Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1998

Bond System Security and Access Control Models

Buibing Hao

Ladislau Bölöni

Dan C. Marinescu

Report Number:

98-019

Hao, Buibing; Bölöni, Ladislau; and Marinescu, Dan C., "Bond System Security and Access Control Models" (1998). *Department of Computer Science Technical Reports.* Paper 1408.
https://docs.lib.purdue.edu/cstech/1408

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.

BOND SYSTEM SECURITY AND ACCESS CONTROL MODELS

Ruibing Hao
Landislau Boloni
Dan C. Marinescu

Purdue University
Department of Computer Science
West Lafayette, IN 47907

CSD-TR #98-019
June 1998

Bond System Security and Access Control Models

Ruibing Hao, Ladislau Bölöni and Dan C. Marinescu

(hao, boloni, dcm@cs.purdue.edu)

Computer Sciences Department, Purdue University

West Lafayette, IN, 47907, USA

Abstract

Bond is a message-oriented middleware for network computing on a grid of autonomous nodes. The Bond environment consists of (a) a distributed object infrastructure, (b) a set of servers, (c) a set of agents, and (d) a set of user objects. In this paper we overview the basic architecture of the system and introduce the security models implemented in the Bond system. User programs as well as data are subject to the security constraints imposed by the local operating system. Bond objects are subject to the access control and the authentication model discussed in this paper. Secure Bond objects are allowed to create their own security agents to implement access control.

Contents:

1. Abbreviations and terms.
2. Introduction to Bond security.
3. Access control.
4. Authentication in Bond; the authentication and software distribution server.
5. Implementation Issues.
6. Conclusions.
7. Literature.

Abbreviations and terms.

A&SD Server - Authentication and Software Distribution Server.
DSVN - Directory Server Virtual Network.
IP - Internet Address of a host.

BID - Bond Id. String uniquely identifying a Bond object.
BVN - Bond Version Number. A string identifying the version of the Bond system software. Stamped on each jar file at the software distribution time.

BUN - Bond User Name. String uniquely identifying a Bond user.
BP - Bond Password.
<Kpub,Kprv> - Pair of seed keys (public and private) assigned to a copy of the Bond system software at the software distribution time.
<SKpub, SKprv> - Pair of object keys (public and private) assigned to a secure Bond object e.g. a workspace.

BSD - Bond Software Distribution object. Vector consisting of four-tupes <IP,Kpub,Kprv,BVN>.
BK - Bond Keys object. Vector consisting of triplets <BID, Kpub, Kprv)one for each secure Bond object.
BU - Bond Users object. Vector consisting of tuples <BUN, BP>.
Secure Bond Object - one with object keys.
Safe Bond Object - a secure Bond object with a security agent.

Introduction to Bond security

Bond is a message-oriented middleware for network computing. When completed, the Bond environment will consist of (a) a distributed object infrastructure, (b) a set of servers, (c) a set of agents, and (d) a set of user objects. A survey of the Bond architecture is presented in [1-4] and reviewed briefly below.

Servers are permanent, their life time is rather long and provide system-wide services e.g. the Bond System Monitor, BSM, Directory Servers, DS, Bond Persistent Storage Server, BPSS, Authentication and Software Distribution Server, A&SD, and so on. Agents are started upon request, work in conjecture with one or more objects, and disappear after completing their function. Examples of agents are Scheduling Agents and Security Agents[1],[4].

Bond system and user objects are persistent and their storage is the responsibility of the BPSS. Each object is uniquely identified by a Bond Id created by concatenating the IP address of the local host with the local time when the object was created. Access to objects can be gained by means of a DS. An important class of Bond objects is executables, e.g. residents and workspaces. A Bond executable consists of a Bond directory, a main thread of control, and a messaging thread with two mailboxes an ***in*** box and an **out** box. Once an object is created it is registered with the local directory. Objects located at different sites communicate with one another as shown in Figure 1. The sender needs to locate the ***in*** box of the executable where an active object is located, or must send it to the ***in*** box of Bond Persistent Storage Server if the object is inactive.

