Securing a Java P2P framework: The JXTA-Overlay case

Joan Arnedo-Moreno
Estudis d'Informàtica,
Multimèdia i Telecomunicació
Universitat Oberta de
Catalunya (UOC)
Rambla de Poblenou, 156
08018 Barcelona, Spain
Rambla de Poblenou, 156
08018 Barcelona, Spain
jarnedo@uoc.edu

Keita Matsuo
Graduate School of
Engineering
Fukuoka Institute of
Technology (FIT)
3-30-1 Wajiro-Higashi,
Higashi-Ku, 811-0295
Fukuoka, Japan
bd07002@bene.fit.ac.jp

Leonard Barolli
Department of Information and
Communication Engineering
Fukuoka Institute of
Technology (FIT)
3-30-1 Wajiro-Higashi,
Higashi-Ku, 811-0295
Fukuoka, Japan
barolli@bene.fit.ac.jp

Fatos Xhafa
Department of Languages and
Informatics Systems
Technical University of
Catalonia (UPC)
Jordi Girona 1-3, 08034
Barcelona, Spain
fatos@lsi.upc.edu

ABSTRACT

In the wake of the success of Peer-to-Peer (P2P) networking, security has arisen as one of its main concerns, becoming a key issue when evaluating a P2P system. Unfortunately, some systems' design focus targeted issues such as scalability or overall performance, but not security. As a result, security mechanisms must be provided at a later stage, after the system has already been designed and partially (or even fully) implemented, which may prove a cumbersome proposition. This work exposes how a security layer was provided under such circumstances for a specific Java based P2P framework: JXTA-Overlay. [1]

Keywords

JXTA, Java, Security, XMLdsig, XMLenc, Peer-to-Peer

1. INTRODUCTION

Just as the popularity of P2P applications has risen, concerns regarding their security have also increased. As P2P applications move from simple data sharing, for example BitTorrent [6], to a broader spectrum, such as e-learning [8], they become more and more sensitive to security threats. Therefore it becomes very important to design P2P architectures taking into account an acceptable security baseline. To achieve this end, it must be taken into account that, since security is an ubiquitous matter within a system's architecture, once the system's design is over, providing security simply as an add-on may prove a difficult and time consuming task. Unfortunately, some systems did not take security into account during the design process, and as a result, there is no option but adding security at a later stage. Under that circumstance, a compromise must be reached so a security baseline is provided without redesigning and implementing the system from scratch.

The JXTA [12] (or "juxtapose") specification provides a set of open protocols that enable the creation and deployment of P2P networks, enabling P2P applications to discover and observe peers, communicate and offer and access resources within the network. Such protocols are generic enough so they are not bound to a narrow application scope, but are adaptable to a large set of application types. For that reason, they also keep implementation independence, so they can be deployed under any programming language or set of transport protocols. Currently, the most advanced reference implementation of such protocols is the Java one [1].

JXTA-Overlay [15] is a JXTA-based framework, which enables the creation of JXTA-based end-user Java applications in an efficient and effective manner. Its main goal is to improve the original JXTA protocols, increasing the reliability of JXTA-based distributed applications and supporting group management and file sharing. However, the design focus on JXTA-Overlay was completely concerned with improving system performance, but not security, a situation which may become a great constraint under today's stan-

[1]This work is partially supported by the Spanish Ministry of Science and Innovation and the FEDER funds under the grants TSI2007-65406-C03-03 E-AEGIS and CONSOLIDER CSD2007-00004 ARES.

dards. Even though the JXTA Java implementation provides some basic security mechanisms, they were not taken into account and none are currently used.

In this paper we present our work to provide a basic security layer to JXTA-Overlay. Our main contribution is proposing an approach which is transparent to the JXTA-Overlay libraries, requiring few modifications on the base code and keeping class coupling low. Furthermore, the security layer is extensible and can be easily adapted to different cryptographic modules, according to the end-user application's security requirements. Finally, minimum effort is necessary by end-user application developers to provide a secure environment, providing an abstraction layer, thus maintaining the philosophy of JXTA-Overlay.

This paper is organized as follows. Section 2 provides a general overview of JXTA and JXTA-Overlay's architecture and capabilities, as well as some insights on the current state of security of JXTA-Overlay and why providing security services is deemed important. The proposal of a basic security framework is presented in Section 3., detailing the security extensions and how they were integrated into the current system. Concluding the paper, section 4 summarizes the main contributions and further work.

2. JXTA-OVERLAY OVERVIEW

JXTA-Overlay is a P2P middleware built on top of the JXTA Java reference implementation, taking advantage of its set of libraries standardizing how different devices may communicate and collaborate. JXTA-Overlay extends the JXTA reference implementation with the goal of overcoming some of its limitations: the need for the developer to explicitly manage the presence mechanism, peer group publication and message exchange. To achieve this end, the JXTA-Overlay libraries provide a set of basic functionalities, named *primitives* and *functions*, intended to be as complete as possible to satisfy the needs of most JXTA-based applications. Therefore, custom Java P2P applications may be built on top of the JXTA-Overlay libraries by using the set of primitives and functions as an Application Programming Interface (API).

