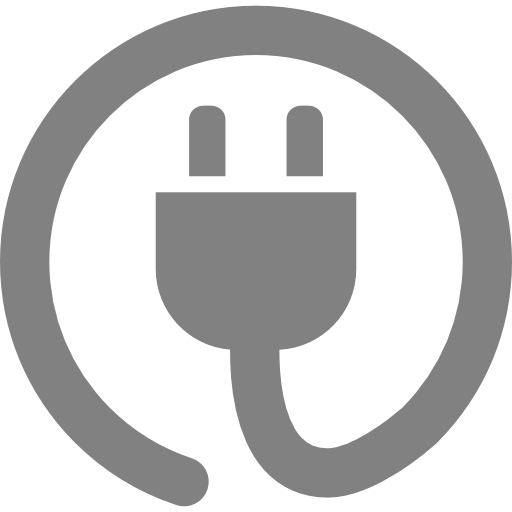
**Implementation of SVC on IP network  
with HTP support**

Internship report

(Draft v0.1)



**Supervisor**

Arnaud FEVRIER

**Student**

Xuan Thong DANG

eRISCS – Polytech Luminy, 07/2016

Contents

[I. Objectives 3](#_Toc456099541)

[II. Context 4](#_Toc456099542)

[1. The need of a new protocol 4](#_Toc456099543)

[2. Secure Virtual Connector 4](#_Toc456099544)

[a. Original Secure Virtual Connector 4](#_Toc456099545)

[b. Modified SVC 5](#_Toc456099546)

[3. Hybrid Transmission Protocol 6](#_Toc456099547)

[c. Structure of a HTP packet 6](#_Toc456099548)

[d. Priority scheduling algorithm 7](#_Toc456099549)

[III. Implementation of SVC on IP network 8](#_Toc456099550)

[1. Overall architecture 8](#_Toc456099551)

[2. SVC interface and implementation of daemon 9](#_Toc456099552)

[a. SVC interface 9](#_Toc456099553)

[b. The SVC daemon 10](#_Toc456099554)

# Objectives

Continuing our work on the support network for a metropolitan bus transport system, this document describes the need of a fast and secure data exchanging protocol on such network and its implementation, with the support of a new transport-layer protocol that has both advantages from UDP and TCP.

# Context

## The need of a new protocol

To protect the passengers’ privacy, the data of a video surveillance system must be encrypted before sending back to the control station. For that, not only a strong cryptographic encryption is required, but also a reliable and lightweight connection is needed to increase the quality of service, by reducing network delay and latency.

Several existing protocols have been considered, but they are either not supporting client’s identity protection or performing overhead message exchanging. Moreover, these protocols don’t offer a flexible method of authentication and often rely on a fixed transmission protocol.

Hence, the new protocol must satisfy those basic requirements and overcome the limits mentioned above.

## Secure Virtual Connector

### Original Secure Virtual Connector

Secure Virtual Connector (SVC) is a protocol of negotiating and establishing secure network data channel. SVC was first introduced by G. RISTERUCCI, T. MUNTEAN and L. MUGWANEZA in their research at ERISCS Research Group.

SVC focuses on real-time and embedded system by reducing the overhead of exchanged messages during channel establishing phase thus reduces network delay and latency. Using most up-to-date encryption algorithms (AES-128/GCM, GHASH, SHA2-256, ECHD), SVC can satisfy most off applications that need a secure data transmission. SVC also offers protection of client’s identity and reduction of attack vector. More about SVC can be found in “*A new Secure Virtual Connector approach for communication within large distributed systems*”.

For the protection of client’s identity makes sense, SVC operates within client-server model, in which the server is the one who initiates the negotiation. A complete schema of message exchanging is shown below:

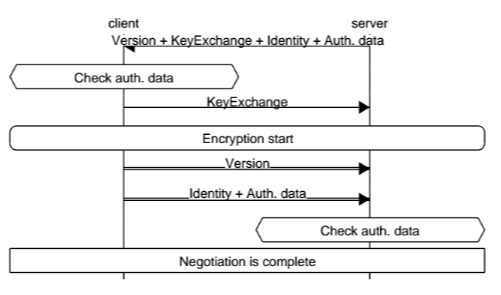


Figure 2.1 – Message exchanging in SVC

### Modified SVC

Despite the advantages of SVC compared to other authentication protocols, there are some points that we need to improve.

