An Aspect-Oriented Approach to Distributed Object Security

Ruibing Hao, Ladislau Bölöni, Kyungkoo Jun, and Dan C. Marinescu
Computer Sciences Department
Purdue University, W. Lafayette, IN. 47907, U.S.A.
hao, boloni, junkk, dcm@cs.purdue.edu

Abstract

In this paper we present a security framework for Bond, a message-oriented distributed object middleware for network computing. Bond Security Framework, BSF, allows developers to exercise performance-security tradeoffs and use the security model best suited for a specific application and for a given environment. BSF consists of an extensible core and a set of well defined security interfaces. Any Bond object can become a secure object when extended with a dynamic property called bondSecurityContext.

1. Introduction to Distributed Object Security and Network Computing

Network computing is a novel paradigm that emphasizes the use of computational resources distributed over the network versus local resources. Several network computing architectures are possible, e.g. client-server, three-tier, transaction-oriented, etc. The phenomenal success of the World Wide Web has generated an interest in a Web of distributed objects capable of supporting network computing.

Security is an important concern for any network environment, as information in transit is vulnerable, and the use of resources in different administrative domain introduces issues of trust and consistency between them. A distributed object system poses new challenges to security mechanisms. For example security auditing should be able to identify correctly the principal, the original issuer of a request, even after a chain of calls involving multiple objects. There is also the need of delegation, the propagation of attributes of the principals between components. Delegation allows one component to act on behalf of a principal.

Applications of network computing have vastly different security requirements and the trade-off between security and performance is application specific. It is infeasible to consider one security model suitable for all applications and all environments [5]. Additional security challenges posed by network computing are discussed below. The user population and the resource pool are large and dynamic. A user may only be aware of a small fraction of the components involved in a computation. The relations among components may be rather complex, a component may act both as a server and a client at the same time. Traditional distributed systems use RPC or TCP/IP as their primary communication mechanism. In contrast, a distributed computing environment may use two-sided communication mechanism like message passing, streaming protocols, multicast, and/or single-sided get/put operations, as well as RPC. Components may communicate through a variety of mechanisms.

The boundaries of trust are more intricate because of dynamic characteristic of components. The trust users have in components is threatened when components can be mobile between hosts and new components can be created on the fly. Boundaries of trust are more complex because an activity typically involves multiple domains with different security policies and security models. Computation may be distributed to many more machines than any given user has control over.

Research in distributed computing security identifies several criteria and principles for security design [7, 10]. Granularity, consistency, scalability, flexibility, heterogeneity and performance are important aspects of distributed object security. A security design implies trade-off among these requirements. For example strong security and good performance are competing requirements. Coarse-grain security is easier to manage than fine-grain.

We survey now two security approaches, taken by Globus and Legion projects. Globus is a research effort to design computational grids [5]. Security is achieved through GSI, Globus Security Infrastructure. The grid environment consists of multiple trust domains - collections of subjects, participants in a security operation and objects - resources being protected, governed by single administration and a single security policy. GSI does not replace or override local policy decision, consequently, it focuses on controlling the inter-domain interactions and the mapping of inter-domain operations into local security policy.

When global and local subjects exist, for each trust domain, there exists a partial mapping from global to local subjects. The existence of the global subject enables the single sign–on. Operations between entities located in different trust domain require mutual authentication. An authenticated global subject mapped into a local subject is assumed to be equivalent to being locally authenticated as that local subject. All access control decisions are made locally on the basis of local subject. By enforcing security at the domain level, GSI implements coarse granularity security easy to manage and scale.

Legion is an object-oriented distributed computing environment [8]. Security in Legion is centered on the object level, making every object responsible for ensuring its own security [11]. Every object class decides who is permitted to communicate with that object. Rather than having a set of standard rights, every object will permit access on a per-method basis.

Legion provides a security framework rather than a specific implementation. Every Legion object may have a number of hooks, whereby additional functionality can be attached. These hooks are used for authentication, message encryption, access control, and delegation. These hooks can be left undefined, or they can be as complex as the object implementors desire. These hooks are implemented as member functions that are declared in some base classes but can be overridden as desired in derived classes.