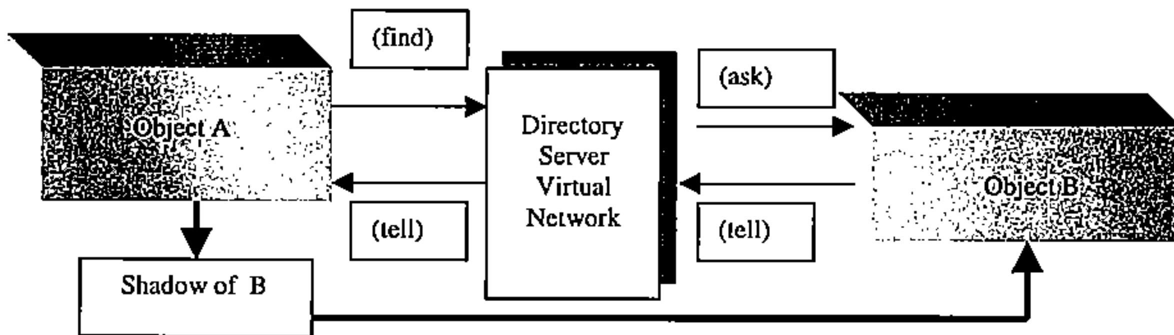

Figure 1. The use of Bond Shadows for inter-object communication. To locate a remote object **B**, object **A** sends a (**find**) performative to the Directory Server is attached to, which in turn multicasts the request to all the members of the Directory Server Virtual Network. Eventually one of the Directory Servers locates object **B**. Object **B** then sends the information necessary to create the Bond Shadow. From this instance on, all messages from **A** to **B** are sent to the shadow of B which acts as a local proxy for the remote object.

In Bond we have defined a light-weight communication abstraction called *the shadow* of an object to support unidirectional communication. The shadow of an object acts as a local proxy. A Bond communication act consists of three steps:

1) Send the message to the local object called Bond shadow, which will perform format conversion and security checks, and then deposit the message into the **out** box of the local messaging thread at the sender's site.
2) Transport the message from the **out** box of the sender to the **in** box of the receiver.
3) The local messaging thread at the destination will get the message in the **in** box of the receiver, parse and deliver the message to its destination object.

Another abstraction widely used in Bond is that of a *virtual network*. A group of objects functionally related to one another form a virtual network. For example the set of all servers supporting the Directory Service form the DSVN, Directory Server Virtual Network.

Each Bond object provides a set of services. These services are available to other objects by means of message patterns called Bond *sub-protocols*[6]. For example, an object reveals its properties and allows other objects to modify them by means of the *property access sub-protocol*. An executable understands the *monitoring sub-protocol* that allows another object to monitor its functions. A Directory Server understands the *directory service sub-protocol* and so on [7].

Each sub-protocol is implemented by exchanging KQML messages [5][6]. KQML is indifferent to the format of the information itself and KQML messages may contain information in the so called `context language'. The meaning of a KQML message is defined in terms of the constraints of the message sender and allows the message receiver to choose a course of action compatible with other aspects of its function.

KQML messages, called performatives allow to encode basic abstractions like asking, replying, achieving, subscribing or notifying while the content of the messages are partially encoded in the parameters, and partially assuming to be known by both parties of communication. So the most suitable approach to implement access control of Bond objects is to define and check the right to use every type of KQML message. Each KQML message in Bond consists of a *performative* field, a *content* field and one or more *parameter* fields. The parameter fields in a KQML message indicate the target of the destination object will operate on.

Bond uses a two-prong approach to system security [7-9]. Bond objects are subject to the security model discussed in this section while each Bond executable running on a host relies on the security features of the local system including password, access rights, quotas, etc. In this section we discuss authentication and access control policies for Bond objects.

