OpenAM 検証報告

OSS 戦略企画室　OSS 技術第一課

改定履歴

Version	編集日	概要
1.0.0	2012/03/26	初版作成

目次

OPENAM 検証報告 1

1. はじめに 5

1.1. 本書の位置づけ 5

1.2. 背景 5

1.3. 前提知識 6

2. 機能検証 7

2.1. OpenAM 概要 7
2.1.1. サポートする認証方式 7
2.1.2. SSO 実現方式 8
2.1.3. OpenAM のアーキテクチャ 10

2.2. 検証目的と検証内容 12
2.2.1. 検証テーマ１　エージェント方式による SSO 対応 12
2.2.2. 検証テーマ２　リバースプロキシ方式による SSO 対応 12
2.2.3. 検証テーマ３　外部サービスも含めた SSO 対応 12
2.2.4. 検証テーマ３　モバイル端末の個体認証 13

2.3. 検証環境 14

2.4. 検証結果 16
2.4.1. 検証テーマ１　エージェント方式による SSO 対応 16
2.4.2. 検証テーマ２　リバースプロキシ方式による SSO 対応 18
2.4.3. 検証テーマ３　外部サービスも含めた SSO 対応 21
2.4.4. 検証テーマ４　モバイル端末の個体認証 24
2.4.5. 機能検証項目および結果一覧 25
2.4.6. 機能検証のまとめと考察 26

3. 非機能検証 28

3.1. 性能試験 28
3.1.1. 検証目的 28
3.1.2. 検証内容 28
3.1.3. 検証環境 30
3.1.4. 検証結果 32

3.2. 可用性試験 38
3.2.1. OpenAM における冗長構成 38
3.2.2. ユーザリポジトリと負荷分散装置における冗長構成 40
3.2.3. 検証目的 41
3.2.4. 検証内容 41
3.2.5. 検証環境 44
3.2.6. 検証結果・考察 45

3.3. セキュリティに関する考察 48

3.4. OpenAM の運用面 51
3.4.1. バックアップとリカバリー 51
3.4.2. OpenAM のログ監視 52
3.4.3. ユーザ管理 52

4. まとめ 53

5. 参考情報 55

5.1. 認証基盤製品比較 55

5.2. OpenAM を使用する上での留意点 56
5.2.1. クライアント証明書認証を使う上での留意点 56
5.2.2. 自動作成された Fedlet を使用する上での留意点 56

5.3. Basic 認証 AP に対する代理認証 57

5.4. セッションフェイルオーバアーキテクチャ 58
5.4.1. Message Queue ブローカ 59
5.4.2. amsessiondb 59
5.4.3. OpenAM 60
5.4.4. 処理の流れ 60
5.4.5. セッションの操作とプロトコルの対応 61

5.5. 負荷分散装置の構成詳細 62
5.5.1. LVS (Linux Virtual Server) 62
5.5.2. keepalived 65
5.5.3. iptables を用いた L2 特殊処理 66

5.6. ssoadm コマンドをサイト構成で利用する際に発生する問題 68

5.7. 関連情報 69

1. はじめに

1.1. 本書の位置づけ

このレポートは、シングルサインオン(SSO)を実現するためのオープンソース・ソフトウェアである OpenAM について実施した検証についての報告をまとめたものです。

1.2. 背景

認証処理は、セキュリティを担保するための必要不可欠な要素であり、様々なシステム・アプリケーションに組み込まれています。しかし現状ではアプリケーションごとにログインを必要としたり、場合によってはいくつもの ID・パスワードを使い分けることを余儀なくされたりしていることも多くあります。

このような課題を解決するためには、ID・パスワードを統合する統合認証と一度のログインですべてのアプリケーション・システムを使うことを可能にする SSO を提供する認証基盤を構築することが有用です。

認証基盤を構築するための商用製品はいろいろなベンダから提供されています。しかし、それらの製品は高機能ですが、ユーザアカウント数に応じたライセンスを必要とし、大規模なシステムでは非常に高価となりコストを圧迫します。

そこで OSS を組合せて認証基盤を構築することにより、コストを抑制することが期待されます。最近では、認証基盤を構築するための OSS が存在し、その導入事例も報告されています。

認証基盤を構築する技術は、SSO の他にもユーザ情報・権限情報などを格納する統合ディレクトリ、人事システムからユーザ情報を取り込み各システムに連携する ID 管理、SSL などの暗号化基盤、電子証明書のための PKI(Public Key Infrastructure)基盤など幅広い領域にわたります。それぞれ OSS を利用することで実現可能です。
図 1 は認証基盤全体の概要を示しています。認証システムの OSS としては OpenAM、統合ディレクトリの OSS としては OpenLDAP が利用できます。

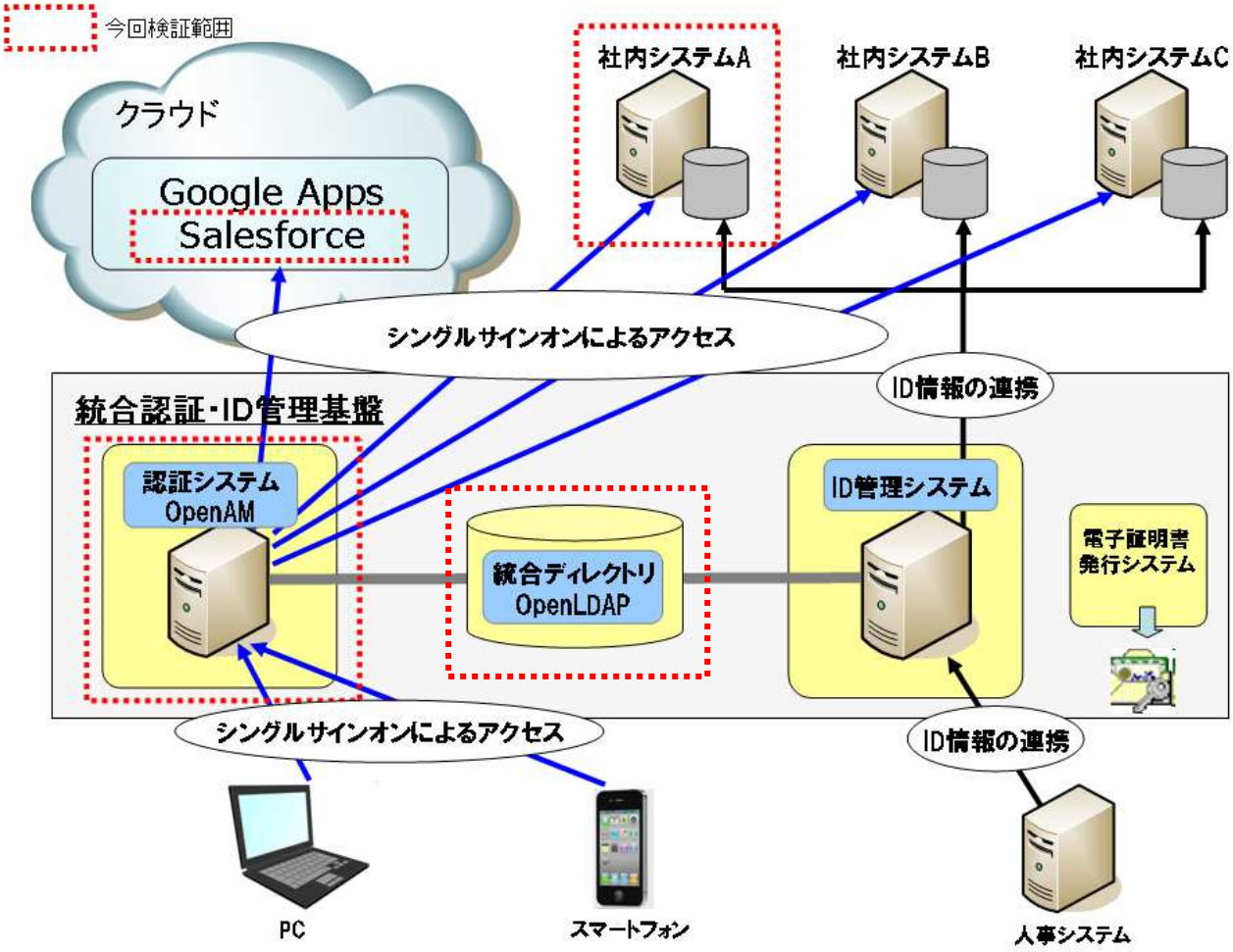

図 1. 認証基盤全体概要

このレポートでは、認証基盤の中核となる技術である SSO を中心として取り上げ、それを実現する OpenAM の機能面・非機能面に関する動作検証を行った結果を報告します。

1.3. 前提知識

本書では、認証・認可の仕様について詳しく説明しておりません。SSO、SAML、XACML、OpenID、LDAP などについての基礎的な知識が前提になります。

2. 機能検証

2.1. OpenAM 概要

OpenAM は、SSO による認証やアクセス制御を実現するためのオープンソース・ソフトウェアです。

Sun Microsystems(現 Oracle)により提供されていた OpenSSO を起源に持ち、現在は、Sun Microsystems の技術者が設立した forgerock 社により開発が進められています。

日本においても、OpenAM の発展と普及を目指し、2010 年 10 月に「OpenSSO&OpenAM コンソーシアム(www.openam.jp)」が設立されました。

2.1.1. サポートする認証方式

OpenAM では、ユーザ名・パスワードを元にしたパスワード認証をはじめ、ワンタイムパスワード認証、クライアント証明書を利用した認証、生体認証などさまざまな認証方式をサポートしています。またそれらを組み合わせた多要素認証（OpenAM では認証連鎖と呼ぶ)を実現することもできます。

2.1.2. SSO 実現方式

SSO を実現する方式として、システム構成上から大きくエージェント方式とリバースプロキシ方式に分けることができます。エージェント方式では、Web アプリケーションサーバに配置されたエージェントが認証処理を行います。リバースプロキシ方式では、Web アプリケーションサーバのフロントに配置されたリバースプロキシが認証処理を行います。

リバースプロキシ方式の特殊な形としてアプリケーションの認証を肩代わりする代理認証方式があります。

また、通常 SAML は、異なる組織間で認証の連携を実現するフェデレーションを行うための標準規格ですが、SAML を利用することでも SSO を実現できます。SSO 実現方式の概要を図 2 に示します。

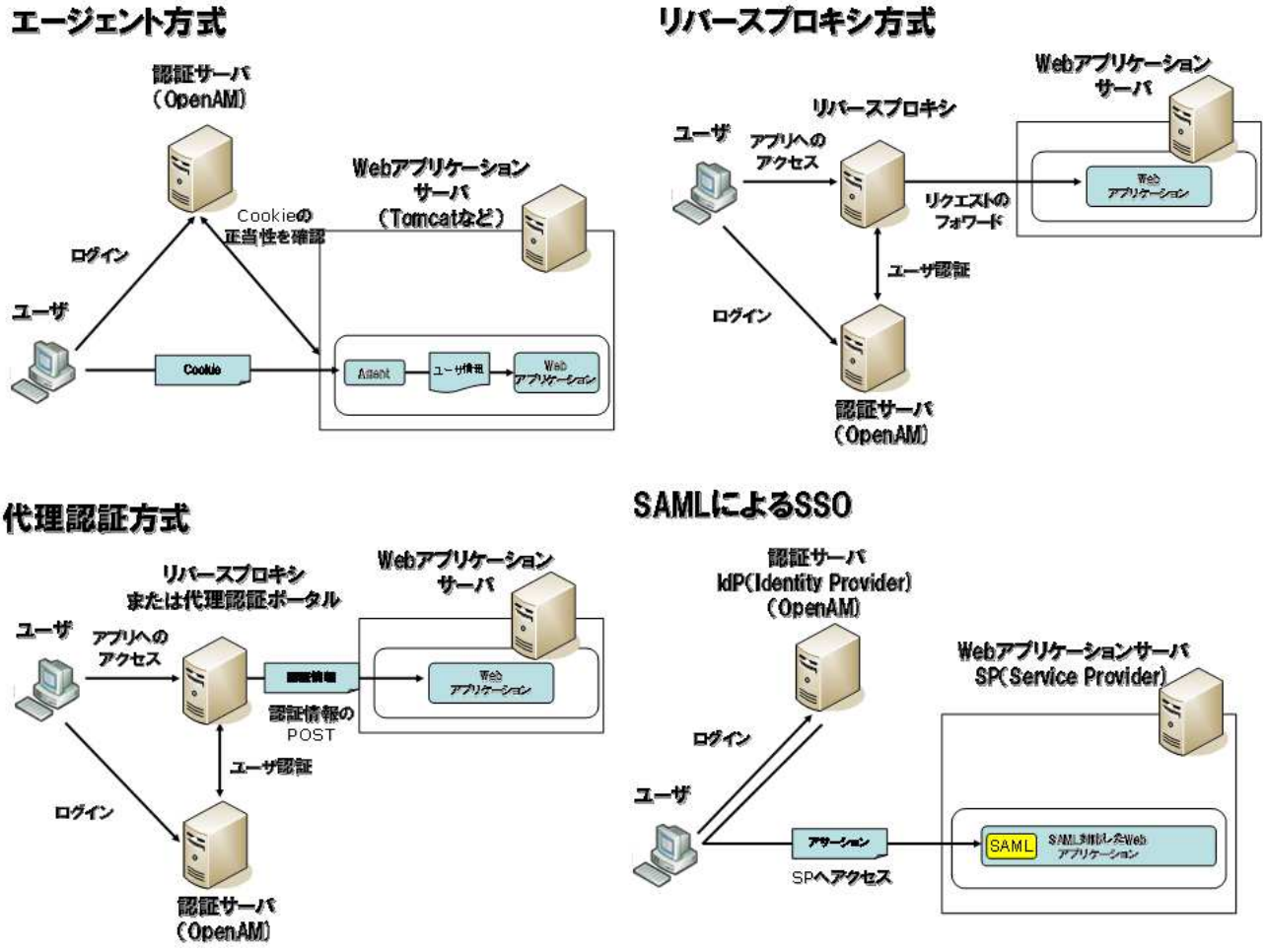

図 2. SSO 実現方式

表 1 はそれぞれの SSO 方式について比較をしたものです。

表 1. SSO 実現方式の比較

認証方式	Webアプリケーションの改修	長所・短所
エージェント	必要	■Webアプリケーションへの全ての通信をエージェントがフックするため、細かなアクセス制御が可能 ■サーバーに対応したエージェントが必要
リバースプロキシ	必要	■Webアプリケーションへの全ての通信をリバースプロキシがフックするため、細かなアクセス制御が可能 ■リバースプロキシがボトルネックになる可能性もある
代理認証	不要	■既存Webアプリケーションの改修が不要 ■代理認証不可能な場合もある
SAML	SAMLに対応していれば不要	■標準的な仕様に準拠したSSOシステムを構築可能。他製品との互換性が高い ■認証サーバー(IdP)を社内に設置し、クラウドサービスであっても、アクセスを社内のみからに制限することも可能（サービスのSAML実装に依る） ■WebアプリケーションがSAMLに対応している必要がある

2.1.3. OpenAM のアーキテクチャ

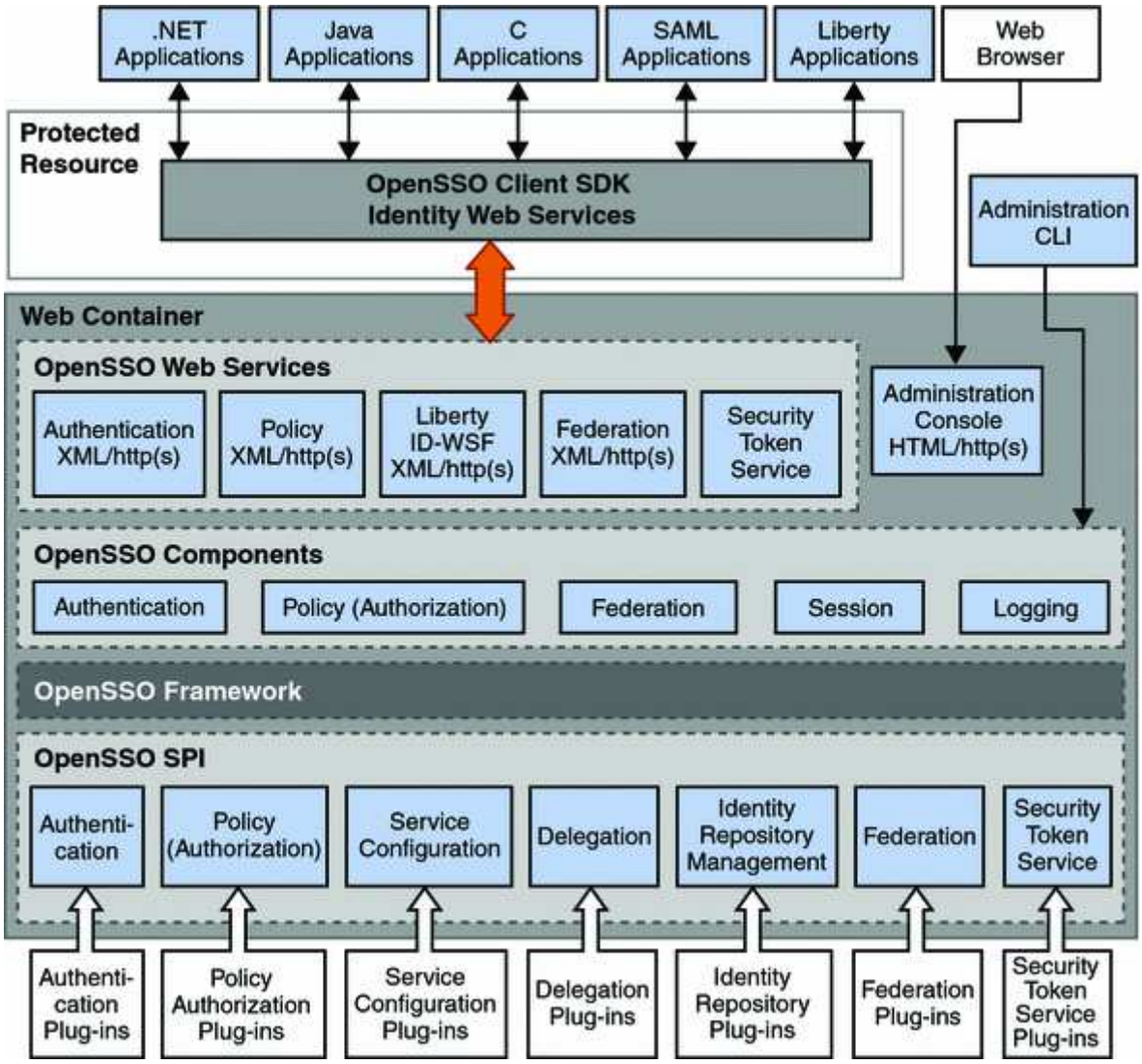

図 3. OpenAM のクライアント/サーバアーキテクチャ

OpenAM は、認証・アクセス制御や安全な Web サービス、フェデレーションなどの機能を提供するために、HTML・XML・SAML・SOAP など多くの標準に準拠しています。構成要素には、クライアント・アプリケーション・プログラミング・インタフェース(Client Software Development Kit(SDK))、ビジネスロジックを実装したサービスのコンポーネント、サービスプロバイダインタフェース(SPI)を実装したプラグインがあります。

それぞれのコンポーネントは、OpenAM 内部 Framework を利用してプラグインや他のコンポーネントが提供しているデータを取得します。プラグインは OpenAM が標準で提供しているもの以外にも SPI に準拠することで独自に開発することができます。認証前後の処理やユーザリポジトリの操作などに独自に開発したモジュールを組み込むことができるので、案件の要件に合せて柔軟に対応した認証基盤を構築することが可能

となります。今までの認証基盤製品ではできなかったことが実現できる可能性を秘めています。

OpenAM のコアサービスを以下に挙げておきます。

- Authentication Service
  特定の認証データストアに対する、ユーザ認証機能を提供する。認証に成功すると、SSO 環境で利用するための認証セッションデータを作成する。
- Policy Service
  認証済みのユーザに関連付けられたポリシーを評価する、認可の機能を提供する。
- Session Service
  Web アプリケーションを横断して、認証済みのユーザのセッションを管理する。
- Logging Service
  各コンポーネントが利用する共通のログ出力機能を提供する。
- Identity Repository Service
  企業の LDAP サーバのように既存のユーザデータストア環境と連携するための機能を提供する。
- Federation Service
  SAML などフェデレーション規格に準拠した機能を提供する。
- Web Services Stack
  OpenAM のサービスを Web サービス経由で提供する。
- Web Services Security and the Security Token Service
  Web アプリケーションを横断して、認証済みのユーザのセッションを管理する。
- Identity Web Services
  認証・認可・セッション管理・ロギングのための Web サービスインタフェースを用意している。主に Policy Agent が利用する。

2.2. 検証目的と検証内容

OpenAM の基本的な動作検証を行うために、以下のような認証基盤の検証テーマを想定しました。

・認証基盤を導入し、既存のアプリケーションを SSO 対応させる（エージェント方式）
・認証基盤を導入し、既存のアプリケーションを SSO 対応させる（リバースプロキシ方式）
・Salesforce などの外部サービスと社内アプリケーションを SSO で利用する
・セキュリティ向上のためモバイル端末の固体認証を行う

それぞれが OpenAM によって実現可能であると確認することを検証の目的としました。

また、設定項目が正常に動作することを確認し、その際に発生した課題・解決方法などを明らかにすることとしました。

2.2.1. 検証テーマ１　エージェント方式による SSO 対応

認証基盤を導入するにあたっては､既存のアプリケーションをどのように SSO に対応させるかが重要です。

そこで既存アプリケーションをエージェント方式により SSO 対応を行い、その際に必要となる設定・既存アプリケーションの修正方法を確認します。対象となる既存アプリケーションとしては会議室予約システムを利用します。

2.2.2. 検証テーマ２　リバースプロキシ方式による SSO 対応

検証テーマ１のエージェント方式では、既存アプリケーションの修正が必要となりますが、リバースプロキシ方式の一つである、代理認証方式を使えば、既存アプリケーションを修正せずに SSO 対応が可能となります。

