Secure Password-Based Authenticated Key Exchange for Web Services

Liang Fang[1,2], Samuel Meder[4], Olivier Chevassut[3], and Frank Siebenlist[1]

Abstract. This paper discusses an implementation of an authenticated key-exchange method (AuthA) rendered on message primitives defined in the WS-Trust and WS-SecureConversation specifications. This IEEE-specified cryptographic method is proven-secure for password-based authentication and key exchange, while the WS-Trust and WS-SecureConversation are emerging Web Services Security specifications that extend the standardized WS-Security specification. A prototype of the presented protocol is integrated in the WS-ResourceFramework-compliant Globus Toolkit V4. Further hardening of the implementation is expected to result in a version that will be shipped with future Globus Toolkit releases. This could help address the current unavailability of decent shared-secret-based authentication options in the Web Services and Grid world. Future work will also be dedicated to integrate One-Time-Password (OTP) features in the authentication protocol.

[1] Mathematics and Computer Science Division, Argonne National Laboratory, *franks@mcs.anl.gov*.
[2] Computer Science Department, Indiana University, *lifang@cs.indiana.edu*.
[3] Computational Research Division, Lawrence Berkeley National Laboratory, *ochevassut@lbl.gov*.
[4] Department of Computer Science, University of Chicago, *meder@mcs.anl.gov*.

1. **Introduction**

1.1 Grid Computing and Web Services

The term “Grid” refers to systems and applications that integrate and manage resources and services distributed across multiple control domains [GRID]. Pioneered in an e-science context, Grid technologies are also generating interest in industry, as a result of their apparent relevance to commercial distributed-computing applications [PHYS]. The results of this research have been incorporated into a widely used software system called the Globus Toolkit ® (GT) [GT, GTO] that uses public key technologies to address issues of single sign-on, delegation, and identity. The Grid Security Infrastructure (GSI) is the name given to the portion of the Globus Toolkit that implements security functionality.

The recent definition of the Web Service Resource Framework (WSRF) specification and other elements of the Open Grid Services Architecture (OGSA) within OASIS and the Global Grid Forum (GGF) introduce new challenges and opportunities for Grid security [WSRF, OGSA, GGF]. In particular, integration with Web services and hosting environment technologies introduces opportunities to leverage emerging security technologies such as described in the WS-Security, WS-Trust and WS-SecureConversation specifications [WSSec, WSTr, WSSC]

1.2 Web Services Security

Web Services Security is still immature in many ways, which is evident by the number of emerging specifications that are competing and in flux. Recently, however, the basic underpinnings for SOAP message security have been defined by the standardized WS-Security specifications in OASIS [WSSec].

WS-Trust and WS-SecureConversation are proposed extensions of the WS-Security specification, defining message primitives and interfaces for security context establishment, sharing, and session key derivation [WSTr, WSSC]. Although these specifications have not yet been standardized, the associated authors have publicly stated their intension to do so.

1.3 Security in Grid Computing

Security is one of the major requirements of Grid computing. Any Grid site deployment must at least provide the basic security mechanisms including authentication, authorization and secure communications. The Grid Security Infrastructure (GSI) component in the Globus Toolkit plays the central role in providing these mechanisms, as well as single sign-on, delegation and mutual authentication using public key cryptography. GSI was initially built upon the Transport Layer Security (TLS) protocol, and was enhanced to provide message level authentication, key exchange, data protection, and delegation through proxy certificates. With the migration towards Web Services, GSI has used WS-security primitives, to support the establishment of a

security context between two parties. GSI-SecureConversation was the first effort of such a protocol, which defines its own session-based security mechanism, similarly to WS-SecureConversation. Its implementation is based on public key certificates.

Recent events, related to the compromise of desktops and servers, have led to a trend where many Grid site deployments are moving to the use of password-based authentication to obtain public key credentials from a credential service. Those recent compromises of user and server machines are resulting in site security policy changes where long-term secrets are no longer to be stored on the user's machines. Instead, long-term credentials will be stored on servers in data centers where their integrity can be better protected. Users will authenticate with a (one-time) password to these credential servers. After successful authentication, the user will obtain short-lived credentials, such as a short-lived X.509 certificate or proxy-certificate in the case of MyProxy, which can subsequently be used for the access of other services on the Grid [MYPR]. This has renewed interest in password-based authentication mechanisms.

1.4 Accomplishments

This paper describes the design and implementation based on the WS-Trust and WS-SecureConversation specifications supporting an authentication method based on a password, i.e. shared secret. A password is a short string chosen from a relatively small dictionary so that it is easier to be memorized than a long symmetric key; however, passwords are subject to various attacks such as dictionary attack and network eavesdropping. Therefore, ideally, passwords should not be used directly as input of signature/encryption schemes. A run-time password-derived secret should be used instead.

Our implementation consists of a password-based authenticated Diffie-Hellman key exchange to agree on a session key, and a key derivation to educe multiple session keys from this master key. Each session is in turn used in conjunction with a symmetric cipher such as the AES, and Message Authentication Code such as HMAC, to implement secure message exchanges. The communications for passing these inner cryptographic primitives and parameters need to be defined for WSRF-compliant clients and services. This requires the definition of operations for all client/service interactions in Web Service Definition Language (WSDL). WSDL is the standard language in XML for defining Web Services interfaces, also called Port Types.

The rest of this paper is organized as follows. In Section 2, we illustrate the security context establishment based on the WS-Trust and WS-SecureConversation specifications. In Section 3, we introduce the password-based key exchange method and explain its integration in WS-Trust and WS-SecureConversation under the Web Services Resource Framework (WSRF). Section 4 concludes this paper and presents our future directions.