The important design decisions for Bond security model are: the granularity, the scope, the policies and the mechanisms used by the model. Bond is an object-oriented system, therefore we decided to support object level security. We opted for a uniform model; the security model does not differentiate among system and user objects. We decided to allow each object to define and implement its own security, we only enforce a uniform interface to the security functions. An object may define its own security agent to implement its security policy. A group of objects may share a security agent.

The mechanisms used to enforce security cover authentication and access control policies. The creator of a Bond object has the option to enforce security. Only a secure Bond object may request authentication and access control. To

implement the access control model discussed in the next section we need to expand the Bond shadow to contain the access control object, called Bond ticket, for the destination object and the public key of the destination object.

Access Control

Bond is a Message Oriented Middleware, therefore we decided to embed access control mechanism into the Bond communication fabric. Bond objects act in response to messages and perform functions according to pre-defined message patterns called *sub-protocols.* The access control mechanism is based upon access rights granted by an object by means of a *Bond ticket* and enforced by a *Bond Security Agent.*

The access control mechanism requires an extension of the Bond shadow objects. Recall that a Bond object **B** sends messages for object **A** to the shadow of **A,** which in turn delivers the messages to **A** (see Figure 1). We now extend the shadow to contain a copy of the Bond ticket. If the shadow has a copy of the ticket granted by **A,** it can filter messages and deliver to **A** only those which conform to the access rights in the granted ticket.

A ticket defines the access rights as the ability to send KQML message with specific *performative*, *content* and parameters. A Bond object can grant the access to its services at three levels. The first allows the use of specific performatives, the second restricts the contents of the performative and the third further restricts the parameters. For example the ticket granted by **B** to **A** may have the following format:

performatives	contents	parameter
tell	-	-
notify	-	-
ask-one	**get**	-
achieve	**set**	**alpha**

In this example **B** can only send to **A** the following types of KQML messages:

1. any (**tell**) and (**notify**);
2. (**ask-one;get**); this message allows **B** to exercise the property access sub-protocol, and can get the value of any property of **A**.
3. (**achive;set;alpha**); this message allows **B** to exercise the property access sub-protocol, and modify (**set**) the value of property **alpha** of object **A**.

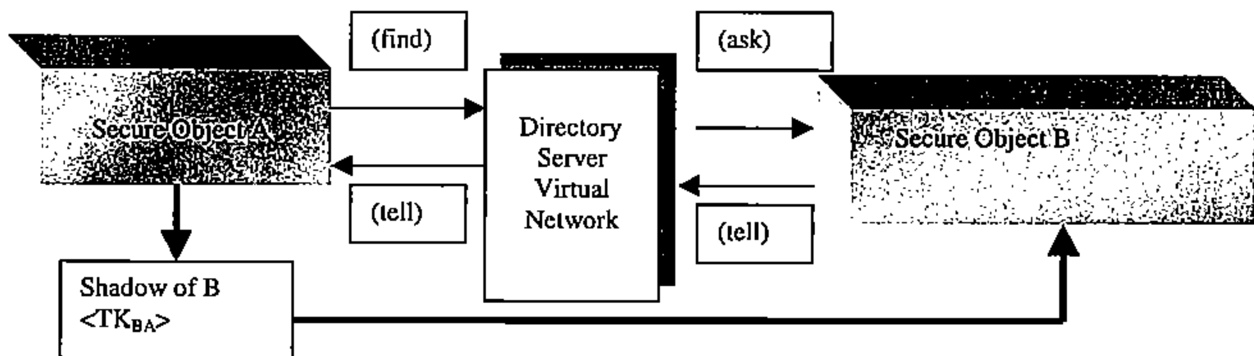

Figure 2. The access control ticket is generated at the time when the Bond shadow is created. Object **A** establishes a communication channel to **B**. **A** sends a request, **(find)** to the Directory Server Virtual Network which locates **B** and requests information necessary to establish a shadow, **(ask)**. Object **B** generates the ticket granted to **A**, TK_{BA}, and provides the information to create the shadow, **(tell)**. The shadow filters all messages from A to B according to the access rights in the ticket.

