Internship report

CbJx: Crypto-Based JXTA

June 24, 2002 - December 20, 2002

Sun Laboratories Europe

Damien Bailly

Principals

Sun Microsystems: Gabriel Montenegro
INSA Lyon: Stéphane Frenot

Abstract

This internship is an important part of the formation delivered by the department of Telecommunication, Services and Uses at the National Institute of Applied Sciences (INSA), Lyon - France. It takes place during the fall semester of the fifth year and represents the opportunity to realize a concrete project in a professional context. It was undertaken at Sun Microsystems Laboratories Europe, Montbonnot - France. This internship aims at improving the security of peer to peer networks by providing a fully distributed and scalable infrastructure.

This is achieved by applying the SUCV (Statistical Uniqueness and Cryptographic Verifiability) technology also known as CGA/I (Cryptographically Generated Address or Identifier) to the Project JXTA.

SUCV technology was first introduced to improve security in IPv6 networks. It was also proved to be useful in neighbour discovery and in securing ad-hoc networks. The latter has strong similarities with peer to peer networks in terms of dynamicity and scalability.

Project JXTA is an open source project initiated by Sun Microsystems. It aims at providing a peer to peer infrastructure on top of which developers can build their own applications. Project JXTA defines a set of protocols to allow communication between peers, and thus is language independant. As of today the JXTA protocols have been implemented in many languages, however the reference implementation is developed under the Java 2 Standard Edition.

This internship provides the reference implementatation with the SUCV technology. This allows applications developped on top of JXTA to rely on a powerful security infrastructure.

Furthermore, principles described in SUCV can easily be extended to group management by improving their security and dynamicity. Project JXTA can benefit a lot from these features since groups are heavily used in order to segment the global network formed by the JXTA peers.

Aknowledgements

First of all I would like to thank Gabriel Montenegro for having proposed such an interesting internship. Moreover it was a pleasure to work with him thanks to his availability and kindness. He was a constant source of knowledge during the six month I spent in SunLabs. Many thanks go out to all the people involved in the Holonet project and to the JXTA community for its active mailing lists.

I would also like to thank everybody at SunLabs for having made my internship a pleasant time.

A special thank goes out to Dominique Use for his technical support.

At last I would like to thank all the people who have made interesting and various talks during the so-called "gourmandises cerebrales".

Contents

Chapter 1

Introduction

1.1 The Holonet Project

As I entered SunLabs Europe I joined the Holonet Project. Holonet investigates how verifiable end to verifiable end can supersede and resolve the current obstacles to end to end communications. The SUCV technology was defined in this project by Gabriel Montenegro (Sun Microsystems) and Claude Castellucia (INRIA - National Research Institute in Computer Science and Automatism). The idea behind SUCV is to provide nodes of a network with identifiers (IDs) generated from cryptographic material and presenting uniqueness properties as well as the ability for a node to prove that it "owns" its ID. It's obvious that such properties would improve security by limiting impersonation attacks.

Proposals have been written on how Mobile IPv6, Neighbor Discovery, Securing Group Management and securing Ad Hoc Networks can profit from SUCV. This principles can be applied to the highest layers up to the users themselves. It can allow user-authentication without having to rely on a trusted third party such as a PKI (Public Key Infrastrucutre). Currently there are four people working on the Holonet Project: Gabriel Montenegro and Claude Castellucia - the authors of SUCV. Julien Laganier, a PhD student working on the implementation of SUCV into the IPv6 layer. At last I am in charge of integrating SUCV within Project JXTA, an open-source project which provides a peer to peer infrastructure on top of which applications can be built.

1.2 Project JXTA

Project JXTA is an open-source project initiated by Sun Microsystems. It aims at defining a common platform which allows the creation of a wide range of distributed applications and services to be built on top of it. This platform is defined by a set of protocols which allow any connected devices on the network to communicate. Currently there is an effort to standardize the JXTA protocols through the IETF (Internet Engineering Task Force) process. Project JXTA is independant of the environment on which it is implemented. However the reference implementation is bound to the Java 2 Standard Edition. Beside this, several other implementations

are available and able to interact with each other (Java Micro Edition, C, Python, Perl, Tini, Ruby, Smalltalk, etc).

Project JXTA is also transport independant. Theoriticaly it works with any transport protocol. In practice this is done by plugging a JXTA transport protocol implementation into the platform module. Currently transports on which JXTA is working are: TCP, HTTP, BEEP and TLS. The JXTA implementation of the UDP transport is under development. High level protocols are qualified as transport protocols because they are used to transport JXTA messages just like any "standard" transport protocol. This is the case of HTTP, for instance, which being used as a transport protocol, allows NAT (Network Address Translator) and firewall traversal.

Project JXTA also comprises a set of sub-projects which are services and applications built on top of the JXTA Platform. The difference between a service and an application is that a service provides a more generic functionnality which can benefit to many applications. Thus services can be seen as building blocks for the applications.

A complete list of projects including all the JXTA implementations, services and applications is available at the JXTA website - *http://www.jxta.org*.

1.3 SUCV: Statistical Uniqueness and Cryptographic Verifiability

SUCV technology was first introduced to improve security in IPv6 networks by addressing the address ownership problem. This was intended as a solution in operations like Binding Updates in mobile IP, source routing, etc, which allow hosts to modify other hosts routes. These operations are subject to highjacking attacks where malicious nodes can redirect traffic away from its destination. The idea in SUCV is to provide hosts with special addresses or identifiers, derived from cryptographic material and presenting the property of being unique within the network. (Hence the name SUCV - Statistical Uniqueness and Cryptographic Verifiability). The concept behind verifiablity is that a host can prove that it owns its address and thus allowing redirection operations only for hosts owning the address being redirected.

This technology was also proved to be useful in a couple of other cases such as ad-hoc networks routing where securing the route discovery protocol was an issue. In this case the problem is still the address ownership problem because a malicious node can initiate a route request on behalf of another node and redirect its traffic. The same problem arises with route replies where anyone can reply with a fake response preventing the traffic to reach its destination. Then again the solution is to allow only discovery messages containing the proof that the sender owns its address.

The SUCV technology can also improve security in group management by addressing the proof of membership problem. Using SUCV, a node can prove to any other node that it is a legitimate and authorized member of a given group. This apply even if the group is a spontaneous group, which is useful for dynamic environments such has peer to peer and ad hoc networks.

1.4 Project CbJx

CbJx stands for Crypto-Based JXTA. This project proposes to apply the principles of SUCV into Project JXTA. Project CBJX was started at the beginning of my internship and will remain open and available to the JXTA community. This project aims at providing a powerful and fully distributed security infrastructure that any applications and services in JXTA can benefit from.

This project focuses on the verifiable end to verifiable end rather than on securing the routing mechanisms which is a topic left open to the JXTA community. Concerning the group management issues an infrastructure is provided to support the proof of membership problem. However in both cases it remains to the developers to improve and use all those features according to their security requirements. At last this project investigates how users can benefit from the SUCV technology by extending the concept of verifiable end to verifiable end to authenticated user to authenticated user.

Chapter 2

Project JXTA

2.1 Motivations

During the past few years peer to peer applications have grown considerably. As of today various peer to peer applications address a wide range of needs such as file sharing, remote collaboration (instant messaging), distributed computing, etc. However these applications use more or less the same basics but they all are incompatible in the sense that each vendor develops its on set of protocols to enable communications between peers. The main idea in Project JXTA is to unify those protocols to permit applications to be built on top of them without "re-inventing the wheel" in each new peer to peer system.

2.2 Design considerations

Project JXTA was initiated by Sun Microsystem in order to solve this problem. The first step was to identify a common set of operations to any peer to peer systems while keeping away any application-specific assumptions and focusing on the core principles of such systems.

This list has led to the definition of six protocols, each of them addressing a specific item. The design of those protocols, the JXTA protocols, was made in order to address the widest set of peer to peer applications. Therefore the protocols remain independant of any operating system, network transport, development language and type of devices (from small cell phones and PDAs - Personal Digital Assistants - to large supercomputers clusters). The fact that peer to peer environments might be highly dynamic - peers joins and leaves the network at any time - was also considered when designing the JXTA protocols. Furthermore the fact that peers are often located at the edge of the network and that they might stand behind NATs and firewalls has been taken into account. As a result the JXTA protocols allow this kind of peers to participate in peer to peer activities. The JXTA protocols state also a list of recommendations such as routing packets for other peers, caching information to reduce traffic, etc. These are not mandatory and should be implemented according to the available resources on the peer.

2.3 JXTA Architecture

This set of six protocols defines the core level of the JXTA architecture. The core level, also known as the platform module, addresses the concept of Project JXTA: providing a common infrastructure on top of which peer to peer services and applications can be built. Logically, services and applications constitute the two other levels of the JXTA architecture.

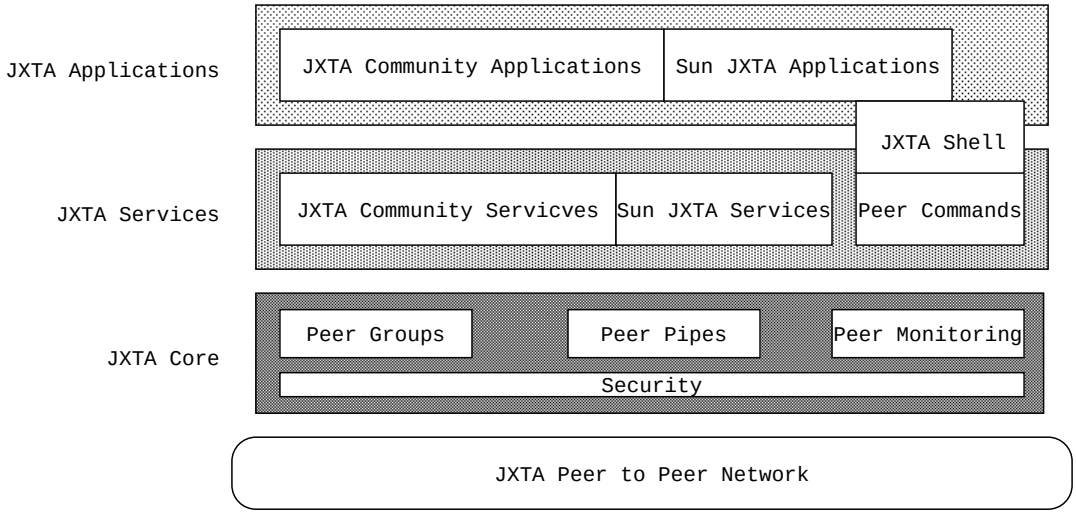

Figure 2.1: *the three layers architecture*

- **The core level** provides essential functionalities in order to form the network on which peer to peer communications can take place. This level is shared by all peer to peer devices so that interoperability becomes possible.
- **The service level** provides usefull functionalities that are not absolutely necessary for the network to operate. Examples of network services, include search and indexing, directory, storage systems, file sharing, distributed file systems, resource aggregation and renting, protocol translation, authentication, etc.
- **The application level** is where specific applications such as file sharing or instant messaging stand. Typically the application interacts with the user whereas services don't. An example of JXTA applications is the JXTA Shell or the InstantP2P (also known as myJXTA), a chat and file sharing application. The boundary between services and applications is sometimes unclear: An application to one customer can be viewed as a service to another customer.

2.4 The Peer to Peer Overlay Network

In any peer to peer systems, a network made of the participating peers is formed. This network ensures traditional network functionalities such as routing, message forwarding, broadcast, etc. Although this network uses "traditional real-world"

networks to physically transmit data, it can be seen as an independant and different network since its topology might have nothing in common with any real networks it uses. Actually this kind of networks is refered to as virtual networks (in opposition to the "real" networks) or as overlay networks which illustrates the idea of a network over one (or more) physical network(s).

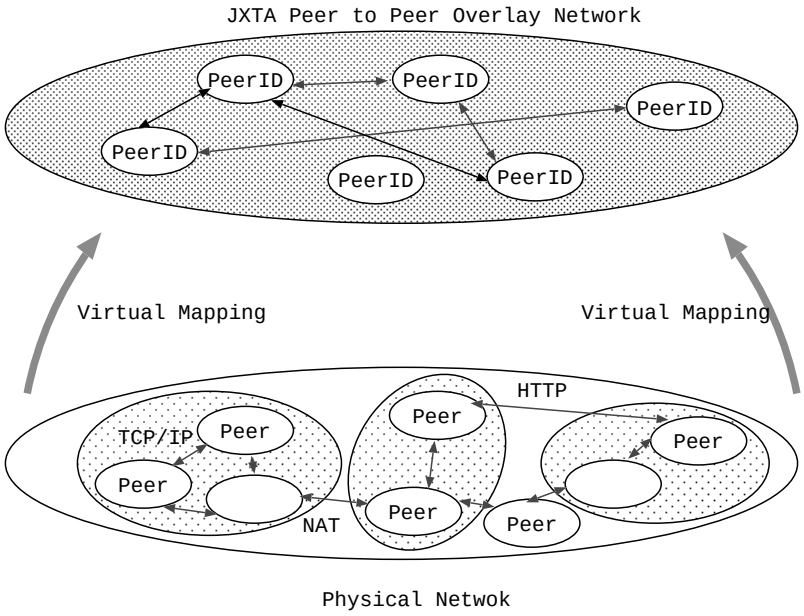

Figure 2.2: *The JXTA Peer to Peer Overlay Network*

The diverse protocols and implementations of the underlying networks are totally abstracted by the overlay network. In JXTA this abstraction is done by the platform module. For instance underlying NAT and firewalls disappear in the JXTA network.

2.5 Peer Groups

The JXTA overlay network might become very large as any peers implementing the JXTA protocols are automatically part of this network. A huge amount of nodes is something that remains unmanageable and not suitable in many cases. For instance it can become difficult to find a particular peer within such a network, even worse if one is looking for specific data and asks each peer until he gets what he's searching for? No need to say that there is a need to segment the JXTA network into smaller parts. To address this issue JXTA introduces an important concept: **peer groups**.

A peer group is a set of peers which share a common interest. For instance peers running the same application can be part of the same peer group. Peer group can provide services which are only available to members of the group. Going back to the search example, the search will be performed only within the peer group scope and not through the entire JXTA network. Peer groups can also be used on security purposes in order to limit the access to unwanted peers. At last one peer can participate in several peer groups.

2.6 Advertisements

In JXTA every resources on the network are described by the means of advertisements. Advertisements are meta-data documents presented in XML. They are composed of series of hierarchically arranged elements which may appear in any order. Advertisements are used to describe peers, peer groups, pipes, content, services and many other types of resources. When a new service is created it can define its own new advertisement or extend existing ones. The JXTA protocols also rely on several type of advertisements. This makes the advertisements an important element of the JXTA infrastructure.

The choice of XML was made for the following reasons:

- XML is programming language agnostic
- XML is self-describing
- XML content can be strongly-typed
- XML ensures correct syntax
- XML has the ability to be translated into other encodings such as HTML and WML. This feature allows peers that do not support XML to access advertised resources.

2.7 The Platform Module

Each JXTA protocol is designed to address one fundamental aspect of peer to peer networking and is divided into two complemantary parts. One part is run by the local peer and corresponds to the generation and sending of messages. The other part is conducted by the remote peer and consists of handling incomming messages and performing some tasks according to the messages. This is often implemented as a query-response system. Typically each peer implements both parts of the protocols. Messages sent by the protocols are structured documents rendered as XML documents.

The JXTA protocols are semi-independant between them. They are organized in a stackable fashion.

A peer can implement only a subset of these protocols, while relying on pre-specified behavior to eliminate the need for a protocol. For instance a peer can rely on a pre-defined set of ressources available on other peers and avoid the need of having a discovery mechanism to locate those resources. However each protocol depends on the underlying protocols and therefore it would be useless to run the discovery protocol without having the resolver protocol and the endpoint routing protocol to handle messages to remote peers.

Some of these protocols are actually implemented as services. Although the JXTA three layers architecture states that services are located within the service level on top of the platform module, these services are considered as core services and therefore are part of the platform module.

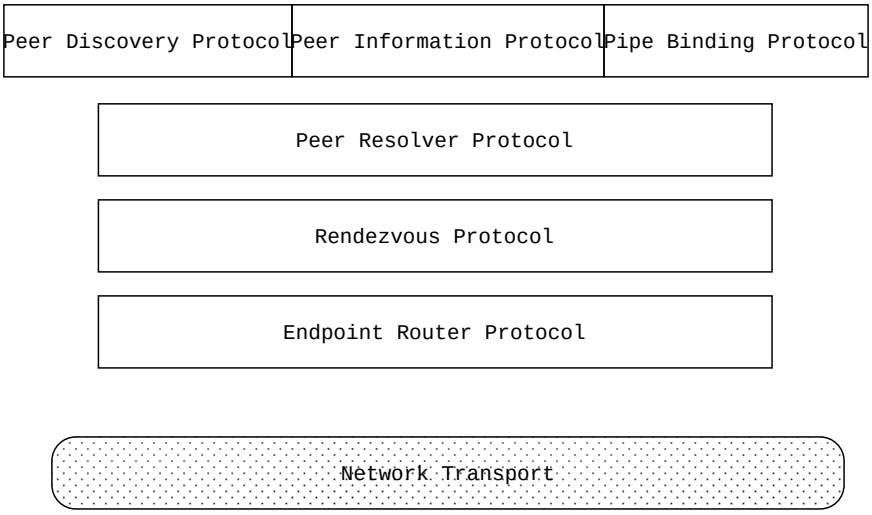

Figure 2.3: *The six JXTA prtotocols*

2.7.1 The Endpoint Routing Protocol

The Endpoint Routing Protocol is in charge of finding routes and sending messages to their destination. Like in any peer to peer environments peers may join and leave the nework at anytime and thus routes between peers may change, become unavailable or new routes may appear. Therefore the endpoint routing protocol is adaptative by nature and it relies on route discovery mechanisms to find routes between peers.

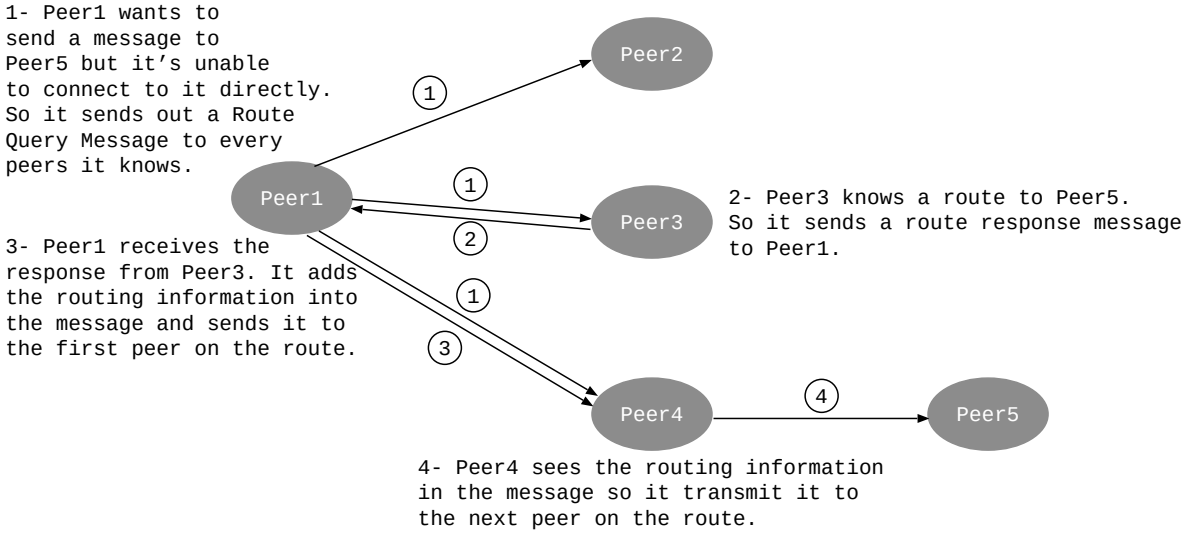

Figure 2.4: *The Endpoint Routing Protocol*

As its name indicates the Endpoint Routing Protocol is running at the endpoint layer. The endpoint layer is the lowest part of the JXTA infrastructure and it handles the interfaces between the JXTA virtual network and the underlying physical

networks. Each peer can have several endpoint interfaces according to the available transport protocols. For instance one can have both a TCP and an HTTP endpoint where the TCP endpoint is used to send messages onto the local network whereas the HTTP endpoint is used to send messages to peers located behind the local network's firewall for instance. Which endpoint to use is determined by the Endpoint Routing Protocol according to the route to the destination peer.

2.7.2 The Rendezvous Protocol

The Rendez Vous Protocol introduces the notion of special peers, called Rendezvous peers, which can be used to re-propagate messages they have received. A peer can dynamically become a rendezvous peer and/or can dynamically connect to a rendezvous peer. The connection between a peer and a rendezvous peer is achieved by an explicit connection, associated to a lease. The notion of rendez-vous peer is also linked to notion of peer groups. A peer acts as a rendez vous peer only within a specific peer group and messages are propagated only within the scope of this peer group. The name rendezvous comes from the fact that those rendezvous peers know many peers and they act as a sort of group's meeting place. This is the reason why rendezvous peers usually cache more information than simple peers do and therefore respond to more requests. They can be seen as the group's super peers.

2.7.3 The Peer Resolver Protocol

The Peer Resolver Protocol addresses the scheme of query-response messages. It allows a peer to send generic queries to one or multiple remote peers within a peer group and identify matching responses. The Peer Resolver Protocol uses the Rendezvous Protocol to disseminate a query to multiple peers. This protocol id intended as a base for higher level applications and services to send queries on the network.

2.7.4 The Pipe Binding Protocol

This protocol introduces the notion of pipes. A pipe is a virtual communication channel between peers. It is used to send data to or receive data from a remote peer. The distinction between send and receive needs to be done because pipes are unidirectional and asynchronous communication channels. The reason lies in the fact that JXTA is transport independent and therefore should work even with transports providing only one-way communications. However bi-directional pipes do exist in JXTA but they are not required by the specifications.

There are currently three different types of pipes:

- *JxtaUnicast*: Unicast, unsecure and unreliable. This type of pipe is used to send one-to-one messages.
- *JxtaUnicastSecure*: Unicast, secure (using TLS - Transport Layer Security). This pipe is equivalent to the UnicastType pipe, except that the data are protected using a TLS connection between the endpoints.

- *JxtaPropagate*: Diffusion pipes. This pipe type is used to send one-to-many messages and is unsecure and unreliable.

If a pipe is established between peer1 and peer2. It will be seen as an output pipe for peer1 (the sender) and an input pipe for peer2 (the receiver).

If peer2 wants to create the input pipe it just binds the pipe to its endpoint waiting for messages to arrive. This operation only creates one half of the pipe. To be working, the other end of the pipe must be bound to the endpoint of peer1. This is done when peer1 creates the corresponding output pipe.

The problem is how peer1 can know that it is binding the same pipe that peer2 is using to its endpoint. This is achieved by the Pipe Binding Protocol. The Pipe Binding Protocol uses Pipe Advertisements to resolve pipes. A pipe Advertisment describes a pipe but it's not related to any specific peer. Hence the need to determine if the end of the pipe into which we want to send data is bound to a peer. Although pipe abstractions are build on top of the endpoint layer the Pipe Binding Protocol uses the Peer Resolver Protocol to find where the other end of the pipe is bound to.

2.7.5 The Peer Discovery Protocol

The Peer Discovery Protocol allows a peer to discover resources within a peer group. Since resources are described by the mean of advertisements the Peer DIscovery Protocol actually discovers advertisements. This protocol is build on top of the Peer Resolver Protocol because the Peer Discovery Protocol uses the generic query-response scheme by asking all its known peer for advertisments. Any details about the requested advertisement might be added into the query and zero or more responses containing matching advertisments are received back. The Peer Discovery Protocol is also used to publish resources. A resource can be discovered by remote peers only if it has been published by the peer.

2.7.6 The Peer Information Protocol

The Peer Information Protocol allows peers to obtain status information from remote peers. This status information is currently limited to the uptime and the amount of traffic processed by the peer. Moreover this protocol is turned off by default. The Peer Information Protocol is also built on top of the Peer Resolver Protocol because of the standard scheme query-response which fits this protocol, as well.

Chapter 3

Statistical Uniqueness and Cryptographic Verifiability

3.1 Verifiable End to Verifiable End

SUCV defines a new type of identifiers: CBID (Crypto-Based IDentifier). The name comes from the fact that CBIDs are derived from a cryptographic material. Each node on the network has its own private-public key pair. From the public key it generates its CBID. As a CBID must uniquely identifies a node on the network, the generation must respect the uniqueness property, this can be done with a secure hash of the public key. The CBID can therefore be seen as an expression of the node's public key. In other words it binds the node's identifier with its public key. This simple but powerful property can then be used to prove the identifier's ownership. (The owner of the identifier is the only one who knows the corresponding private key).

This is particularly well suited for a node to prove that it really is the sender of a message. A node, called Node A, wants to send a message to another node, called Node B. To accept the message Node B needs to be sure that the message really comes from Node A (and not from any malicious nodes) and that the content of the message wasn't changed.

For node B to be able to verify the identity of the sender Node A must add in its message the following fields before sending it on the network:

- *dest CBID*: the CBID of the destination node. It's not required by SUCV however it's needed for the message to be delivred to it's destination.
- *source CBID*: Node A's CBID. It is not required if the message contains Node A's public key from which Node A's CBID is derived.
- *Public Key*: Node A's public key. It's not a requirement to include the public key in the message. However the receiver must know this key in a way or another in order to verify the message.
- *Data*: the data to be sent.

- *Signature*: the signature of the content of the message using the node's private key. The content includes all the fields of the message and not only the data.

Figure 3.1: *the fields of a verifiable message*

Upon reception of the message, Node B (or anyone else on the network) is able to authenticate the sender's CBID and check the integrity of the data. This is achieved by performing the following steps:

- Node B extracts from the message the signature and the public key.
- It verifies the signature with the public key. This operation proves both the integrity of the data and the fact that the sender is the corresponding private key owner.
- In order to check the authenticity of the sender's CBID, Node B re-generates the sender's CBID by performing a secure hash of the message's public key. This newly generated CBID is by definition the sender's CBID.
- At last, the comparison between the CBID source of the message and the newly generated CBID achieves the authentication procedure.