`MayI()` is the Legion function responsible for enforcing access control. This function is implicitly called whenever an object attempts to invoke one of the object's member functions. Optionally, `MayI()` could issue licenses/tickets that permit later function calls to bypass it for a limited time or number of invocations.

Legion implements a very thin granularity security by providing security at the object level. Yet the only way to change the security models of an object is to override the implementation of the security hooks, it is not possible to modify dynamically the security model of an existing object when the environment has changed.

In Bond we opted for an extensible core object that can support multiple security models and can be added dynamically to existing object. This philosophy leads to several design principles. The first is to provide a framework for security, not force an implementation. Bond leaves the decision of choosing the format of credentials, the authentication policy, the access control policy, and so on, to the system developer or the system administrator. Bond Security Framework, BSF, is implemented as an extensible core Bond object called *BondSecurityContext* and a set of well-defined security interfaces.

The second principle is that various aspects of a complex object design, including security, should be separated from one another. In the initial design and implementation phase the creator of an object should only be concerned with functionality. Once the object is fully functional the creator needs to investigate the security requirements and augment the object with the proper security context by including a *probe* called BondSecurityContext. This dynamic property of a Bond object sets up a secure perimeter for the object, it intercepts all incoming and outgoing messages and enforces the security and access control models selected by the creator of the object.

The third principle is to support multiple authentication and access control models. This goal is achieved by defining a common interface for different security functions, like credential, authentication and access control.

The paper is organized as follows. Next section provides an overview of Bond. Section 3 outlines security and access control models. Section 4 presents the architecture and the implementation of the Bond Security Framework. An application is discussed in Section 5, followed by a summary.

2. Dynamic Properties of Bond Objects

Bond is a message-oriented middleware, for network computing [2]. It consists of a communication fabric and several frameworks. Bond servers are active objects providing long-term services for a group of users active in a Bond domain. Servers provide a variety of services e.g. directory service, authentication, persistent storage, versioning. Agents are active objects providing scheduling, brokerage, and so on. More information about Bond is available elsewhere [1, 3].

Bond objects are persistent network objects, that communicate with each other, can be instantiated and run remotely, and can be saved on persistent storage. They communicate with each other through messages. Some objects are active, they have one or more threads of control running, e.g. servers, agents, others are passive, e.g. the metaobjects used to annotate data, programs and hardware.

The upper segment of the Bond object hierarchy is presented in Figure 1. In this hierarchy, *bondObject* implements the common fields of all Bond objects, e.g. name, unique bondID, address, *static fields*, and *dynamic fields*. The static fields are declared during compile time and are implemented as data fields of the class. The dynamic fields are created during run-time and are implemented in the internal hashtable.

A property of a Bond object is a pair consisting of a name and a value object. The value object can be a single value or a collection of properties. For example, the name could be "credential" and the value *bondPAPCredential*, an object that holds the username and password. Bond objects have static and dynamic properties and implement them by means of static and dynamic fields. The following example shows the implementation of a credential as a dynamic property. The property consists of name, "creden-

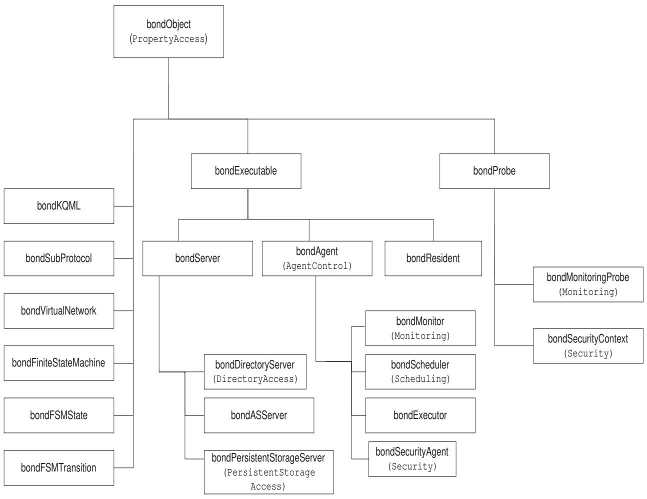

Figure 1. The upper part of the Bond object hierarchy. The sub-protocols implemented by class definitions appear in parenthesis.

