JBoss 7

Tableau 5 - Temps d'exécution total moyen, en seconde, d'une requête, dans un contexte de transactions XA, pour différents niveaux de concurrence avec n = 100

	1	2	4	10	20
Temps Total	0,517	0,675	1,232	2,715	6,375
Temps RST	0,315	0,441	0,823	2,102	5,081
% RST	61 %	65 %	66 %	77 %	80 %

Tableau 6 – Temps d'exécution total moyen, en seconde, d'une requête, sans transactions XA, pour différents niveaux de concurrence avec n = 100

	1	2	4	10	20
Temps Total	0,433	0,678	1,190	2,330	5,418
Temps RST	0,299	0,535	0,940	1,970	4,614
% RST	69 %	79 %	79 %	85 %	85 %

Le premier élément qui ressort de ces résultats est que, vu la simplicité de notre logique d'affaires et de notre modèle de données, le délai d'exécution des requêtes, principalement dû aux deux requêtes de jetons de sécurité (RST), est grand. Nous croyons que le délai des RST est attribuable à la taille des jetons SAML, aux opérations de signature et aux opérations d'analyse et de formatage du code XML. Un exemple de jeton SAML est présenté à l'Annexe D. Nous proposons deux optimisations à notre réalisation afin d'améliorer ce délai. Premièrement, nous suggérons de conserver en mémoire cache, pendant la durée de leur vie, les jetons de sécurité aux fins de réutilisation. Ceci diminuerait le nombre d'appels au STS qui ne servent qu'à convertir un jeton de sécurité à un autre (cookie vers SAML) ou encore à en obtenir un neuf. Deuxièmement, nous proposons de limiter l'utilisation de jetons SAML aux communications avec les services partenaires ou avec les applications de tierces parties et de plutôt opter pour un type de jeton léger à l'interne, possiblement propriétaire, tel que le LTPA (*Lightweight Third-Party Authentification*) d'IBM. Ainsi, notre solution continuerait de tirer profit du caractère « spécification standardisée » de SAML pour intégrer rapidement les plateformes hétérogènes et connaîtrait un gain de performance au

niveau des communications internes. Le traitement d'un jeton propriétaire serait effectué à l'aide d'intercepteurs propriétaires.

Le deuxième élément qui ressort de nos résultats est la différence entre les pourcentages du temps d'attente des RST selon la présence ou non de transactions XA. On voit que ces pourcentages sont légèrement plus élevés lorsqu'on retire les transactions XA et, donc, que ces dernières impactent la performance du système. Curieusement, selon nos résultats, l'importance de l'impact des transactions XA tend à diminuer lorsque nous augmentons le niveau de concurrence. Nous croyons que ce phénomène est simplement dû au fait que le projet OpenAM souffre beaucoup plus que les bases de données qui, elles, sont spécialisées dans la gestion de la concurrence. À priori, on peut supposer que les limites de notre solution sont au niveau des technologies SOA plutôt qu'au niveau du projet EB^{3}SEC. Cependant, les modèles de données utilisées ont été gardés simples et nous recommandons une étude plus poussée sur l'impact réel des transactions distribuées à l'intérieur d'un système plus complexe. Dans l'éventualité où ces dernières limiteraient la performance de la solution, nous proposons l'étude des mécanismes de compensations. Ces derniers sont un compromis aux transactions distribuées en ce sens qu'ils proposent d'éviter de verrouiller les ressources dépendantes et de plutôt prévoir une série d'opérations pour effacer les traces de la transaction en cas d'échec de cette dernière. Cette approche n'est cependant pas toujours possible, car elle implique que des ressources puissent être temporairement désynchronisées.

Conclusion

Contribution

Nous avons montré à travers ce travail que le projet EB^{3}SEC, développé par l'équipe du GRIL de l'Université de Sherbrooke et ayant trait à la sécurisation des processus d'affaires d'une entreprise, peut être intégré à l'intérieur d'une infrastructure moderne de sécurité orientée service où le contrôle d'accès peut être décrit par le modèle XACML. Cependant, cette intégration a demandé une revue du modèle XACML afin qu'il tienne compte du besoin de EB^{3}SEC d'avoir accès à l'historique des actions posées sur la ressource accédée pour prendre une décision d'autorisation. Pour ce faire nous avons ajouté un point de consultation de l'historique des actions et également un point de mise à jour de cet historique afin d'assurer la cohérence de son état avec celui du système d'information. Puis, nous avons déterminé que, dans le cadre du projet EB^{3}SEC, la gestion de l'historique devait être faite de manière explicite en raison de la fine granularité de ses politiques de sécurité et de manière centralisée afin d'en faciliter la consultation. Ensuite, nous avons montré que le projet EB^{3}SEC devait être implémenté à l'aide d'une infrastructure basée sur le modèle d'autorisation purement centralisé, notamment en raison de la complexité que représenterait un déploiement de EB^{3}SEC sur chacun des services de l'entreprise, de la particularité d'EB^{3}SEC de baser certaines de ses décisions sur l'historique des actions des utilisateurs et de la fine granularité des politiques d'accès que supporte EB^{3}SEC. Finalement, nous avons établi que la synchronisation systématique de l'historique des actions des utilisateurs avec l'état du système nécessite le recours aux transactions distribuées.