ここでは、代理認証による既存アプリケーションの SSO 対応を行い、その際に必要となる設定・留意点を確認します。対象となる既存アプリケーションとしては検証テーマ１と同様に会議室予約システムを利用します。

2.2.3. 検証テーマ３　外部サービスも含めた SSO 対応

企業での利用が本格化してきている Google Apps、Salesforce などを SSO で使いたいという要求は、今後も増加してくると予想されます。

そこで外部サービスが SSO で使えることを確認します。対象となる外部サービスは Salesforce の CRM を利用し、社内システムとしては検証テーマ１と同様に会議室予約システムを利用します。

2.2.4. 検証テーマ４　モバイル端末の個体認証

スマートフォンなどの普及により、外出先からも社内システムへ接続する機会が多くなりました。その際にセキュリティを向上させるため、ID/パスワードを知っていたとしても、正式に登録されたスマートフォン以外からはログインできないようにすることが要求されることもあります。

ここでは OpenAM によってモバイル端末の個体認証が実現できることを確認します。モバイル端末としては iPhone/iPad を利用します。

2.3. 検証環境

動作検証に関しては、Linux 上の Tomcat6.0 に OpenAM および認証対象の Web アプリケーションをデプロイして使用しました。また、SAML を使用したフェデレーションの検証は、外部サービスとして Salesforce を使用しました。クライアントは、Windows PC 上の Firefox, Chrome ブラウザおよび iPhone や iPad の Mobile Safari ブラウザを使用しました。

代理認証機能に関しては、標準の OpenAM では提供されていないため、野村総合研究所（以下「NRI」）版の OpenAM に付属の機能を利用して検証を行いました。

iPhone や iPad から接続を行うために、OpenAM および認証対象の Web アプリケーションは、OSS 戦略企画室のラボ環境と Amazon EC2 のそれぞれに同様の環境を用意しました。

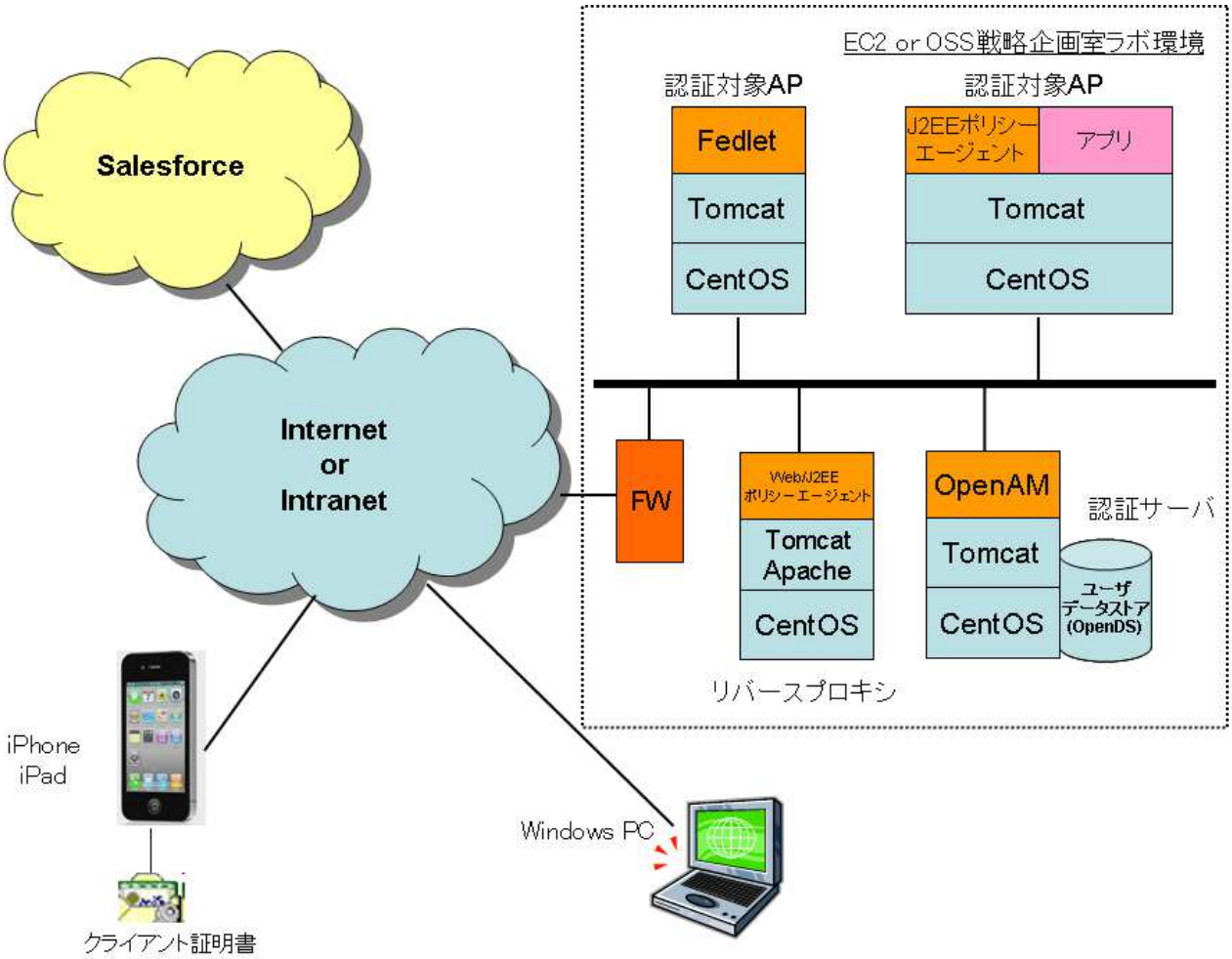

図 4. 検証環境

表 2. 検証環境構成

種別	名称	バージョン
認証サーバ	OpenAM	9.5.3
	OpenDS	(OpenAM 内蔵)
	代理認証モジュール	NRI 版 OpenAM 付属
リバースプロキシ	Apache	2.2.21
	Tomcat	6.0.32
	J2EE/Web ポリシーエージェント	3.0.3_20110718
	代理認証モジュール	NRI 版 OpenAM 付属
AP Server	Tomcat	6.0.32
	J2EE ポリシーエージェント	3.0.3_20110718
OS	CentOS	5.6(5.4)
RDB	MySQL	5.0.77
SSL	OpenSSL	1.0.0a
クライアント	Windows XP Firefox	3.X, 4.X, 10.X
	IE	6.X, 7.X, 8.X
	Chrome	-
	iPhone 3GS	4.3.5

2.4. 検証結果

2.4.1. 検証テーマ 1　エージェント方式による SSO 対応

前述のように OpenAM ではいくつかの SSO の実現方式をサポートしており、システム要件により柔軟に選択することができます。

この検証ではエージェント方式により既存アプリケーションの SSO 対応ができることを確認できました。Web アプリケーションサーバである Tomcat に OpenAM の Policy Agent を導入し、既存アプリケーションとした会議室予約システムにおいては認証処理の替わりに HTTP ヘッダからユーザ情報を取得するように修正しました。これらの対応により既存アプリケーションの SSO 対応が実現できることが確認できました。

エージェント方式は、既存アプリケーションの修正を必要としますが、HTTP ヘッダからユーザ情報を取得するなど比較的簡単なアプリケーションの修正により SSO の対応が可能となりました。

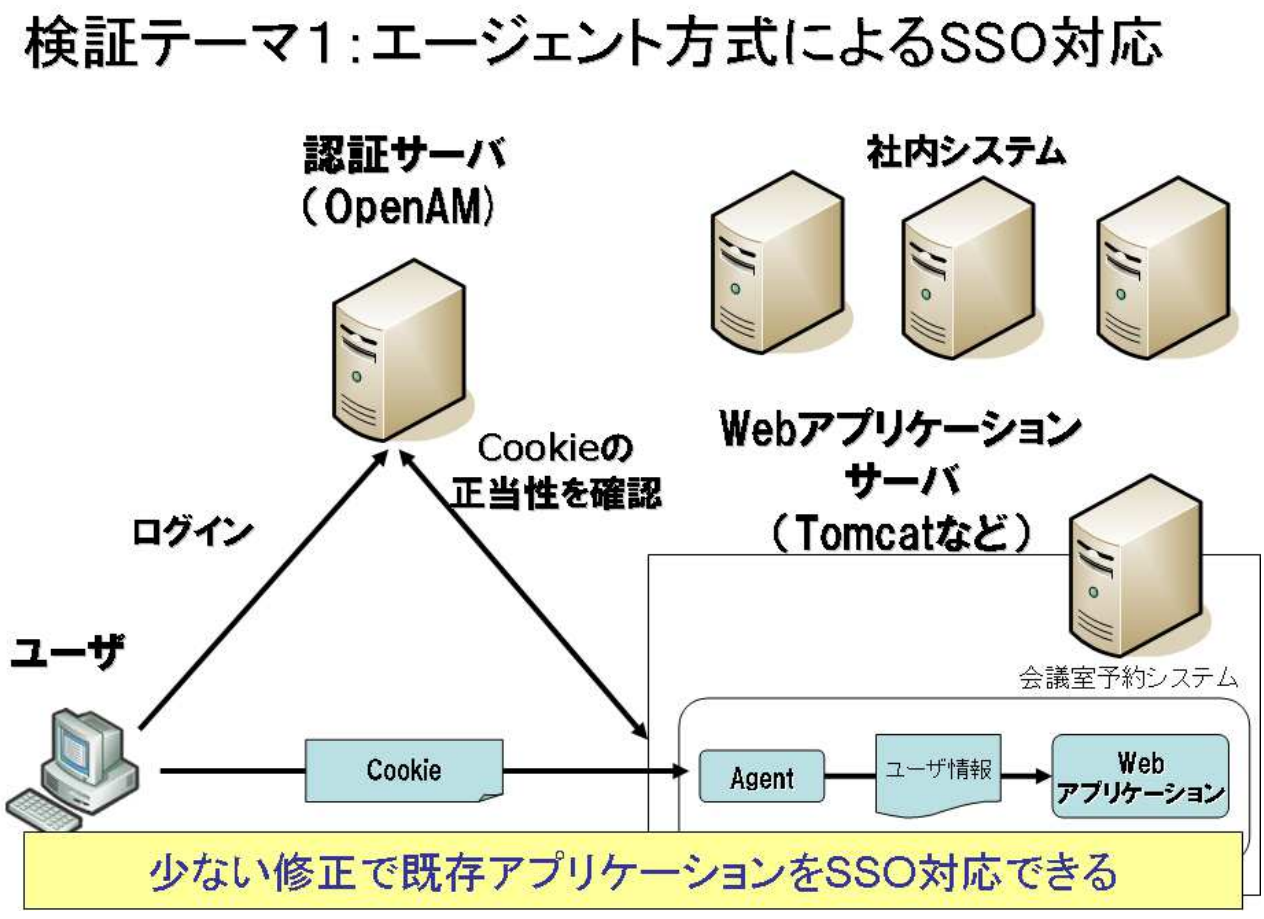

図 5. エージェント方式による SSO 対応

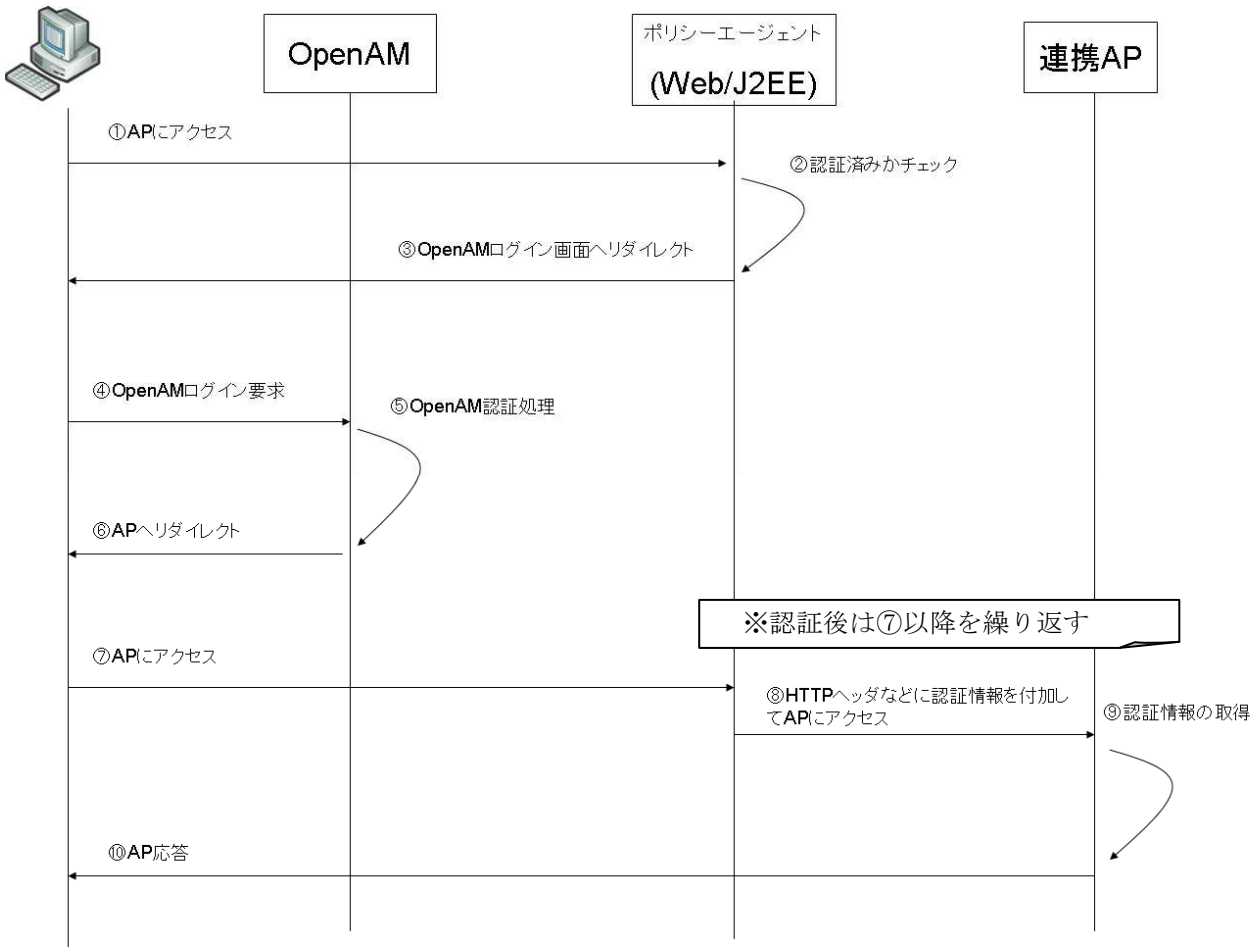

図 6. エージェント方式の処理の流れ

エージェント方式では、図 6 のように PolicyAgent がクライアントからのアクセスを最初に処理します。Apache Web サーバの場合は apache のモジュールとして、Tomcat の場合は認証用拡張モジュールとして実装されています。

PolicyAgent は未認証のクライアントからのリクエストを受け付けると、OpenAM のログイン画面へリダイレクトします。認証済みのクライアントからのリクエストの場合は、認証情報を付加して連携 AP へリクエストを転送します。

どのような認証情報を付加するかは OpenAM の管理コンソールで設定できます。ID などユーザプロファイルに含まれている情報を HTTP ヘッダやクッキーなどに付加することが可能です。

連携 AP では、HTTP ヘッダやクッキーなどからリクエストに付加された認証情報を取得することができます。

2.4.2. 検証テーマ２　リバースプロキシ方式による SSO 対応

この検証では代理認証により既存アプリケーションの SSO 対応ができることを確認しました。
検証テーマ１とは異なりリバースプロキシサーバに OpenAM の Policy Agent を導入し、アプリケーションが動作する Web アプリケーションサーバおよび、既存アプリケーションである会議室予約システムには手を入れること無く SSO 対応が実現出来ました。

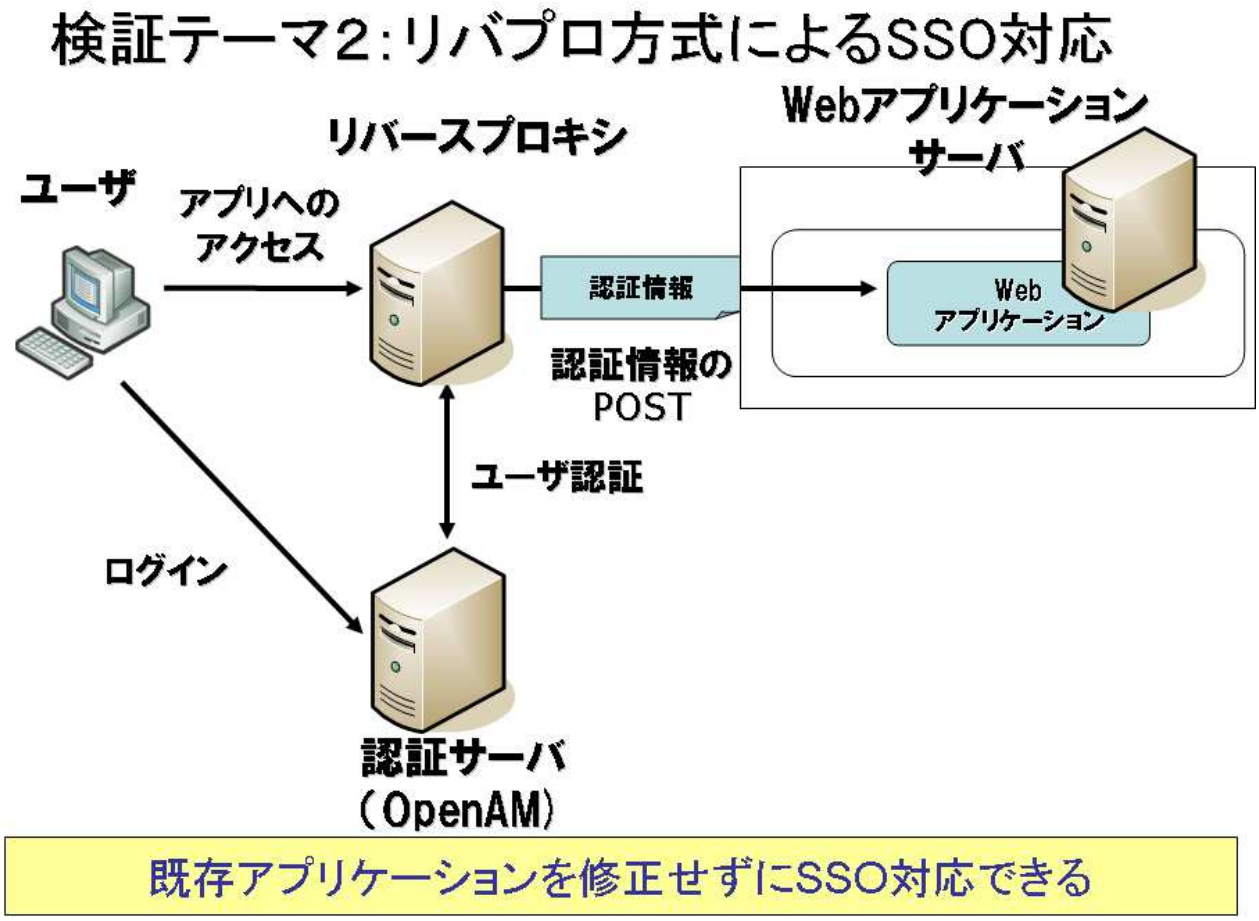

図 7. リバプロ方式による SSO 対応

代理認証機能は、現在の OpenAM では標準機能として提供されていないため、NRI のパッケージ版 OpenAM を利用して検証しました。以下に代理認証の処理の流れを示します。

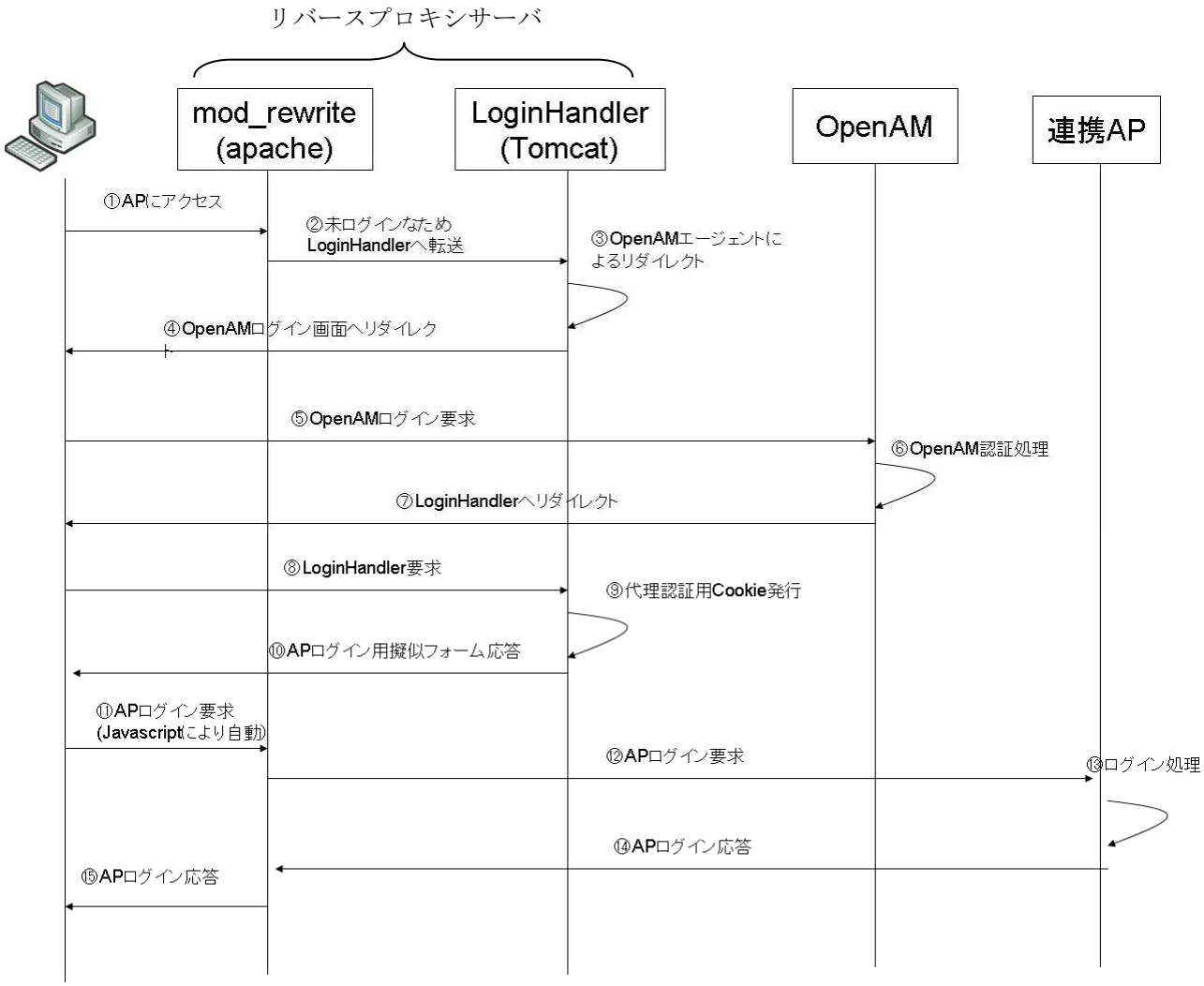

図 8. 代理認証の処理の流れ

図 8 のように NRI 版 OpenAM の代理認証機能は、リバースプロキシ上の Apache で動作する mod_rewrite モジュールと Tomcat で動作する LoginHandler(代理認証モジュール)により実現しています。連携 AP へのログイン処理は JavaScript により自動的に実施されます。連携 AP のログイン後は、LoginHandler を経由せず mod_rewrite から連携 AP へ転送されます。