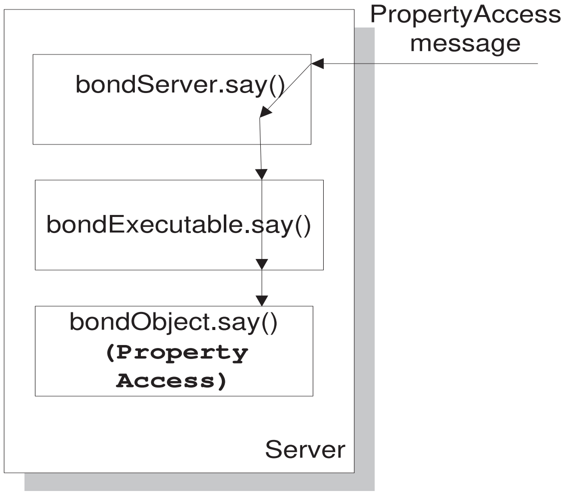

Figure 2. The handling of an incoming message. The message is passed along the object hierarchy to `say()` function of `bondObject` that implements PropertyAccess sub-protocol.

tial" and an instance of bondPAPCredential class as value object. Two static fields, username and password are available in class bondPAPCredential. The `get()` and `set()` methods provide access to static and dynamic fields of a `bondObject`. Both methods check the property names in the static fields before searching the hashtable that holds the dynamic properties.

```
Class bondPAPCredential extends bondObject {
  String username;
  String password;
}
Class MyProgram extends bondObject {
  bondPAPCredential mycredential
     = new bondPAPCredential();
  //creation of username property
  mycredential.set("username", "hao");
  //creation of password property
  mycredential.set("password",  "abcde");
  // creation of quote property
  this.set("credential", mycredential);
  bondPAPCredential mycredential
    = this.get("credential");
}
```

Bond uses KQML, Knowledge Querying and Manipulation Language [4], as an inter–object communication language. A KQML message consists of a *performative* and a set of order-independent parameters. The performative encodes basic abstractions like asking, replying, achieving, subscribing, etc. The number and semantics of parameters are dependent on the performative except some reserved parameters.

A sub-protocol is a sub-set of KQML messages [3]. It contains messages needed to perform a particular task and it is closed in the sense that the reply or acknowledgement to a message is always a member of the same sub-protocol. Each message is stamped with the sub-protocol it belongs to. Examples of Bond sub-protocols are given in [3]. The objects that *implement* a sub-protocol understand its syntax and semantics. For example, two objects that implement *PropertyAccess* sub-protocol can remotely interrogate and set each other's properties.

Bond objects are grouped together in *containers*. A Bond *resident* is such a container and consists of several threads of control, a local directory and a communication interface. Bond is based upon active messages, every incoming message is delivered to the `say()` function of the destination object by the messaging thread of the container where the object is located. The `say()` function implements parsing and handling of the message. The `say()` function is either inherited from a parent or overridden by the object. The sub-protocol implementation is also inherited. If an incoming message is not understood, it is passed to the `say()` function of the immediate ancestor in the object hierarchy until the message is understood or there are no more elements in the hierarchy. Every object can understand the PropertyAccess sub-protocol implemented by the root of the object hierarchy as shown in Figure 2.