2. Security Context Establishment using WS-Trust and WS-SecureConversation

2.1 WS-Trust

The Web Services Trust Language is an extension to WS-Security. It defines syntax for security token exchanges to build up trust relationship among different web service domains. It also provides a set of mechanisms to allow a range of security protocols to fit in, such as the Web Services Trust Model. In this Web Services Trust model, if the service requestor does not have the required tokens for a target service, it turns to an authority for them. Such an authority is called *Security Token Service*. WS-Trust also defines multi-message exchange mechanisms, such as the challenge-response protocol. Most of the tokens used in WS-SecureConversation are actually defined in WS-Trust, including the most frequently used *RequestSecurityToken* (RST) and *RequestSecurityTokenResponse* (RSTR).

2.2 WS-SecureConversation

The Web Services Secure Conversation Language is based on WS-Security and WS-Trust to allow security context establishment, sharing, and session key derivation. It defines a *Security Context Token* (SCT) shared among the communicating parties for the session lifetime. A SCT usually contains an identifier pointing to the security tokens being shared.

WS-SecureConversation gives three scenarios for establishing security contexts. The first one is SCT created by a security token service; the second one is by one of the communicating parties and propagated with a message; the last is through negotiation and exchanges. These scenarios, however, do not exclude each other. For instance, a security token service may have to create security context tokens through negotiation.

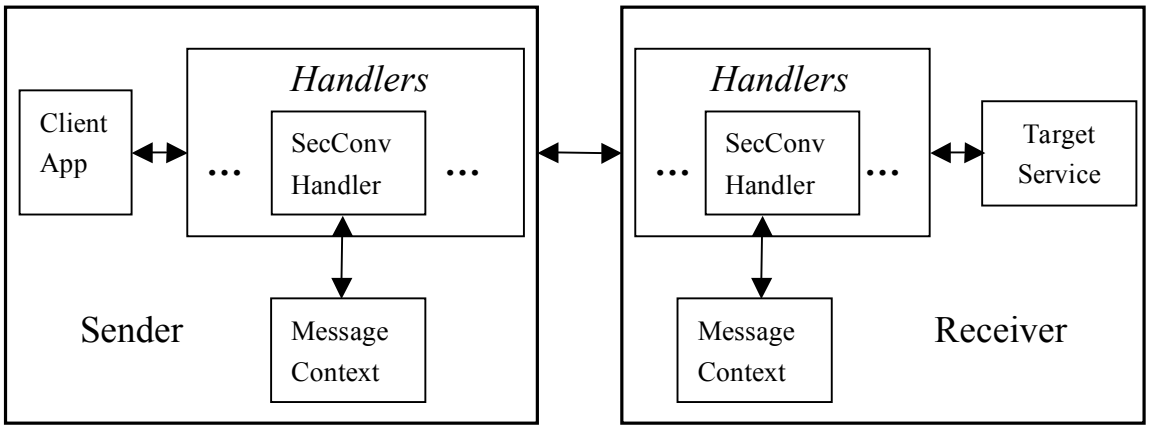

Figure 1 Secure Conversation in Handlers

Although there is no open source implementation available yet of the WS-Trust&SecureConversation, in our unreleased version of WSS4J [WSS4J], the authors have implemented a preliminary version of those specs.
The process flow is shown in **Figure 1**, which assumes that both sides have previously exchanged their public keys. The sequence is as follows:

1. The sender, which is usually a client, defines the cryptographic algorithms to use, and generates a random session key suitable for the selected algorithms for either signature or encryption or both.
2. It then encrypts the generated session key with the public key of the receiver (with the assumption that the sender already knows where to get the receiver's public key).
3. It embeds the encrypted session key into a SOAP message. If necessary, he may choose to encrypt the message with the session key.
4. The receiver, upon receiving the SOAP message, decrypts the session key with his own private key. From then on, both sides have possessed the session key for further signature and verification or encryption and decryption operations.

The goal of WS-SecureConversation initialization process is to finish the key exchanges before passing the first remote invocation message to the service provider. Intuitively, this solution looks straightforward and easy to be implemented for the goal; however, the problem is that, with the dependency on two handlers, it literally has no ability to do multi-round key exchange interactions. The reason lies in the service side WS-SecureConversation handler. WSS4J, as a sub-project of Axis, follows the design of Axis. Axis is an open source SOAP engine for building generic Web Service applications [AXIS]. Simply speaking, every SOAP message goes through the Axis SOAP engine to its desired target along a pre-configured path. The path is made up of a chain of processor nodes, also known as handlers. For example, the encryption handler is responsible for encrypting or decrypting SOAP messages when it receives them. The service provider handler is a special handler that contains the real service logic and sits at the end of the path. Except the service provider, a handler can never short-circuit the message flow by returning the message without passing it on to the next handler on the path.

In a secure Web Service, the WS-SecureConversation handler at the service side is not the service provider but one of the underlying serving handlers. It must faithfully pass the remote invocation message on, until it reaches the service provider, even though the key exchange interaction may not have finished and the session key may not be ready. Therefore it is impossible for a server-side WS-SecureConversation handler to finish a multi-round interaction. This solution only works for those cases that no further interaction is necessary such as the one assumed in WSS4J.

WS-Trust and WS-SecureConversation do not define any specific key exchange methods for getting session keys; therefore a good WS-Trust and WS-SecureConversation implementation should allow any methods to be integrated. These methods include Diffie-Hellman key exchange, independent security token services and other services that propagate the tokens and so on. This approach restricts the WS-SecureConversation initialization from being really interactive, as explained.

To solve the problem, our WS-SecureConversation implementation inherits the approach that GSI-SecureConversation adopted by implementing the negotiation mechanism as a separate service. The details will be addressed in the next section.