Dans un autre ordre d'idées, en implémentant l'architecture de référence, nous avons montré que, contrairement aux promesses faites par les fournisseurs de produits et d'outils SOA, l'implémentation et la sécurisation d'une architecture orientée services

sont des tâches complexes et ardues et qu'elle a résulté en une solution gourmande en ressources. En effet, suite à des tests de charge composés de 100 requêtes exécutées parallèlement en groupe de 20, les résultats ont indiqué un temps moyen de 5 secondes pour sécuriser une requête web traversant deux services SOAP. Toutefois, nous pensons que cette mauvaise performance est principalement due à trois facteurs. Premièrement, la taille considérable des jetons de sécurité que nous avons choisis, soit les jetons SAML, a pour conséquence d'alourdir leur génération et leur traitement. Deuxièmement, la mauvaise gestion que nous avons faite de ces mêmes jetons, c'est-à-dire de ne jamais réutiliser un jeton encore valide (non expiré), a eu pour conséquence que chaque appel à un service a nécessité une nouvelle demande de génération de jeton. Dernièrement, nous croyons que nous avons atteint la capacité maximale de notre instance unique de notre STS, OpenAM.

Finalement, nous avons montré que les transactions distribuées nécessaires pour EB^{3}SEC impactent la performance du système, mais que cet impact n'était pas considérable dans notre architecture.

Critique du travail

Le projet EB^{3}SEC n'était pas terminé au moment de nos travaux. Ainsi, ce dernier a été simplement simulé à l'aide d'un SGBD. Les résultats sont donc basés sur un système incomplet.

Le code a été développé rapidement et quelques optimisations simples ont été oubliées. Ainsi, l'implémentation d'une stratégie de réutilisation des jetons de sécurité aurait permis d'améliorer les statistiques de temps moyen des requêtes. De plus, la gestion des erreurs n'est pas complète, ce qui peut compliquer sa réutilisation par l'équipe du GRIL.

L'utilisation du fournisseur d'identité OpenAM de RockForge, dérivé du projet OpenSSO de Sun, représente un autre point faible de notre solution. En effet, ce dernier n'est pas performant, est difficile à installer et à configurer et ne supporte pas les versions

les plus récentes des serveurs applicatifs JEE. Nous croyons que l'utilisation d'un fournisseur d'identité commercial aurait présenté une meilleure performance.

Finalement, les modèles de données manipulés à l'intérieur des transactions XA sont très simples et ne représentent pas bien la réalité. Ainsi, il est possible que les transactions XA aient un impact plus important sur la performance du système que celui mesuré dans ce travail.

Travaux futurs

Bien entendu, d'éventuels travaux seront nécessaires afin d'apporter les optimisations suggérées et afin d'intégrer le projet EB^{3}SEC à notre réalisation, une fois ce dernier terminé. De cette façon, son impact réel sur la performance du système, notamment au niveau des transactions XA, pourra être évalué. L'étude d'un mécanisme de compensation en remplacement du mécanisme de transaction distribué serait également intéressante.

Annexe A – Exemples de requêtes REST

Cette annexe présente deux exemples de requêtes REST. La première, au Code 1, en est une de lecture où un client demande les informations de l'utilisateur « Robert », et la deuxième, au Code 2, en est une de mise à jour, où un client met à jour les informations de l'utilisateur « Robert ». On note que le mode de ces requêtes (consultation ou mise à jour) est reconnaissable grâce aux mots clés HTTP GET et POST. On note aussi que le corps des requêtes et des réponses peuvent être de différents formats. Ces derniers sont généralement spécifiés dans les en-têtes HTTP « Accept » et « Content-Type », mais ne sont pas obligatoires. Finalement, on note que la première requête peut facilement être générée à partir d'un fureteur.

```
GET /users/Robert HTTP/1.1
Accept : application/xml
```