Bond uses *probe* [3] to add new functionality to Bond objects dynamically. A probe is an object that implements a certain sub-protocol and is attached to an object as a dynamic property. The implementation of `bondObject` ex-

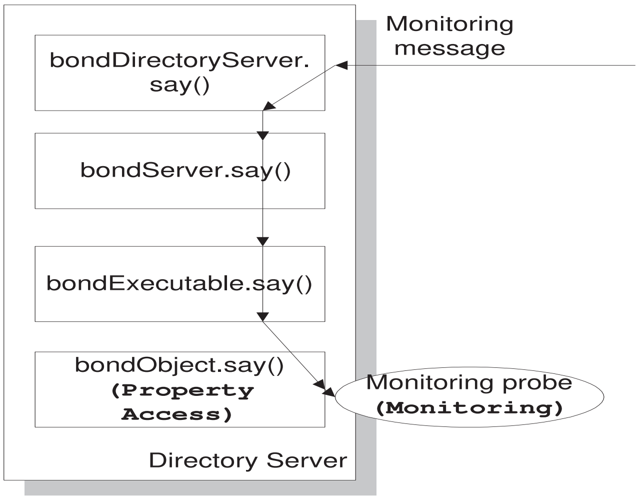

Figure 3. Since `bondDirectoryServer` **is unable to understand the monitoring sub- protocol, a message is passed along the hierarchy to the monitoring probe.**

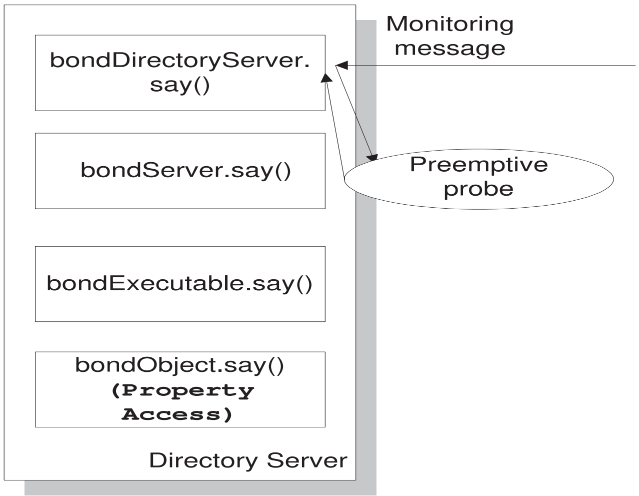

Figure 4. `bondDirectoryserver` **has a monitoring probe as a preemptive probe, by which monitoring message is handled.**

ploits inheritance and examines the list of dynamic properties to process incoming messages. When an object cannot understand a message by tracing upward the object hierarchy, it searches dynamic properties looking for a probe capable of handling the sub-protocol the message belongs to and lets the probe handle the message. For example, the Bond Monitoring Framework is based upon a monitoring probe [3], as shown in Figure 3.

The Bond security framework is based upon the concept of *preemptive probe*, an abstraction that supports the aspect-oriented design outlined above. The preemptive probe is activated before any attempts are made to deliver the message to the object, it intercepts all messages sent to the object. Figure 3 and 4 show the strategies used to deliver a message to an object. The set of dynamic properties of an object is searched first looking for a preemptive probe. If the preemptive probe is not found, then the message is passed along the object hierarchy. Finally, if no ancestor of the object understand the sub- protocol, then the dynamic properties of the target object are searched for a regular probe able to understand the sub-protocol. Multiple probes and only one preemptive probe may be attached to an object.

3. Security and Access Control Models

Security in a network environment includes authentication and access control [9, 6]. Authentication refers to the process of identifying an individual, usually based onx username and password. Access Control is the process of granting or denying access to a network based on a two-step process, authentication to ensures that a user is who he/she claims to be, and access control policy which allows the user access to various resources based on the user's identity.

Some of the authentication models are [10]:

- PAP - Password Authentication Protocol. The most basic form of authentication, the user's name and password are transmitted over the network and compared to a table of name-password pairs. Typically, the stored passwords are encrypted.
- CHAP - Challenge Handshake Authentication Protocol. The authentication agent, typically a network server, sends the client program a key to encrypt the username and the password.
- Kerberos - ticket-based authentication. The authentication server assigns a unique key, called a ticket, to each user that logs on to the network. The ticket is then embedded in every message to identify the sender of the message.
- Certificate-based authentication. This model is based on public key cryptography. Each user holds two different keys: public and private. The user can get a certificate that proves the binding between the user and its public key from a third party. The private key is used to generate evidence that can be sent with the certificate to server side. The server uses the certificate and evidence to verify the identity of the user.

