イベント駆動型XMLパーサーを用いた WebSphere Application ServerのWS-Securityのパフォーマンス改善

野ヶ山 尊秀　高瀬 俊郎　上野 憲一郎

Performance Improvement of Web Services Security in a WebSphere Application Server Using an Event Driven XML Parser

Takahide Nogayama, Toshiro Takase and Kenichiro Ueno

WS-SecurityはWebサービスにセキュリティーを適用する仕様であるが，このWS-Securityを適用した場合に性能が著しく劣化する課題がある．これは，暗号化，復号化，電子署名に関する処理のほかに，XML文書に関するさまざまな処理が増えるためである．特にXML文書を格納するための木構造データを作成する処理の負荷が大きい．本論文ではWS-Securityの処理をイベント駆動型XMLパーサーのイベント上で行い，木構造データの作成を避けることで性能改善を図った．この提案手法をWASのWS-Security 実装に応用し，性能改善を確認した．

WS-Security is known as a specification to apply security for web services. But if we apply WS-Security to web services, then the performance of the web services is significantly decreased not only because of encryption decode and digital signatures but also extra costs for XML related processing. In particular, the computational cost for creating tree structured data for XML documents is very expensive. This paper proposes WS-Security processors which work on XML parser events. They enable us to avoid creating a tree structured data for XML documents. We applied this approach to WS-Security in WAS and confirmed that our approach enhanced WS-Security performance.

Key Words & Phrases：XML，Webサービス，WS-Security, 高速化，イベント駆動型XMLパーサー
XML, Web Services, WS-Security, Performance Optimization, Event Driven XML Parser

1．はじめに

Webサービスは，コンピューター間に相互運用性を持つ通信を提供するソフトウェア・システムのことを指す．さまざまな異なるプラットフォーム間の通信に適しているため，普及が進んでいる．標準的なWebサービスの仕様としてJava API for XML-based Web Services（JAX-WS）2.0［1］が一般に普及している．JAX-WS 2.0は，SOAPプロトコル［2］によりExtensible Markup Language（XML）［3］文書を通信するための仕様である．

Web Services Security（WS-Security）［4］はWebサービスにセキュリティーを適用するプロトコルである．XMLデジタル署名［5］によるメッセージの署名機能，XML暗号化［6］によるメッセージの暗号化機能，メッセージに鍵情報や認可情報を運ぶセキュリティー・トークンを付加する機能が規定されている．

一方，セキュアな通信プロトコルとして，ネットワーク層での保護を行うSecurity Architecture for Internet Protocol（IPSec）［7］，トランスポート層での保護を行うSecure Socket Layer（SSL）［8］や，Transport Layer Security（TLS）［9］や，Secure Shell（SSH）［10］などのプロトコルが知られている．これらのプロトコルはネットワーク層やトランスポート層の通信路のセキュリティーを保証しているが，アプリケーション層の通信路のセキュリティーは保証していない．

例えば，あるユーザーがある企業のデータセンター内のサーバーにSSL/TLSを用いて通信する場合を考える．多くの場合，SSL/TLS接続はユーザーのコンピュー

提出日：2008年5月12日　再提出日：2009年6月1日

ターとゲートウェイ間の通信路を暗号化するが，ゲートウェイで一度復号化され，平文による通信としてサーバーに接続される．もしデータセンター内に攻撃者がいた場合，この通信を容易に盗み見ることが可能となる．

このように，データセンター内のネットワークに攻撃者がいる可能性を認めた場合，アプリケーション層でのセキュリティーが必要となる．また複数の経路をたどるような複雑な通信においては，データの選択的な保護が必要となる．WS-Security はこのようなセキュリティー要件を可能とする．

IT ビジネスの普及とともに，このような複雑なセキュリティーを要求する環境が急増している．戦略的アウトソーシングやビジネス・プロセス・アウトソーシングなどの外部委託や，クラウド・コンピューティングなどの普及により，これまで一カ所で管理していたデータが複数の異なる場所に分散し始めている．すなわち，自社のデータを他社のデータセンターやクラウド内に配置することや，自社のデータセンターやクラウド内に他社のアプリケーションが配置されることが一般的に行われるようになった．