A noticable property of CBID is that it never relies on any centralized infrastructure such as a PKI (Public Key Infrastructure). This makes it highly dynamic and scalable. This technology is suitable to thousands (even millions) of nodes as long as a new CBID remains unique in the network. Using 128 bits long identifiers, which is the case in JXTA, permits a population of CBID uniformely distributed (because of the secure hash properties) among the total available CBIDs: 2^{128} different ones. No need to say that this can fit very well even into very large peer to peer environments.

3.2 Group Management

3.2.1 A Naive Approach

The CBID principles can be applied for securing groups management while improving dynamicity and flexibility. Here groups are also identified by means of CBIDs. It implies that a group must generate its own public-private key pair and, here again, the group's CBID is the secure hash of the public key.

Nodes can prove their membership into a group in a very similar way as they prove their CBID ownership when sending messages. They just have to prove they "own" the group's private key. Let's look at a very basic scenario which illustrates this idea.

Node A creates a new group, called Group 1. Node A generates a pair of public-private key for Group 1 and from the public key it computes Group 1's CBID. At this point Node A can prove to anyone else that it is a member of Group 1 by showing that it knows the group private key. For instance it can send a message containing the following fields:

- *the group's CBID*
- *the group's public key*
- *the signature* of the two preceeding elements with the group private key.

Any other node can verify the membership of the sender by checking the signature and the CBID as previously described. It proves that the sender knows the group private key and therefore that it is a legitimate member of that group.

If another node is admitted into the group it needs the group keys to be able to prove its membership. The group keys can be encrypted and sent over the network to the new member.

However this simple scenario presents several drawbacks. The group's private key is transmitted on the network, which, though encrypted, can become a security hazard. Another drawback is that each node plays an equal role within the group. They all can act as group managers as owning the private key confers them the right to admit new members into the group. Therfore this solution is not suitable for secure and dynamic group management.

3.2.2 Second Approach

All these caveats can be addressed by the means of authorization certificates.

Authorization certificates are different as names certificate which is the traditional way certificates are used. A name certificate binds a fully qualified name to a public key whereas an authorization certificate gives a certain right to a public key (or to its hash).

Node A is the group creator so it issues the group's keys and CBID as before. However to proof of membership is now done by means of authorization certificates. As Node A is the group manager it can issue a certificate for itself using the group's keys. This certificate gives Node A the right to be a member of the group.

To prove its membership within the group, Node A sends a verifiable message containing its certificate.

The verification is still pretty straightforward. Upon reception of a message containing a membership certificate, one must verify the source and message integrity as usual and then the certificate's validity. The term "validity" here means the validity period but also the correctness of the certificate's signature and the fact that the issuer's CBID is the hash of the group's public key. One must also check that the subject is the CBID of the sender (the hash of the message public key).

An important field in the authorization certificate is the delegation field. It specifies wether or not a node is allowed to propagate its rights or authorizations, here the membership property, to other nodes. Concretely it simply allows a node to admit new members into the group.

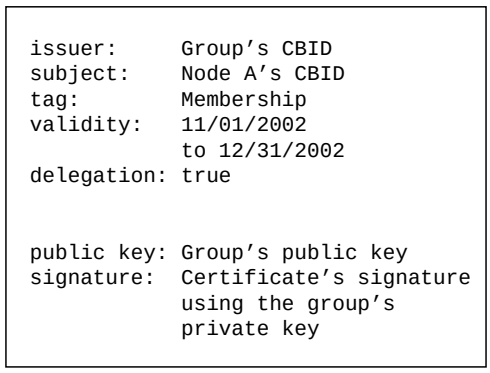

Figure 3.2: *the authorization certificate delivered by the group to Node A*

Figure 3.3: *the fields of a message proving membership within a group*

Since a member allowed to admit new nodes within the group doesn't know the group's private key it issues a new authorization certificate with its own keys and sends all its certificates to the new node.

A node needs to provide a complete certificates chain starting by a certificate issued by the group end ending with a certificate authorizing the node itself to prove it is a valid member.

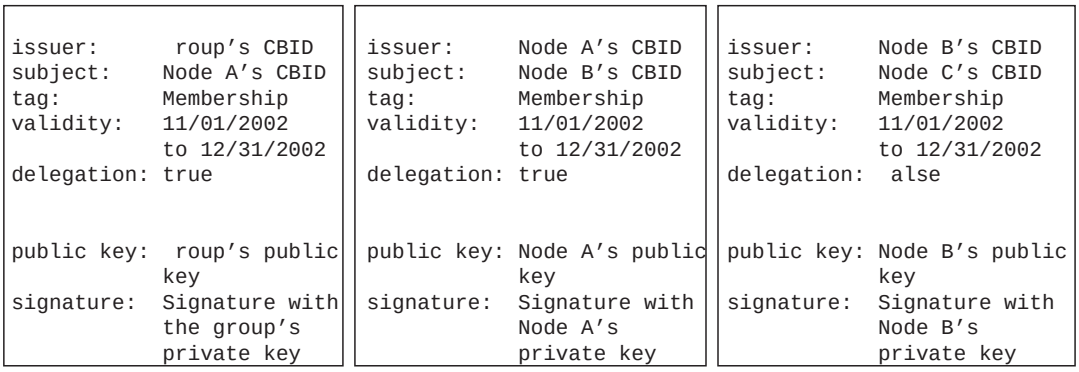

Figure 3.4: *An example of a certificates chain. The delegation field must be enabled except in the last certificate where it is optional.*

To prove its membership a node includes its complete chain of certificates into a verifiable message.

Then the receiver checks the message as usual and instead of checking one certificate it verifies the whole list.

By means of authorization certificates the need to distribute private keys all over

the network is avoided. It also gives a much thiner control of the distribution of group controllers. An important feature which was never mentionned is how group accesses are granted to new members. This part is not SUCV's business but a policy based on a per group needs and requirements. The same applies to wether or not delegation should be given to a peer.

Chapter 4

Secure Cocktail Effect Authentication

4.1 Motivations

Up to now only authentication between nodes have been discussed and the identifier ownership problem has been addressed using CBIDs. However moving up to the user level, authenticated nodes doesn't solve the user authentication problem.

The user authentication problem can be illustrated by the following example: User Alice uses a device, device A, connected to the network and she'd like to establish a communication with another user, Bob, which uses device B connected on the same network. How Alice can then be sure she's communicating with Bob's device and not with any other device on the network?

4.2 Principles

Here again SUCV and CBIDs can help to solve this problem. The previous chapter shows how network devices can be authenticated by means of their CBIDs. Therefore Alice can easily know which network device she's talking to. However she needs to establish a binding between user Bob and Bob's device's CBID.

One easy way is to use another authenticated channel than the network and ask user Bob for his device's CBID. The authenticated channel can be for instance a human channel such as the telephone network where Alice have authenticated Bob by voice recognition or Alice and Bob can be at the same place and talk to each other. (Alice has authenticated Bob by visual recognition, etc ...). Once such a channel has been established Bob can tell Alice his CBID and then Alice device's can send messages to Bob's device.

The issue is that there is no good way for humans to deal with long sequence of bits and therefore in such a scenario the human bootstrapping process of the communication represents one of the weakest links in the chain. The idea to address this problem is to transform CBIDs into something that can be easily handled by human beings, for instance a bunch of words. It is of first importance that this transformation doesn't leverage the security level of CBIDs.

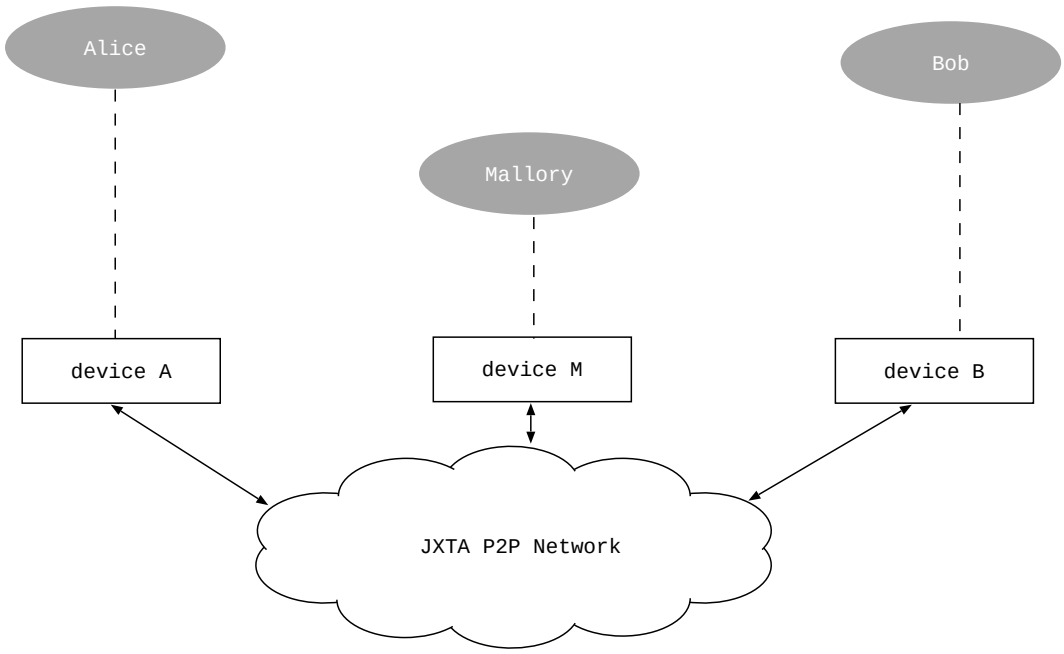

Figure 4.1: *How Alice can be sure she is sending messages to Bob's device and not to anyone else?*

4.3 Protocol

The Sceau protocol can be used to establish communication between two users (Alice and Bob, for instance) or between a user and a network device such as a printer or an access point, etc. In both cases Alice knows and has identified her correspondant by the means of an authentic (possibly "human") channel. In the case of a network device she can recognize the device because of its shiny logo printed on it, the sticker specifing the device name, etc. This human channel is authentic but not secret, for instance Alice and Bob may be at a cocktail party with other participants or Alice can be at an airport lounge and anything she observes on the device can be seen by other people.

The scenario presented below is a brief summary of the protocol and omits several options which can be used for optimization.

Bob's device listens for requests on the network, so Alice can broadcast a request on the network (assuming that Alice has no idea of Bob's CBID yet). This message reaches Bob's device which replies with the following response:

- *Bob's CBID*
- *Bob's public key*
- *signature of the message* using Bob's private key

As in the previous part, upon reception Alice is able to check the integrity of the message and determine that the source of the message is really Bob's CBID. She

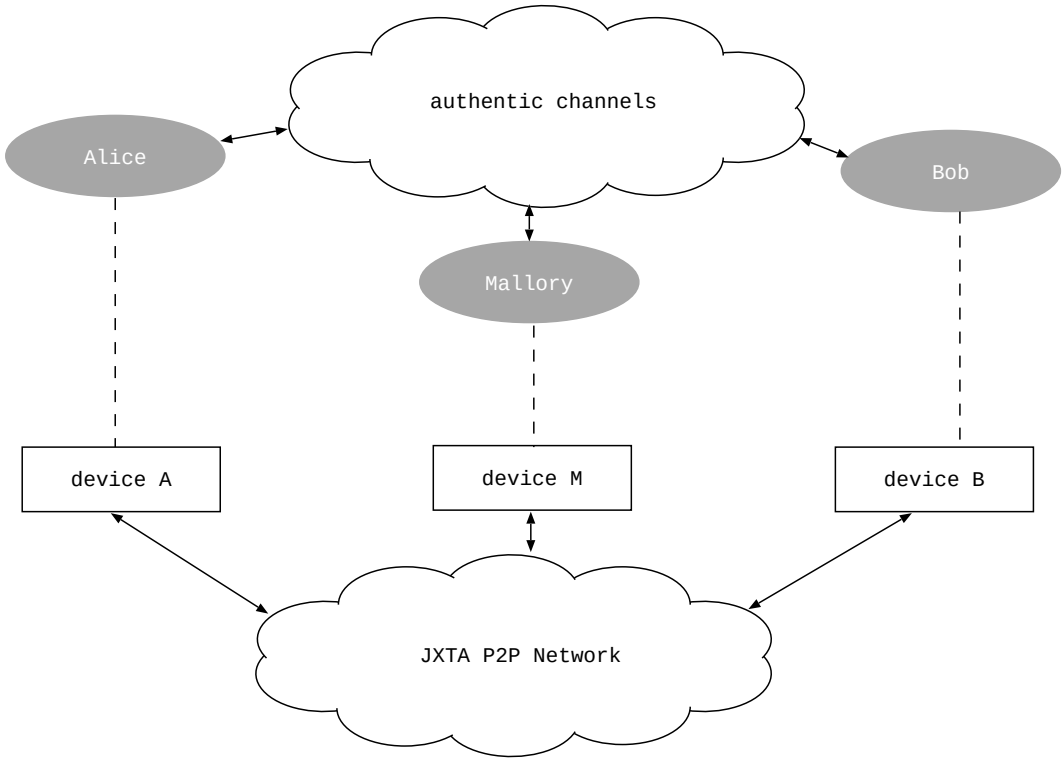

Figure 4.2: *Alice and Bob can exchange their CBID on the authentic channel.*

might also receive replies from other nodes which were listening for requests, so she need a way to identifies which request comes from Bob's device. Even if she receives only one request she must check that the message really comes from Bob and not from anybody else.

Alice's device displays the list of the received responses on a screen, for instance. Alice, as any other human is not at ease to handle the raw bits of CBIDs. So CBIDs are translated into a set of words which is much more user-friendly. This set of words is refered to as a *sentence*.

Alice can now ask Bob on the authentic channel for his *sentence*. Bob then tells his sentence on the authentic channel and Alice identifies which one it is on her device.

This protocol is asymetric, meaning that at the end of the exchange only Alice knows Bob's CBID. For Bob to know Alice CBID the Sceau Protocol must be run once again with Bob acting as the requester.

Once Alice and Bob know each other CBIDs they can engage in secure communication using any of several protocols proposed in the literature.

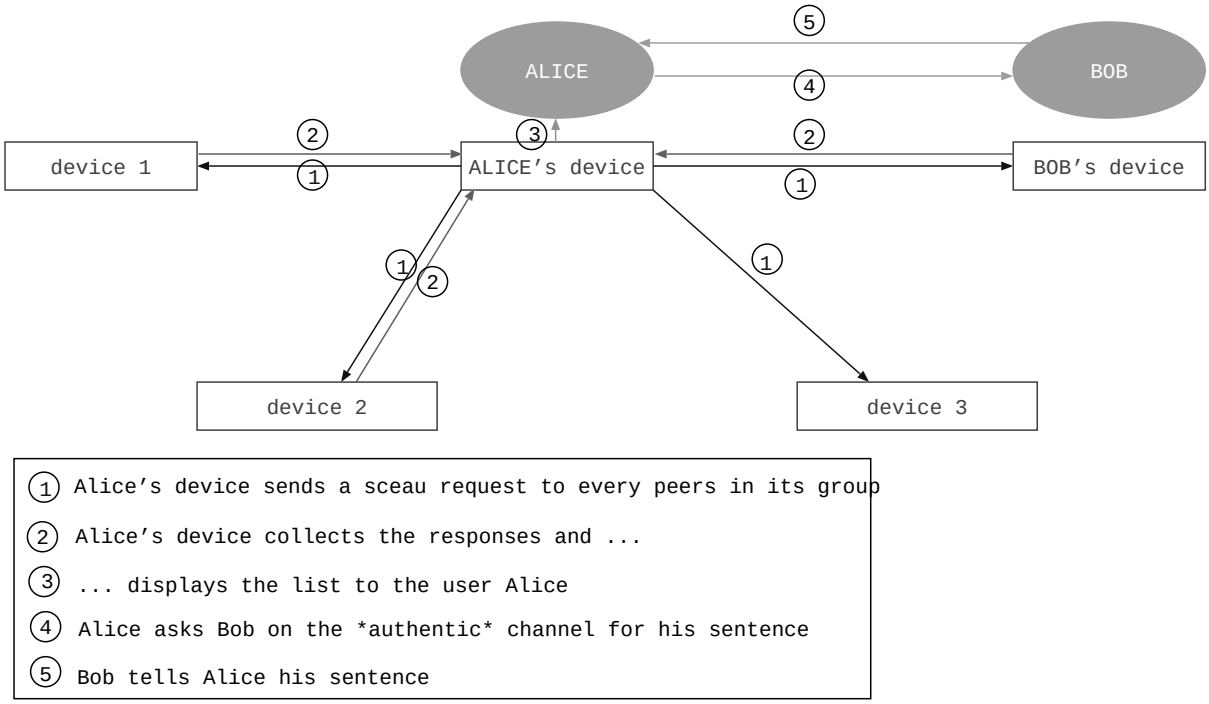

Figure 4.3: *Illustration of the Sceau protocol.*

Chapter 5

CbJx: Crypto-Based JXTA

This project aims at applying the concepts of SUCV and Sceau into JXTA in order to provide a powerful and fully distributed security infrastructure. This project, started at the begining of the internship, will remain open to the JXTA community. It implies that this project is an open-source project according to the JXTA license. (The JXTA license is available in appendix C page 66).

5.1 CBID

5.1.1 JXTA ID

In JXTA IDs are used to refer to peers, peer groups, pipes and other JXTA resources. JXTA IDs provide unambiguous references to the various JXTA entities. These IDs are presented as URNs (Uniform Resource Name) which are themselves presented as text. The fact that JXTA IDs uniquely identify resources in the overlay network implies the problem that when a process generates a new ID it is not aware of all the already existing ones. So it has no way to guarantee the uniqueness property of its new ID. JXTA solves this by using UUIDs (Uniformely Unique Identifier). UUIDs are pseudo-random numbers which thanks to their length (128 bits) and randomness property are very much like being unique in the network. A JXTA ID represented as a URN may look like:

urn:jxta:uuid-<128 bits represented as an hex string>

IDs represent diverse resources on the network, and in order to differentiate those different types, a two digits (8 bits) code is appended at the end of the URN. For instance a peergroup ID (code 02) looks like:

urn:jxta:uuid-<128 bits represented as an hex string>02

Moreover certain types of IDs depend on the peer group in which they are located. In such cases the URN identifying the resource contains both the peer group UUID (the 128 bits identifying the peer group) and the resource UUID (the 128 bits identifying the resource within the peergroup). This leads to an URN of the form:

urn:jxta:uuid-<peer group's 128 bits><resource's 128 bits><resource's type code>

For instance for a PeerID:

urn:jxta:uuid-<peer group's 128 bits><peer's 128 bits>03

Peer IDs depends on the peer group where the peer is located. However peers can change groups and even belong to several peer groups. So theoritically a peer can own several peer IDs (according to the peergroup they have joined). In practice this mechanism is not implemented and each peer has only one ID. When new peers join the JXTA network they automatically become a member of the world/net peer group. It leads to the following ID:

urn:jxta:uuid-59616261646162614A78746150325033<peer's UUID>03

where '*59616261646162614A78746150325033*' is the well-known UUID of the world peer group.

Changing peer identifier as groups are joined is an issue that should be addressed in the upcoming versions of the JXTA platform.

5.1.2 Introducing CBIDs in JXTA

Project CBJX needs to introduce a new type of IDs within JXTA: CBIDs. CBIDs play a very similar role as UUIDs. The main difference lies in the way they are generated. Using CBIDs instead of UUIDs leads to URNs of the form:

urn:jxta:cbid-<peer group's 128 bits>02 for a peer group ID.

urn:jxta:cbid-<peer group's 128 bits><peer's 128 bits>03 for a peer ID.

The '*cbid*' string replacing '*uuid*' into the URN makes clear the signification of the bits. CBIDs are only used for peer IDs and peer group IDs which is enough to address the three topics presented in the two previous chapters. (verifiable-end to verifiable-end, securing group management and user authentication).

At this stage one may notice the following problem: In the case of a URN containing both a peer group identifier and a resource specific identifier, cbids and uuids can stand together in the same url leading to something like:

urn:jxta:cbid-<uuid of the peer group><cbid of the peer><resource's type> or

urn:jxta:uuid-<cbid of the peer group><uuid of the resource><resource's type>

The first case applies only to Peer IDs where the peergroup part is always the identifier of the world/net peer group. This group is very particular because it contains automatically all the peers on the JXTA overlay network and its identifier is fixed and well-known. In such conditions a fake CBID (not derived from a public key) can be generated from the well-known bits and used instead of a real CBID.

In the second case the generation of a new resource's ID requires the peer group ID to be a UUID instance. Project CbJx solves the problem by converting the CBID into a UUID. This operation is possible because CBID are derived from a secure hash and presents similar uniqueness properties as UUIDs.

5.1.3 Generation of CBID

Knowing how CBIDs can be handled by JXTA it's time to focus on their generation. The chapter 3 state that a CBID is a secure hash of a public key. Each resource (peer and peergroups) must therefore have its own key pair, however the CbJx implementation of CBID's generation is slightly different.

JXTA allows TLS (Transport Layer Security) to be used as a transport protocol which provides security by means of cryptographic materials: X.509 certificates and RSA keys. This material is generated when JXTA is launched for the first time and then stored in the jxta's Personal Security Environment directory (pse/). According to the computational cost to generate these materials, Project CbJx tends to re-use the same materials to generate peers' CBID. Therefore CBIDs are not directly derived from a public key but from the X.509 certificate used in the TLS endpoint. This certificate is called the peer root certificate and is included in the peer advertisement.

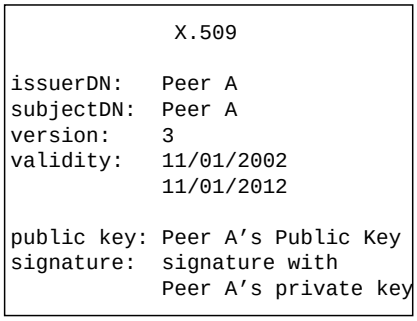

Figure 5.1: *a self-generated root certificate for a peer.*

X509 certificates contain the peer's RSA public key and they add flexibility to the infrastructure. Expiration date are used for the certificate (and therefore for the keys and the cbid). One can also imagine to implement CRL (Certificate Revocation List) where invalid or "bad" peers are listed, etc. The secure hash function is the SHA-1 hash. The output of the SHA-1 hash is 160 bits long and is troncated to the 128 most significant bits.

Peer groups identifiers are generated in the same manner except that a new RSA key pair and X.509 certificate are generated. Certificates and the corresponding encrypted private key are also stored into the PSE directory for later use. For a description of the PSE directory see the developer guide in appendix B (page 62).

5.2 Verifiable-End to Verifiable-End

5.2.1 Sending messages in JXTA

In JXTA a message consists of several elements. For instance there are an element for the destination address, one for the source address, one for the routing information, etc. Services and applications handling messages can, of course, add their own

elements. The number of elements is not specified and varies according to the diverse operations performed on the message.

According to chapter 3, when a peer sends out a message it must include its CBID, public key (which is now replaced by its root certificate), and finally sign the message. These operations are the last ones to be performed before sending out the message. Therefore they must occur at a very low layer of the platform: the Endpoint.

The JXTA Endpoint handles interactions between the rest of the JXTA platform and the transport networks. As different transport protocol may be used according to the destination of the message, the endpoint must interacts with different transport implementations. This interaction is done by means of "Endpoint Protocols" which are interfaces between the JXTA Endpoint layer and a specific transport protocol.

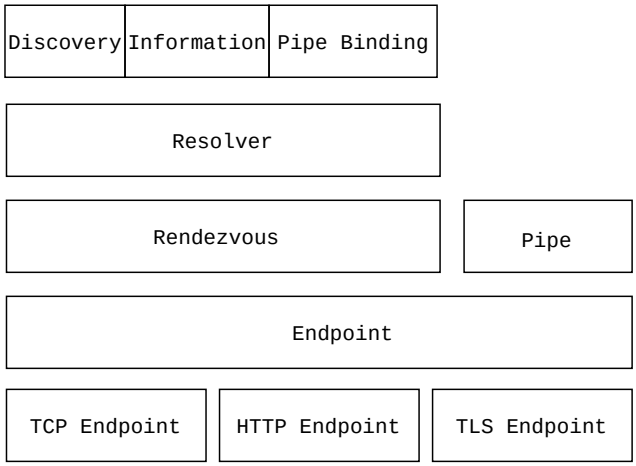

Figure 5.2: *the Platform layers and the different Endpoint Protocols.*

To send a verifiable message the endpoint just needs to add one more element in the message. This element must contain the source peerID, the peer certificate and the signature of all the elements (except some partcular cases - see the developper-guide appendix B (page 54) for the list of such elements). A simple and easy solution could have been to place a filter which catches all the outgoing messages and adds an element containing the required information. Unfortunately this solution is not doable (except in special cases) because of the way messages are sent.

In JXTA messages are sent using a particular object: a Messenger. A messenger is bound to a specific destination address and is the one who really sends the message to that address. Therefore sending a message leads tot he following scenario:

Assuming that an upper layer process wants to send a message to a destination address, it asks the endpoint layer to give him a messenger object it can use to send messages. It is the role of the Endpoint to determine which messenger to return according to the destination address.