3. Password-Authenticated Key Exchange in WS-Trust and WS-SecureConversation

3.1 Method for Password-Based and Key-Exchange

In the last decade, a set of cryptographic methods for password-based authentication and key exchange have emerged, such as EKE [EKE], AuthA [AuthA-1], and [STW, JA, WU]. The AuthA method has been proposed to the IEEE P1363 Study Group as a standard for public-key cryptography [SSPKC]. It was first presented by Bellovin and Merritt [AuthA-1], and proved secure recently by Bresson, Chevassut and Pointcheval [AuthA-2, AuthA-3]. On the implementation side, Steiner et al. added a refined password-based Diffie-Hellman Key Exchange method to TLS [SBEW].

Methods for Authenticated Key Exchange (AKE) are for making an agreement on a common secret key between two parties. The secret key is then stored in security context in order to establish a secret channel for a lifetime. Password-based AKE is under the assumption of an existing password, of unknown quality. The goal is to derive a session key from the given password while providing mutual authentication. This problem was raised by Bellovin and Merritt with their first solutions in 1992 [EKE].

Based on the Diffie-Hellman key agreement protocol [DH76], these methods use passwords as authentication information, encryption seed, and a factor for derivation of the final session key. With password protection, it prevents the key exchange process from attacks such as the man-in-the-middle one, to which Diffie-Hellman protocol alone is vulnerable. Just like in Diffie-Hellman protocol, the negotiated session key never has to be sent over the wire in the whole process. The passwords used here are usually hashed from the original ones of unknown quality.

Like other AKE methods, The AuthA is based on the Diffie-Hellman key agreement protocol. It protects the negotiation process from the man-in-the-middle attack by encrypting public values with a previously known password. It also provides mutual authentication and forward secrecy. Forward secrecy means the attacker will not be able to distinguish a session key generated before any compromise of the long term private keys.

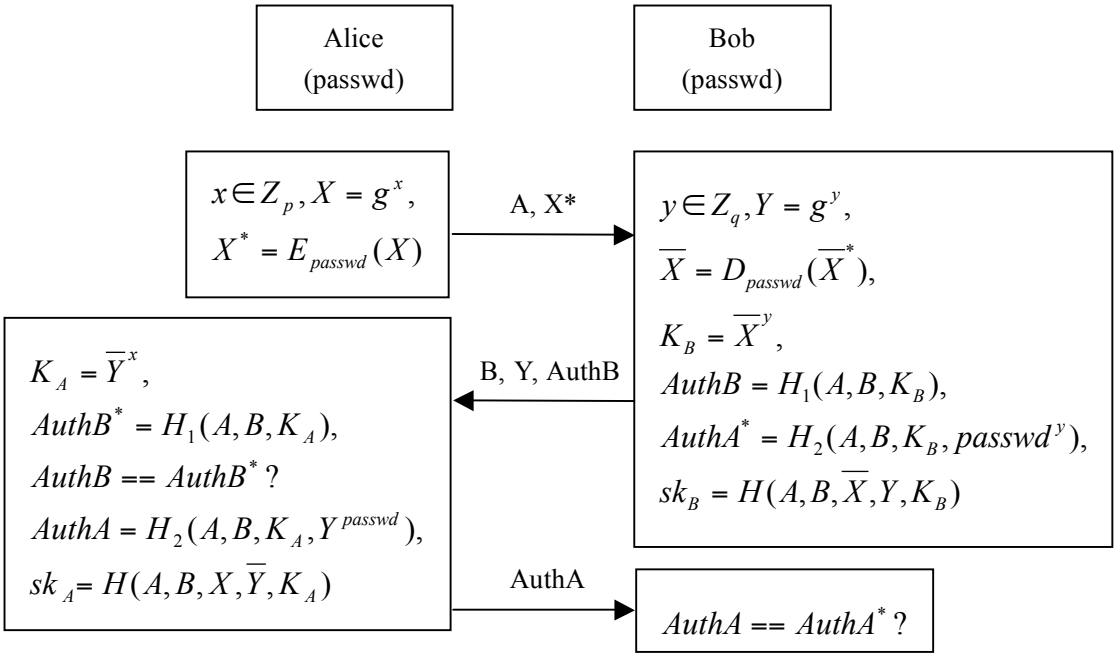

Figure 2 The AuthA Method

A typical workflow of the AuthA method is shown in **Figure 2**. Here Alice and Bob are two communicating parties. They share a previously known password *passwd*. G is a cyclic group on which the Diffie-Hellman problem is hard. (The group used in our implementation is the group $G = Z^*_n$ with n a prime number and $p|n$.) The group is produced by a generator g. By default, G, p, and g, are all well-known.

Then, Alice randomly picks up an x from Z_p, and calculates X from the generator g. Instead of putting the plain X on the wire like in the vanilla Diffie-Hellman protocol, Alice encrypts it with the password *passwd*, and sends the encrypted X^* over to Bob along with her name. Notice that all the exponentiation operation results from module p. On the other side, upon receiving the request, Bob randomly chooses a y from Z_p, and calculates Y with generator g. Meanwhile, he decrypts X with the password *passwd* from X^*. Considering X^* could have been modified or replaced by a third party attacker, we mark the X^* received as $\overline{X}^*$, and the decrypted X as $\overline{X}$. $X^* = \overline{X}^*$ and $X = \overline{X}$, if no change happens. With X obtained, Bob is able to get the Diffie-Hellman key K_B. However, in AuthA, we do not use Diffie-Hellman key directly as the session key, but as a factor to generate the session key, along with some other information including the password and the names of both parties. The session key sk_B is derived from a hash function upon concatenation of all the information above altogether. Besides the session key, we also need to get *AuthB* and *AuthA** for authentication purpose. In the next step, Bob sends Y, *AuthB*, as well as his name, to Alice. Different from X, Y is not required to be encrypted, which saves the expensive encryption operation.