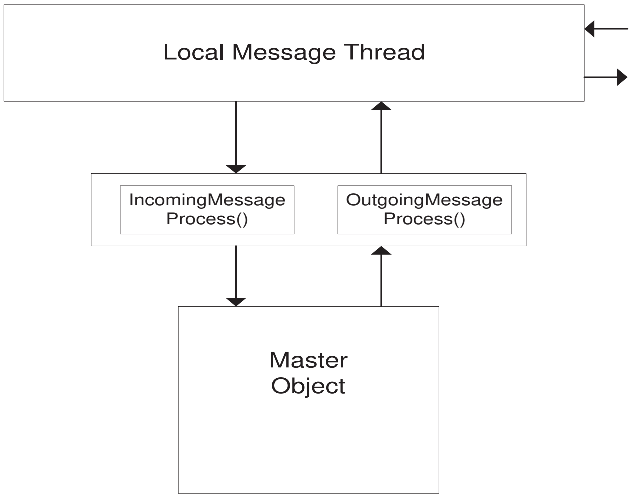

Figure 5. Message flow for an object with a preemptive probe, `bondSecurityContext`

Credential is a secret code that proves the identity of an individual. Authentication models use different credentials, e.g. username/password in PAP and CHAP, user identifier/ticket in ticket-based authentication, and user certificate/private key in the certificate-based authentication.

Access control models include firewall and access control list, ACL. Firewall grants or denies access based upon the IP address of the requester. An access control list specifies what operations a user may perform on each resource.

4 The Architecture and Implementation of BSF

BSF is implemented through an extensible core called BondSecurityContext and a set of well-defined security interfaces. BondSecurityContext is a preemptive probe that establishes a defense perimeter for the object it is attached to, by intercepting incoming and outgoing messages with two methods: `incomingMessageProcess()` and `outgoingMessageProcess()` as shown in Figure 5.

A BondSecurityContext contains security-related objects that implement various security functions. Each of those objects implements one of the security interfaces defined by Bond:

- *BondCredentialInterface* - defines the method to access the credential possessed by the current BondSecurityContext. The format of a credential was discussed in the previous section. This interface provides two groups of methods:
  - Methods to respond to authentication request from a remote object. Usually a challenge is contained in the authentication request, and the response is derived from both the challenge and the information provided by the credential. The response is generated differently depending on security models.
  - Methods to generate a user identifier and a proof to be embedded in each outgoing message and prove to the receiver the identity of sender. The proof has different meaning in different security models. In a username/password model, the proof can be a password, or an encrypted password, in a ticket based security model, the ticket itself can be a proof, in a certificate–based model, the evidence generated by encrypting a random string with the private key can be an eligible proof.
- *BondAuthenticatorInterface* - defines the authentication method for each message received by an object. The developer or the administrator may deploy one of the authentication models mentioned earlier. The only restriction is to adhere to this interface. The only method provided by this security interface is `authenticateClient()`, which returns an authenticated user identifier. This identifier can be used for access control or auditing.
- *BondAccessControlInterface* - defines the access control method for each message received by an object. The methods provided by this security interface are `initACL()` and `checkRight()` based upon the authentication models discussed earlier.

The code below illustrates the implementation of bondSecurityContext that supports authentication and access control in the `incomingMessageProcess()`.

```
bondSecurityContext extends bondProbe {
 private bondCredentials bcs;
 private bondAuthenticatorInterface bau;
 private bondAccessControlInterface bac;

 // incomingMessageProcess
 // is called by the message
 // thread on each received message
 public void incomingMessageProcess(m, sender){
  // 1.authenticate the received message
  authenticated_user_id
   = bau.authenticateClient(m);
  if( authenticated_user_id == null ){
    sender.say( sorry message );
    return;}
  // 2.enforce access control
  result
   = bac.checkRight(authenticated_user_id,m);
  if( result == false){
```