Moreover some parameters may be addded to the address: the service's name and parameters. They indicate the destination endpoint to which service the message

addresses	endpoint protocol
tcp://<ip-address>:<port#>	tcp endpoint
jxtatls://<Peer-ID>	tls endpoint
http://JxtaHttpClient<Peer-ID>	http endpoint

Table 5.1: *The mapping between addresses and Endpoint Protocols: the first word of the destination address determines the endpoint protocol used to send the message.*

must be delivred. Service's name and parameters are appended to the end of the address:

tcp://<ip-address>:<port#>/servicename/serviceparameters

jxtatls://<Peer-ID>/servicename/serviceparameters

http://JxtaHttpClient<Peer-ID>/servicename/serviceparameters

In many cases upper level processus do not know which endpoint transports are available on the destination peer so they wish to send messages to the following address:

jxta://<Peer-ID>/servicename/serviceparameters

Such addresses are handled by the Endpoint Router (a special Endpoint Protocol) which determines which transport-specific Endpoint protocol (TCP, HTTP, TLS, etc) to use to reach the destination. If it can't find one out it performs a route discovery and returns its own messenger to the upper layers. When the upper process sends the message, the endpoint router messenger adds the routing information (obtained in the route discovery procedure) into the message and uses an approriate transport-specific messenger. This messenger is selected according to the routing information. All these operations are performed transparently with regard to upper layers (upper layers do not know which transport-specific endpoint protocol is really used to send the message).

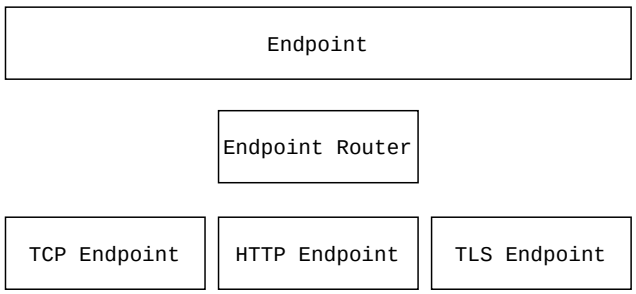

Figure 5.3: *The endpoint router.*

5.2.2 The CbJx Endpoint

Sending verifiable messages can be achieved by the creation of a new special endpoint protocol (the CbJxEndpoint) which applies similar principles as the Endpoint Router. When a message is sent using the CbJx messenger (the messenger of the CbJx Endpoint) then certificate, CBID and signature are added into a new element and the messenger of another endpoint protocol is used to really send the message. As the CbJx endpoint doesn't perform routing services it can not know which transport specific messenger to use. So it asks the endpoint router for an approriate messenger. The sending operation now contains one more step: going through the CbJx Endpoint.

When a process wants to send a message it asks the endpoint for a messenger corresponding to a destination address of the form:

cbjx://<Peer-ID>/servicename/service parameters

The CbJx endpoint must return a messenger but it doesn't know which transport-specific messenger to use so it asks the endpoint router by changing the destination address into:

jxta://<Peer-ID>/cbjxservice

The endpoint router which handles such addresses returns an appropriate messenger to the CbJxEndpoint. The CbJx endpoint returns its own messenger which actually uses the endpoint router messenger when a message is sent. Then the process uses the CbJx messenger to send its message. The CbJx messenger creates two elements. One containing the peer root certificate, its CBID and the message signature and another one containing the original destination address (the orginal addresses are used on the destination peer for the message to be delivred to the right service, so it's important to keep them somewhere in the message). It appends those elements to the message and sends it using the endpoint router messenger.

This scenarion is summarized in figure 5.4.

5.2.3 The case of propagate messages

Propagate messages are messages sent to many peers (similar to broadcast messages). In such cases messenger are not used but the endpoint asks each endpoint transport to propagate the message. The only endpoint transport capable to perform this task is the TCP endpoint (which actually uses UDP multicast in this particular case). The solution is to apply an endoint output filter to all of those messages before they are sent. The output filter adds the certificate, the CBID and the signature before the message is given to the endpoint transport. A such easy solution is not compatible with unicast messages and the use of messengers.

5.2.4 Receiving messages

When a message is received it must be delivred to the approriate service. Here again this is the task of the endpoint. It uses the service's name and parameters appended

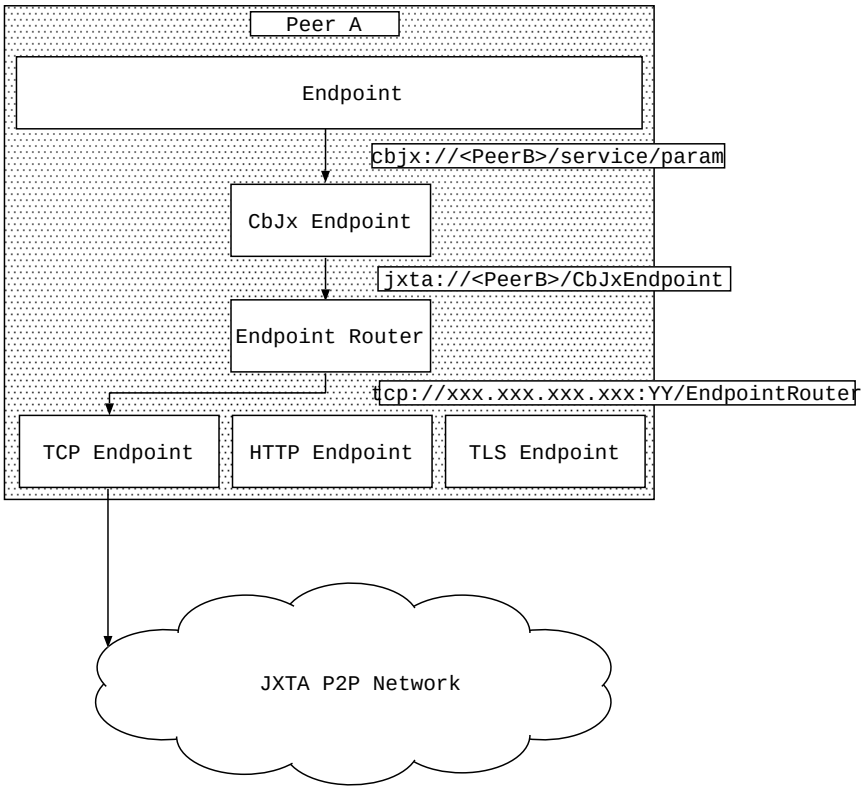

Figure 5.4: *Sending a message.*

at the end of the destination address to determine to which service it must give the message.

Verifiable messages containing a CBID, a root certificate and a signature must also be verified before they are delivred to upper layers. In this case the filter mechanism applies very well.

Each message arriving at a peer endpoint and containing the source CBID, certificate and the message signature goes through an input filter: the CbJx input filter. The input filter checks the CBID and the signature ensuring the integrity of the data and the authenticity of the sender. It replaces the element containing this information with a new element claiming the correctness of the message. Upper layer can then check for the presence of this element in order to know if the message was a valid CbJx message.

This procedure works perfectly with propagate messages. In the case of unicast messages one more operation is needed. The destination address is changed by the CbJx Endpoint of the sender and therefore the original address must be set back in the message in order to be delivred to the right service on the destination peer.

This is possible because the original address is stored into the message and the message is delivred to the CbJx Endpoint.

The sender's CbJxEndpoint has changed the destination address into:

jxta://<Peer-ID>/CbJxEndpoint

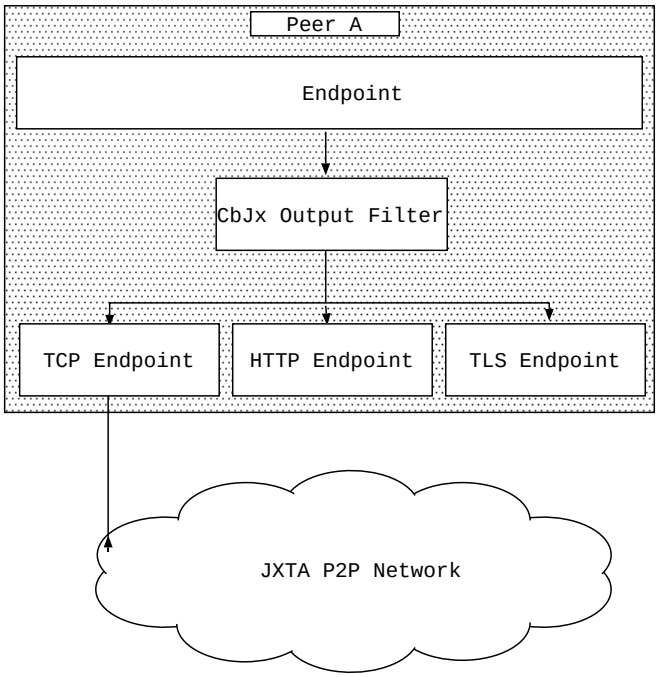

Figure 5.5: *Sending a propagate message.*

which means that the message will be delivred to the *CbJxEndpoint* on the destination peer. The latter just changes the destination address by setting back the original one (which is stored in the message) and gives the message back to the endpoint which transmits it to the approriate service.

All these operations are completely transparent for upper layers. An upper layer process just has to check if the message contains the element added by the CbJx input filter to prove its validity. Otherwise if it doesn't care about it it can process the message as usual.

Example where this is used is the Rendezvous protocol where only valid messages are accepted for lease negociation, however relaying a message within the group is performed wether or not the message was properly signed.

The resolver service also uses this element in order to ensure integrity of the queries/responses. It also checks that the requester ID is the same as the message sender ID. On the other hand a resolver response doesn't provide a field for the ID of the responder. So the only thing one can be sure when he receives a response is its integrity but it can't know who sends it. (unless he uses its own authentication mechanism).

The pipe abstraction (unlike the resolver service) handles messages so any application or services using pipes to communicate are able to check wether or not the messages were properly processed by the CbJx endpoint.

For more detailed information on the CbJx Endpoint see the developer guide in appendix B (page 54).

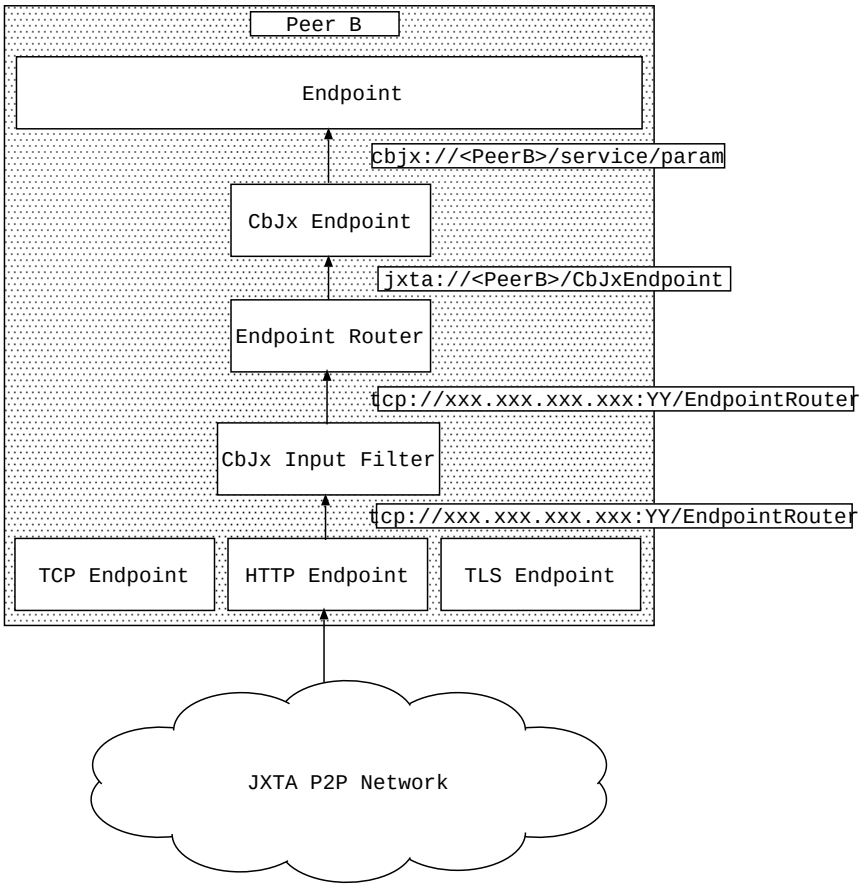

Figure 5.6: *Receiving a message. The CbJx Input Filter does not change the address. (It only verifies the message validity).*

5.3 Peer Group Management

5.3.1 Creation of a peer group

To create a new peer group the first thing to do is to construct a peer group advertisement. A peer group advertisement describes the peer group and contains the following fields:

- *peer group id*: the urn uniquely identifying the peer group
- *module spec id*: the urn of the module specification. This can be used to retrieve the module implementation advertisement of the group which contains a list of all the services available within the group.
- *name*: the name of the peer group (not necessarily unique)
- *description*: the description of the peer group.

In order to generate the peer group ID an X.509 certificate with a private/public key pair is generated. This certificate is similar to the peer root certificate presented

in figure 5.1. The first 128 bits of the hash SHA-1 of the certificate form the peer group CBID. The CBID is included in the URN which forms the peer group ID.

As peer root certificates, group certificates are stored in the personal security environment directory. See the developer guide in appendix B (page 62) for more details on the structure of the PSE directory.

From the peer group advertisement, the group can be instantiated.

5.3.2 Joining a peer group

Once the group as been instantiated (from an advertisement created by the peer or discovered on the network), one needs to become a member to be able to use services within this group. Note: Currently many core services do not check that a peer is a member of the group before allowing it to use this service. As a result a peer doesn't have to be a member of the group to use its services.

Joining a group is done in two steps using the group membership service. The group membership service is in charge of managing the access of peers within the group. It's (theoritically) the only service one can use without being a member of the group.

Project CbJx proposes several membership services. However its highly recommended to develop its own membership service based on the control access policy one wants to implement. All the CbJx membership services are based on different simple control access mechanisms, but they are mainly intended as examples on how CBIDs can be used to improve security in group management. All of them match the JXTA two steps scheme for membership appliance.

The first step is to call the apply method of the membership service. CbJx membership services uses this step to contact a group controller (possibly the peer itself). According to the control access policy, a list of certificates is returned or not to the requester.

With such certificates one can now call the join method of the membership service. According to the correctness of the certificates provided it gets back a credential containing the list of certificates. This credential can then be used to prove membership when contacting group services.

The authorization certificates are SPKI certificates.

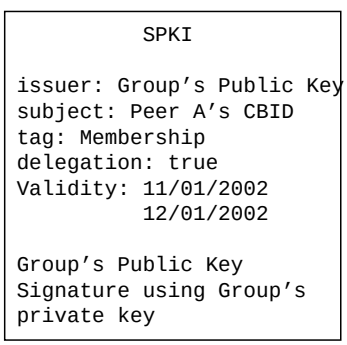

Figure 5.7: *a SPKI authorization certificate.*

Because CBIDs are derived from certificates instead of public keys twice more certificates are needed to prove membership within a group. The procedure for a peer to prove it is a member of the group remains the same. The list of certificates (or the credential which contains the list) must be included into a message or another signed document. The signature must be verified using the peer's public key as well as its CBID with the peer root certificate.

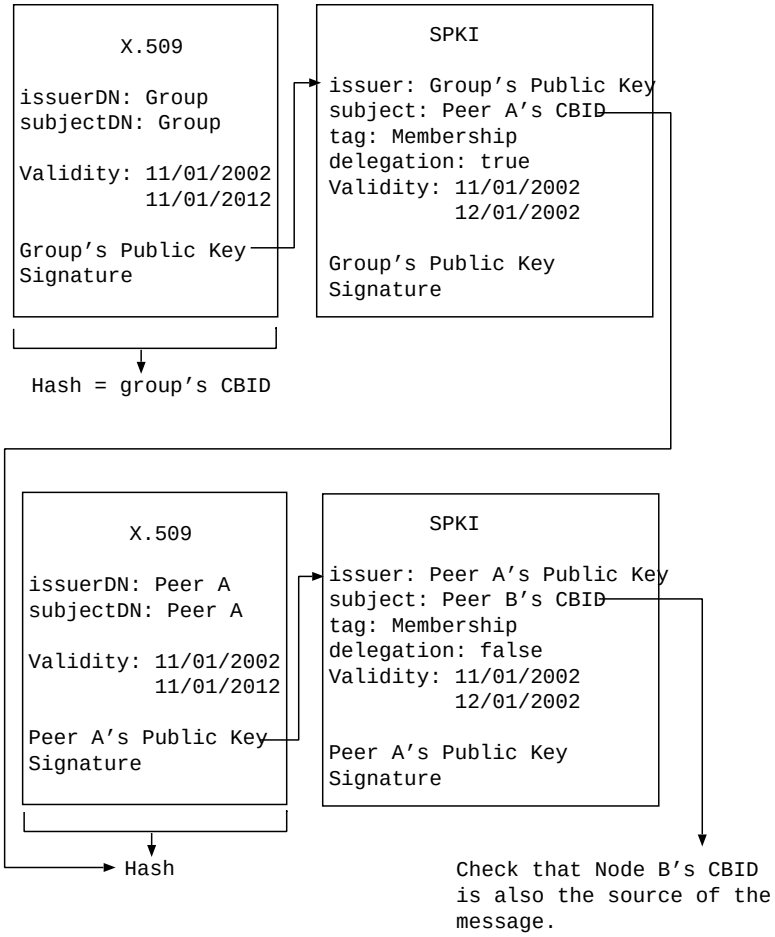

Figure 5.8: *an example of a certificates list used to prove membership within a group.*

5.4 Sceau

5.4.1 The Sceau Service

The Sceau protocol presented in the previous chapter is implemented as a JXTA service. In order to use this service a new peer group containing this service must be created.

As the JXTA protocols, this service is divided in two parts: one sends the requests and the other receives and treats them, and possibly sends back a reply. Although matching very well the query/response scheme and the use of the resolver service,

the sceau service is built using pipes to achieve communication. The reason is that pipes handle messages and not any upper level object and therefore contains the security elements added by the CbJx endpoint. This service relies on the CbJx mechanisms and handling some upper level documents would have led to perform twice some operations such has signing both the document and the message, etc.

As it stands in chapter 2 pipes are unidirectional communication channel and therefore to establish a query/response scheme two different pipes are needed.

- On the requesting peer (also known as the client side) an output pipe is used to send messages to other peers running the Sceau service. This pipe is a propagate pipe which allows messages to reach all the peers having bound the other end of the pipe to their endpoint. A standard unicast input pipe is also needed in order to receive responses from other peers.
- On the receiving peer (also known as the server peer), the input propagate pipe is bound to the endpoint in order to receive queries. Queries are then handled by the service and if a response needs to be sent a unicast output pipe is opened to the requester.

The requester sends a Sceau query wich is an XML document. containing the following fields:

- *Type*: request type.
- *ID*: used to match query-responses
- *PipeAdvertisement*: the pipe advertisement corresponding to the pipe used to send the response.
- *PeerAdvertisment*: the peer advertisment of the requester.
- *BitsValue*: some bits of the CBID one wants to authenticate. (optional used to narrow down the amount of responses)
- *BitsNumber*: the number of bits specified in *BitsValue*.

The message containing the request doesn't have to contain any CbJx elements. If a reply needs to be sent it will be sent whoever is the requester. Sending a response doesn't disclose any private information about the peer so it doesn't matter who gets it.

If the bits specified in the request match the peer's CBID then a response containing the following fields is sent:

- *Type*: response type.
- *ID*: used to match query-responses
- *PeerAdvertisment*: the peer advertisment of the responding peer.

The request contains a pipe advertisement used to create an output pipe from a remote peer to the requester. Unlike queries, responses message must go through the CbJx endpoint because the source authentication and the data integrity of the response are key-elements of the Sceau protocol.

Finally the requester receives responses from zero or more peers. Messages containing responses are checked and then the application process which originated the request is notified that responses have been received.

The task of the Sceau service stops here. However the Sceau protocol specifies user interaction using an authentic channel. This part of the protocol must be handled by the highest JXTA level: the application. For instance a list of responses might be displayed to the user so that he can ask a remote user for the '*sentence*' representing its CBID. The requester user can then select the correct response in the list.

5.4.2 Sentence

In the previous chapter the difficulty for a human to handle long sequence of bits was mentioned and *sentences* were proposed as a user-friendly representation of a CBID to address this issue. Now that CBIDs have been defined and explained in details it's time to see how a CBID can be represented by a set of words: the *sentence*.

CBIDs are 128 bits sequence. In order to translate those bits into words a dictionary is used. The dictionary comes from the One Time Password (OTP) protocol where a dictionary of 2048 words is defined. 2048 different values can be represented by a sequence of 11 bits. Therefore each words represents 11 bits.

11 bits sequences can not represent exacly 128 bits. 11 words of 11 bits each represents 121 and 12 words 132 bits. The most obvious things to do seems to use 12 words and having 4 bits unnused. In this case a sentence would be a set of 12 random juxtaposed words. A sequence of twelve random words is a bit too long to be user friendly (which is the aim of *sentences*). So its size is limited to 10 words in order to keep enough bits (110 bits) to ensure uniqueness of the sentences and improving a little bit the user-friendliness.

In an ideal world a sentence would consist of three or four words but this leads to huge dictionaries which would take lot of space in a device's memory. Another solution which deserved to be investigated is the use of grammars to form sentences having real meanings and which despite their length can easily be handled by human users.

The generation of a *sentence* is pretty straightforward. The CBID is divided into two bits sequences of 64 bits: the most and the least significant bits. Each 55 most significant bits of the 64 bits sequences are translated into a sequence of 5 words. The sentence is then formed by appending the 5 least significant words to the 5 most significant words.

5.4.3 Applications based on Sceau

Project CbJx doesn't stop here and proposes to save the CBID of users that have been identified by means of the Sceau protocol. It avoids going through the Sceau

the CBID: *8924C6EAF6D7F99D4A5E5B5F684276CE*
the most significant bits: *8924C6EAF6D7F99D* the 5 most significant words: *GANG LOY MESS TONY FEST* the last 9 bits of *8924C6EAF6D7F99D* are not used.
the least significant bits: *4A5E5B5F684276CE* the five least significant words: *AJAR UTAH SCOT ROAD MAT* the last 9 bits of *4A5E5B5F684276CE* are not used.
the *Sentence*: *GANG LOY MESS TONY FEST AJAR UTAH SCOT ROAD MAT*

Table 5.2: *The generation of a sentence from a CBID.*

protocol again when establishing communication with an already authenticated peer. Saving a CBID actually implies the creation of a particular authorization certificate.

```
issuer:     Alice's public key
subject:    Bob's CBID
tag:        myBoss
validity:   11/01/2002
            to 12/31/2002
delegation: false


public key: Alice's public key
signature:  Certificate's signature
            using Alice's
            private key
```

Figure 5.9: *the SPKI certificate used to save an authenticated CBID.*

Unlike membership certificates where the authorization is the membership property, here the authorization is the right to use a given name. This name is chosen by the user who saves the CBID and its scope is only local (meaning that a name is unique only with regard to a specific peer).

Using this process one is able to build a list of known peers identified by locally unique names. Such names can then be used by user and applications to refer to peers in a more friendly fashion.

The user guide in appendix A (page 43) describes how a list of known peers can be built using the applications developed in the CbJx Project.

This list of known peers can also be used to constitute an very basic access list for the membership service. For instance Project CbJx proposes membership services where only peers authenticated by the Sceau protocol and which have been saved by the user are able to join a group. The user and developer guides provide more information about this (appendix A and B).

Name authorization certificates are not limited at saving peers' CBIDs. They can be used for a wide-range of purposes. CbJx also proposes to use name authorization certificates to handle names within a group. One of the CbJx membership services uses such certificates to ensure unique names within the group. The peer who wants to join the group asks the membership service for a given name. If this name is

not already used within the group the peer receives an authorization certificate (an SPKI certificate) issued by the group and stating that the requested name can be used by the requesting peer (identified by its CBID) within this group. Here the group constitutes another scope in which names are unique.

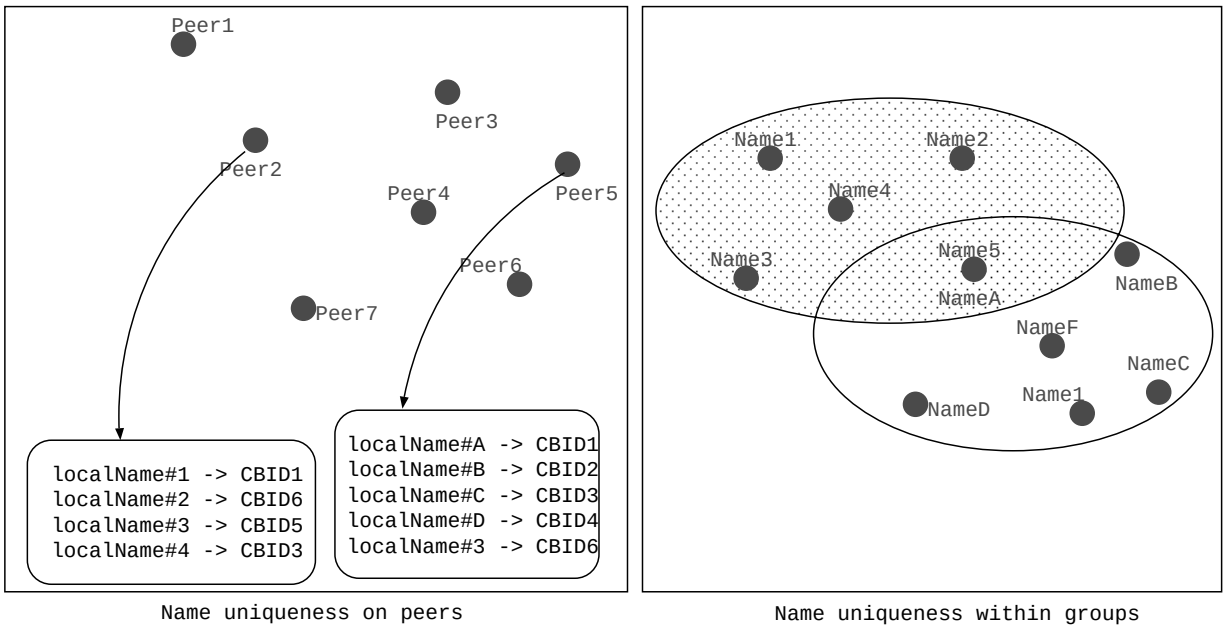

Figure 5.10: *Name uniqueness within different scopes.*

Chapter 6

Future work

Project CbJx currently provides an implementation of CBIDs, verifiable end to verifiable end communication, secure group management and user authentication. But much more can be developed with these principles. Here are some ideas.