このような高い要望に反して，WS-Security は SSL/TLS に比べ普及していない．この理由は主に 2 つ考えられる．1 つは設定の複雑さである．WS-Security はさまざまな設定が可能な反面，設定項目が複雑になる．もう 1 つは性能劣化である．この論文の狙いは，WS-Security の普及のために性能劣化を解消することである．

一般に通信路にセキュリティーを適用すると性能が劣化する．すなわちスループットの低下や応答時間の増加を招く．WS-Security を Web サービスに適用すると性能が著しく劣化する．SSL/TLS を適用した場合に比べて顕著に劣化するため，WS-Security の普及を妨げる一つの大きな要因となっている．

SSL/TLS と比べ WS-Security の性能が劣化する主な要因は，XML に関連する処理が増えているためである．SSL/TLS の場合は，データ部分のバイト列に対して暗号化，復号化，ハッシュ計算などを行えばよい．しかし WS-Security の場合はデータの内容が XML 文書のため，暗号化されたデータを取り出すために木構造データを作成し探索する処理が必要になる．一般に木構造データはメモリー使用量が大きいため，そのメモリー開放が劣化要因となることもある．

木構造データの作成を高速に行う方法として，牧野らは，差分 XML パージングを用いて木構造データの作成を減らす手法［11］により性能改善を図っている．寺口らは，牧野らの手法を改良し差分 XML パージングのメモリー使用量を削減する手法［12］により，さらなる性能改善を図っている．これらの手法は最終的には木構造データを作成するため，メモリー使用量の問題は残る．

木構造データを作成しない方法として，牧野らはイベント駆動型 API の一つである Simple API for XML（SAX）［16］を用いてストリーム型の WS-Security を実装し，木構造データを作成せずに WS-Security の処理を行う手法［13］により性能改善を図っている．この手法は，Web サービス・アプリケーションも SAX を用いた実装に変える必要がある．しかし Web サービス仕様である JAX-WS 2.0 では，Web サービス・アプリケーションは Java Architecture for XML Binding（JAXB）2.0［14］オブジェクトを処理すると規定しているため，SAX イベントを一度シリアライズし，JAXB オブジェクトを作成するコンポーネントに入力することになり，同じ XML に対して 2 度 XML パージングを行うことになってしまう．

本論文ではイベント駆動型 API である Streaming API for XML（StAX）［17］上で稼働するストリーム型の WS-Security を実装し，さらに JAXB オブジェクトの作成までの一連のストリームをつなげる方法を提案する．これにより木構造データを作成せずに WS-Security の処理を実行できる．

本論文ではまず背景として，一般的な XML の知識と XML の処理方法について述べる．次に Web サービスと WS-Security について述べ，その課題点を指摘する．最後に筆者らが提案する手法を述べ，性能評価実験によってその有効性を示した．また，提案手法を WebSphere® Application Server（WAS）の WS-Security 実装に組み込み，エンド・ツー・エンドでの性能評価実験を行い，有効性を確認した．

2. Webサービス・エンジンのXML文書の処理

2.1 2種類のXMLパーサーAPI（木構造データ型APIとイベント駆動型API）

XML を木構造データとして読み込む API として Document Object Model（DOM）［15］が知られている．一方イベント駆動型 API として SAX や StAX が知られている．

木構造データ型 API の利点は，もともと木構造である XML 文書を直感的に扱えることである．また部分木の削除や挿入を容易に行うことができる．しかし XML 文書全体をメモリー上に読み込んでしまうため，動作が遅い．

イベント駆動型 API の場合，XML 文書を一連のイベントとして処理し，処理し終わったイベントは破棄される．

XML 文書全体をメモリーに読み込むことがないため，木構造データ型 API よりも高速である．またイベントを次々とイベント処理プログラムに渡すことで，簡単に並行処理プログラムを記述することができる．しかしイベントが過ぎ去ってしまった個所を再び読みたい場合，再び XML 文書の先頭から読み込まなければならない．

