­­­­SCAPI Communication Layer

R&D Group

User Manual

Written by Yael Ejgenberg

Created 29 May 2013

Modified 29 May 2013

PUBLIC

Change Control Information

|  |  |  |  |
| --- | --- | --- | --- |
| Version No. | Date | Changed by | Scope of Change |
| 1.00 | 08-Jan-2006 |  | Initial version |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[1. Scope 4](#_Toc357950348)

[1.1 Introduction 4](#_Toc357950349)

[*1.1* Purpose 4](#_Toc357950350)

[2. Definitions Acronyms and Abbreviations 4](#_Toc357950351)

[3. References 4](#_Toc357950352)

[4. General description 5](#_Toc357950353)

[5. Functionalities 6](#_Toc357950354)

[5.1 Setup the communication services 6](#_Toc357950355)

[5.2 Use the communication services 6](#_Toc357950356)

[6. Static View 7](#_Toc357950357)

[6.1 CommunicationSetup 8](#_Toc357950358)

[6.2 Channel 8](#_Toc357950359)

[6.2.1 PlainTcpChannel 8](#_Toc357950360)

[6.2.2 EncryptedChannel 9](#_Toc357950361)

[6.2.3 AuthenticatedChannel 9](#_Toc357950362)

[6.3 ConnectivitySuccessVerifier 10](#_Toc357950363)

[7. What type of data can be sent over a channel? 12](#_Toc357950364)

[8. Example 13](#_Toc357950365)

[8.1 Plain Security Level 13](#_Toc357950366)

[8.2 Authenticated Security Level 14](#_Toc357950367)

[8.3 Encrypted Security Level 14](#_Toc357950368)

# Scope

## Introduction

The communication layer provides communication services to any interactive cryptographic protocol. It uses cryptographic tools such as digital signatures and encryption in order to provide secure and private channels. This layer is heavily used by the interactive mid-crypto protocols and MPC protocols. It can also be used by any other cryptographic protocol that requires communication.

* 1. Purpose

The purpose of this layer is to provide communication services to any cryptographic protocol. It aims to hide all the details of implementation, thus allowing the user to concentrate on the implementation of the CP without caring about how the communication is done.

Different levels of security are provided by this layer:

* Plain- It only ensures that the message will get from one party to another,
* Encrypted- It ensures that the message will be encrypted by the sending party and decrypted by the receiving party. All the work is done under the hood. It uses symmetric encryption.
* Authenticated- It ensures that the message will be authenticated by the sending party and verified by the receiving party. All the work is done under the hood.

# Definitions Acronyms and Abbreviations

|  |  |
| --- | --- |
| MPC | Multi Party Computation |
| CP | Cryptographic Protocol |

# References

[1] Doc1

[2] Doc2

# General description

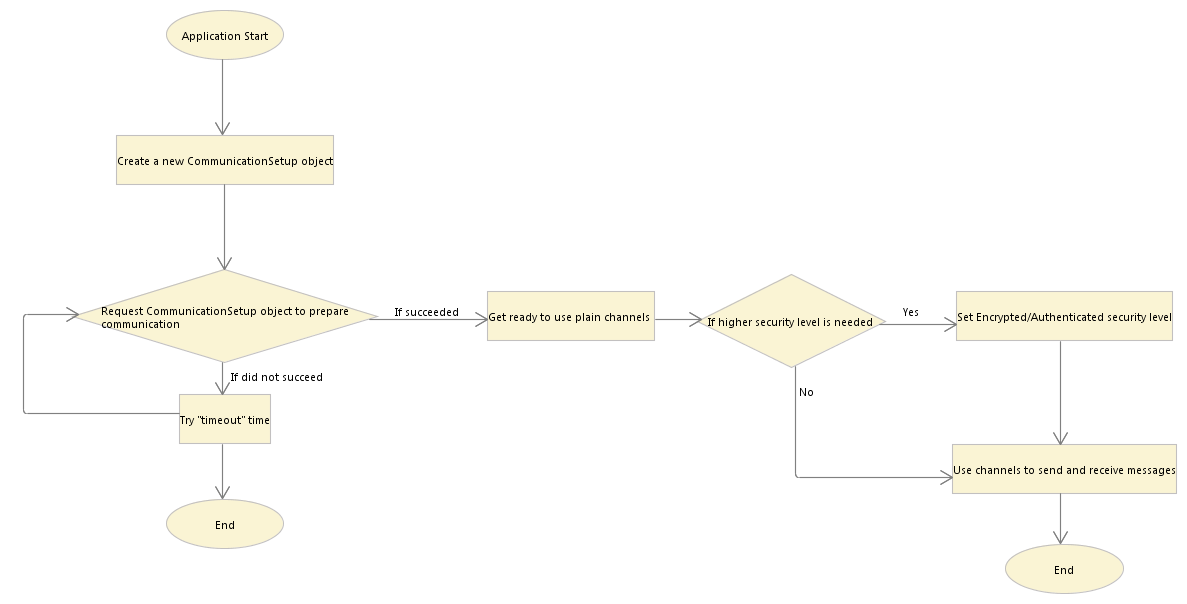
The Communications Layer is a tool used by a party that is interested in setting up connections between itself and other parties. Given a list of parties, the layer attempts to set connections to them according to parameters given by the calling application.