6.1 Routing mechanism

One important security issue within JXTA concerns the routing mechanism. Although JXTA allows everyone to implement its own routing mechanism, the specifications rely on route discovery to find route between peers. According to the results of the route discovery, source routing is performed to deliver a message to its destination.

The JXTA Endpoint Routing Protocol is therefore highly subject to misdirection attacks. Anyone is able to send fake route discovery responses and redirect the messages of a peer.

CbJx can be used to secure this mechanism by allowing only the destination peer (or a set of trusted peer) to reply to a query.

6.2 Interaction between CbJx and JXTA applications

Currently there are no way to identify uniquely peers in a user-friendly manner in applications. Either peers are identified with their ID (unique but not friendly) or with the name provided in the Peer Advertisement (friendly but not unique).

Using the locally unique names chosen by the user when an authenticated CBID is saved is a way to have user-friendly unique references to peers in applications.

6.3 Securing advertisement

Currently advertisements in JXTA are absolutely not secure. Once they are published they are available within the whole peer group and anyone in that group can

access and modify them (even if the peer who published the advertisements is offline). It makes it very easy to alter the content of an advertisement. Signing the advertisement and checking its source can improve a lot this security flaw.

6.4 Distributed Hashtable

Distributed hashtable is often implemented on peer to peer networks. (Chord, Pastry, CAN, etc). They allow to store and retrieve data all over the network. They use nodes and keys identifiers to find a node responsible for some data. Actually they all form an overlay network with its own routing mechanism where the principle is to route queries to the node responsible for the data. They all are subject to security hazards since no one is able to prove that it is responsible for some data. As a result any one can claim that it is responsible though it is not. The routing mechanism is also a weak point concerning the security.

Though it is thought CBID can help to improve security in this domain, more research and investigation are needed to propose a workable solution.

6.5 The tie with federated naming

As seen in Chapter 5 (page 37) the use of authorization certificates to ensure uniqueness of names within a given scope can be developed to form a federated naming infrastructure.

6.6 And more ...

As mentioned at the beginning of this report, Project CbJx will remain open to the JXTA community. It involves that the project will continue to evolve in this community in order to meet everyone's expectations. According to the enthusiasm met when contacting the Sun JXTA community, the current lack of security within JXTA and the powerful properties of CBIDs and authorization certificates, the future of CbJx remains completely open ...

Chapter 7

Conclusion

CbJx aims a applying the principles of SUCV into JXTA. The simplicity of these principles make CbJx both powerful and fully distributed. In CbJx each peer is identified by a self-generated certificate but it is not convenient to use it as an identifier so instead a secure hash of the certificate is used as the peer identifier. This very simple principle is the key to all the applications developed in this project.

Currently CbJx provides three main security application which can then be used to improve other security weaknesses.

- Verifiable-end to verifiable-end communication. Every one is able to know the ID of the sender of a message.
- Secure group management. An infrastructure addressing the proof of membership problem. It doesn't provide control access to groups nor it checks that a peer is a valid member of the group when a service is contacted for instance. It is up to the developers to choose how these topics should be implemented (since it may change according to the application one is running).
- User authentication. Provides a way for user to authenticate themselves and then engage secure communication by means of "traditional" security tools. This is performed by the Sceau protocol which is fully distributed. It doesn't rely on a PKI (which is a centralized infrastructure and has the weakpoint of unchecking the binding between names and users).

Although CbJx improves a lot the security level in JXTA it doesn't mean that everything is now completely secure in this peer to peer infrastructure. For instance the routing mechanism remains a security hazard and should be addressed in the future. CbJx can be used to reach this goal. Advertisments, used to describe resources within the JXTA network are also insecure (see chapter 2).

However the advantage of CbJx is to provide an infrastructure to improve security. It means that many applications can be built on it and its use within the JXTA community remains open. This project fills a need in JXTA and the echoes heard from the community are very encouraging for the future.

Appendix A

User Guide

Two JXTA applications take advantages of the CbJx infrastructure. They have been developed to test and demonstrate the CbJx Project's capabilities. They are the JXTA Shell and MyJXTA (also known as the InstantP2P). The JXTA Shell is a command line interpreter, similar to the Unix shell. MyJXTA is a GUI (Graphical User Interface) application allowing file-sharing and chatting.

This short guide presents the functionalities added (to those applications within the CbJx project. For more information on how to configure a peer or on the already existing functions refer to the JXTA website at *http://www.jxta.org*.

A.1 JXTA Shell

All the shell commands provide manual pages accesible through the '*man*' command. These are helpful information on how and for what a command is used.

In this part each command is briefly described and illustrated by a concrete example.

A.1.1 The *peers* command

This command is used to retrieve and display a list of peer advertisements located in the curent group.

'*peers -r*' sends a discovery message in the current group in order to get a list of peer advertisement published in this group.

Example

With the commands '*peers -r*' (sends a discovery message) and then '*peers*' (displays a list of discovered peers) one can see something like the following results:

A.1.2 The *cat* command

The '*cat*' command is used to display Documents. Its syntax is simply '*cat variable_containing_the_document*'.

peer0: Alice
peer1: Bob
peer2: Conrad
peer3: Dave
peer4: Someone
peer5: SomeoneElse

The '*cat* command can be used to display any kind of documents. For more information use the '*man cat*' command.

Example

Going on with the previous example if one wants to see Alice's Peer Advertisement the command is '*cat peer0*'.

A.1.3 The *digit* command

This command displays some digits of a peer CBID. Each peer is identified by a URN of the form:

urn:jxta:cbid-<peergroup id (128 bits)><peer id (128bits)>03

'*03*' at the end indicates that the URN represents a PeerID. The 128 bits of the '*<peer id>*' part is the peer CBID. To see your own CBID the command is '*digit*'.

To see only the first bits of your cbid use the '*digit -l bit_length*' command.

Valid bit lengths are 8, 16, 32, 64 and 128.

Those commands also apply to other peers with '*digit -p peerX*'. '*digit -p peerX -l bit_length*' displays the first number of bits of peerX's CBID. The number of bits being specified by '*bit_lengths*'.

In all the above, bits are displayed in hexadecimal strings.

Example

If one wants to see Alice's CBID one uses '*digit -p peer0*' and the result is:

peer0's CBID: 8924c6ea-f6d7-f99d-4a5e-5b5f684276ce

A.1.4 The *sentence* command

This command displays some words of a peer sentence. A sentence is a set of 10 words representing a CBID. The usage of this command is similar to the '*digit*' command.

'*sentence*' displays your own sentence and '*sentence -l nb_words*'displays only the first *nb_words* of your sentence where *nb_words* is a number between 1 and 10.

It is also possible to display other peers' sentences with the -p option: '*sentence -p peerX* or *sentence -p peerX -l nb_words*'.

Example

'*sentence -p peer0*' displays Alice's sentence:

peer0's sentence: GANG LOY MESS TONY FEST AJAR UTAH SCOT ROAD MAT

A.1.5 The *getCBID* command

This command uses the Sceau service in order to get the CBID of a remote user. First you must start the Sceau service by creating a listener which waits for queries. This is done with the command: '*getCBID -l*'.

After this you may want to send a query using the sceau service with the commands: '*getCBID -d [digits]* or '*getCBID -w [words]*'.

Those commands sends a request to every peer having launched a Sceau service listener. You might get back an important amount of responses which is not convenient to use. So the '*-d*' option allows you to specify the first digits of the peer from which you want the CBID. The '*-w*' option is intended in the same purpose but instead of specifying the first digits of the CBID you specify the first words of the sentence of the remote peer. The sentence is a ten words sequence and is a representation of a CBID. The advantage is its user-friendliness since it's more convenient to deal with (for a human point of view) than a string of 128 bits. At last the '*getCBID*' command without options displays the list of the results of the latest query. (Note that only results from the latest query are accessible. Sending a new query therefore resets the list of results).

Example

For instance assuming that Alice and Bob, identified by the following CBIDs (and sentences), are, each of them, running an instance of the Sceau service. So do many other unknown users.

Alice	
CBID	8924C6EAF6D7F99D4A5E5B5F684276CE
Sentence	GANG LOY MESS TONY FEST AJAR UTAH SCOT ROAD MAT

Bob	
CBID	91BFF2AC2F10751134B39F682C68F6F9
Sentence	GUST YEAR LATE BRAD BIB POP HUNT SHAW BLAB GOWN

We assume that Alice wants Bob's CBID. She can send a request with the commands '*getCBID -d*' or '*getCBID -w*' (which are equivalent when no arguments are specified) but she will probably get too much responses to deal with.

In order to limit the amount of responses she can ask Bob, (at the phone or directly if they are not far away from each other, etc), for the beginning of its CBID ('*91*') or its first word ('*GUST*'). (Bob can use the '*digit*' and '*sentence*' commands

to get this information). She can now send a request with the commands '*getCBID -d 91*' or '*getCBID -w GUST*'.

This time only peers having their CBID starting with '*91*' or their sentence starting with '*GUST*' will respond to the request, limiting the amount of responses Alice will get back. If she still gets to much responses she can specify more digits or words in the request.

She can look at the results with the '*getCBID*' command and then ask Bob for its complete sentence that she will identify among her results:

result0: GUST SULK MEET BEN GILL BOUT BUSH AIRY THEE JACK
result1: GUST POT MEAN HIRE JOKE RITE WALT OR WANG HEAD
result2: GUST DESK JOLT OVER ASH SING GARB BAM SEEN DOG
result3: GUST YEAR LATE BRAD BIB POP HUNT SHAW BLAB GOWN
result4: GUST WANG NASH AMEN DISH SUIT SLIM TEST FOWL JOVE

Here Bob's sentence is the result number 3.

A.1.6 The *save* command

This command saves a result of the '*getCBID*' command under a given name. Name are unique with regards to a peer. It means that you cannot give the same name to two different results. To save a result of the '*getCBID*' command the syntax is: '*save -r result_number Name*'.

To see the list of peers already saved the command is simply '*save*' without any options.

Example

Let's go on with the example. Alice now wants to save Bob's CBID under the name "Boss". (assuming that Bob is in fact Alice's boss). This is done with the command '*save -r 3 Boss*'

The '*-r*' option specifies which results to save. In this case the result number 3 is saved, which is the one corresponding to Bob. (Remember that to see the list of results of the latest request the command is '*getCBID*'). "Boss" is just the name that Alice wants to give to Bob (of course it could have been anything else she wanted).

A.1.7 The *delete* command

This command deletes a record made with the '*save*' command. The syntax is pretty straight forward: '*delete Name*'.

Example

For instance if Alice wants, for any reasons, delete Bob's record that she has saved under the locally unique name "Boss", she just types '*delete Boss*'.

A.1.8 The *createGroup* command

This command creates PeerGroupAdvertisments. This is used to create new PeerGroups. The membership service used to access the group must be specified. Therefore the syntax is: '*createGroup -m membership groupName*'.

'*groupName*' is the name of the group and membership is one of the CbJx membership service:

- NullCbJxMembershipService: Allows anyone to join the group.
- BootstrapMembershipService: Allows any peers saved by at least one member of the group to join the group.
- UniqueNameMembershipService: Allows only peers known by the group creator to join the group. Moreover the requested name by a peer when it joins the group must be unique within the group.

Example

Back to the Alice's example. We assume Alice wants to create a group with Bob (her Boss) and a couple of its co-workers. As the group will be used for working purposes she doesn't want any unknown peers to be able to join the group. She uses the '*getCBID*' and '*save*' commands to save everybody she wants to be in her group (as she did to save Bob under the name "Boss"). Then she creates the new PeerGroupAdvertisement for her group with the command '*createGroup -m UniqueNameMembershipService AliceWorkingGroup*'

AliceWorkingGroup is the name of the group and the '*-m*' sets the membership service granting access to the group to be the UniqueNameMembershipService. This allow only peers which have been saved by Alice to join her group. Furthermore each member must use a unique name of its choice within the group.

A.1.9 The *groups* command

This command wasn't modified in this project. It aims at retrieving PeerGroupAdvertisements. It usage is similar to the '*peers*' command.

'*groups -r*' sends a discovery message in order to get PeerGroupAdvertisements published within the current group.

'*groups*' with no options displays the list of PeerGroupAdvertisements.

Example

For instance Alice can check with the '*groups*' command that her PeerGroupAdvertisement ,corresponding to AliceWorkingGroup, has been published.

For instance the output of this command might be:

Here Alice's PeerGroupAdvertisement is stored in the variable '*group2*'.

Assuming that Alice and its co-workers are in the same group, for instance the NetPeerGroup, Alice's co-workers can get Alice's PeerGroupAdvertisement with the '*groups -r*' command and then check they have received it by typing '*groups*'.

group0: chatGroup
group1: testGroup
group2: AliceWorkingGroup
group3: AnotherGroup

A.1.10 The *join* command

This command was modified to work with the CbJx membership services. However its syntax remains the same. To join a group from which we have the advertisemnt the command is '*join -d groupX*' where '*groupX*' is the variable containing the advertisement. For a complete description of this command, see the manual pages with '*man join*'.

Example

Alice wants to join her group so she uses the command '*join -d group2*'

She is then prompted to enter her username specific to this group. She can choose whatever she wants since she is the first one to join the group. It is important that Alice joins the group first because she is the one which have created the group and no one else will be able to join the group before her. Then any of Alice's co-workers can use the '*join -d groupX*' command to join Alice's group where '*groupX*' is the variable containing Alice's PeerGroupAdvertisement.

Note that Alice is not on her own list of saved peers but she can join because she is the creator of the group.

A.2 MyJXTA

A new tab named Sceau has been added into myJXTA (also known as InstantP2P). This new tab contains an application, the Sceau application, that uses the Sceau service in order to build a list of authenticated peers.

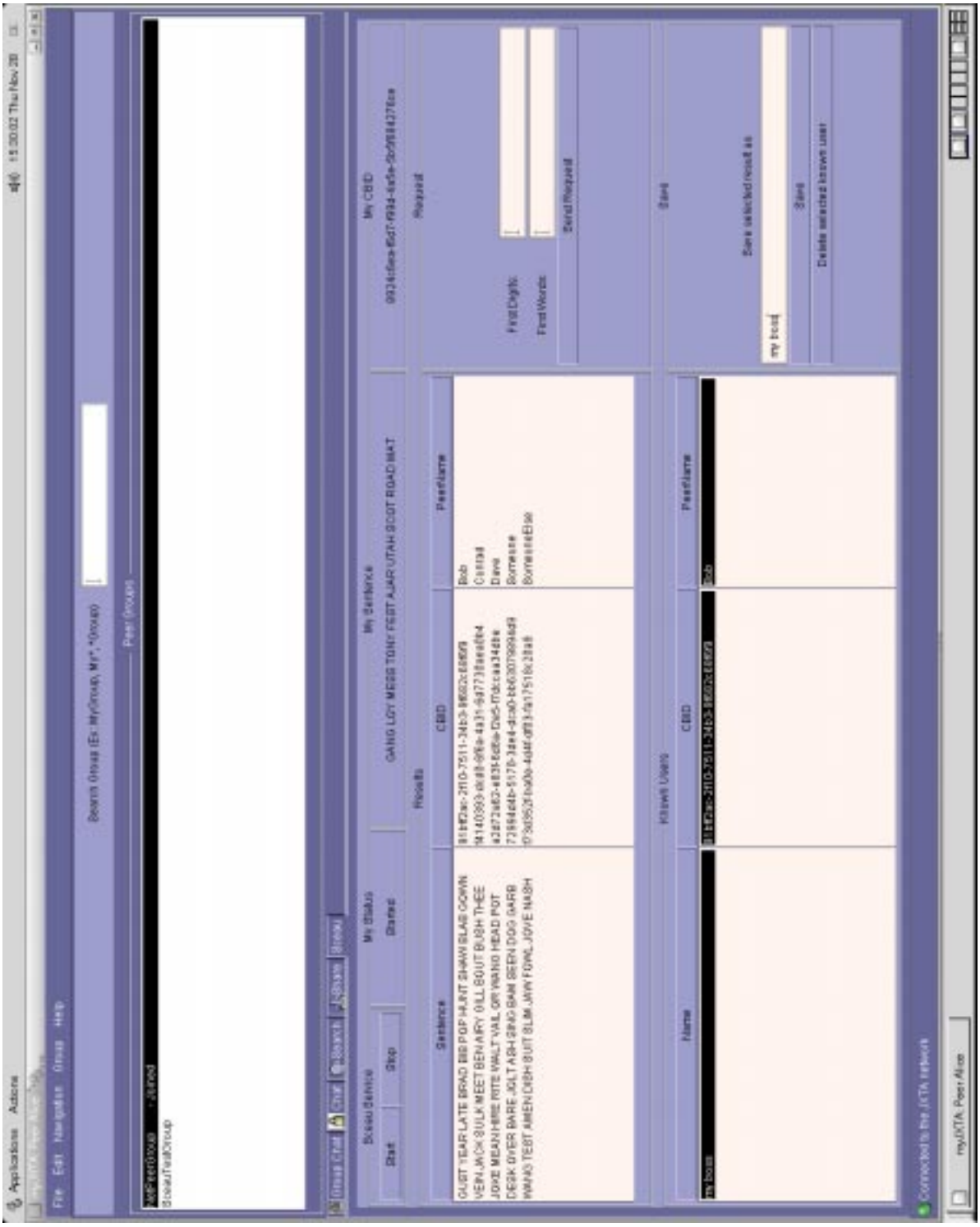

Figure A.1: *The Sceau Service panel.*

The functions of each panel contained in the Sceau tab are described below.

A.2.1 The *Sceau Service* panel

This panel contains the button to start and stop the Sceau service. The Sceau service is stopped by default and one needs to press the start button to turn it on.

A.2.2 the *My Status* panel

This panel contains the status of the Sceau service: 'started', 'stopped', 'waiting for results', etc. It's also used to display some error messages.

A.2.3 The *My Sentence* panel

This panel contains the sentence representing your own CBID.

A.2.4 The *My CBID* panel

This panel contains your own CBID.

A.2.5 The *Results* panel

This panel contains the results of he last query.

A.2.6 The *Request* panel

This panel contains an area where you can specify the first digits of the remote CBID you want to obtain. You can also set the first words of the sentence representing the CBID you want to obtain. Note that it is not possible to specify both digits and words simultaneously. At last there is a button (the '*Send Request*' button) to send out a request. A request can be sent even if no words and no digits are specified. The Sceau Service must be started to send a request. (Check the '*My Status*' panel).

A.2.7 The *Known Users* panel

This panel contains the list of users which have been identified and saved by means of the Sceau protocol. This list is used as a control access list for groups having the option '*control who can join the group*' enabled.

A.2.8 The *Save* panel

This panel contains an area to enter the name under which you want to save a selected result. The '*save*' button saves the selected results with the name specified in the text field above. At last the '*delete selected known user*' button delete a user previously saved.

A.2.9 The *Group* menu

In the '*Group*' menu, the window that pops up when '*Create New Group*' is clicked has been modified. There is a new option: '*control who can join the group*'. This option limits the group access to the peers being in your list of '*Known Users*'.

A.2.10 Example

Let's again consider the example of Alice, her boss Bob and a couple of Alice's co-workers. Each of them has launched MyJXTA and started the Sceau service by clicking on the '*start*' button in the '*Sceau Service*' panel.

Figure A.2: *The Sceau Service panel.*

Alice sends a request by clicking on the '*Send Request*' button of the '*Request Panel*'.

Figure A.3: *The Request panel.*

After a while a list of results is displayed in the '*Results*' panel. If there are too many results, Alice can send another request where she specifies the first words or digits of Bob.

Figure A.4: *The Results panel.*

Once she has the results displayed on her screen she asks Bob for his sentence. It is important to not rely on the peerName which is also displayed because PeerNames are not unique and thus not secure. (it can happens that two users have chosen the name Bob and it can even happen that the peer with the name Bob is not the one that Alice knows).

Bob just reads on his device his sentence and says it to Alice. Bob doesn't use the network for this but he goes and tells Alice: "my sentence is 'GUST YEAR LATE BRAD BIB POP HUNT SHAW BLAB GOWN'". This exchange can take place on the phone or on any other communication means as long as Alice is sure that she is really talking with Bob.

Alice identifies Bob's result to be the first line so she selects the line and in the '*Save*' panel, she writes "my Boss" and click on the '*Save*' button.

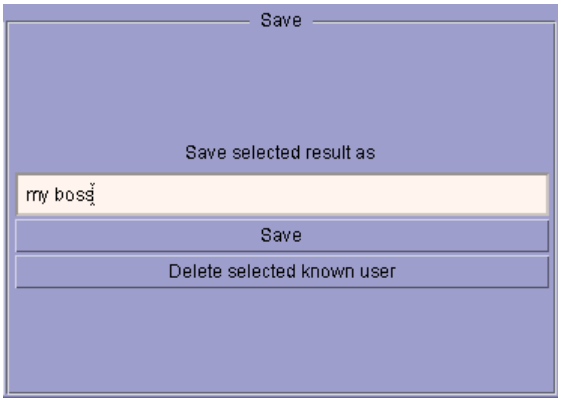

Figure A.5: *The Save panel.*

Bob is then added to the list of known users under the name "my Boss'. The name "my Boss" is only visible to Alice and nobody else. Currently this name is not used but one can easily imagine that in the Alice's '*Chat*' tab the name of Bob can be replaced with "my Boss".

Figure A.6: *The Known Users panel.*

Alice can then repeat this with each of her co-workers until her list of known users contains all her colleagues.

Alice then creates a group for only her boss and her co-workers. She click on the '*Group*' menu and then on '*Create New Group*'.

Figure A.7: *The menus.*

In the pop up window she enters the name of the group, "AliceWorkingGroup" for instance. She selects '*control who can join the group*' and click on the '*OK*' button.

Figure A.8: *The pop up window to create a group.*

After a while (it takes sometime to generate a new CBID for the group) "AliceWorkingGroup" appears in the group list. Then Alice joins the group by selecting her group in the group list and by clicking on '*Join Group*' in the '*Group*' menu. She is then asked for a name to use within this group. She enters "salesDirector" for instance and becomes the first member of the group.

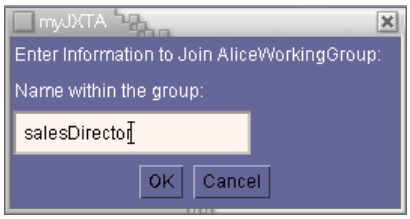

Figure A.9: *The pop up window to join a group.*

Note that the creator of the group doesn't have to be on the list of known users to enter the group.

Any of Alice's colleagues can now enter the group by following the same procedure: Selecting "AliceWorkingGroup" in the list of groups, clicking in the '*Group*' menu on 'Join Group' and finally by providing a unique name to use within the group.

Appendix B

Developer Guide

This guide is intended to help future developers to improve or build applications based on the CbJx infrastructure. For more detailed information one can look in the documentation in appendix E.

B.1 Dependancies

Project CbJx was started as an independant project. However its low level operations and the definition of a new ID type have led to modifications in the platform module itself. Therefore some classes developed in this project have directly been placed into the platform. These are the cases of the *net.jxta.impl.id.CBID* and *net.jxta.impl.endpoint.cbjx* packages. The JXTA Shell and MyJXTA/InstantP2P applications also rely on the platform and CbJx modules. All these dependancies are summarized in figure B.1.

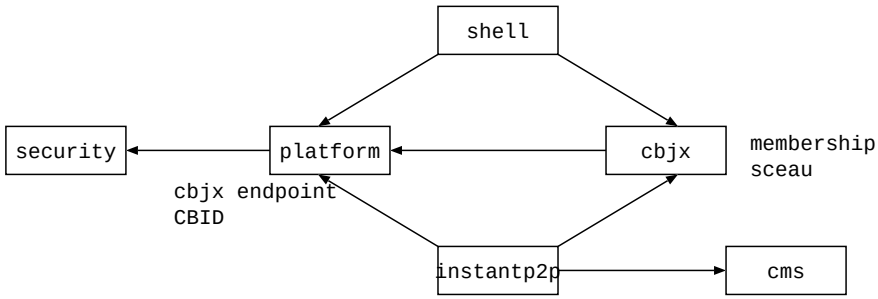

Figure B.1: *The dependancies between the JXTA modules used in project CbJx.*

B.2 The *Platform* module

This section presents the different packages which have been added or modified in the platform module. For a complete description of these packages see the documentation in appendix E (page 88) and a complete list of all the changes made in this module is available in appendix D (page 68).

B.2.1 The *net.jxta.impl.id.CBID* package

This package is inspired from the *net.jxta.impl.id.UUID* package. It contains classes to generate and handle CBIDs. Currently CBIDs are used only for PeerIDs and PeerGroupIDs.

B.2.2 The *net.jxta.impl.endpoint.cbjx* package

The CbJxEndpoint

This package contains the CbJxEndpoint which manipulates messages before they are sent. The most important method is *addCryptoInfo* which adds a special element into the message. This element contains the peer root certificate, the peerID and the signature of the message.

This method is static so that it can be called before sending a message on any endpoint protocols (tcp, http, etc). The message will then automatically be checked upon reception by the *CbJxInputFilter* of the destination peer . It allows someone to take advantage of the CbJx infrastructure without having to use the CbJxEndpoint. It can be used to improve performances by sending messages directly through the right endpoint protocol.

In the *CbJxEndpoint* this method is called both by the *CbJxMessenger* when it sends a message and by the *CbJxOutputFilter* which filters every outgoing propagate messages.

The message elements

The CbJxEndpoint uses several message elements to store information or to handle the operations it has to perform.

- The **cbjx:CryptoInfo** element contains a *CbJxMessageInfo* document. This document contains the PeerID and the root certificate of the sender, the signature of the message and the list of the message elements included in the signature. (Some message elements are not included in the signature because their content might change between the source and the destination peer. This is the case of the elements used for routing purposes, for instance). The name of elements which are never included in the signature are:

  jxta:EndpointDestinationAddress
  jxta:EndpointSourceAddress
  jxta:EndpointHeaderSrcPeer
  cbjx:Processing

- The **cbjx:Addresses** element contains a *CbJxMessageInfo* document. It contains the original source and destination addresses of the message. It's added by the *CbJxMessenger* when a unicast message is sent.