イベント駆動型 API にはプッシュ型とプル型の 2 つのタイプがある．プッシュ型は XML パーサーがイベント処理プログラムに一方的にイベントを送る．プル型はイベント処理プログラムが XML パーサーにイベントのリクエストを行い，イベントを受け取る．このような特徴のため，プッシュ型 API は XML 文書全体をイベントに変換してしまうが，プル型は任意の場所でイベントの作成を止めておくことが可能である．

木構造データ型 API とイベント駆動型 API の両方の特徴を備えた API として，JDOM［18］，dom4j［19］，XOM［20］，Axis Object Model（AXIOM）［21］などが知られている．イベント駆動型 XML パーサーを木構造データ作成プログラムで包むことで実現している．

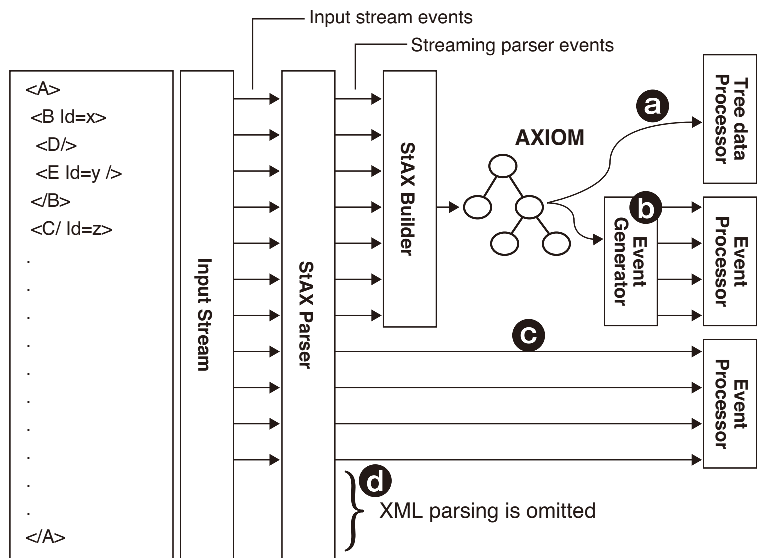


図1. AXIOMによるさまざまなXML文書の利用形態

2.2 2種類のXMLパーサーAPIを活用できるデータ構造 AXIOM

WebSphere Application Server 6.1 Web Services Feature Pack（WAS 6.1 FP）以降の Web サービス・エンジンは，AXIOM を通してイベント駆動型 API と木構造データ型 API の両方で XML 文書にアクセスすることができる．

AXIOM では，XML パーサーとして StAX パーサーを使用し，木構造データ AXIOM の作成に StAX ビルダーを使用する．これらの利用形態について図 1 に示した．

AXIOM は，遅延評価を用いて，XML パージングと木構造データの作成を遅らせ，必要になったときに必要な処理のみを行う．まず AXIOM のルート要素のみを作成し，AXIOM を扱うプログラムが子要素を取得しようとしたときに初めて StAX パーサーにイベントの要求を行い，子要素に対応する AXIOM を作成する（図 1 の a）．このため，プログラムから要求されなかった個所の構文解析と木構造データの作成を省くことができる（図 1 の d）．

高速化やメモリー節約のため，AXIOM の作成を省き，イベント駆動型プログラムを用いて直接 StAX パーサーからの StAX イベントを処理すること（図 1 の c）ができる．ただしこの場合は StAX イベントが破棄されてしまうため，2 回以上アクセスする必要がある場合は利用できない．

また，AXIOM から StAX イベントを作成し，イベント駆動型プログラムで処理することが可能である（図 1 の b）．この StAX イベントは，StAX パーサーが生成する StAX イベントと同じである．この機能により，AXIOM が作成されてしまった要素に対して，イベント駆動型プログラムで処理することが可能となる．