2.1 The JXTA-Overlay network

In a JXTA-Overlay network, the main interacting entities are:

End-users connect to the JXTA-Overlay network by previously authenticating via a username and password. Once the authentication process is successfully completed, they are organized into different overlapping groups, so only members of the same group may interact. This authentication process is the only security mechanism currently deployed in JXTA-Overlay.

A *client peer* represents a custom application built on top of the JXTA-Overlay libraries, which end-users use to communicate and share resources between them. Client peers effectively act as end-user proxies within the JXTA-Overlay network, forwarding end-user data to client peers that belong to other end-users of the same group. A client peer is assumed to belong to the same groups as its current end-user.

The *brokers'* main task is access control, requesting end-user authentication, and helping client peers interact between themselves by propagating their related information. Brokers also act as beacons used by client peers which have recently gone online to join the network. In contrast with client peers, brokers are not custom made applications, but fully functional peers provided by the JXTA-Overlay libraries.

All the information related to user configuration (username, password and group membership) is stored in a *central database*. Only brokers may access the database content, in order to verify end-user authentication attempts and organize them into groups. It is assumed that some *administrator* takes care of properly manage the database.

2.2 General architecture

The architecture of the JXTA-Overlay framework defines three distinct layers, which let the different entities described in section 2.1 communicate: the client layer, the broker layer and the control layer. Together, they form an abstraction layer on top of JXTA, as shown in Figure 1. The JXTA packages are divided in those related to the generic protocol specification (`net.jxta`) and those implementation specific (`net.jxta.implem`).

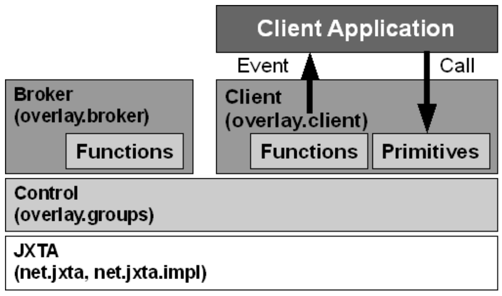

Figure 1: JXTA-Overlay architecture.

- The *client layer* (package `overlay.client`) defines all necessary primitives and functions for peer clients to join a JXTA-Overlay network and interact with other peers and brokers. Primitives serve as entry points to execute tasks and send messages across the network, whereas functions process incoming messages and generate events which may be captured by the application, using a Model-View-Controller pattern approach. Five groups of primitives and functions exist depending on the kind of tasks processed, each group defined in a different Java interface: Discover, Executable, Files, Learning and Messenger.

  Primitives are defined in the `overlay.client.primitives` package, whereas events are defined in the `overlay.client.core.functions` package. There is a total amount of 122 primitives and 84 possible events.

- The *broker layer* (package `overlay.broker`) defines all the functions that client peers may call upon a broker in order to be granted access to the the network, create and publish groups or retrieve some other client peers' status information. After a broker function has been executed, it always produces a reply from the broker to the calling client. A fully functional broker peer is already developed in the `overlay.broker.Broker` class.

- The *control layer* (packages `overlay.groups` and `overlay.utils`) acts as an intermediate layer between the broker and client layers, providing the generic functionalities on regards to group management and messaging. All interaction with JXTA libraries is performed at this layer. Developers never directly interact with this layer's classes.

It must be remarked that JXTA-Overlay does not provide any full client peer, apart from some demo applications that may be used as all purpose clients or code examples (the `demoApplication.Client` and `demoApplication.SimpleClient` packages). Only brokers are provided as a fully developed application.

2.3 Message exchange

JXTA-Overlay uses JXTA *pipes* at the control layer to exchange messages between any peer type (both brokers and client peers). A JXTA pipe acts a a virtual communication channel between peer endpoints, configurable to different underlying transport protocols. Any transport capable of unidirectional asynchronous unreliable communication can be used, however, JXTA-Overlay exclusively relies on TCP unicast transport. Client peers have an input pipe for each group it belongs to, so other group members may send messages by specifically using the input pipe associated to that group. Brokers have a single input pipe which is shared for all incoming messaging.

Before any message may be sent to any peer's input pipe, its *pipe advertisement* must be previously located and retrieved. The JXTA specification defines several types of advertisements, metadata XML documents used to distribute resource information between peers. Periodically, the control layer at each peer republishes pipe advertisements, so newcomers to the network are able to receive them, or network changes may be properly updated (for example, peer disconnections).

A pipe advertisement contains four basic fields. The *Id* and *Type* are mandatory fields used to define the advertisement unique identifier, and the message transport pipe type. The *Name* and *Desc* are optional fields, but JXTA-Overlay uses them in order to define which client peer is the advertisement owner, by including the Peer ID in the *Name* field, and which end user is connected to that client peer, by including the username. In this manner, JXTA-Overlay may easily search for some pipe advertisement by username and send messages to the associated pipe. Since all messaging capabilities ultimately rely on each peer's input pipes, pipe advertisements are very important within the JXTA-Overlay network.