Code 1 – Requête de lecture REST

```
POST /users/u1234 HTTP/1.1
Content-Type: application/xml
<?xml version="1.0"?>
<user>
  <userid>u1234</userid>
  <lastname>Morrissette</lastname>
  <firstname>Robert</firstname>
</user>
```

Code 2 – Requête d'insertion REST

Annexe B – Exemple d'un document WSDL avec *WS-Policy, WS-PolicyAttachment, WS-SecurityPolicy*

Cette annexe présente au Code 3 un exemple de document WSDL décrivant un service SOAP de cotations et auquel des politiques de sécurité *WS-SecurityPolicy* ont été attachées à l'aide des spécifications *WS-Policy* et *WS-PolicyAttachment.* Les politiques de sécurité sont présentes dans le premier bloc de code apparaissant en caractères gras. L'attribut « wsu:Id » de la balise racine « wsp:policy » est un identificateur unique permettant d'identifier cette politique et de lui faire référence plus loin dans le document. La balise suivante, « sp:SymmetricBinding », fait référence à un des trois profils de sécurité définis par la spécification *WS-SecurityPolicy.* Elle indique que la communication doit être protégée à l'aide de la spécification WS-Security et que cette dernière doit utiliser des clés symétriques. La balise « sp:ProtectionToken » décrit le type de jeton de sécurité autorisé. La balise « sp:IssuedToken » indique que le jeton doit être obtenu d'un fournisseur de jeton et son attribut « sp :IncludeToken=".../IncludeToken/Once" » spécifie que ce jeton ne doit être inclus qu'une seule fois dans le message. La balise « sp:issuer » contient les informations du fournisseur de jetons. La balise « sp:SignBeforeEncrypting » indique que le message doit être signé avant d'être crypté. La balise « sp:EncryptSignature » signifie que la signature doit être chiffrée. Finalement, les balises « sp:SignedPart » et « sp :EncryptedPart » permettent respectivement de spécifier quelles parties du message sont signées, ici le corps de la requête ainsi que l'entête d'adressage, et quelles parties sont chiffrées, ici le corps de la requête. Cette politique de sécurité est attachée au point d'entrée « quoteServiceBinding » à l'aide de la dernière balise en caractères gras « wsp:PolicyReference ».

```
<wsdl:definitions name="QuoteService"
  targetNamespace="http://impl.service.usherbrooke.ca/"
  xmlns:ns1="http://schemas.usherbrooke.ca/service/quoteService.xsd"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://impl.service.usherbrooke.ca/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-utility-1.0.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">

  <!-- Politiques de sécurité -->
  <wsp:Policy wsu:Id="maPolitique">
    <sp:SymmetricBinding>
      <wsp:Policy>
        <sp:ProtectionToken>
            <sp:IssuedToken
              sp:IncludeToken=".../IncludeToken/Once" >
            <sp:Issuer>...</sp:Issuer>
        </sp:ProtectionToken>
        <sp:SignBeforeEncrypting />
        <sp:EncryptSignature />
      </wsp:Policy>
    </sp:SymmetricBinding>
    <sp:SignedParts>
      <sp:Body/>
      <sp:Header
Namespace="http://schemas.xmlsoap.org/ws/2004/08/addressing" />
    </sp:SignedParts>
    <sp:EncryptedParts>
      <sp:Body/>
    </sp:EncryptedParts>
  </wsp:Policy>

  <!-- Types XML -->
  <wsdl:types>
    <xs:schema attributeFormDefault="unqualified"
      elementFormDefault="unqualified"

targetNamespace="http://schemas.usherbrooke.ca/service/quoteService.xsd"
      xmlns="http://schemas.usherbrooke.ca/service/quoteService.xsd"
      xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:import
namespace="http://schemas.usherbrooke.ca/service/quoteService.xsd"
          schemaLocation="quoteService.xsd" />
    </xs:schema>
  </wsdl:types>

  <!-- Messages -->
  <wsdl:message name="UnknownProductException">
    <wsdl:part element="ns1:ProductException"
      name="ProductException">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="getQuote">
```

```
    <wsdl:part element="ns1:getQuote" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="getQuoteResponse">
    <wsdl:part element="ns1:getQuoteResponse" name="parameters">
    </wsdl:part>
  </wsdl:message>

  <!-- Port du service (interface) -->
  <wsdl:portType name="QuoteService">
    <wsdl:operation name="getQuote">
      <wsdl:input message="tns:getQuote" name="getQuote">
      </wsdl:input>
      <wsdl:output message="tns:getQuoteResponse"
        name="getQuoteResponse">
      </wsdl:output>
      <wsdl:fault message="tns:ProductException"
        name="ProductException">
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:portType>

  <!-- Définition du point d'entré -->
  <wsdl:binding name="quoteserviceSoapBinding" type="tns:QuoteService">

    <!-- Attachement de la politique au point d'entré -->
    <wsp:PolicyReference URI="#maPolitique" wsdl:required="true" />

    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http" />

    <wsdl:operation name="getQuote">
      <soap:operation soapAction="" style="document" />
      <wsdl:input name="getQuote">
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output name="getQuoteResponse">
        <soap:body use="literal" />
      </wsdl:output>
      <wsdl:fault name="UnknownProductException">
        <soap:fault name="UnknownProductException" use="literal" />
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>

  <!-- Localisation du service -->
  <wsdl:service name="QuoteWSService">
    <wsdl:port binding="tns:quoteserviceSoapBinding" name="QuoteWSPort">
      <soap:address location="http://server-
b.usherbrooke.ca:8080/quoteService/spring-ws/quoteservice" />
    </wsdl:port>
  </wsdl:service>

</wsdl:definitions>
```