連携先の AP が SSL であったり、複数の AP であったりした場合も Apache の仮想ディレクトリや仮想ホストの機能を利用して対応できるはずなので今回は検証対象としていません。

今回の検証で利用した会議室予約システムは、Form 認証を元にしていたので、アプリケーションを修正することなく代理認証による SSO 対応することができました。しかし、Basic 認証を元にしたアプリケーションはこのままでは対応できないので、別の仕組みを検討する必要があります。OpenSSO コミュニティで、OpenAM の認証後処理モジュールにおいて Basic 認証で必要な Authentication ヘッダ情報を生成する方法が紹介されています(http://java.net/jira/browse/OPENSSO-2889)。詳細は「5.3Basic 認証 AP に対する代理認証」を参照ください。

なお、リバースプロキシ方式の場合、アプリケーション内で相対 URL を使っている事が前提となります。絶対 URL を使っている場合は、別途コンテンツ書き換えをリバースプロキシで行う必要があります。特に JavaScript 内部で絶対 URL を利用している場合は対応できない可能性もあります。その場合はエージェント方式による対応を取らざるを得ないでしょう。

代理認証機能は、既存 AP をそのまま SSO 対応させることを可能にする技術でありながら、現在の OpenAM の標準機能としては提供されていません。そのため、OpenAM のソリューションを提供している各社が独自に対応して機能を提供しています。

今回の検証では、NRI 社の製品版 OpenAM を使い代理認証の機能を検証しました。連携先のアプリケーションが Form 認証の場合は、この機能で代理認証に対応することができます。ただし前述したように、Basic 認証のアプリケーションを代理認証で SSO 対応するには、別の仕組みが必要になります。また、リバースプロキシに Apache と Tomcat の両方を必要とするなどやや煩雑な構成となっています。

OpenAM の次期バージョンである 10.0 では、リバースプロキシである OpenIG(Identy Gateway)が導入されます。これを使うことにより Form 認証・Basic 認証の双方に対応した代理認証を実現することができます。

2.4.3. 検証テーマ３　外部サービスも含めた SSO 対応

OpenAM では多くのフェデレーション標準規格に準拠しているため、外部サービスの SSO 対応も容易に可能です。

今回の検証では、SAML により Salesforce とのフェデレーションを実現し、社内システムである会議室予約システムとの SSO を実現できることを確認しました。Salesforce は SAML の HTTP Post Binding 方式ですが、OpenAM では HTTP Artifact Binding にも対応しています。以下に SAML の２つの方式の処理の流れを示します。

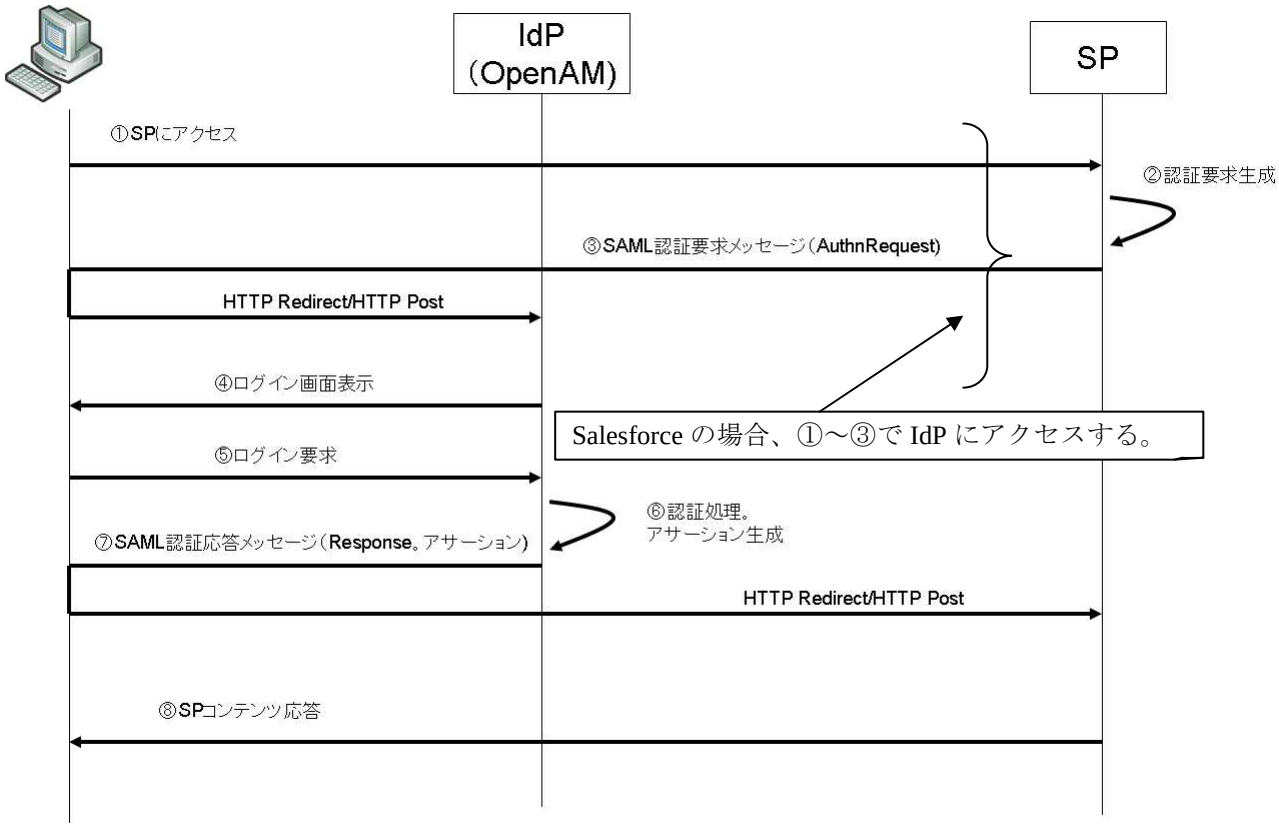

図 9. SAML 認証シーケンス(HTTP Redirect Binding/HTTP Post Binding)

SAML では、認証処理の開始として SP を起点にする場合(SP Initiated)と IdP を起点にする場合(IdP Initiated)が定められています。

Salesforce の場合は、SP を起点とする認証はサポートしていないので、まず IdP にアクセスします。その後は④からの流れとなります。

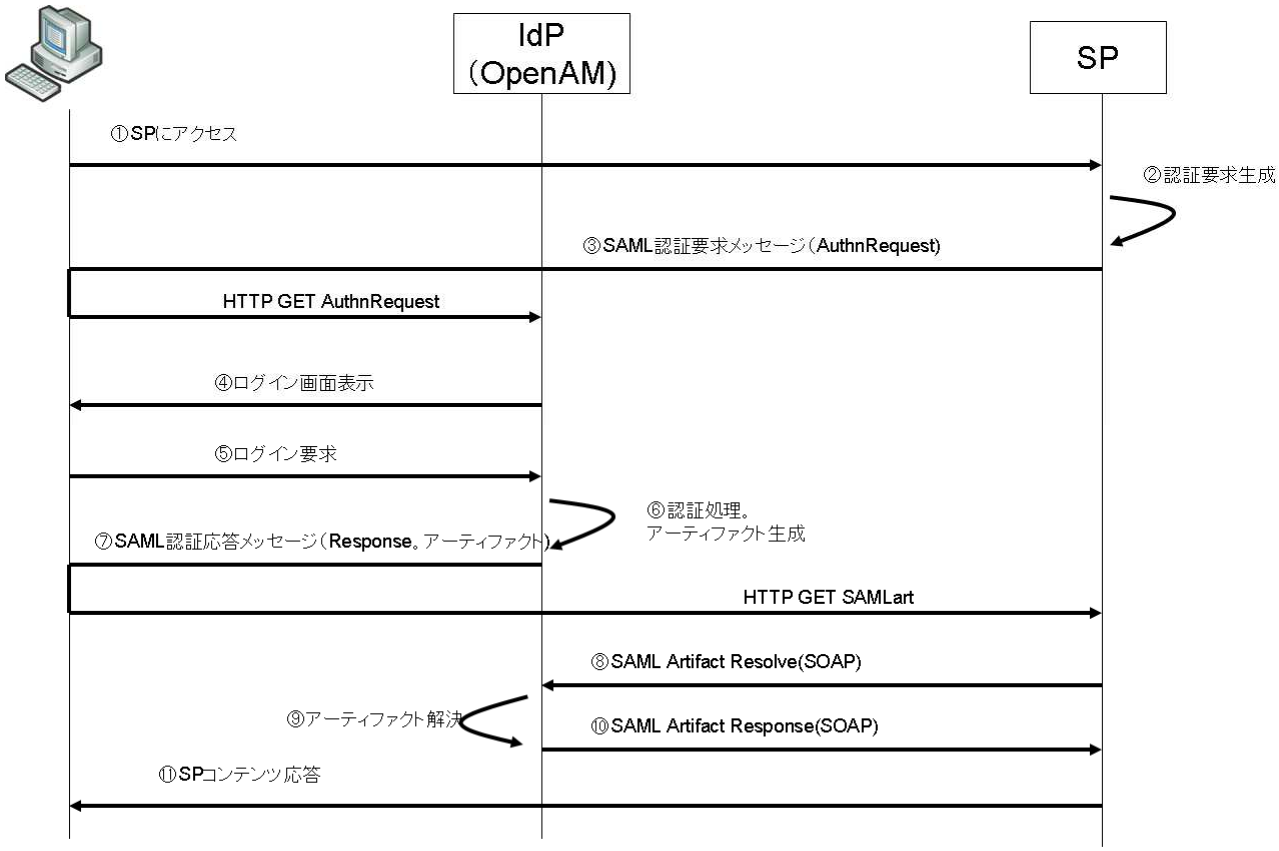

図 10. SAML 認証シーケンス(HTTP Artifact Binding)

HTTP Redirect Binding/HTTP Post Binding と HTTP Artifact Binding の大きな違いは、SP から IdP への通信が発生するかどうかです。

Artifact Binding では図 10 のようにユーザから送られて来たアーティファクトを解決するために SP から IdP への通信が必要となります。外部サービスと SAML で連携する場合、HTTP Redirect Binding/HTTP Post Binding 方式を利用すれば、IdP を内部ネットワークに配置することも可能です。

なお、Artifact Binding にも SP Initiated と IdP Initiated の二通りがあります。

今回の検証では企業ユースを視野に入れて、Google Apps や Salesforce で採用されている SAML を優先的に検証しましたが、コンシューマ向けの外部サービスでは OpenID も利用されているので今後必要であれば調査対象としたいと考えています。

図 11. 外部サービスも含めた SSO 対応

なお SAML を利用することで社内システムなど既存アプリケーションを SSO 対応させることも可能ですが、独自に SAML を実装することは容易ではありません。そのため、OpenAM では Fedlet という機能が用意されており、OpenAM の管理コンソールから自動作成した Fedlet の雛形を利用して SAML 対応させることができます。しかし、Fedlet はアプリケーションの実装上の制約も大きいので、既存のアプリケーションに利用するのは不向きでしょう。

また、自動作成した Fedlet には一部問題があります。詳細は「5.2.2 自動作成された Fedlet を使用する上での留意点」を参照ください。

2.4.4. 検証テーマ４　モバイル端末の個体認証

OpenAM でモバイル端末の個体認証を実現するため、クライアント証明書による PKI 認証の構成を行い、動作を確認しました。

クライアント証明書が導入された端末からしか接続できないことと、パスワード認証と組み合わせた際に、クライアント証明書のユーザと一致しない場合に認証エラーとなることが確認できました。

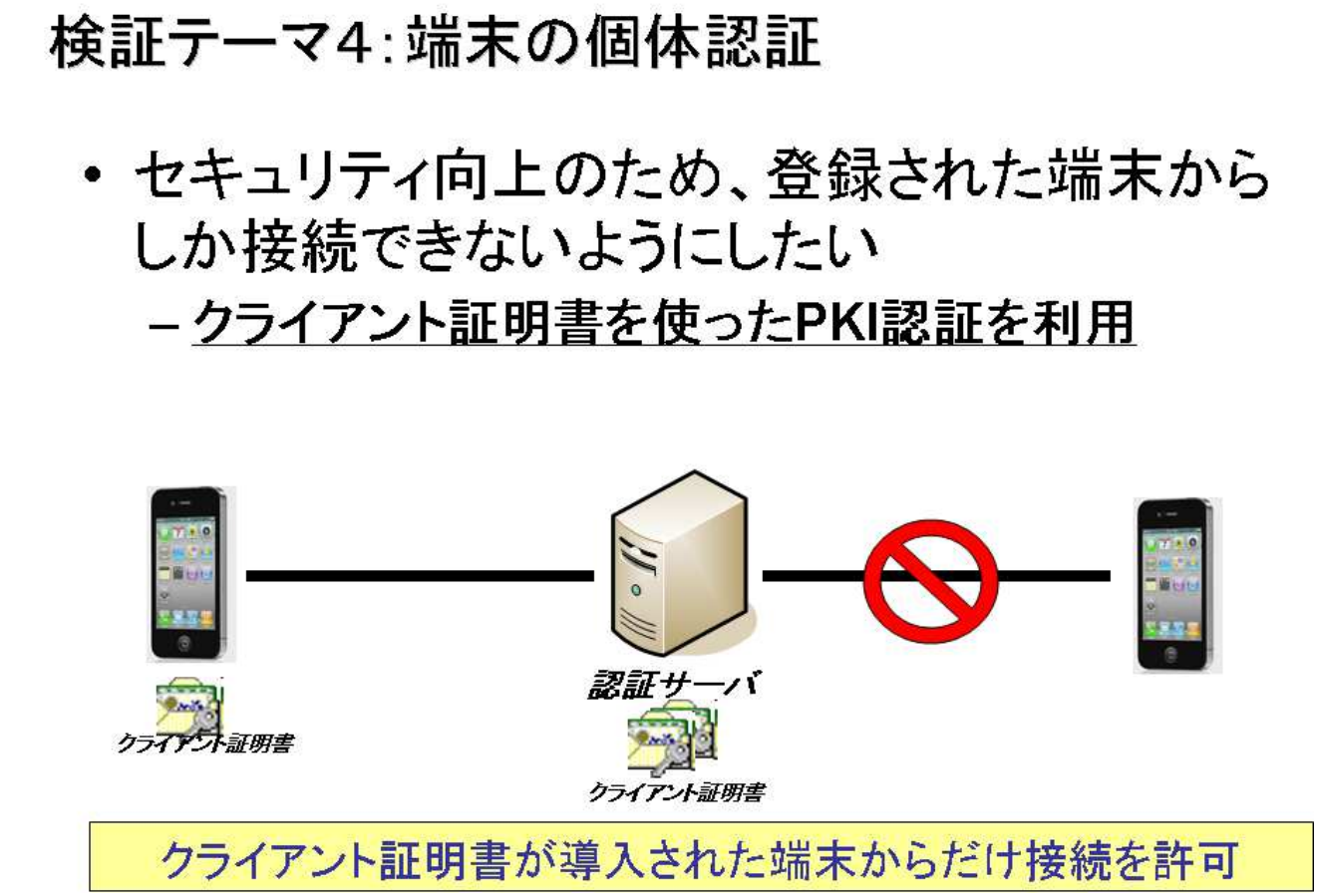

図 12. 端末の固体認証.

2.4.5. 機能検証項目および結果一覧

以下に機能検証項目とそれぞれの結果を示します。

表 3. 機能検証結果一覧

分類	項目	検証内容	検証結果
認証モジュール	パスワード認証	登録したユーザ名、パスワードでログインできること	○
	ワンタイムパスワード認証	パスワード認証と組み合わせて、メールで受信したパスワードを入力してログインできること	○
	証明書認証	パスワード認証と組み合わせて、OpenAM にクライアント証明書を登録済みのユーザにてログインできること、異なる証明書ではログインできないこと	○
	認証連鎖	上記のモジュールを組み合わせ、それぞれ必須(REQUIRED), 必要(REQUISITE), 十分(SUFFICIENT), 任意(OPTIONAL)の条件を設定して、仕様どおり動作すること(※1)	○
ユーザリポジトリ	LDAP	OpenLDAP を認証用のユーザリポジトリとして利用できること。	○
	Active Directory	Active Directory を認証用のリポジトリとして利用できること	○
SSO 実現方式	エージェント方式	Policy Agent を使用して、Web アプリケーションのシングルサインオンが実現できること	○
	リバースプロキシ方式 (代理認証)	代理認証機能を利用して、Web アプリケーションに手を入れずにシングルサインオンが実現できること	○
	SAML	管理コンソールで生成した Fedlet を SP としたシングルサインオンが実現できること	○(※2)
フェデレーション	SAML	SAML1.1, 2.0 を使用して、Salesforce を SP(Service Provider)、OpenAM を IdP(Identity Provider)としてシングルサインオンが実現できること	○
	OpenID	(未実施)	-

※1: 十分：成功したらそこで終了, 必要: 成功しても失敗しても次に継続, 必須: 失敗したらそこで終了, 任意: 認証結果には関係しない付随的な処理

※2: クライアントは Windows PC のみ確認

2.4.6. 機能検証のまとめと考察

今回の検証にあたり、認証基盤で 4 つの検証テーマを想定しました。

・認証基盤を導入し、既存のアプリケーションをエージェント方式により SSO 対応させる
・認証基盤を導入し、既存のアプリケーションをリバースプロキシ方式により SSO 対応させる
・Salesforce などの外部サービスと社内アプリケーションを SSO で利用する
・セキュリティ向上のため端末の個体認証を行う

機能検証では、エンタープライズの認証基盤では基本的だと考えられる３つの構成と、今後利用が進むと考えられる携帯端末の認証の構成を検証しました。

エージェント方式では、既存 AP を HTTP ヘッダから認証情報を取得するように修正することにより SSO に対応させることができました。特に認証のために複雑な API を実装することも無く比較的少ない修正により対応することができることが確認できました。

リバースプロキシ方式では、代理認証機能を利用することにより既存 AP を修正すること無しに SSO に対応させることができました。

表 4. エージェント方式とリバースプロキシ方式の比較

認証方式	AP 修正	メリット	デメリット
エージェント	必要	APにアクセスするURLはそのままで影響を受けない	比較的少ない修正でSSO対応が可能だがAPの修正が必要
リバースプロキシ	不要	APの修正は基本的に不要	リバースプロキシを通すためURLが変更となる影響がある

2 つの SSO 方式を比較すると、代理認証機能を使ったリバースプロキシ方式には、既存 AP を修正することなく SSO に対応できるという大きなメリットがあります。ただし、リバースプロキシを通すことによる URL 変更の影響を受けるデメリットもあります。AP 内で相対 URL による画面遷移を行なっていれば影響も少ないのですが、絶対 URL を使っている場合は書き換えをリバースプロキシで別途実装する必要があります。JavaScript などで絶対 URL を利用する実装をしている場合は、代理認証機能の利用ができない場合も想定されます。

エージェント方式は、既存 AP の修正を必要とするものの、比較的少ない修正で既存 AP を SSO 対応とすることが可能になります。AP 内での URL 取り扱いに関する制約も無いので、新規に AP を SSO 対応する場合は有力な選択肢となるでしょう。

SAML を利用することにより Salesforce と社内アプリケーションを SSO で利用できることも確認できました。特に、SAML の HTTP Post Binding 方式を利用すれば、認証サーバである IdP や認証 DB であるユーザリポジトリを社内に配置したまま、クラウドなどの外部サービスを SSO で利用することが可能となります。セ

キュリティを確保しながら、外部サービスを利用する際の利便性も向上することができます。

既存アプリケーションを SSO 対応するために、SAML を実装する方式を取ることも可能です。しかし、SAML を実装するのは困難であり、Fedlet を使ったとしても既存アプリケーションを SAML 対応させることは現実的ではありません。SAML の利用は外部サービスとの連携に留めておき、既存アプリケーションを SSO 対応させるためには、エージェント方式やリバースプロキシ方式を選択すべきでしょう。

認証 DB となるユーザリポジトリとしては、代表的な OpenLDAP と Active Directory について利用できることを確認しました。他の商用ディレクトリサービス製品にも対応しているので選択可能です。また、現在は実験的な機能ですが、大規模なシステム向けに RDBMS をユーザリポジトリとして選択することもできます。また特殊な案件の用途に合せて独自にユーザリポジトリを実装することも可能です。