2.4 JXTA-Overlay and security

As previously exposed, JXTA-Overlay's design was not concerned with security, with the only exception of end-user network access control via a username and password. As a result, it is unable to provide any degree of security to its end-users as well as vulnerable to different security threats which may jeopardize the network. We must take into consideration the fact that not only entities external to the JXTA-Overlay network may try to subvert it, but also malicious legitimate users.

Some of the greatest security concerns in JXTA-Overlay are the following ones:

- Transmitted data may be easily eavesdropped, since no data privacy is provided. This is specially critical in those cases when sensitive data is being exchanged, such as sending the username and password in the initial authentication to a broker. Both fields are currently sent in plain text. Therefore, intermediate peers or any device at broadcast range may read this information with a network protocol analyzer (such as Wireshark [7]).
- Client peers connect to a self-proclaimed broker, but never check if it is a legitimate one. Even in the case that client peers are connecting to the proper broker address, there are no guarantees that the broker is a legitimate one, since it is possible that traffic is being redirected to a fake one via methods such as DNS spoofing [10].
- A malicious client peer may forge advertisements with no fear of reprisal. No integrity or source authenticity is enforced. False fields, such as the source client peer identifier or username may be used on advertisements, which will be automatically distributed by the broker and accepted by all group members, unaware of the false data they contain.

As can be seen, some of the current JXTA-Overlay vulnerabilities are quite obvious ones, such as transmitting sensitive data with no real privacy. Therefore, it would be very interesting to extend the system by providing a security layer.

3. SECURITY EXTENSION FOR JXTA AND JXTA-OVERLAY

In this section we present an extended version of JXTA-Overlay which provides a baseline for protecting end-user applications against the current vulnerabilities exposed in section 2.4. In out proposal, we combine several approaches, adapting them to JXTA-Overlay's specific architecture and network setup. Client peers are protected against impersonation by extending the `Discover` primitives and functions with broker authentication. In addition, a secure login primitive is provided to protect the username and password from eavesdroppers. Finally, we deploy a basic method for key distribution between group members, once they have been granted access to the JXTA-Overlay network by the broker. From this point, security capabilities may be easily added to JXTA-Overlay by extending the system with additional secure primitives and functions.

Our main goal is extending JXTA-Overlay with the least amount of modifications to the base libraries and always following its base architecture design, providing a security layer operating in the most transparent way to a client peer developer. Even though some changes in the source code are unavoidable, we take special care in concentrating them in very specific, easily manageable, places, as well as keeping class coupling low.

3.1 JXTA's security mechanisms

Before a security layer extension for JXTA-Overlay may be proposed, it is useful to review which are the current security mechanisms available to JXTA-based applications. From this review, it is possible to study which may prove useful or suitable to JXTA-Overlay's architecture and network setup specifics. In this section, only some of the more important highlights to our particular case, securing JXTA-Overlay, will be provided. A more detailed analysis is provided in [3].

On regards to message security via pipes, the JXTA reference implementation provides two mechanisms: TLS [14] (Transport Layer Security) and CBJX [4] (Crypto-Based JXTA Transfer). The former provides private, mutually authenticated, reliable streaming communications, whereas the latter provides lightweight secure message source verification (but not privacy).

JXTA provides its own definition of standard TLS as a transport protocol. The JXTA definition of TLS is composed of two subprotocols: the TLS Record Protocol and the TLS Handshake Protocol. The TLS Record Protocol provides connection security using symmetric cryptography for data encryption. The keys for this symmetric encryption are generated uniquely for each connection and are based on a secret negotiated by the TLS Handshake Protocol. In addition,the connection is reliable by including message integrity check using a keyed MAC.

On the other hand, CBJX is a JXTA-specific security layer which pre-processes messages to provide an additional secure encapsulation, creating a new message that is then relayed to an underlying transport protocol. The original message's is signed, and an additional information block, is also added to the secured message. This information block contains the source peer credential, both the source and destination addresses, and the source peer ID.

The current JXTA reference implementation also provides a method to get some degree of advertisement security by providing the option to sign advertisements. No distinction between different types of advertisements is made, all become a new type of advertisement when signed: the Signed Advertisement. A Signed Advertisement encapsulates the original XML advertisement as plain text encoded via the Base64 algorithm [9].

Unfortunately, in order to use all the secure mechanisms in the JXTA reference implementation it is mandatory to use a specific implementation of the JXTA Memberhip Service, one of the JXTA core services which takes care of group membership and identity management: the *Personal Secure Environment* (PSE). None of the described security mechanisms may be applied without PSE. However, JXTA-Overlay does not rely on this membership service at all, using a simpler one (the *None* membership service). This is a great constraint, since the choice of membership service has strong implications during the design and implementation stages, being pervasive through the application's code. Therefore, it is difficult to use PSE in JXTA-Overlay without many changes. Furthermore, PSE has a limited range of cryptographic module support, solely supporting Java keystore files [13] and X509 certificates [5] as credentials.

