# IST-2000-25350 - SHAMAN

| | |
|---|---|
| **Deliverable Number** | **D11** |
| **Deliverable Title** | **Specification of prototypes** |
| **Date of delivery** | |
| **Document Reference** | **SHA/DOC/ATEA/WP5/D11/v2.0** |

| | |
|---|---|
| **Contractual Delivery Date** | **14-March-03** |
| **Actual Delivery Date** | **14-March-03** |
| **Editor** | **Stefan Goeman (Siemens ATEA)** |
| | **(For list of authors see page 4)** |
| **Participant(s):** | **ATEA, G&D, RHUL, VOD** |
| **Workpackage** | **WP 5** |
| **Est. person months** | |
| **Security** | **Public** |
| **Nature** | **Report** |
| **Version** | **2.0** |
| **Total number of pages** | **58** |

**Abstract:**

This document gives a specification of the SHAMAN demonstrator. Based on some high-level use cases, the critical security aspects of WP1 and WP2 have been identified, and demonstrator scenarios, to be implemented, are derived. The document gives the details of the demonstrator hardware and software architecture. With respect to the hardware architecture, real wireless communications will be used, i.e. Bluetooth and WLAN, and smart cards will be used for the cryptographic computations. With respect to the software architecture, several security related protocols will be implemented, i.e. JFK, EAP-AKA and Imprinting/MANA.

**Keyword list:** authentication, communication security, demonstrator, heterogeneous access networks, imprinting, JFK, key distribution, key exchange, EAP-AKA, MANA, mobility, network access, next-generation mobile networks, Personal Area Network (PAN), Personal CA, PAN Security Domain (PSD), Public Key Infrastructure (PKI), security architecture, security associations, security of wireless links, trust model

The work described in this paper has been supported in part by the European Commission through the IST Programme under Contract IST-2000-25350.

# Executive summary

SHAMAN Deliverable D11 provides a high level specification of the SHAMAN demonstrator. The demonstrator will implement critical components of the security architectures developed in other work packages within SHAMAN. This version of D11 provides an update of the original D11, submitted to the Commission in September 2002, to reflect the content of the demonstrator in March 2003.

The deliverable provides a few high-level use cases describing some scenarios that are relevant to current and future mobile communication systems. From these scenarios, the most critical security requirements are derived and these are further elaborated. The security requirements that have been identified are based on the work done in SHAMAN Deliverable D13 annex 1 [1] on secure access to heterogeneous networks and D13 annex 2 [1] on the security of personal area networks.

We have split the demonstrator work into two phases. In the first phase, we have concentrated on the implementation of new security protocols that were found relevant by other WPs. For WP1 we have focused on the implementation of the JFK protocol while for WP2, we have focused on the implementation of the MANA (MANual Authentication) and imprinting protocols. Two MANA protocols (MANA I and MANA II) have been implemented. The main idea behind the implementation of these protocols is to show the functionality of the smart card for these protocols. So, a reasonable work split between the smart card and the terminal equipment, which was defined in WP4, has been used, and is shown in the demonstrator. In the second phase, we have implemented a second authentication and key agreement protocol, namely EAP-AKA, which is symmetric key based. Additionally, a scenario is implemented where two devices are first imprinted using the MANA/imprinting protocol suite and afterwards, these two devices run the JFK protocols to establish a security association. This scenario shows basic PAN secure communication, or the forming of a simple PSD (PAN Security Domain.).

Further, we also give an overview of the hardware architecture of the demonstrator. We present a high level architecture describing what future mobile communication systems will look like. From this general network, we have derived a hardware architecture for the actual demonstrator, consisting of an access network, and some mobile nodes. The communication between the mobile node and the access network happens via IEEE 802.11 WLAN. The architecture also supports the concept of a personal area network with mobile nodes and other personal area network devices being able to communicate with each other via Bluetooth. An important component in the security architecture developed by SHAMAN is the use of tamper resistant security modules. In this respect the demonstrator will implement critical functionality in smart cards attached to mobile nodes and personal area network devices.

The deliverable also discusses the software architecture of the demonstrator. The demonstrator will consist of three main building blocks: the controller subsystem, the protocol subsystem and the card subsystem. The communication between these building blocks is via a socket interface. In this way, all the building blocks can be designed separately and in principle it should be possible to implement the different building blocks in different programming languages. This is especially important for the protocol subsystem as this subsystem is designed in order to allow easy integration of new protocols. With the card subsystem, besides using a real smart card, it is also possible to simulate a smart card by using a software-based crypto library. This will be used to obtain valuable statistics about cryptographic calculations performed by the smart card.

Three annexes are also included. In the first, further details of the interface between the protocol subsystem and the card subsystem are given. The second annex describes, with accompanying figures, how to configure and use the demonstrator; this is illustrated for the JFK protocol. The third annex includes the SHAMAN leaflet, showing the SHAMAN work and the SHAMAN demonstrator will be introduced at the Cebit fair.

# Authors


| Name | Affiliation | Email | Phone |
|---|---|---|---|
| Goeman, Stefan | Siemens Atea (ATEA) | stefan.goeman@siemens.com | +32 14 25 30 20 |
| Erwin Vackier | Siemens Atea (ATEA) | Erwin.Vackier@siemens.com | +32 244 93 29 |
| Salvatore Castellano | Siemens Atea (ATEA) | Salvatore.Castellano@siemens.com | +32 9 244 93 44 |
| Heinrich, Hans-Juergen | Giesecke & Devrient (G&D) | hans-juergen.heinrich@de.gi-de.com | +49 89 4119 2625 |
| Ertl, Hubert | Giesecke & Devrient (G&D) | Hubert.ertl@de.gi-de.com | +49 89 4119 2796 |
| Howard, Peter | Vodafone Group (VOD) | peter.howard@vodafone.com | +44 1635 676206 |
| Schwiderski-Grosche, Scarlet | Royal Holloway, University of London (RHUL) | scarlet.schwiderski-grosche@rhul.ac.uk | +44 1784 414346 |

# Abbreviations

| | |
|---|---|
| AAA | Authentication Authorisation and Accounting |
| AAAB | AAA-broker |
| AAAH | AAA-server of Home Domain |
| AAAL | AAA-server of the (local) access network |
| ACA | AAA Client Answer |
| ACR | AAA Client Request |
| AP | Access Point |
| APDU | Application Protocol Data Unit |
| API | Application Programming Interface |
| AR | Access Router |
| BAR | BRAIN Access Router |
| BPS | Background Payment System |
| BRAIN | Broadband Radio Access for IP based Networks |
| CA | Certificate Authority |
| CC | Client Challenge |
| CCC | Cost Charging Centre |
| CCS | Credit Control Server |
| CK | Cipher Key |
| CORBA | Common Object Request Broker Architecture |
| COM | Component Object Model |
| CRL | Certificate Revocation List |
| DH | Diffie Hellman |
| DoS | Denial of Service |
| EAP | Extensible Authentication Protocol |
| GPRS | General Packet Radio Service |
| GSM | Global Systems for Mobile Communications |
| GUI | Graphical User Interface |
| HA | Home Agent |
| HC | Home Challenge |
| HD | Home Domain |
| HMAC | Keyed-Hash Message Authentication Code |
| HSS | Home Subscriber Server |
| ICMP | Internet Control Message Protocol |
| IETF | Internet Engineering Task Force |
| IK | Integrity Key |

| | |
|---|---|
| IKE | Internet Key Exchange protocol |
| IP | Internet Protocol |
| IPsec | IP security |
| IPv4 | IP version 4 |
| IPv6 | IP version 6 |
| JFK | Just Fast Keying protocol |
| LC | Local Challenge |
| LV | Length-Value duplet |
| MAC | Message Authentication Code |
| MANA | Manual Authentication protocol |
| MN | Mobile Node |
| NAI | Network Address Identifier |
| PAN | Personal Area Network |
| PANA | Protocol for carrying Authentication data for Network Access |
| PK | Public Key |
| PKI | Public Key Infrastructure |
| PSD | PAN Security Domain |
| RADIUS | Remote Authentication Dial In User Service |
| RMI | Remote Method Invocation |
| SA | Security Association |
| SIM | Subscriber Identity Module |
| SM | Security Module |
| SPI | Security Parameter Index |
| TE | Terminal Equipment |
| TLV | Tag-Length-Value triplet |
| TLS | Transport Layer Security |
| TRD | Tamper Resistant Device |
| UI | User Interface |
| UMTS | Universal Mobile Telecommunications System |
| USB | Universal Serial Bus |
| UTRAN | UMTS Terrestrial Radio Access Network |
| WLAN | Wireless Local Area Network |

**TABLE OF CONTENTS**

# 1 Introduction

The goal of the SHAMAN demonstrator is to prototype critical components of future mobile security architectures, which have been determined by the project. The demonstrator will test the technical feasibility of these critical components and allow the project to gain valuable insight into the practical aspects of the technical solutions being proposed within the project.

The demonstrator will concentrate as much as possible on the mobile security aspects of various user scenarios and will reuse existing implementations as much as possible. With respect to the mobile technology, 'real' technology will be used where available, including smart card technology for implementing critical security components at the user side, and Bluetooth for personal area network communications. Otherwise, PC software emulation tools will be used.

The rest of the document is structured as follows. In Section 2, user scenarios, which form the basis for the demonstrator, are described. Based on the user scenarios, the functions and protocols to demonstrate are identified in Section 3. Section 4 then describes the hardware platform for the demonstrator while Section 5 describes the software architecture. In Section 8 we draw some conclusion.

Also three Annexes are included. The first annex further elaborates on the interface between the card subsystem and the protocol subsystem, while the second annex shows how to configure and use the demonstrator to run the JFK protocol. Finally, in the third annex we include the SHAMAN leaflet showing how the SHAMAN work and the demonstrator will be shown at the Cebit fair.

In the document we refer to work done within the different technical workpackages of the SHAMAN project. The technical workpackages in SHAMAN are listed below for reference:

- WP1 - Security for global roaming in IP-based mobile networks with heterogeneous access networks
- WP2 - Unified security architecture for future mobile terminals and applications
- WP3 - Public Key Infrastructure for next generation mobile telecommunications
- WP4 - Security modules
- WP5 - Prototypes and demonstrators

# 2 User scenarios

In this section we provide a few high-level use cases describing some user scenarios that are relevant to current and future mobile communication systems. From these user scenarios, the most critical security requirements are derived and these are further elaborated. The security requirements that have been identified are based on the work done in SHAMAN Deliverable D13 annex 1 [1] on secure access to heterogeneous networks and in D13 annex 2 [1] on the security of personal area networks.

Due to time and resource constraints, the user scenarios discussed in this section and that have been considered for implementation as part of the demonstrator represent a subset of all the user scenarios under consideration in SHAMAN. The selected scenarios represent the ones that offer the best combination of security features that could be implemented in a realistic environment, given these time and resource constraints. Scenarios that would involve building large-scale network infrastructures were ruled out and instead the scenarios focus on demonstrating security features from the user equipment perspective, in particular those features that involve the user's security module.

## 2.1 Secure access to heterogeneous networks

The work on secure access to heterogeneous networks is discussed in great detail in the WP1 part of SHAMAN Deliverable D13 [1]. A distinction can be made between access based on secret key

techniques and access based on public key techniques. In the following subsections, scenarios will be presented where the authentication and key agreement are based on secret key techniques and where the authentication and key agreement are based on public key techniques. In both cases we assume that the MN has a home domain and that the access network has a roaming agreement with the MN's home domain.

#### *2.1.1.1 Secure access based on secret key techniques*

In D13 annex 1 [1], several examples of secure access based on secret key techniques have been discussed in great detail. It is not our intention to repeat this discussion here. However, from an architectural point of view, all scenarios are more or less the same. The process of initial authentication and key establishment is divided into two parts: first, between the mobile node and an entity in the access network and second, between that entity and an AAA-infrastructure.

One forum within IETF where the first part is dealt with is the PANA (Protocol for carrying Authentication data for Network Access) working group. The goal is to design a protocol that transports authentication data between a node seeking access to a network and an entity in the access network. One candidate protocol to be used here is EAP (Extensible Authentication Protocol). EAP does not specify any authentication method in itself but defines only a transport mechanism, so that concrete authentication methods for EAP are defined separately. For the second part, the interaction with an AAA-infrastructure, the Diameter protocol is preferred [2]. Diameter is a successor to RADIUS, fixing some of RADIUS' limitations. From a DIAMETER perspective, an entity in the access network, called the "attendant" or synonymously the "AAA-client", receives a node's access request and contacts a local AAA-server (AAAL) to perform authentication. If the local AAA-server is not able to authenticate the node, it tries to find out about the node's home domain (e.g. by means of a NAI that the node has supplied in its access request) and forwards the authentication request to the AAA-server in the node's home domain (AAAH). In case the AAAL doesn't know the AAAH, it can contact an AAA-broker (AAAB) that is assumed to be able to find out about the node's home domain. The AAAB may then forward the request on behalf of the AAAL or inform the AAAL about the AAAH's address so that the communication can continue directly between AAAL and AAAH (see Figure 1).

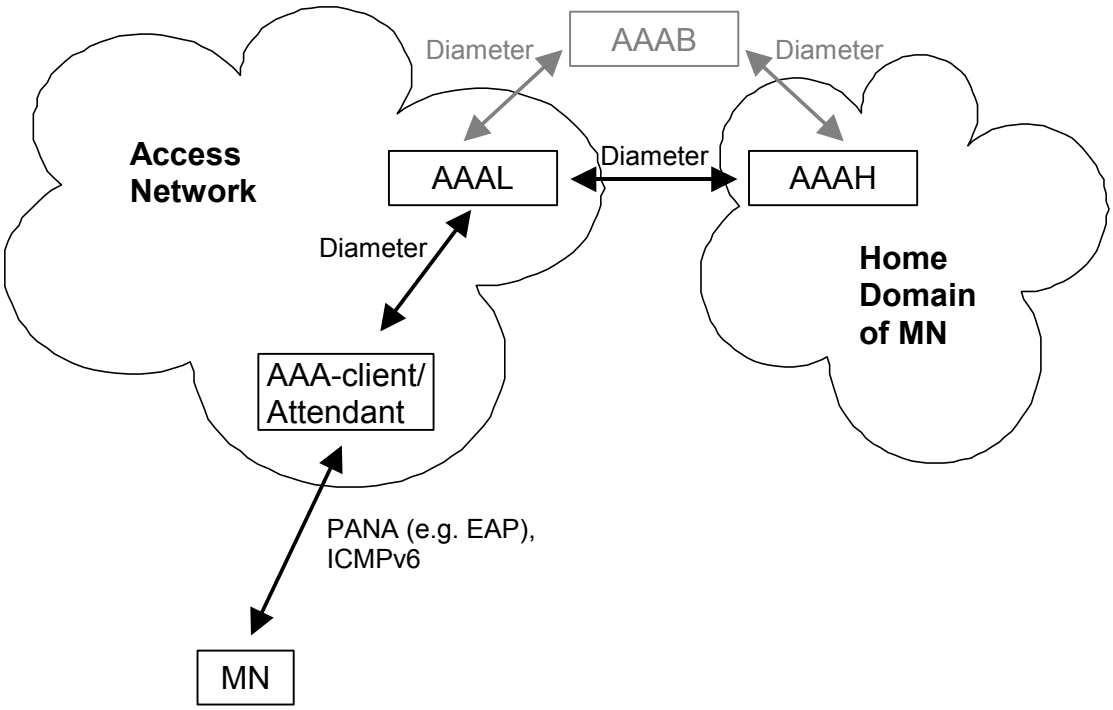


**Figure 1: Protocols and entities in IPv6 network access.**

In the following, we briefly discuss the UMTS AKA protocol as an example protocol for secure network access. In the example we use EAP to transport the UMTS AKA messages to and from the Mobile Node according to the EAP-AKA specification [3]. In Figure 2, we give a schematic overview of the protocol steps. The UMTS AKA protocol is a well-known improvement of the GSM protocol providing additional functionality including mutual authentication to help defeat false-base station attacks. A detailed description of the UMTS AKA protocol and the functionality provided can be found in [4] and the GSM protocol is available at [5]. The protocol described in this section does not consider forwarding multiple authentication vectors to the visited network to allow subsequent re-authentications to be done by the visited network without home network involvement since only the interaction between the client (in our scenario the MN) and the Authenticator (i.e. the BAR (Brain Access Router) in our case) is described. This could be considered as a difference to the UMTS AKA protocol as specified in [4]. The actual processing is done by the EAP server, i.e. by the AAAH. Note that the AAAH server does not necessarily need to play the role of the Authentication Center (AuC), which contains the subscribed user information. The AuC is allowed to forward the authentication vector to the AAAH and so existing infrastructure can be reused. In the subsequent flow only the basic and successful authentication case is described. For message flows describing a sequence number synchronisation failure and various authentication failures [3] can be consulted. Furthermore a description of the IMSI privacy mechanism will not be discussed here. Note that the description in [3] does not describe which protocol is used to carry the EAP messages between the MN and the BAR.