With the aim of protecting the client’s identity and configuration, the client-side version of SVC and the identity are not sent until encryption process has been established. Unfortunately, this can be simply compromised if an attacker intercepts the connection request from the server and reuse this information later (Man-in-the-middle, replay attack). The client innocently verifies the server’s identity without doubt, and then starts a secure connection to the faker. Client’s identity and its version of SVC can be disclosed from then on.

To overcome this problem, in the modified version of SVC, we introduce an extra step at the beginning of the protocol. It’s simply a connection request from client including a challenge of authentication and the version of client-side SVC. We don’t intent to hide the version of client-side SVC because we find this unnecessary. This info can be easily obtained by analyzing the traffic sent from the server and the status of the connection.

The presence of a challenge from the client makes it theoretically impossible to forge a fake response from the attacker. The version information tells the server to either accept or refuse the connection request, thus helps economizing memory and processing power, avoiding DoS attack.

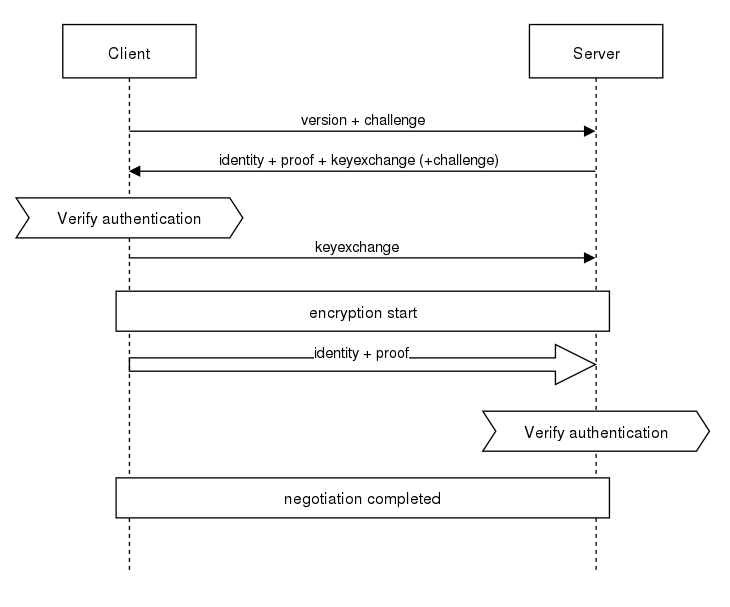


Figure 2.2 – Modified versions of SVC

Compared to the original SVC, the modified version just added one more message to the negotiation process. This doesn’t affect much the principle of SVC, but even makes it more secure.

## Hybrid Transmission Protocol

When it’s coming to choose a transmission protocol for a network application, we often stand between the choices of TCP and UDP. TCP is optimized for accurate delivery rather than timely delivery. Therefore, it sometimes incurs relatively long delays while waiting for out-of-order messages or retransmissions of lost messages. TCP is also sensible with network change, as it maintains a connection between the client and the server. In our bus transport system, network switch is intentionally performed when the station moves between network access points. Because of that, TCP is obviously not the choice.

Nevertheless, UDP cannot be used “as it is”. In a multifunctional system, applications are often classed by priority. However, UDP has no method of distinguishing those priorities. Either, it cannot provide a delivery guarantee, in case of need.

To overcome these limits, a Hybrid Transmission Protocol (HTP) is introduced. Like UDP, HTP is connection-less, and tents to reduce TCP’s sophisticated control mechanisms. Sharing the same header with UDP makes HTP able to be implemented in both transport and application layers.