- The **cbjx:Processing** element is a temporary element which is added only when a message is processed by the *CbJxEndpoint*. It avoids to perform twice the same operation on a single message.
- The **cbjx:MsgValid** element is added by the method *checkCryptoInfo* which is called by the *CbJxInputFilter* and which checks that the information contained in the *cbjx:CryptoInfo* element is correct. If everything is valid then the *cbjx:CryptoInfo* element is removed and a *cbjx:MsgValid* element is added. This element contains a *CbJxMessageInfo* document containing the source peer ID (which has been verified) and the list of the message elements that were signed.

  This is the only element upper layers should interact with. For instance it can be used to retrieve the source of the message and which message elements were signed.

Usefull classes

This package contains also some classes which are useful to handle keys, certificates and signatures:

- **CbJxManager** handles certificates and keys storage/retrieval. It also generates new certificates.
- **CbJxUtils** provides useful functionalities such as the path to various elements of the PSE (Personal Security Environment) directory, certificates encoding/loading from streams, signature computation/vertification, conversions bytes/hex string.
- **CbJxDefs** contains the constants such as the names of the cryptographic algorithms and the names of the elements of the PSE directory.

The *CbJxMessageInfo* document is also located in this package.

B.2.3 The *net.jxta.impl.rendezvous* package

The Rendezvous service uses the *cbjx:MsgValid* element added by the *CbJxEndpoint* to check the identity of the sender and the integrity of the elements it has to deal with.

This is used to handle:

ConnectRequest
DisconnectRequest
RdvAdvReply
ConnectedPeerReply
ConnectedRdvAdvReply

(The *PingRequest* and *PingReply* are not secure).

B.2.4 The *net.jxta.impl.resolver* package

Like the Rendezvous service the Resolver uses the *cbjx:MsgValid* element added by the *CbJxEndpoint* to check that the contents of requests and responses were signed.

The Resolver provides the PeerID, source of the queries, to upper levels. With CbJx the Resolver makes sure that the source is really the sender of the message. If it is not the case the query is discarded. This is unfortunately not possible for the replies because they do not contain the same source field as queries do.

B.2.5 Core services

All the core services (including the two previous one and also standard unicast pipes) send messages to addresses of the form *cbjx://<PeerID>/service/params* instead of *jxta://<PeerID>/service/params*. This makes all messages go through the *CbJxEndpoint* and therfore allows the destination node to verify their integrity and authenticate the sender.

B.3 The *CbJx* module

The code of the CbJx project keep the same organization as the platform module: the *api* directory contains interfaces and common classes whereas the *impl* directory contains implementation specific classes. More details are also available in the documentation in appendix E (page 88).

B.3.1 The *jxta.cbjx.sceau* package

This package specifies general classes and interfaces. Its implementation can be found in the *jxta.cbjx.impl.sceau* package. For instance the Sceau service is described in the *jxta.cbjx.sceau.SceauService* interface and implemented in the *jxta.cbjx.impl.sceau.SceauServiceImpl* class.

The sceau service implementation uses pipes to achieve communication, (chapter 2) but one can develop his own implementation of this service as long as he respects the service interface.

This package is composed by the following classes and interfaces:

- **Dictionary**: The OTP (One Time Password) dictionary. This class can be used to translate bits into words and vice versa (a word into bits). A word is an 11 bits value which is represented as an integer. This class contains also some methods which transform the 128 bits (represeneted as 2 long of 64 bits) of a CBID into a set of words or sentence.

- **SceauService**: This interface defines the Sceau service. This service contains methods to send requests. A listener mechanism is used to notify the application when a response has been received. The listener registration is done when the query is sent and the Sceau service provides a method to remove a listener when the application doesn't need to be notified anymore.

- **SceauResponseListener**: The class which needs to be notified when a response has been received must implement this interface. The method *processSceauResponse* is called when a response is received.
- **SceauResponseEvent**: This class represents the event which is generated when a response is received by the Sceau service. It contains the PeerAdvertisement received in the response. This event is transmitted by the Sceau service to the appropriate listener (if any).

The *jxta.cbjx.impl.sceau* package contains the implementation of the *SceauService* and of the *SceauResponseEvent*.

The Sceau service implementation uses pipes to establish communication. A propagate pipe is used to send queries to all the listening peer whereas unicast pipes are used to send responses.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jxta:PipeAdvertisement>
<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
  <Id>
    urn:jxta:uuid-50A97F5270D74CCEA26FBB91552108EB
    B837838FA1474C50B9AB897306AB11A204
  </Id >
  <Type >
    JxtaPropagate
  </Type >
  <Name >
    Sceau Server Pipe
  </Name >
</jxta:PipeAdvertisement>
```

Figure B.2: *The XML representation of the pipe advertisement used to transmit queries.*

B.3.2 The *jxta.cbjx.impl.membership* package

The CbJx Project provides three membership services. They are mostly intended as examples and not as practical membership services. One should use them in order to develop his own membership service according to his needs.

As a peer must contact the group membership service (or more exactly an instance of the membership service running into the group) a way to communicate between the peer membership service and the group membership service is required. In those three membership services it is done using the Resolver service. But one can implement other communications channels such as pipes, for instance.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jxta:PipeAdvertisement>
<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
  <Id>
    urn:jxta:uuid-50A97F5270D74CCEA26FBB91552108EB
    B837838FA1474C50B9AB897306AB11A204
  </Id >
  <Type >
    JxtaUnicast
  </Type >
  <Name >
    Sceau Client Pipe
  </Name >
</jxta:PipeAdvertisement>
```

Figure B.3: *The XML representation of the pipe advertisement used to transmit responses.*

The *NullCbJxMembershipService*

It is an example on how certificates are issued when a peer group is joined. Rather than being seen as a useful membership service it must be considered as a base or general pattern that one can use to implement his own membership. (The two other membership services are inspired from this one but they each improves the service's functionalities).

When a peer calls the *apply* method a resolver query message is sent to the group membership service (actually the destination address is set to "*null*" which propagates the query into the whole group). The peer who receives the query replies to it by sending back a *CertListDocument* containing a list of certificates.

Then the peer needs to call the *join* method of the membership service which checks the validity of the list and returns a *Credential* document containing the list of certificates.

In this case the resolver query is not important because access is granted to everyone without any further verifications. Therefore the request consists only of the *AuthenticationCredential* document.

The *BootstrapMembershipService*

Based on the *NullCbJxMembershipService* it provides a simple access list control.

The request is still the *AuthenticationCredential* with the same elements. The peer receiving the request checks if it has the requesting peer's CBID in its list of authenticated peers. If the peer knows the requesting CBID it delivers a *CertList-Document* containing a list of certificates like in the *NullCbJxMembershipService*.

In these two membership services delegation is enabled which means that any members are allowed to admit new peers into the group. In the present case any peers known by at least one member of the group can join the group.

The *UniqueNameMembershipService*

This membership service is built on the same pattern as the previous ones. Like the *BootstrapMembershipService* it relies on an access list of known peers which is elaborated using the sceau protocol. However the delegation is disabled and the only peer acting as a group controller (which can admit new peers in the group) is the peer which has created the group. Therefore a peer must be on the access list of the group controller to be able to enter the group.

The extra feature provided by these membership service is the support for unique names within the group. This is done by the following mechanism:

When a peer sends a resolver query to join the group it adds a *UniqueNameRequest* into its *AuthenticationCredential* (in the optional *Identity* element). The *UniqueNameRequest* contains the name the peer would like to use in the group.

Upon reception of the request the group controller checks if it has this peer on its access list but it also checks if the requested name is not already used within the group. If these conditions are met it issues the same list of certificates as before and one SPKI certificate which allows the requesting peer to use the requested name within the group.

The SPKI certificate authorizing to use a given name is also added within the credential and can be used by the peer for whatever it wants. Currently there are no applications or services using such certificates.

B.3.3 The *jxta.cbjx.doc* package

As the *jxta.cbjx.sceau* package, the implementation of the classes of this package are located in *jxta.cbjx.impl.doc*.

All classes represent documents which can be rendered as XML.

CertListDocument

This document contains a list of certificates which is used to prove membership within a group.

```
<?xml version="1.0" encoding="UTF-8"?>
<CertList>
  <X509Cert>. . . . </X509Cert >
  <X509Cert>. . . . </X509Cert >
   . . . .
  <SPKICert >. . . . </SPKICert >
  <SPKICert >. . . . </SPKICert >
   . . . .
</CertList>
```

Figure B.4: *The XML representation of a CertListDocument.*

- **X509Cert**: This element contains an X.509 certificate Base64 encoded. The document can contain as many *X509Cert* element as necessary. The first X.509 certificate is always the group certificate.
- **SPKICert**: This element contains an SPKI certificate Base64 encoded. The document can contain as many *SPKICert* element as necessary.

UniqueNameRequest

This document is the request used by the *UniqueNameMembershipServicce* when a peer wants to join a group.

```
<?xml version="1.0" encoding="UTF-8"?>
<UniqueNameRequest>
  <RequestedName>. . . . </RequestedName >
  <PeerCert>. . . . </PeerCert >
  <Signature>. . . . </Signature >
</UniqueNameRequest>
```

Figure B.5: *The XML representation of a UniqueNameRequest.*

- **RequestedName**: This element contain the name that the peer would like to use in the group.
- **PeerCert**: The root certificate of the peer Base64 encoded.
- **Signature**: The signature of this document.

UniqueNameResponse

This document is the response used by the *UniqueNameMembershipService* when a peer wants to join a group.

```
<?xml version="1.0" encoding="UTF-8"?>
<UniqueNameResponse>
  <Membership>. . . . </Membership >
  <Name>. . . . </Name >
</UniqueNameResponse>
```

Figure B.6: *The XML representation of a UniqueNameResponse.*

- **Membership** This element contains a *CertListDocument* with the list of certificates proving membership within the group.
- **Name** This element contains a SPKI certificate authorizing the peer to use the specified name within the group.

SceauDocument

This document is used by the Sceau service to represent both queries and responses.

```
<?xml version="1.0" encoding="UTF-8"?>
<SceauDocument>
  <Type>. . . . </Type >
  <ID >. . . . </ID >
  <PipeAdvertisement >. . . . </PipeAdvertisement >
  <BitsNumber >. . . . </BitsNumber >
  <BitsValue >. . . . </BitsValue >
  <PeerAdvertisement >. . . . </PeerAdvertisement >
</SceauDocument>
```

Figure B.7: *The XML representation of a SceauDocument.*

- **Type**: Specify if this document is a request or a response.
- **ID**: Used to match responses with the corresponding query.
- **PipeAdvertisement**: The advertisement of the pipe to use to send a response.
- **BitsNumber**: The number of the specified bits in the request. The syntax is ' *nb_most_significant_bits#nb_least_significant_bits*'. This is due to the way sentences are generated.
- **BitsValue**: The specified bits. The bits are written as a numeric string (base 10) and the synatx is similar as in *BitsNumber*: '*most_significant_bits#least_significant_bits*'.
- **PeerAdvertisement**: The PeerAdvertisement of the peer sending this document.

B.4 The Personal Security Environment

CbJx uses self-generated certificates which are stored into the PSE (Personal Security Environment) directory. This direcory is located into the *.jxta* directory which is created in the directory where the JXTA application was launched. Project CbJx adds a bunch of new items into this directory as shown in figure B.8.

- **etc/passwd** contains the user's encrypted password.
- **client/peer.phrase** contains the randomly-genrated passphrase encrypted with the user's password.

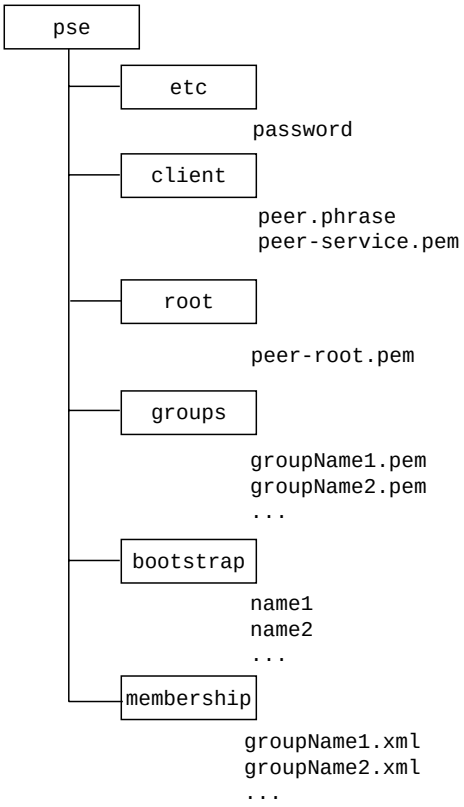

Figure B.8: *The structure of the pse directory.*

- **client/peer-service.pem** contains the peer's X509 certificate. This certificates is generated by the peer itself because it acts as its own certificate authority (CA). The appended private key is encoded with the passphrase stored in client/peer.phrase.
- **root/peer-root.pem** contains the X.509 certificate of the peer's CA. As the peer acts as its own CA this certificate is also self-generated and is refered as the peer root certificate. The peer's CBID is a hash of this certificate. The appended private key is encoded with the passphrase stored in client/peer.phrase.
- **groups/groupName.pem** contains group root certificates which are very similar to the peer root certificates. The CBID of the group is the hash of this certificate. The appended private key is encoded with the passphrase stored in client/peer.phrase.
- **membership/groupName.xml** contains the list of certificates which proves the peer's membership. This document contains Base64 encoded certificates and is rendered as an xml document.
- **bootstrap/localy-unique-name** contains an SPKI certificate which binds a localy unique name chosen by the user to the CBID of a remote peer identified with the Sceau protocol.

The authorization certificates are SPKI certificates. They are handled by the *jsdsi* package created by the MIT (Massachussets Institute of Technology). The version used in CbJx has been slightly modified to keep only required classes.

B.5 The JXTA Shell

A classic in the JXTA applications. Usefull to expirements JXTA services, it is the first application which uses the CbJx infrastructure. The first implemented command was the '*createGroup*' command. Similar to '*mkadv -g*', it creates peergroup advertisements using one of the three CbJx membership services.

The '*join*' command was also modified in order to instantiate and join groups created with the '*createGroup*' command.

A set of new commands was also added, the most important one being '*getCBID*'. This command runs the Sceau service in order for a user to authenticate another peer's CBID. It creates a new group (which do not use CBID) where the sceau service is running. This special group is hard-coded into the command and is instantiated and joined with '*getCBID -l*'.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jxta:PGA>
<jxta:PGA xmlns:jxta="http://jxta.org">
  <GID>
    urn:jxta:uuid-50A97F5270D74CCEA26FBB91552108EB02
  </GID >
  <MSID >
    urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE00000001
    2E7359516BE747E69793A5E55EFD96DB06
  </MSID >
  <Name >
    SceauTestGroup   </Name >
  <Desc >
    A peer group to test the sceau service
  </Desc >
</jxta:PGA>
```

Figure B.9: *The XML representation of the PeerAdvertisement of the group where the Sceau service is running.*

The Sceau service is used to send request and received replies. The replies are stored in a Vector which is itself stored in a Shell variable called *sceauResults*. This variable is used by another command '*save*' which issues a SPKI certificate associating the result's CBID to a given name.

B.6 MyJXTA

MyJXTA (also known as the InstantP2P) is mostly intended as a window of the JXTA technology. A new application was added into this window: the Sceau application. This application is a graphical version of the shell command '*getCBID*'. The *net.jxta.instantp2p.sceau* package contains two classes: *Sceau* and *SceauApplication*. *SceauApplication* contains the GUI (Graphical User Interface) whereas *Sceau* contains the "engine". It uses the same group as the '*getCBID*' shell command to run the Sceau service. Therefore the Shell application and MyJXTA are able to communicate through the Sceau service. (i.e. when a request is sent from one of these applications responses might be received from both peers running the JXTAShell and peers running myJXTA).

The mechanisms to create and join a new Group were also modified in order to works with the *UniqueNameMembershipService*.

For more information on those applications refer to the user guide in appendix A.

Appendix C

Licenses

C.1 The JXTA License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment:

   "This product includes software developed by the Sun Microsystems, Inc. for Project JXTA."

   Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear.

4. The names "Sun", "Sun Microsystems, Inc.", "JXTA" and "Project JXTA" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact Project JXTA at http://www.jxta.org.

5. Products derived from this software may not be called "JXTA", nor may "JXTA" appear in their name, without prior written permission of Sun.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL SUN MICROSYSTEMS

OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software consists of voluntary contributions made by many individuals on behalf of Project JXTA. For more information on Project JXTA, please see <http://www.jxta.org/>.

This license is based on the BSD license adopted by the Apache Foundation.

C.2 The JSDSI license

Copyright 2002 Massachusetts Institute of Technology

Permission to use, copy, modify, and distribute this program for any purpose and without fee is hereby granted, provided that this copyright and permission notice appear on all copies and supporting documentation, the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the program without specific prior permission, and notice be given in supporting documentation that copying and distribution is by permission of M.I.T. M.I.T. makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

JSDSI is the package used to handle SPKI certificate and was developed at the MIT (Massachussets Institute of Technology).

Appendix D

List of the *Platform* Modifications

This chapter presents the modifications made to the platform modules in the CbJx Project. The version of the platform presented here is the stable built of 09-24, 2002.

D.1 New Classes

D.1.1 Package *net.jxta.impl.endpoint.cbjx*

- **CbJxEndpoint.java**: The CbJx endpoint adds cryptographic information into outgoing messages and checks incoming ones.
- **CbJxMessageInfo**: The document containing the information added into messages by the *CbJxEndpoint*.
- **CbJxDefs**: It contains the constants used to handle CBIDs and certifiates.
- **CbJxManager**: Creates, stores and retrieves certificates.

  SPKI certificates are directly generated using the jsdsi package (the classes of this package are stored in the *miniJSDSI.jar* in the platform's *lib* directory).

  X.509 certificates are created using the *net.jxta.impl.endpoint.tls.PeerCerts* which was modified.
- **CbJxUtils**: Useful classes to manipulate certificates, keys and signatures. Base64 encoding and decoding operations are done with the *COM.claymore-systems.cert.WrappedObject* class. This class is located in *cryptix32.jar* in the platform's *lib* directory.

D.1.2 Package *net.jxta.impl.id.CBID*

This package is similar to *net.jxta.impl.id.UUID*. It supports only CBIDs for *PeerID* and *PeerGroupID*.

D.2 Modified Classes

D.2.1 *net.jxta.id.IDFactory*

lines 174 to 184

A new method to generate a PeerGroupID. The group name is provided because it is needed to generate the group root certificate.

```
/**
 *  Creates a new PeerGroupID Instance. A new random peer group id
 *  will be generated.
 *
 *  @since JXTA 1.0
 *  @see net.jxta.peergroup.PeerGroupID
 *
 *  @param groupName the name of the group
 *  @return the newly created ID.
 **/
PeerGroupID newPeerGroupID( String groupName );
```

lines 284 to 287

The identifier of the CBID format.

```
/**
 *  Identifies the ID format to use when creating new CBID instances.
 **/
private String cbidNewInstances = null;
```

lines 296 to 317

Adds the CBID format to the ID instances. Initializes the String cbidNewInstances.

```
/**
 *  Registers the pre-defined set of ID sub-classes so that
 *  this factory can construct them.
 *
 *  @since JXTA 1.0
 *
 *  @return booelan true if at least one of the ID sub-classes
 *  could be registered otherwise false.
 **/
private boolean doRegisterIDTypes()  {
  // FIXME    20010520    bondolo@jxta.org
  //This should use a config.properties property as documented.
  final String idInstances = "net.jxta.id.jxta.IDFormat "+
       "net.jxta.impl.id.UUID.IDFormat "+
```

```
        "net.jxta.impl.id.unknown.IDFormat "+
       "net.jxta.impl.id.CBID.IDFormat";
    // FIXME    20010520    bondolo@jxta.org
    //This should use a config.properties property as documented.
    idNewInstances = new String( "uuid" );
    cbidNewInstances = new String( "cbid" );

    return factory.registerFromString( idInstances );
  }
```

lines 522 to 540

Creates a new PeerID using the CBID format.

```
  /**
   *  Creates a new PeerID Instance. A new random peer id will be
   *  generated. The PeerID will be a member of the provided group.
   *
   *  @since       JXTA  1.0
   *  @see net.jxta.peer.PeerID
   *
   *  @param groupID    the group to which this PeerID will belong.
   *  @return the newly created ID.
   **/
  public static PeerID newPeerID( PeerGroupID groupID ) {
    if( ! factory.loadedString )
      factory.loadedString = factory.doRegisterIDTypes();

    Instantiator instantiator =
      (Instantiator)factory.getInstantiator(factory.cbidNewInstances);

    return instantiator.newPeerID( groupID );
  }
```

lines 561 to 579

Creates a new *PeerGroupID* using the CBID format. A group name is provided because it is used to generate the group root certificate.

```
  /**
   *  Creates a new PeerGroupID Instance. A new random peer group id
   *   will be generated.
   *
   *  @since JXTA 1.0
   *  @see net.jxta.peergroup.PeerGroupID
   *
   *  @param groupName the name of the group
```

```
   *  @return the newly created ID.
   **/
  public static PeerGroupID newPeerGroupID( String groupName ) {
    if( ! factory.loadedString )
       factory.loadedString = factory.doRegisterIDTypes();

    Instantiator instantiator =
      (Instantiator)factory.getInstantiator(factory.cbidNewInstances);

    return instantiator.newPeerGroupID( groupName );
  }
```

D.2.2 *net.jxta.id.jxta.Instantiator*

lines 177 to 182

The creation of a *PeerGroupID* with a group name provided in arguments.

```
  /**
   * {@inheritDoc}
   **/
  public PeerGroupID newPeerGroupID(String groupName) {
    throw new ProviderException( "unsupported id type" );
  }
```

D.2.3 *net.jxta.peergroup.PeerGroup*

lines 405 to 410

The *ModuleClassID* for the CbJx Endpoint.

```
  /**
   * Well known module class identifier: CbjxEndpoint
   */
  public static final ModuleClassID cbjxEndpointClassID =
    (ModuleClassID)IdMaker.mkID("DeadBeefDeafBabaFeedBabe0000000F"
                        + "05" );
```

lines 555 to 561

The *ModuleSpecID* for the CbJx Endpoint

```
  /**
   * Well known protocol: the CbJxEndpoint protocol
   */
  public static final ModuleSpecID refCbJxEndpointSpecID =
    (ModuleSpecID)IdMaker.mkID( "DeadBeefDeafBabaFeedBabe0000000F"
                        + "01"
                        + "06" );
```

D.2.4 *net.jxta.impl.endpoint.tls.PeerCerts*

lines 143 to 157

Creates and saves an X.509 certificate for a group. The group private key is encryted using the passphrase and appended at the end of the file.

```
public static void genGroupRootCert(String fileName,
                                     String groupName,
                                     byte[] passphrase)
throws IOException {
  // generate and save cert

  SSLDebug.debug(SSLDebug.DEBUG_JXTA, "Generating root cert ...");

  IssuerInfo info = genCert(fileName, groupName, null);

  info.passwd = passphrase;

  // Append private key to generated cert
  appendPrivateKey(info, fileName);
}
```

D.2.5 *net.jxta.impl.id.UUID.IDFormat*

lines 195 to 221

Translates a *PeerGroupID* generated from a CBID into a *PeerGroupID* generated from a UUID.

This is done by creating a new UUID with the bits of the CBID. This is possible because CBIDs (genrated from a secure hash SHA-1) present the same uniqueness properties as UUIDs.

This method avoids to rise a *ClassCastException* when a *PipeID*, or *CodatID*, etc is created within a peer group which ID is a CBID.

```
/**
 * translate a CB peergroup ID into a UU peergroup ID
 * This is possible because CBID presents the same uniqueness
 * properties as UUID
 *
 * @param groupID the groupID to translate (if it's not an
 *        instance of jxta.cbjx.impl.id.CBID.PeerGroupID
 *        then it's returned without any changes)
 * @return the translated peergroupID
 */
static net.jxta.peergroup.PeerGroupID translateToUUID(
   net.jxta.peergroup.PeerGroupID groupID)
{
```

```
     if(groupID instanceof net.jxta.impl.id.CBID.PeerGroupID)
     {
net.jxta.impl.id.CBID.CBID cbid =
  ((net.jxta.impl.id.CBID.PeerGroupID)groupID)
            .getPeerGroupCBID();
  long mostSig = cbid.getMostSignificantBits();
  long leastSig = cbid.getLeastSignificantBits();
  UUID uuid = new UUID(mostSig, leastSig);
  return new PeerGroupID(uuid);
    }
    else
    {
       return groupID;
    }
  }
```

D.2.6 *net.jxta.impl.id.UUID.Instantiator*

lines 184 to 201

Before the creation of a new *CodatID* the *PeerGroupID* is translated into an instance of *net.jxta.impl.id.UUID.PeerGroupID*.

```
  /**
   *  Creates a new CodatID Instance. A new random Codata ID
   *  is created for the provided Peer Group. This type of
   *  CodatID can be used as a canonical reference for dynamic
   *  content.
   *
   *  @since JXTA 1.0
   *  @see net.jxta.codat.CodatID
   *
   *  @param groupID    the group to which this content will belong.
   *  @return the newly created ID.
   **/
  public net.jxta.codat.CodatID newCodatID(
final net.jxta.peergroup.PeerGroupID groupID ) {

    net.jxta.peergroup.PeerGroupID  gid =
      (net.jxta.peergroup.PeerGroupID)
IDFormat.translateFromWellKnown( groupID );

    PeerGroupID peerGroupID =
      (PeerGroupID)IDFormat.translateToUUID(gid);

    return new CodatID( peerGroupID );
  };
```

lines 203 to 228

Before the creation of a new *CodatID* the *PeerGroupID* is translated into an instance of *net.jxta.impl.id.UUID.PeerGroupID*.

