Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports | Department of Computer Science

1998

An Aspect-Oriented Approach to Distributed Object Security

Riubing Hao

Ladislau Bölöni

Kyungkoo Jun

Dan C. Marinescu

Report Number:

98-038

Hao, Riubing; Bölöni, Ladislau; Jun, Kyungkoo; and Marinescu, Dan C., "An Aspect-Oriented Approach to Distributed Object Security" (1998). *Department of Computer Science Technical Reports.* Paper 1425.
https://docs.lib.purdue.edu/cstech/1425

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

AN ASPECT-ORIENTED APPROACH TO DISTRIBUTED OBJECT SECURITY

Ruibing Hao
Ladislau Boloni
Kyungkoo Jun
Dan C. Marinescu

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR #98-038
November 1998

An Aspect-Oriented Approach to Distributed Object Security

Ruibing Hao, Ladislau Bőlőni, Kyungkoo Jun, and Dan C. Marinescu
(hao, boloni, junkk, dcm@cs.purdue.edu)
Computer Sciences Department
Purdue University
West Lafayette, IN, 47907, USA

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.*

Contents:

1. Abbreviations and Terms
2. Introduction to Distributed Object Security
3. Dynamic Properties of Bond Objects
4. Security and Access Control Models
5. Bond Security Framework
6. Applications
7. Summary
8. Acknowledgments
9. Literature

1. Abbreviations and Terms

Bond Object – a collection of data fields and methods. The data component of an object consists of several *fields* defined at compile-time and *dynamic properties* defined at run-time.

Message-oriented system – a distributed system whose components communicate only by means of messages. Messages are used to implement remote data access and remote procedure calls. Every message in Bond is a sentence in the KQML agent communication language.

Bond sub-protocol – a closed subset of KQML messages, a specialized language to perform a task. Each Bond message is stamped with the sub-protocol it belongs to. A Bond object understands all sub-protocols implemented by its own class or inherited from its ancestors.

Bond probe - an object that implements a certain sub-protocol and can be attached to another object as a dynamic property to extend messaging capability.

Preemptive probe – a probe that intercepts all incoming and outgoing messages.

Bond security context – a preemptive probe that implements security-related functions. When attached to a Bond object, it sets up a security perimeter for the object and intercepts every incoming and outgoing message to enforce security and access control.

Secure Bond Object – an object augmented with a security context.

Security Interface – defines a set of common methods to access an object that implements a security and an access control model.

Credential – a secret code that can prove the identity of an individual.

Authentication - the process of ensuring that an individual is whom he/she claims to be.

Access Control - the process of granting or denying access to a resource, usually based on authentication.

2. Introduction to Distributed Object Security and Network Computing

Network computing is a novel paradigm that emphasis 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 to support 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 are application specific. It is unfeasible to consider one security model suitable for all applications and all environments [1]. 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 [2,3]. 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 [1]. 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 [4]. Security in Legion is centered on the object level, making every object responsible for ensuring its own security [5]. 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 implementor desires. 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. BSF is implemented as an extensible core Bond object called *BondSecurityContext* and a set of well-defined security interfaces.

The second design 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 design 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 4 outlines security and access control models. Section 5 presents the architecture and the implementation of the Bond Security Framework. Applications are discussed in Section 6, followed by a summary.

3. Dynamic Properties of Bond Objects

Bond is a message-oriented middleware, for network computing [6]. 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 [7,8].

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 hash table.