### Structure of a HTP packet

**Data**

Source port

Destination port

Data length

Checksum

0

16

32

Resent counter (optional)

**Ordinary UDP header**

**HTP header**

P

Flags

 Packet ID

Options

Window Size (optional)

Figure 2.2 – Structure of a HTP packet

Explanation of attributes:

* Source port: the port of sender application
* Destination port: the port of receiver application
* Data length: the length of the data packet, including header
* Checksum: header checksum
* Packet ID: the unique ID of a packet during a data communication. 232 ID will be available, start by 0, increase by 1 for each packet and reset to 0 when bypassing 232 – 1.
* P[0-1]: packet priority. More about priority scheduling algorithm will be described on section b.
  + 00: low priority
  + 01: normal priority
  + 10: high priority
  + 11: urgent
* Flags[2-7]: control flags
  + Flags[2] – TCP: enable control protocol on current packet. Window Size field is added. Default to 0.
  + Flags[3] – ACK: indicate the ACK response for packet that has ID of Packet ID. Data in ACK response packet will be omitted. Default to 0.
  + Flags[4] – RSN: indicate that the current packet is being resent. Resent counter will be decreased by one. Default to 0.
  + Flags[5-7] – not used (yet)
* Options[8-15]
  + Options[8] – DLA: discard late-arriving packet. Default to 0.
  + Options[9] – RSP: resent policy, 0 for “stop and wait” and 1 for “selective repeat”
  + Options[10-16]: not used (yet).
* Resent counter: used with TCP control to resend missing packets.
* Window Size: used with TCP control to give information about receiving/sending capacity.
* Data: data payload.

### Priority scheduling algorithm

To keep the logic of priority packet scheduling, HTP maintains a global set of message queues corresponding to the packet priorities: urgent, high, normal, low.

Packets from application layer arrived at HTP will be sorted into these four queues before being encapsulated with IP. HTP does an infinite loop to pick packet from these queues which follows this rules:

* + Pick one packet from **urgent** queue
  + Pick one packet from **high** queue
  + Repeat (i) and (ii) 4 times
  + Pick one packet from **urgent** queue
  + Pick one packet from **normal** queue
  + Repeat (iv) and (v) 2 times
  + Pick one packet from **urgent** queue
  + Pick one packet from **low** queue
  + Return to (i)

By doing so, urgent messages will be sent alternately among other messages, consuming 50% of sending process. High, normal and low messages will be sent with 4:2:1 ratio, respectively.

# Implementation of SVC on IP network

## Overall architecture

We hereby introduce the SVC interfaces, along with the SVC daemon. SVC interfaces include the necessary functions which will be used to implement the authentication procedure. The business of cryptographic encryption will be left for the daemon. This allows applications using SVC to be as up-to-date as the daemon.