After receiving Y, Alice virtually has got all she needs to generate the session key. She then first checks whether the *AuthB* from Bob matches her *AuthB**. If so, she continues to calculate the session key and *AuthA*, correspondingly; otherwise, Bob cannot be authenticated, and the handshake fails. At last, Alice returns Bob the

AuthA for authenticating herself. If *AuthA* matches *AuthA**, the whole process is done.

In **Figure 2**, the session key sk_A is equal to sk_B, as long as $X = \overline{X}$, and $Y = \overline{Y}$, thus both parties get the session key and all its direct derivations without passing it over the wire.

3.2 Design Criteria for WS-SecureConversation

As we explained in section 2.2, because the WSS4J solution does not serve for the multi-round interaction cases such as the AuthA key exchange method, we have to provide our own solution to support the AuthA and any other initialization requirement.

In our WS-SecureConversation implementation design, we adopt the GSI-SecureConversation approach by implementing the server side negotiation mechanism as a separate service in the same container as its serving services. A WS-SecureConversation handler is provided at the client side.

This approach takes the advantages of Web Service Resource Framework (WSRF) by having service contexts including the service security context as stateful resources. The client side WS-SecureConversation handler interacts with the remote WS-SecureConversation service for as many round trips as needed. The whole workflow is shown in **Figure 3**, illustrated in two phases. The first phase is the initial interaction for session key exchange between the client side hander and the remote service of WS-SecureConversation; in the second phase, both sides communicate with each other with the derived session key stored in their contexts. The steps of the first phase are as follows:

1.0. The client application starts an initial remote call through a SOAP message.

1.1. The SOAP message is handled by the WS-SecureConversation handler, which finds that a secure conversation context is supposed to be established. It then blocks the message and makes a remote *RequestSecurityToken* (RST) call defined in WS-Trust to the WS-SecureConversation service, which is located in the same service container as the remote target service. Depending on the key exchange method, the SOAP message may contain the selected methods and algorithms, the required parameters, public keys, entropies, generated random session keys and other information.

1.2. As the WS-SecureConversation service receives the RST call, it interprets the information attached in the SOAP message according to its knowledge to the specified method, which is defined as schemas in most cases. It then fetches the security related information, such as credentials, from a security context resource if necessary. The security context resource has the general information about the owner who is running the service container.

1.3 The initialization interaction may continue for more than one round trip. The following SOAP message exchanges are under the name of *RequestSecurityTokenResponse* (RSTR) call, defined in WS-Trust. The RST with one or more *SecurityContextToken*s (SCT) is returned. A SCT has an identifier referring to a token established or to be established in the contexts. Finally, a session key will be agreed upon and stored in both sides' session contexts. At service side, the session context lies in the service's resource.

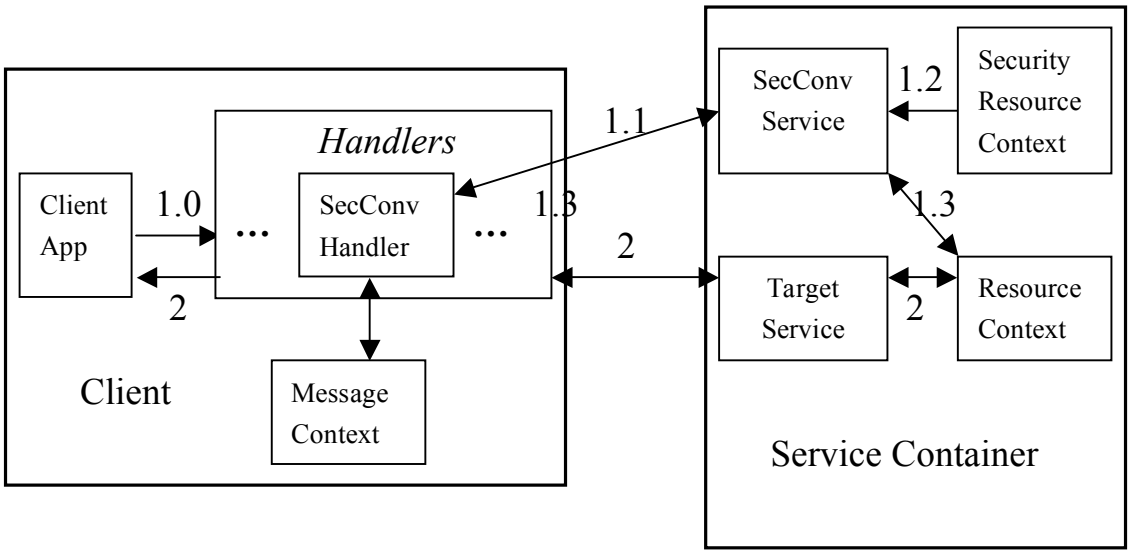

Figure 3 Server side WS-SecureConversation in Service

With the security context established, the WS-SecureConversation handler passes the first SOAP message on to the next handler, and thus the client begins to interact with the real target service in phase 2.

The model in **Figure 3** could be generalized to be the Web Services Trust model, as what **Figure 4** shows, with an independently functioning Service Token Service, described in WS-Trust. The service plays the role as a security authority. However, an externalized Service Token Service means extra trust relationship to be established between the Service Token Service and the target service. The service itself needs a full fledged authentication and authorization mechanism. Moreover, the WS-SecureConversation service will no longer interact with the target service's resource internally, but through the interfaces provided by a standard stateful service. Separating the WS-SecureConversation service from the targeting service container helps alleviate the load, as we can deploy or move the WS-SecureConversation service to another host.