If succeeds, it returns these connections so that the calling party can send and receive data over them.

An example of usage of the layer by an application running some MPC follows:

* Instantiate an object of type CommunicationSetup.
* Call the prepareForCommunication method of that object with a list of parties to connect to and other setup parameters. (prepareForCommunication functions are the only public method of this class).
* Get from prepareForCommunication a container holding ready-to-use connections that have Plain security level (explained below).
* If necessary, set a higher security level such as Encrypted or Authenticated.
* Call the send and receive methods of the ready connections as needed by the MPC.

The application may be interested in putting each connection in a different thread but it is up to the application to do so and not the responsibility of the Communications Layer. This provides more flexibility of use.

In the following figure we show the flowchart of the example presented above:

# Functionalities

## Setup the communication services

The first thing that needs to be done to obtain communication services is to setup the connections between the different parties. Each party needs to run the setup process at the end of which the established connections are obtained. The established connections are called channels.

This functionality is performed by the CommunicationSetup::prepareForCommunication(…) functions:

* **prepareForCommunication**(List<Party> listOfParties,ConnectivitySuccessVerifier successLevel, long timeOut, boolean enableNagle) : Map<InetSocketAddress, Channel>
* **prepareForCommunication**(List<Party> listOfParties,ConnectivitySuccessVerifier successLevel, long timeOut) : Map<InetSocketAddress, Channel>

## Use the communication services

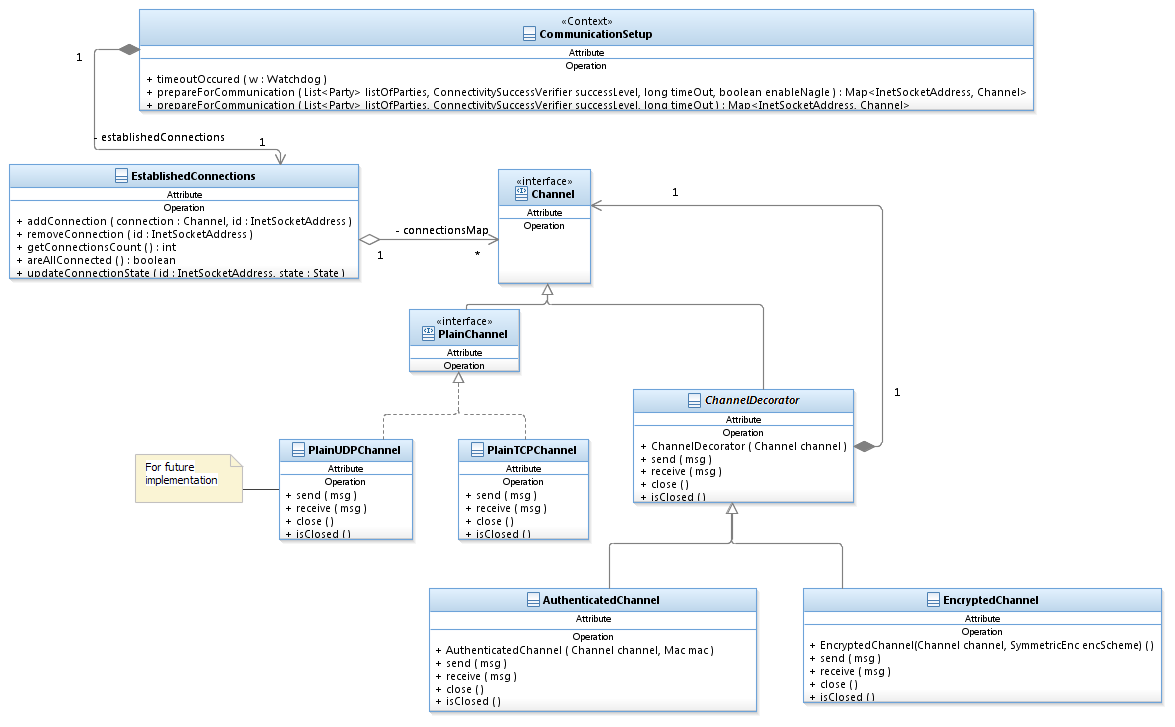
A channel represents an established connection between two parties: me and another one in the list of parties. As mentioned above, a channel can have Plain, Encrypted or Authenticated Security level, depending on the requirements of the application.