```
  /**
   * Creates a new CodatID Instance. A new random Codata ID is
   * created for the provided Peer Group and contains a hash
   * value for the Codat data. This type of Codat ID is most
   * appropriate for static content. By including a hash value
   * this form of Codat ID provides greater assurance of the
   * canonical property of IDs. It also allows the document
   * content returned when this ID is used to be verified to
   * ensure it has not been altered.
   *
   * @param  groupID The group to which this ID will belong.
   * @param  in  The InputStream from which the content hash
   * is calculated.
   *  The stream is read until EOF and then closed.
   *
   *  @since JXTA 1.0
   *  @see net.jxta.codat.CodatID
   *
   *  @return the newly created ID.
   *  @throws IOException I/O Error reading document
   **/
  public net.jxta.codat.CodatID newCodatID(
                     final net.jxta.peergroup.PeerGroupID groupID,
              java.io.InputStream in)
  throws IOException {
    net.jxta.peergroup.PeerGroupID gid =
      (net.jxta.peergroup.PeerGroupID)
IDFormat.translateFromWellKnown( groupID );

    PeerGroupID peerGroupID =
      (PeerGroupID)IDFormat.translateToUUID(gid);

    return new CodatID( peerGroupID, in );
  };
```

lines 230 to 246

Before the creation of a new *PeerID* the *PeerGroupID* is translated into an instance of *net.jxta.impl.id.UUID.PeerGroupID*.

```
  /**
   *  Creates a new PeerID Instance. A new random peer id
```

```
   *  will be generated. The PeerID will be a member of
   *  the provided group.
   *
   *  @since       JXTA  1.0
   *  @see net.jxta.peer.PeerID
   *
   *  @param groupID the group to which this PeerID will belong.
   *  @return the newly created ID.
   **/
  public net.jxta.peer.PeerID newPeerID(
                final net.jxta.peergroup.PeerGroupID groupID ) {
     net.jxta.peergroup.PeerGroupID  gid =
       (net.jxta.peergroup.PeerGroupID)
 IDFormat.translateFromWellKnown( groupID );
     PeerGroupID peerGroupID =
       (PeerGroupID)IDFormat.translateToUUID(gid);

     return new PeerID( peerGroupID );
  };
```

lines 261 to 273

Creates a new *PeerGroupID* with a given name. Since UUIDs do not require the name of the group, the code remains the same as the *newPeerGroupID()* method without any arguments.

```
  /**
   *  Creates a new PeerGroupID Instance. A new random peer group id
   *   will be generated.
   *
   *  @since JXTA 1.0
   *  @see net.jxta.peergroup.PeerGroupID
   *
   *  @param groupName the name of the group
   *  @return the newly created ID.
   **/
  public net.jxta.peergroup.PeerGroupID newPeerGroupID(
                                String groupName){
    return new PeerGroupID( );
  };
```

lines 275 to 291

Before the creation of a new *PipeID* the *PeerGroupID* is translated into an instance of *net.jxta.impl.id.UUID.PeerGroupID*.

```
  /**
```

```
   *  Creates a new PipeID Instance. A new random pipe id will be
   *  generated.
   *
   *  @since JXTA 1.0
   *  @see net.jxta.pipe.PipeID
   *
   *  @param groupID  the group to which this Pipe ID will belong.
   *  @return the newly created ID.
   **/
  public net.jxta.pipe.PipeID newPipeID(
       final net.jxta.peergroup.PeerGroupID groupID ) {
     net.jxta.peergroup.PeerGroupID  gid =
       (net.jxta.peergroup.PeerGroupID)
IDFormat.translateFromWellKnown( groupID );

     PeerGroupID peerGroupID =
       (PeerGroupID)IDFormat.translateToUUID(gid);

     return new PipeID( peerGroupID );
  }
```

lines 293 to 320

Before the creation of a new *PipeID* the *PeerGroupID* is translated into an instance of *net.jxta.impl.id.UUID.PeerGroupID*.

```
  /**
   *  Creates a new PipeID Instance. A new pipe id will be
   *  generated with the provided seed information. The Pipe ID
   *  will be a member of the provided group. The seed information
   *  should be at least four bytes in length, though longer
   *  values are better. This varient of Pipe ID allows you to
   *  create "Well-known" pipes within the context of diverse
   *  groups. This can be useful for common services that need to
   *  do discovery without advertisements or for network
   *  organization services. Because of the potential for ID
   *  collisions and the difficulties with maintaining common
   *  service interfaces this form of Pipe ID should be used
   *  sparingly.
   *
   *  @since       JXTA  1.0
   *  @see net.jxta.pipe.PipeID
   *
   *  @param groupID  the group to which this Pipe ID will belong.
   *  @param seed The seed information which will be used in
   *  creating the pipeID.
   *  @return the newly created ID.
```

```
   **/
  public net.jxta.pipe.PipeID newPipeID(
      final net.jxta.peergroup.PeerGroupID groupID,
      byte [] seed ) {
    net.jxta.peergroup.PeerGroupID  gid =
      (net.jxta.peergroup.PeerGroupID)
IDFormat.translateFromWellKnown( groupID );
    PeerGroupID peerGroupID =
      (PeerGroupID)IDFormat.translateToUUID(gid);

    return new PipeID( peerGroupID, seed );
  }
```

D.2.7 *net.jxta.impl.id.unknown.Instantiator*

lines 144 to 146

Creates a new *PeerGroupID* with a group name provided as a parameter. Since it's the 'unknown' *ID* type, *PeerGroupID* creation is not supported.

```
  public net.jxta.peergroup.PeerGroupID newPeerGroupID(String Name){
    throw new ProviderException( "unsupported id type" );
  }
```

D.2.8 *net.jxta.impl.peergroup.Configurator*

line 115

The *CbJxManager* class which is in charge of handling keys and certificates. This class also needs to be initialized at boot time.

```
  import net.jxta.impl.endpoint.cbjx.CbJxManager;
```

line 218 to 220

The *PeerID* is generated after the creation of the peer root certificate and thus it cannot be set in the advertisement before the certificate generation.

```
  //Add the peerID when the root cert has been generated
  advertisement.setPeerID(
      IDFactory.newPeerID(PeerGroupID.worldPeerGroupID));
```

line 291 to 298

The *CbJxManager* is initialized.

```
  try
  {
     net.jxta.impl.endpoint.cbjx.CbJxManager.init(password);
```

```
}
catch(Exception e)
{
   platformCanceled = true;
}
```

lines 841 to 842

The *PeerID* is set later on into the peer advertisement.

```
//advertisement.setPeerID(
//IDFactory.newPeerID(PeerGroupID.worldPeerGroupID));
```

D.2.9 *net.jxta.impl.peergroup.Platform*

lines 217 to 221

Adds the *CbJxEndpoint* into the list of protocols.

```
moduleAdv =
  mkImplAdvBuiltin(refCbJxEndpointSpecID,
  "net.jxta.impl.endpoint.cbjx.CbJxEndpoint",
  "Reference Implementation of the CbJx Endpoint");

protos.put(cbjxEndpointClassID, moduleAdv);
```

D.2.10 *net.jxta.impl.pipe.NonBlockingOutputPipe*

lines 93 to 106

Uses the cbjx endpoint instead of the jxta endpoint. The returned endpoint address starts with '*cbjx://*' instead of '*jxta://*'.

```
private EndpointAddress mkAddress(String destPeer, String pipeId) {
  try {
    PeerID asID = (PeerID)
       IDFactory.fromURL(IDFactory.jxtaURL(destPeer));
    String asString = "cbjx://" + asID.getUniqueValue().toString();

    EndpointAddress addr = endpoint.newEndpointAddress(asString);
    addr.setServiceName("PipeService");
    addr.setServiceParameter(pipeId);
    return addr;
  } catch (Exception e) {
     if (LOG.isEnabledFor(Priority.ERROR))
         LOG.error("invalid PeerID string: " + destPeer);
     return null;
  }
}
```

D.2.11 *net.jxta.impl.rendezvous.PeerConnection*

lines 289 to 302

Uses the cbjx endpoint instead of the jxta endpoint. The returned endpoint address starts with '*cbjx://*' instead of '*jxta://*'.

```
/**
 * Build an EndpointAddress the EndpointRouter can use.
 */
private EndpointAddress mkAddress( ID destPeer) {
  try {
    String asString = "cbjx://" +
      destPeer.getUniqueValue().toString();

    EndpointAddress addr =
      endpoint.newEndpointAddress(asString);

    return addr;
  } catch (Exception e) {
     if (LOG.isEnabledFor(Priority.WARN))
         LOG.warn("Invalid peerID " + destPeer );
     return null;
  }
}
```

D.2.12 *net.jxta.impl.rendezvous.RendezVousServiceImpl*

lines 82 to 84

The classes to handle cryptographic information within a message.

```
import net.jxta.impl.endpoint.cbjx.CbJxMessageInfo;
import net.jxta.impl.endpoint.cbjx.CbJxEndpoint;
```

line 323

Uses the cbjx endpoint instead of the jxta endpoint. The endpoint address starts with '*cbjx://*' instead of '*jxta://*'.

```
localPeerAddr = "cbjx" + "://" + localPeerId.toString();
```

lines 831 to 836

Prints a debug line if the received message doesn't contain a '*cbjx:MsgValid*' element. (This element proves that the content of the message was signed by the sender).

```
  if(!msg.hasElement(CbJxEndpoint.validElementName))
  {
     if(LOG.isEnabledFor(Priority.DEBUG))
LOG.debug("demuxing insecure message");
  }
```

lines 975 to 976

Prints a debug line if the received message doesn't contain a '*cbjx:MsgValid*' element. (This element proves that the content of the message was signed by the sender).

```
  if(!message.hasElement(CbJxEndpoint.validElementName) &&
     LOG.isEnabledFor(Priority.DEBUG))
     LOG.debug("!! the message #" + msgId + " is not secure !!");
```

lines 1260 to 1273

Uses the cbjx endpoint instead of the jxta endpoint. The endpoint address starts with '*cbjx://*' instead of '*jxta://*'.

```
  /**
   *Description of the Method
   *
   * @param  destPeer  Description of Parameter
   * @return           Description of the Returned Value
   */
  private EndpointAddress mkAddress(ID destPeer) {

    String asString = "cbjx://" + destPeer.getUniqueValue().toString();

    EndpointAddress addr = endpoint.newEndpointAddress(asString);

    return addr;
  }
```

lines 1553 to 1554

Adds the cryptographic information into the message before it is sent.

```
  //add CBJX Message info
  msg = CbJxEndpoint.addCryptoInfo(msg);
```

In method '*private void processConnectRequest(Message msg)*', the message is discarded if the elements were not signed or ifthe message was not checked by the *CbJxEndpoint* (i.e. if there is no '*cbjx:MsgValid*' element in the message).

lines 1621 to 1629

```
  if(!msg.hasElement(CbJxEndpoint.validElementName))
  {
     if(LOG.isEnabledFor(Priority.DEBUG))
     {
  LOG.debug("not a secure message (no '" +
  CbJxEndpoint.validElementName + "' element)");
     }
     return;
  }
```

lines 1650 to 1678

```
  //check that the adv element was signed
  CbJxMessageInfo secure = null;
  try{
    MessageElement el = msg.getElement(CbJxEndpoint.validElementName);
    secure = new CbJxMessageInfo(el.getStream());
    String[] list = secure.getList();
    boolean signed = false;
    for(int i=0; !signed && i < list.length; i++)
    signed = ConnectRequest.equals(list[i]);
    if(!signed)
    {
if(LOG.isEnabledFor(Priority.DEBUG))
   LOG.debug(ConnectRequest + " was not signed");
return;
    }
  }
  catch(Exception e){
    if(LOG.isEnabledFor(Priority.DEBUG))
       LOG.debug("failed to retrieve secure info");
    return;
  }

  //check that the peerAdv is really the one from the sender
  if(!secure.getSourceID().equals(adv.getPeerID().toString()))
  {
     if(LOG.isEnabledFor(Priority.DEBUG))
        LOG.debug("   not the peer adv of the sender");
     return;
  }
```

In method '*private void processDisconnectRequest(Message msg)*' the message is discarded if the elements were not signed or if the message

was not checked by the *CbJxEndpoint* (i.e. if there is no '*cbjx:MsgValid*' element in the message).

lines 1713 to 1719

```
if(!msg.hasElement(CbJxEndpoint.validElementName))
{
  if(LOG.isEnabledFor(Priority.DEBUG))
     LOG.debug("not a secure message (no '" +
CbJxEndpoint.validElementName + "' element)");
  return;
}
```

lines 1734 to 1763

```
//check that the diconnect req element was signed
CbJxMessageInfo secure = null;
try{
  MessageElement el = msg.getElement(CbJxEndpoint.validElementName);
  secure = new CbJxMessageInfo(el.getStream());
  String[] list = secure.getList();
  boolean signed = false;
  for(int i=0; !signed && i < list.length; i++)
signed = DisconnectRequest.equals(list[i]);
  if(!signed)
  {
if(LOG.isEnabledFor(Priority.DEBUG))
   LOG.debug(DisconnectRequest + " was not signed");
return;
  }
}
catch(Exception e){
  if(LOG.isEnabledFor(Priority.DEBUG))
     LOG.debug("failed to retrieve secure info");
  return;
}

//check that the peerAdv is really the one from the sender
if(!secure.getSourceID().equals(adv.getPeerID().toString()))
{
   if(LOG.isEnabledFor(Priority.DEBUG))
      LOG.debug("   not the peer adv of the sender");
    return;
}
```

In method '*private void processConnectedReply(Message msg)*' the message is discarded if the elements were not signed or if the message was not

checked by the *CbJxEndpoint* (i.e. if there is no '*cbjx:MsgValid*' element in the message)

lines 1780 to 1786

```
if(!msg.hasElement(CbJxEndpoint.validElementName))
{
  if(LOG.isEnabledFor(Priority.DEBUG))
     LOG.debug("not a secure message (no '" +
CbJxEndpoint.validElementName + "' element)");
     return;
}
```

lines 1790 to 1803

```
//check that the adv element was signed
CbJxMessageInfo secure = null;
String[] list = null;
boolean signed = false;
try{
  MessageElement el =
     msg.getElement(CbJxEndpoint.validElementName);
  secure = new CbJxMessageInfo(el.getStream());
  list = secure.getList();
}
catch(Exception e){
  if(LOG.isEnabledFor(Priority.DEBUG))
     LOG.debug("failed to retrieve secure info");
  return;
}
```

lines 1808 to 1820

```
for(int i=0; !signed && i < list.length; i++)
   signed = ConnectedRdvAdvReply.equals(list[i]);
if(!signed)
{
  if(LOG.isEnabledFor(Priority.DEBUG))
     LOG.debug(ConnectedRdvAdvReply + " was not signed");
}
else{
   try {
       adv = AdvertisementFactory.newAdvertisement(textXml, is);
   } catch (Exception e) {
   }
```

lines 1827 to 1832

```
if(!secure.getSourceID().equals(
     ((PeerAdvertisement)adv).getPeerID().toString()))
{
   if(LOG.isEnabledFor(Priority.DEBUG))
      LOG.debug("   not the peer adv of the sender");
   return;
}
```

lines 1860 to 1868

```
//check that the adv element was signed
signed = false;
for(int i = 0; !signed && i < list.length; i++)
  signed = ConnectedLeaseReply.equals(list[i]);
if(!signed){
  if(LOG.isEnabledFor(Priority.DEBUG))
     LOG.debug(ConnectedLeaseReply + " was not signed");
  return;
}
```

lines 1891 to 1898

```
signed = false;
for(int i = 0; !signed && i < list.length; i++)
   signed = ConnectedPeerReply.equals(list[i]);
if(!signed){
   if(LOG.isEnabledFor(Priority.DEBUG))
LOG.debug(ConnectedPeerReply + " was not signed");
   return;
}
```

In the method '*private void processRdvAdvReply(Message msg)*' the message is discarded if the elements were not signed or if the message was not checked by the *CbJxEndpoint* (i.e. if there is no '*cbjx:MsgValid*' element in the message).

lines 1943 to 1970

```
if(!msg.hasElement(CbJxEndpoint.validElementName))
{
   if(LOG.isEnabledFor(Priority.DEBUG))
LOG.debug("not a secure message (no '" +
CbJxEndpoint.validElementName + "' element)");
    return;
}
```

```
CbJxMessageInfo secure = null;
try{
  //check that the adv element was signed
  MessageElement el = msg.getElement(CbJxEndpoint.validElementName);
  secure = new CbJxMessageInfo(el.getStream());
  String[] list = secure.getList();
  boolean signed = false;
  for(int i=0; !signed && i < list.length; i++)
signed = RdvAdvReply.equals(list[i]);
  if(!signed)
  {
if(LOG.isEnabledFor(Priority.DEBUG))
   LOG.debug(RdvAdvReply + " was not signed");
return;
  }
}
catch(Exception e){
   if(LOG.isEnabledFor(Priority.DEBUG))
      LOG.debug("failed to retrieve secure info");
   return;
}
```

lines 1983 to 1989

```
//check that the peerAdv is really the one from the sender
if(!secure.getSourceID().equals(
    ((PeerAdvertisement)adv).getPeerID().toString()))
{
if(LOG.isEnabledFor(Priority.DEBUG))
   LOG.debug("   not the peer adv of the sender");
return;
}
```

D.2.13 *net.jxta.impl.resolver.ResolverServiceImpl*

lines 105 to 107

The classes to handle cryptographic information within a Message.

```
import net.jxta.impl.endpoint.cbjx.CbJxMessageInfo;
import net.jxta.impl.endpoint.cbjx.CbJxEndpoint;
```

lines 181 to 197

Uses the cbjx endpoint instead of the jxta endpoint. The returned endpoint address starts with '*cbjx://*' instead of '*jxta://*'.

```
private EndpointAddress mkAddress(String destPeer,
                                  String serv,
                                  String parm) {
  try {
     PeerID asID = (PeerID)
       IDFactory.fromURL(IDFactory.jxtaURL(destPeer));
     String asString = "cbjx://" + asID.getUniqueValue().toString();

     EndpointAddress addr = endpoint.newEndpointAddress(asString);
     addr.setServiceName(serv);
     addr.setServiceParameter(parm);
     return addr;
   }
   catch (Exception e) {
     if (LOG.isEnabledFor(Priority.WARN))
         LOG.warn("Invalid peerID string " + destPeer);
     throw new RuntimeException("Error creating Peer Address :" +
           e.toString());
   }
}
```

In method '*public void processIncomingMessage(Message message, EndpointAddress srcAddr, EndpointAddress dstAddr)*', the message is discarded if the elements are not signed by the sender and if the source PeerID of a request is not the sender of the message.

lines 513 to 514

```
if(LOG.isEnabledFor(Priority.DEBUG) &&
   !message.hasElement(CbJxEndpoint.validElementName))
      LOG.debug("!! the message is not secure !!");
```

lines 534 to 555

```
MessageElement elem =
   message.getElement(CbJxEndpoint.validElementName);
CbJxMessageInfo secure = new CbJxMessageInfo(elem.getStream());

//check that the element containing the query was signed
boolean signed = false;
String[] list = secure.getList();
for(int i = 0; !signed && i < list.length; i++)
  signed = list[i].equals(tagString);
if(!signed)
{
   if(LOG.isEnabledFor(Priority.DEBUG))
      LOG.debug(tagString + " was not signed !!");
```

```
    return;
}

//check that the request comes from the source
if(!doc.getSrc().equals(secure.getSourceID()))
{
    if(LOG.isEnabledFor(Priority.DEBUG))
        LOG.debug("the query was not sent by the sender of the message");
    return;
}
```

In method '*public void processIncomingMessage(Message message, EndpointAddress srcAddr, EndpointAddress dstAddr)*' (in the inner class *RecvDemux*), the message is discarded if the elements are not signed by the sender.

lines 619 to 623