3.2 Secure primitives

As presented in section 2.2, JXTA-Overlay defines an interface for each primitive group, specifying the expected behaviour for each primitive. In this manner, it is easy to integrate different primitive implementations. JXTA-Overlay provides one default implementation for each interface, which is enough for most scenarios, in the form of "`Impl`" classes (for example, the `DiscoverImpl` class, the `MessageImpl` class, etc.). They also belong to the `overlay.client` package. Anyway, applications call primitives only through the interfaces. We have expanded the original set of primitives with secure versions for some of them, providing the building blocks for a secure layer.

In order to keep class coupling low, we have defined a special class, the `SecureManager`, which serves as the main API to all security services at the security layer. All secure primitives solely use methods defined in this class to perform security-specific operations. The `SecureManager` uses a Singleton pattern to guarantee that a single entrypoint exists, as well as making it easy to locate the instance from any code location.

All the necessary operations dependant on the particular cryptographic module chosen by the client peer application developer are defined in the `CryptoManager` interface. The `SecureManager` relies on an implementation of this interface to complete any operation related to cryptographic data management and operations.

The main reason for the `CryptoManager` is acting as a proxy for any cryptographic module. It enables the integration of the security layer with any kind of cryptographic module such as hardware cryptographic tokens. This approach takes into account the fact that not all such modules are accessed via plain text passwords (for example, using biometrics or hardware tokens). Client peer application developers will implement the `CryptoManager` interface according to the needs of the specific cryptographic provider being used. The application developer must provide a proper implementation of this interface when initializing the `SecureManager` singleton.

We provide a default implementation, which is enough for most standard scenarios which do not rely on hardware modules, the `CryptoManagerOv` class. X509Certificates are used as credentials, and public key cryptography is assumed for cryptographic operations. The local client peer's key pair and all credentials are stored using volatile memory (specifically, a JCE `KeyStore` instance).

The base class diagram for this common API to the security layer is shown in Figure 2.

The `CredentialRequest` class is used as an intermediate data representation for credential requests, as will be explained in the following 3.2.2 subsection.

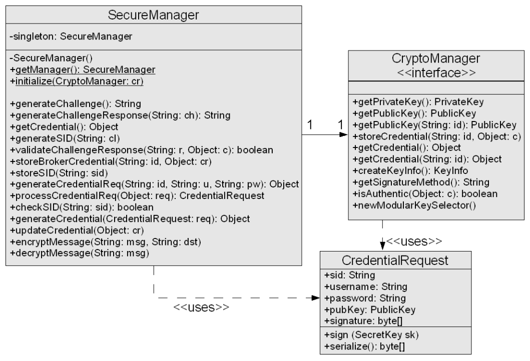
SecureManager
-singleton: SecureManager
-SecureManager()
+getManager(): SecureManager
+initialize(CryptoManager: cr)
+generateChallenge(): String
+generateChallengeResponse(String: ch): String
+getCredential(): Object
+generateSID(String: cl)
+validateChallengeResponse(String: r, Object: c): boolean
+storeBrokerCredential(String: id, Object: cr)
+storeSID(String: sid)
+generateCredentialReq(String: id, String: u, String: pw): Object
+processCredentialReq(Object: req): CredentialRequest
+checkSID(String: sid): boolean
+generateCredential(CredentialRequest: req): Object
+updateCredential(Object: cr)
+encryptMessage(String: msg, String: dst)
+decryptMessage(String: msg)
1
1
CryptoManager
<<interface>>
+getPrivateKey(): PrivateKey
+getPublicKey(): PublicKey
+getPublicKey(String: id): PublicKey
+storeCredential(String: id, Object: c)
+getCredential(): Object
+getCredential(String: id): Object
+createKeyInfo(): KeyInfo
+getSignatureMethod(): String
+isAuthentic(Object: c): boolean
+newModularKeySelector()
<<uses>>
CredentialRequest
+sid: String
+username: String
+password: String
+pubKey: PublicKey
+signature: byte[]
+sign (SecretKey sk)
+serialize(): byte[]
<<uses>>

Figure 2: Common security layer API class diagram.

Since securing every single primitive is beyond the space limitations of this work, in this section we will focus in two particular cases. Other secure primitives follow the same approach as the ones described here, making use of the `SecureManager` instance as necessary. First, we will describe the *connect* and *login* primitives, two very important ones related to discovery. These primitives search for an available broker and authenticate to it in order to join the network. They are important since, being the initial negotiations to join the JXTA-Overlay network, they allow to setup the base client peer credentials and cryptographic data. Then, we will describe two primitives related to instant messaging.

For the rest of this section, in each figure, classes shadowed in grey are those from our secure layer, whereas white colored ones are those from the original JXTA-Overlay or JXTA libraries.

3.2.1 *Secure connection*

Broker connection, via the `connect` primitive, is the first step before any client peer may try to join the JXTA-Overlay network. This non-secure method just locates a broker. Once one has been found, the client peer waits until a connection becomes actually available using the `waitRdv` primitive. No message exchange happens between the client peer and the broker at this stage.