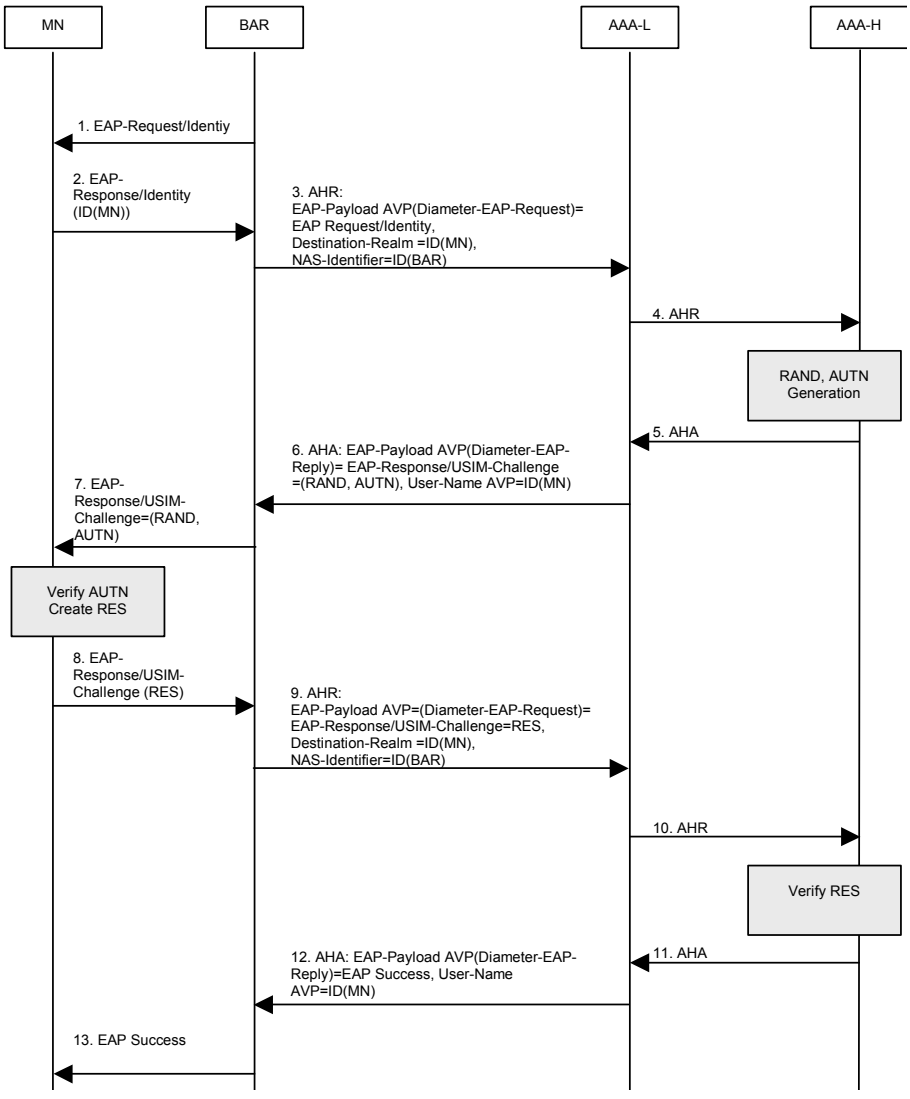


**Figure 2: EAP AKA Authentication.**

The message flow can be summarized as follows:

1. The BAR issues an EAP Request/Identity request towards the MN.
2. The MN answers with the corresponding EAP Response/Identity.
3. At the BAR, a Diameter-EAP-Request is constructed, including the EAP attributes obtained by the MN and other information like Destination-Realm and NAS identifier. This Diameter message is sent towards the AAA-L.
4. The AAA-L further forwards this Diameter-EAP-Request to the correct AAA-H (based on the Destination-Realm) where the user will be authenticated.
5. The AAA-H server runs the UMTS algorithm and generates a RAND and AUTN. The AAA-H server constructs a Diameter-EAP-Response message including the challenge for the MN (RAND and AUTN). This Diameter message is sent towards the AAA-L.
6. The AAA-L further forwards this Diameter message towards the BAR.
7. The BAR sends an EAP-Request/AKA-Challenge to the MN, including RAND and AUTN.
8. The MN run the UMTS algorithm on the USIM, verifies AUTN and derives RES. The RES is sent back to the BAR in an EAP-Response/AKA-Challenge.
9. The similar procedure as before repeats itself, i.e. the BAR translates the EAP-Response in a corresponding Diameter-EAP-request and sends this to AAA-L.
10. AAA-L further forwards the Diameter-EAP-Request to the AAA-H, where the MN can now be authenticated.
11. AAA-H sends a positive Diameter-EAP-Response back to AAA-L.
12. AAA-L forwards this message to the BAR.
13. and the BAR translates this into a corresponding EAP-Success message.

#### *2.1.1.2 Secure access based on public key techniques*

Basically, there are two different cases where public key authentication can be used for network access. In the first case, the “traditional” case, the user of a MN is subscribed to a home network and requires access to an unknown access network that has a trust relationship (i.e. a roaming agreement) with the home network. In this case, the user can exploit his long-term relationship with the home network for gaining network access. The second case is the alternative access case (see [1]) where a user of a MN is not subscribed to a home network, or the home network is not trusted or not accessible by the current access network. Instead, the user aims at securing the wireless link between the MN and the AR using public key techniques.

Here we summarize the different steps:

1. The MN pages for network APs that advertise services for its home operator or another trusted third party (e.g. a roaming broker). The MN announces the identity of this home operator or trusted third party. The access network has a number of public key certificates, at least one signed by each home network or trusted third party with which it has a roaming agreement. The AP selects the right public key certificate and sends it to the MN. By verifying this certificate the MN is assured that a roaming agreement exists between the MN's home network and the foreign network.
2. The MN directs its authentication request to the access network. This message authenticates the MN. There are two main possibilities here, depending on whether the MN’s identity is encrypted with:
    a. *The public key of the home network*: In this case the access network is not able to authenticate the MN. Assuming that the message contains information regarding the

MN's home network, the access network forwards the message to the MN's home network which is able to verify the MN's identity and sends a corresponding message to the access network. In this case, the MN remains anonymous towards the foreign network.

b. *The public key of the foreign network*: In this case the access network is able to verify the identity of the MN. Hence, there is no necessity to involve the home network in the authentication process (except if revocation checking of the MN certificate is required).

So, the MN and the AR need to establish an SA (and generate a secret key from which other keys, for authentication and encryption, can be derived). On the network layer, IPsec provides a solution. Today, in IPsec, these SAs can be established by running the IKE protocol between the MN and the AR. However, IKE is considered a complex protocol. This issue has been recognised by the IETF IPsec working group and two proposed "son-of-IKE" protocols have been discussed, i.e. JFK (Just Fast Keying) and IKEv2. D13 summarizes the usefulness of these two protocols for the scenario of secure access based on public keys. In section 3, the JFK protocol will be discussed further, giving detailed protocol steps.

In order for public keys to add any security, it should be complemented by a public key infrastructure (PKI). Here, although this is not supported in the real demonstrator, we will give a short overview of the basic components necessary to support a PKI for a network access scenario. Moreover, the constraints inherent for this scenario that may effect the deployment of a PKI are listed. However, more complicated PKI issues like certificate revocation and certificate verification are not considered.

In order to support a PKI for secure heterogeneous access, several components are necessary, namely:

- **Certification Authorities (CAs)** – A CA is a trustworthy source that issues public key certificates, that is, it certifies the public keys of users by providing a digital signature binding this public key to a specified end entity. According to [6], certificates should be produced off-line, but can afterwards be stored in a directory, because a certificate is typically a publicly available piece of information. CAs are also responsible for generating certificate revocation lists (CRLs), where a CRL lists certificates that have been revoked prior to their expiration date, e.g. because the private key corresponding to the public key in the certificate has been compromised.
- **Certificate Servers** for storing certificates and CRLs. The home network would typically act as a certificate server towards the access network.
- **Public key users** that need to communicate securely. These users must be able to obtain reliable copies of each other's public keys, e.g. in order to authenticate one another, verify each other's signatures, encrypt messages to each other, and agree on a session key. In our scenario, the entities requiring a public key certificate are the MN, the BRAIN Access Network, and the Home Domain of the MN.

The design of a PKI in the network access scenario (over a wireless interface) is influenced by the following constraints:

- **Storage constraints** – The PKI information to be stored includes public keys, certificates, and CRLs. If the Home Domain of the MN acts as a certificate server, it stores the certificates of all users associated with it. The BRAIN Access Network stores the certificates of all users that are currently attached to it and the MN stores the public key certificate of its Home Domain (the home root key) and his own public key certificate.
- **Processing constraints** – Certificate lists and CRLs are generated off-line. However, verification is usually done on-line. In particular, the PKI operations performed on the MN induce a considerable processing overhead.
- **Bandwidth constraints** – Since PKI-related data, such as certificates and CRLs, may be transmitted over the radio interface, bandwidth constraints must be taken into account.

For more information regarding this scenario, we refer the reader to D13 [1].

## 2.2 Secure personal area network communications

In WP2, a security architecture has been designed to serve the needs for Personal Area Network (PAN) communication (see D13 annex 2 [1]). The security architecture is based on trust relationships between the components. Several component classes can be distinguished, with respect to a reference component:

- **First Party components**: These components have the same owner as the reference component. Furthermore, all these components are able to identify all other first party components and distinguish a first party component from a second party or untrusted component.
- **Second Party components**: A second party component has a different owner than the reference component. Second party component identities can be verified, i.e. authenticated. It is also possible to make a secure key exchange with a second party component. This can be done in several ways, including, but not necessarily requiring, manual interaction by the user(s).
- **Untrusted components**: Untrusted components are by definition all PAN components that the reference component has no security relation with and that it has not (yet) been able to identify and/or verify the identity of. For example, any new component that a user buys is an untrusted component from the perspective of all other components belonging to the same user.

Now, in order for a (reference) component to be able to classify surrounding components into these classes, the PAN components need to have some data that can be used to identify them. The creation of this data and the installation of this data onto the components, is called component initialisation, and is done by manual authentication. Since the PAN situation is different from a general network situation in that one can assume a close physical proximity between the components, it will be possible to transfer this information over a "secure" human channel in addition to the PAN interface connection. In the PAN context, it is assumed that the initial identification of the devices is based on the action of the users of the PAN components. For this purpose a protocol for conveying the result of the identification to the components is required. Using such a protocol, two PAN components can verify that they share the same information, which is relevant for some application to be executed between the devices.

In WP2, manual authentication or imprinting has been studied where the imprinted devices obtain a certificate on a public key (the private/public key pair has first been generated in the PAN component itself). In order to provide this functionality, WP2 and WP3 have developed a new PKI concept called a "Personal CA". The personal CA is used by an ordinary user for home or small office deployment. As with any other PKI, we would like all units in a communication network to share a common root public key and use certificates issued by a trusted CA corresponding to the public root key. One of the personal components must act as a "Personal CA". Such a component is able to issue certificates to all other personal components. Hence, since all the personal components can be equipped with certificates issued by the same CA, i.e. the personal CA, they will all share a common root public key. Consequently, the public keys in the certificates can be used to exchange session keys or authenticate personal components in a PAN.

Then, after being imprinted, two first party components can exchange any information authenticated and confidentiality protected, based on a derived shared secret or on their certificates and the public key of the personal CA. It is possible to run a higher layer authentication protocol based directly on this security association, and derive mutually known authenticated secret keys for confidentiality and integrity protection of the communication. Another possibility is to perform only the authenticated key exchange at the higher layer. The mutual authentication of the components is performed at the link layer, where the keys for integrity and confidentiality protection of the data are derived. This approach is favourable for devices running a wireless link technology with good authentication and key agreement functions. This is illustrated in Figure 3, where after the imprinting step of both devices, a higher layer EAP/TLS based handshake protocol is run to exchange the certificates between the components, and create and exchange a secret key. However, in principle any higher layer key

exchange/ key agreement protocol could be used. This secret key would then be used to derive the necessary link keys for authentication and confidentiality protection at the link layer.

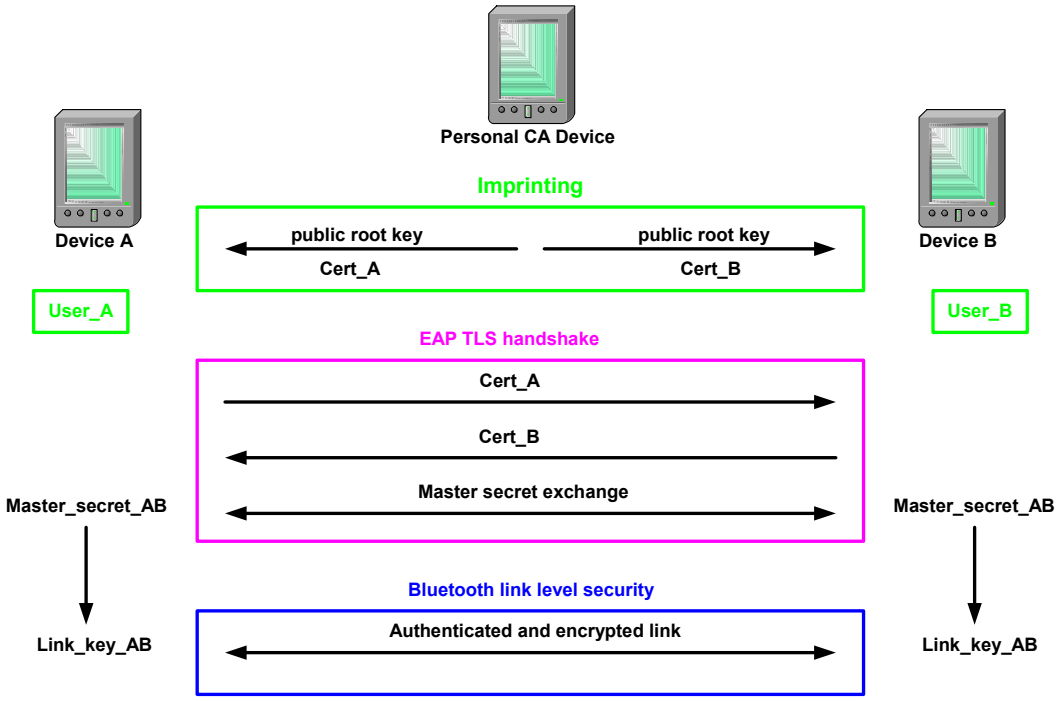


**Figure 3: Schematic overview of the Imprinting protocol and subsequent authentication.**

In addition to this, WP2 has also introduced the concept of a "PAN Security Domain" (PSD). A PSD is "a group of components inside a PAN where each component can be authenticated, trusted and securely communicated by means of some common security association". By definition, all first party components form a PSD in a PAN. The motivation behind introducing a PSD is that all components in a PSD can share a common security policy and have controlled access to each other's resources. Such group services might be needed for a set of components inside a PAN consisting of both first and second party components from the reference component point of view. Hence, a PSD should be dynamically created and dissolved depending on the needs of a PAN "group communication service".

The advantages of a PSD as an extension of the trust model based on first, second and untrusted components are listed below:

- It is not necessary for a new PSD member to share security associations with all existing PSD members in order to establish trusted communications with them.
- Consequent reduction in the user interaction required as the number of imprinting events is reduced. For a PSD of n components, only n-1 imprinting sessions are necessary, compared to n(n-1)/2 without the PSD concept.
- Use of the device with the best UI for enrolling new members allows the most user friendly imprinting protocols to always be used.
- Use of the device with revocation checking facilities allows revocation checks to be performed when new devices with certificates are enrolled.
- Consistent resource information across all devices.
- Resources can be shared with other users without having to compromise interactions between one's own devices.
- Designation of group roles:

  - Designation of a single device to perform the role of a gateway between all PSD devices and external devices.
  - Designation of devices to perform specialised tasks, for example calendar synchronisation, revocation checking.
- Use of the shared security associations to perform secure broadcasts.
- A device can be nominated by the user to perform administrative tasks on his behalf, i.e. a PSD controller.

As already said, the simplest example of group trust is when a single user owns all devices in a PSD and treats them equally (this directly follows from the current trust model when all devices are first party devices). Such a configuration of devices will not contain any restrictions based on the identity of a device. All shared resources will be made available to all the group member devices.

In Figure 4 and Figure 5 we further illustrate the manual configuration of a PSD with devices belonging to different owners. Figure 4 illustrates a PAN containing six devices, designated from A to F. Devices A, B and C are owned by the same user (user 1) while D and E are owned by another user, say user 2. The third user owns device F. Now from the definition of a PAN we know that these devices are capable of communicating with other devices, using local interfaces. And from the trust model we can deduce that devices A, B and C are all first party devices (with respect to each other) while D and E are also first party devices with respect to each other.

Provided that the first user trusts his devices equally, by definition there exists a default PSD involving devices A, B and C. These devices will be able to share resources and communicate with each other securely. Based on this set up, the three devices could be configured to implement a single common policy throughout this default PSD. A similar configuration is possible between devices D and E.

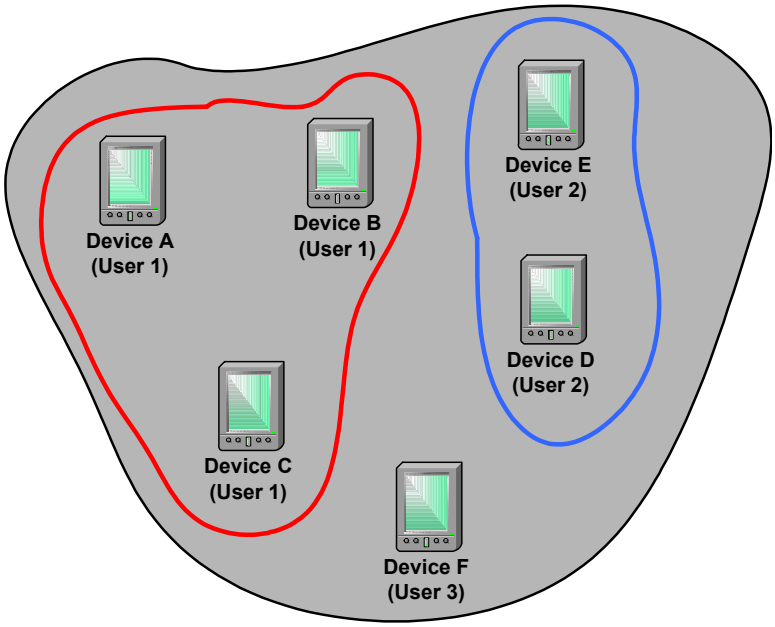


**Figure 4: PAN containing two different PSDs belonging to user_1 and user_2.**

If PSD membership is limited to devices from a single user, two users will not be able to share any resources. The only way this could be achieved is if they reconfigure their existing PSDs such that they have matching policies and trust each other equally. This is not always practical and nor is it safe. Furthermore, from a PSD owner's point of view, restricting resource sharing between his own devices in order to share resources with another user is not a very attractive proposition.

An effective way for the two users to share resources is to establish a new PSD, depending on the situation, this PSD could be a temporary or a permanent PSD (as outlined in yellow in Figure 5) involving the devices with the resources they want to share. Alternatively, the users could pair two

devices (one from each user) and then add further members as required using one of the original devices as the PSD controller.