Code 3 – Exemple de document WSDL auquel sont attachées des politiques *WS-Policy*

Annexe C – Installation et configuration d'OpenAM

Cette annexe s'adresse au lecteur désirant installer l'application OpenAM et décrit comment cette étape a été effectuée dans le cadre de la réalisation d'un exemple d'architecture de sécurité SOA pour le projet EB^{3}SEC.

OpenAM est offert sous le format d'archive web JEE (WAR) déployable dans les conteneurs implémentant la version égale ou supérieure à 2.3 des « *servlets* ». Contrairement au reste de notre solution, nous avons choisis le conteneur Tomcat 7 de Apache en raison de sa simplicité, sa légèreté et parce qu'il présente toutes les fonctionnalités que requière OpenAM. Avant de déployer l'application, il faut s'assurer que toutes les adresses devant être protégées par OpenAM sont complètement qualifiées avec le même nom de sous-domaine. Par exemple, nous avons nommé les domaines de notre système « sso.usherbrooke.ca », « portal.usherbrooke.ca » et « services.usherbrooke.ca ». Cette uniformité des noms de sous-domaine permet le partage des cookies entre les différents domaines et est essentielle au fonctionnement d'OpenAM. Ensuite, tel que recommandé dans le manuel d'utilisation de OpenAM, la taille de la mémoire maximale allouée à la machine virtuelle de Java a été augmentée en ajoutant la ligne de code présentée au Code 4 dans le fichier « startup.bat » de Tomcat. L'application a ensuite été déployée en copiant l'archive à l'intérieur du répertoire « webapps » de Tomcat.

```
set JAVA_OPTS=-Xmx1024M -XX:MaxPermSize=128M
```

Code 4 – Ajout de variables systèmes pour augmenter la taille maximale de la mémoire allouée à la JVM de *Tomcat*

Une fois l'archive déployée, l'installation *d'OpenAM* a été complétée à l'aide de son interface web, accessible par le nom de domaine complètement qualifié de son hôte, dans ce cas http://sso.usherbrooke.ca:8080/opensso/. C'est dans cette étape qu'on peut spécifier le répertoire d'utilisateurs LDAP à partir duquel OpenAM authentifiera les utilisateurs. Nous notons que cette étape ne fonctionne pas toujours du premier coup.

Dans le cas où une erreur survient, nous recommandons la consultation du journal d'installation, situé dans le répertoire de configuration nouvellement créé pour OpenAM. Si l'information contenue dans ce journal n'est pas suffisante, nous suggérons d'éteindre Tomcat, de supprimer les répertoires du déploiement de l'archive web et de configuration, de redémarrer *Tomcat* et de reprendre l'installation, mais cette fois avec un autre fureteur si possible.