The `authBroker` primitive is the secure version of `connect`. However, in this case, a two-way message exchange is produced in order to authenticate the broker and setup some initial cryptographic data exchange. The resulting message executes the `authRequest` function at the broker side, and the final broker response executes the `authResponse` function at the original client. At this stage, the client is ready to try to join the network.

The sequence diagrams for `authBroker`, `authRequest` and `authResponse` are shown in Figures 3, 4 and 5 respectively. The `GroupManager` class is the one responsible for sending messages across the network. All interactions with the secure layer are done exclusively through the `SecureManager`.

During this process, the client peer locates a broker and waits for a connection to open. Then, authenticates the broker by initiating a challenge-response protocol [11] (Figure 3).

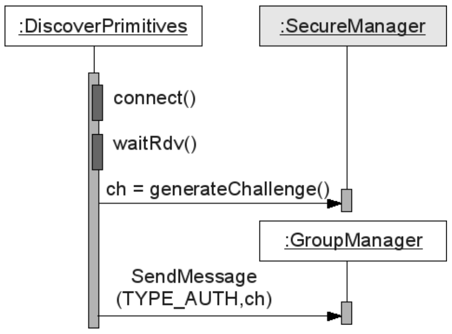

Figure 3: (Client) Authentication sequence diagram: `authBroker` primitive.

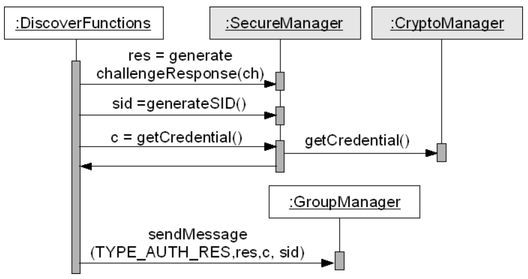

Figure 4: (Broker) Authentication sequence diagram: `authRequest` function.

The broker, responds to the challenge by signing the data and sending its own credentail. A sufficiently long (128-160 bits) session identifier SID is also generated and sent (Figure 4). Finally, the client peer verifies the challenge signature and the credential. If the signature is correct and the credentail valid, it can be assumed that a connection to a legitimate broker has been achieved. Both the SID and the broker's credential are locally stored in the client peer (Figure 5)

3.2.2 *Secure login*

Once a connection to a broker has been established, the client peer may try to authenticate via the `login` primitive, sending its end-user username and password to the broker to be validated against JXTA-Overlay's database. The message sent to the broker executes the `login` function at the broker side. Depending on the username and password check result, a response is sent back to the client, executing either the `loginOk` or `loginError` function. The `login` primitive is usually called as soon as the `connect` primitive finish event is generated.

The secure version of this primitive is named `secureLogin`. It maintains the message exchange, but signs and encrypts the username and password sent to the broker, together with the session identifier, SID, obtained from the previous `authBroker` primitive call. The message is encrypted using the broker's public key, retrieved during broker authentication, when its credential was retrieved. The session identifier and the signature are used to avoid replay attacks, since just encryption does not protect against them. Furthermore, a

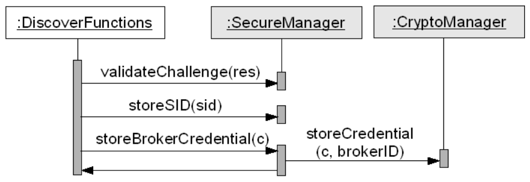

Figure 5: (Client) Authentication sequence diagram: `authResponse` function.

credential is requested to the broker, which may be used at a later stage to authenticate to other client peers and as a means to transport and publish public key information. All this information is processed at the broker-side function `secureLogin`.

The sequence diagrams for the `secureLogin` primitive and function are shown in Figures 6 and 7, respectively.

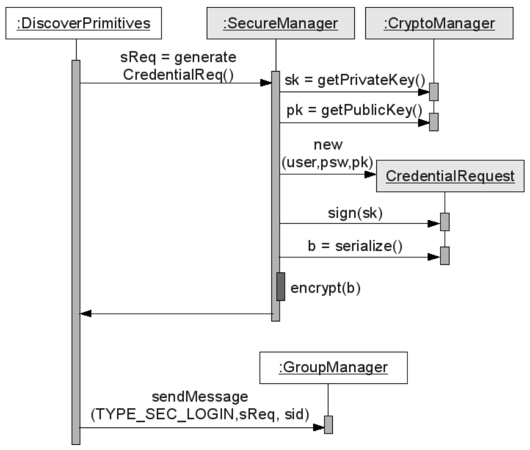

Figure 6: (Client) Login sequence diagram: `secureLogin` primitive.

During this protocol exchange, an instance of the `CredentialRequest` class is used to encapsulate all data related to the credential request, as well as the username and password. The actual format of this class is transparent to both the control layer and JXTA messaging, since it is serialized. This serialized form is then encrypted and Base64 encoded, thus becoming an ordinary `String` as far as message transport is concerned (see Figure 7).

3.2.3 *Secure instant messaging*

Once client peers are connected to the JXTA-Overlay network in a secure manner, possessing of a credential and the corresponding cryptographic keys, they are able to exchange secure messages. As a proof of concept, the instant messaging primitives, `sendMessage` and `sendGroupMessage`, have been extended with secure versions. The following primitives have been added:

- `secureSendMessage`: A simple text message is sent to another user. This secure version encrypts data using the destination's public key via a wrapped key approach.