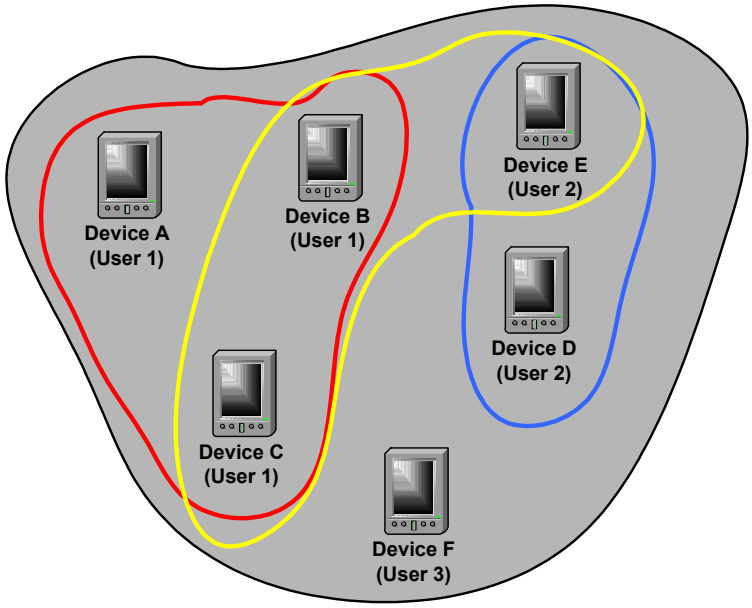


**Figure 5: A temporary PSD is established between user_1 and user_2 in order to share resources between devices B, C and E.**

## 2.3 Security requirements derived from the scenarios

In the above sections, we have presented three distinct scenarios. The first two basically focus on network access, one based on secret key techniques and one based on public key techniques, while the third focuses on secure PAN communication. Although several other scenarios could be imagined, the basic security functions needed to support these scenarios will be the same. In this section, we will summarise these basic security functions.

1. Imprinting of PAN components (scenario 3 only): a mechanism where PAN components with no prior security association interact to establish security credentials that may subsequently be used to establish secure PAN communications.
2. Authentication:
    - Network access security (scenarios 1 and 2): When a mobile node (MN) wants to access a network, the access network is likely to want to authenticate the user. Also, the MN may have the need to authenticate the network.
    - PAN security (scenario 3 only): Also, in PAN scenarios, it is necessary that the communicating PAN components are able to authenticate each other. In this way a PAN component is able to classify surrounding PAN components based on their trust relationships. In order to use these trust relationships, the PAN components need to be imprinted.
3. Secret Key Establishment:
    - Network access security (scenarios 1 and 2): In order to be able to protect the communication between the MN and the access network, a secret key has to be established, in a secure way, between the MN and the access network. This is typically done as part of the authentication protocol.

- PAN security (scenario 3 only): In a PAN, the components also need to establish secret keys among each other, to secure their communication. Moreover, this secret key that is negotiated on a higher layer can be used to derive the Bluetooth link layer keys. The key establishment is typically done as part of the authentication protocol.

4. Secure communications:

   - Network access security (scenarios 1 and 2): To protect the communication between the MN and the access network, the established secret key will be used to derive corresponding encryption and integrity protection keys in order to protect the communication channel.
   - PAN security (scenario 3 only): To protect the communication between PAN components, the established secret key is used to derive encryption and integrity protection keys, in order to protect the communication channel.

In the following chapter, we will further discuss these basic security functions and give a detailed description of how they will be realised in the demonstrator.

# 3 Overview of demonstrator functions

In this section, we list the functions to be demonstrated. However, as it is not expected to be able to implement all the functions due to the limited time frame, the work has been prioritised and split into two phases:

- Phase 1 (baseline demonstrator): The following functions will be implemented:
    - The JFK authentication and key agreement protocol.
    - The Imprinting and MANA protocols.
- Phase 2 (enhanced demonstrator): The following functional enhancements will be considered in the following order of priority:
    - A secret key based authentication and key agreement protocol.
    - A simple PSD scenario: Two first party components, previously imprinted by a PAN CA, establish a security association.

General requirements:

- The demonstrator should allow protocols to be run on the same PC or between two PCs.
- Where the protocol is run between two PCs it shall be possible to use either a fixed connection, WLAN or Bluetooth.
- The demonstrator should provide a graphical user interface (GUI).
- The demonstrator should implement critical security functions at the user side on a smart card attached to the PC. It should also be possible to simulate the smart card in software.

## 3.1 Baseline demonstrator

### 3.1.1 JFK

The JFK protocol is specified in [7] and has been studied in greater detail in D13 annex 3 [1]. As identity confidentiality for the network side is not required the JFKi variant is implemented in the demonstrator, with the mobile node as the initiator (I) and the network as the responder (R). To prevent linking of authentication exchanges involving the same mobile node, $g^i$ is changed for each protocol run. Critical functions at the initiator side are also implemented in a real smart card. Two options for splitting the functionality between the smart card and the mobile node have been implemented.

The open source implementation available at [8] has been adapted in order to fit in the demonstrator architecture.

Security association negotiation for IPsec and Bluetooth link layer security are not implemented. Only session key establishment is demonstrated.

The notation used in the description of JFK is presented here.

| | |
|---|---|
| $E\{K\}(M)$ | Encryption of $M$ with secret key $K$. |
| $HMAC\{K\}(M)$ | Keyed hash of $M$ using key $K$ in an HMAC scheme. |
| $SIG\{x\}(M)$ | Digital signature of $M$ using the private key belonging to principal $x$ (Initiator or Responder). It is not assumed to be a signature with message recovery (but it can be). |
| $G^x$ | Diffie-Hellman exponentials, also identifying the group-ID. |
| $g^i$, $g^r$ | Initiator and Responder exponentials. |
| $N_i$ | Initiator nonce, a random bit-string. The Initiator must pick a fresh nonce at each invocation of the JFK protocol. |
| $IP_i$ | The Initiator's network identifier (IPv4 or IPv6 address). It is used by the Responder to counter a "cookie-jar" attack[1], when verifying the authenticator upon receipt of message 3. |
| $N_r$ | Responder nonce, a random bit-string. The Responder must pick a fresh nonce at each invocation of the JFK protocol. The nonces are used in the session key computation, to provide key independence when one or both parties reuse the same Diffie-Hellman exponential; the session key will be different between independent runs of the protocol, as long as one of the nonces or exponentials changes. |
| $Sa$ | Defines the cryptographic and other properties of the Security Association (SA) the Initiator wants to establish. It contains a Domain-of-Interpretation, which JFK understands, and an application-specific bit-string. |
| $Sa'$ | Any information that the Responder needs to provide to the Initiator with respect to the application SA (e.g., the Responder's SPI, in IPsec). |
| $HK_r$ | A transient hash key private to the Responder; this is a global parameter for the Responder (i.e., it is not different for every different protocol run), which changes periodically: the Responder must pick a new $g^r$ every time $HK_r$ changes. |
| $K_{ir}$ | A shared key derived from $g^{ir}$, $N_i$, and $N_r$, used as part of the application SA (e.g., IPsec SA). |
| $K_e$ | A shared key derived from $g^{ir}$, $N_i$, and $N_r$, used to protect messages 3 and 4 of the protocol. Although the input parameters are the same with $K_{ir}$, a different key derivation mechanism is used to ensure key independence. |
| $ID_i$, $ID_r$ | Initiator and Responder certificates or public-key identifying information. Multiple such payloads may appear in a message, to indicate multiple certificates, CRLs etc. For simplicity and clarity, the notation in this draft shows only one such payload per message. |
| $ID_{r'}$ | An indication by the Initiator to the Responder as to what identity (and corresponding key material) the latter should use to authenticate to the former. The Responder may ignore this hint. This field may contain the certificate of a CA trusted by the Initiator (which means that the Initiator is requesting that the Responder authenticate with a certificate chain "rooted" at that CA), or the certificate of the Responder (effectively identifying the public, and corresponding private, key of the Responder). |
| $GRPINFO_r$ | A list of all Diffie-Hellman groups supported by the Responder, the symmetric algorithm used to protect messages 3 and 4, and the hash function used for key |

[1] An attacker accumulates lots of cookies from lots of IP addresses over time and then replays them all at once to overwhelm the Responder.

| | |
|---|---|
| | algorithm used to protect messages 3 and 4, and the hash function used for key generation. |
| $GRP\_ID_i$, $GRP\_ID_r$ | Identification of the Diffie-Hellman group used by the Initiator ($GRP\_ID_i$) and the Responder ($GRP\_ID_r$) |

**Protocol JFKi**

Message_1, I->R: $N_i, g^i, GRP\_ID_i, ID_r$,

Message_2, R->I: - $N_i, N_r, g^r, GRPINFO_r, GRP\_ID_r, ID_r$,

$SIG\{r\}(g^r, GRPINFO_r)$,

$HMAC\{HK_r\}(g^r, N_r, g^i, N_i, IP_i)$

- or a rejection message: $GRPINFO_r$

Message_3, I->R: $N_i, N_r, g^i, GRP\_ID_i, g^r, GRP\_ID_r, HMAC\{HK_r\}(g^r, N_r, g^i, N_i, IP_i)$,

$E\{K_e\}(ID_i, sa, SIG\{i\}(N_i, N_r, g^i, g^r, ID_r, sa))$

Message_4, R->I: - $E\{K_e\}(SIG\{r\}(N_i, N_r, g^i, g^r, ID_i, sa, sa'), sa')$

- or a rejection message

The key used to protect Messages (3) and (4), $K_e$, is computed as $HMAC\{g^{ir}\}(N_i, N_r, 1)$. The session key used by IPsec (or any other application), $K_{ir}$, is $HMAC\{g^{ir}\}(N_i, N_r, 0)$.

#### *3.1.1.1 Activities at the Initiator side*

In the JFK protocol, most of the work is left for the Initiator. The idea behind this is that the Responder, typically a server, should not create any state information and perform heavy cryptographic calculations before he is assured that he is connected with a real Initiator; this to prevent denial of service attacks (DoS). So, the Initiator can actually be defined by a finite state machine (Figure 6). In the following, we will describe the activities of the Initiator.

- In state_1: The initiator generates message_1, containing a key in a certain DH-group that he hopes will be accepted by the Responder. Message_1 also contains a nonce and an indication as to which ID the Initiator would like the Responder to use for authentication towards the Initiator. The Initiator then enters state_2 and waits for message_2 from the Receiver.
- On receipt of message_2 (in state_2): The Initiator checks the message type that was received.
    - If it is a rejection message coming from the Responder, the Initiator returns to state_1 and creates a new message_1, to be sent to the Responder, with a DH-key in a group supported by the Responder. Preferably, also a new nonce $N_i$ is generated.
    - If it is a real message_2 type, the Initiator checks whether the $N_i$ received is the same as the one he has sent in message_1 (it could be that the responder is answering to an older message_1, sent by the Initiator). If this is not the case, the Initiator returns to state_1 and restarts the protocol run. Then, the Initiator checks if he still supports the DH-group of the Responder (it could be that during different protocol runs, the Initiator has changed his DH-group), If this is not the case, the Initiator returns to state_1 and restarts the protocol. Then, the Initiator checks the Responder certificate and the digital signature. If there is some error here, the Initiator returns to state_1 and restarts the protocol. If all the checks complete successfully, the initiator enters state_3
- In state_3: When the Initiator enters state_3, he is assured that a valid message_2 message has been received from the Responder. Now, the Initiator creates message_3. Basically, message_3 echoes back the data sent by the Responder, including the authenticator. The

message also includes the Initiator's identity and a service request (the actual security association sa to be created), a signature computed over the nonces, the two exponentials and the Responder's identity. This latter information is all encrypted under a key derived from the DH-computation and the nonces $N_i$ and $N_r$. the encryption and authentication use algorithms specified in $GRPINFO_r$. After sending message_3, the Initiator goes into state_4 and awaits the response.

- On receipt of message_4 (in state_4):
  - The Initiator checks if he has received a message. If there is something wrong with the general format of the message, the Initiator returns to state_3 and resends message_3.
  - If he did receive a valid message, the Initiator checks whether it is a rejection message. If this is the case, the Initiator returns to state_1 and restarts the protocol.
  - If the received message is actually a message_2 (retransmitted due to security problems), the Initiator returns to state_2 and processes this message_2.
  - If the received message is a real message_4, the Initiator decrypts the message, checks the signature and checks the negotiated SA, if there is something wrong here, an unrecoverable error has occurred and the protocol reports an error condition. Consequently, no SA has been set up between Initiator and Responder. If nothing went wrong, which should normally be the case, a SA has been achieved and set up between the Initiator and the responder.

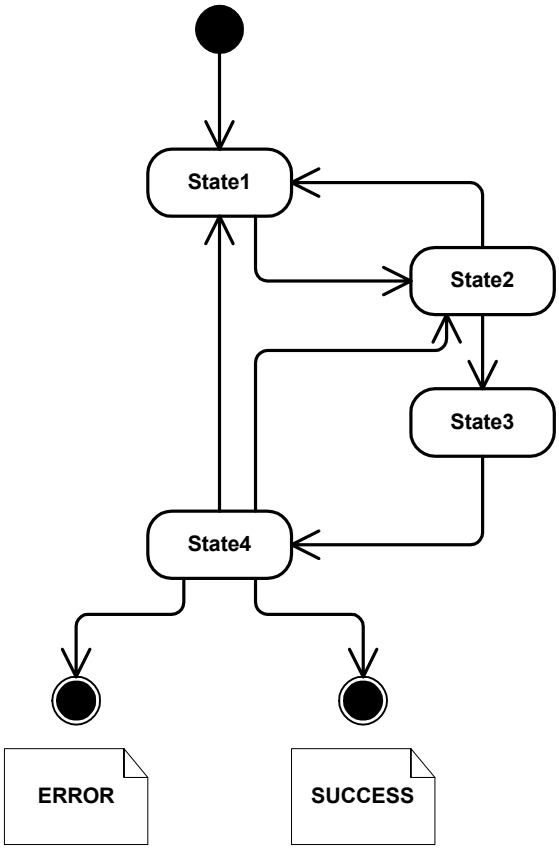


**Figure 6: JFKi Initiator state machine.**

#### *3.1.1.2 Activities at the Responder side*

- On receipt of Message_1: at the Responder side, the responder checks if he supports the DH-group suggested by the Initiator. (Note that the Responder does not need to create any state information at this point, the only "expensive" operation the Responder needs to do is a MAC calculation.)
    - If the DH is not supported, a rejection message is send back, including the DH-groups supported by the Responder.
    - If the Responder supports the proposed DH-group, he creates message_2 and sends it to the Initiator. Now, the Responder replies with a signed copy of his own exponential (in the same group), information on what secret key algorithms are acceptable for the next message, a random nonce, his identity (certificates or a bit string identifying his public key), and an authenticator calculated from a secret key $HK_r$ only known to the Responder. This authenticator is computed over the two exponentials and nonces, and the Initiator's network address. The authenticator key is changed at least as often as $g^r$, thus preventing replays of stale data.
- On receipt of message_3
    - The Responder checks whether the hash is contained in any blacklist maintained by the Responder. Received hash values (from message_3) are being put in this blacklist when there is something wrong with the decryption of the encrypted data, or when there is something wrong with the certificate of the Initiator or with the signature of the Initiator. If the received hash is indeed in the blacklist, the Responder stops processing this message and does nothing anymore. It is the task of the Initiator to detect problems and to restart the protocol, if needed.
    - If the received hash is not in the blacklist, the Responder continues processing the message. The Responder recomputes the hash with the data provided in the message and compares this with the hash value contained in the message. If both hashes are not equal, something wrong has occurred and the Receiver stops processing the request.
    - If the hashes are equal, the Responder continues with decrypting the encrypted part of message_3 and checks the certificate of the Initiator and the signature. If something goes wrong here, the hash value contained in message_3, is put in a blacklist.

The Responder keeps a copy of recently received message_3's and their corresponding message_4. Receiving a duplicate (or replayed) message_3 causes the Responder to simply retransmit the corresponding message_4, without creating a new state or invoking a new IPsec SA. If message_3 is really new, a message_4 is created and sent to the Initiator. The (message_3, message_4) pairs are stored at the Responder side, to handle duplicate (or replayed) messages_3 as fast as possible. Message_4 contains application specific information (such as the Responder's IPsec SPI), and a signature on both nonces, both exponentials, and the Initiator's identity. Everything is encrypted by $K_e$, which is derived from $N_i$, $N_r$ and $g^{ir}$ (the result of the DH-computation).

#### *3.1.1.3 Functional split of the JFKi protocol between the MN and the TRD*

The functions at the initiator side may be split between the MN and the TRD. In the following we explain the two options that are described with reference to the functions implemented on the TRD, and that have been implemented in the demonstrator. The options are listed in order of increasing functionality on the TRD. The following notation is used:

I JFK initiator (Mobile Node)

T TRD (Tamper Resistant Device) attached to JFK initiator

R JFK responder (Network)

A short security analysis of the 2 options is given at the end of the paragraph.

Option 1: Signature generation only

In this option, the initiator's private key, which is stored on the TRD, is used to sign input data. The input data consists of the hash of $N_i$, $N_r$, $g^i$, $g^r$, $ID_r$ and sa. In order to prevent chosen-hash attacks (see discussion later on), the incoming hash data are hashed once again on the TRD before they are signed. The hash algorithm is SHA-1. The signed data are returned from the TRD to the initiator.

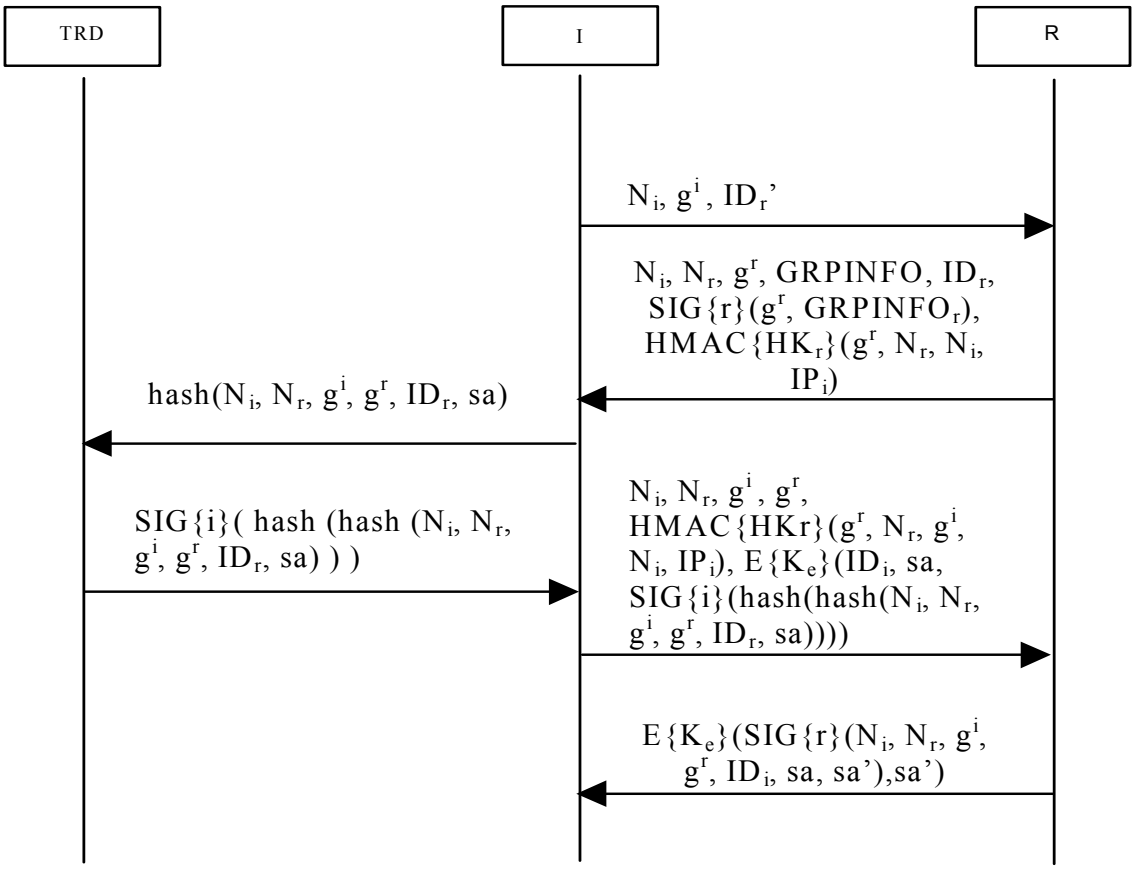


**Figure 7: JFK functionality split, option 1.**

Option 2: Signature generation, Diffie-Hellman exponentiation and key generation

In this option, the TRD generates a signature as in option 1 (after rehashing the incoming data), but it also generates the common Diffie-Hellman secret and the integrity and encryption key. The TRD initiates the protocol run by sending a nonce $N_i$ and its Diffie-Hellman value $g^i$ to the initiator. On receipt of the responder's message, it rehashes and signs the incoming hashed data and uses $g^r$ and $N_r$ to calculate the temporary keys $K_e$ and $K_{ir}$.