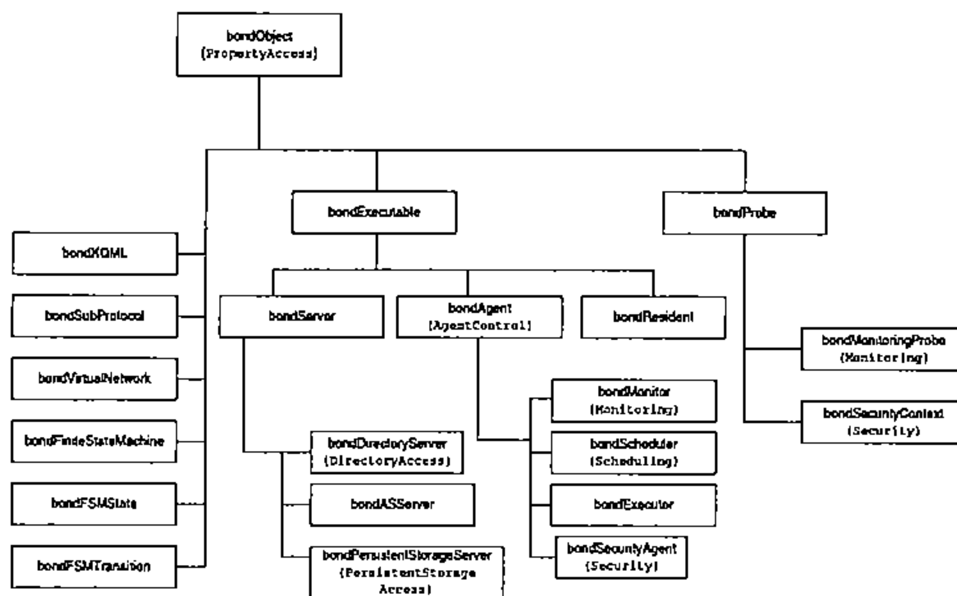

Figure 1. The upper part of the Bond object hierarchy. The subprotocols implemented by class definitions appear in parenthesis.

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, "credential" 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();
    mycredential.set("username", "hao");    //creation of username property
    mycredential.set("password", "abcde"); //creation of password property
    this.set("credential", mycredential);    // creation of quote property
    bondPAPCredential mycredential = this.get("credential");
}
```

Bond uses KQML, Knowledge Querying and Manipulation Language [9], 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 [8]. 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 [8]. The objects that *implement* a sub-protocol understand its syntax and semantics. For example, two objects that implement the *PropertyAccess* sub-protocol can remotely interrogate and set each other's properties.

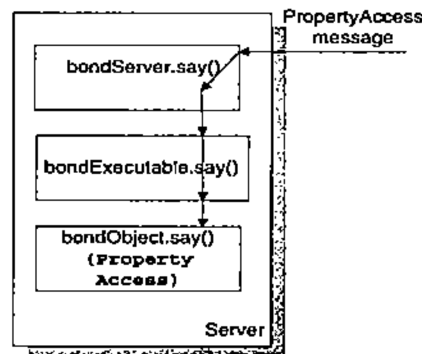

Figure 2: The handling of an incoming message. The message is passed along the object hierarchy to the *say ()* function of the *bondObject* that implements the *PropertyAccess* sub-protocol.

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 the 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 the concept of *probe* [8] to add new functionality to a Bond object dynamically. A probe is an object that implements a certain sub-protocol and it is attached to an object as a dynamic property. The implementation of a *bondObject* exploits inheritance and examines the list of dynamic properties to process incoming messages. When an object cannot understand a message by tracing the object hierarchy, it searches its list of dynamic properties looking for a probe that can handle the sub-protocol the message belongs to and leaves the message handling to the probe. For example, the Bond Monitoring Framework is based upon monitoring probes [10], as shown in Figure 3.

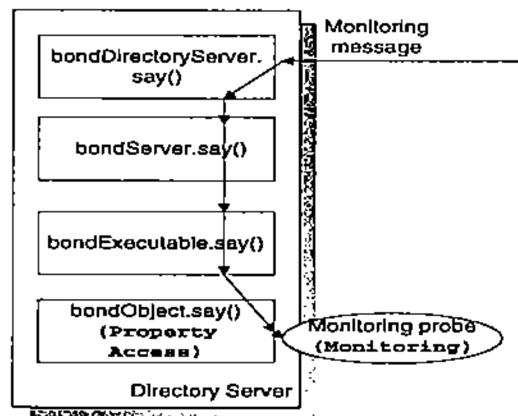

Figure 3. The effect of augmenting an object with a probe. The destination object, *bondDirectoryServer*, does not understand the monitoring sub-protocol. The message is passed along the hierarchy to the *say ()* function of the *bondObject* but none of the ancestors understand the sub-protocol. Finally, the messaging thread searches the dynamic of properties of the destination object, detects a probe that understands the sub-protocol and delivers the message to it.

The Bond security framework is based upon the concept of preemptive probe, an abstraction that supports the aspect-oriented design outlined above. A *preemptive probe* is a special probe activated before any attempt is made to deliver the message to the object, it intercepts all messages sent to the object. Figures 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.

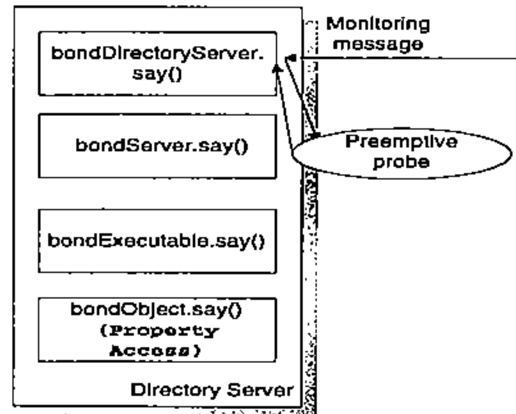

Figure 4. The *bondDirectoryserver* object has a preemptive probe attached to it and a monitoring message is handled by the probe.

4. Security and Access Control Models

Security in a network environment includes authentication and access control [11][12]. Authentication refers to the process of identifying an individual, usually based on a 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 proper which allows the user access to various resources based on the user's identity.

Some of the authentication models are [3]:
(1) 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.
(2) 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.
(3) 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.
(4) 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.

A 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 includes 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.

5. 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 using two methods: *incomingMessageProcess()* and *outgoingMessageProcess()* as shown in Figure 5.

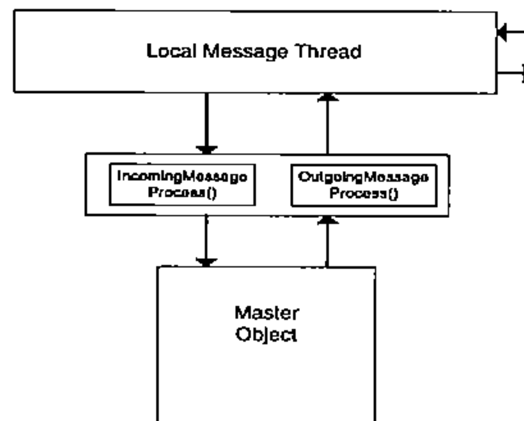

Figure 5. Message flow for an object with a preemptive probe, *bondSecurityContext* attached to it.

A *BondSecurityContext* contains security-related objects that implement various security functions. Each of these 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:

(1) 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 in different security models.
(2) Methods to generate a user identifier and the 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 the 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()*, that 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 current implementation of *bondSecurityContext* supports authentication and access control in the *incomingMessageProcess() method,* as shown in Figure 6, and credential setup in the *outgoingMessageProcess*() method. Figure 6 also shows several objects that implemented the security interfaces defined above. Member variable *bcs* is a collection of credentials needed for accessing different services, every element inside this collection implements the *BondCredentialInterface*; Member variable *bau* is a authenticator object that implements the *BondAuthenticatorInterface*; Member variable *bac* is a access controller object that implements the *BondAccessControlInterface*;