  Once the data is encrypted, it is encoded back to a `String` using the Base64 algorithm, and then uses the original non-secure version to transmit the data.

- `secureSendGroupMessage`: This method just iteratively calls `secureSendMessage` to transmit encrypted data to all members of a group.

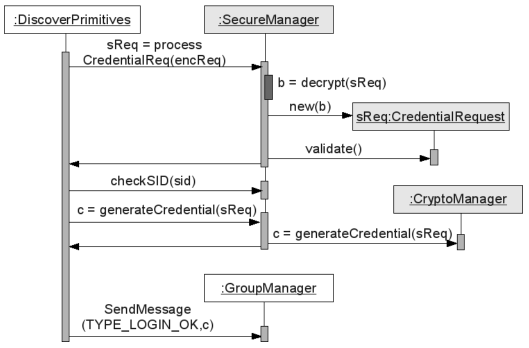

Figure 7: (Broker) Login sequence diagram: `secureLogin` function.

3.3 Advertisement security and key distribution

Client peers' credentials must be distributed to other group members, before it is effectively possible to secure data exchanges (for example, in messenger primitives), since they contain each client peer's public key. The credential distribution method must take into account the fact that in a P2P network nodes may go online and offline at any moment, as well as allowing credential updates in the case that, for some reason, a new one must be generated. To achieve this end, we apply the scheme defined in [2], based on XMLdsig, where the credential is included into an advertisement, piggybacking on JXTA-Overlay's pipe advertisements propagation mechanism. The reasons for this choice are twofold.

First of all, all JXTA-Overlay's messaging capabilities between group members rely on the input pipes, as explained in section 2.3, which can only be accessed by previously retrieving its associated advertisement. Therefore, client peers cannot exchange messages unless they have each others' pipe advertisement. Consequently, by publishing credentials using this advertisement type, it is always guaranteed that both parties have each others' public key before any message exchange begins. This approach also avoids relying on additional protocols for credential distribution.

Additionally, the scheme allows the signature of pipe advertisements, providing effective protection against advertisement forgery, as exposed in section 2.4. It must be heavily remarked that each client peer's input pipe is very important for its operation within the network. Once a broker has granted access to a client peer, absolutely all incoming messages are received via this pipe. Therefore, it is very important to secure the distribution of each client peer's pipe advertisement to avoid that a malicious peer may publish a forged advertisement, claiming that its own input pipe is assigned to some other peer identifier. In that scenario, all messages outbound to that peer identifier would be automatically redirected to the malicious peer.

A crucial advantage offered by this XMLdsig-based scheme, instead of JXTA's Signed Advertisement, apart from obviating the need for PSE, is its capability to become invisible to standard JXTA-Overlay operation, instead of adding a new advertisement type, completely opaque to advertisement indexing and retrieval services (all advertisement fields disappear). In this manner, JXTA-Overlay's standard mechanism of searching for some peer or end-user's pipe advertisement via the *Name* and *Desc* fields is not interfered.

3.3.1 Integration to JXTA classes

Pipe advertisements are instantiated at the control layer, managed by the `PipeOv` class, and propagated across the network at using the JXTA layer. Therefore, advertisement signature and credential encapsulation should be invisible to both layers to minimize their impact on the base libraries. We define secure advertisements as the `SignedPipeAdvertisement` class, an extension of JXTA's pipe advertisement.

JXTA advertisements are instantiated using a Factory design pattern, via the `AdvertisementFactory` class. We take advantage of this fact to define a secure advertisement instantiator, which is registered to the advertisement factory replacing the original one. Furthermore, a Proxy design pattern is applied on `SignedPipeAdvertisement`, so its behaviour and type is exactly the same as an original pipe advertisement. The general class diagram for secure advertisements using this strategy is shown in Figure 8. The `PipeAdvertisement` class is abstract, specifying the expected behaviour of a pipe advertisement (from the `jxta.net` package) whereas the `PipeAdv` class is the actual implementation (from the `jxta.net.impl` package). It is not possible to simply apply inheritance from the pipe advertisement implementation, since its constructors are private.

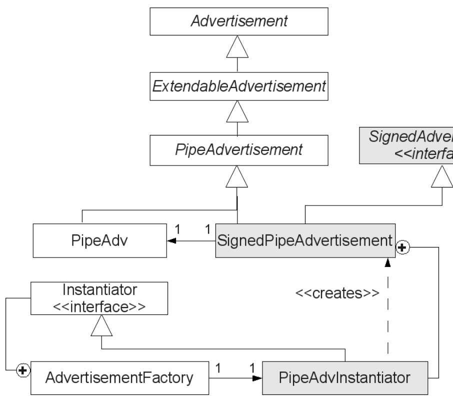

Figure 8: Signed pipe advertisement class diagram with JXTA classes.