There is also a rekeying option which can be used for speeding up further protocol runs. Besides $K_e$ and $K_{ir}$, the TRD also calculates a key $K_s$ as follows: $K_s$ = HMAC{$g^{ir}$}($N_i$, $N_r$, 3). In the following protocol run, instead of a signature, a HMAC of the rehashed input data can be calculated, using $K_s$ as the HMAC key. The HMAC is faster than a signature operation. It has to be made sure that the $K_s$ which has been calculated from the last protocol run is used. In the demonstrator, the rekeying option is always used if the JFK protocol is run for the second or more consecutive time.

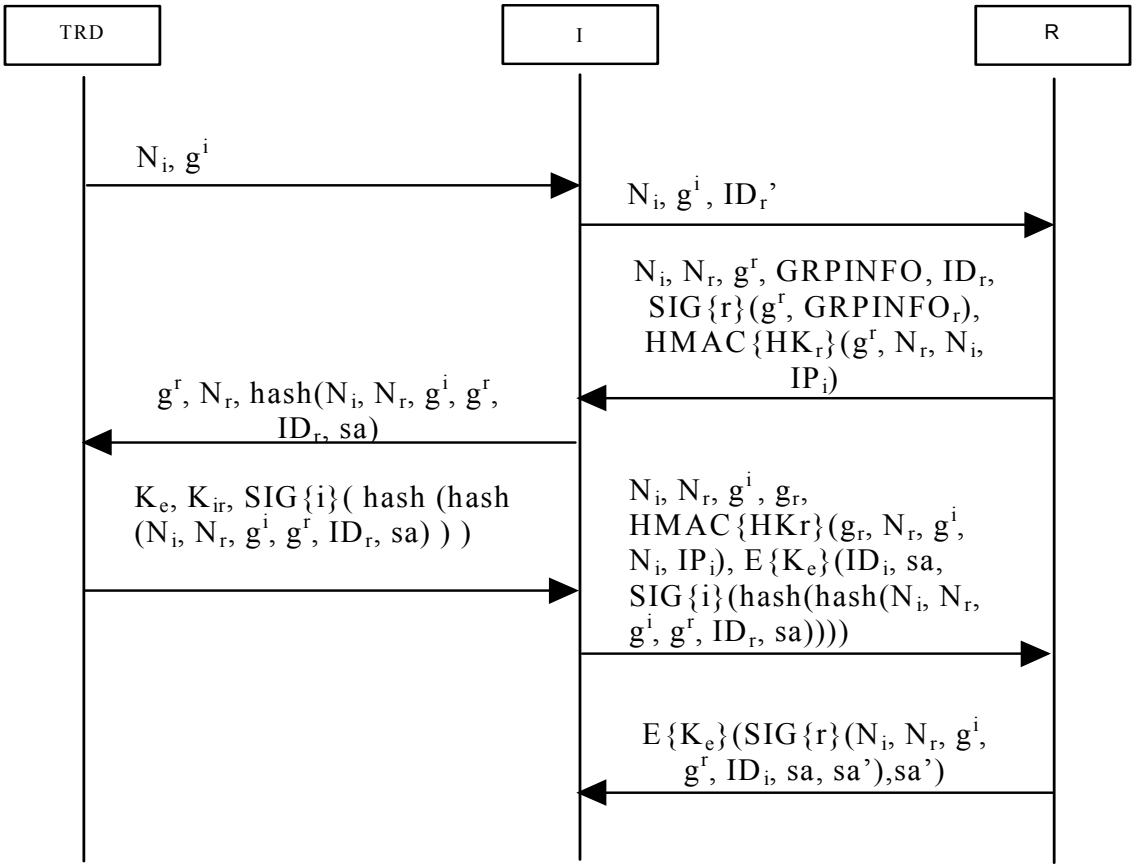


**Figure 8: JFK functionality split, option 2.**

In the demonstrator, the JFK protocol can be run with or without the TRD (this means also without a TRD simulation). If the TRD is used, the Diffie-Hellman group is always MODP2, as specified in [9]. Therefore, no GRPINFO has to be evaluated by the TRD.

For the signature, either PKCS1-padding (this is the only option when using the TRD simulation) or zero padding (in the case of a real TRD or no TRD use at all) for the input data can be used.

Security analysis

It is obvious that the private key stored on the TRD – which is a long-term secret - should never be revealed to the outside, so it is mandatory that signature operations on the initiator side take place on the TRD. Both options fulfil this requirement. The input data for the signature is hashed before signing in order to reduce the demands on storage and bandwidth for the TRD (which is usually a limited device). It was shown above that the input data for the TRD are rehashed before the actual signing operation takes place. This significantly reduces the probability of a chosen-hash attack, where an attacker sends chosen byte streams to the TRD in order to gain knowledge about the stored private key; if the incoming data is rehashed before signing, the attacker cannot arbitrarily choose the byte patterns being signed.

If $g^i$ changes for every protocol run, option 1 is sufficient to protect the only long-term secret (the private signature key). However, if rekeying shall be used, it must be made sure that rekeying is only possible in the presence of the TRD; for this purpose, Diffie-Hellman exponentiation and key generation are required, as in option 2.

Other options, like the encryption of user identity on the TRD, seem not very beneficial because the corresponding unencrypted data must be at the disposal of the initiator I anyway.

It could turn out useful in practice to have the TRD provide certificates; however, in the demonstrator this option has not been realised, since it offers no substantial security advantage.

### 3.1.2 Imprinting and MANA protocol

A personal area network may contain components owned by different users. The components are either trusted or untrusted. In this section we describe procedures for setting up an initial security association between components.

A component has a pre-determined task in the PANs it is going to participate in. The act of component initialisation is always preceded by a step where the component is identified and authorised as suitable and trustworthy to perform its task. The procedures of initial authorisation consist typically of non-cryptographic security methods.

The cryptographic initialisation or what is also called *imprinting*, is a procedure of equipping the component with a secret value of a cryptographic parameter. The type of parameter varies a lot depending on its subsequent use in security mechanisms. It may be an unstructured randomly generated string, from which future key material is derived. It may also be a key related to an asymmetric cryptographic mechanism, in which case it has a well-defined structure. The procedure of imprinting, where the initial secret cryptographic parameters are set in the component, is the most sensitive part of the communication. If tampering or eavesdropping the imprinting step is possible, then the security of all future communication based on imprinting is ruined. Therefore imprinting requires some auxiliary temporary secure channel to be used. In the PAN context, the secure channel can be based on one of the following technologies:

- Fixed connection such as cable, USB interface, bar-code reader, smart-card reader
- Human involvement, communication of passkeys, entering passkeys
- Second wireless (e.g., infrared)
- Other proximity based technology (low power channel)

The purpose of imprinting is to equip a PAN component with a cryptographic piece of information by which it can be authenticated by other first-party components at least. When two different-party or untrusted (not yet imprinted) PAN components communicate for the first time, they do not have any logical means to verify each other's identity, or the origin of the exchanged data.

In PAN context it is assumed that the initial identification of the devices is based on the action of the users of the PAN components. For this purpose a protocol for conveying the result of the identification to the components is required. Using such a protocol two PAN components can verify that they share the same information, which is relevant for some application to be executed between the devices.

In the WP5 demonstrator, we have implemented the imprinting protocol, based on an underlying public key system. This means that the PAN components will use some public key system as a basis for securing the internal PAN communication. This also means that each PAN component has its own private key and public key, and the public key must be certified in such a way that other PAN components can verify the validity of the certificate. In Deliverable D13 annex 3 [1] definitions and requirements for a Personal Certification Authority (the "Personal CA" concept) have been set. For this imprinting protocol to work, we also need some form of Manual Authentication Protocol (MANA). With this protocol, the user of both components can verify that the two components share exactly the same data items. This MANA protocol will be used inside the imprinting protocol; and the data items will include the public key of the Personal CA, the public key of the other component, to be imprinted, and any other information this component wants to see added in his public key certificate, which must be created by the Personal CA component.

We will first describe the MANA protocols (MANA I and MANA II) that have been implemented in the demonstrator. Afterwards, we will give more details on the imprinting protocol.

#### *3.1.2.1 The imprinting protocol for a Public Key System*

In D13 annex 2 [1], two different cases of imprinting for a Public Key System are described. In the first case, the components, to be imprinted, generate their own public and private key pair whereas in the second case, the keys are generated by some external key management facility and then imported to the components. This second case could be useful for components that don't have the computational capability of generating a key pair, however this scenario also introduces new security problems (somehow, one must be assured that the key generating facility does not misuse the private keys it generates destined for other components). For the WP5 demonstrator, we have chosen the first option.

The imprinting protocol then operates as follows:

1. The PAN CA must be reliably informed of the identifier for the PAN component. This could, for example, be achieved by the user typing the identifier for the PAN component into the keyboard of the PAN CA. However, it could also be achieved as part of the protocol itself (see below).
2. The PAN CA sends its public key $P_{CA}$ to the PAN component, and the PAN component sends its public key $P_M$ to the PAN CA. This transfer is assumed to take place via the wireless interface. Along with $P_M$, the PAN component can send any other information it wishes to have included in the public key certificate that the PAN CA will generate (again via the wireless interface). This could, for example, include the identifier for the PAN component.
3. The PAN CA, the PAN component and their users now perform the MANA protocol to verify that the exchanged public keys are correct. The PAN CA takes the role of the first device. The data *D,* using which the MANA protocol is performed consists of $P_{CA}$, $P_M$ and any other data supplied by the PAN component and the PAN CA. This additional data may include the unique identifiers of the PAN CA and the PAN component.
4. If (and only if) the PAN component emits a success indication, the user instructs the PAN CA to generate an appropriate public key certificate. This certificate generation must only take place **after** the PAN component has given the required positive indication. This certificate can then be sent (unprotected) to the PAN component via the wireless interface.
5. The PAN component now performs two checks before accepting the certificate. Firstly the PAN component checks the signature using the PAN CA's public key ($P_{CA}$). Secondly the PAN component verifies that the data fields within the certificate (including the public key $P_M$ and the identifier for the PAN component) are all as expected. The protocol is now complete.

#### *3.1.2.2 The MANA I protocol*

We assume that two components have exchanged some information consisting of data items D (see section 3.1.2.1). Here we describe the Manual Authentication Protocol I (MANA I) by which the users of the components can verify that the two components share exactly the same data items D. In principle, the protocol can be adapted for two different situations depending on the types of interfaces the devices have available for this purpose. The underlying requirements with respect to I/O-interface are the following:

1. The first component (this will be the Personal CA) has an output interface suitable for short strings of alphabetic and numeric symbols and a simple input interface.
2. The second component, the component to be imprinted, has an input interface suitable for short strings of alphabetic and numeric symbols and a simple output interface.

The protocol steps of the MANA I protocol are as follows (Figure 9):

1. Both components output a signal as to acknowledge that they have received data *D* and that they are now ready for the verification. User receives the signal from both components and generates a short random key *K*, where *K* is suitable for use with a MAC function shared by the two components. The user enters a signal in one of the devices (called the first component in the sequel) to notify that the protocol can start.

2. The first component generates a random key *K*, where *K* is suitable for use with a MAC function shared by the two components. Using this key *K*, the first component computes a MAC as a function of the data *D*. The MAC and the key *K* are then output by the output interface of the first component. The user now reads the MAC and key *K* from the output of the first component.
3. The user enters the output of the first component to the second component using the input interface. The second component uses the key *K* to recalculate the MAC value as a function of its stored version of data items *D*. If the two MAC values agree then the second component outputs a success signal to the user. Otherwise it gives a failure signal.
4. The user enters the result to the first component. In case of success, the components accept the data *D*.

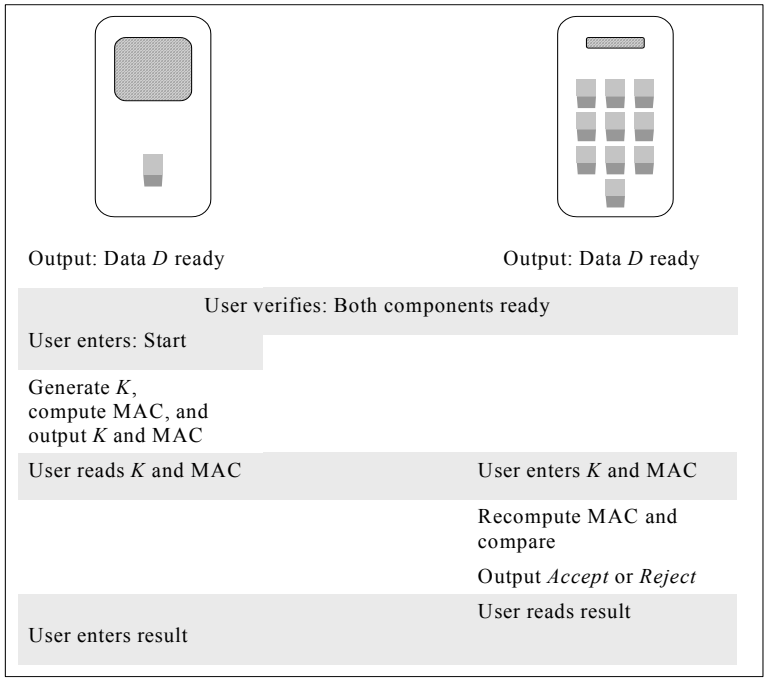


**Figure 9: Manual Authentication Protocol MANA I.**

### *3.1.2.3 The MANA II protocol*

In this section we will briefly discuss the message flow of the MANA II protocol. The same assumptions hold as for the MANA I protocol. The protocol steps of the MANA II protocol are as follows (Figure 10):

1. Both components output a signal as to acknowledge that they have received data *D* and that they are now ready for the verification. User receives the signal from both components and enters a signal in one of the devices (called the first component in the sequel) to notify that the verification can start.
2. The first component generates a random key *K*, where *K* is suitable for use with a MAC function shared by the two components. Using this key *K*, the first component computes a MAC as a function of the data *D*. The component outputs the MAC, and transmits the key *K* to the second component over the insecure wireless link.
3. The second component uses the key *K* to compute the MAC value as a function of its stored version of data items *D*, and outputs the key *K* and the computed MAC value.

4. The user compares the two MAC values. If they agree, then the user enters a signal of acceptance in both components, and the components can accept data *D* as the basis of their subsequent operation. If the MAC values are different then data *D* must not be accepted.

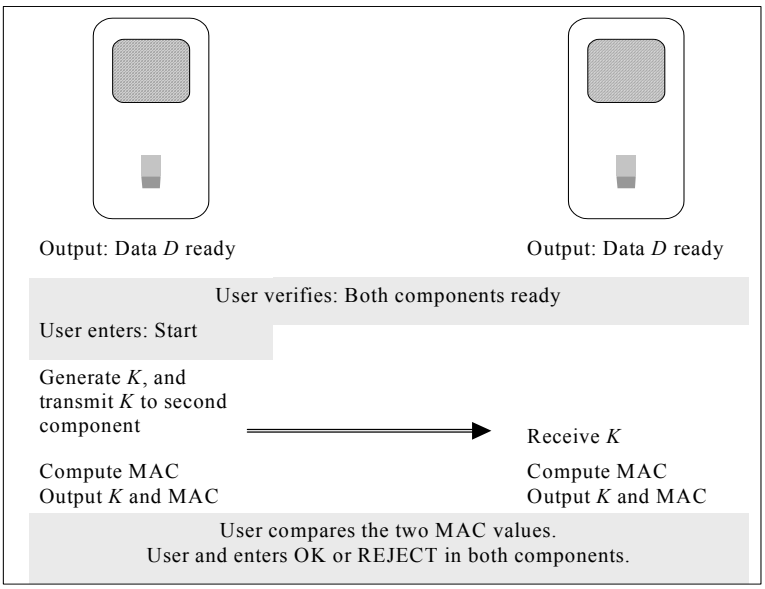


**Figure 10: Manual Authentication Protocol MANA II**

#### *3.1.2.4 Functional split of the protocols between the MN and the TRD*

All MANA functions are assumed to be implemented in the PAN component. However certain critical functions performed by the personal CA as part of the imprinting protocol may be implemented in a TRD. In particular the TRD may be involved in certificate generation.

In the demonstrator, we realised the option to let the TRD sign certificate requests. As in the JFK functionality split, this has the obvious benefit that the private key stored on the TRD, which corresponds to the private key of the PAN CA, is not revealed to the outside.

It would also be convenient to let the TRD construct the certificate request itself. However, this doesn't add significantly to security, and is therefore not implemented.

## 3.2 Enhanced demonstrator

### 3.2.1 The simple PSD scenario

As a first step towards an enhanced demonstrator, the "simple" PSD scenario, as already outlined in section 2.2, has been implemented in the demonstrator.