Toujours à partir de son interface web, nous avons configuré OpenAM de façon à garder les communications simples. Ainsi, le service de jetons de sécurité à premièrement été configuré de manière à authentifier les demandes de jetons à l'aide d'une simple combinaison, codée en dure, d'identifiant et de mot de passe. Cette configuration peut être réalisée en suivant les menus *Configurations -> Globals -> Security Token Service*, puis en cochant la case *UsernameToken-Plain* dans la section *Security* et en spécifiant le nom et le mot de passe sous la forme du Code 5 dans la boîte de texte nommée *User Identification Information.* Deuxièmement, une configuration centralisée destinée aux applications qui communiquent avec le service de jeton de sécurité à l'aide de la bibliothèque cliente offerte par OpenAM (openssosdk.jar) a été créée afin que ces applications utilisent le mécanisme d'authentification approprié pour demander un jeton, soit une combinaison identifiant et mot de passe. Cette configuration peut à son tour être réalisée en suivant les menus *Access Controls -> / -> Agents -> STS Client* et en répliquant la configuration du service de jetons de sécurité. Troisièmement, une configuration centralisée destinée aux applications web protégées par le filtre JEE de OpenAM, dans notre cas le portail, a été créée par les menus *Access Controls -> / -> Agents -> JEE.* Quatrièmement, à partir du menu *Access Controls ->Policies*, des politiques de sécurité ont été créées pour permettre à tous les utilisateurs authentifiés d'accéder aux URLs du portail. Finalement, un utilisateur a été créé à partir du menu *Access Controls ->Subjects.* Nous notons que ces politiques et ces utilisateurs sont en réalité conservés dans le répertoire LDAP sous-jacent.

```
UserName:xxxx|UserPassword:xxxx
```

Code 5 – Syntaxe pour coder en dure un nom jeton « nom et mot de passe » dans OpenAM.

Annexe D – Exemple d'un jeton SAML

Le Code 6 présente la taille impressionnante d'un jeton SAML.

```
<?xml version='1.0' encoding='UTF-8'?>
<saml:Assertion xmlns="" xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
AssertionID="uuid-20702747-d1d7-4f6e-a639-a45829891029"
IssueInstant="2011-08-07T17:42:43Z"
Issuer="SunSTS" MajorVersion="1" MinorVersion="1">
  <saml:Conditions NotBefore="2011-08-07T17:42:43Z" NotOnOrAfter="2011-08-07T17:47:43Z">
    <saml:AudienceRestrictionCondition>
      <saml:Audience>http://server-a.agb.com:8080/services/bookService</saml:Audience>
    </saml:AudienceRestrictionCondition>
  </saml:Conditions>
  <saml:AuthenticationStatement AuthenticationInstant="2011-08-07T16:00:06Z"
    AuthenticationMethod="urn:com:sun:identity:Application">
    <saml:Subject>
      <saml:NameIdentifier>
        id=beaupral,ou=user,dc=opensso,dc=java,dc=net
      </saml:NameIdentifier>
      <saml:SubjectConfirmation>
      <saml:ConfirmationMethod>
        urn:oasis:names:tc:SAML:1.0:cm:sender-vouches</saml:ConfirmationMethod>
      </saml:SubjectConfirmation>
    </saml:Subject>
  </saml:AuthenticationStatement>
  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo>
      <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
      <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
      <ds:Reference URI="#uuid-20702747-d1d7-4f6e-a639-a45829891029">
        <ds:Transforms>
          <ds:Transform
          Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
          <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
        </ds:Transforms>
        <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
        <ds:DigestValue>dKyNuC+Nz5J9tH8KVAjB17df7ZY=</ds:DigestValue>
      </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>
jhyUpgaV6vptSULy83uT99evzpCwCLbT9y1kmM2+DRXKNwEKc5mEjdLAuKPU8SwxqP9OgN2B+imf
DvgmwAI350ENUnWCMyras9xmRtRaoTBQNEtEdCu8EvWW2boXvQw2N0QPqTGF8PgXn41deuRAlOet
xYqIp3gmgmjH3msamjA=
    </ds:SignatureValue>
    <ds:KeyInfo>
      <ds:X509Data>
        <ds:X509Certificate>
MIICQDCCAakCBEeNB0swDQYJKoZIhvcNAQEEBQAwZzELMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNh
bGlmb3JuaWExFDASBgNVBAcTC1NhbnRhIENsYXJhMQwwCgYDVQQKEwNTdW4xEDAOBgNVBAsTB09w
ZW5TU08xDTALBgNVBAMTBHRlc3QwHhcNMDgwMTE1MTkxOTM5WhcNMTgwMTEyMTkxOTM5WjBnMQsw
CQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZvcm5pYTEUMBIGA1UEBxMLU2FudGEgQ2xhcmExDDAK
BgNVBAoTA1N1bjEQMA4GA1UECxMHT3B1b1NTTzENMAsGA1UEAxMEdGVzdDBnzANBgkqhkiG9w0B
AQEFAAOBjQAwgYkCgYEArSQc/U75GB2AtKhbGS5piiLkmJzqEsp64rDxbMJ+xDrye0EN/q1U5Of+
RkDsaN/igkAvV1cuXEgTL6RlafFPcUX7QxDhZBhsYF9pbwtMzi4A4su9hnxIhURebGEmxKW9qJNY
Js0Vo5+IgjxuEWnjnnVgHTs1+mq5QYTA7E6ZyL8CAwEAATANBgkqhkiG9w0BAQQFAAOBgQB3Pw/U
QzPKTPTYi9upbFXlrAKMwtFf2OW4yvGWWv1cwcNSZJmTJ8ARvVYOMEVNbsT4OFcfu2/PeYoAdiDA
cGy/F2Zuj8XJJpuQRSE6PtQqBuDEHjjmOQJ0rV/r8mO1ZCtHRhpZ5zYRjhRC9eCbjx9VrFax0JDC
/FfwWigmrW0Y0Q==
        </ds:X509Certificate>
      </ds:X509Data>
    </ds:KeyInfo>
  </ds:Signature>
</saml:Assertion>
```