携帯端末による認証構成の検証をテーマとして、iPhone/iPad の個体認証に PKI のクライアント証明書認証を OpenAM で利用できることを確認できました。PC でも同様にしてクライアント証明書を利用することができます。ただし、Android の場合、現在のところ標準機能としてはクライアント証明書を利用することができません。

今回の検証では、ブラウザベースの Web アプリケーションを利用しました。しかし、スマートフォンではブラウザベースでないネイティブアプリケーションも多く利用されています。ネイティブアプリケーションで OpenAM の認証を利用するためには、Web サービスベースの API を利用することで可能となりますが、開発を効率的に行うにはネイティブ API ライブラリが整備されることが望まれます。

外部サービスの連携では、SAML を利用して Salesforce との連携を検証しました。今後機会があれば、Google Apps などの SAML サイトの他、別のフェデレーション規格である OpenID に関しても検証したいと考えています。

OpenAM は認証基盤を構築する上で今回想定したシナリオにおいて十分な機能を備えていることが確認できました。また「認証基盤製品比較」でも挙げているようにカタログスペックで比較すると認証基盤製品と比較しても遜色のない機能であることが分かります。ただし、それぞれの提供する機能の程度の差異に関しては実際に検証して確認・評価する必要があるでしょう。

3. 非機能検証

OpenAM の非機能検証では、主に性能面と可用性の面について検証環境を構築し試験を行います。また、セキリティ面と運用面においても考察します。

3.1. 性能試験

3.1.1. 検証目的

ここでは、OpenAM を利用して企業の認証基盤を構築した際に、性能面において実用に耐えうるかを検証します。その際に性能上の課題が存在しないか、ボトルネックの存在、検証環境のハードウェアスペックにおけるパフォーマンスの指標などを明らかにすることを目的とします。

想定した企業の規模、利用シナリオは以下の通りです。

- 登録ユーザ数　10,000 人
- 10 万アクセス/日、ピーク時 1.4 アクセス/秒
- ユーザは、出社後ポータルへログインし、スケジューラなどのアプリケーションを利用
- 同時利用ユーザ　1,000 人

3.1.2. 検証内容

以下の２つのシナリオにより負荷を発生させます。

- 認証のみ(OpenAM へのログイン・ログアウト)
- 認証＋連携アプリケーション
  会議室予約システムを代理認証方式で連携

負荷の発生には、Apache が提供している OSS の JMeter を利用します。テストシナリオの作成は、JMeter のレコーディング機能を利用してブラウザの操作を記録したものをベースに想定した要件に合せて修正します。

均等に負荷が掛かるように、JMeter のテストシナリオを調整し、リクエスト毎に異なるサーバへ向くようにします。

負荷発生時に各サーバのリソース状況（CPU、メモリ、IO、JVM、など）を確認します。また、ログ出力内容もチェックし致命的な障害が発生していないことを確認します。リソース状況の取得には espf のリソース情報収集ツールを利用します。

各ミドルウェアは、基本的にはデフォルト設定を元にし、性能目標を達成できない時にチューニングを実

施するという方針とします。

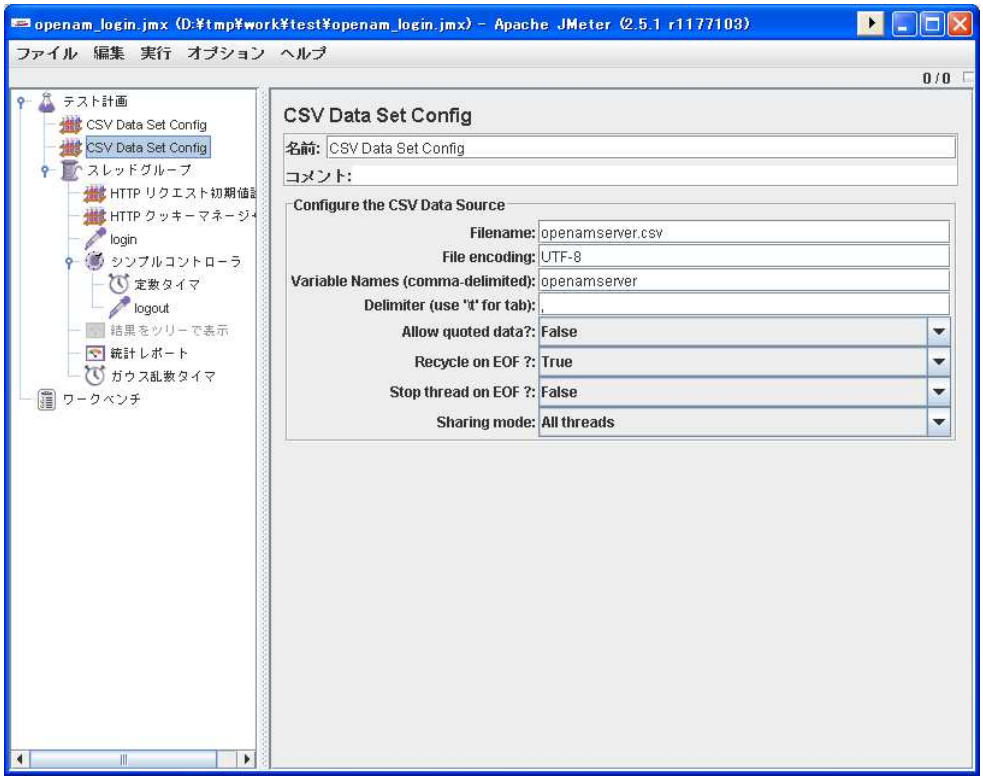

図 13. 認証のみ(OpenAM へのログイン・ログアウト)

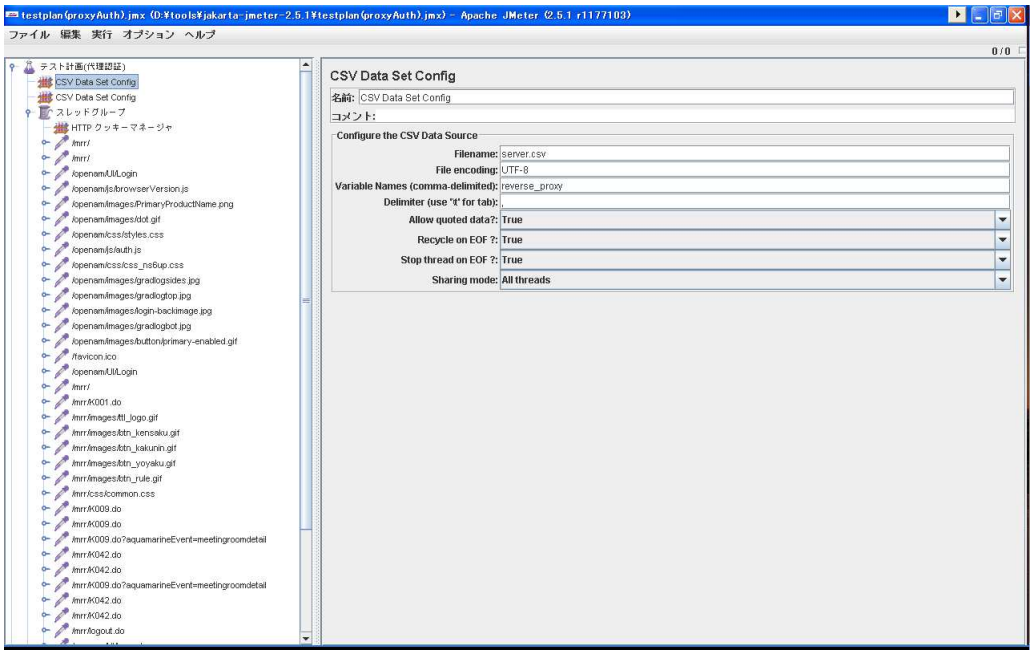

図 14. 認証＋連携アプリケーション

3.1.3. 検証環境

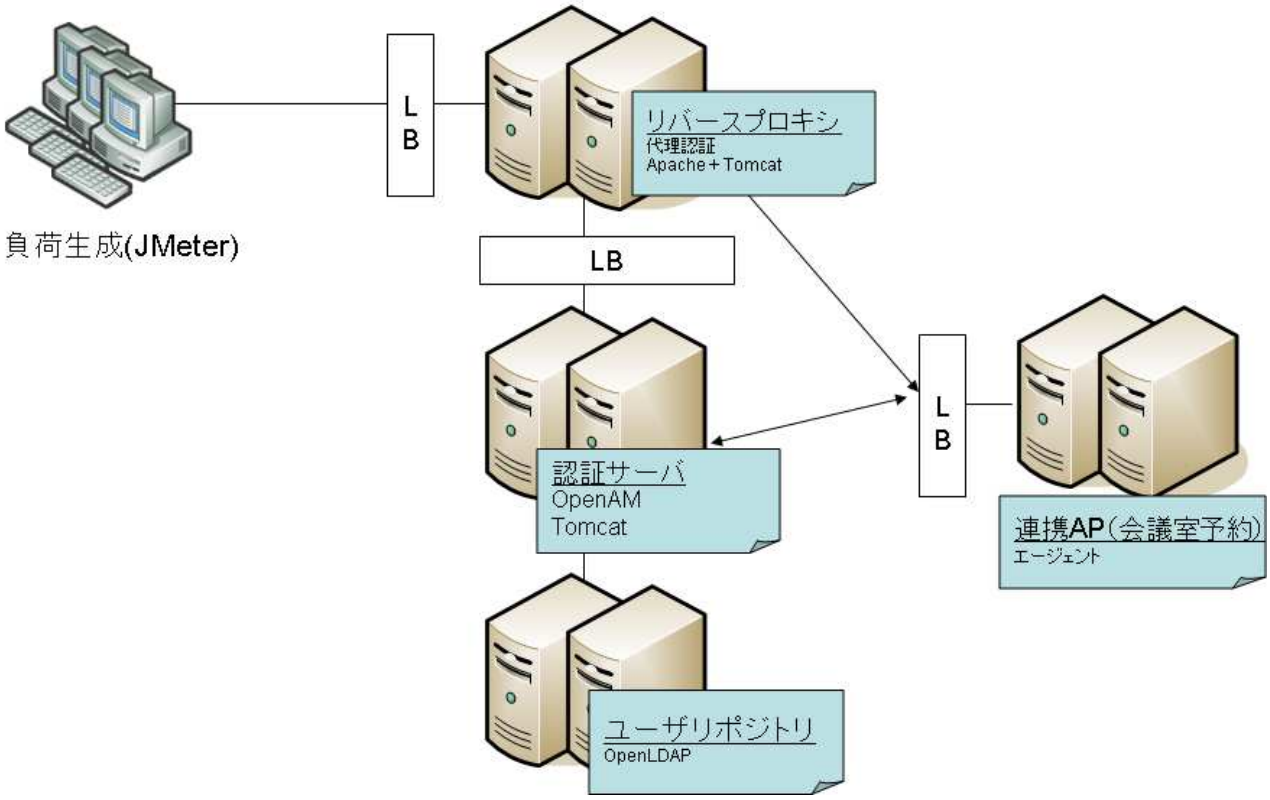

図 15. 性能試験検証環境

性能試験を実施するにあたり、AP の連携方式として利用シーンが多いと想定される、代理認証方式を採用し、各サーバを冗長構成にした検証環境を構築しました。

表 5 に構成要素を示します。

表 5. 性能試験検証環境構成

ソフトウェア	バージョン
認証サーバ	
Tomcat	6.0.33
JDK	1.6u29
OpenStandia版 OpenAM	9.5.3
OpenStandia版 代理認証モジュー	ReplayPassword.zip
リバプロ	
Apache HTTPD	2.2.21
Tomcat	6.0.33
JDK	1.6u29
apacheエージェント	apache_v22_Linux_64_agent_304
Tomcatエージェント	tomcat_v6_agent_303
OpenStandia版 代理認証モジュー	proxyAuth.zip
ユーザリポジトリ	
OpenLDAP	2.3.43
連携AP	
Tomcat	6.0.33
JDK	1.6u29
Tomcatエージェント	tomcat_v6_agent_303
Core会議室	-
LB	
LVS	1.2.1
KeepAlived	1.2.2
負荷生成クライアント	
JDK	1.6u29
JMeter	2.5.1
インフラ	バージョン・スペック
CPU	Xeon5504 2.0GHz 2CPU(8core)APサーバのみ2CPU害
メモリ	2GB(APサーバは3GB)
OS	CentOS5.7

表 6。仮想環境スペック

HP ProLiant DL350G6 QC XE5504 2.00GHZ 2P/8C
4GB 2RX4 PC3-10600R メモリキット
500GB SATA HDD
NC360T デュアルポート GIGABIT アダプタ
SMART アレイ 256MB キャッシュ
SMART アレイ BBWC ヨウバッテリ
Citrix XenServer version5.5.0

3.1.4. 検証結果

3.1.4.1. 性能試験シナリオ 1 の結果

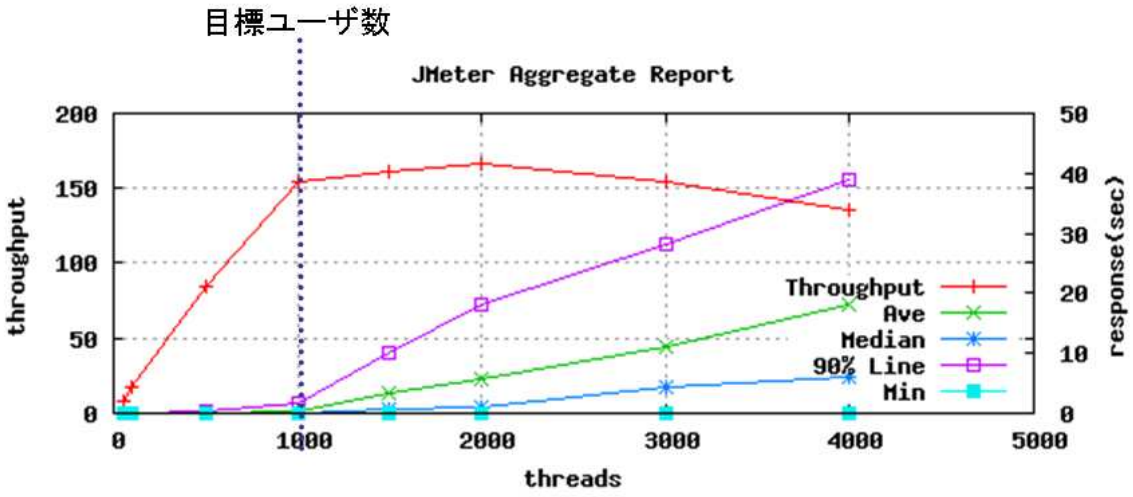

図 16. 性能試験シナリオ 1 結果

OpenAM の認証のみのシナリオを実施した結果では、同時 1,000 ユーザ利用時に、スループットが 150 リクエスト/秒、90%タイル 1.6 秒となっており、性能目標をクリアしました。

ただ、スループットに関しては同時 2,000 ユーザをピークに減少傾向が見られます。

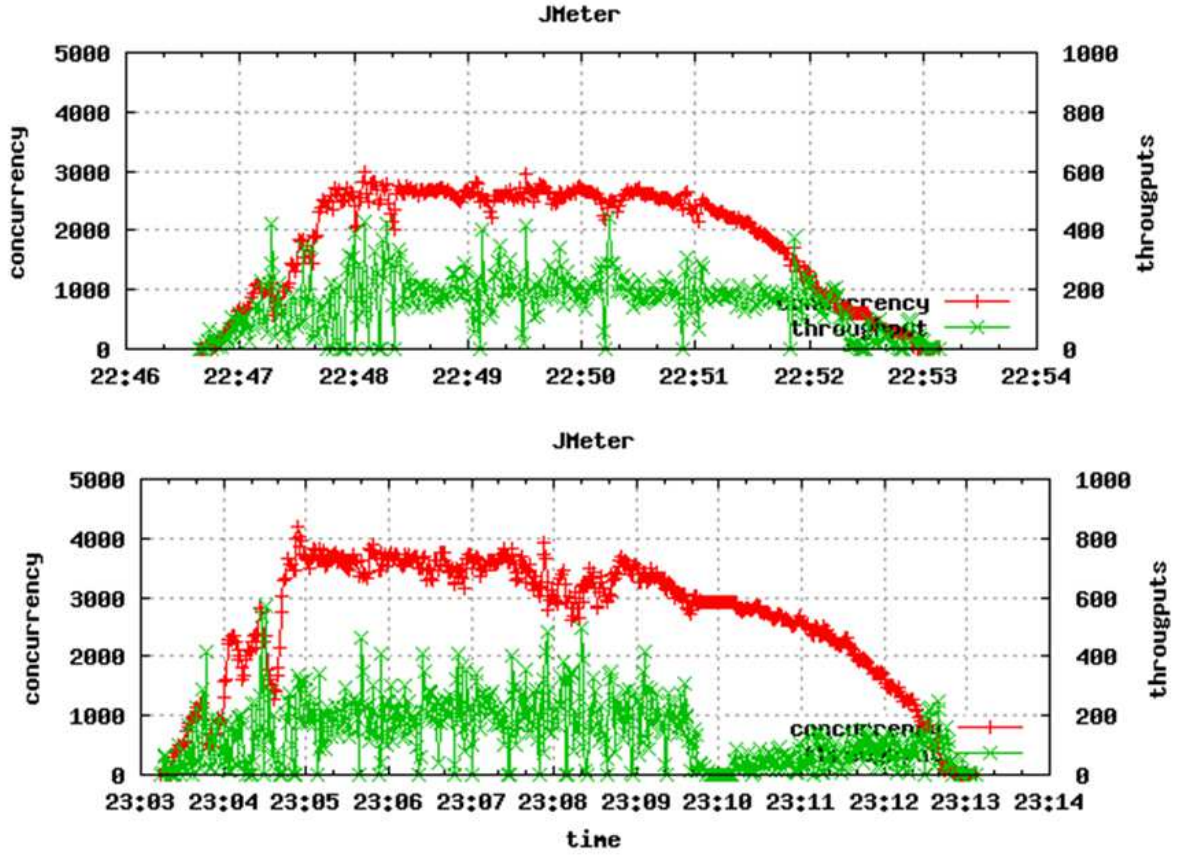

図 17 は 3,000 ユーザ・4,000 ユーザで負荷試験を行った際の同時接続数と、スループットをプロットしたものです。これを見ても試験開始後 5 分を過ぎた辺からスループットの落ち方が激しく検証環境の物理的な限界に達したものと予想されます。

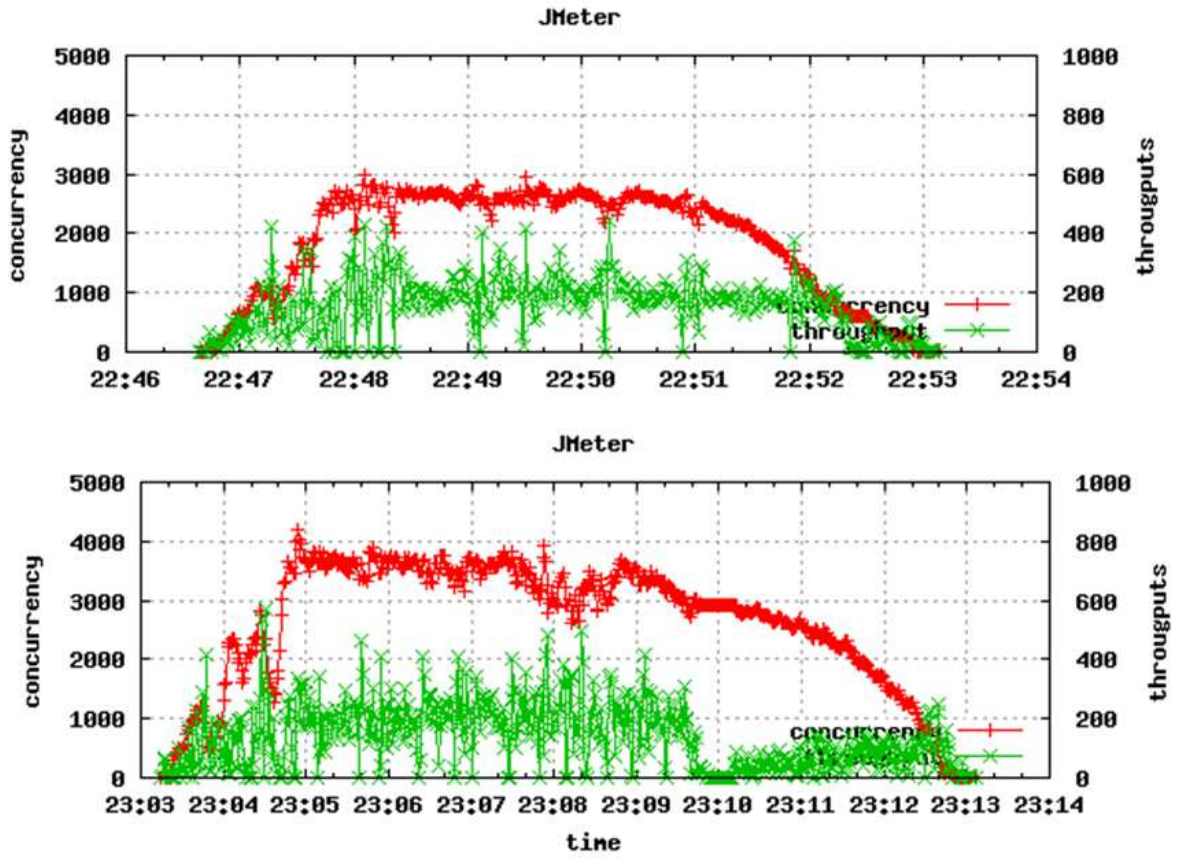

図 17. 性能試験シナリオ 1 (3,000・4,000 スレッド)