In order to demonstrate this scenario, three PAN components are necessary. As a basic minimum, two components could be used as well; but in this case, the PAN CA is reused in the key agreement protocol. The scenario can be summarized as follows, where we assume that three PAN components (A, B and the PAN CA) are used:

1. The first PAN component A is being imprinted with the use of the PAN CA. During this step, component A obtains the root public key of the PAN CA and a certificate binding the component's identity to its public key.
2. The second PAN component B is also being imprinted via the PAN CA. This component also obtains the root public key of the PAN CA and a certificate binding its identity to its public key.

3. Now, both components A and B have the root public key and a certificate over their public keys. Both components run a key establishment protocol by which they will establish a secret key to be used as the basic key to derive other key for integrity protection and confidentiality protection. For this key establishment protocol, we have reused the JFK protocol. In principle, after the JFK run, the components A and B have agreed on a session key. This key could be further used to derive Bluetooth link layer keys, in case Bluetooth is being used as the wireless technology.

### 3.2.2 EAP-AKA, a secret key based authentication and key agreement protocol

This paragraph describes how the EAP-AKA protocol is realized in the SHAMAN demonstrator.

The demonstrator will not implement the full EAP-AKA draft standard as defined in [3] but rather a subset of it. In the following, it will be made explicit which parts of the standard are realized and which are out of scope.

The EAP-AKA standard was developed with respect to a typical scenario consisting of entities like a Network Access Server (NAS) (which is called Brain Access Router or BAR in [12]), an AAA-server and so on. This scenario will not be fully reproduced, rather some of these entities will be colocated in the demonstrator, so some of their functionalities will be simulated, and not all protocols and interfaces involved in reality will be implemented.

The following picture shows the entities that exist in the demonstrator. 'USIM' corresponds to a real 3G smart card which implements the AKA mechanism defined for UMTS [4]. 'Client' is a software emulation of a mobile subscriber in a 3G-telecom network. 'Server' is a software emulation of an authenticator and EAP-server at the same time. This corresponds to a colocated BAR, AAA-L and AAA-H.

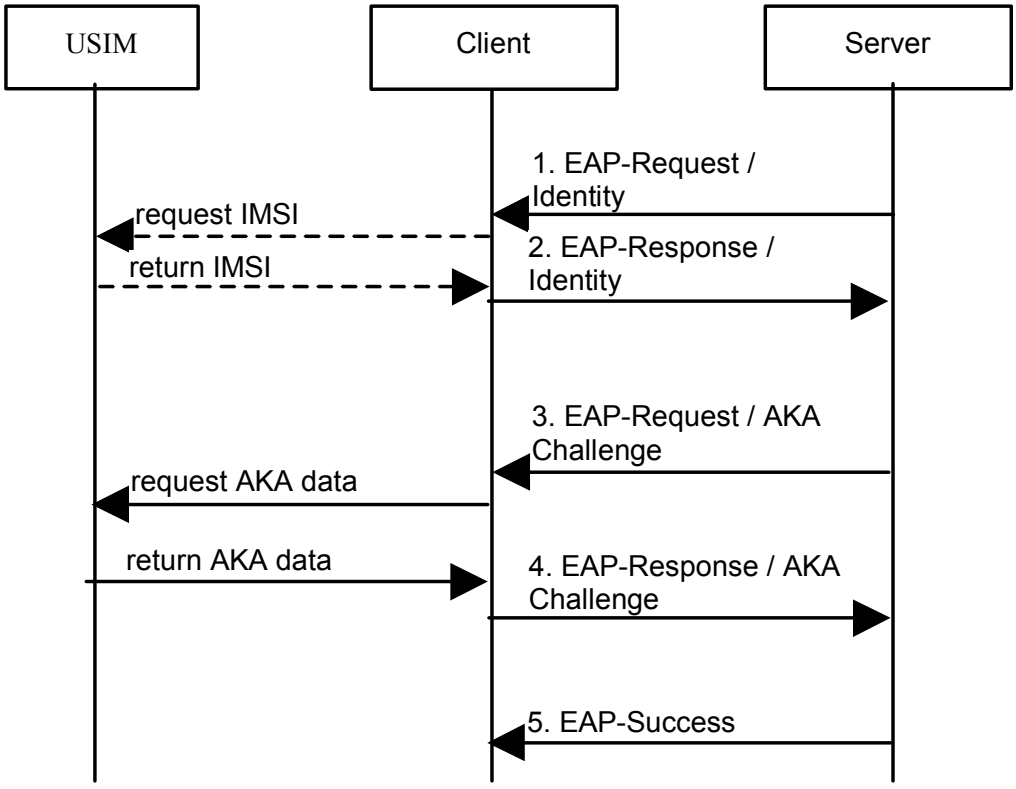


**Figure 11: EAP AKA message flow.**

Figure 11 shows the regular case (successful authentication).

Thus, the task of the server is to create AKA challenge data, to encapsulate this data into EAP data packets and send them to the client. The response from the client has to be parsed and verified for correctness. Therefore, the server must be able to understand the EAP protocol, the EAP-AKA

protocol and must support the UMTS AKA protocol and the specified algorithms (e.g. the MILENAGE algorithms [4]).

There will be no AAA protocol implementation, since the communication between a NAS and an EAP server is out of scope.

The client must also be able to create and interpret EAP messages. The processing of the UMTS-AKA EAP method, however, is done in the USIM card at the client side. Thus, the client extracts the AKA data from the EAP messages and sends them to the card. Afterwards, it encapsulates the AKA response from the card into new EAP messages and sends them to the server.

The identity of the client will be based on the IMSI that is stored in the USIM.

The following cases shall be implemented:

1) Successful mutual authentication
2) Failed server authentication: Server sends wrong data and Client declines authentication, because it calculates a wrong MAC (in the demonstrator, the user can tell the server to create wrong data)
3) Failed client authentication: Client sends wrong data and server declines authentication (this can only be simulated; a card with a wrong AKA secret key, for example, would lead to a MAC failure, as in case 2)

The following features will *not* be implemented:

1) Multiple authentication vector creation by the server (it is assumed that there is only 1 client)
2) Replay attack and re-authentication
3) NAI for client identification (see [3]); only the IMSI is used for client identification
4) Encrypted EAP messages (see paragraph 7.2 in [3])
5) Support for the GSM authentication protocol
6) Identity privacy (see paragraph 4.3 in [3])
7) Nested attributes (attributes encapsulated in other attributes, as described in chapter 6 of [3])
8) Usage of the negotiated keys for IPsec security associations

Realisation:

- The transport protocol is UDP (as in the implementation for JFK and imprinting)
- The implemented authentication algorithms are MILENAGE [4] and the XOR USIM-test algorithm (see [11], paragraph 8.1.2).

The EAP message exchange between the client and the EAP server as well as between the USIM and the client will be shown in a message window and the payload being transported will be shown in the "message detail" window. The negotiated session keys CK and IK will be revealed to the demonstrator user at the end of the protocol run.

# 4 Definition of the Hardware platform

## 4.1 General platform

From the work in WP1, we conclude that a BRAIN-like architecture would most fit our needs. For more information on the BRAIN architecture, see [12]. The main motivations for BRAIN to design a mobile access network based on IP technology falls into three parts: economic, engineering and improvement for the end user. The economic and engineering part all come down to the advantages of using "all IP" based networks and using IP is the correct future proof choice. The security aspects within BRAIN were covered only in a very general level. So, by defining a demonstrator in close relation with the BRAIN architecture, this project will present more added value with respect to security issues in a BRAIN-like network.

A BRAIN-like network mainly consists of a fixed IP network, which actually represents the IP network of an operator, and a wireless part. Both network parts are connected via a gateway/router-functionality.

The fixed network part typically consists of a network reflecting an operator infrastructure with the HSS database containing all relevant information of this operator's subscribers, an AAA and/or CA-server, an application server and routers/gateways to connect the wireless network(s) with the fixed network.

For the wireless part, several technologies like GSM, GPRS/UTRAN, WLAN or Bluetooth are possible.

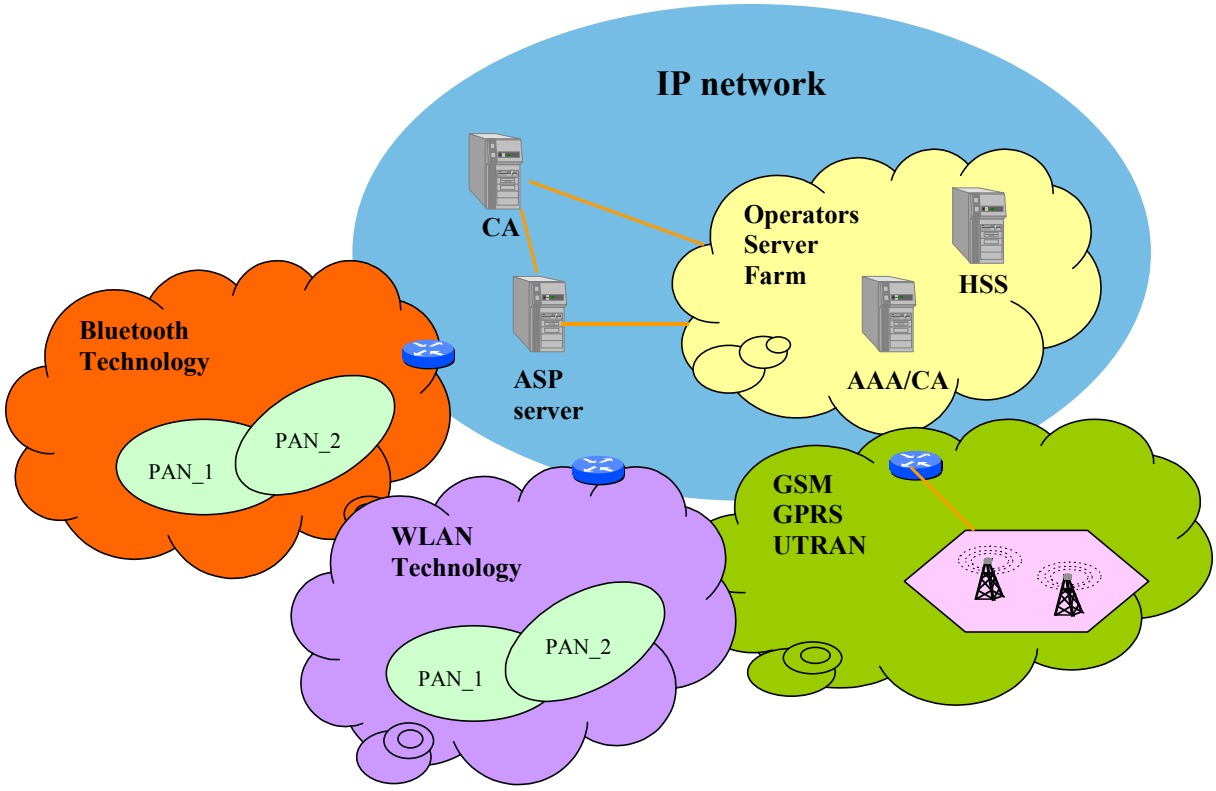


**Figure 12: BRAIN-like architecture.**

## 4.2 Targeted hardware set-up

As mentioned in the previous sections, the demonstrator architecture should concentrate on those aspects of SHAMAN pertaining to the security aspects of mobile security.

No attempt will be made to build a real BRAIN network; instead, the BRAIN access and home network will be software emulations on one or more PCs. For the mobile nodes, "real" technology will be used as much as possible, especially with regard to the link layer. However, higher layer functionality will have to be implemented and integrated on PCs, enabling the developer to implement the protocols in a way that allows the user to gain insight into their functionality. For the security-critical parts of the protocols, a real smart card will be used.

The following picture (Figure 13) shows that the BRAIN access and home network will be emulated on one PC. The mobile nodes (the picture shows 3, MN1 to MN3) form a personal area network (PAN) and are interconnected via Bluetooth connections. One of the mobile nodes in the PAN (MN1) communicates with the access network via a WLAN access point (AP). Some of the mobile nodes (possibly all) possess a security module (denoted as "Card" in the picture). In phase 1 of the demonstrator, the connections indicated as wireless links may also be wired links. Furthermore, the entities in the picture (the mobile nodes as well as the BRAIN network) do not necessarily have to run all on different machines.

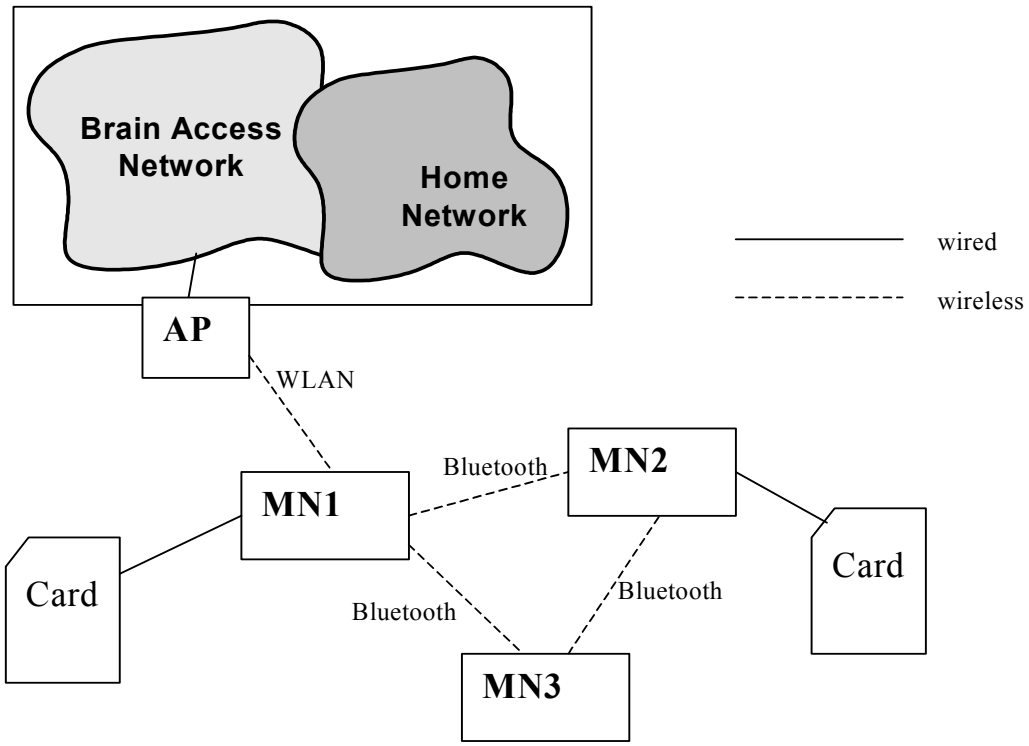


**Figure 13: Targeted system architecture.**

The following pictures show what the hardware configuration will look like for the respective phases of the demonstrator (see chapter 3).


**Figure 14: Phase 1 hardware set-up.**

In phase 1 (where the JFK and Imprinting/MANA protocols shall be demonstrated), there must be a laptop connected to a cardreader; a second laptop is convenient, but not really necessary, and would be connected to the first laptop via a wired link (Figure 14). The protocols to be demonstrated can also run on a single PC, supported by the smart card in the reader. This means that all entities, including server network and all mobile nodes, run on one machine

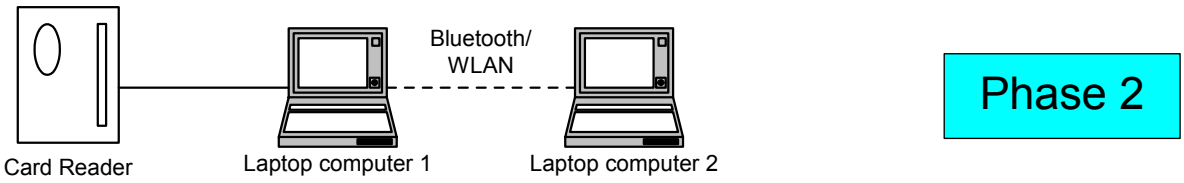


**Figure 15: Phase 2 hardware set-up.**

In phase 2 (Figure 15), wireless connections between different entities will be used, so there must be at least one more laptop. In the picture, two machines are connected via Bluetooth, so they can be imagined to constitute a PAN. For the emulation of a BRAIN access network, a WLAN connection between the peers would seem more natural, however it does not influence the functionality shown, so for a demonstration WLAN or Bluetooth could be used interchangeably.

## 4.3 Hardware and software to be used

1) The PCs: HP Omnibook laptops will be used.
2) The card readers are PCT-200 by Giesecke & Devrient, although a variety of others can also be used.
3) The Bluetooth adapters are supplied by Digianswer.
4) WLAN adapters: Cisco Aeronet 350.
5) The default operating system on the laptops will be Windows 2000.

# 5 Demonstrator Software Architecture

## 5.1 Design Goals

The architecture must support all selected use cases required by the other SHAMAN workpackages. Besides, the following design goals are to be met:

- A smart card as well as a smart card simulation should be usable; it should be possible to switch to new cards and new card middleware without significantly changing the rest of the system.
- Several protocols could be utilised, added and replaced without significantly changing the rest of the system (plug-in-principle).
- It should be possible to use existing protocol implementations, eventually written in different programming languages.
- The architecture could be extended and reused in future projects.
- Portability and interoperability (e.g., usage of a heterogeneous environment like LINUX and Windows machines at the same time).

## 5.2 Basic Architecture

The generality described in 5.1 can be achieved by choosing an architecture consisting of three building blocks where the different blocks communicate with each other via sockets. The information to be exchanged consists only of protocol messages with a relatively simple structure, so a more high-level communication (distributed objects, like CORBA, COM, RMI) is not of much use. A CORBA implementation, for example, would significantly add to the complexity because a CORBA implementation would have to be selected in this case, properly integrated and used; this would certainly not pay off in our situation. A socket interface is the best choice in terms of openness, portability and extensibility. Of course, a socket interface also has its drawbacks: It is low-level and therefore not very easy to program, and issues like thread handling may be tricky. However, these problems should be controllable and it is possible to choose a software design that alleviates the handling of the low-level socket interface.

Figure 16 shows the basic building blocks of the architecture.