The scheme described above works well if the grantor of a ticket trusts the grantee. Filtering messages at the source guarantees that no unwanted messages cross the network. But we need to provide also a scheme that goes beyond the trust relationship and enforces the access control scheme. Such a scheme is based upon security agents. A security agent is one capable to (a) generate and store Bond tickets for one or more Bond objects and (b) enforce the access control. Instead of sending a message to the destination object its shadow will send the message to the security agent which acts as a proxy at the receiving end and enforces the access control specified by the ticket.

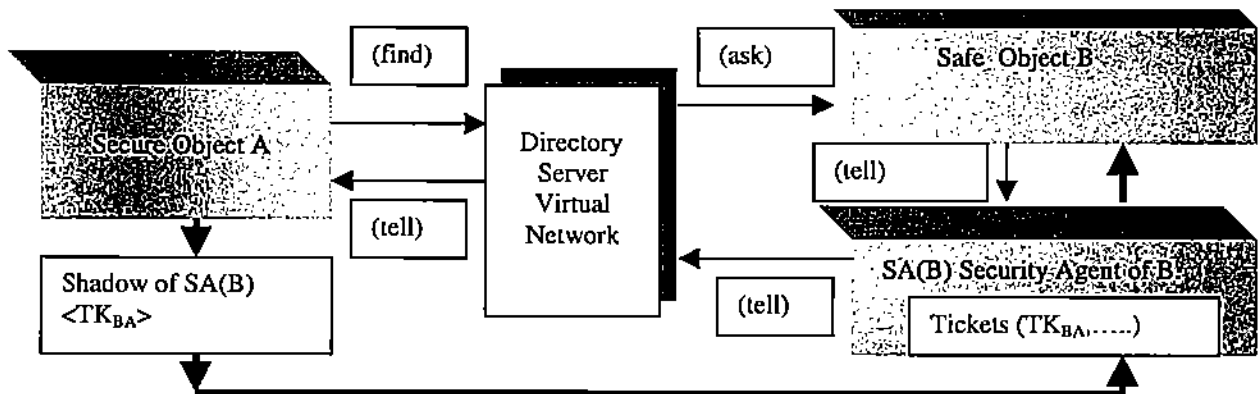

Figure 3. Access control for a safe Bond Object **B**. Object **A** establishes a communication channel to a safe object **B** following a procedure similar to the one in Figures 1 and 2. But now the shadow of the security agent of **B** is the local proxy rather than the shadow of **B.** Messages are filtered at the source by the shadow object and at the destination by the security agent.

Figures 2 and 3 illustrate the generation of the access control tickets and the communication path for both trusted and enforced access control. The scheme which enforces the access control adds additional overhead and is less performance but more secure.

Authentication in Bond, the authentication and software distribution server.

The A&SD server provides persistent storage for security critical objects: the Bond User object, the Bond Key object, and the Bond Software Distribution object. BU is a vector consisting of Bond User Name and Bond Password pairs, <BUN,BP>. A BK object consists of triples, Bond Id, public and private keys, <BID, Kpub, Kprv> for all secure Bond objects. The BSD object consists of tupes, IP address, public and private seed keys and the Bond Version Number, <IP, Kpub, Kprv, BVN>.

The Authentication and Software Distribution server is responsible for:

(a) Downloading on request of the latest version of the Bond system software.
(b) Creation and maintenance of Bond accounts.
(c) Creation and maintenance of Bond security keys.
(d) Authentication.

Each of these functions is discussed separately now. Figure 4 illustrates the process of downloading the current version of the Bond system code. The code includes an initiator for Bond Workspace objects and a Bond user has the option to initiate a secure workspace on the target host. The seed security keys are shared among all Bond users who start Bond sessions from the same host.