In all those cases the channel has two main functions:

* **send**(Serializable msg)
* **receive**() : Serializable

This means that from the applications point of view, once it obtains the channels and sets their Security Level it can completely forget about it and just send and receive messages knowing that all the encryption or authentication work is done under the hood.

# Static View

In this diagram we show the relationships between the different public classes (and their public functions) of the Communication Layer.

## CommunicationSetup

An application requesting from CommunicationSetup to prepare for communication needs to provide the following information as input:

* The list of parties to connect to. As a convention, we will set the first party in the list to be the requesting party, that is, the party represented by the application.
* The type of connecting success required. (See explanation [below](#_ConnectivitySuccessVerifier)).
* A time-out (in milliseconds) specifying how long to wait for connections to be established and secured.

CommunicationSetup implements the org.apache.commons.exec.TimeoutObserver interface. This interface supplies a mechanism for notifying classes that a timeout has arrived.

## Channel

A connection is represented by a Channel class. We use the Decorator Design Pattern to implement different types of channels. The basic components are: a concrete channel represented by the class PlainTCPChannel and decorator classes represented by AuthenticatedChannel and EncryptedChannel.

In order to enforce the right usage of the Channel family we restricted the ability to instantiate an undecorated channel only to classes within the CommunicationLayer’s package by making the relevant undecorated channels’ constructors’ package private. Once the connections are established and returned by the CommunicationSetup the calling application can choose to decorate each channel with a suitable decoration such as encryption or authentication. This allows for maximum flexibility for each separate connection. One thing to take into account is that to be able to decorate a channel the basic channel must be in READY state and this can only be achieved by calling the CommunicationSetup. Therefore, the enforcement we required is achieved. This means that the constructor of the undecorated channel is unreachable from another package. However, the send, receive and close functions are declared public, therefore allowing anyone holding a channel to be able to use them.

### PlainTcpChannel

PlainTcpChannel has no public constructor as explained [above](#_Channel).

Function: send(Serializable msg)

* Write the message to the output stream of this channel’s socket.

Function: receive(): Serializable do:

* Read data in the input stream of this channel’s socket.

### EncryptedChannel

This channel ensures CPA security level. The owner of the channel is responsible for setting the encryption scheme to use and make sure the encryption scheme is initialized with a suitable key. Then, every message sent via this channel is encrypted and decrypted using the underlying encryption scheme.

The user needs not to worry about any of the encryption or decryption tasks. The owner of this channel can rest assure that when an object gets sent over this channel it gets encrypted with the defined encryption scheme. In the same way, when receiving a message sent over this channel (which was encrypted by the other party) the owner of the channel receives an already decrypted object.

Constructor: public EncryptedChannel(Channel ch, SymmetricEnc enc) : SecurityLevelException do:

There is only one public constructor for EncryptedChannel. It checks that the Security Level of the encryption scheme is at least CPA. If so, goes on constructing the channel; else, it throws SecurityLevelException. The channel has to be in READY state, otherwise an InvalidChannelException is thrown. The channel can be in READY state only if it was created by the CommunicationSetup::prepareForCommunication function.

Function: send(Serializable msg) do:

* Encrypt the message using the algorithm and keys specified in the channel.
* Send the encrypted message by calling the send method of the underlying channel.

Function: receive(): Serializable do:

* Call the receive method of the underlying channel and get the encrypted message.
* Decrypt the message using the algorithm and keys specified by the channel.
* Return the decrypted message.

Function: setKey(SecretKey key) do:

* Sets the key of the underlying encryption scheme. This function must be called before sending or receiving messages, if the encryption scheme passed to this channel had not been set with a key yet. The key can be set indefinite number of times depending on the needs of the application.

### AuthenticatedChannel

This channel ensures UnlimitedTimes security level. The owner of the channel is responsible for setting the MAC algorithm to use and make sure the MAC is initialized with a suitable key. Then, every message sent via this channel is authenticated using the underlying MAC algorithm and every message received is verified by it.

The user needs not to worry about any of the authentication and verification tasks. The owner of this channel can rest assure that when an object gets sent over this channel it gets authenticated with the defined MAC algorithm. In the same way, when receiving a message sent over this channel (which was authenticated by the other party) the owner of the channel receives an already verified and plain object.