As a result, new the advertisement type is invisible to both the JXTA and control layers, since they are still able to operate with `PipeAdvertisement` instances, ignoring the additional signature and credential encapsulation.

The class diagram with all classes related to secure advertisement generation is shown in Figure 9. Just like in the case for secure primitives in section 3.2, the `CryptoManager` instance is used as the main interface to cryptographic modules. In the case of secure advertisements, the `CryptoManager` also performs some extra tasks related to XML signature generation and processing.

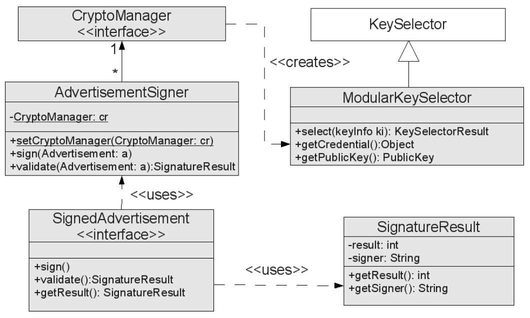
CryptoManager <<interface>>
1
*
KeySelector
<<creates>>
AdvertisementSigner
-CryptoManager: cr
+setCryptoManager(CryptoManager: cr)
+sign(Advertisement: a)
+validate(Advertisement: a):SignatureResult
ModularKeySelector
+select(keyInfo ki): KeySelectorResult
+getCredential():Object
+getPublicKey(): PublicKey
<<uses>>
SignedAdvertisement <<interface>>
+sign()
+validate():SignatureResult
+getResult(): SignatureResult
<<uses>>
SignatureResult
-result: int
-signer: String
+getResult(): int
+getSigner(): String

Figure 9: Secure pipe advertisement class diagram detail with related classes.

The class responsibilities are:

- `SignedAdvertisement`: Specifices the generic behaviour of any secure advertisement, so it is possible to extend security to different advertisement types at a later stage.
- `SignatureResult`: Encapsulates the signature validation result, defining a set of constants for different outcomes: `VALID`, `INVALID`, `ERROR`, `NO_SIGNATURE`, `NOT_AUTHENTIC`, `NOT_VALIDATED`, `MISSING_KEY`. The signer identifier as well as some additional information (such as the error cause, for an `ERROR` result) are also included.
- `AdvertisementSigner`: Manages all details related to XML signature generation and validation.
- `ModularKeySelector`: Retrieves the credential from an XML signature. The detalis on the purpose of this class will be shortly explained in the following subsection 3.3.2.

3.3.2 Advertisement signature generation and validation

Pipe advertisements may be instantiated at any time in the JXTA libraries using the `AdvertisementFactory`, via the `newAdvertisement` method. However, this method is overloaded to take into account two different possibilities. One of them uses no parameters and is always used by JXTA-Overlay's control layer to create a new advertisement, so it can be transmitted across the network. The other possibility is using an XML document as a parameter, so an advertisement may be instantiated from some received data. This method is exclusively used by the JXTA layer when advertisements are searched, processing incoming data. JXTA-based applications always operate with `PipeAdvertisement` instances. We use this fact to tie advertisement signature and credential encapsulation to the former `newAdvertisement` overload, and advertisement validation and credential retrieval to the latter one.

The sequence diagram for pipe advertisement generation and signing at the `PipeOv` class is shown in Figure 10. Since the signature must be generated after all advertisement fields have been properly initialized (using setter methods, `setXX`), the `sign` method is called at the control layer. It is the only modification to the original `PipeOv` class.

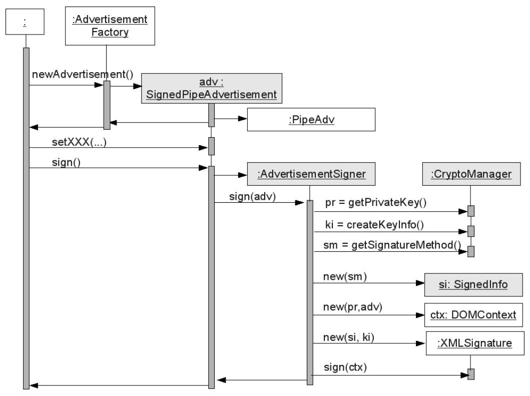

Figure 10: Signed pipe advertisement instantiation: new advertisement sequence diagram.

Since the `CryptoManager` arbitrates credentials and is the only class aware of the real credential's format, one of its tasks during the signature generation process is providing all the credential-dependant XML signature fields, via the `getSignatureMethod` and `createKeyInfo` methods.

The sequence diagram for pipe advertisement reception and validation is shown in Figure 11.

In contrast with signature generation, validation may be automatically executed upon advertisement instantiation from retrieved XML data. Therefore, the `validate` method is executed in the factory instantiator. The ensuing signature validation result remains available accessible using the `SignedAdvertisements`'s `getSignatureResult` method.