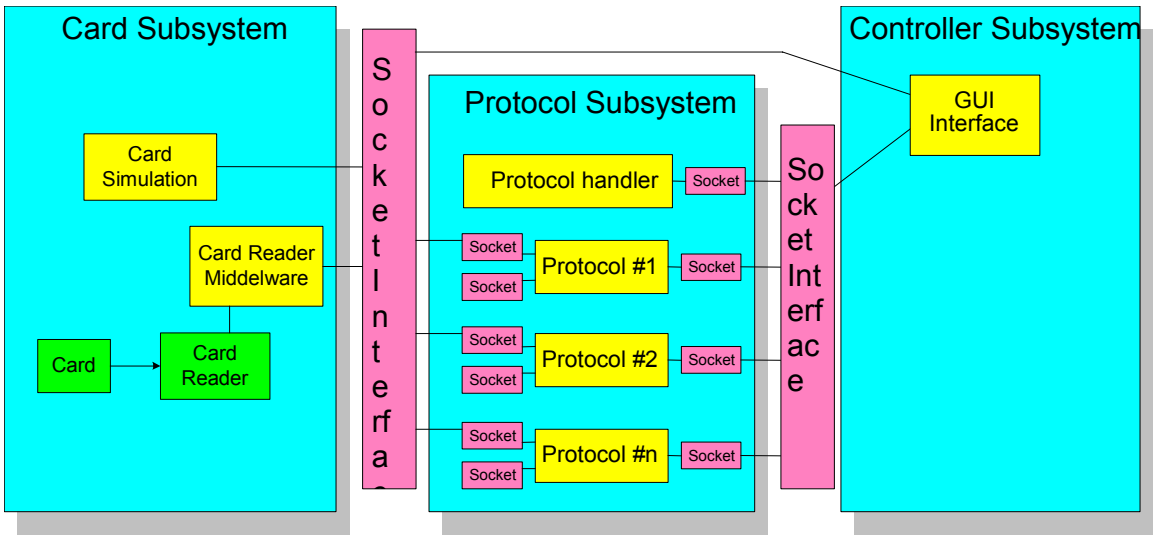


**Figure 16: Demonstrator Software Architecture.**

The architecture consists of three building blocks called controller-, protocol- and card subsystem.

The purpose of the controller subsystem is to start the whole system (i.e. protocol handler and card subsystem), offer the user some way of interaction (e.g. choose initiator or responder role, select/start protocol, trigger protocol to request parameters from user) and display all interesting events on a GUI. The protocol subsystem contains a generic protocol handler, which is responsible for starting and stopping protocols, and all protocol implementations. Each protocol implementation comprises both the actual communication part and the GUI (i.e. dialog) in which the user can enter the protocol specific parameters. Those can be open source implementations, which would in this case just have to be adapted to the architecture, namely supplied with the proper interfaces and breakout calls. The card subsystem consists either of the card itself with the corresponding card reader (hardware), the driver and the so-called "middleware" (which provides an API for communication between application programs and a smart card), or of a card simulation that can be used in case there is no smart card or reader attached.

The controller subsystem and the protocol handler do not depend on the protocols. New protocols can be added by just adding their name and implementing class to a configuration list. Modifying protocols simply involves replacing their implementing class. If the GUI and the protocol subsystem run in distinct processes, the protocols can be implemented in any programming language, since they only exchange information with the controller subsystem via a socket interface. In the current demonstrator version, the controller and protocol subsystems run in different threads of the same process; this could however be generalized to using distinct processes without too much effort. Another socket interface exists for the communication between the controller subsystem (for starting purposes only) and the protocol and the smart card subsystem. Therefore, the card reader middleware and the card simulation can also be implemented in any language. In the current demonstrator, the card subsystem runs in its own, separate process.

The demonstrator architecture will only works if an interface between the card, the protocol and the controller subsystems can be defined which is sufficiently generic and flexible. The interface to the card subsystem basically consists of the smart card command set, the corresponding return values and a message format. This is further described in the paragraph about the socket interfaces.

The interface between the protocols and the controller subsystem must specify what the streams to be written on the sockets shall transport. In one direction, this is merely the command to start or stop a certain protocol or to trigger it to request its parameters from the user. In the other direction, it is a protocol message with all the information to be displayed in the GUI. This will be further described below.

## 5.3 The dynamic behaviour

In general, the communication takes place between two protocol entities residing on different PCs. For the JFK protocol, for example, the two sides are denoted as initiator and responder.

Figure 17 shows what the flow of events would look when a protocol is started:

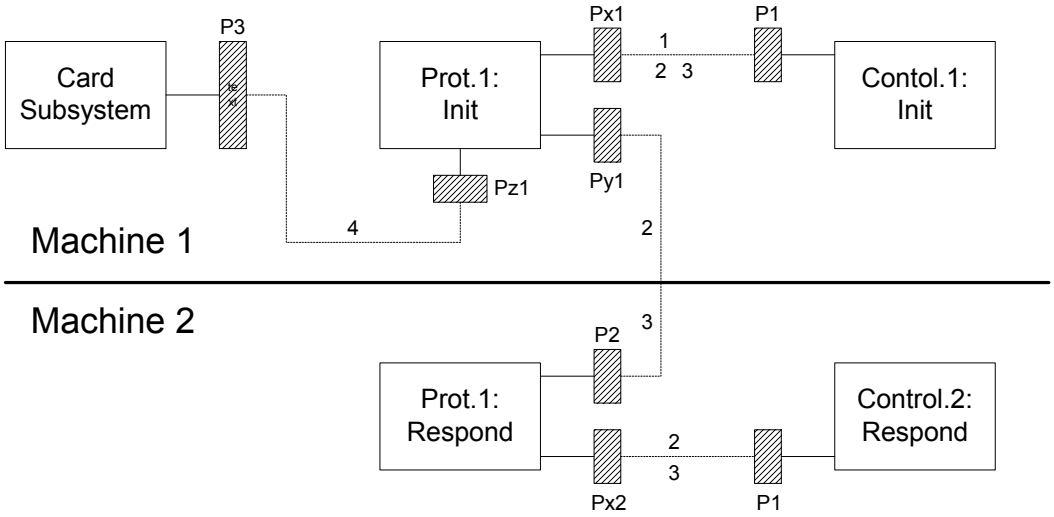


**Figure 17: Internal structure of two demonstrator machines communicating.**

The dashed boxes denote the sockets; the dashed lines denote the messages sent via them. Pxxx is the number of the corresponding port. The ports are for the communication between:

a) The protocol and the controller subsystem (P1)

b) The protocol entities initiator and responder (P2).

c) The card and the protocol subsystem (P3)

The numbers beneath the dashed lines indicate the order in which messages are sent.

To begin the protocols, both of the machines participating in the demonstrator process have to be started manually by the user. In the above diagram, it is assumed that one process is dedicated to the initiator (Control 1) and the other to the responder (Control 2). This starts a server socket on a predefined port P1 on the controller subsystem. The protocol subsystem will open a client connection towards this socket. The details of the actions taken at start-up are described in section 5.3.4.

A protocol (JFK, for example) is started by a user interaction with the GUI on the initiator's side (like the user pressing a button "start JFK"). This causes Control 1 to send message 1 to the protocol subsystem (Prot. 1) telling it which protocol to start. This communication happens from P1 to Px1. Once this is carried out, Prot.1 generates the first protocol message. This is sent (message 2) to Control 1 (so that the GUI is able to display the message flow and the message content); this communication goes from Px1 to P1. Of course, the protocol message must also be sent to the responder on a predefined and protocol specific port P2 (communication from Py1 to P2). This leads to an update on the responder GUI (message 2 on port P1 between Prot.2 and Control 2).

The responder then generates a response (message 3), which must, like message 2, be sent to Control 2 (for GUI update), to Prot.1 and from Prot.1 to Control 1.

Eventually, Prot.1 needs support from the smart card, in this case it sends a message 4 to the smart card subsystem on port P3 (communication from Pz1 to P3) and receives an answer on Pz1. This must be shown in the GUI of Control 1 as well, so that message and answer must also be transferred to port P1.

### 5.3.1 The controller subsystem

The controller subsystem controls the proper set-up and initialisation of all other modules and displays all interesting events in a GUI. It exchanges information between itself and the protocol and card subsystems via a socket interface.

#### *5.3.1.1 The Graphical User Interface*

5.3.1.1.1 Demonstrator instantiation and configuration

Launching the application will start one instance of the demonstrator. This will open a window, which presents the user with the choice to start this instance as an Initiator or Responder (the role of this instance is indicated in the GUI). So, in order to run a complete demo, two processes must be started, one as initiator and the other as responder. The main panel will provide the user with an interface to configure the protocols and start the communication between initiator and responder.

For more detail on how to use the demonstrator, we refer to Annex II.

5.3.1.1.2 System Status

The user must have an overview of the system *status*. This pertains to the protocols installed, the state of the card (plugged / not plugged / simulation used) and so on. Likewise, configuration information must be made visible for the user on request, and he must also be able to manipulate it (e.g., choose the IP-address of the peer process). The GUI should therefore allow the following information to be displayed:

- Protocol installed – This gives the exact details of the protocol being used in the protocol subsystem, and allows the user to change it if there is more than one protocol implementation available for the demonstration.
- Card State – This will give the status of the card entities being used in the protocol demonstration. If there is a hardware implementation of the smart card then this will be indicated along with the status of the card's connectivity (online or offline). If the card is being simulated on the computer, this should also be indicated.
- Configuration information – For the protocol selected, this will allow the user to enter in all the device parameters in order for the protocol to run. For the MANA I protocol, this may include a specification of data items D, and possibly the components' MAC addresses (although this could probably be generated automatically). For JFKi, this will include the nonce length, Diffe-Hellman group selection and hash algorithm selection. Also entity name, peer IP address (may be localhost if peer is on same PC), role in protocol (e.g. initiator or responder). Essentially the information displayed consists of all parameters that the user can configure when the protocol entity is created.

5.3.1.1.3 Message flow/detail

Each protocol entity will have its own GUI and there will be two main items in the principal GUI display:

1. Message Flow:

   This will display the flow of messages relative to the entity concerned. Messages will be labelled with meaningful names and may include symbolic representations of the parameters that are being sent. The message flow will include:

   - Messages to/from a smart card if the protocol entity concerned makes use of a smart card
   - Messages to/from the other protocol entity (which may be on the same PC or a different one)

   If the other protocol entity is running on the same PC then the message flow diagram on one of the entities may be minimised/hidden if all the relevant information is already displayed on the other protocol entity.

2. Message Detail:

   Only certain message parameters passed to the GUI from the protocol subsystem will be displayed in the Protocol Message Flow. All other protocol and timing information considered useful will be displayed in the Message Detail part. The Message Detail part will provide a representation of the actual parameter values exchanged between the protocol entities and the smart card (in

hexadecimal representation). The Message Detail part may be minimised/hidden if the other protocol entity is running on the same PC. However, the message details relevant to each protocol entity on the same PC may be different (e.g. timing information) so the demonstrator should allow both Message Detail Windows to be displayed simultaneously. Message detail is also stored in a log file.

#### 5.3.1.1.4 Protocol interaction

The user must be provided with the necessary interaction items (pushbuttons, menus). He must be able to select, start, interrupt and repeat protocols. In some protocols, user interaction is required, so additional buttons and entry fields are needed.

The user can start, stop or interrupt the protocol once all the components have been created and the necessary parameters have been specified. In general both entities in each protocol should have the same level of control.

The protocol subsystem could simply execute a given protocol at the speed the computer is running at. However, this may not be very useful for the user. Instead it is required to allow the protocol messages to be displayed at a slower rate. Therefore we have chosen the option to allow the controller subsystem to tell the protocol subsystem to wait for a specified amount of time between execution steps of a protocol.

#### 5.3.1.1.5 Example screenshot

Figure 18 shows a screenshot of the SHAMAN demonstrator GUI. Here, we will briefly explain the functionality of the different windows. For a more detailed explanation on how to use the demonstrator, we refer to Annex II, where we illustrate all steps necessary to run the JFK protocol.

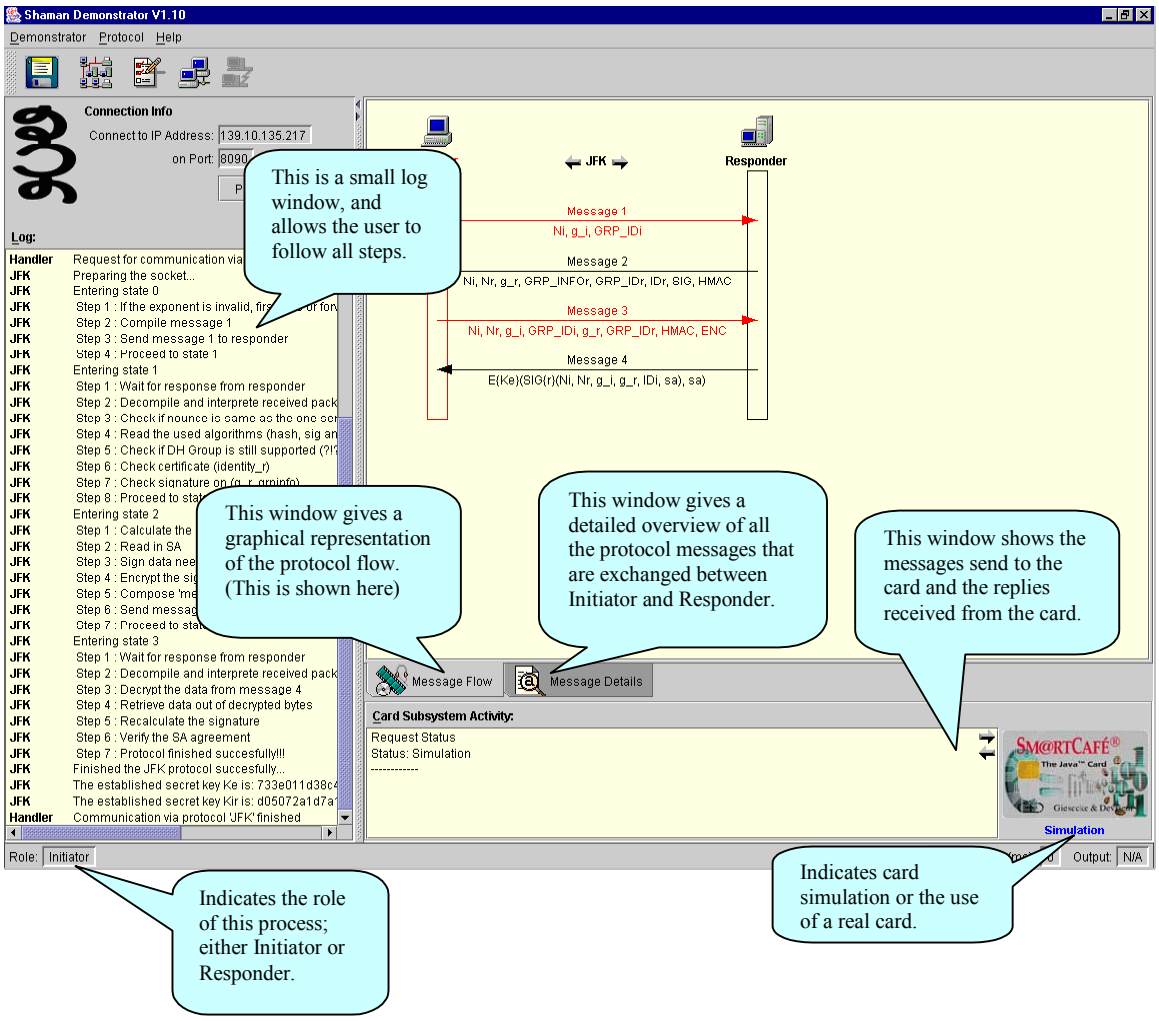


**Figure 18: Demonstrator GUI Interface.**

### 5.3.2 The protocol subsystem

Conceptually, the protocol subsystem is the easiest. However, it must be able to (1) run the protocols independently (i.e. asynchronously) from the controller subsystem and (2) easily handle new protocols. To achieve this, the subsystem has a generic protocol handler (refer to the class design) that runs in its own separate thread (communicating with the controller subsystem through a socket interface) at its core. On request by the controller, the protocol handler can load and start any protocol dynamically. The only prerequisite is that it receives the name of the target protocol class from the controller and that this class implements a generic protocol interface with which the handler can interact.

In the current architecture, the protocol handler can switch between protocols at any time. However, only one protocol can be loaded/active at any given time.

Once the protocol handler has started a protocol, it passes the socket through which it communicates with the controller to the protocol class so that the latter can directly communicate with the controller (e.g. to convey logging and communication information).

Each protocol class is completely autonomous and is, among other things, responsible for getting its own user input (i.e. parameters via an input dialog). However, meaningful default values are provided for all parameters so that a non-expert operator can also use each protocol. This also means that there is no need for external configuration files.

### 5.3.3 The card subsystem

This section explains the card subsystem in more detail. It provides some background information which is not absolutely necessary for understanding the architecture.

The smart card subsystem consists of hardware (the smart card itself and a card reader) and software, namely the so-called middleware that enables application programs running on a PC to communicate with the card. The smart card is autonomous in the sense that it has its own operating and file system. Inserting the smart card into a card reader, it can communicate with the outside world via APDUs (Application Protocol Data Units) (Figure 19).

In order to communicate with the smart card, all elements of the chain (reader, reader driver and smart card) must be operative. If this is not the case, the demonstrator can make use of a smart card simulation instead. This is denoted as “simulation mode”, as opposed to “real mode”; the real mode can be inactivated in several ways, e.g. by not plugging the card, removing the card reader or uninstalling the driver. In simulation mode, the card subsystem will basically delegate procedure calls to a crypto library and transform the result accordingly. Section 5.4 describes how the simulation and the real smart card system can be smoothly integrated into the architecture.

The protocol stack consists of only 3 layers: a) the physical layer b) the transmission layer, where card-specific protocols like T=0, T=1 etc. are used (specified by ISO/IEC) c) the application layer (with protocols like GSM 11.11).

Every smart card provides a *command set*, i.e. the set of commands it supports, with all parameters. Most of these commands are standardised, as in the ISO/IEC 7816/4 standard, which specifies (among others) all commands for the access to the card‘s file system. In SHAMAN, the card must provide commands for the creation of signatures, storage of keys, support of protocols and so on, some of which are available in common standards, some not. In any case, the command set for the SHAMAN security module must be properly defined.

In the proposed architecture, sending a command to the card means to pack it into a message and write it on the respective socket, namely the one attached to the card middleware. The exact format of the message must be understood by the middleware, so that this format is another thing that must be properly defined. One task of the middleware should be to hide the details of card-specific protocols from the implementers of the protocol subsystem.

In the demonstrator, we use the Starcos library™ by Giesecke & Devrient as the middleware. Only its most basic functions, namely for sending APDUs to the card, need to be used. It provides a Java Native Interface (JNI), so that the biggest part of the card subsystem can readily be implemented in Java.

In Figure 19 the smart card subsystem as proposed in the architecture is visualised for clarity.