赤：concurrency, 緑：througputs

3.1.4.2. 性能試験シナリオ２の結果

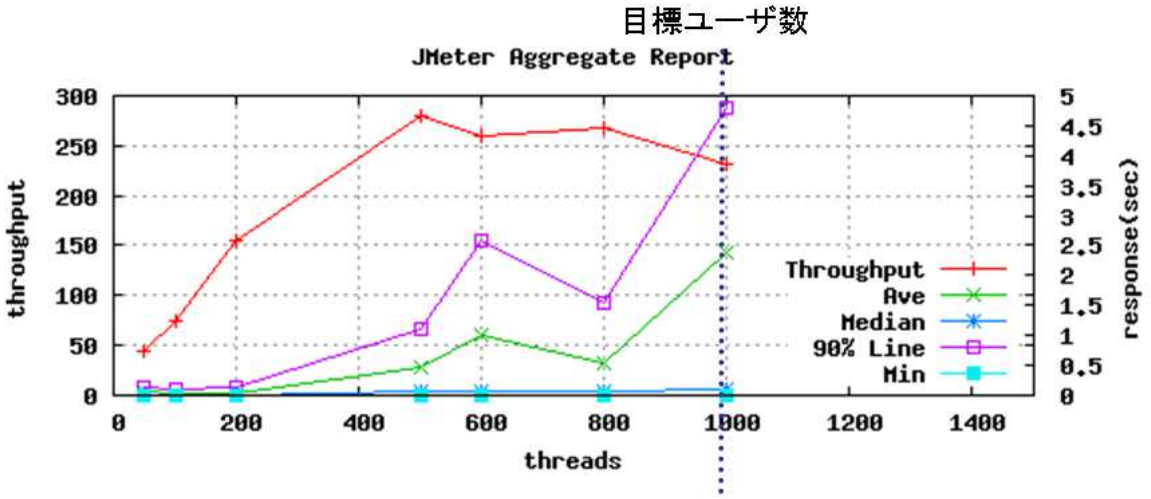

図 18. 性能試験シナリオ 2 結果

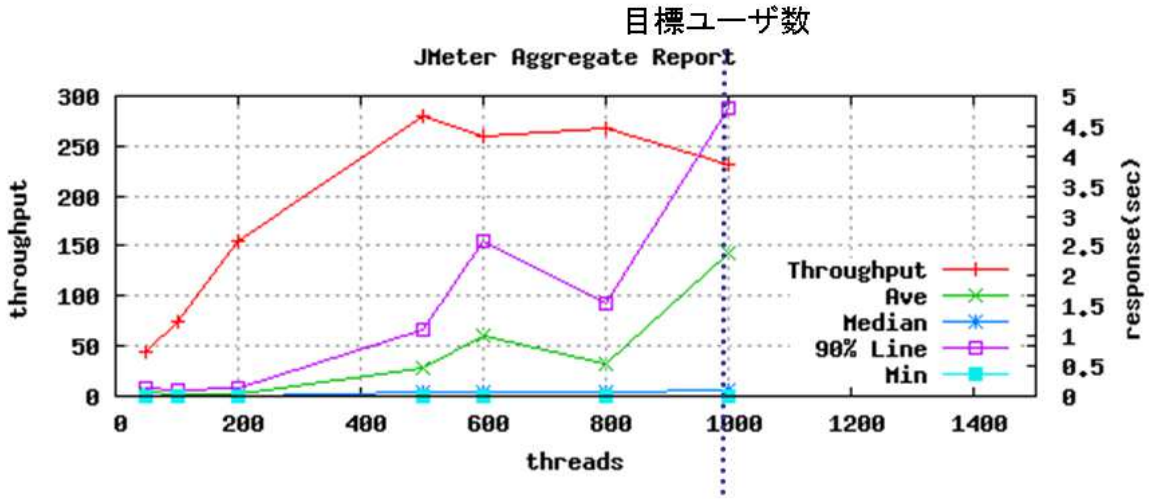

図 18 は連携 AP まで含めた負荷試験の結果を示します。

同時 1,000 ユーザの負荷を発生させた時に、スループット 230 リクエスト/秒、90%タイル 5 秒となり性能目標は達成することができました。ただし、主に AP サーバの処理限界に達しており、それ以上の負荷を与えると安定した試験結果を得ることはできませんでした。

また、AP サーバのメモリリークの兆候が見られたために、原因を調査したところ、ログアウト時にセッションデータが削除されていないことと、Tomcat7 で開発され Tomcat6 にも導入されたメモリリーク検知機能(*JreMemoryLeakPreventionListener)*が原因であることが判明しました。

アプリケーションおよび AP サーバ(Tomcat)で行ったチューニング内容は以下のようになります。

・ログアウト時にセッションデータを削除する画面(JSP)を追加しテストシナリオに追加

・設定ファイル(server.xml)から JreMemoryLeakPreventionListener をコメントアウト
・Tomcat の起動時に JVM におけるヒープ割当ての調整
  - *-XX:+UseConcMarkSweepGC*
  - *-XX:+CMSParallelRemarkEnabled*

3.1.4.3. 性能試験結果のまとめ

OpenAM を企業の認証基盤で利用するために以下のような性能目標を想定しました。

・登録ユーザ数　10,000 人
・10 万アクセス/日、ピーク時 1.4 アクセス/秒
・同時利用ユーザ 1,000 人

シナリオ１の OpenAM 単体では性能目標を問題なく達成しており、検証環境のような一般的なスペックの仮想マシン（CPU:Xeon2.0GHz 相当×１、メモリ２GB 詳細は表 6 参照）の２台構成で特に問題なく処理できることが検証出来ました。

シナリオ２の OpenAM と連携 AP まで含めた試験の場合は、性能目標は達成しましたが、以下のような課題がありました。

・AP 処理の影響が大きいのでサーバのスケールアウトやチューニングが必要
・同時５００ユーザを超えると性能劣化傾向が見られる

今回の検証では OpenAM の性能試験ということで、アプリケーションのチューニングには踏み込んでいませんが、実際のシステムを構築する際には重要なポイントとなります。

リバースプロキシ方式の場合、連携するアプリケーションが増えてきた場合、リバースプロキシサーバの処理がパフォーマンスのネックになることが懸念されます。今回の性能検証では、リバースプロキシサーバがボトルネックにはなっていませんでしたが、連携するアプリケーションが増えた場合は、合せてリバースプロキシサーバの台数を増やすことを検討すべきでしょう。

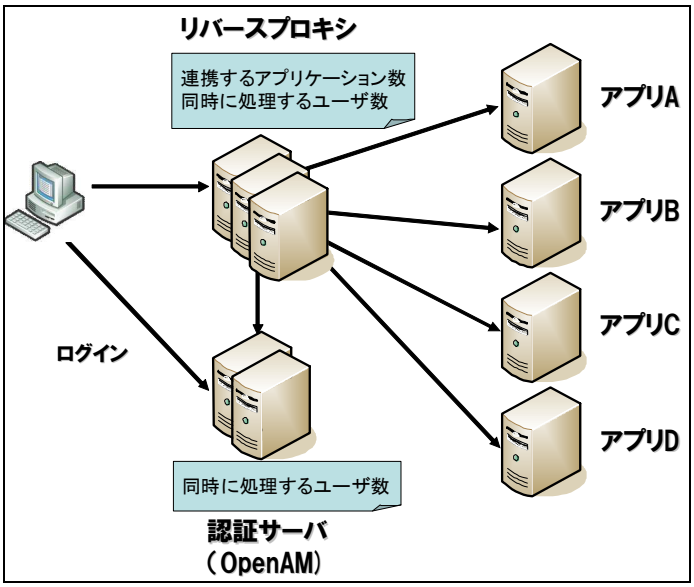

図 19. スケールアップ時の考慮点

3.2. 可用性試験

認証基盤は、企業システムの重要な構成要素となるため、一部に障害が発生したとしてもサービスを継続するだけの十分な可用性を担保することが求められます。可用性を向上するためには、一般的に冗長構成を検討する必要がありますが、その際に

・OpenAM における冗長構成
・ユーザリポジトリと負荷分散装置における冗長構成

の両面を考慮する必要があります。以降ではそれぞれについて説明していきます。

3.2.1. OpenAM における冗長構成

OpenAM では、認証基盤の可用性を向上するために冗長構成をとることができます。サイト構成とセッションフェイルオーバ構成の２つの方式があります。

3.2.1.1. サイト構成

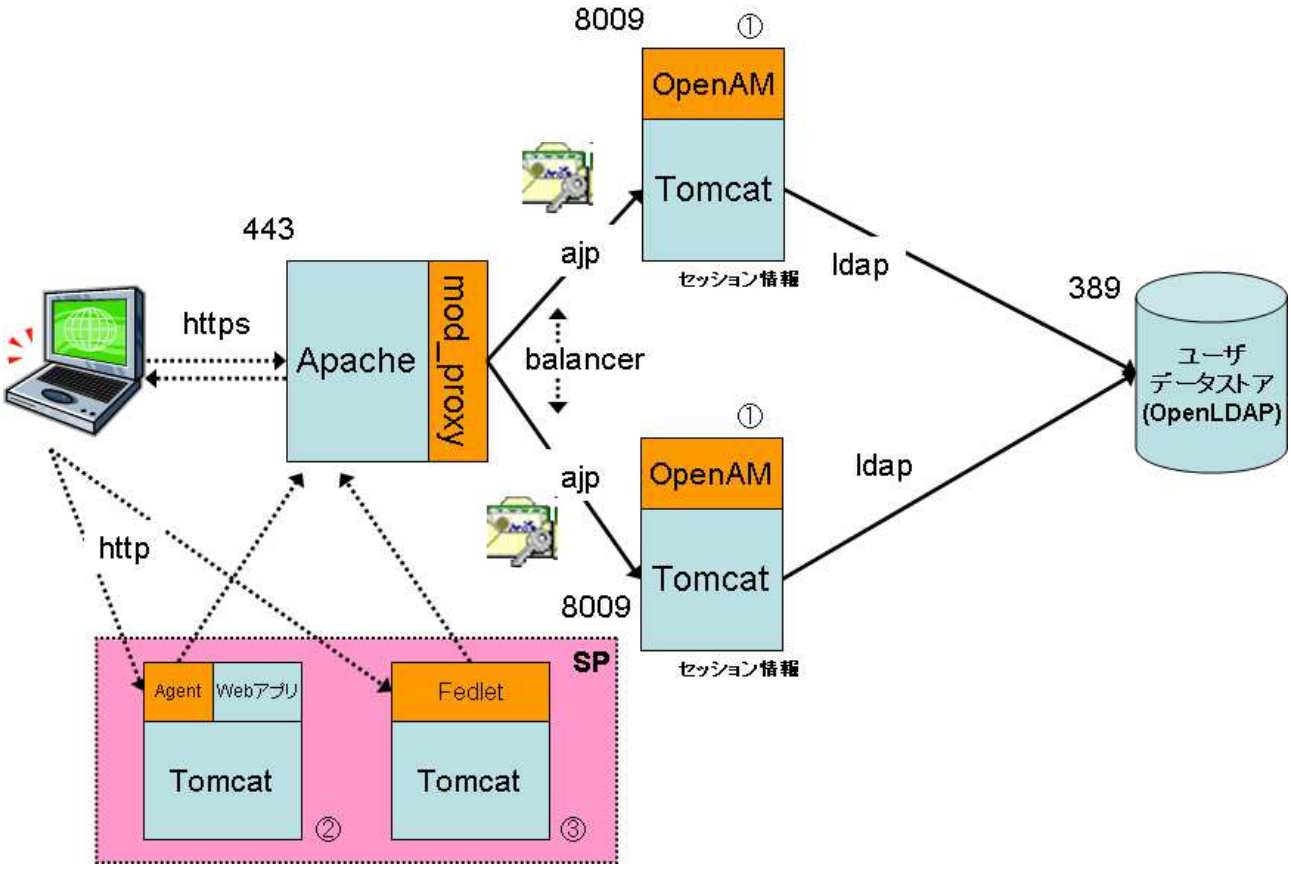

図 20. サイト構成

サイト構成の場合、サイトに参加している OpenAM はお互いに構成情報のやり取りを行います。そのため管理コンソールはどちらのサーバにアクセスしても構成情報は共有されます。
一方、ユーザのセッション情報に関しては個々の OpenAM で独立して保持し、共有しません。よって 1 台の OpenAM が落ちるとそのサーバで認証したセッション情報にアクセスできないため、ユーザは再ログインをする必要があります。

負荷分散装置は、OpenAM のセッション情報を元にクライアントからのアクセスを振り分ける必要があります。

3.2.1.2. セッションフェイルオーバ構成

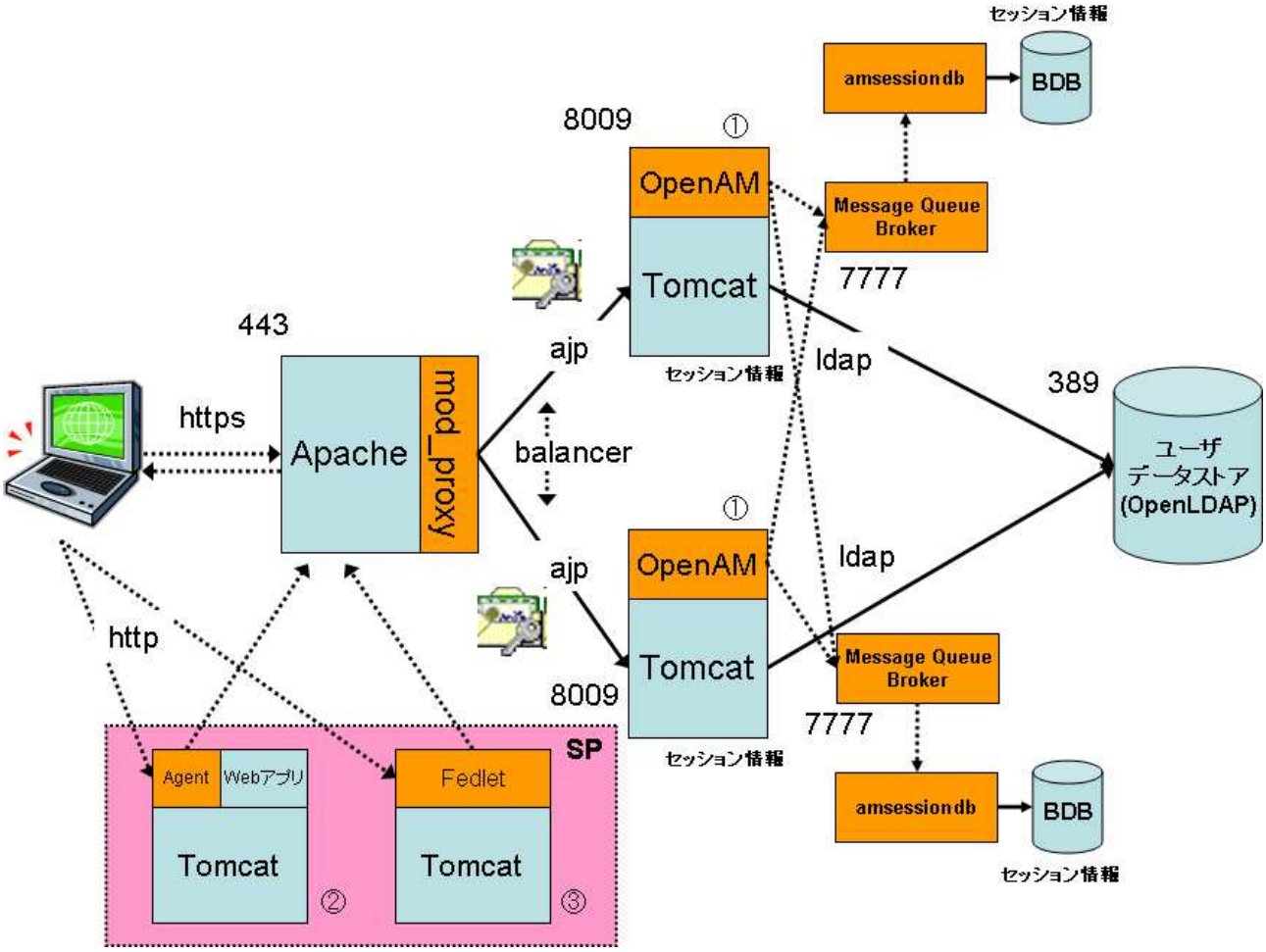

図 21. セッションフェイルオーバ構成

セッションフェイルオーバ構成では、サイト構成に加えてセッション情報を障害に備えて保持しておく機能が組込まれています。

そのため、一台の OpenAM が落ちても別のインスタンスからセッション情報にアクセスできるため、ログ

イン済みのユーザは処理を継続できます。

セッション情報は、Message Queue Broker を経由して amsessiondb に保持されます。OpenAM は自分が保持していないセッション情報でも Message Queue Broker を経由で問い合わせることに取得することができます。これにより、ユーザがログインした時と別の OpenAM にアクセスした際でも、OpenAM は処理を受け付けることができます。

3.2.2. ユーザリポジトリと負荷分散装置における冗長構成

3.2.2.1. ユーザリポジトリ

ユーザリポジトリには OpenLDAP を利用しています。レプリケーション方式は、大きく slurpd という専用のデーモンプログラムを利用しマスタサーバから変更情報をスレーブサーバへ送付する形式と、overlay 機能で提供される syncprov モジュールを利用してスレーブサーバからマスタサーバを検索して変更情報を取得する２つの方式が存在します。

今回の検証では、今後の利用が多くなると予測される syncprov モジュールを利用したレプリケーション方式を採用しています。

3.2.2.2. 負荷分散装置

今回構築した OpenAM 検証環境では、リバースプロキシサーバ、認証サーバ、アプリケーションサーバの冗長性を確保するために負荷分散装置（ロードバランサ）を配置しています（図 22 にて LB と表記）。負荷分散装置はそれぞれのサーバへのリクエストの振り分けとサービス監視をおこなっており、サーバに障害が発生した場合でも全体としてサービスが継続できるようにするための処理を行なっています。一般のシステムにおいてよく用いられる負荷分散装置としては、F5 ネットワークス社の Big-IP などがありますが、今回の検証では、Linux システムにおける負荷分散ソリューションである LVS(Linux Virtual Server)を中心に用いて検証環境を構築しました。LVS の他にも keepalived や iptables などのソフトウェアコンポーネント（OSS）を以下の用途で組み合わせて、負荷分散装置全体の機能を実現しています。

- LVS を用いたネットワークトラフィック分散
- keepalived を用いたサービス監視と障害サーバ回避
- iptables を用いた仮想 IP アドレス実現

それぞれの詳細については末尾の「負荷分散装置の構成詳細」を参照ください。

3.2.3. 検証目的

認証基盤は企業システムの中で重要な役割を持つため、例え一部の機能に障害が発生したとしてもサービスを継続することが求められます。

そのため、この検証では、OpenAM を中心とした認証基盤の構成要素を冗長構成にして可用性を向上させることを図ります。それぞれの構成要素に障害が発生したとしても、認証基盤全体としてはサービスが継続できることを確認します。

3.2.4. 検証内容

OpenAM の冗長構成として以下の２つの方式について試験を実施します。

- サイト構成
- セッションフェイルオーバ構成

それぞれの構成について、各構成要素を起動・停止を行い、認証基盤のサービスが継続できるかを確認します。

テストシナリオは、性能試験と同様に、以下の２つを利用して可用性の試験を実施します。

- シナリオ１：認証のみ(OpenAM へのログイン・ログアウト)
- シナリオ２：認証＋連携アプリケーション
  会議室予約システムを代理認証方式で連携

表 7. サイト構成試験テストケース

	テストケース	テスト対象	
		シナリオ1	シナリオ2
1-1	別ブラウザで別OpenAMにリクエストが分散される(ただし、Sticky)	○	○
1-2	OpenAM1台停止後、未認証のクライアントからアクセスできる	○	○
1-3	自身のセッション情報を持たないOpenAMを停止後、OpenAMにアクセスできる	○	○
1-4	自身のセッション情報を持つOpenAMを停止すると、ログイン画面にリダイレクトされ	○	○
1-5	1-4でOpenAM再起動後、未認証のクライアントは正常に使用できる	○	○
1-6	1-4でOpenAM再起動後、認証済みのクライアントはログイン画面にリダイレクトされ	○	○
1-7	OpenLDAP1台停止後、認証済みのクライアントは正常に使用できる	○	○
1-8	OpenLDAP1台停止後、未認証のクライアントは正常に使用できる	○	○
1-9	OpenLDAP再起動後、認証済みのクライアントは正常に使用できる	○	○
1-10	OpenLDAP再起動後、未認証のクライアントは正常に使用できる	○	○
1-11	リバプロ1台停止後、認証済みのクライアントは正常に使用できる	−	○
1-12	リバプロ1台停止後、未認証のクライアントは正常に使用できる	−	○
1-13	リバプロ再起動後、認証済みのクライアントは正常に使用できる	−	○
1-14	リバプロ再起動後、未認証のクライアントは正常に使用できる	−	○
1-15	リバプロ1台のApache停止後、認証済みのクライアントは正常に使用できる	−	○
1-16	リバプロ1台のApache停止後、未認証のクライアントは正常に使用できる	−	○
1-17	リバプロのApache再起動後、認証済みのクライアントは正常に使用できる	−	○
1-18	リバプロのApache再起動後、未認証のクライアントは正常に使用できる	−	○
1-19	リバプロ1台のTomcat停止後、認証済みのクライアントは正常に使用できる	−	○
1-20	リバプロ1台のTomcat停止後、未認証のクライアントは正常に使用できる	−	○
1-21	リバプロのTomcat再起動後、認証済みのクライアントは正常に使用できる	−	○
1-22	リバプロのTomcat再起動後、未認証のクライアントは正常に使用できる	−	○

表 7 はサイト構成におけるテストケースを示します。

1-1 では、別のブラウザからアクセスすることにより、別の OpenAM にリクエストが分散されることを確認します。どの OpenAM にアクセスしているかは、OpenAM の管理コンソールに表示されているサーバ名で確認できます。

1-2 では、一台の OpenAM を停止した後、まだログインをしていないブラウザからアクセスすることにより、認証システムが継続してサービスを提供していることを確認します。