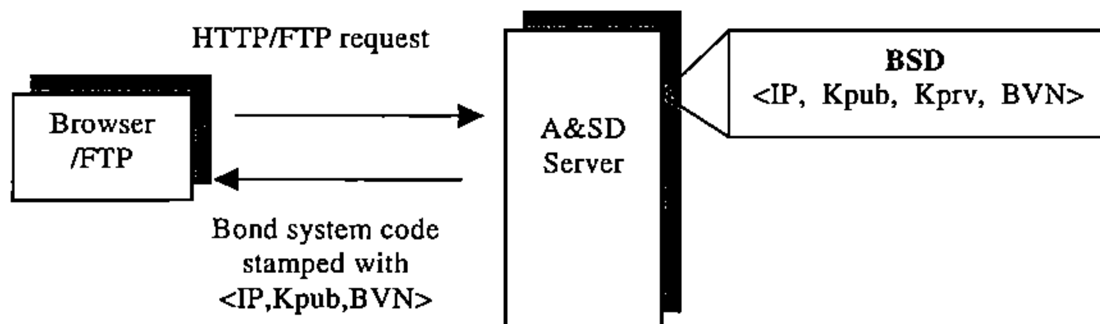

Figure 4. Downloading the Bond system code. The A&SD server provides HTTP and FTP access. The code is stamped with the IP address of the requester, the public seed key and the Bond Version Number. The Bond Software Distribution object is updated with a new entry including the IP address, the private and public seed keys and the Bond Version Number of the downloaded code.

The procedure to create Bond accounts is shown in Figure 5. First, a user establishes an HTTP connection with the A&SD server and fills out a form requesting a Bond account. Once a Bond account is established the Bond Users object is updated with a new entry containing the Bond User Name and the Bond Password for the new account.

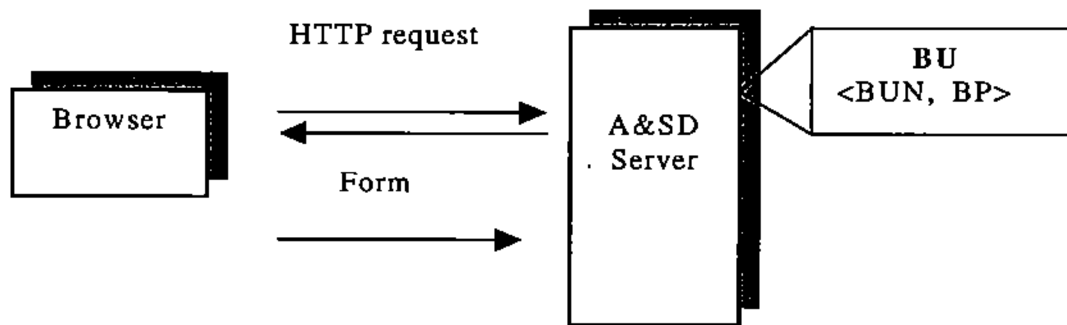

Figure 5. Establishing a new Bond account. The A&SD server provides a form and if the request is approved a Bond User Name and a Bond Password are generated and a new entry is added to the Bond Users object.

To start a Bond session a user follows the procedure illustrated in Figure 6. First a workspace is started, then a sign on request is sent to the A&SD server. The server uses the Bond User Name to find the Bond Password, decrypts the seed public key and validates the request. It also decrypts the Bond Version number and informs the user about the need to download the current version if the code is outdated.