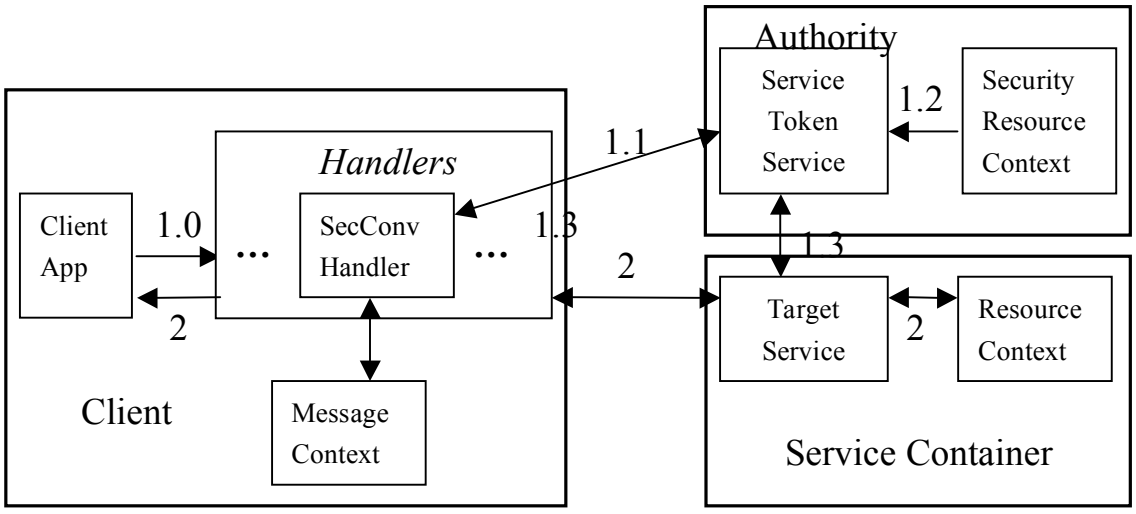

Figure 4 A Service Solution with Externalized Security Token Service

3.3 Integration of Password-based Key Exchange in WS-SecureConversation

The WS-SecureConversation related operations are actually defined in WS-Trust specification in a WSDL file. We created a *SecurityRequestor* interface to inherit the operations defined in the WS-Trust WSDL, and to import XML schemas of WS-Trust, WS-SecureConversation, and others. The WSDL inheritance is supported by GWSDL, an extension of WSDL 1.1 in Globus, and will soon be supported in WSDL 1.2. Any WS-SecureConversation services using this framework have to implement the *SecurityRequestor* interface to comply with WS-Trust and WS-SecureConversation.

In order to allow different security context establishment methods to be integrated, the method general behaviors are abstracted as interfaces *ClientNegotiator* and *ServerNegotiator* for both client side handler and server side service respectively. To integrate the AuthA method, we need to implement the corresponding interfaces as *AuthAClientNegotiator* and *AuthAServerNegotiator*. The AuthA method specific parameters are grouped as customized tokens, which are defined in XML Schema, including *ClientInitToken* (CIT), *ServerResponseToken*, (SRT) and *ClientResponseToken* (CRT). Each token contains one or more cryptographic parameters, such as the public keys and the authentication bits. These parameters can be either required or optional, depending on the working mode.

The instances of *AuthAClientNegotiator* and *AuthAServerNegotiator* are responsible for processing those tokens. WS-SecureConversation handler and service wrap up them with the WS-SecureConversation specific tokens, when processing a response. When dealing with a request, they unwrap the WS-SecureConversation specific tokens for protocol specific tokens. The instance of *AuthAServerNegotiator* is stored in the resource context for the current session, so that in the next round, WS-SecureConversation service could straightly fetch it from the resource context according to the session context ID number.

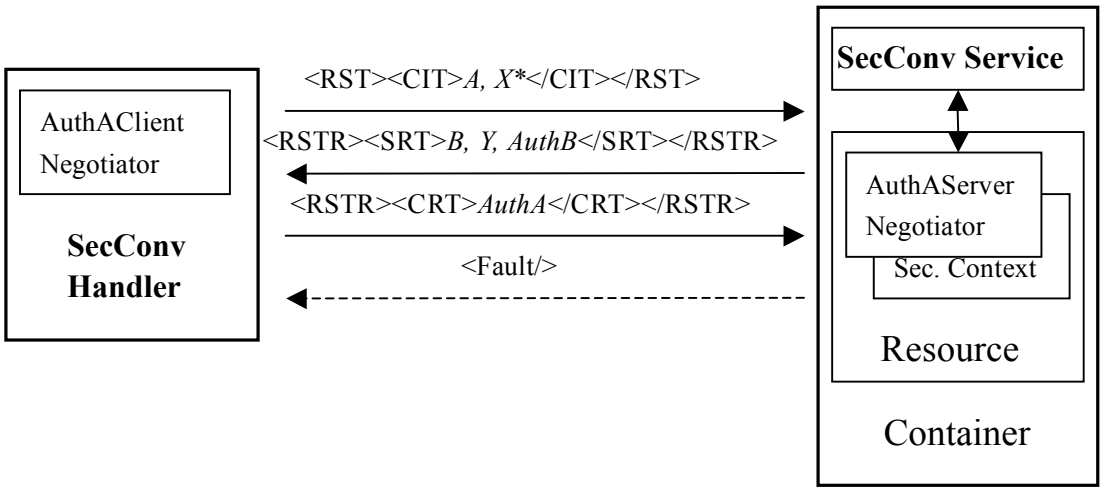

Figure 5 WS-SecureConversation Interaction in AuthA

Besides, we need to provide the corresponding signature and encryption handlers. These handlers mainly have two tasks. The first one is to fetch the secrets from the security contexts. The security context is in the resource context at the service side; while it lives in the background message context at the client side. The other task is to sign or encrypt the desired sections of the SOAP envelope with the symmetric keys obtained.