Table 1. Authentication models

Type	Bond Credential Interfaces	Bond Authenticator Interfaces
Username and Plain password	bondPAP-Credentail	bondPassword-Authenticator
Challenge Handshake Authenticator Protocol (CHAP)	bondCHAP-Credential	bondChallenge-Authenticator

Table 2. Access control models

Type of access control	Access Control Interface	Required authenticator
IP address based (Firewall)	bondIPAddress-AccessControl	-
Access control list	bondNameBased-AccessControl Or bondRightBased-AccessControl	bondChallenge-Authenticator or bondPassword-Authenticator

```
  sender.say( sorry message );
  return;}
 // 3.send the passed message to the object
 parent.say(m, null); }
```

The code also shows several objects that implemented the security interfaces defined above.

Table 1 lists the authentication models and Table 2 lists the access control models implemented in Bond release 2.0.

All authenticators in Table 1 need a Bond Authentication Server maintaining the usernames and the passwords. If the service provider uses one type of authenticator, the client should use the corresponding credential to make the authentication successful.

5. Application

The following example illustrates how to construct secure objects using BSF. Assume that we have a Bond domain consisting of one client, two generic servers and an authentication server that provides account management and authentication services. The client uses an existing account (`uid=hao` and `passwd= abcde`) to access services provided by the two servers, as shown in Figure 6. One of them, `serverA` enforces plain password-based authentication and firewall-based access control, while `serverB` enforces CHAP-based authentication and name–based access control.

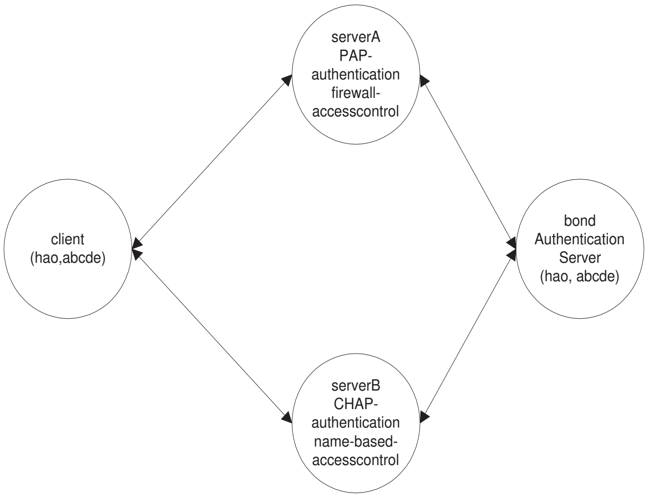

Figure 6. A Bond domain consisting of a client, two computational servers and an authentication server.

The code below shows how to setup serverA as a secure object enforcing plain password-based authentication and firewall-based access control.

```
// create a new servr object, serverA
server serverA = new server();

// create a plain-password based
// authenticator
bondPasswordAuthenticator bau
  = new bondPasswordAuthenticator(baserver);

// create a firewall-based access
// controller and initialize it
bondIPAddressAccessControl bac
  = new bondIPAddressAccessControl();
bac.initACL("firewall.acl");

// create a security context
bondSecurityContext gatekeeper
  = new bondSecurityContext(serverA);

// set the access controller and
// authenticator of the security context
gatekeeper.setAccessControl(bac);
gatekeeper.setAuthenticator(bau);

// set the security context into serverA
serverA.setSecurityContext(gatekeeper);
```

The format of the access control list file ‘firewall.acl’ is:

```
* Firewall configuration file consisting
* of pairs <hostname><mask>
uhuru.cs.purdue.edu 255.255.255.0
```

Hosts in the same sub-net with the machine uhuru.cs.purdue.edu can access serverA.