User space

Kernel space

SVC interfaces

SVC daemon

UNIX domain socket

IP

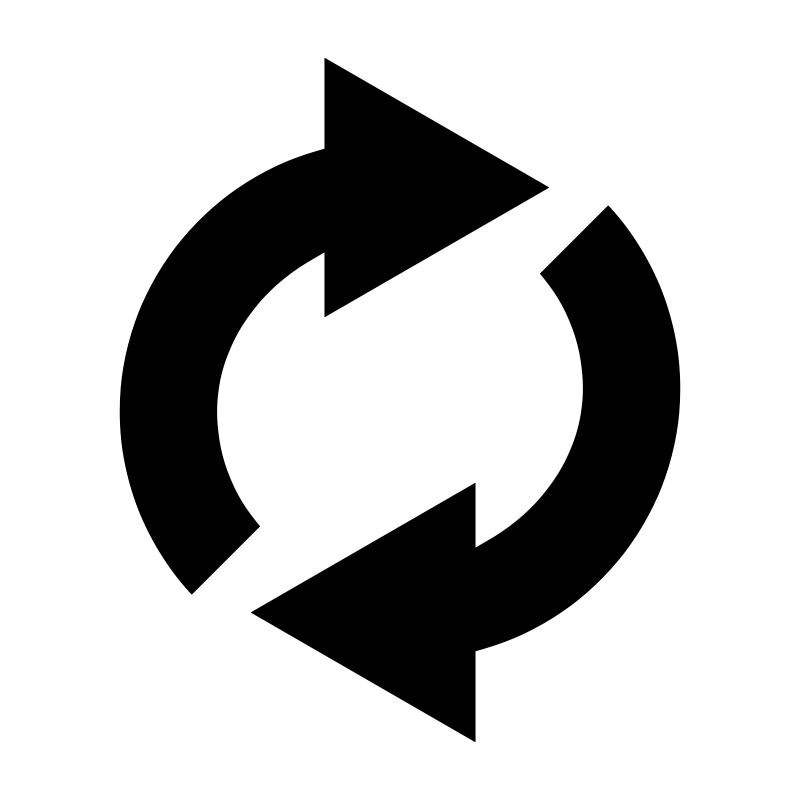
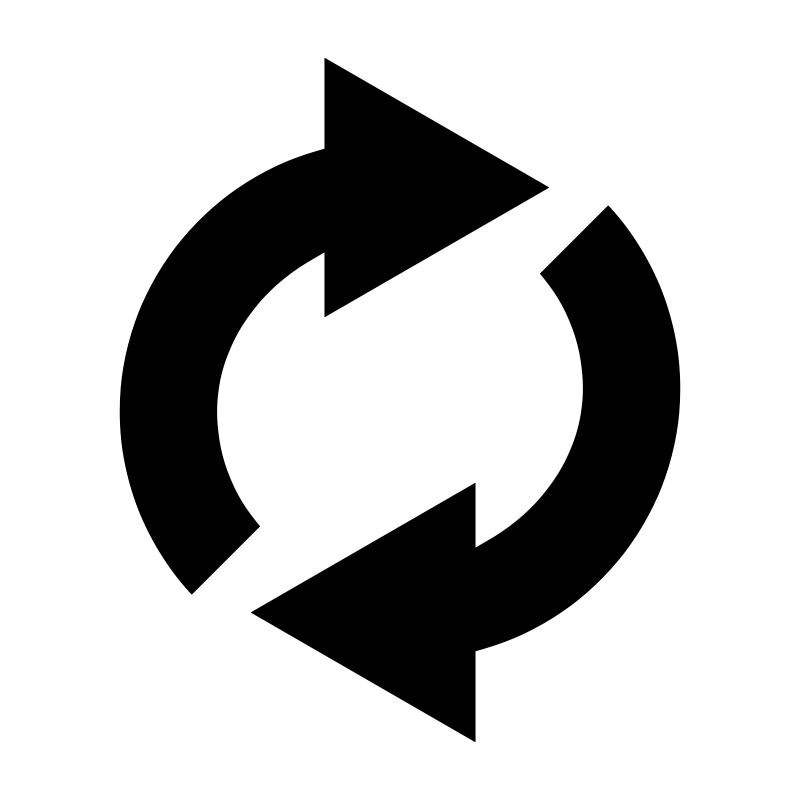
UDP

**HTP**

Kernel socket

Sockets from other applications

Flow control, Congestion control



Application

Message  
queues

Network interface

Figure 3.1 – Overall architecture of a SCV implementation

Note that there will be only one instance of SVC daemon to be running. Thus, there will be many connections to the daemon. These connections will be distinguished by the IDs of applications, which help the daemon to forward the arriving packets correctly.

## SVC interface and implementation of daemon

### SVC interface

Let’s explain the reason of this design.

Firstly, whatever application that wants to use SVC must inherit from SVCApp and implement getAppID. This method returns a string which will be used by the SVC daemon to distinguish between applications. Operating in client-server model, one end of communication must self-declare as server by returning isServer of true, the other end will be client.

To perform authentication, the application then has to implement SCVAuthenticator. The principle of authentication is based on “identity” and “proof”, like “username” and “password”. Sometimes, to generate proof, we have to pass some challenge, depending on the algorithm.