```
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){

    // Step 1.authenticate the received message
    authenticated_user_id = bau.authenticateClient(m);
    if( authenticated_user_id == null ){
            sender.say( sorry message );
            return;
    }

    // Step 2.enforce access control
    result = bac.checkRight(authenticated_user_id,m);
    if( result == false){
            sender.say( sorry message );
            return;
    }

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

Figure 6. BondSecurityContext member variables and the *incomingMessageProcess()* method. Steps 1, 2 and 3 enforce authentication, access control, and message delivery respectively.

A developer may implement application specific security models by adhering to the defined security interfaces, and deploy these implementations in the *BondSecurityContext* object. For example, the member variable *bau* can be set as an authenticator that implements the certificate-based authentication model. The only requirement is it must implement the *bondAuthenticatorInterface* interface.

A developer may derive a new class from bondSecurityContext and override the incomingMessageProcess() and outgoingMessageProcess(). For example one may add accounting and logging functions.

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

Type	Bond Credential Interfaces	Bond Authenticator Interfaces
Username and Plain password	bondPAPCredential	bondPasswordAuthenticator
Challenge Handshake Authentication Protocol (CHAP)	bondCHAPCredential	bondChallengeAuthenticator

Table 1. Authentication models in Bond.

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 in Table 1, the client object should use the corresponding credential in Table 1 to make the authentication feasible.

Type of access control	Access Control Interface	Required authenticator
IP address based (Firewall)	bondIPAddressAccessControl	-
Access control list	bondNameBasedAccessControl Or bondRightBasedAccessControl	bondChallengeAuthenticator or bondPasswordAuthenticator

Table 2. Access control models implemented in Bond.

6. Applications

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. One of them, server_A enforces plain password-based authentication and firewall-based access control, while server_B enforces CHAP-based authentication and name-based access control.