3. WebサービスとWS-Security

3.1 イベント駆動型APIを利用したWeb サービスの処理

JAX-WS 2.0 は Java 上で実現する Web サービスの

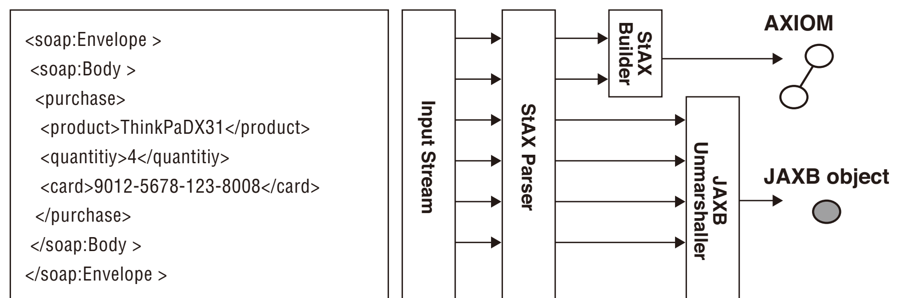


図2. イベント駆動型APIを利用したWebサービスの受信側での処理

仕様について定めている．JAX-WS 2.0では，JAXBによるXMLデータ・バインディングを用いて，XML文書をJAXBオブジェクトに変換してからWebサービス・アプリケーションに渡すように規定している．

AXIOMによるWebサービスの受信側でのXMLの処理の概要を図2に示した．SOAP EnvelopeとSOAP BodyのAXIOMだけが作成され，SOAP Bodyの子要素のStAXイベントはJAXB Unmarshallerによって処理されたのち破棄され，JAXBオブジェクトが作成される．

3.2 木構造データ型APIを利用したWS-Securityの処理

Webサービス送信側で典型的なWS-Security 設定（SOAP BodyにXML電子署名を施したのち，SOAP Bodyの子要素がXML暗号化により暗号化される）が適用されたメッセージを処理するWebサービスの受信側の概要を図3に示した．

Webサービスの受信側では，まずXML暗号により暗号化されたSOAP Bodyの子要素を復号する．SOAP Bodyの子要素は<xenc:EncryptedData>であり，この要素は暗号化されたデータを保持している．まずDecryption Handlerが<xenc:EncryptedData>に格納されたSOAP Bodyから暗号化されたデータを取り出し，復号を行う．復号化された値はXMLの断片のバイト列となっている，この断片はStAXパーサーとStAXビルダーによってAXIOMに変換され，<xenc:EncryptedData>と入れ替えられる．

次にSOAP Bodyに対して付与されているXML電子署名の検証を行う．まずSOAP Body全体のAXIOMを走査してExclusive XML Canonicalization（XML C14N）［22］によるXML正規化を行う．正規化されたSOAP Body要素からdigest値を計算し，送信側が添付したdigest値との比較を行い署名の検証が終わる．

最後に，JAXBオブジェクトを作成し，Webサービス・アプリケーションに渡す．復号されたSOAP Bodyの子要素のAXIOMからStAXイベントを生成し，JAXB UnmarshallerがそのイベントをR処理してJAXBオブジェクトが作成される．

3.3 木構造データ型APIを利用したWS-Securityの課題

一般にセキュリティーを適用した場合，必ず暗号化，復号化，電子署名の添付，電子署名の検証の処理は増える．しかしWS-Securityではそれ以外にXMLに関連した処理が増える．

WS-Securityを適用していないWebサービスの実装の場合，平文のStAXイベントがそのままJAXB Unmarshallerに入力されている（図2）．しかしWS-Securityを適用した場合は平文のStAXイベントはAXIOMに変換され，StAXイベントを再び作成してJAXB Unmarshallerに入力している．このため，StAXイベントからAXIOMを作成する処理とAXIOMからStAXイベントを作成する処理が余分に必要となる．また，暗号化されたデータを含む要素<xenc:EncrtyptedData>のAXIOMを作成する処理も余分に必要となる．