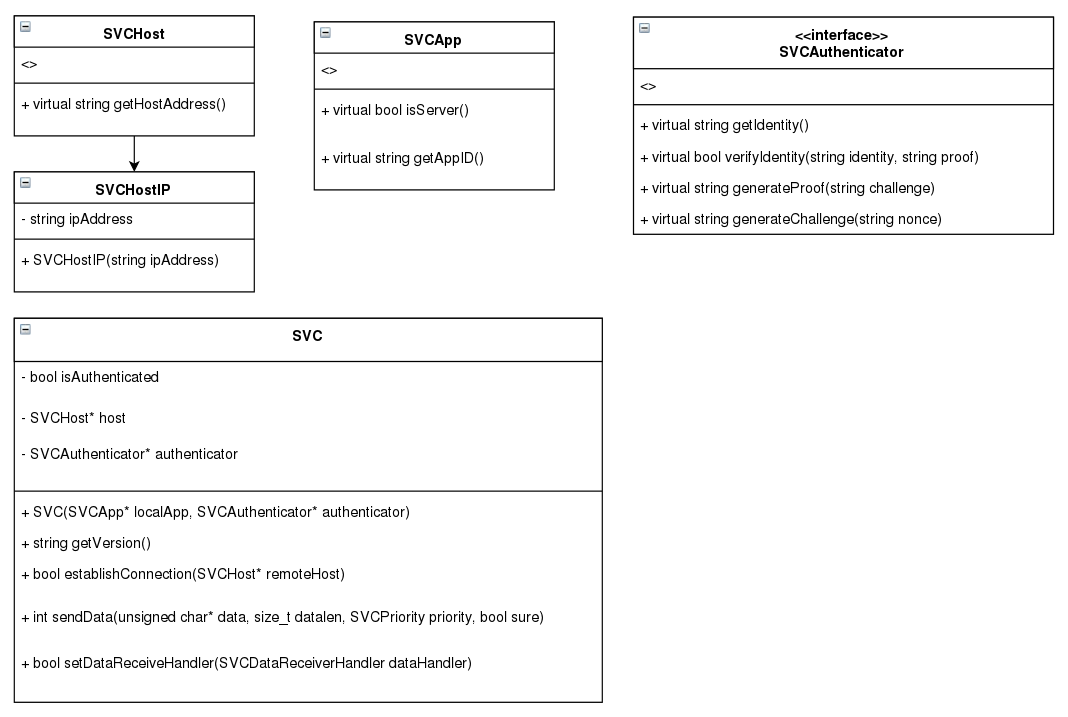


Figure 3.2 – SVC interface and class definition

SVC is designed for an application-to-application communication. On an IP network, an application runs on a host determined by its IP address. SVCHostIP is that way of host identification.

SVC class is the interface between the application and the daemon. SVC maintains a socket connection to the daemon and performs the most basic operations of the authentication and encryption processes. After successfully establishing a connection to a remoteHost, isAuthenticated will be set to true and sendData or setDataReceiveHandler are available to be used.

### The SVC daemon

The SVC daemon is the implementation of cryptographic negotiation and encryption business. The reason for using only one instance of SVC daemon is to keep the logic of non-backward compatibility, and the fact that applications’ binary don’t rely on SVC allow us to update SVC without recompiling any code.

Because of that, SVC implements a table of connected socket from application, associated with an ID to be distinguished. The ID is the first 32-bit of the hash of application ID, given by getAppID. To be able to communicate, the applications on the two ends must provide the same value of getAppID. Collision check will be performed to ensure that there is no duplicated ID.

|  |  |  |
| --- | --- | --- |
| AppID | ID | Socket |
| App1 | 34 AE 76 12 | <<instance1>> |
| App2 | A6 DC 2D 4F | <<instance2>> |
| … | … | … |

SVC daemon also implements the non-blocking priority queues, to profit from the use of HTP. These queues are used as buffers of incoming and outgoing traffics, having the same scheduling algorithm as those of HTP.