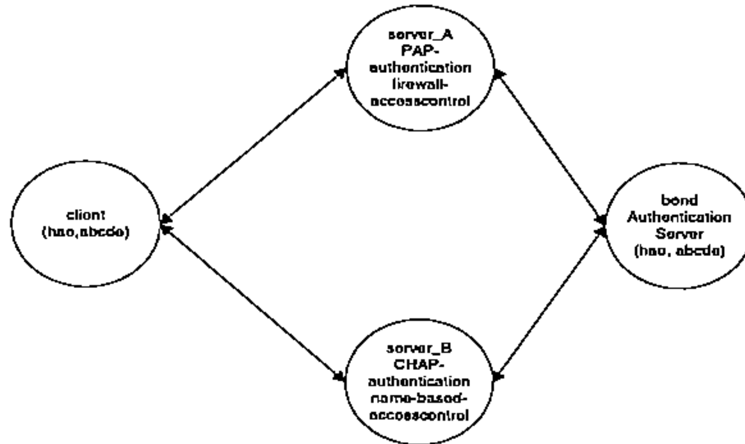

Figure 7. A Bond domain consisting of a client, two computational servers and an authentication server.

The sample code to setup *server_A* as a secure object enforcing plain password-based authentication and firewall-based access control is presented in Figure 8. The steps involved in this process are: (1) create a new server object *server_A*, (2) create a plain-password based authenticator *bau* and a firewall-based access controller *bac*, (3) create a security context called *gatekeeper* and set *bau* and *bac* as the authenticator and the access controller respectively, and (4) set *gatekeeper* as the security context of *server_A*.

```
{
 server server_A = new server();
 // create an authenticator for the security context
 bondPasswordAuthenticator bau = new bondPasswordAuthenticator(baserver);

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

 // create a security context for the newly created object
 bondSecurityContext gatekeeper = new bondSecurityContext(server_A);

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

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

Figure 8. Sample code to create a secure server enforcing plain password-based authentication and firewall-based access control.

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 server_A.

The procedure to create a secure object enforcing CHAP-based authentication and name-based access control presented in Figure 9 consists of the following steps: (1) create a new server object *server_B*, (2) create a CHAP based authenticator *bpau* and a name-based access controller *bac*, (3) create a security context called *gatekeeper* and set *bpau* and *bac* as its authenticator and access controller respectively, and (4) set *gatekeeper* as the security context of the server.

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 *server_A* and *server_B* share the account information stored in the *baserver*, the authentication server of the domain.

```
{
 server server_B= new server();
 // create an authenticator for using in the security context
 bondChallengeAuthenticator bpau =  new bondChallengeAuthenticator(baserver);

 // create 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(server_B);

 // 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 server object
 server_B.setSecurityContext(gatekeeper);
}
```

Figure 9. Sample code to create a secure server enforcing plain password-based authentication and IP-based access control.

Now consider the client side. The procedure to setup a secure client object shown in Figure 10 shows involves the following steps: (1) create a new client object *clio*, (2) create a security context *bsc* for the object (3) create a bondPAPCredential *bc1* to hold the username and password, and set it as the credential for accessing *server_A* in the security context, (4) create a bondCHAPCredential *bc2* which to hold the username and

password, and set it as the credential for accessing *server_B* in the security context, and (5) set *bsc* as the security context of the client object. The security context of the client adds the corresponding credential to each request sent to one of the server objects without the client intervention.