The security mode "WS-SEC-CONV" is also added. The programmers can state it programmatically or in

configuration files with minimal programming work, when writing their WS-SecureConversation applications.

3.4 Status

The AuthA method implementation is done in Java in the form of a library, with a dependency on Bouncy Castle library [BC]. The WS-Trust and WS-SecureConversation implementations are built on top of the pre-release version of Globus Toolkit 4.

4. Conclusion and Future Work

In this paper, we brought the password-based authenticated key exchange method to the message level security for run-time session key derivation. We first briefly described the WS-Trust and WS-SecureConversation specifications, which are emerging message level security specifications for security context establishment, sharing and derivation among multiple trust domains. We then introduced the AuthA method as a standardized password-based AKE method. Finally, we illustrated, how we have implemented a working system based on those specifications and the WSRF-compliant Globus Toolkit.

Our future work will first be dedicated to the hardening of the implementation such that we feel confident that it can be deployed in real production-like environments.

High on our priority is to investigate how the AuthA method could be integrated with One-Time Password (OTP) systems [OTP]. By treating the OTP as a shared secret, we could eliminate the threat of any man-in-the-middle who could potentially hijack the OTP-authentication and in addition provide mutual authentication in the process. Work is already underway to add OTP-features to the AuthA protocol [ACP03].

Furthermore, the mutual authentication version of AuthA method takes one and a half round trip in theory; however, in practice, it takes two full round trips, as the client's call to the service has to be returned, even with an empty envelope for the last message. We are investigating whether we could further optimize the use of the protocol by also exchanging protected application messages during the second round-trip, which would effectively reduce the overhead of the authenticated key exchange to a single round trip.

Lastly, the number of Denial of Services (DoS) attacks through the Internet has grown tremendously in the last couple of years. The effectiveness of DoS attacks can be decreased through the use of specific cryptographic mechanisms. Our previously published work on the AuthA method [AuthA-3] treats the amount of Perfect Forward-Secrecy (PFS) as an engineering parameter that can be traded off against resistance to DoS attacks. We plan to incorporate those techniques in future versions of our implementation.

Acknowledgements

We are pleased to acknowledge contributions to the implementation by Rachana Ananthakrishnan and Jarek Gawor, and we would like to thank Dennis Gannon and Ian Foster for their continuous support. The authors also thank David Pointcheval for its invaluable discussions on cryptographic issues related to this document.

References

[AuthA-1] M. Bellare and P. Rogaway, "The AuthA Protocol for Password-based Authenticated Key Exchange", March 14, 2000,

[AuthA-2] E. Bresson, O. Chevassut, and D. Pointcheval, "Security Proofs for an Efficient Password-Based Key Exchange", the 10th ACM Conference on Computer and Communication Security, Oct., 2003

[AuthA-3] E. Bresson, O. Chevassut, and D. Pointcheval, "New Security Results on Encrypted Key Exchange", the 7th International Workshop on Theory and Practice in Public Key Cryptography, March, 2004.

[AXIS] Apache Web Services Project, Axis, http://ws.apache.org/axis/

[BC] Legion of Bouncy Castle, Bouncy Castle library, http://www.bouncycastle.org/

[DH76] W. Diffie and M. Hellman, "New directions in cryptography", IEEE Transactions on Information Theory IT-22, 6 (Nov.), 1976, pp. 644-654.

[EKE] S. M. Bellovin and M. Merritt. "Encrypted Key Exchange: Password-Based Protocols Secure against Dictionary Attack", the Proc. Of the Symposium on Security and Privacy, IEEE, 1992, pp. 72-84.

[GRID] Foster, I. and Kesselman, C. Computational Grids. Foster, I. and Kesselman, C. eds. The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, 1999, 248. 2002.

[GGF] Global Grid Forum, http://www.ggf.org

[GT] I. Foster, C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit", Intl J. Supercomputer Applications, 11(2):115-128, 1997.

[GTO] Globus Alliance, http://www.globus.org

[OTP] IETF, "A One-Time Password System (RFC 2289)", February, 1998. http://www.ietf.org/rfc/rfc2289.txt

[MYPR] J. Novotny, S. Tuecke, and V. Welch, "An Online Credential Repository for the Grid: MyProxy", Proceedings of the Tenth International Symposium on High Performance Distributed Computing (HPDC-10), IEEE Press, Aug. 2001.

[OGSA] Open Grid Services Architecture, OGSA-working-group at GGF, https://forge.gridforum.org/projects/ogsa-wg

[PHYS] Foster, I., Kesselman, C., Nick, J. and Tuecke, S. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, Globus Project, 2002. http://www.globus.org/research/papers/ogsa. pdf.

[SBEW] M. Steiner, P. Buhler, T. Eirich, and M. Waidner, "Secure Password-Based Cipher Suite for TLS", the Proceedings of Network and Distributed Systems Security Symposium, San Diego, CA, Feb. 3-4, 2000, pp. 129-142.

[SecPerf] S. Shirasuna, A. Slominski, L. Fang, and D. Gannon, "Performance Comparison of Security Mechanisms for Grid Services", the 5th IEEE/ACM International Workshop on Grid Computing. Pittsburgh, Nov. 8, 2004.

[SSPKC] IEEE Standard 1363-2000. Standard Specifications for Public Key Cryptography. http://grouper.ieee.org/groups/1363, Aug. 2000.

[WSRF] OASIS, "Web Service Resource Framework", March 2004.