```
if (!message.hasElement(CbJxEndpoint.validElementName)){
  if (LOG.isEnabledFor(Priority.DEBUG))
      LOG.debug("!! the message is not secure !! discard it");
  return;
}
```

Appendix E

Java Documentation

E.1 jxta.cbjx.doc.CertListDocument

Public abstract Class. Extends *java.lang.Object*

This class defines the XML document used to store certificates lists to prove membership into a group.

Constructor

public **CertListDocument** *()*

getDocument

public abstract net.jxta.document.Document **getDocument** *(net.jxta.document.MimeMediaType asMimeType)*

Returns a Document containing the certificate list's document tree

Parameter:*asMimeType* - the desired MIME type for the cert list rendering

Returns:*Document* - the Document containing the cert list's document tree

getSPKICerts

public jsdsi.Certificate[] **getSPKICerts** *()*

Retrieves the list of SPKI certificates

Returns:*jsdsi.Certificate[]* the list of SPKI Certificates

getX509Certs

public java.security.cert.X509Certificate[] **getX509Certs** *()*

Retrieves the list of X.509 certificates

Returns:*java.security.cert.X509Certificate[]* the list of X.509 certificates

setSPKICerts

public void **setSPKICerts** *(jsdsi.Certificate[] certs)*
Sets the list of SPKI certificates

Parameter:*certs* - the list of SPKI certificates

setX509Certs

public void **setX509Certs** *(java.security.cert.X509Certificate[] certs)*
Sets the list of X.509 certificates

Parameter:*certs* - the list of X.509 certificates

setSPKICert

public void **setSPKICert** *(jsdsi.Certificate cert)*
Adds an SPKI certificate at the end of the SPKI list

Parameter:*cert* - the SPKI certificate to append to the list

setX509Cert

public void **setX509Cert** *(java.security.cert.X509Certificate cert)*
Adds an X.509 certificate at the end of the X.509 list

Parameter:*cert* - the X.509 certificate to append to the list

verify

public boolean **verify** *(net.jxta.impl.id.CBID.*
CBID groupCBID, net.jxta.impl.id.CBID.CBID peerCBID, boolean delegation)
Verifies that the list of certificates proves membership within the group. The group root certificate is the first X509 certificate. Each list (SPKI and X509) must be well ordered. It must start with the group certificate followed by delegated peers certificates.
Example:

- 1st X509: the group root certificate
- 2nd X509: peer1's root certificate
- 3rd X509: peer2's root certificate
- 4th X509: your own root certificate
- 1st SPKI: authorization from the group to peer1
- 2nd SPKI: authorization from peer1 to peer2

- 3rd SPKI: authorization from peer2 to yourself

Parameter:*groupCBID* - the group's CBID

Parameter:*peerCBID* - the peer's CBID

Parameter:*delegation* - true if delegation must be checked

Returns:*boolean* - true if this document proves membership within the group

E.2 jxta.cbjx.doc.SceauDocument

Public abstract Class. Extends *java.lang.Object*

This class defines the XML document used by the Sceau service to represent queries and responses.

Constructor

public **SceauDocument** *()*

getDocument

public abstract net.jxta.document.Document **getDocument** *(net.jxta.document.MimeMediaType asMimeType)*

Returns a Document containing the information's document tree

Parameter:*asMimeType* - the desired MIME type for the document rendering

Returns:*Document* - the Document

getType

public java.lang.String **getType** *()*

Returns the type of the document (query or response).

Returns:*String* - the type of the document

getId

public int **getId** *()*

Returns the ID of the Document. IDs are used to match queries with their eventual responses.

Returns:*int* - the ID

getPipeAdvertisement

public net.jxta.protocol.PipeAdvertisement **getPipeAdvertisement** *()*

Returns the PipeAdvertisement to use to send an eventual response.

Returns:*PipeAdvertisement* - the PipeAdvertisement

getMostSignificantBits

public long **getMostSignificanBits** *()*
Returns the most significant bits of the requested CBID.

Returns:*long* - the most significant bits.

getLeastSignificantBits

public long **getLeastSignificanBits** *()*
Returns the least significant bits of the requested CBID.

Returns:*long* - the least significant bits.

getNbMostBits

public long **getNbMostBits** *()*
Returns the number of most significant bits of the requested CBID specified in the request.

Returns:*long* - the number of specified most significant bits.

getNbLeastBits

public long **getNbLeastBits** *()*
Returns the number of least significant bits of the requested CBID specified in the request.

Returns:*long* - the number of specified least significant bits.

getPeerAdvertisement

public net.jxta.protocol.PeerAdvertisement **getPeerAdvertisement** *()*
Returns the PeerAdvertisement of the sender.

Returns:*PeerAdvertisement* - the PeerAdvertisement

setType

public void **setType** *(java.lang.String type)*
Sets the type of the document (query or response).

Parameter:*type* - the type of the document.

setId

public void **setId** *(int id)*
Sets the id of the document.

Parameter:*id* - the id of the document.

setPipeAdvertisement

public void **setPipeAdvertisement** *(net.jxta.protocol.PipeAdvertisement pipeAdv)*
Sets the PipeAdvertisement to use for the response.

Parameter:*pipeAdv* - the PipeAdvertisement.

setPeerAdvertisement

public void **setPeerAdvertisement** *(net.jxta.protocol.PeerAdvertisement peerAdv)*
Sets the sender's PeerAdvertisement.

Parameter:*peerAdv* - the PeerAdvertisement.

setMostSignificantBits

public void **setMostSignificantBits** *(long bits)*
Sets the most significant bits of the requested CBID.

Parameter:*bits* - the most significant bits.

setLeastSignificantBits

public void **setLeastSignificantBits** *(long bits)*
Sets the least significant bits of the requested CBID.

Parameter:*bits* - the least significant bits.

setNbMostBits

public void **setNbMostBits** *(int nb)*
Sets the number of specified most significant bits of the requested CBID.

Parameter:*nb* - the number of specified most significant bits.

setNbLeastBits

public void **setNbLeastBits** *(int nb)*
Sets the number of specified least significant bits of the requested CBID.

Parameter:*nb* - the number of specified least significant bits.

E.3 jxta.cbjx.doc.UniqueNameRequest

Public abstract Class. Extends *java.lang.Object*
This class defines the XML document used to send apply request when joining a group with the *UniqueNameMembershipService*.

Constructor

public **UniqueNameRequest** *()*

getDocument

public abstract net.jxta.document.Document **getDocument** *(net.jxta.document.MimeMediaType asMimeType)*
Returns a Document containing the request's document tree

Parameter:*asMimeType* - the desired MIME type.

Returns:*Document* - the Document containing the request's document tree.

getRequestedName

public java.lang.String **getRequestedName** *()*
Returns the requested name.

Returns:*String* - the requested name.

getPeerCert

public java.security.cert.X509Certificate **getPeerCert** *()*
Returns the peer root certificate.

Returns:*X509Certificate* - the peer root certificate.

getSignature

public byte[] **getSignature** *()*
Returns the signature of this document.

Returns:*byte[]* - the signature.

setRequestedName

public void **setRequestedName** *(java.lang.String name)*
sets the requested name.

Parameter:*name* - the requested name.

setPeerCert

public void **setPeerCert** *(java.security.cert.X509Certificate cert)*
Sets the peer root certificate.

Parameter:*cert* - the peer root certificate.

setSignature

public void **setSignature** *(byte[] sign)*
sets the signature of this document.

Parameter:*sign* - the signature.

E.4 jxta.cbjx.doc.UniqueNameResponse

Public abstract Class. Extends *java.lang.Object*
This class defines the XML document used to send apply responses when joining a group with the *UniqueNameMembershipService*.

Constructor

public **UniqueNameResponse** *()*

getDocument

public abstract net.jxta.document.Document **getDocument** *(net.jxta.document.MimeMediaType asMimeType)*
Returns a Document containing the response's document tree

Parameter:*asMimeType* - the desired MIME type.

Returns:*Document* - the Document containing the responses's document tree.

getMembership

public jxta.cbjx.doc.CertListDocument **getMembership** *()*
Returns the document containing the list of certificates which proves membership within the group.

Returns:*CertListDocument* - the list of certificates.

getNameCert

public jsdsi.Certificate **getNameCert** *()*
Returns the SPKI certificate containing the authorization to use a given name within the group.

Returns:*Certificate* - the SPKI certificate.

isAlreadyUsed

public boolean **isAlreadyUsed** *()*
Returns true if the requested name is already used within the group.

Returns:*boolean*

setNameCert

public void **setNameCert** *(jsdsi.Certificate cert)*
Sets the name authorization certificate.

Parameter:*cert* - the SPKI certificate.

setMembership

public void **setMembership** *(jxta.cbjx.doc.CertListDocument certList)*
Sets the document containing the list of certificates.

Parameter:*certList* - the CertListDocument.

setUsedName

public void **setUsedName** *()*
Called if the requested name is already used within the group.

E.5 jxta.cbjx.sceau.Dictionary

Public Class. Extends *java.lang.Object*
This class contains the OTP (One Time Password) used to translate bits into words and vice versa.

Constructor

public **Dictionary** *()*

getWords

public static java.lang.String[] **getWords** *(long mostSig, long leastSig)*
Returns a sentence representing a CBID. The CBID (128 bits) is given as 2 parameters (long) of 64 bits. As each word represents a 11 bits value, only 55 bits of a long are used. Thus only 10 words (representing 110 bits) are returned.

Parameter:*mostSig* - the most significant bits of the CBID.

Parameter:*leastSig* - the least significant bits of the CBID.

Returns:*String[]* - an array containing the 10 ordered words of the sentence.

getSentence

public static java.lang.String **getSentence** *(long mostSig, long leastSig)*

Returns a sentence representing a CBID. The CBID (128 bits) is given as 2 parameters (*long*) of 64 bits. As each word represents an 11 bits value, only 55 bits of a long are used. Thus only 10 words (representing 110 bits) are returned.

Parameter:*mostSig* - the most significant bits of the CBID.

Parameter:*leastSig* - the least significant bits of the CBID.

Returns:*String* - A String containing the ten words of the sentence.

getWord

public static java.lang.String **getWord** *(int index)*

Returns the word at the given position in the dictionary.

Parameter:*index* - the position of the word (an 11 bits value).

Parameter:*String* - the corresponding word.

getBits

public static int **getBits** *(java.lang.String word)*

Returns the index (bits) corresponding to the given word.

Parameter:*word* - the word of the dictionary to translate into bits.

Parameter:*String* - the corresponding 11 bits as an *int*.

E.6 jxta.cbjx.sceau.SceauResponseEvent

Public abstract Class. Extends *java.lang.Object*

The event generated when a response is received by the Sceau service.

Constructor

public **SceauResponseEvent** *()*

getPeerAdvertisement

public net.jxta.protocol.PeerAdvertisement **getPeerAdvertisement** *()*

Returns the PeerAdvertisement contained in the response received by the Sceau service.

Returns:*PeerAdvertisement* - the PeerAdvertisement.

setPeerAdvertisement

public void **setPeerAdvertisement** *(net.jxta.protocol.PeerAdvertisement adv)*
set the PeerAdvertisement. (called by the Sceau service)

Parameter:*adv* - the PeerAdvertisement contained in the response.

getSentence

public abstract java.lang.String **getSentence** *()*
Returns the sentence of the peer which has sent the response.

Returns:*String* - the sentence.

getWords

public abstract java.lang.String[] **getWords** *()*
Returns the words of the sentence into an array.

Returns:*String[]* - the words of the sentence.

E.7 jxta.cbjx.sceau.SceauResponseListener

Public Interface.
The interface implemented by classes which need to be notified of *SceauResponseEvent*.

processSceauResponse

public void **processSceauResponse** *(jxta.cbjx.sceau.SceauResponseEvent event)*
Processes a *SceauResponseEvent*. Called by the Sceau service to notify registered classes that a new response was received.

Parameter:*event* - the SceauResponseEvent.

E.8 jxta.cbjx.sceau.SceauService

Public Interface. Extends *net.jxta.service.Service*
The definition of the Sceau service.

sendSceauRequest

public void **sendSceauRequest** *(java.lang.String[] words, jxta.cbjx.sceau.SceauResponseListener listener)*
Sends a request containg the first words of the requested sentence.

Parameter:*index* - the first words of the sentence.

Parameter:*listener* - the class that will be notified upon reception of responses.

sendSceauRequest

public void **sendSceauRequest** *(byte[] bits, jxta.cbjx.sceau.SceauResponseListener listener)*
Sends a request containg the first bits of the requested CBID.

Parameter:*index* - the first bits of the CBID.

Parameter:*listener* - the class that will be notified upon reception of responses.

removeListener

public void **removeListener** *(jxta.cbjx.sceau.SceauResponseListener listener)*
Unregisters the provided listener. This class will no longer be notified of incoming responses.

Parameter:*listener* - the class that will no longer be notified upon reception of responses.

E.9 jxta.cbjx.impl.doc.CertListDoc

Public Class. Extends *jxta.cbjx.doc.CertListDocument*
An implementation of the *CertListDocument*.

Constructor

public **CertListDoc** *()*
Creates a new certificates list object.
public **CertListDoc** *(java.io.InputStream stream)*
Creates a new certificates list object by parsing the given stream.

Parameter:*stream* - the stream containing the certificates list document.

public **CertListDoc** *(net.jxta.document.Element document)*
Creates a new certificates list object from the data of the given document.

Parameter:*document* - the document containing the certificates list.

getDocument

public net.jxta.document.Document **getDocument** *(net.jxta.document.MimeMediaType asMimeType)*
Returns a Document object containing the certificates list's document tree.

Parameter:*asMimeType* - the desired MIME type for the certificates list rendering.

Returns:*Document* - the Document containing the certificates list.

readDocument

public void **readDocument** *(net.jxta.document.Element document)*
Parses the given document tree for the certificates list.

Parameter:*document* - the object containing the certificates list data.

toString

public java.lang.String **toString** *()*
Returns an XML String reprensenting the certificates list document.

Returns:*String* - the XML representation of the document.

verify

public boolean **verify** *(net.jxta.impl.id.CBID.CBID groupCBID, net.jxta.impl.id.CBID.CBID peerCBID, boolean delegation)*
Verifies that the list of certificates proves membership within the group.

Parameter:*groupCBID* - the CBID of the group.

Parameter:*peerCBID* - the CBID of the peer.

Parameter:*delegation* - true if th peer must have the delegation enabled.

E.10 jxta.cbjx.impl.doc.SceauDoc

Public Class. Extends *jxta.cbjx.doc.SceauDocument*
An implementation of the *SceauDocument*.

Constructor

public **SceauDoc** *()*
Creates a new Sceau document object.
public **SceauDoc** *(java.io.InputStream stream)*
Creates a new Sceau document object by parsing the given stream.

Parameter:*stream* - the stream containing the Sceau document.

getDocument

public net.jxta.document.Document **getDocument** *(net.jxta.document.MimeMediaType asMimeType)*
Returns a Document object containing the Sceau document tree.

Parameter:*asMimeType* - the desired MIME type for the Sceau document rendering.

Returns:*Document* - the Document object representing the Sceau document.

readDocument

public void **readDocument** *(net.jxta.document.Element document)*
Parses the given document tree for the Sceau document.

Parameter:*document* - the object containing the Sceau document data.

toString

public java.lang.String **toString** *()*
Returns an XML String reprensenting the Sceau document.

Returns:*String* - the XML representation of the document.

E.11 jxta.cbjx.impl.doc.UniqueNameRequestDoc

Public Class. Extends *jxta.cbjx.doc.UniqueNameRequest*
An implementation of the *UniqueNameRequest*.

Constructor

public **UniqueNameRequestDoc** *()*
Creates a new request object.
public **UniqueNameRequestDoc** *(java.io.InputStream stream)*
Creates a new request object by parsing the given stream.

Parameter:*stream* - the stream containing the request document.

public **UniqueNameRequestDoc** *(net.jxta.document.Element document)*
Creates a new request object from the data of the given document.

Parameter:*document* - the document containing the request.

getDocument

public net.jxta.document.Document **getDocument** *(net.jxta.document.MimeMediaType asMimeType)*
Returns a Document object containing the request's document tree.

Parameter:*asMimeType* - the desired MIME type for the request rendering.

Returns:*Document* - the Document containing the request.

readDocument

public void **readDocument** *(net.jxta.document.Element document)*
Parses the given document tree for the request.

Parameter:*document* - the object containing the request data.

toString

public java.lang.String **toString** *()*
Returns an XML String reprensenting the request document.

Returns:*String* - the XML representation of the document.

E.12 jxta.cbjx.impl.doc.UniqueNameResponseDoc

Public Class. Extends *jxta.cbjx.doc.UniqueNameResponse*
An implementation of the *UniqueNameResponse*.

Constructor

public **UniqueNameResponseDoc** *()*
Creates a new response object.
public **UniqueNameResponseDoc** *(java.io.InputStream stream)*
Creates a new response object by parsing the given stream.

Parameter:*stream* - the stream containing the response document.

public **UniqueNameResponseDoc** *(net.jxta.document.Element document)*
Creates a new response object from the data of the given document.

Parameter:*document* - the document containing the response.

getDocument

public net.jxta.document.Document **getDocument** *(net.jxta.document.MimeMediaType asMimeType)*
Returns a Document object containing the response's document tree.

Parameter:*asMimeType* - the desired MIME type for the response rendering.

Returns:*Document* - the Document containing the response.

readDocument

public void **readDocument** *(net.jxta.document.Element document)*
Parses the given document tree for the response.

Parameter:*document* - the object containing the response data.

toString

public java.lang.String **toString** *()*
Returns an XML String reprensenting the response document.

Returns:*String* - the XML representation of the document.

E.13 jxta.cbjx.impl.sceau.SceauResponseEventImpl

Public Class. Extends *jxta.cbjx.sceau.SceauResponseEvent*.
An implementation of the *SceauResponseEvent*.

Constructor

public **SceauResponseEventImpl** *(net.jxta.protocol.PeerAdvertisement adv)*

getCBID

public net.jxta.impl.id.CBID.CBID **getCBID** *()*
Returns the CBID of the peer which sent the response.

Returns:*CBID* - the CBID of the sender of the response.

getSentence

public java.lang.String **getSentence** *()*
Returns the sentence of the peer which sent the response.

Returns:*String* - the sentence of the sender of the response.

getWords

public java.lang.String[] **getWords** *()*
Returns the words of the sentence of the peer which has sent the response.

Returns:*String[]* - An array containing the words of the sentence of the peer which has sent the response.

E.14 SceauServiceImpl

Public Class. Extends *java.lang.Object*, implements *jxta.cbjx.sceau.SceauService*, *net.jxta.pipe.PipeMsgListener*
An implementation of the *SceauService* using pipes to establish communications.

Constructor

public **SceauServiceImpl** *()*

getInterface

public net.jxta.service.Service **getInterface** *()*
Service objects are not manipulated directly to protect usage of the service. A Service interface is returned to access the service methods.

Returns:*Service* - the public interface of the service.

getImplAdvertisement

public net.jxta.document.Advertisement **getImplAdvertisement** *()*
Returns the service's advertisement.

Returns:*Advertisement* - the service's advertisement.

init

public void **init** *(net.jxta.peergroup.PeerGroup group, net.jxta.id.ID assignedID, net.jxta.document.Advertisement implAdv)*
Initializes the service.

Parameter:*group* - the group where the service is running.

Parameter:*assignedID* - the identity of the module within the group.

Parameter:*implAdv* - the module implementation advertismeent.

startApp

public int **StartApp** *(java.lang.String[] args)*
Starts the service.

Parameter:*args* - the parameters for the module.

Returns:*int* - the error code.

stopApp

public void **stopApp** *()*
Stops the service.

pipeMsgEvent

public void **pipeMsgEvent** *(net.jxta.pipe.PipeMsgEvent event)*
Called by the Pipe service when a message is received.

Parameter:*event* - the event generated upon reception of a message by the Pipe service.

sendSceauRequest

public void **sendSceauRequest** *(byte[] bits, jxta.cbjx.sceau.SceauResponseListener listener)*
Sends a Sceau request.

Parameter:*bits* - the first bits of the requested CBID.

Parameter:*listener* - the class that will be notified of responses.

sendSceauRequest

public void **sendSceauRequest** *(java.lang.String[] words, jxta.cbjx.sceau.SceauResponseListener listener)*
Sends a Sceau request.

Parameter:*words* - the first words of the requested sentence.

Parameter:*listener* - the class that will be notified of responses.

removeListener

public void **removeListener** *(jxta.cbjx.sceau.SceauResponseListener listener)*
Removes a listener.

Parameter:*listener* - the class that will no longer be notified of responses.

E.15 jxta.cbjx.impl.membership.NullCbJxMembershipService

Public Class. Extends *net.jxta.membership.MembershipService*, implements *net.jxta.resolver.QueryHandler*

The Null CbJx membership service provides a Membership Service implementation which is based on certificates to prove membership within the group. It is intended mostly as an example of a simple Membership Service rather than a practical secure Membership Service. It delivers certificates to every peer asking to join the group. If a given peer wants to join a group to which the *NullCbJxMembershipService* is associated, the peer calls the apply method with an *AuthenticationCredential*. Then if the peer does not have the group private key nor a *CertListDocument* that proves that it is part of the group, a request is sent to the group. It gets in response a list of certificates allowing him to join. When the join method is called the certificates list must be included in the *Authenticator*. If this list is valid then it is saved for later use and the peer is admitted within the group. This *MembershipService* provides delegation to members of the group. (Any members can then authorize new peers to join).

Constructor

public **NullCbJxMembershipService** *()*
Default constructor. Normally only called by the peer group.

init

public void **init** *(net.jxta.peergroup.PeerGroup group, net.jxta.id.ID assignedID, net.jxta.document.Advertisement impl)*
Initializes the service.

Parameter: *group* - the group where the service is running.

Parameter: *assignedID* - the module identity.

Parameter: *impl* - the module implementation advertisement.

getInterface

public net.jxta.service.Service **getInterface** *()*
Returns the public interface of the service.

Returns: *Service* - the public interface of the service.

getImplAdvertisement

public net.jxta.document.Advertisement **getImplAdvertisement** *()*
Returns the module implementation advertisement.

Returns: *Advertisement* - the module implementation advertisement.

startApp

public int **startApp** *(java.lang.String[] args)*
Starts the service.

Parameter: *args* - the service's parameters.

Returns: *int* - the error code.

stopApp

public void **stopApp** *()*
Stops the service.

getPeerGroup

public net.jxta.peergroup.PeerGroup **getPeerGroup** *()*
Returns the group associated with this service.

Returns: *PeerGroup* - the group where the service is running.

apply

public net.jxta.membership.Authenticator **apply** *(net.jxta.credential.AuthenticationCredential application)*
Requests the necessary credentials to join the group with which this service is associated.

Parameter: *application* - The *AuthenticationCredential* associated with this membership application.

Returns: *Authenticator* - the *Authenticator*

getDefaultCredential

public net.jxta.credential.Credential **getDefaultCredential** *()*

Returns the default credential for this peer. This is normally the "nobody" credential which is provided by default.

Returns: *Credential* - the default credential.

getCurrentCredentials

public java.util.Enumeration **getCurrentCredentials** *()*

Returns the current credentials for this peer. The elements of the enumeration are all of types derived from '*Credential*'. Before *join()* or after *resign()* are called, this enumeration will consist of a single element, a default credential which usually has the identity "nobody". This credential always exists, but likely offers few, if any, privledges within this group.

Returns: *Enumeration* - the *Credentials* currently associated with this peer for this peergroup.

getAuthCredentials

public java.util.Enumeration **getAuthCredentials** *()*

Returns the current credentials for this peer.

Returns: *Enumeration* - the *AuthenticationCredentials* which were used to establish the current identities.

join

public net.jxta.credential.Credential **join** *(net.jxta.membership.Authenticator authenticated)*

Joins the group by virtue of the completed authentication provided.

Returns: *Credential* - the credential for this completed authentication.

resign

public void **resign** *()*

Leave the group to which this policy is attached.

makeCredential

public net.jxta.credential.Credential **makeCredentials** *(net.jxta.document.Element element)*

Given a fragment of a *StructuredDocument*, it reconstructs a *Credential* object from that fragment.

Parameter: *element* - The *StructuredDocument* fragment to use for building the credential.

Returns: *Credential* - the *Credential*.

processQuery

public net.jxta.protocol.ResolverResponseMsg **processQuery** *(net.jxta.protocol.ResolverQueryMsg query)*

Processes the request from other peers to join the group. A response is returned only if the peer is a group controller or has delegation. This method is called by the resolver when a query is received.

Parameter: *query* - the query.

Returns: *ResolverResponseMsg* - the response.

processResponse

public void **processResponse** *(net.jxta.protocol.ResolverResponseMsg response)*

Processes responses from the group controller(s). Only the first valid response is considered.

Returns: *response* - the response.

E.16 jxta.cbjx.impl.membership.BootstrapMembershipService

Public Class. Extends *net.jxta.membership.MembershipService*, implements *net.jxta.resolver.QueryHandler*

The Bootstrap membership service is very similar to the *NullCbJxMembershipService*. However the control access to the group goes a little further as one needs to be known from the bootstrap process (which is performed by the Sceau service) in order to be admitted within the group. A classic scenario is the following: Peer1 knows Peer2 from the bootstrap process. Peer1 creates a new group 'Group1', it joins this group (because it is the creator of the group he can issue certificates for itself). Then Peer2 calls the apply method with an *AuthenticationCredential*. Peer2's request is sent to the group and will obviously reach Peer1. Peer1 knows Peer2 so he issues a certificates list and sends it back to Peer2. Then the remaining is the same as in the *NullCbJxMembershipService*. This MembershipService provides delegation to members of the group. (Any members of the group can then authorize new peers to join). For instance if Peer3 is known by Peer2 and not by Peer1, Peer3 can join the group because its request will reach both Peer1 and Peer2 and Peer2 can issue the certificates list for Peer3.

Constructor

public **BootstrapMembershipService** *()*
Default constructor. Normally only called by the peer group.

init

public void **init** *(net.jxta.peergroup.PeerGroup group, net.jxta.id.ID assignedID, net.jxta.document.Advertisement impl)*
Initializes the service.

Parameter: *group* - the group where the service is running.

Parameter: *assignedID* - the module identity.

Parameter: *impl* - the module implementation advertisement.

getInterface

public net.jxta.service.Service **getInterface** *()*
Returns the public interface of the service.

Returns: *Service* - the public interface of the service.

getImplAdvertisement

public net.jxta.document.Advertisement **getImplAdvertisement** *()*
Returns the module implementation advertisement.

Returns: *Advertisement* - the module implementation advertisement.

startApp

public int **startApp** *(java.lang.String[] args)*
Starts the service.

Parameter: *args* - the service's parameters.

Returns: *int* - the error code.

stopApp

public void **stopApp** *()*
Stops the service.

getPeerGroup

public net.jxta.peergroup.PeerGroup **getPeerGroup** *()*
Returns the group associated with this service.

Returns: *PeerGroup* - the group where the service is running.

apply

public net.jxta.membership.Authenticator **apply** *(net.jxta.credential.AuthenticationCredential application)*

Requests the necessary credentials to join the group with which this service is associated.

Parameter: *application* - The *AuthenticationCredential* associated with this membership application.

Returns: *Authenticator* - the *Authenticator*

getDefaultCredential

public net.jxta.credential.Credential **getDefaultCredential** *()*

Returns the default credential for this peer. This is normally the "nobody" credential which is provided by default.

Returns: *Credential* - the default credential.

getCurrentCredentials

public java.util.Enumeration **getCurrentCredentials** *()*

Returns the current credentials for this peer. The elements of the enumeration are all of types derived from '*Credential*'. Before *join()* or after *resign()* are called, this enumeration will consist of a single element, a default credential which usually has the identity "nobody". This credential always exists, but likely offers few, if any, privledges within this group.

Returns: *Enumeration* - the *Credentials* currently associated with this peer for this peergroup.

getAuthCredentials

public java.util.Enumeration **getAuthCredentials** *()*

Returns the current credentials for this peer.

Returns: *Enumeration* - the *AuthenticationCredentials* which were used to establish the current identities.

join

public net.jxta.credential.Credential **join** *(net.jxta.membership.Authenticator authenticated)*

Joins the group by virtue of the completed authentication provided.

Returns: *Credential* - the credential for this completed authentication.

resign

public void **resign** *()*
Leaves the group to which this policy is attached.

makeCredential

public net.jxta.credential.Credential **makeCredentials** *(net.jxta.document.Element element)*