The code below shows how to create a secure object enforcing CHAP-based authentication and name- based access control.

```
// create a new server object, serverB
server serverB= new server();

// create a CHAP-based authenticator
bondChallengeAuthenticator bpau
  = new bondChallengeAuthenticator(baserver);

// create name-based access controller
// and initialize it
bondNameBasedAccessControl bac
  = new bondNameBasedAccessControl();
bac.initACL("names.acl");

// create a security context which
// has the server object as the master object
bondSecurityContext gatekeeper
  = new bondSecurityContext(serverB);

// set the access controller and
// authenticator of this security context
gatekeeper.setAccessControl(bac);
gatekeeper.setAuthenticator(bpau);

// set the security context as a
// dynamic property of serverB
serverB.setSecurityContext(gatekeeper);
```

The format of the access control list file names.acl is:

```
*
* Name based ACL, the format of
* this file is as following
* <name> <right1,right2,rightN>
hao persistent-object-read,persistent-object-
write
```

This means that user `hao` is allowed to save objects to and reload them from this server.

The parameter, `baserver`, is used to create the authenticators in both cases. This means serverA and serverB share the account information stored in the `baserver`, the authentication server of the domain.

The procedure to set up a client as a secure object is shown below.

```
// create a client object
client clio = new client();

// create a security context
// which has the client as master object
bondSecurityContext bsc
  = new bondSecurityContext(clio);

// setup a PAP credential which
// contains the username and password
bondPAPCredential bc1
  = new bondPAPCredential("hao","abcde");
bsc.setCredential(bc1, "serverA");
```

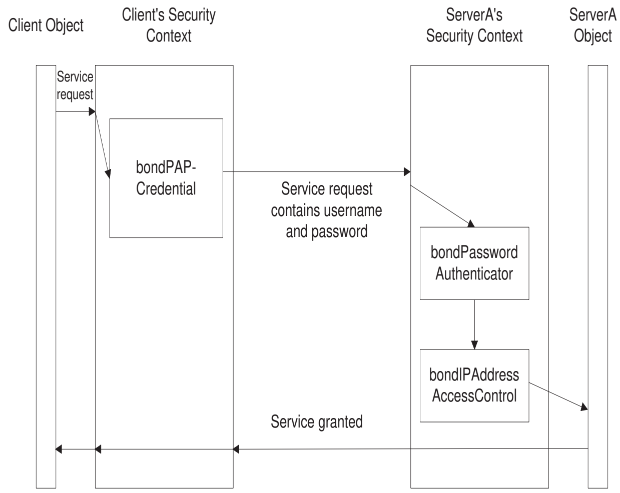

Figure 7. Processing of a service request.

```
// setup a CHAP credential which
// contains the username and password
bondCHAPCredential bc2
  = new bondPAPCredential("hao","abcde");
bsc.setCredential(bc2, "serverB");

// setup this security context
// as client's security context
clio.setSecurityContext(bsc);
```

Once properly set up, `bsc` adds appropriate credentials to outgoing requests by checking destinations. In the above example, requests to serverA are associated with bondPAPCredential, while those to serverB are with bondCHAPCredential.

A scenario involving the interaction between the client and serverA is shown in Figure 7. The client sends serverA a request messsage, which is in turn intercepted by the security context of the client. After inserting a username `hao` and a password `abcde` into the message, the security context sends it to serverA. When the message reaches serverA, it is intercepted by the security context of serverA enforcing authentication and access control. After validating the username and password, serverA grants the service.

The scenario illustrated in Figure 7 is appropriate when the server trusts the identifier and the proof contained in a message. But the identifier and proof can be captured by a malicious third party and used to obtain unauthorized access to the server. To prevent such attacks, the security context of the server may use a stronger authentication scheme as shown in Figure 8.