Constructor: public AuthenticatedChannel(Channel ch, Mac mac): SecurityLevelException do:

There is only one public constructor for AuthenticatedChannel. It checks that the Security Level of the authentication is at least UnlimitedTimes. If so, goes on constructing the channel; else, it throws SecurityLevelException. The channel has to be in READY state, otherwise an InvalidChannelException is thrown. The channel can be in READY state only if it was created by the CommunicationSetup::preapreForCommunication function.

Function: send(Serializable msg) do:

* Sign the message first using algorithm and keys specified in the channel.
* Send the signed message by calling the send method of the underlying channel.

Function: receive(): Serializable do:

* Call the receive method of the underlying channel and get the authenticated message.
* Verify the message using the Mac. If the message verifies, then return the message; else, return false.

Function: setKey(SecretKey key) do:

* Sets the key of the underlying MAC algorithm. This function must be called before sending or receiving messages if the MAC algorithm passed to this channel had not been set with a key yet. The key can be set indefinite number of times depending on the needs of the application.

## ConnectivitySuccessVerifier

Different MPCs may require different levels of success when checking the connections between all the parties that were supposed to participate. For example, for some protocols it may be enough that some percentage of the parties succeeded in connecting and for other protocols it may be absolutely necessary to ensure that all the parties are connected to all the parties. The verification of each level of success is implemented by a different class in the ConnectivitySuccessVerifier family.

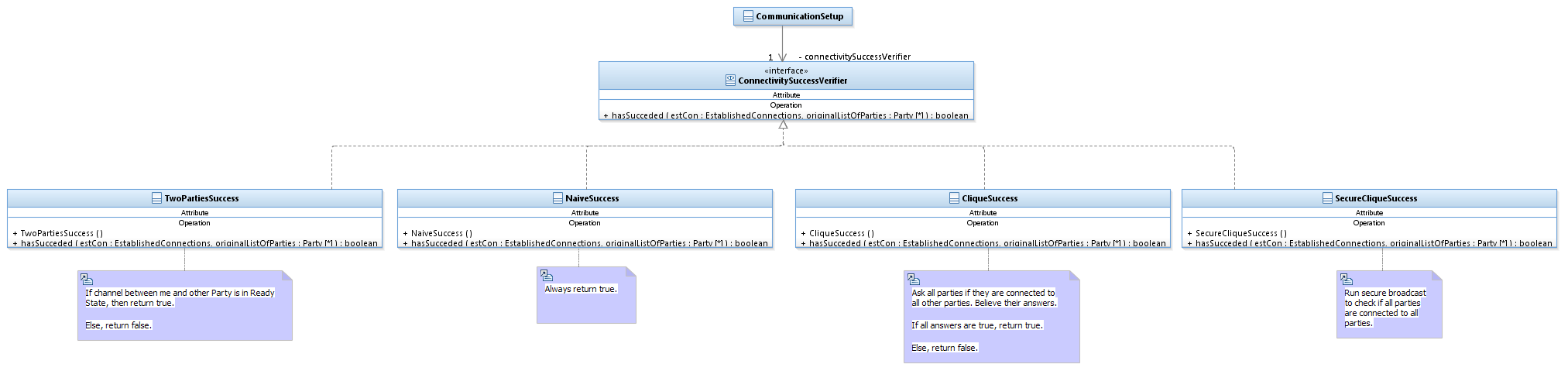
The application running the protocol will ask the protocol which level it needs, and will pass the CommunicationSetup an instance of ConnectivitySuccessVerifier that corresponds to the required type. We use here the Strategy Pattern to allow us to change the algorithm accordingly.

In all cases

* the verifier will get the information about the established connections as input and the original list of parties,
* it will run a certain algorithm,
* it will return true or false.

Note: So far, only the NaiveSuccess level has been implemented, which always returns true.