また，木構造データ上で行うXML C14Nでは，名前空間のスコープの確認を何度も行うことになるが，その度に先祖の要素をたどってXML名前宣言［23］を探す

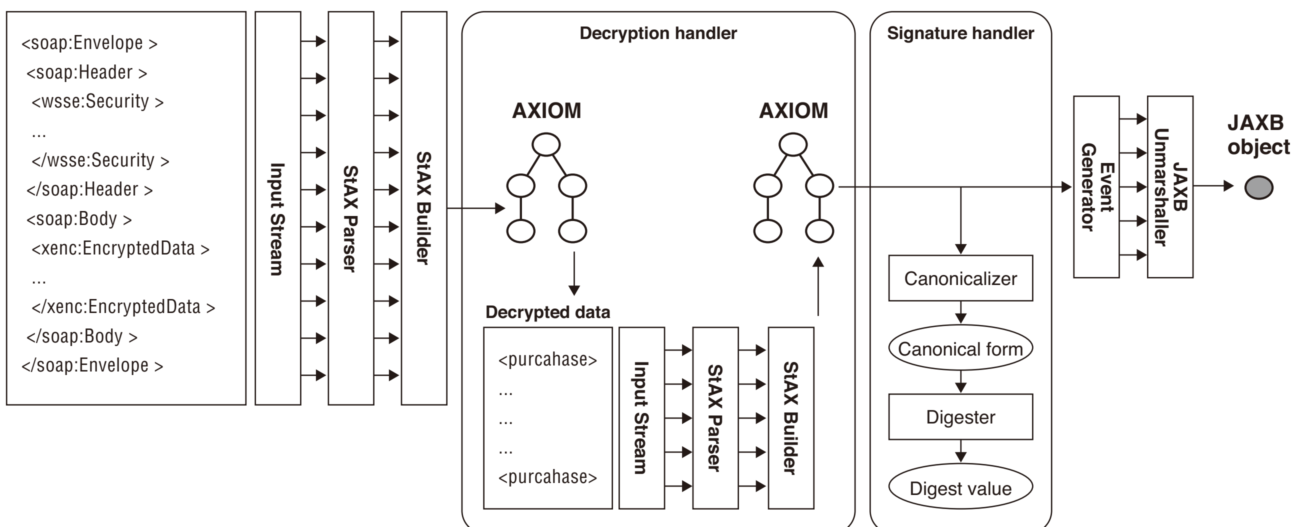


図3. 典型的なWS-Securityのシナリオを処理する木構造データを利用したプログラムの処理フロー

処理を行う．この処理はQNameの個数と同じ回数行われる．XML文書ではQNameは多く登場するため，木構造データ上でのXML C14Nの処理量は大きい．

AXIOMの生成，AXIOMからStAXイベントの生成，AXIOM上でのWS-Security処理の3つを避けることで高速化が可能であると予想し，次節で述べる手法を提案する．

4. イベント駆動型APIを利用した提案手法によるWS-Securityの性能改善

本論文では，StAXイベント上でWS-Security処理を行うフィルターを提案する．この手法をStAXフィルターと呼ぶ．本論文では，復号を行うDecryption Filter，正規化を行うC14N filterを提案する．図4に提案手法を利用したWS-Security処理の全体像を示した．

まずSOAP HeaderとSecurity Headerは木構造データを作成するため，StAXイベントはそのままStAXビルダーに渡す．なぜなら，Security headerには挿入，削除，探索などの複雑な処理が行われるため，ストリーミングで処理することが困難なためである．

次に<xenc:EncryptedData>のStAXイベントはDecryption filterに吸収され，処理された後に破棄される．Decryption filterは暗号データを復号し，復号データのStAXイベントを作成し，次のフィルターに渡す．

C14N filterはSOAP Body要素のスタート・タグのStAXイベント，復号データのStAXイベント，SOAP Body要素のエンド・タグのStAXイベントの順にイベントを受け取り，正規化を行う．C14N filterは届いたイベントに対して何も変更せずに次のJAXB Unmarshallerに渡す．この時，名前空間の情報はStAXパーサーがXMLパージングのために保持しており，この情報を使えば名前空間のスコープの確認は，木をたどる手法に比べ，高速に完了する．