The client sends a service request message to serverB. The security context of the client does not modify the message because it detects that a `bondCHAPCredential` is used. The message is captured by the security context of

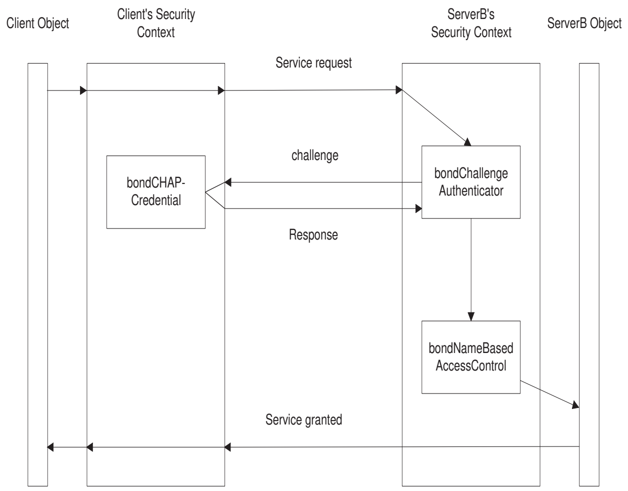

Figure 8. Processing of a service request using CHAP credential.

serverB. The authenticator component of the security context of the server sends a challenge to the credential component of the security context of the client and expects a response derived from both the challenge and information contained in client's credential. Then the authenticator uses the challenge and corresponding response to authenticate the client. If the service request is validated, the server object grants the service.

6. Summary

In this paper we present Bond Security Framework. We opted for an extensible core object capable of supporting multiple security models and can be added dynamically to existing object. We decided to provide a framework for security, not to force an implementation. Bond leaves the decision of choosing the format of credentials, the authentication policy, the access control policy, and so on, to the system developer or the system administrator. BSF is implemented as an extensible core Bond object and a set of well-defined security interfaces.

Various aspects of a complex object design, including security, are separated from one another. In the initial design and implementation phase the creator of an object should only be concerned with functionality. Once the object is fully functional the creator needs to investigate the security requirements and augment the object with the proper security context by including a preemptive probe. This dynamic property of a Bond object sets up a secure perimeter for the object, it intercepts all incoming and outgoing messages and enforces the security and access control models selected by the creator of the object.

We support multiple authentication and access control models. This goal is achieved by defining a common interface for different security functions, like credential, authentication and access control.

The security framework presented here is included in the current Bond release. The information about the Bond system and the code is available for downloading at `http://bond.cs.purdue.edu`.

7. Acknowledgements

The research reported in this paper is partially supported by the National Science Foundation grant MCB-9527131, by the Scalable I/O initiative, by a grant from the Computational Science Alliance, and by a grant from Intel Corporation.

References

[1] L. Bölöni. Bond Objects - a White Paper. Technical Report CSD-TR 98-002, Comp. Sci. Dept. Purdue University, 1998.

[2] L. Bölöni, R. Hao, K. Jun, and D. C. Marinescu. Structural Biology Metaphors Applied to the Design of a Distributed Object System. To appear in 4th IEEE Symposium on Computers and Communications.

[3] L. Bölöni, R. Hao, K. Jun, and D. C. Marinescu. Subprotocols: an Object Oriented Solution to Semantic Understanding of Messages in a Distributed Object System. Submitted.

[4] T. Finin and et al. Specification of the KQML Agent-Communication Language. DARPA Knowledge Sharing Initiative draft, June 1993.

[5] I. Foster and C. Kesselman. The Globus Project: A Status Report. In *Proc. IPPS/SPDP'98 Heterogeneous Computing Workshop*, pages 4–18, 1998.

[6] R. Hao, K. Jun, and D. C. Marinescu. Bond System Security and Access Control Model. In *Proc. IASTED International Conference on Parallel and Distributed Computer and Networks*, pages 520–524. ACTA PRESS, December 1998.

[7] J. Knudsen. *Java Cryptography*. O'Reilly, 1998.

[8] M. Lewis and A. Grimshaw. The Core Legion Object Model. In *Proc. 5th IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, August 1996.

[9] A. Rubin, D. Geer, and M. Ranum. *Web Security Sourcebook*. John Wiley & Sons Inc., 1997.

[10] B. Schneier. *Applied Cryptography, 2nd*. John Wiley & Sons Inc., 1996.

[11] W. Wulf, C. Wang, and D. Kienzle. A New Model of Security for Distributed Systems. Technical Report CS-95-34, Dept. of Computer Science, University of Virginia, 1995.