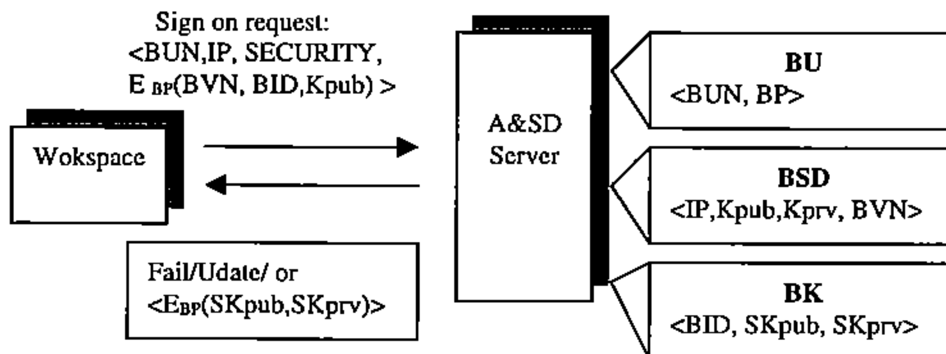

Figure 6. Starting a Bond session. The user starts a workspace that creates its unique Bond Id and inherits the Bond Version Number, the Public Key and the IP address from the downloaded Bond system software. Then the user provides the Bond User Name, the Bond Password and specifies if a secure workspace is desired. The workspace sends as plain text the Bond User Name, the IP address of the host, and the security switch as well as the Bond Version Number, the Bond Id of the workspace and the public key of the downloaded code encrypted with the Bond Password. If a secure workspace is desired then public and private session keys are generated and stored in the Bond Keys object indexed by the Bond Id. After a successful login a secure workspace gets the session keys encrypted with the Bond Password.

Another function of the A&SD server is to support object authentication. The basic idea for object authentication is illustrated in Figure 7. Assume that

both A and B are secure Bond objects. To authenticate object B, A generates a random message and asks B to encrypt it using its own private key (1) and send the encrypted message back, together with its own Bond Id (2). Then A sends the encrypted message to the authentication server (3) and it compares the message decrypted by the A&SD server (4) with the original random string it had generated.

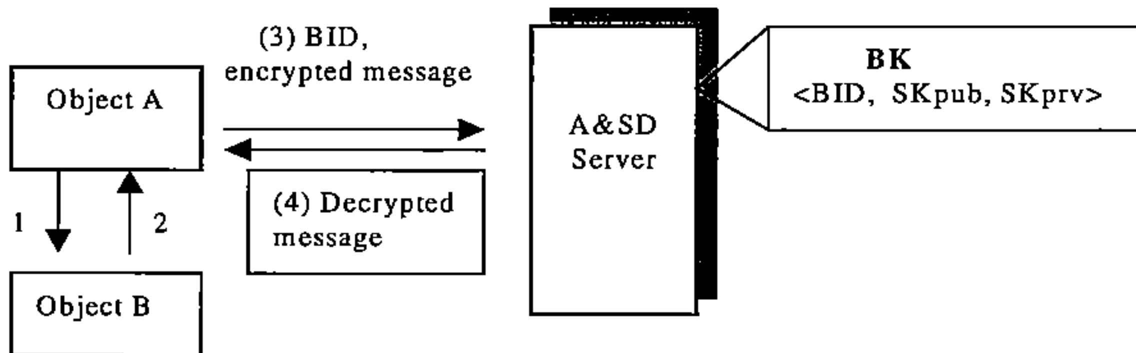

Figure 7. Object authentication in Bond.

Implementation issues

Each Bond object can choose its own security policy, it can vary from no security check to very rigorous security check. But interface of security check method if it is presented must be uniform. We can define Java abstract classes which for all the security e.g. security check and service ticket generation. Each object can make its decision to implement this security abstract class or not. If implemented, it can also override some methods by its own implementation. Bond also provides default security class implementations which user can choose if they do not want to implement their own. This design leaves the decision right of security to each object instead of using a uniform security policy in whole system.

We defined some abstract security classes in Bond to fulfill the requirements of what we had discussed in above sections. Following are several important abstract security classes.