JAXB UnmarshallerではC14N filterから受け取った復号データのStAXイベントからJAXBオブジェクトを作成し，イベントを破棄する．

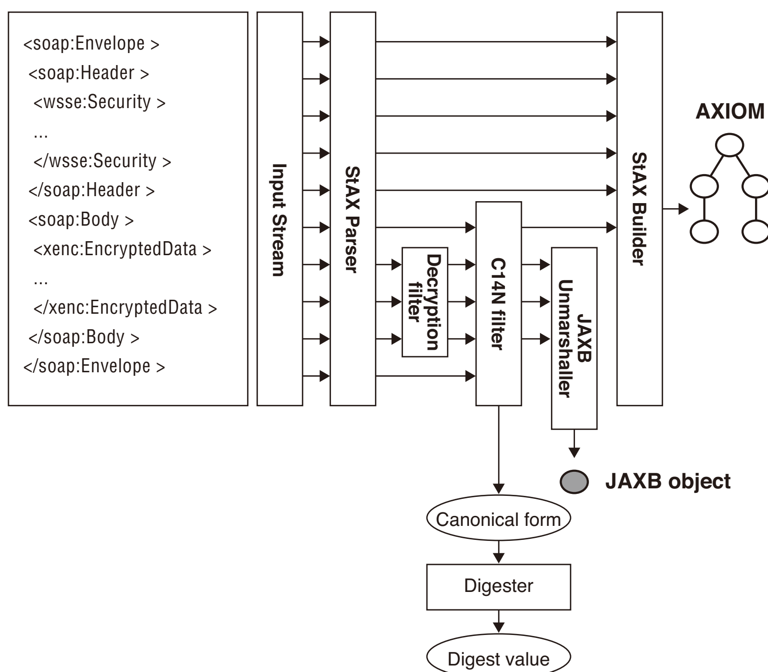


図4. StAXフィルターによるWS-Security処理

木構造データ型APIを利用したWS-Securityと，イベント駆動型APIを利用した提案手法を比較すると，(a) <xenc:EncryptedData>の木構造データの作成の省略，(b) 復号データの木構造データ作成の省略，(c) 復号データの木構造データからStAXイベントの作成の省略，(d) 木構造データ上でのWS-Security処理からイベント駆動型のWS-Security処理に変更，の4つの観点で高速化が図られている．

5. 提案手法の性能評価

5.1 性能評価実験

木構造データ型APIを利用したWS-Security処理に比べ，提案手法がどの程度性能が向上したのかを実験により評価した．

木構造データを利用するWS-Security処理は図3と同じ処理を行うプログラムを作成した．提案手法のStAXフィルターでのWS-Security処理は，図4と同じ処理を行うプログラムを作成した．両方のプログラムへの入力はメッセージのバイト列，出力はdigest valueと作成されたJAXBオブジェクトである．

SOAP Body内の平文のXML文書は，保険会社の車両保険を扱うWebサービスで通信される内容を模したXML文書を3kバイト，10kバイト，100kバイトの3種類用意した．

暗号化アルゴリズムはAES-CBC 128bit，正規化アルゴリズムはExclusive C14N without comment，ダイジェスト・アルゴリズムはRSA-SHA1を利用した．

この実験ではSOAP Bodyに対するWS-Security処理の評価が目的であるため，

表1. スタンド・アロン・プログラムによる性能比較

Message size（byte）	AXIOM-based WS-Security［a］	StAX filter-based WS-Security［b］	Rate［b/a］
3k（byte）	12,969（ms）	6.750（ms）	52%
10k（byte）	26,015（ms）	11,047（ms）	46%
100k（byte）	220,985（ms）	99,250（ms）	45%

SOAP Header内のSecurity headerは削除し，この部分のAXIOM作成が計測に含まれないようにした.

実験はIBM xSeries x365（Xeon MP 3.0GHz x 4 CPUs, 8GB RAM, 4MB L3）Windows 2003 Enterprise Edition（32bit），IBM J9 VM（build 2.3, J2RE 1.5.0 IBM J9 2.3 Windows XP x86-32 j9vmwi3223-20071007（JIT enabled））上で行った. Javaのoptionには“-Xms1024m -Xmx1536m”を使用した.