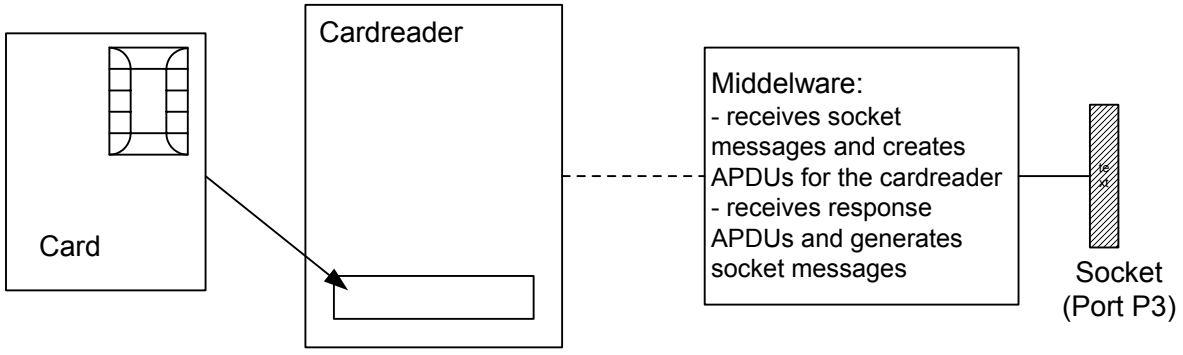


**Figure 19: The smart card subsystem.**

The following picture (Figure 20) shows the protocol stack.

On the application layer, a command from the card's command set is chosen and supplied with the required parameters. In our example, this is a command for the calculation of a hash function.

On the transmission layer, it is coded as an APDU; the CLA field in the T=0 protocol is the so-called "class byte" which indicates the utilised standard; INS stands for "instruction byte" which indicates that the instruction is "PERFORM SECURITY OPERATION". In the P1-field it would be indicated that the security operation should be a hash function, and so on.

The way to code all this should be hidden from the other subsystems in the demonstrator. However, the interface between the middleware and the protocol subsystem remains to be specified.

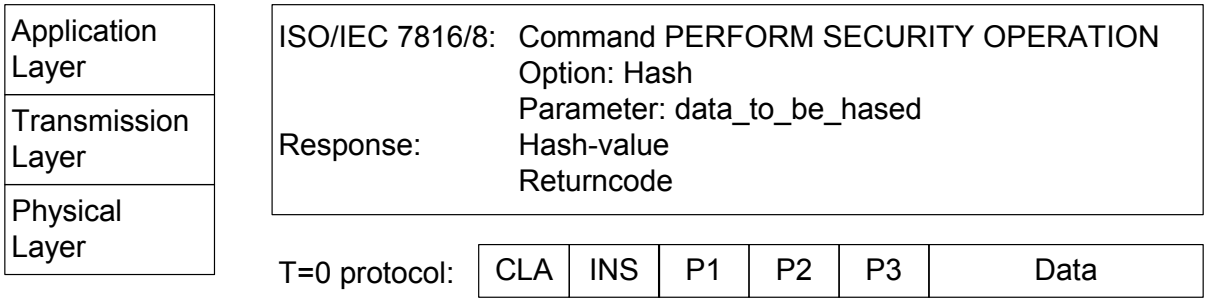


**Figure 20: Smart card protocol stack.**

### 5.3.4 Actions during startup

As already pointed out, generally two or more protocol entities reside on different machines, e.g. a client and a server process. It would be more user-friendly if starting the client controller would also bootstrap all other system parts, like the protocols and card subsystem on the client side and the analogue processes on the server side. However, starting processes on remote machines is perhaps not straightforward, so it is assumed that the client's and the server's controller have to be started manually on their machines.

All processes, except the card subsystem, can be started either as a client or as a server process, e.g. by specifying a command parameter.

The card subsystem process can be started either as a 'real' card or as a simulated system.

In the following, the procedure during start-up is described.

1. On machine 1, the controller process is started manually (as a client). Likewise, on machine 2, the controller process is started as a server. The IP address of the peer can be given as another command line parameter or will be controlled via the GUI.
2. The controller processes launch their respective protocol handlers in separate threads. Additionally, they start their card subsystems as completely separate processes.
3. The protocol handlers establish a socket connection towards the controllers on port P1 (see Figure 17). The controller subsystems will hold the client sockets (on port P1) and the protocol subsystems the server sockets (ports Px1 and Px2).
4. The controller subsystems initialise the card subsystem and report the outcome. The controller GUI indicates to the user that he is either using the simulation or the real card.
5. The controller subsystem is now ready for user input: i.e. normal sequence: start a protocol, ener the protocol parameters, start the communication. The user can also specify a file in which all logging information for the session is written.
6. When the users selects a protocol (e.g. JFK), the controller subsystem sends the appropriate message to the protocol handler, which loads the corresponding protocol class.
7. When the user starts the actual communication, the protocol client tries to find its peer on port P2 (the peer's address can be configured via the protocol specific GUI. If it doesn't find the peer, it

must be possible to change the address and restart this step. Afterwards, client and server establish a socket connection. Note that *socket* server and *socket* client do not have to coincide with the client and server roles in the protocol, although this is reasonable if there is more than 1 client, and because it's less confusing. If the socket communication cannot be established due to some problem, this is signalled to the controller, and the user can choose to terminate the process or to restart it (after having solved the problem),

8. The protocol client connects to the card subsystem.
9. Now the protocol client starts the protocol run.

Through its socket interface with the protocol subsystem, the controller receives progress and status information about the active protocol and about the card subsystem. This information is visualized in the GUI and written to a log file if so requested by the user.

## 5.4 Description of the subsystem interfaces

### 5.4.1 Interface between controller and protocol subsystem

As already mentioned, the controller subsystem also uses a socket interface to communicate with the protocol subsystem. To do this, it starts the generic protocol handler (on a separate thread, refer to Class Design), which creates a server socket on port P1. The controller subsystem connects to and uses this socket for all further communication.

The actual messages that are sent to the protocol subsystem are Java classes that are automatically serialized/deserialized for transmission/reception over the sockets.

Essentially, there are 3 classes of messages to be exchanged: control messages, logging messages and protocol messages. Control messages are for the start and termination of protocols and for the transport of configuration information. Logging messages convey information about events and errors to the controller, either about generic communication (e.g., socket) problems, protocol-specific errors or problems resulting from a faulty controller configuration. The protocol messages are for the transport of information that is specific for the protocol being demonstrated.

### 5.4.2 Interface between protocol and card subsystem

The socket interface is specified as a set of messages with a direction, each containing a set of parameters. Every message is coded on byte level. The parameters would usually be coded as a TLV (tag – length – value) structure, which means that typically 1 byte indicates what type of parameter it is, the length indicates the number n of bytes required to code the parameter's value, and the following n bytes are the parameter's value itself. However, since until now all payloads have a fixed length, the parameters can just be concatenated in a defined manner, and no length indication is necessary.

Once the card subsystem is started it generates a server socket that all protocols can connect to. The card reader inside the card subsystem expects APDUs. APDUs are quite similar in structure to the messages defined in a socket interface. So why define a socket interface at all and not use APDUs directly? The answer is that, in the SHAMAN context, more complex actions may have to be defined. One of those complex actions may lead to several card APDUs, or it might depend on the state of a state machine; there may even (theoretically) be actions that refer to the reader only and not to the card (like "eject card"). Therefore, the socket interface describes 2 message types: one for "simple" commands that have a one-to-one correspondence to card APDUs, and another one for the others. The "simple" commands are basically just passed through to the card and the others require more complex interaction and processing.

The following socket interface applies:

There are 2 types of messages, namely "card commands" and "control commands". The former will lead to processing by the card (or simulation) itself, whereas the latter are commands to the card subsystem in general (e.g., the command to shutdown the card subsystem or to return the status).

Among the card commands, we have “simple” and “complex” commands, in the sense already described above: Simple commands can directly be mapped to card APDUs, whereas the complex commands usually lead to more than one APDU sent to the card.

The message format is as follows:

| | |
|---|---|
| 1 \| 2 \| 3..... | type: 01 for card commands |
| length \| type \| Rest.... | 02 for control commands |

“Rest” will basically be an APDU for simple card commands, for control and complex commands it will have a proprietary structure described in the annex.

The APDU format follows the ISO/IEC 7816-4 standard.

One example shall be presented here, namely the GET CHALLENGE command that generates a random number on the smart card:

| COMMAND | CLASS | INS | P1 | P2 | $L_e$ |
|---|---|---|---|---|---|
| GET CHALLENGE | 'A0' | '84' | '00' | '00' | Max. |

Here the only variable parameter is Le, containing the maximum length of the expected response. The “class byte” denotes the type of card, e.g. ‘A0’ for all commands for a card corresponding to the GSM 11.11 standard.

The APDUs to be supported by the SHAMAN security module and the structure of the generic commands are described in Annex I.

## 5.5 Class design

This paragraph describes ideas for the demonstrator’s class design. This will mainly be a description of the business classes, i.e. no attempt will be made to break down the design to all auxiliary classes and structures.

The purpose is to realise the design goals stated at the beginning of this chapter. The class design is basically architecture-driven, i.e. it is not governed by a complex business logic; rather the classes are helpers to make the code integration into the socket-based architecture as robust, extensible and user-friendly as possible.

### 5.5.1 Class design of the controller subsystem

The controller subsystem must provide a graphical user interface, modules to start the protocol and card subsystems and a log-to-file mechanism.

The controller subsystem only needs to be programmed in one language, we choose Java.

Figure 21 shows what the class design for the controller might look like.

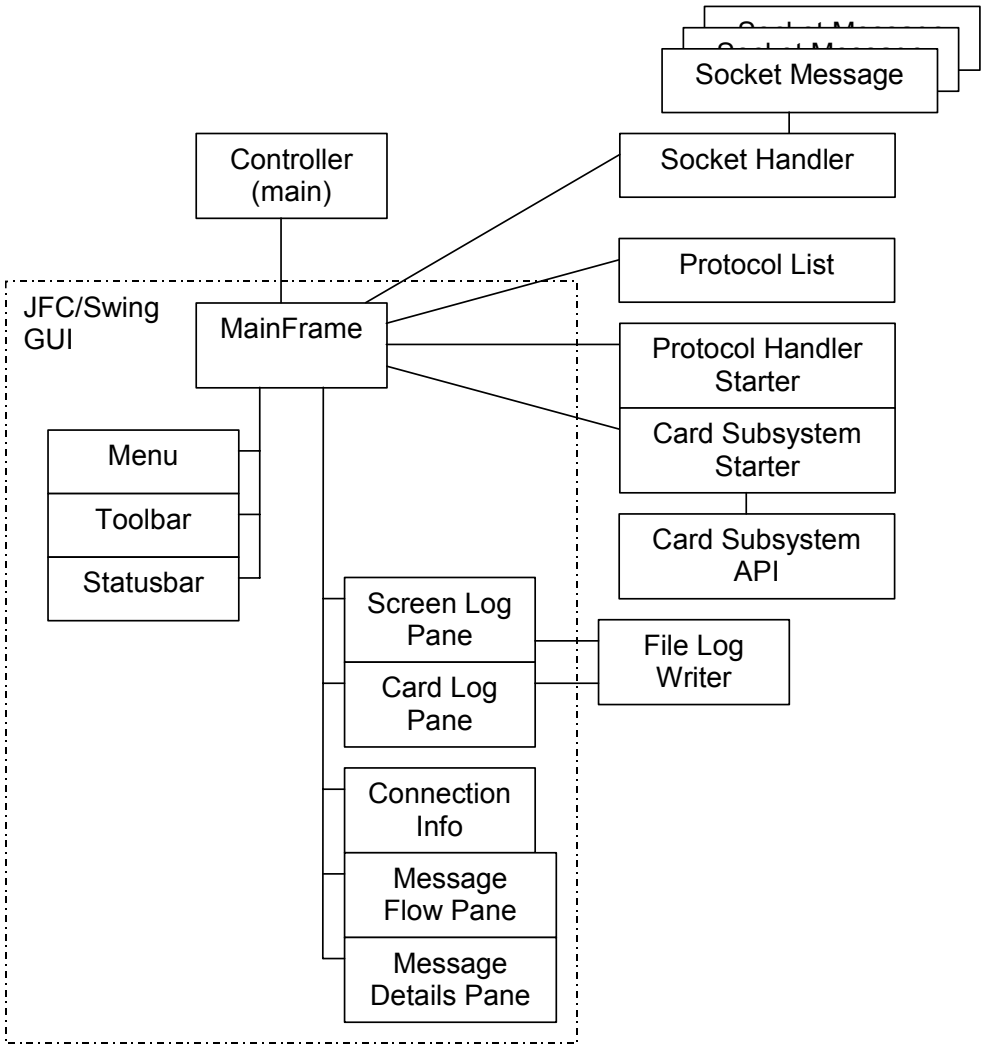


**Figure 21: Class design of the controller subsystem.**

The boxes surrounded by the dotted line symbolise the classes belonging directly to the GUI. This is not to say that there shall be no more classes in the GUI; they are by no means complete.

However, the boxes outside the dotted line show the business classes of the controller subsystem; there should not be substantially more than that.

The lines between the classes show the communication relations.

The main procedure for the entire controller subsystem is defined in the Controller class. The only role of this procedure is to create the main frame window class. It is this class that controls the rest of the controller and, for that matter, the protocol and card subsystems and that knows all (at least, most of) the other objects. As already mentioned at the beginning of this paragraph, there are classes for the following tasks:

1) Start the protocol handler
2) Start the card subsystem
3) Control writing to and reading from the socket (SocketHandler).
4) Write to an optional log file.

Besides (not shown in the diagram) there may be classes to improve the structure of socket handling, for example concerning error treatment and recovery.

### 5.5.2 Class design of the protocol subsystem

As already stated, the protocol subsystem is conceptually easy. However, the implementation might be complicated by the fact that it has 3 socket interfaces as opposed to only 1 for the other subsystems: 1 towards the controller, 1 towards the card subsystem and 1 towards its peer on the other machine (server or client). However, only one interface, the UDP sockets to and from the peer, is directly controlled by the protocol implementations themselves. The interface with the controller is handled by the central socket handler already discussed for the controller subsystem. The communication with the card subsystem is taken care off by the card API that is discussed in the next section.

The central Protocol Handler class is responsible for starting protocols when so requested by the controller subsystem. In true OOP fashion, the actual protocol implementations are all derived from a single base protocol , which defines the interface between the protocols and the handler. The base protocol also takes care of all the communication with the controller subsystem (e.g. for sending logging and message information).

The protocol handler has no idea about which, or how many, protocols are available. To start a particular protocol, it simply loads and calls the class specified by the controller. In other words, to add a new protocol only involves deriving a new class from the Base Protocol class and implementing its interface. If the protocol implementation is available in a different programming language, then a wrapper class that provides the interface expected by the Protocol Handler must be created.

The protocol implementations are also autonomous as far as their user specifiable parameters are concerned. Each protocol has its own input dialogs and takes care of its own parameter storage.

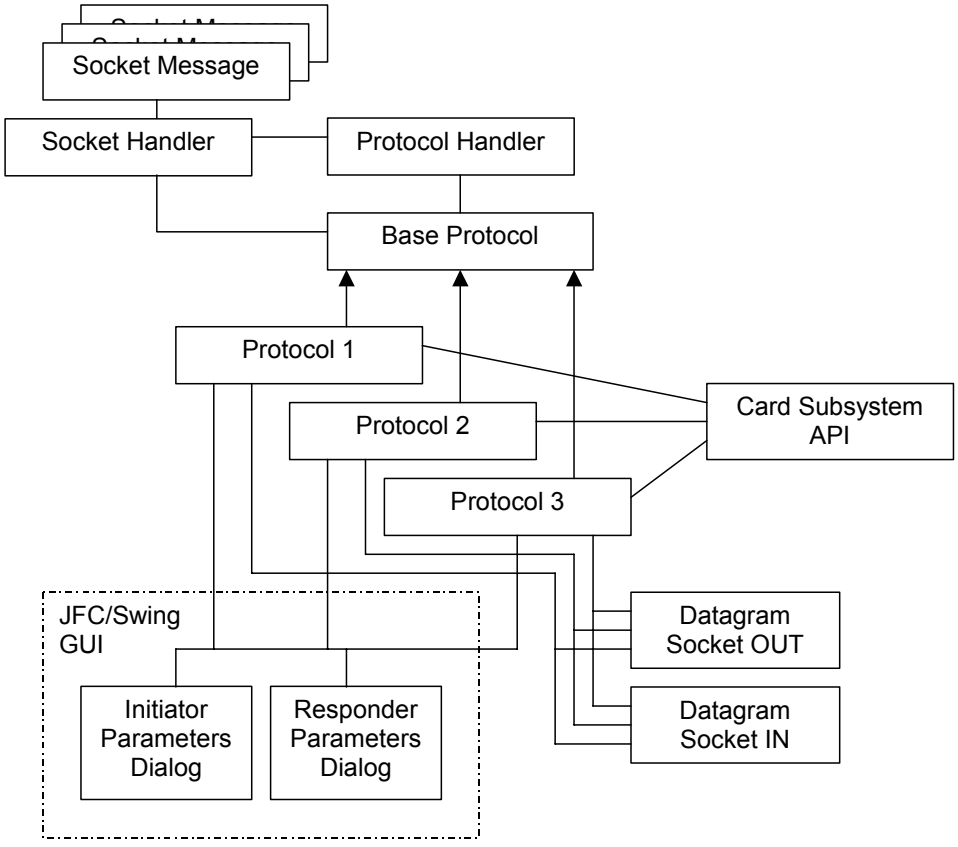


**Figure 22: Class design of the protocol subsystem.**

### 5.5.3 Class design of the card subsystem

The card subsystem receives socket messages from the protocol subsystem. Some of them may be passed on to the card reader middleware with very little further processing (the APDUs, see paragraph

5.4.2), others need some advanced parsing and are possibly transformed into special commands or command sequences to the card.

Anyhow, the socket messages must be parsed and converted; therefore the card subsystem needs a converter class.

The response from the card must also be handled. If a command to the card was received successfully, the card usually responds with a positive acknowledgement (ack). Positive acks do not have to be passed on to the protocol subsystem, but negative acks must be, in order to signal to the user that something went wrong. The actual answer that one is really interested in (the result of a cryptographic operation, for example) must be fetched with an additional card command. This can be done invisibly for the other subsystems. An example: suppose the protocol requires an authentication of the initiator. This could possibly be achieved by the smart card command INTERNAL_AUHENTICATE. It is a standardised smart card command that corresponds to a "simple" command, in the diction of paragraph 5.4.2. So the protocol subsystem's converter creates a socket message that wraps a card APDU and sends it. The card subsystem's converter reads it from the socket and can tell by the first byte that it wraps an APDU, so it can just strip off the first byte and pass on the rest to the card. The card answers with 2 bytes (the so-called "status bytes"). The processing of these status bytes should now be done inside the card subsystem, and not left to any of the other subsystems. If there was an error, the converter should generate a readable error message and signal it to the protocol subsystem, which will also give it to the controller. If there was no error, the converter should "swallow" the status bytes and directly send a GET_RESPONSE command to the card, which answers with the desired bytes (containing the result of a cryptographic operation, to be used for authentication). The steps of "swallowing" the status bytes and sending an additional command to the card should happen invisibly for the other subsystems, so the protocol subsystem just receives the desired encrypted bytes.