```
{
 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, "server_A");
 // 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);
}
```

Figure 10. Sample code to create a client able to interact with the two secure servers.

A scenario involving the interaction between the client and server_A is shown in Figure 11. The client sends a service request to server_A. This request is intercepted by the security context of the client, which inserts the username *hao* and the password *abcde* and then sends the message. Messages reaching server_A are intercepted by its own security context which enforces authentication and access control. If the service request is validated the server object grants the service.

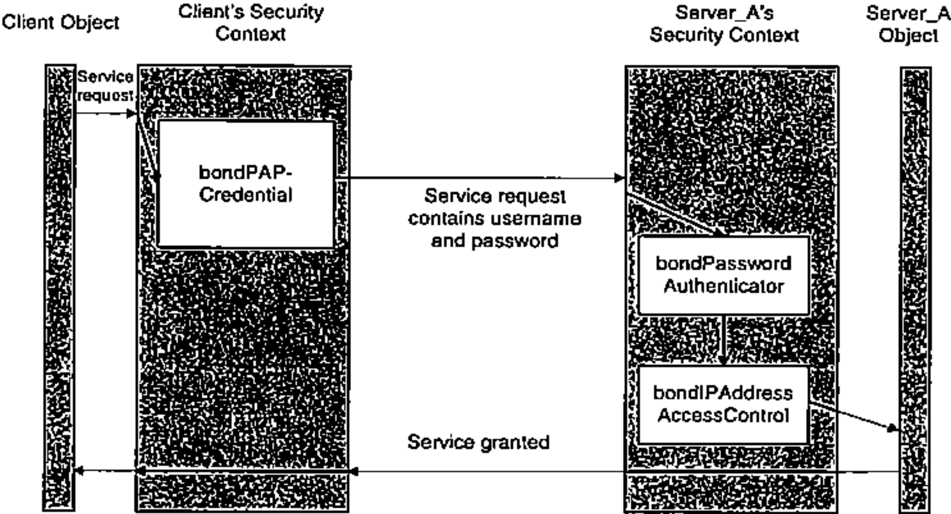

Figure 11. Processing of a service request.

The scenario illustrated in Figure 11 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 12.

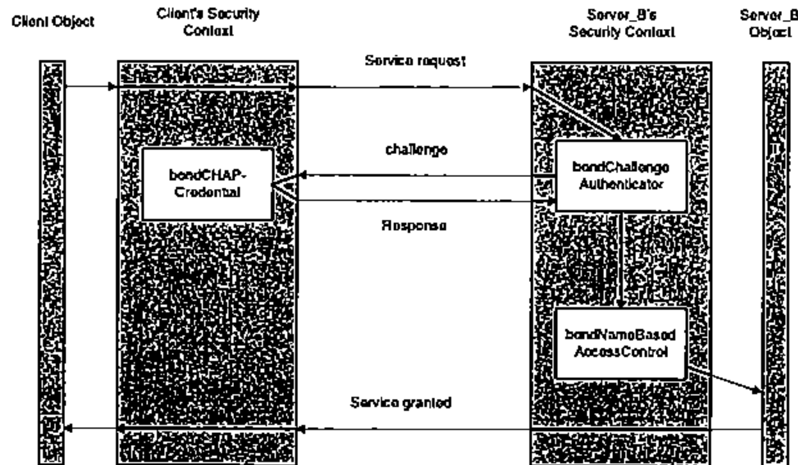

Figure 12. Processing of a service request using CHAP credential.

The client sends a service request message to *server_B*. 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 *server_B*. 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.

7. Summary

In this paper we present Bond Security Framework. We opted for an extensible core object that can support multiple security models and can be added dynamically to existing object. We decided 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. 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, http://bond.cs.purdue.edu.

8. Acknowledgments

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.

9. Literature

1. I. Foster and C. Kesselman. *The Globus Project: A Status Report*. In Proc. IPPS/SPDP'98 Heterogeneous Computing Workshop, pp 4-18, 1998.
2. J, B. Knudsen. Java Cryptography. O'Reilly, 1998.
3. Bruce Schneier. Applied Cryptography, Second Edition. John Wiley & Sons, Inc., 1996.
4. M.Lewis and A. Grimshaw. The Core Legion Object Model. In Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing, IEEE Computer Society Press, Los Alamitos, California, August 1996.
5. W.A. Wulf, C.X. Wang, D. Kienzle. A New Model of Security for Distributed Systems. Computer Science Technical Report CS-95-34, University of Virginia.
6. L. Bölöni, R. Hao, K.K. Jun, and D.C. Marinescu, *Structural Biolgy Metaphors Applied to the Design of a Distributed Object System*, November 1998, (submitted).
7. L. Bölöni, *Bond Objects -- a white paper*, Department of Computer Sciences, Purdue University CSD-TR #98-002
8. L. Boloni, R. Hao, K. Jun, and D. C. Marinescu. *Subprotocols: an object oriented solution to semantic understanding of messages in a distributed object system* (submitted).
9. T. Finin, et al. *Specification of the KQML Agent-Communication Language*, DARPA Knowledge Sharing Initiative draft, June 1993.
10. K. Jun, L. Boloni, R. Hao, and D. C. Marinescu, *Bond system monitor,* Technical Report CSD-TR #98-026, Purdue University, 1998.
11. A. D. Rubin, D. Geer and M. J. Ranum, Web Security Sourcebook. John Wiley & Sons, Inc., 1997.
12. R. Hao, K. Jun, and D.C. Marinescu, *Bond System Security And Access Control Model,* IASTED International Conference on Parallel and Distributed Computer and Networks, ParkRoyal Brisbane, Brinbane, Australia, December 14-16, 1998.