[WSS4J] Apache WSS4J http://ws.apache.org/ws-fx/wss4j

[WSSec] OASIS, "Web Services Security: SOAP Message Security" March 15 2004.

[WSSC] OASIS, "Web Services Secure Conversation Language" May 2004.

[WSTr] "Web Services Trust Language," BEA, Computer Associates, IBM, Layer7, Microsoft, Netegrity, Oblix, OpenNetwork, Ping Identity, Reactivity, RSA Security, VeriSign, Westbridge, March 2004.
[XMLSig] W3C Recommendation, “XML Signature Syntax and Processing” February 12 2002.
[XMLEnc] W3C Recommendation, “XML Encryption Syntax and Processing” December 10 2002.
[ACP03] M. Abdalla, O. Chevassut, and D. Pointcheval, Corruption in Password-Authenticated Key Exchange, Submitted for publication, August 2003.

Appendix

1. XML Schema for AuthA Tokens

```
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
                xmlns:aa='http://anl.gov/autha'
                targetNamespace='http://anl.gov/autha'
                elementFormDefault='qualified' >

  <xs:annotation>
    <xs:documentation>
      AuthA Schema for WS-SecureConversation security context establishment
    </xs:documentation>
  </xs:annotation>

  <xs:element name='ClientInitToken' type='aa:ClientInitTokenType' />
  <xs:complexType name='ClientInitTokenType' >
    <xs:sequence>
      <xs:element ref='aa:ClientName' minOccurs='0' />
      <xs:element ref='aa:ServerName' minOccurs='0' />
      <xs:element ref='aa:BitLength' />
      <xs:element ref='aa:P' />
      <xs:element ref='aa:G' />
      <xs:element ref='aa:X' />
      <xs:any namespace='##any' processContents='lax' minOccurs='0' maxOccurs='unbounded' />
    </xs:sequence>
    <xs:attribute ref='wsu:Id' use='optional' />
    <xs:anyAttribute namespace='##other' processContents='lax' />
  </xs:complexType>

  <xs:element name='ClientName' type='xs:string' />
  <xs:element name='ServerName' type='xs:string' />
  <xs:element name='BitLength' type='xs:int' />
  <xs:element name='P' type='xs:integer' />
  <xs:element name='G' type='xs:integer' />
  <xs:element name='X' type='xs:base64Binary' />

  <xs:element name='ServerResponseToken' type='aa:ServerResponseTokenType' />
  <xs:complexType name='ServerResponseTokenType' >
    <xs:sequence>
      <xs:element ref='aa:Y' />
```

```
        <xs:element ref='aa:AuthB' minOccurs='0' />
        <xs:any namespace='##any' processContents='lax' minOccurs='0' maxOccurs='unbounded' />
      </xs:sequence>
      <xs:attribute ref='wsu:Id' use='optional' />
      <xs:anyAttribute namespace='##other' processContents='lax' />
   </xs:complexType>

   <xs:element name='Y' type='xs:base64Binary' />
   <xs:element name='AuthB' type='xs:base64Binary' />

   <xs:element name='ClientResponseToken' type='aa:ClientResponseTokenType' />
   <xs:complexType name='ClientResponseTokenType' >
      <xs:sequence>
        <xs:element ref='aa:AuthA' minOccurs='0' />
        <xs:any namespace='##any' processContents='lax' minOccurs='0' maxOccurs='unbounded' />
      </xs:sequence>
      <xs:attribute ref='wsu:Id' use='optional' />
      <xs:anyAttribute namespace='##other' processContents='lax' />
   </xs:complexType>

   <xs:element name='AuthA' type='xs:base64Binary' />

</xs:schema>
```

2. SOAP Messages Sample in WS-SecureConversation

First Round Trip:

==========

Listen Port: 8080

Target Host: localhost

Target Port: 9090

==== Request ====

POST /wsrf/services/SecureCounterService-authService HTTP/1.0

Content-Type: text/xml; charset=utf-8

Accept: application/soap+xml, application/dime, multipart/related, text/*

User-Agent: Axis/1.2beta

Host: localhost:8080

Cache-Control: no-cache

Pragma: no-cache

SOAPAction:

"http://wsrf.globus.org/core/2004/07/security/secconv/SecurityRequestor/RequestSecurityTokenRequest"

Content-Length: 1742

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">
<soapenv:Header>
<wsa:MessageID soapenv:mustUnderstand="0">uuid:93F43…D15</wsa:MessageID>
<wsa:To
soapenv:mustUnderstand="0">http://localhost:8080/wsrf/services/SecureCounterService-authService</wsa:To>
<wsa:Action
soapenv:mustUnderstand="0">http://wsrf.globus.org/core/2004/07/security/secconv/SecurityRequestor/RequestSecurityTokenRequest</wsa:Action>
<wsa:From soapenv:mustUnderstand="0">
<Address
xmlns="http://schemas.xmlsoap.org/ws/2004/03/addressing">http://schemas.xmlsoap.org/ws/2004/03/addressing/role/anonymous</Address>
</wsa:From>
</soapenv:Header>
<soapenv:Body>
<RequestSecurityToken xmlns="http://schemas.xmlsoap.org/ws/2004/04/trust">
<TokenType>http://anl.gov/authaToken</TokenType>
<RequestType>http://schemas.xmlsoap.org/ws/2004/04/security/trust/Issue</RequestType>
<ns1:ClientInitToken xmlns:ns1="http://anl.gov/autha">
<ns1:ClientName>Liang</ns1:ClientName>
<ns1:ServerName>Ying</ns1:ServerName>
<ns1:BitLength>512</ns1:BitLength>
<ns1:P>119851960…3764553</ns1:P>
<ns1:G>726631…542917257</ns1:G>
<ns1:X>SYzFPU…YyKMpHNQ=</ns1:X>
</ns1:ClientInitToken>
</RequestSecurityToken>
</soapenv:Body>
</soapenv:Envelope>
==== Response ====
HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 1882