SVC daemon runs at port 1122 to process connection request and receive data.

The two mains functionalities of SVC daemon are described in the figure below:

* New connection: handle new connection from an application that uses SVC.
* Negotiation: perform cryptographic negotiation and encryption

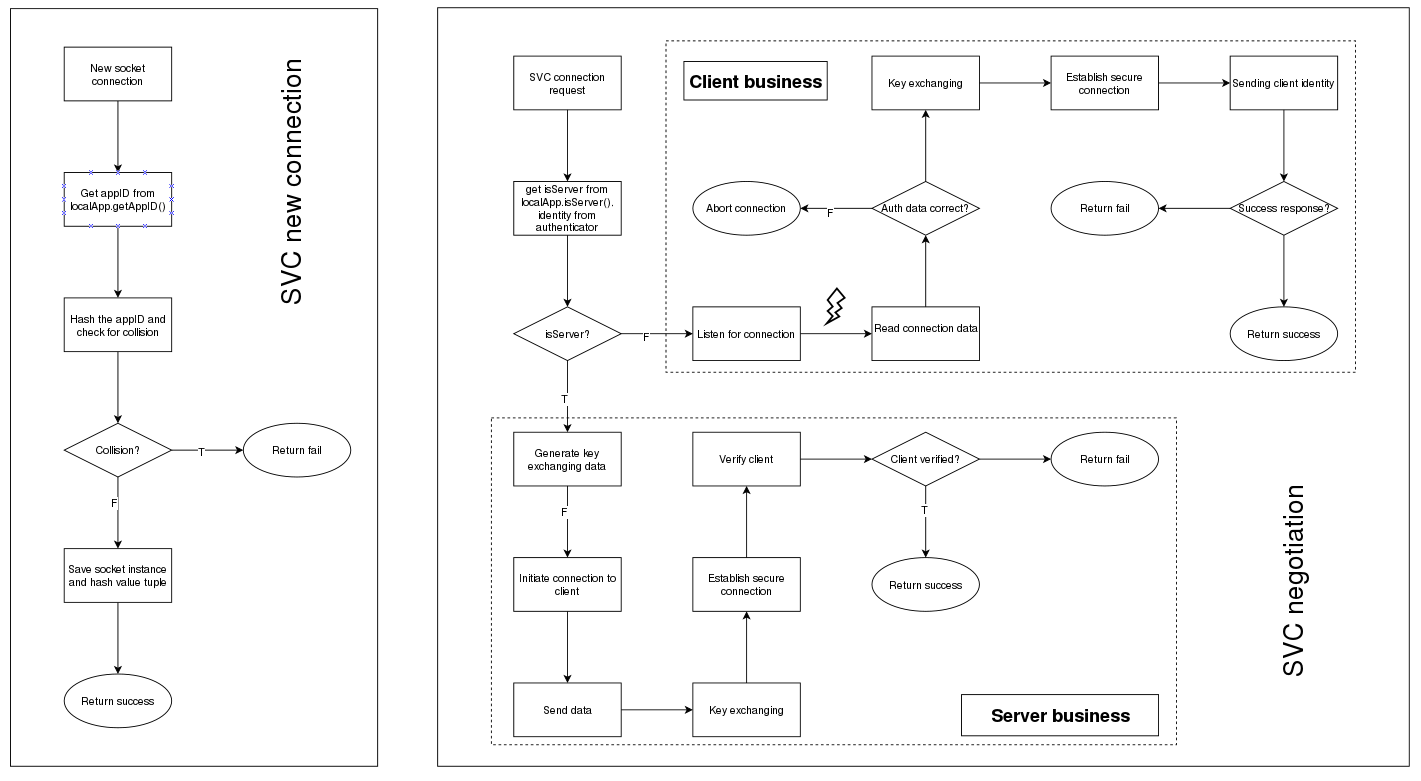


Figure 3.3 – Two main functionalities of SVC daemon