実験結果を**表1**に示した. 2列目と3列目の値は10,000回の繰り返しに要した時間である. 4列目は2つの処理の比であり，値が少ない方が提案手法による性能改善の効果が大きいことになる. 提案手法がWS-Security処理を約50% 高速化している.

また，WASのWS-Security実装を提案手法で置き換え，別のクライアント・マシンからWebサービスを呼び出し，スループットを計測することで，エンド・ツー・エンドの性能比較を行った. 提案手法を採用したWASではリクエストとレスポンスがともに3kバイトのWebサービス・アプリケーションにおいて約10%，10kバイトのケースにおいて約20%，100kバイトのケースにおいて約30% のスループット増加を確認した.

5.2 性能評価実験の考察

実験の結果，SOAP Bodyの内容のメッセージ・サイズが大きくなればなるほど提案手法の性能改善の効果が大きくなった. これには，処理比率の変化と，Garbage Collection（GC）の2つの要因が考えられる.

WS-Securityの処理には，メッセージと関係ない処理と，メッセージ・サイズの増加によって処理量も増す処理がある. 例えば，暗号化エンジンの初期化やダイジェスト値の比較などの処理量は，メッセージ・サイズに依存せず一定である. 一方，木構造データの作成，暗号化，そしてダイジェスト値の計算の処理量は，メッセージ・サイズに依存して増加する.

提案手法の効果はメッセージに関する処理を削減している. このためメッセージ・サイズが大きくなれば全処理量に占めるメッセージの処理量の比率が大きくなるため，提案手法の性能改善の効果も大きくなる.

メモリー確保と同様にメモリーの開放も処理量が大きい. JavaではGCが起動すればすべての処理が一時停止するため，GCの頻度が高いプログラムは遅いプログラムである. GCは空きメモリー量が減少した場合に起動する. このため，単位時間当たりに新たなオブジェクトを作成し開放した量がGCの頻度に比例する.

提案手法は木構造データを作成しないため，作成と開放するオブジェクトの量が少ない. このため，木構造データ型APIを利用したWS-Security処理に比べ，GCの発生回数を抑えることができた.

6. おわりに

本論文では，WS-Securityの性能を向上させるための手法としてイベント駆動型プログラムによるWS-Securityの処理を行うStAXフィルターを提案した. スタンド・アロンの性能評価実験により，WS-Security実装の範囲においておよそ50% の性能改善を確認した. WASを用いたエンド・ツー・エンドの性能評価実験では，10% から30% の性能改善を確認した.

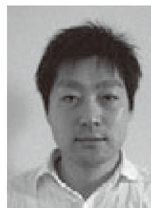
しかし，提案手法はすべてのXML処理には適用できない. XMLパーサーのイベント・フィルターによる処理を行う場合，過ぎ去ってしまったイベントに対する読み込みや書き込みなどの処理はできない. この場合はもう一度読み直すか，木構造データ型APIを利用した方法を採用すべきである.

同様のストリーミング処理はメッセージを送り出す処理にも適用可能である. Webサービス・アプリケーションはJAXBオブジェクトをXML文書に変換して送出する. WS-Securityを適用すれば，このJAXBオブジェクトを一度AXIOMに変換しなければならない. 木構造データ作成の問題を解決するWS-Security処理の送信プログラムの高速化が今後の課題である.

本論文ではSOAP Headerの木構造データ作成の処理性能の改善は考慮しなかったが，一般に典型的なWS-Securityの設定を有効にすればSOAP Headerのサイズは約7kバイト以上になる. XMLの構造も複雑となるため，SOAP Headerに対する性能改善は今後の重要な課題である.