1-3、1-4 では、ログイン済みのブラウザを利用して、セッション情報の有無による OpenAM サーバ停止の影響を確認します。

1-5、1-6 では、OpenAM サーバ再起動による影響を確認します。

1-7、1-8、1-9、1-10 では、OpenLDAP サーバ一台の停止、再起動が OpenAM の認証システムに影響を与えないことを確認します。

1-11～1-14 では、リバプロ一台のホスト停止、再起動が OpenAM の認証システムに影響を与えないことを確認します。

1-15～1-18 では、リバプロ一台の Apache サービス停止、再起動が OpenAM の認証システムに影響を与えないことを確認します。

1-19～1-22 では、リバプロ一台の Tomcat サービス停止、再起動が OpenAM の認証システムに影響を与えないことを確認します。

表 8. セッションフェイルオーバ構成試験テストケース

	テストケース	テスト対象	
		シナリオ1	シナリオ2
2-1	別ブラウザで別OpenAMにリクエストが分散される(ただし、Sticky)	○	○
2-2	MQBrokerを停止しても、正常に使用できる	○	○
2-3	OpenAM1台停止後、未認証のクライアントからアクセスできる	○	○
2-4	自身のセッション情報を持たないOpenAMを停止後、OpenAMにアクセスできる	○	○
2-5	自身のセッション情報を持つOpenAM(およびMQBroker、amsessiondb)を停止	○	○
2-6	2-5でOpenAM再起動後、未認証のクライアントは正常に使用できる	○	○
2-7	2-5でOpenAM再起動後、認証済みのクライアントは正常に作業が続けられる	○	○
2-8	MQBrokerのみ停止しても、正常に使用できる	○	○
2-9	amsessiondbのみ停止しても、正常に使用できる	○	○
2-10	OpenLDAP1台停止後、認証済みのクライアントは正常に使用できる	○	○
2-11	OpenLDAP1台停止後、未認証のクライアントは正常に使用できる	○	○
2-12	OpenLDAP再起動後、認証済みのクライアントは正常に使用できる	○	○
2-13	OpenLDAP再起動後、未認証のクライアントは正常に使用できる	○	○
2-14	リバプロ1台停止後、認証済みのクライアントは正常に使用できる	-	○
2-15	リバプロ1台停止後、未認証のクライアントは正常に使用できる	-	○
2-16	リバプロ再起動後、認証済みのクライアントは正常に使用できる	-	○
2-17	リバプロ再起動後、未認証のクライアントは正常に使用できる	-	○
2-18	リバプロ1台のApache停止後、認証済みのクライアントは正常に使用できる	-	○
2-19	リバプロ1台のApache停止後、未認証のクライアントは正常に使用できる	-	○
2-20	リバプロのApache再起動後、認証済みのクライアントは正常に使用できる	-	○
2-21	リバプロのApache再起動後、未認証のクライアントは正常に使用できる	-	○
2-22	リバプロ1台のTomcat停止後、認証済みのクライアントは正常に使用できる	-	○
2-23	リバプロ1台のTomcat停止後、未認証のクライアントは正常に使用できる	-	○
2-24	リバプロのTomcat再起動後、認証済みのクライアントは正常に使用できる	-	○
2-25	リバプロのTomcat再起動後、未認証のクライアントは正常に使用できる	-	○
2-26	全てのMQ Brokerが停止しているとき、認証済みのクライアントは正常に作業	○	○
2-27	全てのMQ Brokerが停止しているとき、未認証のクライアントは正常に利用で	○	○
2-28	全てのamsessiondbが停止しているとき、認証済みのクライアントは正常に作	○	○
2-29	全てのamsessiondbが停止しているとき、未認証のクライアントは正常に利用	○	○

表 8 はセッションフェイルオーバ構成におけるテストケースを示します。

2-1 では、別のブラウザからアクセスすることにより、別の OpenAM にリクエストが分散されることを確認します。どの OpenAM にアクセスしているかは、OpenAM の管理コンソールに表示されているサーバ名で確認できます。

2-2 では、一台の MQ Broker を停止しても、OpenAM の認証システムに影響を与えないことを確認します。

2-3 では、一台の OpenAM を停止した後、まだログインをしていないブラウザからアクセスすることにより、認証システムが継続してサービスを提供していることを確認します。

2-4、2-5 では、ログイン済みのブラウザを利用して、セッション情報の有無による OpenAM サーバ停止の影響を確認します。

2-6、2-7 では、OpenAM サーバ再起動による影響を確認します。

2-8、2-9 では、OpenAM セッションフェイルオーバの構成要素である、MQ Broker、amsessiondb をそれぞれ一台ずつ停止しても、OpenAM の認証システムに影響を与えないことを確認します。

2-10～2-13 では、OpenLDAP サーバ一台の停止、再起動が OpenAM の認証システムに影響を与えないこと

を確認します。

2-14～2-17 では、リバプロ一台のホスト停止、再起動が OpenAM の認証システムに影響を与えないことを確認します。

2-18～2-21 では、リバプロ一台の Apache サービス停止、再起動が OpenAM の認証システムに影響を与えないことを確認します。

2-22～2-25 では、リバプロ一台の Tomcat サービス停止、再起動が OpenAM の認証システムに影響を与えないことを確認します。

2-26、2-27 では、全ての MQ Broker が停止していても、片側の OpenAM の認証システムだけは利用できることを確認します。

2-28、2-29 では、全ての amsessiondb が停止していても、片側の OpenAM の認証システムだけは利用できることを確認します。

3.2.5. 検証環境

検証環境も基本的に性能試験と同一の環境を利用して試験を実施します。

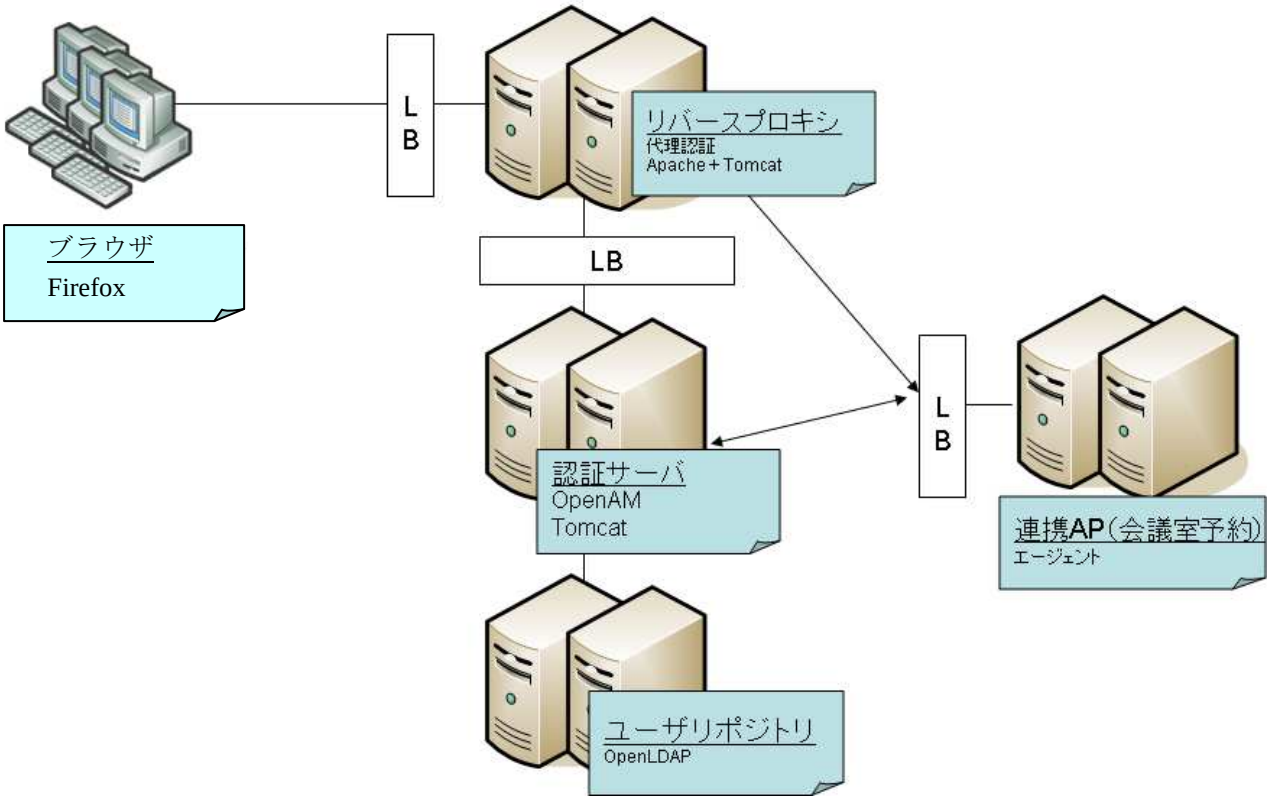

図 22. 可用性検証環境

可用性検証を行う環境としては、以下に留意しておく必要があります。

・AP サーバのセッションレプリケーション設定が必要
・負荷分散装置でリバプロの監視をするときは、Apache と Tomcat の両方のサービスを考慮が必要

3.2.6. 検証結果・考察

表 9. サイト構成試験結果

	テストケース	試験結果	
		シナリオ1	シナリオ2
1−1	別ブラウザで別OpenAMにリクエストが分散される(ただし、Sticky)	○	○
1−2	OpenAM1台停止後、未認証のクライアントからアクセスできる	○	○
1−3	自身のセッション情報を持たないOpenAMを停止後、OpenAMにアクセスできる	○	○
1−4	自身のセッション情報を持つOpenAMを停止すると、ログイン画面にリダイレクトされ	△	○
1−5	1−4でOpenAM再起動後、未認証のクライアントは正常に使用できる	○	○
1−6	1−4でOpenAM再起動後、認証済みのクライアントはログイン画面にリダイレクトされ	○	○
1−7	OpenLDAP1台停止後、認証済みのクライアントは正常に使用できる	○	○
1−8	OpenLDAP1台停止後、未認証のクライアントは正常に使用できる	○	○
1−9	OpenLDAP再起動後、認証済みのクライアントは正常に使用できる	○	○
1−10	OpenLDAP再起動後、未認証のクライアントは正常に使用できる	○	○
1−11	リバプロ1台停止後、認証済みのクライアントは正常に使用できる	−	○
1−12	リバプロ1台停止後、未認証のクライアントは正常に使用できる	−	○
1−13	リバプロ再起動後、認証済みのクライアントは正常に使用できる	−	○
1−14	リバプロ再起動後、未認証のクライアントは正常に使用できる	−	○
1−15	リバプロ1台のApache停止後、認証済みのクライアントは正常に使用できる	−	○
1−16	リバプロ1台のApache停止後、未認証のクライアントは正常に使用できる	−	○
1−17	リバプロのApache再起動後、認証済みのクライアントは正常に使用できる	−	○
1−18	リバプロのApache再起動後、未認証のクライアントは正常に使用できる	−	○
1−19	リバプロ1台のTomcat停止後、認証済みのクライアントは正常に使用できる	−	○
1−20	リバプロ1台のTomcat停止後、未認証のクライアントは正常に使用できる	−	○
1−21	リバプロのTomcat再起動後、認証済みのクライアントは正常に使用できる	−	○
1−22	リバプロのTomcat再起動後、未認証のクライアントは正常に使用できる	−	○

△停止直後は引き続きアクセス可能。しばらくするとログイン画面にリダイレクトされる

表 10. セッションフェイルオーバ構成試験結果

	テストケース	試験結果	
		シナリオ1	シナリオ2
2-1	別ブラウザで別OpenAMにリクエストが分散される(ただし、Sticky)	○	○
2-2	MQBrokerを停止しても、正常に使用できる	○	○
2-3	OpenAM1台停止後、未認証のクライアントからアクセスできる	○	○
2-4	自身のセッション情報を持たないOpenAMを停止後、OpenAMにアクセスできる	○	○
2-5	自身のセッション情報を持つOpenAM(およびMQBroker、amsessiondb)を停止	○	○
2-6	2-5でOpenAM再起動後、未認証のクライアントは正常に使用できる	○	○
2-7	2-5でOpenAM再起動後、認証済みのクライアントは正常に作業が続けられる	○	○
2-8	MQBrokerのみ停止しても、正常に使用できる	○	○
2-9	amsessiondbのみ停止しても、正常に使用できる	○	○
2-10	OpenLDAP1台停止後、認証済みのクライアントは正常に使用できる	○	○
2-11	OpenLDAP1台停止後、未認証のクライアントは正常に使用できる	○	○
2-12	OpenLDAP再起動後、認証済みのクライアントは正常に使用できる	○	○
2-13	OpenLDAP再起動後、未認証のクライアントは正常に使用できる	○	○
2-14	リバプロ1台停止後、認証済みのクライアントは正常に使用できる	－	○※1
2-15	リバプロ1台停止後、未認証のクライアントは正常に使用できる	－	○※1
2-16	リバプロ再起動後、認証済みのクライアントは正常に使用できる	－	○※1
2-17	リバプロ再起動後、未認証のクライアントは正常に使用できる	－	○※1
2-18	リバプロ1台のApache停止後、認証済みのクライアントは正常に使用できる	－	○※1
2-19	リバプロ1台のApache停止後、未認証のクライアントは正常に使用できる	－	○※1
2-20	リバプロのApache再起動後、認証済みのクライアントは正常に使用できる	－	○※1
2-21	リバプロのApache再起動後、未認証のクライアントは正常に使用できる	－	○※1
2-22	リバプロ1台のTomcat停止後、認証済みのクライアントは正常に使用できる	－	○※1
2-23	リバプロ1台のTomcat停止後、未認証のクライアントは正常に使用できる	－	○※1
2-24	リバプロのTomcat再起動後、認証済みのクライアントは正常に使用できる	－	○※1
2-25	リバプロのTomcat再起動後、未認証のクライアントは正常に使用できる	－	○※1
2-26	全てのMQ Brokerが停止しているとき、認証済みのクライアントは正常に作業	○	○
2-27	全てのMQ Brokerが停止しているとき、未認証のクライアントは正常に利用で	○	○
2-28	全てのamsessiondbが停止しているとき、認証済みのクライアントは正常に作	○	○
2-29	全てのamsessiondbが停止しているとき、未認証のクライアントは正常に利用	○	○

※ 1 AP サーバのセッションレプリケーション設定を行う必要がある

可用性試験については、サイト構成とセッションフェイルオーバ構成の２つの方式についてそれぞれ想定した通りの結果を確認できました。

サイト構成の場合、ログインしたユーザのセッション情報は、OpenAM のサーバ間で共有されないので、一台の OpenAM サーバを停止した場合、そのサーバが保持していたユーザの処理は継続できず、新たにログインをし直す必要があります。ただし、ログインしていないユーザが新たにログインをすることには影響を与えません。

セッションフェイルオーバ構成の場合、OpenAM のサーバ間でセッション情報を共有するため、一台の OpenAM サーバを停止した場合でも、そのサーバがセッション情報を保持していたユーザの処理を継続することができます。構成要素である MQ Broker と amsessiondb のインスタンスがサイト内に最低限１台ずつ残っていれば機能を果たすことが確認できました。しかし、構成要素が複雑になる、OpenAM を 3 台以上に増

やした際にスケールするかなどの懸念もあります。

システムの可用性として、OpenAM サーバの障害時にユーザの再ログインを許容できるかがどちらの構成を選択するかのポイントに繋がります。再ログインをすることによりサービスが継続できるレベルの可用性であれば、シンプルなサイト構成を選択することができます。障害発生時にユーザの再ログインを許容できないのであれば、セッションフェイルオーバ構成を選択する必要があります。

次期バージョンの OpenAM ではセッションフェイルオーバのアーキテクチャが見直され、Message Queue などを必要としないシンプルな構成が予定されています。新しいセッションフェイルオーバ機能に関しては、改めて検証する機会を設けたいと考えています。

3.3. セキュリティに関する考察

OpenAM はアーキテクチャ上、ユーザはログインの際に OpenAM の認証画面に直接アクセスする必要があります。そのため、外部ネットワークからの接続をユーザに対して提供する場合は、OpenAM サーバ自体を DMZ に配置する必要があります。その際には、外向きの FireWall により OpenAM サーバへのアクセスを制限する、OpenAM サーバでの余分なサービスを上げない、リモートからの OS ログインを制限するなど一般的なセキリティ対策は必須となります。

また OpenAM では、標準のフル機能を備えた WAR モジュール以外に、管理コンソールのみ・管理コンソール以外、など用途を制限して配置可能となっています。DMZ には管理コンソール以外の機能を提供する OpenAM を配置し、内部ネットワークに管理コンソールのみを配置すれば、よりセキュリティを向上することができます。

認証 DB となる OpenLDAP などのユーザリポジトリは、直接ユーザからアクセスする必要は無いので、内部ネットワークへ配置し、OpenAM サーバからのみアクセスできるように内部 FireWall で制限をしておくべきでしょう。

まとめると、セキュリティを考慮した OpenAM の構成に関する考慮点は以下のようになります。

・OpenAM サーバは、DMZ に配置する際は管理コンソールの機能を外したものとする
・ユーザリポジトリは、内部ネットワークに配置

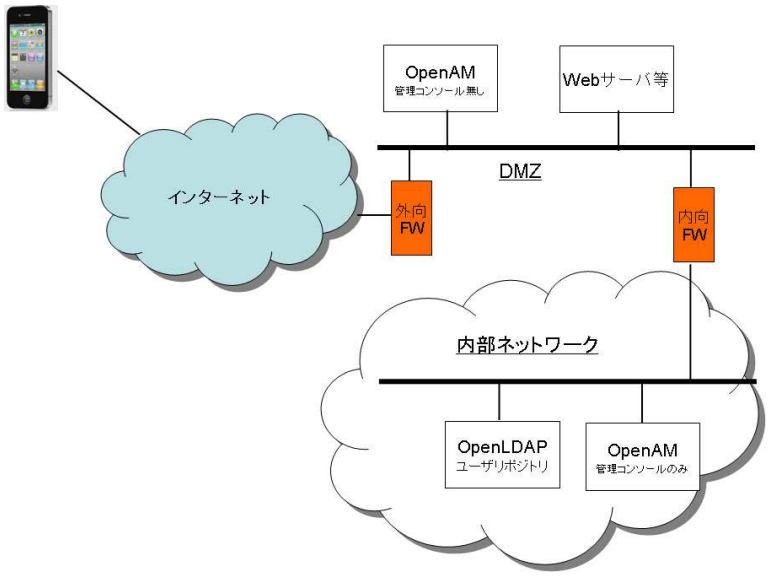

図 23. セキュリティを考慮した OpenAM の構成

OpenAM は Java を中心に実装された Web アプリケーションを基本構成としているため、一般的な Web ア

プリケーションの脆弱性（パラメータ改ざん、SQL インジェクション、など）が存在するリスクもあります。そのような脆弱性を確認するための手法としてチェックツールを使う方法があります。なお OpenAM の前身である、OpenSSO はセキュリティの国際規格である ISO/IEC15408(Common Criteria)に準拠しています[1]。従って OpenAM も同様に準拠していることが期待できます。

ここでは、Google により開発・提供されている脆弱性チェックツール ratproxy と skipfish を使って OpenAM のチェックを実施しました。これらのツールには、大規模なインターネットサービスを提供する Google 社が実際に遭遇した攻撃パターンなどのノウハウが蓄積されており、また OSS として提供されているため広く使われています。ratproxy は proxy 型の脆弱性チェックツールで、ratproxy をプロキシとして設定して Web アプリケーションを操作することで、XSS(Cross Site Scripting)や不適切な XSRF(Cross Site Request Forgeries)対処など、各種の脆弱性を検出できます。skipfish は、指定した URL をクロールすることで同様の脆弱性を検出できます。表 11、表 12 にそれぞれのツールによる脆弱性のチェックを OpenAM の管理コンソールに対して実施した結果を示します。

表 11. ratproxy1.58 によるチェック結果

リスク	重要度	件数	リスクの説明	評価
①XSRF 対策をしていない POST クエリ	高	3件	セキュリティトークン無しの POST リクエストパラメータを受け付けている。XSRF 攻撃を受ける可能性がある	△
②不審なパラメータ渡し方法	中	3件	パラメータ名が OGNL 式や PHP グローバル変数、その他の直接サーバ側のコードに影響を与える仕組みに見える。適切なセキュリティ対策をしてないとデータインジェクションの可能性がある。	○
③クエリパラメータ内のファイルパス	中	1件	ファイルパスのようなクエリパラメータをエコーバックしていない。これは必ずしも脆弱性でないが、ディレクトリトラバーサルテストのための高いリスク対象となる。	○
④XSRF 対策をしていない Cookie	中	1件	新しい Cookie を受け付けるがセキュリティトークンが無い。セッションフィクセーションやその他の重要な攻撃を受ける可能性がある。	△

○：問題なし、△：要検討、×：問題有り

表 12. skipfish2.05b によるチェック結果

リスク	重要度	件数	リスクの説明	評価
⑤シェルインジェクション	高	4 件	`true` や `false` のレスポンスが `uname` と異なる。従ってシェル呼出を行なっていない場合は誤検出となる。	○
⑥サーバサイド XML インジェクション	高	1 件	<sfish></sfish> のレスポンスが </sfish><sfish> と異なって見える。XML を使って無い場合は誤検出となる。	○
⑦ディレクトリトラバーサルの可能性	中	1 件	.¥val のレスポンスが ..¥val のレスポンスと異なって見える。ファイル操作を行なっていなければ誤検出となる。	○

○：問題なし、△：要検討、×：問題有り

表 11 より、①、④で XSRF 対策に高・中のリスクが指摘されています。XSRF の根本的な対策は OpenAM

[1] http://docs.oracle.com/cd/E19316-01/820-3886/ghhsp/index.html

側での対応を待つ必要があります。しかし、XSRF の脆弱性に対しては、現状の OpenAM を利用する場合は、管理コンソールを扱えるユーザを制限し、かつ管理者は外部の不明なサイトへアクセスさせないという基本的な対策を講じる事で対処可能です。②、③に対しては内部でパラメータのチェックを行なっているため脆弱性はありません。表 12 に関しては、リスクの説明にあるように、OpenAM には脆弱性は無く今回のチェックでは誤検出でした。