During signature validation, the encapsulated credential must be retrieved and stored within the `CryptoManager`, as well as included into the `SignatureResult` instance. However, the real credential format should be transparent to the `AdvertisementSigner` class. This goal is achieved with help of the `ModularKeySelector` class, a `KeySelector` implementation. The `KeySelector` interface is provided by the XML signature libraries, the `javax.xml.crypto.dsig` package, to retrieve the public key from the signature body via a callback to the `select` method, so the signature may be properly validated. An the implementation must be provided according to the expected XML signature format. Therefore, we use this expected behaviour during XML signature validation, storing the retrieved credential for later processing.

4. CONCLUSIONS

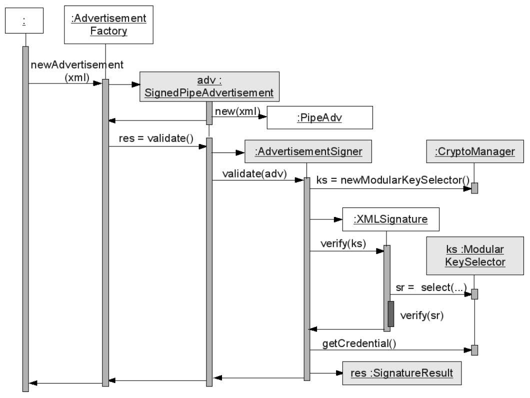

Figure 11: Signed pipe advertisement instantiation: retrieved advertisement sequence diagram.

An extension of the JXTA-Overlay libraries has been proposed and developed, in order to provide a basic secure layer to an already implemented framework. Even though the system design hardly took into account security mechanisms, it was possible to provide a baseline with minor modifications to the base code and keeping a low coupling between the original classes and the ones in the secure layer. Secure message exchanges are invisible to JXTA's underlying layer. The main contributions form this experience are twofold.

First of all, effective secure credential distribution is provided by judicious use of design patterns. Pipe advertisements are secured, using standard JXTA-Overlay and JXTA procedures for publication and update, guaranteeing that keys are always available whenever messages must be exchanged between peers. As a result key distribution becomes invisible to the control and JXTA layers and end-user application developers need not concern himself about how the key distribution mechanism works.

Finally, the proposed framework is completely modular and can be adapted to different scenarios (different types of credentials or cryptographic modules) suitable to the application developers' needs. This is also an improvement over the security mechanisms provided by JXTA, which tie end-user applications to a very specific credential and cryptographic module type. In our implementation, public/private keys are used, and credentials are issued in the form of X509 certificates, but the system accepts any cryptographic module and credential type in a modular way via different `CryptoManager` implementations. Once the basic blocks of the security layer have been deployed, it is easy to extend JXTA-Overlay's primitives and functions with secure alternatives.

Further work goes towards extending the security layer to additional primitives deemed sensitive to attacks, in a manner that they complement existing ones, but not forcibly replace them. Of special note are those of the executable set of primitives, related to remote code execution.

5. REFERENCES

[1] Jxta 2.5 rc1, June 2007.
`http://download.java.net/jxta/build`.

[2] J. Arnedo-Moreno and J. Herrera-Joancomartí. Persistent interoperable security for jxta. In *Proceedings of the Second International Workshop on P2P, Parallel, Grid and Internet Computing (3PGIC) 2008*, pages 354–359. IEEEPress, 2008.

[3] J. Arnedo-Moreno and J. Herrera-Joancomartí. A survey on security in jxta applications. *Journal of Systems and Software*, page *To be published*, 2009.

[4] D. Bailly. Cbjx: Crypto-based jxta (an internship report), July 2002.

[5] CCITT. The directory authentication framework. recommendation, 1988.

[6] B. Cohen. Incentives build robustness in bittorrent. *1st Workshop on the Economics of Peer-2-Peer Systems*, 2003.

[7] G. C. et al. Wireshark, 1998. `http://www.wireshark.org/`.

[8] K. Matsuo, L. Barolli, F. Xhafa, A. Koyama, and A. Durresi. Implementation of a jxta-based p2p e-learning system and its performance evaluation. *International Journal of Web Information Systems*, 4(3):352–371, 2008.

[9] E. S. Josefsson. Ietf rfc 3548 - the base16, base32, and base64 data encodings, 2003. `http://www.ietf.org/rfc/rfc3548.txt`.

[10] D. Sax. Dns spoofing (malicious cache poisoning), 2003. `http://www.sans.org/rr/firewall/DNS_spoof.php`.

[11] W. Simpson. Ppp challenge handshake authentication protocol (chap), 1996. `http://tools.ietf.org/html/rfc1994`.

[12] SUN Microsystems Inc. Jxta v2.0 protocols specification, 2007. `https://jxta-spec.dev.java.net/nonav/JXTAProtocols.html`.

[13] SUN Microsystems Inc. Java cryptography architecture (jca), 2008. `http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html`.

[14] C. A. T. Dierks. Ietf rfc 2246: The tls protocol version 1.0, 1999. `http://www.ietf.org/rfc/rfc2246.txt`.

[15] F. Xhafa, R. Fernandez, T. Daradoumis, L. Barolli, and S. Caballe. Improvement of jxta protocols for supporting reliable distributed applications in p2p systems. In *International Conference on Network-Based Information Systems (NBiS)*, pages 345–354, 2007.