In the diagram below we show the Strategy Pattern we use and the algorithms we intend to implement in the future:



# What type of data can be sent over a channel?

As mentioned above a Channel has two main functions send and receive. But what can be sent and received with a Channel? We need to have a general interface that should be implemented by any object in SCAPI that needs to be sent or received by a Secure Computation protocol. Fortunately, Java already comes with a built-in mechanism for sending objects that were generated by one VM to another VM. This mechanism is called Serialization. Serialization is used to persist a JAVA object over time and physical boundaries. In particular, a JAVA object can be “saved” for later use in a file, or it can be sent via a socket to another entity. The Serialization mechanism streams the object to a stream of bytes containing all the information necessary to recreate the object afterwards and/or in another machine.(For more detailed information about the Serialization mechanism see: <http://docs.oracle.com/javase/6/docs/api/java/io/Serializable.html>) Since our basic concrete Channel, the PlainTCPChannel is based on a TCP socket, in order to use this mechanism all we need to do is to make the classes in SCAPI that need to be sent over channels Serializable. In general, we will say that any class that contains data should be Serializable. For secure coding reasons (see: Effective Java – Second Edition by Joshua Bloch, page 289) we will not make an interface representing something that might be a data object Serializable, but rather we will mark each concrete implementing class as such whenever necessary. For example, if we wish to send a Plaintext (Plaintext is an element of the Mid-layer of SCAPI) we will not mark the interface Plaintext as Serializable because this will cause all the implementing classes of Plaintext to be Serializable and this may not be necessarily a good thing. Instead, for example, we will mark the implementing class ByteArrayPlaintext as Serializable.

Note that the Serialization mechanism provides default implementation for serializing primitive types like int and byte and also basic classes such as arrays, String and BigInteger have a default implementation.

We said that in general any SCAPI class that contains data should be Serializable. Let’s look at the list of elements that we think a protocol may need to send over a channel:

1. Keys (already marked as Serializable by JAVA)
2. Plaintexts
3. Ciphertexts
4. Signatures
5. TaggedObjects (an object with its respective MAC)
6. Group Params (the parameters of a Dlog group)
7. TPElements
8. GroupElements

However, TPElements and GroupElements , although we would like to send them over a channel they present a problem. It is not possible to construct an object of type RabinElement or RSAElement without the actual corresponding permutation, as well as it is not possible to construct objects of type ZpElement, ECF2mElement or ECFpElement without the actual corresponding Dlog group. This is a feature by design. For example a RabinElement (which is an integer value between 1 and modulus -1 such that it has a quadratic residue) cannot exist outside the environment of the RabinPermutation that has the modulus. In the same manner a point in an Elliptic Curve cannot exist without the curve. This would imply that all the relevant information be passed in the constructor of each element. This is obviously a huge burden on the system, especially when lots of data need to be sent. Imagine a protocol that assumes a certain Elliptic Curve was agreed upon by both parties and now as part of the protocol thousands of points on the curve need to be sent by one party to the other. This would mean that in all those thousands of times in order to reconstruct the element on the other side the information about the group has to be sent as well. This has two main drawbacks: the obvious one, why send thousands of time back and forth data that both parties already have; a less obvious one is that a malicious party may send a non-valid group element for the Dlog group that was agreed upon, but if all the data about the Dlog group is sent together with the group element this data will not be the right Dlog group but will create a “valid” group element in the receiving side.

So, how are these elements created if their constructors cannot be called? Well, their constructors can be called by classes within the package. Specifically, the way to create a TPElement is by calling the generateElement of the relevant TPPermutation, and the way to create a GroupElement is by calling the generateElement of the relevant Dlog group. This way, all that needs to be sent over the channel is the data needed for the TPPermutation or the Dlog Group to generate the requested element. This design has another advantage. It allows sending a Group Element or a TPElement to a party that might use another library, for example if a computation is run between a PC and a mobile. This greatly increases SCAPI’s portability between different platforms. For this to work we will add auxiliary classes (that will be Serializable) for TPElements and GroupElements that will contain only the internal data we wish to send over a channel. Also, we will add a new function to all these elements: generateSendableData(): Serializable that will return the relevant auxiliary class. For example, a party that wishes to send a ECF2mPoint to the other send will call:

channel.send(myEcF2mPoint.generateSendableData())

a party that receives the sent point will call:

ECF2mPoint myPoint = myDlogGroup.generateElement(channel.receive())

# Examples

Very simple examples of how to create Encrypted, Authenticated and Secure Channels provided a PlainTcpChannel has been obtained via the CommunicationSetup can be found in:

<http://crypto.biu.ac.il/scapi/s_examples.html>.

More complex examples that show the whole process are detailed below.

## Plain Security Level

We present here a simple example of how to setup the communication between two parties that only need Plain Security Level; and how to send and receive some GroupElements of a DlogGroup.

Each party is represented by another application App1 and App2; in a real setup each application would run in a different machine.

Double-click the elements below to see the actual implementing example:



## Authenticated Security Level

This example is based on the Plain Security Level one. It runs by using the same runner classes and properties files, all you need to change is the applications:



## Encrypted Security Level

The decoration of the channel is done in a similar manner to the Authenticated Security Level.