3.4. OpenAM の運用面

ここでは運用面に関して OpenAM が提供する機能・考慮する点を紹介します。

3.4.1. バックアップとリカバリー

OpenAM の設定データをバックアップするには、以下の 2 つを考慮する必要があります。

- ファイルシステム上の設定ファイルのバックアップ
- OpenAM の設定データストアに保存されたデータのバックアップ

ファイルシステム上の設定ファイルは、通常のバックアップツールやアーカイブコマンドを利用してバックアップする事ができます。リカバリーする際も使用するバックアップツールやアーカイブコマンドの方法に従います。

3.4.1.1. OpenAM の設定データストアのバックアップ

OpenAM の管理コンソールで設定した情報は、設定データストアに保存されます（デフォルトでは付属の OpenDS）。これらは特殊な LDAP スキーマに保存されているため、バックアップ用のユーティリティが用意されています。

バックアップには、ssoadm[2]コマンドを以下のように使用します。

```
# $SSOADM_PATH/bin/ssoadm export-svc-cfg –u amadmin –f パスワードファイル –e 暗号化パスワード –o /tmp/openam-backup.xml
```

パスワードファイルには、管理者のパスワードを保存しておきます。このファイルは所有者のみが読取りできるパーミッションに設定しておく必要があります。また、バックアップしたデータは XML ファイルとなるのでパスワード情報などが平文で含まれているとセキュリティ上問題があります。そのため暗号化パスワードを指定することにより、XML ファイル内のパスワード情報などは暗号化されて保存されます。

このように設定データストアの情報をバックアップしておくことにより、ハードウェア障害の他、管理者の設定ミスなど人為的な障害にも備えることができます。

3.4.1.2. OpenAM の設定データストアのリカバリー

設定データストアのリカバリーには、前節で作成したバックアップファイルを利用します。リストア用のコマンドも ssoadm コマンドを使用します。

```
# $SSOADM_PATH/bin/ssoadm import-svc-cfg –u amadmin –f パスワードファイル –e 暗号化パスワード –X /tmp/openam-backup.xml
```

[2] ssoadm コマンドをサイト構成で利用する場合、正常に動作しない問題があります。詳細は「ssoadm コマンドをサイト構成で利用する際に発生する問題」を参照ください。

暗号化パスワードは、バックアップ時に指定したものと同じものを指定する必要があります。

バックアップ・リカバリーの仕組みを、テスト環境から本番環境へ移行するために利用することも可能です。バックアップした設定データストアの情報は XML ファイルとなっていますので、管理コンソールで行うよりも設定の変更が容易にできます。

3.4.2. OpenAM のログ監視

OpenAM では、コンポーネント毎にデバッグログの出力を行なっており、例外のスタックトレースなどの情報を障害時解析などに利用することができます。ログの粒度を示すデバッグレベルは、実行時に変更することができるので、トラブルシューティング時などに必要なときにだけ、詳細なデバッグログを出力するように、動的に設定することが可能です。また、デフォルトではコンポーネント毎に別ファイルにデバッグログを出力するようになっていますが、一つのファイルに出力するように変更することも可能です。

デバッグログと同様に監査ログ出力も行なっています。監査ログもコンポーネント毎にアクセスファイルとエラーファイルの 2 つずつを出力しています。例えば、SSO に関する監査ログは amSSO.access と amSSO.error となります。監査ログの有効化・無効化、ログの粒度も実行時に変更することが可能です。運用の際にはログファイルの最大長や、履歴ファイルの数を考慮する必要があるでしょう。監査ログの書式は、一般的なタブ区切りのテキストファイルですので別途監視ツールや集計ツールなどでも利用することは容易ですが、OpenAM では特別なツールが用意されているわけではありません。

また、デフォルトでは監査ログはファイルベースになっていますが、RDBMS のテーブルに出力するように設定することも可能です。大規模なユーザが使う場合、ファイルベースの監査ログよりも RDBMS の監査ログの方が適しているかもしれません。

ファイルベースのログには、署名により保護されたログを出力する機能もあります。この機能によりログの改竄を防ぐことが可能となります。出力されたログに対して amverifyarchive コマンドを使用して妥当性を検証することで改竄されていないかのチェックを行います。

3.4.3. ユーザ管理

OpenAM の管理コンソールを利用してユーザの追加・修正・削除などを行うことも可能ですが、大量のユーザのメンテナンスを行うには管理コンソールは適していません。

ユーザデータストアに OpenLDAP を利用している場合は、LDIF ファイルを用意して、ldapmodify コマンドを使用して一括で変更することが可能です。

企業の基幹システムの認証基盤として使用するのであれば、何らかの IDM(ID 管理)機能を利用することが望まれます。OpenAM では IDM の機能を提供していませんが、OpenAM をベースとした独自の製品・ソリューションを展開している各社は IDM 機能を提供しています。Forgerock 社も IDM 機能を提供する、OpenIDM を開発中です。

4. まとめ

本レポートでは、OpenAM を中心として OSS を活用した認証システムを構築可能であるか否かを、様々な観点から検証しました。

検証では、OpenAM を中心に据えたシングルサインオン（SSO）による認証システムを構築し、システム全体の評価を行う事を目標としました。評価指標は機能、信頼性、可用性、運用性という基本的なシステム評価尺度に基いています。

加えて OpenAM ソフトウェア自体の脆弱性およびその対策にも触れていますが、認証システムへはインターネットなどからアクセスする場合が想定される為であり、安全なシステム構築の為には必要であると考えた為です。

機能検証でのシステム構成は実用性が高いと言われているエージェント方式とリバースプロキシ方式の２方式としました。また、認証対象は社内システムとインターネットサービス（Salesforce、Google など）とし、被認証側は PC とスマートデバイス（iPhone）という構成にする事で、最近の新しい認証システムの利用形態を想定しました。

性能検証では、アプリケーションの負荷が性能に与える影響は大きいですが、アプリケーション部分の処理負荷を事前に想定する事により、可能な限り認証システム側の性能を測定する事に配慮しました。

検証環境のハードウェアスペック（仮想マシン HP 社製（CPU：Intel　Xeon2.0GH ｚ × １、メモリ ２GB）の２台構成で、同時 1,000 ユーザ程度の認証処理を行っています。

また、システムへの処理要求が増加した際の対応方法についても検証過程において明らかにする事ができました。

可用性検証では、サイト構成（コールドスタンバイ方式に相当）とセッションフェイルオーバ（アクティブスタンバイ方式に相当）構成を構築して、システム障害時の対応動作を確認しました。ただし、セッションフェイルオーバ構成は、障害時の復旧時間は短縮できるものの、OpenAM の次バージョンではソフトウェア構造の変更が予定されており、再検証が必要です。

システム運用については、OpenAM 自体は多くの設定作業を必要としない為、複雑な作業は発生しません。しかしながら、認証対象の ID の追加、削除、修正などの運用作業を効率的に行う為には、OpenAM の機能だけでは不足している為、別途ソフトウェア製品を利用するか独自のツールの開発が必要となります。

認証システムは、インターネットを経由してアクセスする際には DMZ などに配置される事になります。
その際には OpenAM がセキュリティ上の脅威にさらされることになるため、本検証では OpenAM のソフトウェアとしての脆弱性についても各種セキュリティチェックツールを利用して評価しました。結果、システム構築時の留意点、対応策を提示していますので是非参考にして頂きたいと思います。

本検証を通じて、OpenAM を利用して実用性の高い認証システムを構築する事が可能です。ただし、商用製品と比較した場合には認証 ID の管理機能を具備しておらず、現時点で OpenAM の導入を検討する際には注意が必要です。しかしながら、OpenAM の開発元 Forgerock 社は今後、認証 ID 管理用ソフトウェア（OpenIDM）を提供予定であり、認証システム全てを OSS で実現する日も間近になっています。

最後に本レポートの結果を活用して頂き、社内でオープンソース・ソフトウェアの利用が進むと共に、多くのソフトウェアが評価された事例と知見が、当社の財産となる事を願っています。

5. 参考情報

5.1. 認証基盤製品比較

表 13. 認証基盤製品比較

	OpenAM（OSS 版）	OpenAM（製品版）	A 社	B 社	C 社	D 社	E 社
エージェント型 SSO	○	○	○	○	○	○	△（※1）
リバースプロキシ型 SSO	○	○	○	○	○	○	○
代理認証型 SSO	×	○（※2）	○	○	○	○	○
Windows デスクトップ SSO	○	○	△（※1）	△（※1）	△（※1）	△（※1）	△（※1）
マルチドメイン対応	○	○	○	○	○	○	○
SAML 対応	○	○	○	○	△（※1）	○	△（※1）
アクセス制御	○	○	○	○	○	○	○
OpenID 対応	○	○	×	×	△（※1）	○	×
ライセンス価格（4 万 ID 時）	無料	600 万円(※3)	約 7.1 億円	約 9,000 万円	約 7,000 万円	約 6,000 万円	約 3,200 万円

（2010 年度野村総合研究所調べ）

※1：△はオプション、※2：標準機能ではなく OpenAM を提供している各社(NRI など)独自拡張して提供

※3：Enterprise Edition の 2 台構成を想定。ユーザライセンスは不要でパッケージライセンスのみでよい

5.2. OpenAM を使用する上での留意点

今回の機能検証では、OpenAM9.5.3 を使用していましたが、そこで発見した２つの問題を留意点として報告します。それぞれ内容の詳細は OpenAM の環境構築ガイドを参照してください。

5.2.1. クライアント証明書認証を使う上での留意点

OpenAM9.5.3 はクライアント証明書認証を使う場合、必ず CRL（証明書失効リスト）を設定する必要があります。

検証環境では、CRL を使用していなかったため問題が発生しましたが、クライアント証明書による認証を実運用で利用する場合、通常 CRL を設定することになりますのでこの問題は発生しません。

5.2.2. 自動作成された Fedlet を使用する上での留意点

OpenAM の管理コンソールで生成された Fedlet を利用する場合、自動生成された雛形のソースコードを修正する必要があります。

検証において、自動生成された Fedlet を利用した際に、そのままでは動作に問題がありました（画面のレスポンスが返ってこない)。自動生成されたソースコードを修正することにより正常に Fedlet が動作することが確認できました。

5.3. Basic 認証 AP に対する代理認証

今回検証した NRI の代理認証モジュールは、Form 認証の AP にしか対応していません。

Basic 認証 AP に対する代理認証に対応する方法が OpenSSO のコミュニティで紹介されています。(http://java.net/jira/browse/OPENSSO-2889)。今回の検証では動作を試していませんが参考情報としてここで取り上げます。

OpenAM には認証の後処理として独自のモジュールで処理を追加する機能が用意されています。この認証後処理において Basic 認証に必要な情報を生成し、ポリシーエージェントでその情報を元に Authentication ヘッダを付加することで、Basic 認証を必要とするアプリケーションに対応する事が可能となります。

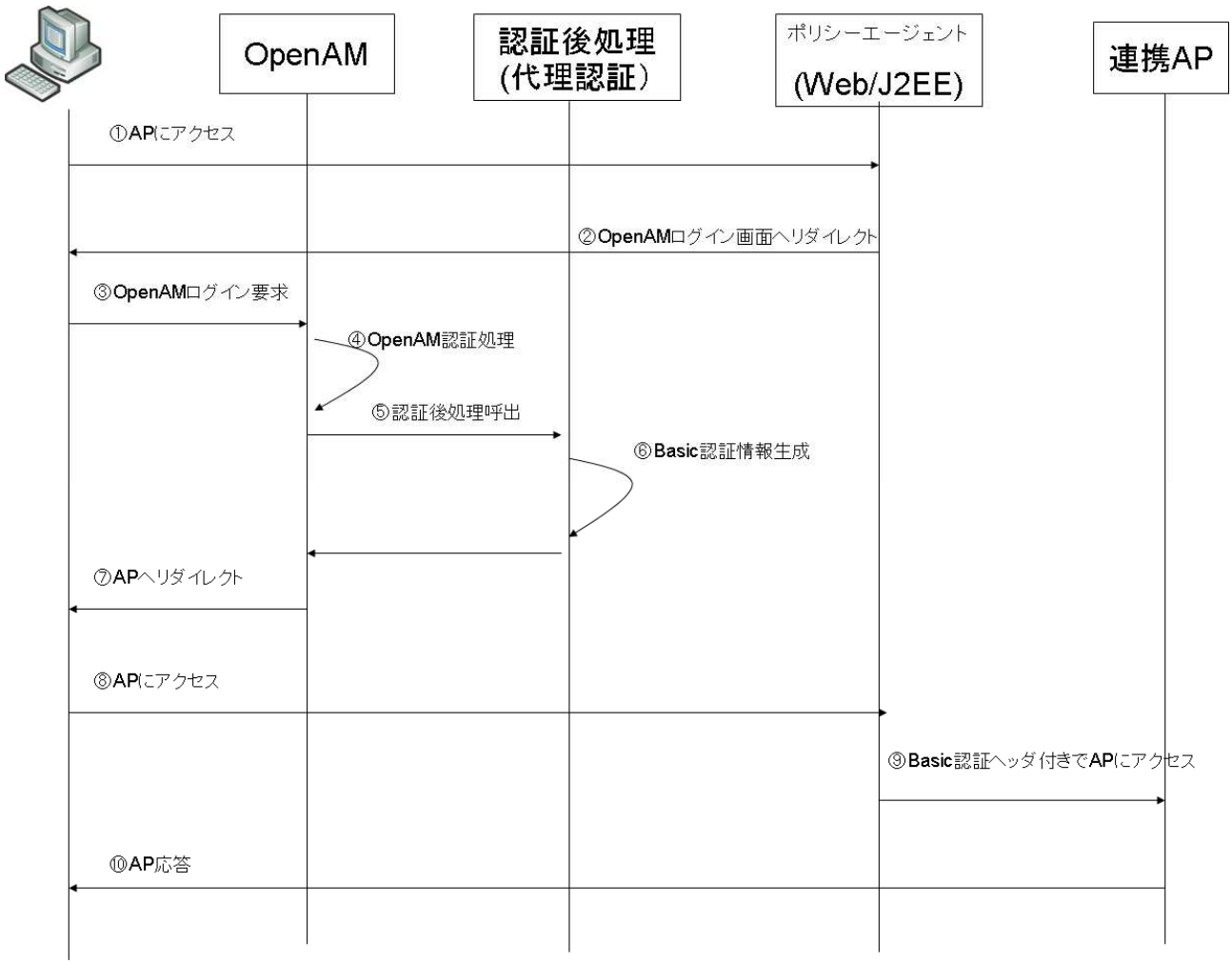

図 24. Basic 認証における代理認証の処理の流れ

5.4. セッションフェイルオーバアーキテクチャ

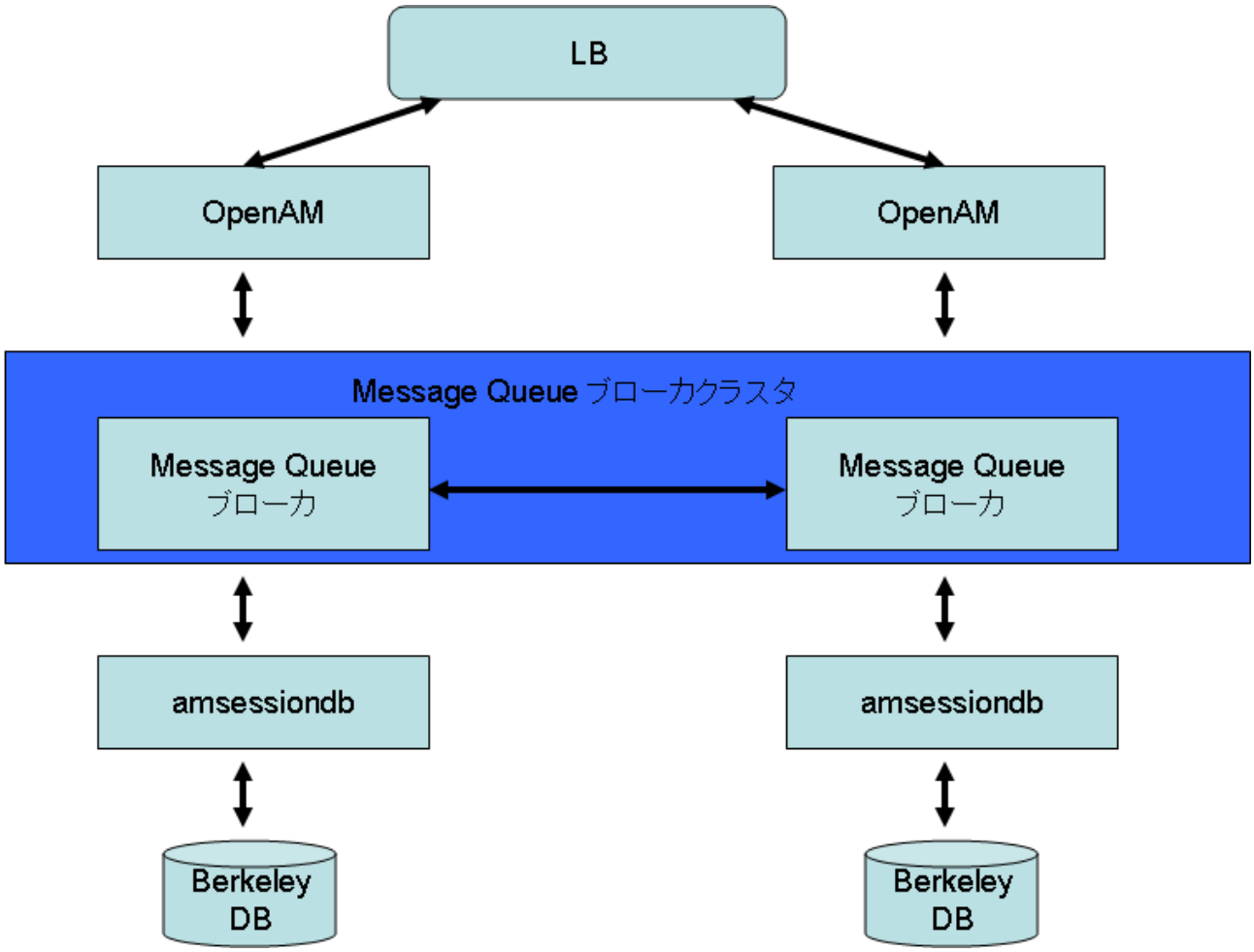

図 25. セッションフェイルオーバアーキテクチャ

OpenAM のセッションフェイルオーバのアーキテクチャは図 25 のようになっています。構成要素は以下の通りです。

- OpenAM
  - ユーザセッションの操作（作成、更新、削除）の際に SessionService を経由して Message Queue ブローカとやり取りする
- Message Queue ブローカ
  - OpenAM と amsessiondb の間でメッセージの中継を行う
  - 実体は GlassFish Message Queue Service(OpenMQ)
- amsessiondb
  - OpenAM からのリクエストを受取、セッション情報を Berkeley DB に格納する。メッセージは Message Queue ブローカ経由でやり取りする

5.4.1. Message Queue ブローカ

JMS（Java Message Service）の実装の一つである GlassFish Message Service を利用したクラスタリング構成を取っています。GlassFish Message Service は、以下の 2 つのクラスタリングモデルをサポートしています。

- Conventional broker clusters(サービス可用性)
- Enhanced broker clusters(サービス可用性＋データ可用性）

OpenAM のセッションフェイルオーバでは前者のモデルを利用しており、データ(メッセージ)に関する可用性は保証していません。

Message Queue ブローカクラスタのクライアントは、接続しているブローカとの通信に障害が発生すると自動的に再接続されます。これにより、Message Queue ブローカに障害が発生しても、サービスは継続されます。

OpenAM のセッションフェイルオーバでは以下の 3 つの Message Queue の宛先を、いずれも Pub/Sub(1:n)モデルで利用します。

- AM_DBREQUEST（OpenAM からのリクエスト）
- AM_DRESPONSE(amsessiondb からのレスポンス)
- AM_DBNODE_STATUS(amsessiondb のステータス)

5.4.2. amsessiondb

Oracle BerkeleyDB Java API を利用してセッション情報の永続化を行います。com.sun.identity.ha.jmqdb.client.FAMHaDB をメインクラスとする Java アプリケーションとなっています。

process()メソッドにより以下の処理を繰り返します。

- Message Queue AM_DBREQUEST を Subscribe し OpenAM からのリクエストを受信
- リクエストの内容に従い BerkeleyDB を操作
- 結果を AM_DRESPONSE 宛に Publish

amsessiondb は、サイト内の他の amsessiondb の状態を監視し、一番長く稼働しているものがマスターとなります。

- AM_DBNODE_STATUS を Subscribe し、Active な amsessiondb の情報を取得すると同時に自分の情報を Publish
- NodeStatusSender, NodeStatusReceiver2 つのスレッドを利用
- 1 秒間隔でチェック、5 秒間レスポンスが無いと死んでいると判断

amsessiondb のマスター特有の機能は以下の通りです。

- READ 処理で not found のレスポンスを返す
- GET_RECORD_COUNT 処理のレスポンスを返す

5.4.3. OpenAM

OpenAM 内部では、AMSessionRepository インタフェース(実装クラスは JMQSessionRepository)が amsessiondb のプロトコルに対応しています。Message Queue Broker への送受信は FAMRecordJMQPersister が担当します。(FAMRecord send(FAMRecord famRecord)メソッド)

留意点として、Message Queue への送受信時に発生したエラーはログに出力するだけであり、クライアントには報告されません。従って、セッションフェイルオーバが機能していない場合でも、OpenAM はサービスを継続することができます。