Given a fragment of a *StructuredDocument*, it reconstructs a *Credential* object from that fragment.

Parameter: *element* - The *StructuredDocument* fragment to use for building the credential.

Returns: *Credential* - the *Credential*.

processQuery

public net.jxta.protocol.ResolverResponseMsg **processQuery** *(net.jxta.protocol.ResolverQueryMsg query)*
Processes the request from other peers to join the group. A response is returned only if the peer is a group controller or has delegation. This method is called by the resolver when a query is received.

Parameter: *query* - the query.

Returns: *ResolverResponseMsg* - the response.

processResponse

public void **processResponse** *(net.jxta.protocol.ResolverResponseMsg response)*
Processes responses from the group controller(s). Only the first valid response is considered.

Returns: *response* - the response.

E.17 jxta.cbjx.impl.membership.UniqueNameMembershipService

Public Class. Extends *net.jxta.membership.MembershipService*, implements *net.jxta.resolver.QueryHandler*
The unique name membership service ensures that each peer uses a unique name within the group. This membership is very similar to the *BootstrapMembershipService*. The main difference is that there is no delegation with this service. (Only the creator of the group can authorize new members to join). Thus the group controller maintains a list of every name used within the group. When a peer wants to join the group it includes the name it would like to use. If this name is not already in

the list of the group controller and if the peer is known from the bootstrap process then the group controller sends back the certificates list to the peer. The certificates list proves membership within the group as usual and the right to use a given name within the group is represented by an SPKI certificate. All these certificates are included in the *Credential*.

Constructor

public **UniqueNameMembershipService** *()*
Default constructor. Normally only called by the peer group.

init

public void **init** *(net.jxta.peergroup.PeerGroup group, net.jxta.id.ID assignedID, net.jxta.document.Advertisement impl)*
Initializes the service.

Parameter: *group* - the group where the service is running.

Parameter: *assignedID* - the module identity.

Parameter: *impl* - the module implementation advertisement.

getInterface

public net.jxta.service.Service **getInterface** *()*
Returns the public interface of the service.

Returns: *Service* - the public interface of the service.

getImplAdvertisement

public net.jxta.document.Advertisement **getImplAdvertisement** *()*
Returns the module implementation advertisement.

Returns: *Advertisement* - the module implementation advertisement.

startApp

public int **startApp** *(java.lang.String[] args)*
Starts the service.

Parameter: *args* - the service's parameters.

Returns: *int* - the error code.

stopApp

public void **stopApp** *()*
Stops the service.

getPeerGroup

public net.jxta.peergroup.PeerGroup **getPeerGroup** *()*
Returns the group associated with this service.

Returns: *PeerGroup* - the group where the service is running.

apply

public net.jxta.membership.Authenticator **apply** *(net.jxta.credential.AuthenticationCredential application)*
Requests the necessary credentials to join the group with which this service is associated.

Parameter: *application* - The *AuthenticationCredential* associated with this membership application.

Returns: *Authenticator* - the *Authenticator*

getDefaultCredential

public net.jxta.credential.Credential **getDefaultCredential** *()*
Returns the default credential for this peer. This is normally the "nobody" credential which is provided by default.

Returns: *Credential* - the default credential.

getCurrentCredentials

public java.util.Enumeration **getCurrentCredentials** *()*
Returns the current credentials for this peer. The elements of the enumeration are all of types derived from '*Credential*'. Before *join()* or after *resign()* are called, this enumeration will consist of a single element, a default credential which usually has the identity "nobody". This credential always exists, but likely offers few, if any, privledges within this group.

Returns: *Enumeration* - the *Credentials* currently associated with this peer for this peergroup.

getAuthCredentials

public java.util.Enumeration **getAuthCredentials** *()*
Returns the current credentials for this peer.

Returns: *Enumeration* - the *AuthenticationCredentials* which were used to establish the current identities.

join

public net.jxta.credential.Credential **join** *(net.jxta.membership.Authenticator authenticated)*

Joins the group by virtue of the completed authentication provided.

Returns: *Credential* - the credential for this completed authentication.

resign

public void **resign** *()*

Leaves the group to which this policy is attached.

makeCredential

public net.jxta.credential.Credential **makeCredentials** *(net.jxta.document.Element element)*

Given a fragment of a *StructuredDocument*, it reconstructs a *Credential* object from that fragment.

Parameter: *element* - The *StructuredDocument* fragment to use for building the credential.

Returns: *Credential* - the *Credential*.

makeAuthenticationCredential

public net.jxta.credential.AuthenticationCredential **makeAuthenticationCredential** *(net.jxta.document.Element element)*

Given a fragment of a *StructuredDocument*, it reconstructs an *AuthenticationCredential* object from that fragment.

Parameter: *element* - The *StructuredDocument* fragment to use for building the authentication credential.

Returns: *AuthenticationCredential* - the *AuthenticationCredential*.

processQuery

public net.jxta.protocol.ResolverResponseMsg **processQuery** *(net.jxta.protocol.ResolverQueryMsg query)*

Processes the request from other peers to join the group. A response is returned only if the peer is a group controller or has delegation. This method is called by the resolver when a query is received.

Parameter: *query* - the query.

Returns: *ResolverResponseMsg* - the response.

processResponse

public void **processResponse** *(net.jxta.protocol.ResolverResponseMsg response)*
Processes responses from the group controller(s). Only the first valid response is considered.

Returns: *response* - the response.

E.18 net.jxta.impl.endpoint.cbjx.CbJxDefs

Public Class. Extends *java.lang.Object*
Contains the constants used to handle CBIDs.

E.19 net.jxta.impl.endpoint.cbjx.CbJxUtils

Public Class. Extends *java.lang.Object*
Provides useful functionalities to handle CBIDs.

Constructor

public **CbJxUtils** *()*

getGroupDir

public static java.lang.String **getGroupDir** *()*
Returns the directory where the groups certifcates are stored.

Returns: *String* - the path to the directory.

getMembershipDir

public static java.lang.String **getMembershipDir** *()*
Returns the directory where the membership certifcates are stored.

Returns: *String* - the path to the directory.

getBootstrapDir

public static java.lang.String **getBootstrapDir** *()*
Returns the directory where the bootstrapping certifcates are stored.

Returns: *String* - the path to the directory.

getPeerFile

public static java.lang.String **getPeerFile** *()*
Returns the name of the file where the peer root certificate is stored.

Returns: *String* - the path to the file.

hash

public static byte[] **hash** *(byte[] data)*
Returns a hash (SHA-1) of the given byte array

Parameter: *data* - the data to be hashed.

Returns: *byte[]* - the hash of the data.

bytes2String

public static java.lang.String **bytes2String** *(byte[] buf)*
Converts a byte array into an hexadecimal String.

Parameter: *buf* - the bytes to translate.

Returns: *String* - the hexadecimal string.

string2Bytes

public static byte[] **string2Bytes** *(java.lang.String digits)*
Converts an hexadecimal String into a byte array.

Parameter: *digits* - the hexadecimal String.

Returns: *byte[]* - the byte array.

computeSignature

public static byte[] **computeSignature** *(java.security.interfaces.RSAPrivateKey key, java.io.InputStream stream)*
Computes the signature of the given stream.

Parameter: *key* - the RSA private key for computing the signature.

Parameter: *stream* - the stream to be signed.

Returns: *byte[]* - the signature of the stream.

verifySignature

public static boolean **verifySignature** *(java.security.interfaces.RSAPublicKey key, byte[] signature, java.io.InputStream stream)*
Verifies the signature of the given stream.

Parameter: *key* - the RSA public key for verifying the signature.

Parameter: *signature* - the signature of the stream.

Parameter: *stream* - the stream to be verified.

Returns: *boolean* - true if the signature is valid.

makeSPKICertificate

public static jsdsi.Certificate **makeSPKICertificate** *(java.io.InputStream stream)*
Makes an SPKI certificate from the data of the stream.

Parameter: *stream* - the stream containing the Base64 encoded certificate.

Returns: *Certificate* - the SPKI certificate object.

makeX509Certificate

public static java.security.cert.X509Certificate **makeX509Certificate** *(java.io.InputStream stream)*
Makes an X509 certificate from the data of the stream.

Parameter: *stream* - the stream containing the Base64 encoded certificate.

Returns: *X509Certificate* - the X509 certificate object.

encodeCertificateBase64

public static java.lang.String **encodeCertificateBase64** *(java.security.cert.Certificate cert)*
Encodes a certificate object into a Base64 String.

Parameter: *Certificate* - the certificate object to be encoded.

Returns: *String* - the encoded certificate.

E.20 net.jxta.impl.endpoint.cbjx.CbJxManager

Public Class. Extends *java.lang.Object*
Manages the peer's keys and certificates.

Constructor

public **CbJxManager** *()*

init

public static void **init** *(java.lang.String password)*

Initializes the *CbJxManager* by loading the passphrase and initializing the mapping between CBIDs and file names.

Parameter: *password* - the user's password used to encode/read the passphrase.

generateGroupCert

public static java.security.cert.Certificate **generateGroupCert** *(java.lang.String groupName)*

Generates a new root certificate for the group.

Parameter: *groupName* - the name of the group.

Returns: *Certificate* - the root certificate object.

genMembershipCert

public static jsdsi.Certificate **genMembershipCert** *(net.jxta.impl.id.CBID.CBID issuerCBID, net.jxta.impl.id.CBID.CBID subjectCBID, boolean delegation)*

Generates a new membership certificate.

Parameter: *issuerCBID* - the CBID of the issuer of the certificate.

Parameter: *subjectCBID* - the CBID of the subject of the certificate.

Parameter: *delegation* - true if the subject is authorize to admit new members in the group.

Returns: *Certificate* - the membership certificate object.

genSPKICert

public static jsdsi.Certificate **genSPKICert** *(java.security.interfaces.RSAPublicKey issuerPub, java.security.interfaces.RSAPrivateKey issuerPriv, net.jxta.impl.id.CBID.CBID subject, java.lang.String tag, boolean delegation, jsdsi.Validity validity)*

Generates a new SPKI Certificate.

Parameter: *issuerPub* - the issuer of the certificate.

Parameter: *issuerPriv* - the private key for signing the certificate.

Parameter: *subject* - the subject of the certificate.

Parameter: *tag* - the authorization name.

Parameter: *delegation* - true if the subject can propagate the authorization.

Parameter: *validity* - the validity period of the certificate.

Returns: *Certificate* - the SPKI certificate object.

genSPKICert

public static jsdsi.Certificate **genSPKICert** *(java.security.interfaces.RSAPublicKey issuerPub, java.security.interfaces.RSAPrivateKey issuerPriv, net.jxta.impl.id.CBID.CBID subject, java.lang.String tag, boolean delegation, jsdsi.Validity validity, java.lang.String display, java.lang.String comment)*

Generates a new SPKI Certificate.

Parameter: *issuerPub* - the issuer of the certificate.

Parameter: *issuerPriv* - the private key for signing the certificate.

Parameter: *subject* - the subject of the certificate.

Parameter: *tag* - the authorization name.

Parameter: *delegation* - true if the subject can propagate the authorization.

Parameter: *validity* - the validity period of the certificate.

Parameter: *display* - the display of the certificate.

Parameter: *issuerPub* - the comments on the certificate.

Returns: *Certificate* - the SPKI certificate object.

getPeerPublicKey

public static java.security.interfaces.RSAPublicKey **getPeerPublicKey** *()*

Returns the peer's public key.

Returns: *RSAPublicKey* - the peer's RSA public key.

getPeerPrivateKey

public static java.security.interfaces.RSAPrivateKey **getPeerPrivateKey** *()*

Returns the peer's private key.

Returns: *RSAPrivateKey* - the peer's RSA private key.

getPeerCert

public static java.security.cert.X509Certificate **getPeerCert** *()*

Returns the peer's root certificate.

Returns: *X509Certificate* - the peer's root certificate.

getGroupPublicKey

public static java.security.interfaces.RSAPublicKey **getGroupPublicKey** *(net.jxta.impl.id.CBID.CBID cbidGroup)*

Returns the group's public key.

Parameter: *cbidGroup* - the CBID of the group.

Returns: *RSAPublicKey* - the group's RSA public key.

getGroupPrivateKey

public static java.security.interfaces.RSAPrivateKey **getGroupPrivateKey** *(net.jxta.impl.id.CBID.CBID cbidGroup)*

Returns the group's private key.

Parameter: *cbidGroup* - the CBID of the group.

Returns: *RSSAPublicKey* - the group's RSA private key.

getGroupCert

public static java.security.cert.X509Certificate **getGroupCert** *(net.jxta.impl.id.CBID.CBID cbidGroup)*

Returns the group's root certificate.

Parameter: *cbidGroup* - the CBID of the group.

Returns: *X509Certificate* - the group's root certificate.

E.21 net.jxta.impl.endpoint.cbjx.CbJxEndpoint

Public Class. Extends *java.lang.Object*, implements *net.jxta.platform.Module*, *net.jxta.endpoint.MessageSender*, *net.jxta.endpoint.MessageReceiver*, *net.jxta.endpoint.MessageTransport*, *net.jxta.endpoint.EndpointListener*

The *CbJxEndpoint* in charge of adding and verifiying cryptographic information in messages. It uses the Endpoint Router to send message.

Constructor

public **CbJxEndpoint** *()*

init

public void **init** *(net.jxta.peergroup.PeerGroup group, net.jxta.id.ID assignedID, net.jxta.document.Advertisement impl)*

Initializes the *CbJxEndpoint*.

Parameter: *group* - the current group.

Parameter: *assignedID* - the identity of the *CbJxEndpoint*.

Parameter: *impl* - the *CbJxEndpoint* implementation advertisement.

startApp

public int **startApp** *(java.lang.String[] args)*

Makes this protocol as up and running. .

Parameter: *args* - the parameters.

Returns: *X509Certificate* - the error code.

stopApp

public void **stopApp** *()*
Closes the *CbJxEndpoint*.

getPublicAddress

public net.jxta.endpoint.EndpointAddress **getPublicAddress** *()*
Returns the *EndpointAddress* which will be used as the source address for all messages sent by this message sender.

Returns: *EndpointAddress* - the public address.

isConnectionOriented

public boolean **isconnectionOriented** *()*
Returns true if the endpoint protocol can establish connection to the remote host (like TCP).

Returns: *boolean* - true if the endpoint protocol can establish connection to the remote host.

allowRouting

public boolean **allowRouting** *()*
Returns true if the endpoint protocol can be used by the *EndpointRouter*.

Returns: *boolean* - true if the endpoint protocol can be used by the *EndpointRouter*.

getPublicAddresses

public java.util.Iterator **getPublicAddresses** *()*
Returns an *Iterator* of all of the *EndpointAddress* by which this *MessageReceiver* is reachable. This is an ordered list in order of "preference" with the most "prefered" Endpoint Addresses being at the begining of the list.

Returns: *X509Certificate* - the list of public addresses.

getProtocolName

public java.lang.String **getProtocolName** *()*
Returns the address family that this endpoint protocol handles.

Returns: *String* - The endpoint protocol name.

getMessenger

public net.jxta.endpoint.Messenger **getMessenger** *(net.jxta.endpoint.EndpointAddress dest)*

Creates an EndpointMessenger for sending messages to the specified destination address.

Parameter: *dest* - the destination address.

Returns: *Messenger* - the *Messenger*.

propagate

public void **propagate** *(net.jxta.endpoint.Message msg, java.lang.String serviceName, java.lang.String serviceParams, java.lang.String prunePeer)*

Propagates a message on this endpoint protocol.

Parameter: *msg* - the message to propagate.

Parameter: *serviceName* - the name of the destination service.

Parameter: *serviceParams* - the parameters of the destination service.

Parameter: *prunePeer* - a peer which should not receive the propagated message.

ping

public boolean **ping** *(net.jxta.endpoint.EndpointAddress addr)*

Returns true if the target address is reachable. Otherwise returns false.

Parameter: *addr* - the address to test.

Returns: *X509Certificate* - true if the address is reachable.

processIncomingMessage

public void **processIncomingMessage** *(net.jxta.endpoint.Message message, net.jxta.endpoint.EndpointAddress srcAddr, net.jxta.endpoint.EndpointAddress dstAddr)*

This method is invoked by the *EndpointService* for each incoming message that is destinated to the listener.

Parameter: *message* - the incoming message.

Parameter: *srcAddr* - the source of the message.

Parameter: *dstAddr* - the destination of the message.

addCryptoInfo

public static net.jxta.endpoint.Message **addCryptoInfo** *(net.jxta.endpoint.Message message)*

Adds cryptographic information into the message so that the destination peer can verify it.

Parameter: *message* - the message into which cryptographic information is added.

Returns: *Message* - the message containing cryptographic information.

checkCryptoInfo

public static net.jxta.endpoint.Message **checkCryptoInfo** *(net.jxta.endpoint.Message message)*

Verifies cryptographic information contained in the message. If everything is valid a special element is added in the message.

Parameter: *message* - the message to be cheked.

Returns: *Message* - the verified message.

E.22 net.jxta.impl.endpoint.cbjx.CbJxMessageInfo

Public Class. Extends *java.lang.Object*

Defines the XML document used to store message information which is useful for the *CbJxEndpoint*.

Constructor

public **CbJxMessageInfo** *()*

Creates a new message information object.

public **CbJxMessageInfo** *java.io.InputStream stream*

Creates a new message information object by parsing the given stream.

Parameter: *stream* - the stream containing message information.

getDocument

public net.jxta.document.Document **getDocument** *(net.jxta.document.MimeMediaType asMimeType)*

Returns a Document containing the information's document tree.

Parameter: *asMimeType* - the desired MIME type for the information rendering.

Returns: *Document* - the Document object containing the information's document tree.

getPeerCert

public java.security.cert.X509Certificate **getPeerCert** *()*
Returns the peer's root certificate.

Returns: *X509Certificate* - the peer's root certificate.

getDestinationAddress

public java.lang.String **getDestinationAddress** *()*
Returns the original destination address.

Returns: *String* - the original destination address.

getSourceAddress

public java.lang.String **getSourceAddress** *()*
Returns the original source address of the message.

Returns: *String* - the original source address.

getSourceID

public java.lang.String **getSourceID** *()*
Returns the peerID of the sender.

Returns: *String* - the peerID of the sender.

getList

public java.lang.String[] **getList** *()*
Returns the list of the message elements included in the signature.

Returns: *String[]* - the list of elements.

getSignature

public byte[] **getSignature** *()*
Returns the signature of the message.

Returns: *byte[]* - the signature of the message.

setPeerCert

public void **setPeerCert** *(java.security.cert.X509Certificate cert)*
Sets the peer root certificate.

Parameter: *cert* - the peer's root certificate.

setDestinationAddress

public void **setDestinationAddress** *(java.lang.String addr)*
Sets the original destination address.

Parameter: *addr* - the original destination address.

setSourceAddress

public void **setSourceAddress** *(java.lang.String addr)*
Sets the original source address of te message.

Parameter: *addr* - the original source address.

setSourceID

public void **setSourceID** *(java.lang.String src)*
Sets the peerID of the sender.

Parameter: *src* - the peerID of the sender.

setList

public void **setList** *(java.lang.String[] list)*
Sets the list of the message elements included in the signature.

Parameter: *list* - the list of elements.

setSignature

public void **getSignature** *(byte[] sign)*
Sets the signature of the message.

Parameter: *sign* - the signature of the message.

readDocument

public void **readDocument** *(net.jxta.document.TextElement document)*
Parses the given document tree for message information.

Parameter: *document* - the object containing message information data .

toString

public java.lang.String **toString** *()*
Returns an XML String representing message information.

Returns: *String* - the XML representation of the document.

E.23 net.jxta.impl.id.CBID.CBID

Public Class. Extends *java.lang.Object*, implements *java.io.Serializable*

A CBID is a 128-bit crypto-based identifier. It's the 128 bits of a secure hash (SHA-1) of the peer root certificate. It presents the same uniqueness properties as a UUID.

Constructor

public **CBID** *(byte[] buf16)*
Simple constructor.

Parameter: *buf16* - the 128 bits in a byte aray.

public **CBID** *(long mostSig, long leastSig)*
Simple constructor.

Parameter: *mostSig* - the most significant 64 bits.

Parameter: *leastSig* - the least significant 64 bits.

getMostSignificantBits

public long **getMostSignificantBits** *()*
Returns the most significant 64 bits of the CBID.

Returns: *long* - the most significant 64 bits.

getLeastSignificantBits

public long **getLeastSignificantBits** *()*
Returns the least significant 64 bits of the CBID.

Returns: *long* - the least significant 64 bits.

hashCode

public int **hashCode** *()*
Returns the hash code of the CBID.

Returns: *int* - the hash code.

equals

public boolean **equals** *(java.lang.Object target)*
Compares two CBIDs for equality.

Parameter: *target* - the CBID to be compared against.

Returns: *boolean* - true if CBIDs are equal, false otherwise.

equals

public boolean **equals** *(net.jxta.impl.id.CBID.CBID sid)*
Compares two CBIDs for equality.

Parameter: *sid* - the CBID to be compared against.

Returns: *boolean* - true if CBIDs are equal, false otherwise.

toString

public java.lang.String **toString** *()*
Returns a 36-character string of six fields separated by hyphens, with each field represented in lowercase hexadecimal with the same number of digits as in the field. The order of fields is: time_low, time_mid, version and time_hi treated as a single field, variant and clock_seq treated as a single field, and node.

Returns: *String* - the String representation of the CBID.

getBytes

public byte[] **getBytes** *()*
Returns the 128 bits of the CBID into a byte array.

Returns: *byte* - the 128 bits of the CBID.

E.24 net.jxta.impl.id.CBID.CBIDFactory

Public Class. Extends *java.lang.Object*
Generates CBIDs.

Constructor

public **CBIDFactory** *()*

E.25 newCBPeerID

public static net.jxta.impl.id.CBID.CBID **newCBPeerID** *()*
Returns a new CBID value for the peer.

Returns: *CBID* - the new CBID.

newCBGroupID

public static net.jxta.impl.id.CBID.CBID **newCBGroupID** *(java.lang.String groupName)*
Returns a new CBID value for a group.

Parameter: *groupName* - the name of the group.

Returns: *CBID* - the new CBID.

fromCertificate

public static net.jxta.impl.id.CBID.CBID **fromCertificate** *(java.security.cert.Certificate cert)*

Returns a CBID generated from the hash of the certificate.

Parameter: *cert* - the certificate.

Returns: *CBID* - the CBID derived from the certificate.

E.26 net.jxta.impl.id.CBID.IDBytes

Public Class. Extends *java.lang.Object*, implmentsjava.lang.Cloneable, *java.io.Serializable*

Represents a JXTA ID.

clone

protected java.lang.Object **clone** *()*

Clones this object. Most of the time this object is held private and immutable, so there is no need to clone it, but should the need arise (the object is used in a mutable manner and shared), then this expensive clone method does the correct thing.

Returns: *Object* - the clone.

equals

public boolean **equals** *(java.lang.Object target)*

Compares two IDs for equality.

Parameter: *target* - the ID to be compared against.

Returns: *boolean* - true if IDs are equal, false otherwise.

hashCode

public int **hashCode** *()*

Public member calculates a hash code for this ID. Used by Hashmaps.

Returns: *int* - The hashcode of this ID.

toString

public java.lang.String **toString** *()*

Public member which returns a string representation of the ID. This implementations encodes the ID into a URI.

Returns: *String* - the String representation of the ID.

E.27 net.jxta.impl.id.CBID.PeerGroupID

Public final Class. Extends *net.jxta.peergroup.PeerGroupID*

This class implements a PeerGroup ID. Each peer group is assigned a unique peer group id. CBIDs are used to implement peer group IDs.

Constructor

public **PeerGroupID** *(java.lang.String groupName)*

Constructor for creating a new PeerGroupID with a unique ID.

Parameter: *groupName* - the name of the group.

public **PeerGroupID** *(net.jxta.impl.id.CBID.CBID groupCBID)*

Creates a PeerGroupID. A PeerGroupID is provided.

Parameter: *groupCBID* - the CBID of the group.

clone

public java.lang.Object **clone** *()*

Clones this object.

Returns: *boolean* - a clone of the *PeerGroupID*.

equals

public boolean **equals** *(java.lang.Object target)*

Compares two *PeerGroupIDs* for equality.

Parameter: *target* - the *PeerGroupID* to be compared against.

Returns: *boolean* - true if *PeerGroupIDs* are equal, false otherwise.

hashCode

public int **hashCode** *()*

Returns a hash code of this object.

Returns: *int* - the hash code.

getIDFormat

public java.lang.String **getIDFormat** *()*

Returns a string identifier which indicates which ID format is used by this ID instance.

Returns: *String* - the format of the ID.

getUniqueValue

public java.lang.Object **getUniqueValue** *()*

Returns an object containing the unique value of the ID. This object must provide implementations of toString() and hashCode() that are canonical and consistent from run-to-run given the same input values. Beyond this nothing should be assumed about the nature of this object. For some implementations the object returned may be *this*.

Returns: *Object* - the unique value.

getURL

public java.net.URL **getURL** *()*

Returns a URI (URL in Java nomenclature) representation of the ID. URLs are the prefered way of externalizing and presenting JXTA IDs. The JXTA ID Factory can be used to construct ID Objects from URLs containing JXTA IDs.

Returns: *URL* - the URL representation of the ID.

getPeerGroupID

public net.jxta.id.ID **getPeerGroupID** *()*

Returns the ID.

Returns: *ID* - the ID.

getPeerGroupCBID

public net.jxta.impl.id.CBID.CBID **getPeerGroupCBID** *()*

Returns the CBID of the group. *PeerGroupID* contains a CBID.

Returns: *CBID* - the group CBID.

E.28 net.jxta.impl.id.CBID.PeerID

Public final Class. Extends *net.jxta.peer.PeerID*

This class implements a Peer ID. Each peer is assigned a unique peer id. CBIDs are used to implement peer IDs.

Constructor

public **PeerID** *(net.jxta.impl.id.CBID.PeerGroupID groupID)*

Constructor for creating a new PeerID with a unique ID.

Parameter: *groupID* - the group to which the peer belongs.

clone

public java.lang.Object **clone** *()*
Clones this object.

Returns: *boolean* - a clone of the *PeerID*.

equals

public boolean **equals** *(java.lang.Object target)*
Compares two *PeerIDs* for equality.

Parameter: *target* - the *PeerID* to be compared against.

Returns: *boolean* - true if *PeerIDs* are equal, false otherwise.

hashCode

public int **hashCode** *()*
Returns a hash code of this object.

Returns: *int* - the hash code.

getIDFormat

public java.lang.String **getIDFormat** *()*
Returns a string identifier which indicates which ID format is used by this ID instance.

Returns: *String* - the format of the ID.

getUniqueValue

public java.lang.Object **getUniqueValue** *()*
Returns an object containing the unique value of the ID. This object must provide implementations of toString() and hashCode() that are canonical and consistent from run-to-run given the same input values. Beyond this nothing should be assumed about the nature of this object. For some implementations the object returned may be *this*.

Returns: *Object* - the unique value.

getURL

public java.net.URL **getURL** *()*
Returns a URI (URL in Java nomenclature) representation of the ID. URLs are the prefered way of externalizing and presenting JXTA IDs. The JXTA ID Factory can be used to construct ID Objects from URLs containing JXTA IDs.

Returns: *URL* - the URL representation of the ID.

getPeerGroupID

public net.jxta.id.ID **getPeerGroupID** *()*
Returns the *PeerGroupID* of the peer group asociated with this peer.

Returns: *ID* - the peer group ID.

getPeerCBID

public net.jxta.impl.id.CBID.CBID **getPeerCBID** *()*
Returns the CBID of the peer. *PeerID* contains a CBID.

Returns: *CBID* - the peer CBID.

Bibliography

[1] Claude Castelluccia, Gabriel Montenegro.
Opportunistic Encryption for IPv6.
INRIA Technical Report, October 2002.
http://www.inria.fr/rrrt/rr-4568.html.

[2] Gabriel Montenegro, Julien Laganier, Claude Castelluccia.
Securing IPv6 Neighbor Discovery.
internet draft, June 2002.
http://search.ietf.org/internet-drafts/draft-montenegro-send-cga-rr-00.txt.

[3] Claude Castelluccia, Gabriel Montenegro.
Securing Group Management in IPv6 with Cryptographically Generated Addresses.
INRIA Technical Report, July 2002.
http://www.inria.fr/rrrt/rr-4523.html.
Presentation available at: http://www.securemulticast.org/GSEC/gsec3_ietf53_SGM6.pdf.

[4] Gabriel Montenegro, Claude Castelluccia.
Statistically Unique and Cryptographically Verifiable Identifiers and Addresses.
February 2002.
http://www.isoc.org/isoc/conferences/ndss/02/proceedings/papers/monten.pdf.

[5] Gabriel Montenegro, Claude Castelluccia.
SUCV Identifiers and Addresses. June 2002.
http://search.ietf.org/internet-drafts/draft-montenegro-sucv-03.txt.

[6] *Working Session on Security in Ad Hoc Networks.*
Lausanne, June 12, 2002.
http://lcawww.epfl.ch/buttyan/June12/presentations.html.

[7] Claude Castelluccia, Gabriel Montenegro.
Crypto-based ID's in MANET's: Some Preliminary Thoughts.
Presentation available at: http://lcawww.epfl.ch/buttyan/June12/presentations/Castelluccia-Montenegro.pdf.

[8] *Project JXTA: Protocols specification v1.0.*
2002.
http://spec.jxta.org/v1.0/docbook/JXTAProtocols.htm

[9] Sun Mincrosystems, Inc.
Project JXTA: Java^{TM}Programer's guide.
2001.
http://www.jxta.org/docs/jxtaprogguide_final.pdf

[10] Brendon J. Wilson.
JXTA
New Riders Publishing.
June 15, 2002.

[11] *RFC3281 - X.509 Certificate* (attribute certificate profile)
http://www.ietf.org/rfc/rfc3281.txt.

[12] *RFC2289 - One Time Password*
http://www.ietf.org/rfc/rfc2289.txt.