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">
<soapenv:Header>
<wsa:MessageID soapenv:mustUnderstand="0">uuid:942…7E0EF</wsa:MessageID>
<wsa:To
soapenv:mustUnderstand="0">http://schemas.xmlsoap.org/ws/2004/03/addressing/role/anonymous</wsa:To>

```
        <wsa:Action
soapenv:mustUnderstand="0">http://wsrf.globus.org/core/2004/07/security/secconv/SecurityRequestor/RequestSecur
ityTokenResponse</wsa:Action>
        <wsa:From soapenv:mustUnderstand="0">
            <Address
xmlns="http://schemas.xmlsoap.org/ws/2004/03/addressing">http://localhost:8080/wsrf/services/SecureCounterServi
ce-authService</Address>
        </wsa:From>
        <wsa:RelatesTo RelationshipType="wsa:Reply"
soapenv:mustUnderstand="0">uuid:93F…BCD15</wsa:RelatesTo>
    </soapenv:Header>
    <soapenv:Body>
        <RequestSecurityTokenResponse xmlns="http://schemas.xmlsoap.org/ws/2004/04/trust">
            <TokenType>http://anl.gov/authaToken</TokenType>
<RequestType>http://schemas.xmlsoap.org/ws/2004/04/security/trust/Issue</RequestType>
            <RequestedSecurityToken>
                <ns2:SecurityContextToken ns1:Id="_865986784"
xmlns:ns1="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
xmlns:ns2="http://schemas.xmlsoap.org/ws/2004/04/sc">
                    <ns2:Identifier> 000000fe-4f86-c681000002e265be</ns2:Identifier>
                </ns2:SecurityContextToken>
            </RequestedSecurityToken>
            <ns3:ServerResponseToken xmlns:ns3="http://anl.gov/autha">
                <ns3:Y>8eWRLb…HNQ=</ns3:Y>
                <ns3:AuthB>XKek…2rdTgpcU=</ns3:AuthB>
            </ns3:ServerResponseToken>
        </RequestSecurityTokenResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

Second Round Trip:

==========

Listen Port: 8080

Target Host: localhost

Target Port: 9090

==== Request ====

```
POST /wsrf/services/SecureCounterService-authService HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/1.2beta
Host: localhost:8080
Cache-Control: no-cache
Pragma: no-cache
```

SOAPAction:
"http://wsrf.globus.org/core/2004/07/security/secconv/SecurityRequestor/RequestSecurityTokenResponseRequest"
Content-Length: 1638

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">
<soapenv:Header>
<wsa:MessageID soapenv:mustUnderstand="0">uuid:94DA…8D9</wsa:MessageID>
<wsa:To
soapenv:mustUnderstand="0">http://localhost:8080/wsrf/services/SecureCounterService-authService</wsa:To>
<wsa:Action
soapenv:mustUnderstand="0">http://wsrf.globus.org/core/2004/07/security/secconv/SecurityRequestor/RequestSecurityTokenResponseRequest</wsa:Action>
<wsa:From soapenv:mustUnderstand="0">
<Address
xmlns="http://schemas.xmlsoap.org/ws/2004/03/addressing">http://schemas.xmlsoap.org/ws/2004/03/addressing/role/anonymous</Address>
</wsa:From>
</soapenv:Header>
<soapenv:Body>
<RequestSecurityTokenResponse xmlns="http://schemas.xmlsoap.org/ws/2004/04/trust">
<TokenType>http://anl.gov/authaToken</TokenType>
<RequestType>http://schemas.xmlsoap.org/ws/2004/04/security/trust/Issue</RequestType>
<RequestedSecurityToken>
<ns2:SecurityContextToken ns1:Id="_865986784"
xmlns:ns1="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
xmlns:ns2="http://schemas.xmlsoap.org/ws/2004/04/sc">
<ns2:Identifier> 000000fe-4f86-c681000002e265be</ns2:Identifier>
</ns2:SecurityContextToken>
</RequestedSecurityToken>
<ns3:ClientResponseToken xmlns:ns3="http://anl.gov/autha">
<ns3:AuthA>8aXvCzwO0Kn052W3GmoN/KhGKvw=</ns3:AuthA>
</ns3:ClientResponseToken>
</RequestSecurityTokenResponse>
</soapenv:Body>
</soapenv:Envelope>

==== Response ====
HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 1100

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">
    <soapenv:Header>
        <wsa:MessageID soapenv:mustUnderstand="0">uuid:953B…D8133</wsa:MessageID>
        <wsa:To
soapenv:mustUnderstand="0">http://schemas.xmlsoap.org/ws/2004/03/addressing/role/anonymous</wsa:To>
        <wsa:Action
soapenv:mustUnderstand="0">http://wsrf.globus.org/core/2004/07/security/secconv/SecurityRequestor/RequestSecur
ityTokenResponseRequestResponse</wsa:Action>
        <wsa:From soapenv:mustUnderstand="0">
            <Address
xmlns="http://schemas.xmlsoap.org/ws/2004/03/addressing">http://localhost:8080/wsrf/services/SecureCounterServi
ce-authService</Address>
        </wsa:From>
        <wsa:RelatesTo RelationshipType="wsa:Reply"
soapenv:mustUnderstand="0">uuid:94DAF…DC8D9</wsa:RelatesTo>
    </soapenv:Header>
    <soapenv:Body>
        <RequestSecurityTokenResponseResponse xmlns="http://schemas.xmlsoap.org/ws/2004/04/trust"/>
    </soapenv:Body>
</soapenv:Envelope>
==========