Code 6 – Exemple d'un jeton SAML

Bibliographie

[1] Britton, P., Bye, P. : *IT architectures and middleware : strategies for building large, integrated systems*. Addison-Wesley, 2004.

[2] Object Management Group : CORBA 1.0. http://www.omg.org/spec/CORBA/1.0/

[3] Erl, T. : *SOA : principles of service design*. Prentice Hall, 2008.

[4] Erl, T. : *SOA design patterns*. Prentice Hall, 2009.

[5] Fielding, R. T. : Architectural Styles and the Design of Network-based Software Architectures. Dissertion, Information and Computer Science, University of California, Irvine, USA. 2000.

[6] Gartner : Gartner says SOA will be used in more than 50% of new mission-critical operational applications and business processes designed in 2007. http://www.gartner.com/it/page.jsp?id=503864

[7] Kanneganti, R., Chodavarapu, P. : *SOA security*. Manning, 2008.

[8] Teo, L. K. Y., Teh, D. W., Corbitt, B. : Service Oriented Architecture (SOA): Implications for Australian university information systems curriculum. Pacific-Asia Conference on Information Systems Proceedings. 2010.

[9] Natis, Y. V. : Service-Oriented Architecture Scenario. Gartner. 2003. www.gartner.com/resources/114300/114358/serviceoriented_architecture_114358.pdf

[10] Oracle Base : Auditing in Oracle 10g Release 2. http://www.oracle-base.com/articles/10g/Auditing_10gR2.php#audit_options

[11] Organization for the Advancement of Structured Information Standards : Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0. http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf

[12] Organization for the Advancement of Structured Information Standards : Web Service Security : SOAP Message Security 1.0. http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf

[13] Organization for the Advancement of Structured Information Standards : SOA reference model. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm

[14] Organization for the Advancement of Structured Information Standards : eXtensible Authorization Control Markup Language (XACML). http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf

[15] Rademakers, T., Dirksen, J. : *ESBs in action : example implementations in Mule and ServiceMix*. Manning, 2009.

[16] Representational State Transfer, http://en.wikipedia.org/wiki/Representational_state_transfer

[17] Rosen, M., Boris, L., Smith, K., Balcer, M. : *Applied SOA : service-oriented architecture and design strategies*. Wiley, 2008.

[18] Schekkerman, J. : *Enterprise Architecture Good Practices Guide*. Trafford, 2008.

[19] The Open Group : Distributed Transaction Processing : The XA Specification. http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf

[20] Web Services Interoperability Organization : Basic Security Profile version 1.0. http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0.html

[21] World Wide Web Consortium (W3C) : Simple Object Access Protocol (SOAP). http://www.w3.org/TR/soap/

[22] World Wide Web Consortium (W3C) : Web Services Description Language (WSDL). http://www.w3.org/TR/wsdl

[23] World Wide Web Consortium (W3C) : Web Services Policy 1.5 - Framework. http://www.w3.org/TR/2007/REC-ws-policy-20070904/

[24] World Wide Web Consortium (W3C) : XML Encryption Syntax and Processing (Second Edition). http://www.w3.org/TR/xmlenc-core/

[25] World Wide Web Consortium (W3C) : XML Signature Syntax and Processing (Second Edition). http://www.w3.org/TR/xmldsig-core/