(1) Abstract class for key pair generation

This abstract class is for the purpose of key pair generation. Different implementations can choose different key strength by changing the length of key or even choose key pair generation algorithms.

```
Abstract Class KeyPair {
        Byte[] PublicKey;
        Byte[] PrivateKey;
}

Abstract Class KeyPairGenerator {
   public void initialKeyPairGenerator()
   public KeyPair generateKeyPair();

}
```

(2) Abstract Class for Encryption/Decryption

Abstract Security class **Cipher** is a class which finishes the function of Encryption and Decryption. User objects can subclass **Cipher** to have their own implementation of Encryption and Decryption.

```
Abstract Class Cipher
{
      public static final int ENCRYPTION 0;
      public static final int DECRYPTION 1;

      public void initCipher(ENCRYPT_MODE, byte[] Key);
      public void addData(Sting);
      public void addData(byte[ ] );
      public String doFinal(String);
      public String doFinal(byte[ ] );
      public String doFinal();

}
```

(3) Abstract Class for Authentication

This abstract security class is for the purpose of authentication. It provides the method for digital signature and method to verify signature. User objects can subclass this abstract security class to provide their own authentication implementation.

```
Abstract Class Authenticator
{

      public void initSign(Cipher)
      public void initVerify(Cipher, String plaintext);
      public void addData(String );
      public void addData(Byte[ ]);
      public String sign( );
      public boolean verify( );

}
```

(4) Abstract Class for Access Control and Security Agent

Bond also defined several abstract security classes for the purpose of access control of Bond objects. Abstract security class **AccessTicket** defines the methods to check a Bond KQML message against the access rights. User objects can subclass Abstract Class **AccessTicket** to define their own access rights and access control.

```
Abstract Class AccessTicket
{
      collection AccessRights;
      public boolean checkAccessRight(bondKQML);
      public boolean checkAccessRight(String message);
}

Abstract Class SecurityAgent
{
      public AccessTicket generateAccessTicket(String);
}
```

Bond provides default implementations for all the defined abstract security class above. For example, Bond implements two ciphers sub-classes from the abstract security class **Cipher**: **bondSymmetricEncryptionCipher** and **bondAsymmtricEncryptionCipher**.

bondSymmetricEncryptionCipher is a Bond class for symmetric encryption and decryption using a single key. This single key can be a password, or a session key in DES. **bondAsymmetricEncryptionCipher** is a Bond class for asymmetric public key encryption and decryption using public/private key pairs. Such a key pair can be generated by the security class **bondKeyPairGenerator**.

Bond also provides implementations of abstract class **AccessTicket and SecurityAgent** called **bondAccessTicket** and **bondSecurityAgent** by using the definition of access right we proposed in this paper.

Conclusions

Security is an important concern in any network computing environment, as information in transit is more vulnerable, and use of resources in different administrative domain introduces issues of trust and consistency between them[8], [9]. Object-oriented network computing environments like Bond are generally more complex than traditional client-server systems, and the security issues are more subtle.

In this paper we discussed Bond's view of security in an object-oriented network computing environment and presented the security model used by Bond for access control. In this model, each Bond object can choose its own security policy, it can vary from no security check to very rigorous security check. Secure Bond objects are allowed to create their own security agents to enforce access control. These functions are supported by a set of uniquely defined abstract security classes. We believe this model is conceptually elegant and flexible.

Literature

1. L. Bölöni, K.K. Jun, M. Sirbu and D.C. Marinescu, *Seamless Metacomputing with Bond*, Purdue University CSD-TR #98-010.
2. L. Bölöni, *Bond Objects -- a white paper*, Department of Computer Sciences, Purdue University CSD-TR #98-002.
3. L. Bölöni, K.K. Jun, T. Daniels and D.C. Marinescu, *Message patterns in the Bond Distributed Object System*, Department of Computer Sciences, Purdue University CSD-TR #98-004.
4. L. Bölöni, R.B. Hao, K.K. Jun and D. C. Marinescu, *Bond Sub-protocols*, 1998, (in preparation).
5. T. Finin, et al. *Specification of the KQML Agent-Communication Language*, DARPA Knowledge Sharing Initiative draft, June 1993.
6. Y. Labrou, T. Finin, *A Proposal for a new KQML Specification*, UMBC TR-CS-97-03.
7. B. Fairthorne, *OMG White Paper on Security*, OMG Security Working Group, April 1994.
8. 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.
9. J, B. Knudsen, *Java Cryptography*, O'Reilly, 1998.