参考文献

［1］ Java Community Process: “JSR 224 Java API for XML-Based Web Services（JAX-WS）2.0,” http://jcp.org/en/jsr/detail?id=224（2006）.
［2］ W3C Recommendation: “SOAP version 1.2,” http://www.w3.org/TR/soap12-part0/（2007）.
［3］ W3C Recommendation: “Extensible Markup Language（XML）1.0,” http://www.w3.org/TR/2006/REC-xml-20060816/（2006）.
［4］ OASIS standard: “Web Services Security: SOAP Message Security,” http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf（2004）.
［5］ W3C Recommendation: “XML-Signature Syntax and Processing,” http://www.w3.org/xmldsig-core/（2001）.
［6］ W3C Recommendation: “XML Encryption Syntax and Processing,” http://www.w3.org/TR/xmlenc-core/（2001）.
［7］ Internet Engineering Task Force: “RFC 2401 Security Architecture for the Internet Protocol（IPSec）,” http://www.ietf.org/rfc/rfc2401.txt（1998）.
［8］ A. Frier, P. Karlton, and P. Kocher: “The SSL 3.0 Protocol,” http://www.mozilla.org/projects/security/pki/nss/ssl/draft302.txt（1996）.
［9］ Internet Engineering Task Force: “RFC 2246 The Transport Layer Security（TLS）Protocol Version 1.0,” http://www.ietf.org/rfc/rfc2246.txt（1999）.
［10］ Internet Engineering Task Force: “RFC4250-4256 Secure Shell（SSH）Protocol,” http://tools.ietf.org/html/rfc4250（2006）.
［11］ Satoshi Makino, Michiaki Tatsubori, Kent Tamura and Yuichi Nakamura: “Improving WS-Security Performance with a Template-Based Approach,” Proceedings of 2005 IEEE International Conference on Web Services（ICWS 2005）, pp.581-588（2005）.
［12］ Masayoshi Teraguchi, Satoshi Makino, Ken Ueno and Hyen-Vui Chung: “Optimized Web Services Security Performance with Differential Parsing,” Proceedings of International Conference on Service-Oriented Computing 2006（ICSOC 2006）, pp. 277-288（2006）.
［13］ Satoshi Makino, Kent Tamura, Takeshi Imamura and Yuichi Nakamura: “Implementation of WS-Security and Its Performance Improvements,” Proceedings of the International Conference on Web Services（ICWS 2003）, pp. 256-264（2004）.
［14］ Java Community Process: “JSR 222 Java Architecture for XML Binding（JAXB）2.0,” http://www.jcp.org/en/jsr/detail?id=222（2006）.
［15］ W3C Recommendation: “Document Object Model（DOM）Level 3 Specification,” http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/（2004）.
［16］ David Brownell: SAX2, O'Reilly, ISBN 0-596-00237-8（2002）.
［17］ Java Community Process: “JSR 173 Streaming API for XML（StAX）Specification,” http://jcp.org/en/jsr/detail?id=173（2004）.
［18］ Jason Hunter and Brett McLaughlin: “JDOM,” http://www.jdom.org/（2000）.
［19］ “dom4j,” http://www.dom4j.org/.
［20］ “XOM,” http://www.cafeconleche.org/XOM/.
［21］ Apache Software Foundation: “Axis Object Model,” http://ws.apache.org/commons/axiom/.
［22］ W3C Recommendation: “Exclusive XML Canonicalization Version 1.0 ,” http://www.w3.org/TR/xml-exc-c14n/（2002）.
［23］ W3C Recommendation: “Namespaces in XML 1.0,” http://www.w3.org/TR/REC-xml-names/（2006）.



日本アイ・ビー・エム株式会社
東京基礎研究所
副主任研究員

野ヶ山 尊秀 Takahide Nogayama

［プロフィール］
2004 年，日本 IBM 入社．東京基礎研究所にてオートノミック・コンピューティングの研究・製品開発に従事．2006 年より Web サービス・エンジン，WS-Security の高速化に従事．

日本アイ・ビー・エム株式会社
東京基礎研究所
主任研究員

高瀬 俊郎 Toshiro Takase

［プロフィール］
2000 年，日本 IBM 入社．東京基礎研究所にて XML，Web サービス処理の高速化の研究に従事．

日本アイ・ビー・エム株式会社
大和ソフトウェア研究所
ITスペシャリスト

上野 憲一郎 Kenichiro Ueno

［プロフィール］
日本 IBM 大和ソフトウェア開発研究所にて，さまざまなお客様における WebSphere 関連プロジェクトに参画．WebSphere パフォーマンス専門家として，セミナーなどで講演も実施．