5.4.4. 処理の流れ

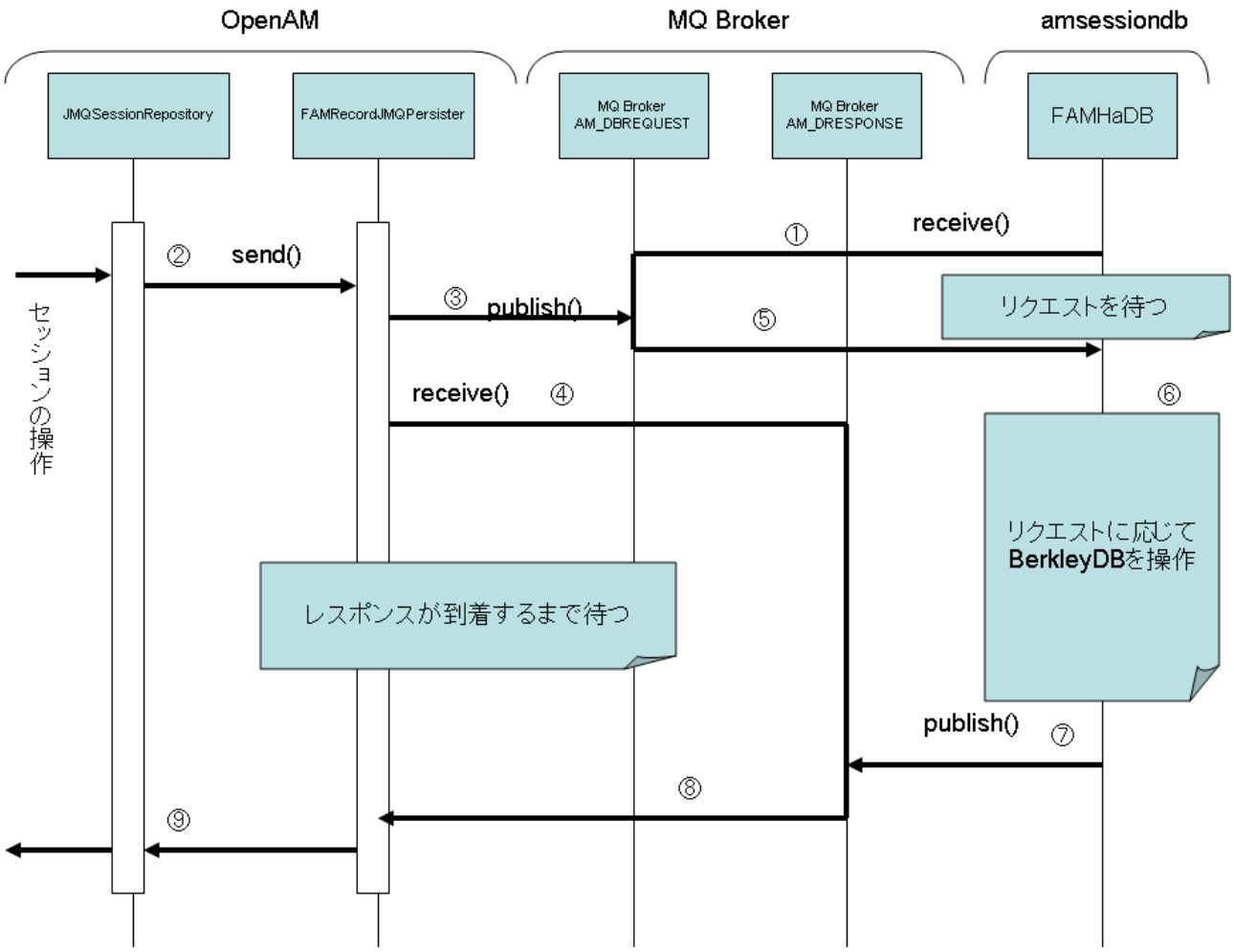

図 26. セッションフェイルオーバ処理の流れ

① amsessiondb では、FAMHaDB クラスが receive()メソッドで MQ Broker の AM_DBREQUEST からリクエストの到着を待ちます。

② OpenAM ではセッションの操作の際に JMQSessionRepository クラスの send()メソッドでセッション情報を送信します。

③ FAMRecordJMQPersister クラスは publish()メソッドで MQ Broker の AM_DBREQUEST 宛にメッセージを送信します。
④ メッセージ送信後、receive()メソッドで MQ Broker の AM_DRESPONSE 宛に amsessiondb からのレスポンスが到着するのを待ちます。
⑤ amsessiondb は OpenAM から到着したリクエストを受信します。
⑥ amsessiondb は、受診したリクエストに応じて BerkeleyDB を操作し、セッション情報を永続化します。
⑦ amsessiondb は publish()メソッドでレスポンスを MQ Broker の AM_DRESPONSE 宛に送信します。
⑧ FAMRecordJMQPersister クラスがレスポンスを受信します。
⑨ JMQSessionRepository クラスの send()メソッドは受信した結果を戻り値として返します。

5.4.5. セッションの操作とプロトコルの対応

以下に OpenAM のセッション操作と amsessiondb におけるプロトコルの対応を示します。

表 14. セッション操作とプロトコルの対応

セッション操作	AMSessionRepository	amsessiondb プロトコル
セッション情報取得	InternalSession retrieve(SessionID sid)	READ
セッション情報保存	void save(InternalSession is)	WRITE
セッション情報削除	void delete(SessionID sid)	DELETE
期限切れセッション情報の削除	void deleteExpired()	DELETEBYDATE
ユーザに紐付くセッションの有効期限を取得	Map getSessionsByUUID(String uuid)	GET_RECORD_COUNT

5.5. 負荷分散装置の構成詳細

今回の検証では、Linux システムにおける負荷分散ソリューションである LVS(Linux Virtual Server)を中心に用いて検証環境を構築しました。LVS の他にも keepalived や iptables などのソフトウェアコンポーネント（OSS）を以下の用途で組み合わせて、負荷分散装置全体の機能を実現しています。

- LVS を用いたネットワークトラフィック分散
- keepalived を用いたサービス監視と障害サーバ回避
- iptables を用いた仮想 IP アドレス実現

以下でそれぞれについて説明します。

5.5.1. LVS (Linux Virtual Server)

・概要

LVS は、linux kernel においてパケットレベルの振り分けを行う機能です。Kernel レベルで機能が実装されており、トランスポートレイヤ（Layer-4）での振り分けを行うことができます。LVS を利用するためには、Linux2.6.10 以上の kernel が必要です。今回の OpenAM 検証環境においては、CentOS6.0 に含まれる Linux2.6.32 を利用しました。

・負荷分散ネットワーク構成

LVS では以下の３方式の負荷分散ネットワーク構成をサポートしています。

(1) NAT 経由仮想サーバ（LVS/NAT）

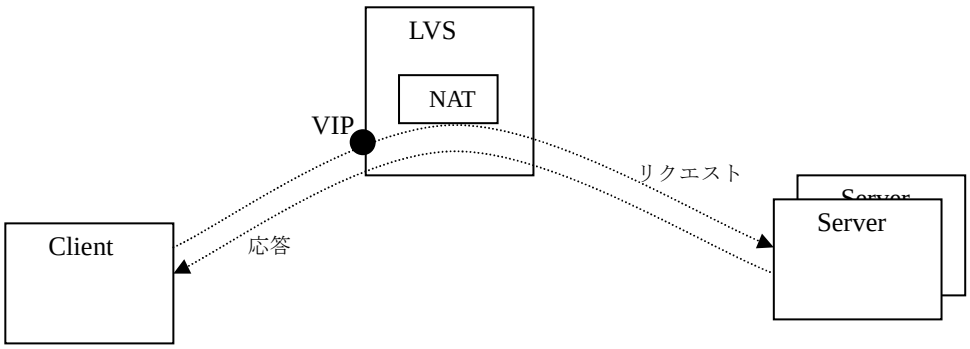

図 27. NAT 経由仮想サーバ(LVS/NAT)構成

一度 LVS の仮想アドレス（VIP）でリクエストを受けた後、宛先アドレス変換（NAT）してから Server にリクエストを送ります。Server からの応答は LVS を経由してクライアントに返されます。Server を構築する際に、LVS 配下にあることを特に考慮する必要がなく、普通にサーバを並べればいいというのがこの構成のメリットです。また、Server-Client 間のすべてのトラフィックが LVS を通ることになるので、LVS

が性能的なボトルネックになってしまうというデメリットがあります。

(2) IP トンネル経由仮想サーバ（LVS/TUN）

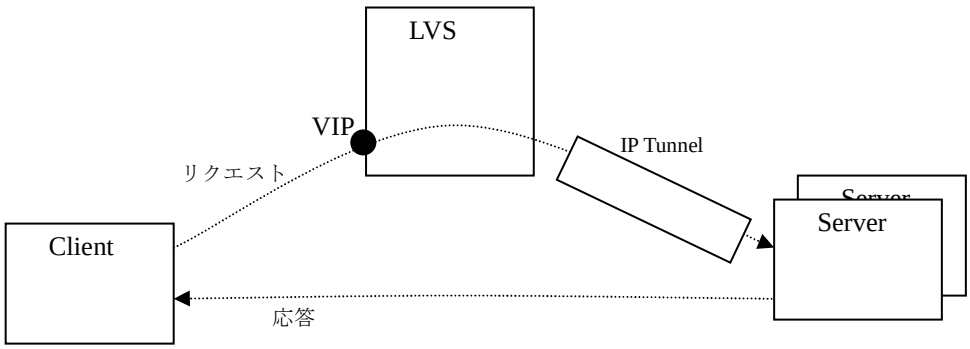

図 28. IP トンネル経由仮想サーバ（LVS/TUN）構成

LVS の仮想アドレスで受けたリクエストを、サーバに向けた IP トンネルで送る方式です。この方式では、Server からの応答は直接 Client へ返されるため、LVS がボトルネックにならないメリットがあります。また、LVS と全ての Server の間にそれぞれ個別に IP トンネルを構築・設定して維持しなければならないことや、すべての Server が Linux と同方式の IP トンネルプロトコルをサポートしなければならないという構成上の制約がある点がデメリットになります。

(3) ダイレクトルーティング（LVS/DPT）

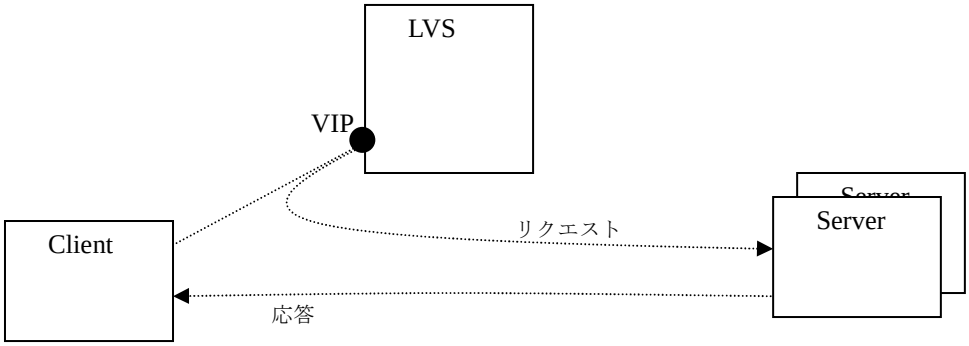

図 29. ダイレクトルーティング（LVS/DPT）構成

LVS/DPT 方式の場合は、Client からリクエストを受ける LVS のインタフェースと Server のインタフェースが同一サブネットに存在しなければならないという構成上の制約があります。動作としては、LVS で受けたリクエストパケットを NAT によるアドレス変換や IP トンネルによる IP ヘッダ付与といった処理を一切行わないで、そのまま直接 Server に転送してしまいます。Server からの応答は Client へ直接返されます。注意すべき点としては、LVS と Server は同じ VIP アドレスを受信するにもかかわらず、同一サブネットで接続されているという点が挙げられます。最初に Client 側から VIP にパケットが送られるときに ARP リクエストがブロードキャストされ、LVS と（複数の）Server 全体に ARP リクエストが届きますが、LVS 以外はこのリクエ

ストに応答してはなりません。しかし、LVS より中継された（中身が全く同一の）パケットは LVS が選定した Server が同一サブネット経由で受信する必要があります。このあたりの L2 の挙動に基づく構成が複雑になってしまうというのが LVS/DPT 方式のデメリットといえます。 この問題に対応する方法としては、Server にて VIP をループバックインタフェースに設定し、同時に ARP に応答しないような特殊な設定をおこなうという方法や、iptables を用いて実現する方式があります。今回の検証では、両パターンについて試行を行いましたが、後者の iptables を使う方式のほうが設定項目がシンプルで、安定的に稼働させることができました。iptables を用いた方式については後述します。

以上、３方式のうち、今回の OpenAM 検証においては、一般的に最も実用的である LVS/DPT 方式で検証を行いました。

・負荷分散方式

LVS では複数のサーバに対して処理を振り分けるスケジューリング方式として、Round-Robin、Weighted Round-Robin、Least-Connection、Weighted Least-Connection の４方式をサポートしています。

(1)Round-Robin

Round-Robin スケジューリングでは、すべてのサーバに対して順番にコネクションを割り振る方式です。サーバのレスポンス時間やコネクション数の違いにかかわらず、すべてのサーバに対して単純に順番に新コネクションを割り振ります。

(2)Weighted Round-Robin

Weighted Round-Robin スケジューリングは、サーバごとに異なる重みを割り当てることができます。重みは整数値でデフォルトは１です。サーバに設定した重みの割合に応じてコネクションを順番に割り振ります。

(3)Least-Connection

Lest-Connection スケジューリングはアクティブなコネクションが一番少ないサーバに新たなコネクションを割り振る方式です。性能が同じ性能のサーバを並べた場合にはうまく平等に負荷を分散することができます。性能に違いがあるサーバを並べた場合でもコネクション数に応じた割り振りが行われるためうまく負荷が分散されるように思えますが、実際には TCP の Time-Wait ステートが２分間続くため、うまく負荷に応じた分散はできません。

(4)Weighted Least-Connection

Weighted Least-Connection スケジューリングはアクティブなコネクション数に応じた割り振りをサーバ毎に異なる重みづけで行うことができます。

・Session Persistence

上記４方式のいずれの負荷分散方式においても、Client より新たなコネクションリクエストが送られると、毎回新たに Server 選定プロセスが実行され、同一 Client から複数 Server にコネクションが張られることになりますが、通常の Web アプリケーションではセッションを維持する必要があるため、Client に対する Server を一度決めたらそれ以降のリクエストは同じサーバに送ることが必要になります。このようなセッション維持の方式には、URL パラメータを用いるものやクッキーを用いるものや IP アドレス情報を利用

する方式がありますが、LVS ではソース IP アドレス方式のみサポートしています。

今回の OpenAM 検証における LVS 構成は以下のとおりとしました。
- 負荷分散ネットワーク構成： LVS/DPT
- 負荷分散方式： Least-Connection
- Session Persistense： あり

OpenAM を LVS で負荷分散する場合には、この構成をとることにより負荷分散がうまく行われることを確認しました。

LVS を動作させるためには、LVS に対応したカーネルを動作させた上で、ipvsadm コマンドによって、カーネル内の振り分けテーブルをひとつひとつ設定する必要がありますが、後述する keepalived などの冗長化ソフトウェアを用いて自動的に LVS の振り分けテーブルを設定するのが一般的です。今回の OpenAM 検証においては keepalived を用いて LVS 設定を行いました。

5.5.2. keepalived

LVS を用いることで負荷分散機能そのものは実現することは可能ですが、LVS には実サーバを監視する機能がなく、実サーバに障害が発生してもその実サーバにリクエストを送り続けてしまいます。この課題に対応するためには別途冗長化ソフトウェアを組み合わせて負荷分散環境を構築する必要があります。今回の OpenAM 検証では、keepalived を用いた方式について検証を行いました。

keepalived は LVS と組み合わせて冗長化機能を実現するための OSS で、keepalived を動作させることにより以下の機能が実現されます。

1. LVS の振り分けテーブルの制御
2. 実サーバの死活チェック
3. LVS 自体の VRRP による冗長化

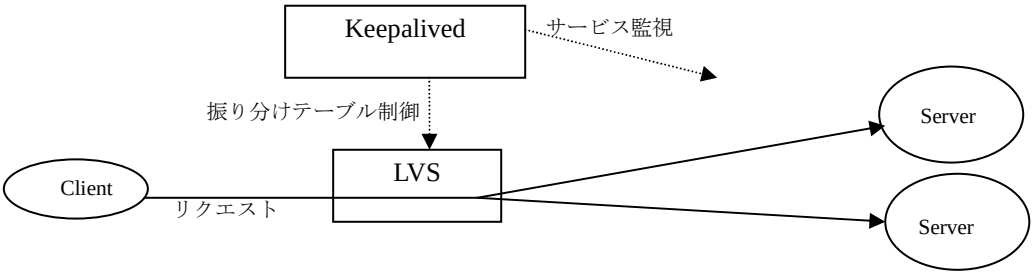

図 30．Keepalived の構成

今回の OpenAM 検証では 1 と 2 について確認を行いました。実際のサービス環境において keepalived を用いる場合には keepalived 自身の冗長化を行ったほうがいいと思いますが、その場合は事前検証を行うべきです。

5.5.3. iptables を用いた L2 特殊処理

LVS の章で述べましたが、今回の OpenAM 検証においては LVS の負荷分散ネットワーク構成として LVS/DPS を採用したため、L2 ネットワーク的に特別に考慮しなければならない点があります。

以下の図は、Client、LB、Server1、Server2 は同一のブロードキャストドメイン（IP サブネット）に接続されている状態を例示したものです。各インタフェースにはそれぞれ同一サブネット内の IP アドレスが設定されており、LB には Real IP アドレスに加えて、Client からのリクエストを受けるための Virtual IP アドレス（VIP）の設定がされています。Client から VIP 宛のパケットを送るとき、まず L2 アドレス解決のために Client より ARP リクエストがブロードキャストされ、これに LB が応答し、VIP 宛のパケットはまず LB に送られることになります。Server1 と Server2 についても LB 経由で VIP 宛のパケットを受け取る必要があるので、VIP に関する設定が必要ではありますが、Client からの ARP リクエストに応答してはならないため、通常の用にインタフェースに IP アドレスを設定することは出来ません（図では括弧つきアドレスで表記）。

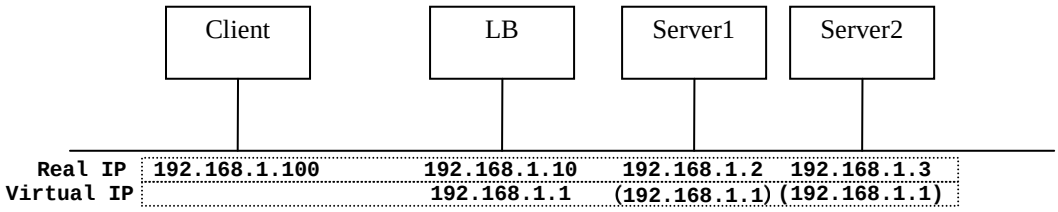

図 31. LVS/DPS のネットワーク構成

LB で振り分け処理が行われた後に Server へパケットが送られますが、この時パケットの宛先アドレスが VIP のままで Server1 か Server2 に送られます。したがって、Server1 と Server2 は VIP 宛のパケットについても自分アドレス宛と同様に受信する必要があります。まとめると、以下のようになります。

(1) LB は VIP 宛のパケットを受信しなければならない
(2) 実サーバ（Server1 と Server2）は VIP 宛のパケットを受信しなければならないが ARP に応答してはならない

(1)を実現するために、LB において以下のように VIP アドレスをインタフェースに追加します。

```
LB# ip addr add 192.168.1.1 dev eth0
```

LB にて、iptalbes およびポリシールーティングを設定します。

```
LB# iptables –t mangle –A PREROUTING –d 192.168.1.1 –j MARK –set mark 1
LB# ip route add local 0/0 dev lo table 100
LB# ip rule add prio 100 fwmark 1 table 100
```

この設定により、LB にて VIP 宛のパケットを処理できるようになります。

(3) を実現するために、実サーバ（Server1 と Server2）において、以下のような設定を行います。

```
Server1# iptables –t nat –A PREROUTING –d 192.168.1.1 –j REDIRECT
```

この設定により、実サーバにおいて、VIP 宛のパケットを処理するようになります。

5.6. ssoadm コマンドをサイト構成で利用する際に発生する問題

ssoadm コマンドを OpenAM のサイト構成で利用する場合、以下のように正常に動作しないという問題があります。[3]

```
# ./ssoadm …
…
Logging configuration class "com.sun.identity.log.s1is.LogConfigReader" failed
com.sun.identity.security.AMSecurityPropertiesException: AdminTokenAction:
FATAL ERROR: Cannot obtain Application SSO token.
Check AMConfig.properties for the following properties
        com.sun.identity.agents.app.username
        com.iplanet.am.service.password
java.lang.NullPointerException
        at java.util.StringTokenizer.<init>(StringTokenizer.java:182)
        at java.util.StringTokenizer.<init>(StringTokenizer.java:204)
        at com.sun.identity.log.LogManager.readAllFields(LogManager.java:162)
        at com.sun.identity.log.LogManager.readConfiguration(LogManager.java:255)
        at com.sun.identity.log.Logger.<clinit>(Logger.java:83)
        at com.sun.identity.cli.CommandManager.destroySSOTokens(CommandManager.java:789)
        at com.sun.identity.cli.CommandManager.<init>(CommandManager.java:208)
        at com.sun.identity.cli.CommandManager.main(CommandManager.java:142)
```

この場合、ssoadm コマンド(スクリプト)を修正し、以下の設定を追加することで正常に動作するようになります。

```
-D"com.iplanet.am.naming.map.site.to.server=
https://lb.example.com:443/openam=https://server1.example.com:443/openam,https:
//lb.example.com:443/openam=https://server2.example.com:4438080/openam
```

ここでは下のような構成を仮定しています。

- https://lb.example.com:443 サイトの URL
- https://server1.example.com:443/openam　サイトの属する OpenAM サーバインスタンス 1
- https://server2.example.com:443/openam　サイトの属する OpenAM サーバインスタンス 2

[3] https://wikis.forgerock.org/confluence/display/openam/Using+the+ssoadm+command+with+a+Site+configuration

5.7. 関連情報

・OpenAM 開発元である ForgeRock 社のサイト
- http://www.forgerock.com/

・OpenAM コンソーシアム
- http://www.openam.jp/