If such a command is sent to a card simulation module, it is obvious that it must be treated differently. Therefore, different subclasses of the converter will have to be implemented, one as a client class for the real card, one for the simulation.

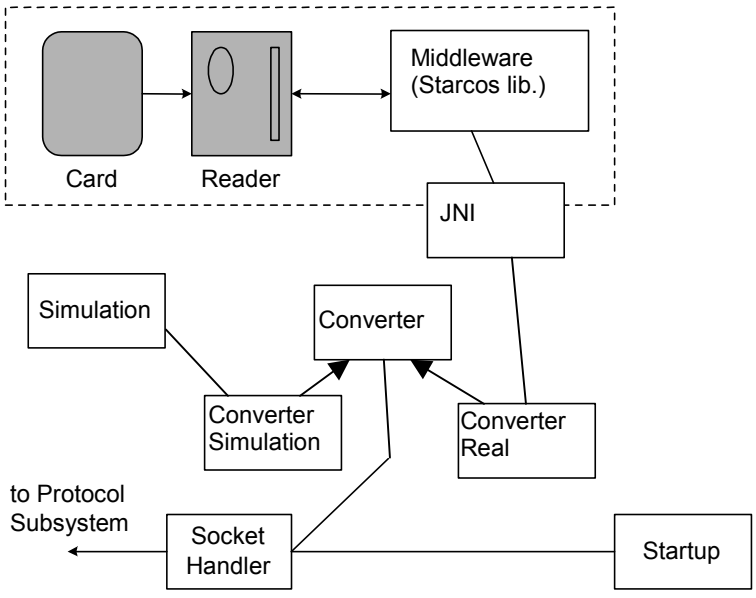


**Figure 23: Class design of the card subsystem.**

# 6 Conclusion

In this deliverable, we have given a specification of the SHAMAN demonstrator, which implements critical components of the security architectures and features identified in WP1 and WP2.

The hardware architecture is described, showing that the demonstrator will consist of several laptops. The connectivity between these laptops will be based on wireless technology. To fully support the security architectures needed for PANs (Personal Area Networks) and PSDs (PAN security domains) developed in WP2, the wireless connection between two laptops, forming a PAN/PSD, will be based on Bluetooth technology. For the scenarios where a mobile node wants to establish a connection with an access network, WLAN technology can also be used.

A detailed study of the WP5 demonstrator software architecture is described. The demonstrator consists of three building blocks, the controller subsystem, the protocol subsystem, and the card subsystem. Communication between these subsystems can be done via a socket interface or a Java API. This allows for an independent design and implementation of each subsystem. Further, the protocol subsystem will allow the easy integration of several protocols; and in principle the implementation of the protocols can be language independent. The card subsystem allows the use of a real smart card to support the cryptographic computations needed in the protocols. Besides the use of a real smart card, this subsystem also included a card simulation module, allowing the demonstrator to work without a real smart card.

# 7 References

[1] IST-SHAMAN Deliverable D13: "Final technical report – results, specifications and conclusions", November 2002, http://www.ist-shaman.org/.

[2] P. Calhoun, J. Arkko, E. Guttmann, G. Zorn, J. Loughney: "Diameter Base Protocol", IETF draft (work in progress), draft-ietf-aaa-diameter-10.txt, April 2002, http://www.ietf.org/.

[3] J. Arkko, H. Haverinen, "EAP AKA Authentication", Internet Draft (work in progress), draft-arkko-pppext-eap-aka-09.txt, February 2003, http://www.ietf.org/.

[4] 3GPP TS 33.102 v4.2.0, 3rd Generation Partnership Project, Technical Specification Group Services and System Aspects, 3G Security, Security Architecture, Release 4, http://www.3gpp.org/.

[5] 3GPP/GSM TS 43.020 v4.0.0, 3rd Generation Partnership Project, Technical Specification Group Services and System Aspects, Security Related Network Functions, Release 4. http://www.3gpp.org/.

[6] ITU-T Recommendation X.509: Data networks and open system communication. Directory information technology-open systems interconnection- the directory authentication framework, November 1993.

[7] W. Aiello, S.M. Bellovin, M. Blaze, R. Canetti, J. Ioannidis, a.D. Keromytis, O. Reingold, "Just Fast Keying (JFK)", draft-ietf-ipsec-jfk-04.txt, April 2002, http://www.ietf.org/.

[8] Several implementations, in JAVA, C and Perl, of the JFK protocol can be downloaded form http://www.cs.columbia.edu/~angelos/JFK/ .

[9] D. Harkins, D. Carrel, "The Internet Key Exchange (IKE)", RFC2409, http://www.ietf.org/.

[10] IST-SHAMAN Deliverable D07, "Intermediate specification of PKI for heterogeneous roaming and distributed terminals", March 2002, http://www.ist-shaman.org/.

[11] 3GPP TS 34.108, Common test environments for User Equipment, conformance testing.

[12] IST-1999-10050 BRAIN Deliverable 2.2, "BRAIN Architecture specification and models, BRAIN functionality and protocol specification", March 2001, http://www.ist-brain.org/.

# 8 Annex I: Socket interface between protocol and card subsystem

The general architecture of the SHAMAN demonstrator, including the socket interface between the 3 subsystems, is described in chapter 5.

The protocol specific information can be found in documents about the functionality split between terminal and smart card (see D13, Annex 4 [1]) as well as in the standards 3GPP TS 33.102.

Tasks that are delegated from a protocol client to a smart card are realised by sending a socket message from the protocol subsystem to the card subsystem. The card subsystem replies with the requested payload or an error message.

This document describes the format of the byte arrays that are sent across this socket interface.

2 types of messages are distinguished: "Card commands" denote those messages that can be directly mapped to smart card APDUs (application protocol data units). "Control commands" are general messages that go to the card subsystem but do not need to involve the card directly (e.g., tell the whole card subsystem to terminate, request the status of the cardreader, etc.).

Generic format:

| Length (2 bytes) | Type (1 byte) | Rest....

Type = 01 for card commands

02 for control commands

P->C means direction from the protocol to the card subsystem, C->P vice versa

### 8.1.1 Card commands and responses (third byte = 01)

Commands: (P->C)

Byte No. 4 is generally the first byte of a card command APDU, it is called "*class byte*".

The following byte is called "*instruction byte*".

HASH_SIGN Used by JFK and imprinting protocol for hashing and signing of input data.

Byte 4: 0xA0

Byte 5: 0xBA

Byte 6: 0x00 (pad with zero bytes) or 0x01 (PKCS1 padding)

Byte 7: 0x00

Byte 8: Number of bytes to be hashed and signed in hex

Bytes 9 and following: input data to be hashed and signed

Example: 0x00, 0x16, 0x01, 0xA0, 0xBA, 0x01, 0x00, 0x10, .... + 16 remaining bytes

Expected response: 128 signature bytes in payload

START_NEW

Starts a new JFK protocol run with Diffie-Hellman calculation for a card in initial state (either no Ks for rekeying available, or Ks shall not be used and will therefore be erased)

Byte 4: 0xA0

Byte 5: 0xBB

Byte 6: 0x00

Byte 7: 0x00

Byte 8: 0x00

Example: 0x00, 0x06, 0x01, 0xA0, 0xBB, 0x00, 0x00, 0x00

Expected response in payload: 128 bytes for gi (Diffie-Hellman-value of initiator), concatenated with 8 bytes for Ni (initiator nonce).

START

Starts a new JFK protocol run with Diffie-Hellman calculation with rekeying.

Byte 4: 0xA0

Byte 5: 0xB9

Byte 6: 0x00

Byte 7: 0x00

Byte 8: 0x00

Example: 0x00, 0x06, 0x01, 0xA0, 0xB9, 0x00, 0x00, 0x00

Expected response: analogous to START_NEW

MESSAGE_2

2nd message between terminal and smart card in a JFK protocol run.

Byte 4: 0xA0

Byte 5: 0xBD

Byte 6: 0x00 (pad with zero bytes) or 0x01 (PKCS1 padding)

Byte 7: 0x00

Byte 8: 0x9C

Bytes 9 – 164: 20 bytes for hash value, 128 bytes for gr, 8 bytes for Nr

Example: 0x00, 0xA2, 0x01, 0xA0, 0xBD, 0x00, 0x00, 0x9C, .... + 156 remaining

Expected response in payload: Concatenated 20 bytes for Ke (encryption key), 20 bytes for Kir (integrity key), 128 bytes for signature or 20 bytes for HMAC (depending on state)

GET_PUBLIC_EXPONENT

Gets the public exponent for the RSA key pair stored on the card

Byte 4: 0xA0

Byte 5: 0xBE

Byte 6: 0x00

Byte 7: 0x00

Byte 8: 0x00

Expected response in payload: Bytes for public exponent of the card's public RSA key

GET_MODULUS

Gets the modulus for the RSA key pair stored on the card

Byte 4: 0xA0

Byte 5: 0xBF

Byte 6: 0x00

Byte 7: 0x00

Byte 8: 0x00

Expected response in payload: Bytes for modulus of the card's RSA key pair

The following card commands are "complex (or *generic*) commands" in the sense that they cannot be directly mapped to a card APDU; they MAY require more than 1 command APDU (e.g., the verification of a PIN or the selection of a EF on the card before the actual command can be executed).

They are recognised by the fourth byte 0xB0 (which is not a genuine class byte, but serves for distinction).

GET_IDENTITY

Returns identity information about the smart card (e.g., the IMSI of a 3G-card)

Byte 4: 0xB0

Byte 5: 0x01

Byte 6: 0x00

Byte 7: 0x00

Byte 8: 0x00

Expected response: The identity information item (exact meaning depends on the implementation chosen)

AUTHENTICATE

Performs an UMTS-AKA-authentication

Byte 4: 0xB0

Byte 5: 0x02

Byte 6: 0x00 for normal authenticate, 0x01 for resynchronisation

Byte 7: 0x00 for XOR test algorithm, 0x01 for MILENAGE algorithm

Byte 8: 0x20 (for 32 bytes of input data)

Bytes 9-24: 16 bytes random value (RAND)

Bytes 25-40: 16 bytes authentication token (AUTN)

Expected response in payload: RES (8 bytes)

Responses: (C->P)

PAYLOAD

Byte 4: 0xA1 (means payload for card response)

Byte 5: same byte as in invoking command

remaining: payload bytes

Example: HASH_SIGN returns 128 bytes containing a signature. The socket message is then: 0x00, 0x83, 0x01, 0xA1, 0xBA, .... + 128 bytes for the signature

ERROR_FROM_CARD

Byte 4: 0xED

Bytes 5 – 6: The error code from the card.

Currently defined proprietary error codes: 6F01 = RSA error, 6F02 = invalid state, 6F03 = Diffie-Hellman error.

Additionally, the error codes defined in ISO/IEC 7816-4 and apply.

### 8.1.2 Control commands and responses (third byte = 02)

Commands: (P->C)

GET_STATUS

0x00, 0x02, 0x02, 0x01

TERMINATE

0x00, 0x02, 0x02, 0x02

Responses: (C->P)

STATUS

Byte 4: 0x13.

Byte 5: card ok = 0x01, card inactive = 0x02, simulation = 0x03

Example: 0x00, 0x03, 0x02, 0x13, 0x03 means simulation

TERMINATED

0x00, 0x02, 0x02, 0x14

UNKNOWN_MESSAGE

Byte 4: 0x15

remaining bytes: echo of the incoming message (without length bytes)

# 9 Annex II: Detailed explanation of the JFK protocol run

In this section we will briefly describe how the demonstrator should be used. Here, we will give the example where we run the JFK protocol between an Initiator and the Responder. In principle, both processes can be started on different machines, and thus real network communication is used.

When the demonstrator process is started, the user has the choice to start this part of the demonstrator as the Initiator or the Responder (see Figure 24). In order to run the real demonstrator, two processes need to be started, one as initiator and one as responder.

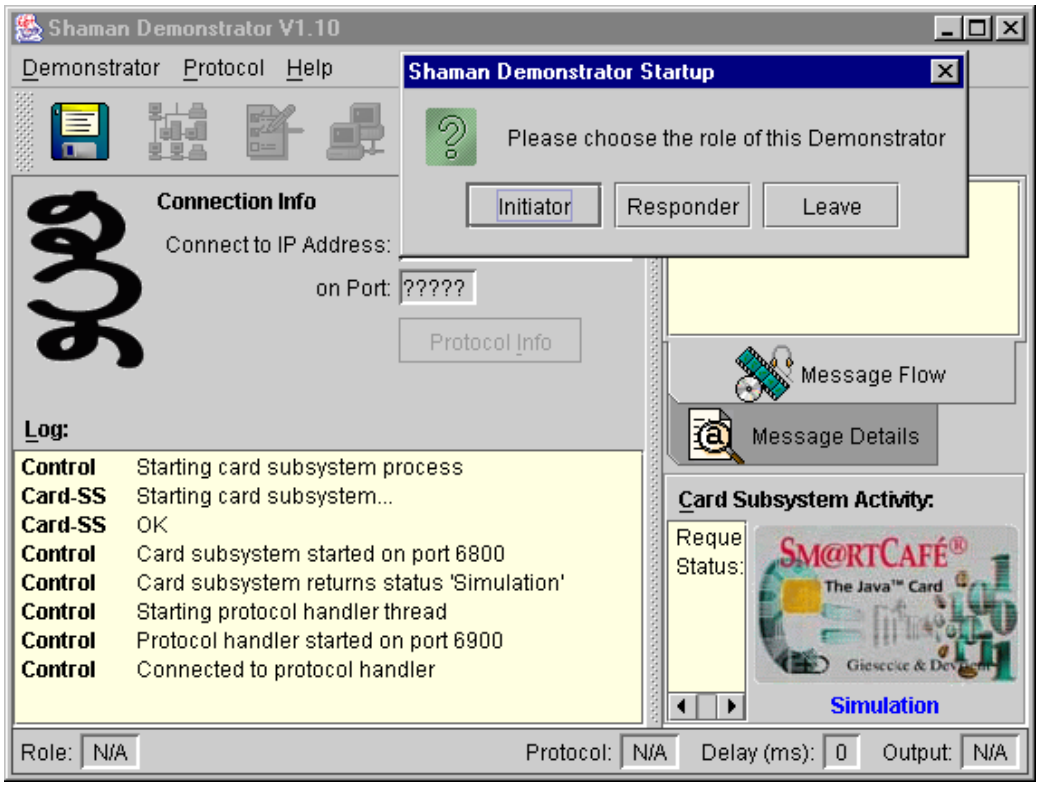


**Figure 24: Demonstrator start-up.**

If we choose this process to run as the Responder, we obtain the initial GUI window as shown in Figure 25; the role is indicated in bottom left corner. The GUI for the Initiator is similar. At the top of the GUI, there are three pull-down windows:

- Demonstrator: This allows one to exit the demonstrator
- Protocol: This is used to choose the protocol to run and to do the subsequent protocols settings.
- Help: additional help on the demonstrator.

Clicking the “Protocol” pull-down list allows us to select which protocol to run. This is illustrated in Figure 26 and Figure 27. Four protocols can be selected, i.e. MANA I and MANA II, for imprinting, EAP-AKA and the JFK protocol. Additionally, a delay time can be introduced, which will slow down the execution of the protocol run (see Figure 27).

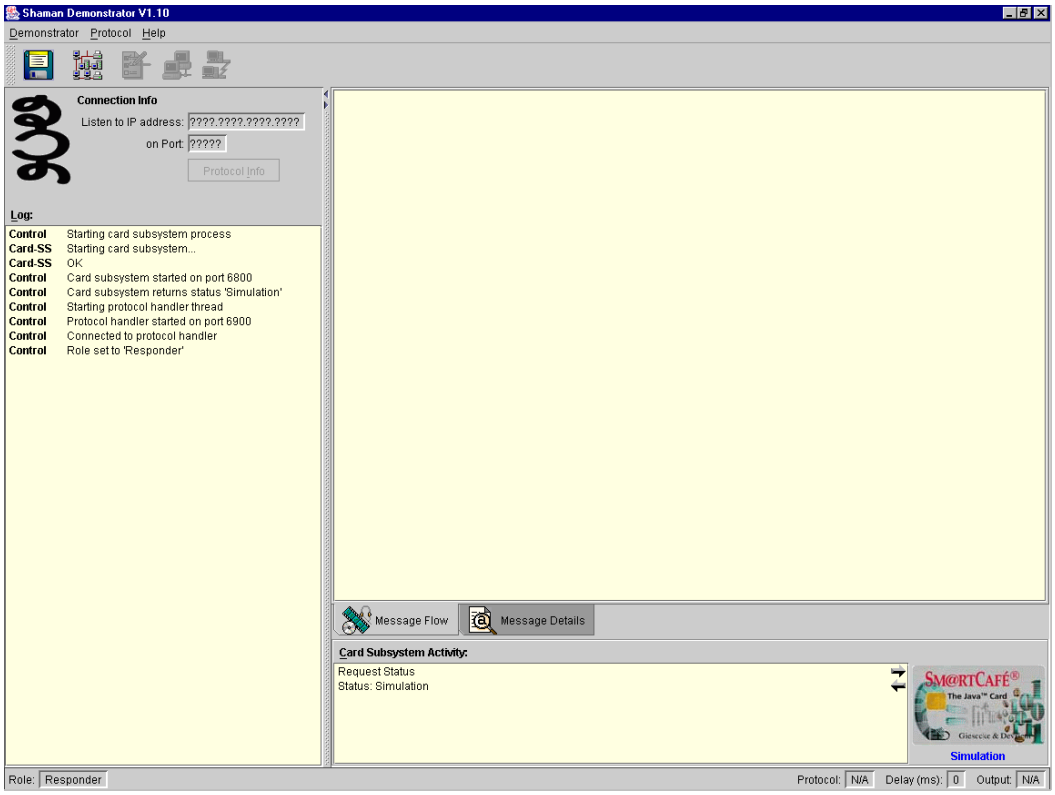


**Figure 25: Initial GUI window for the JFK Responder.**

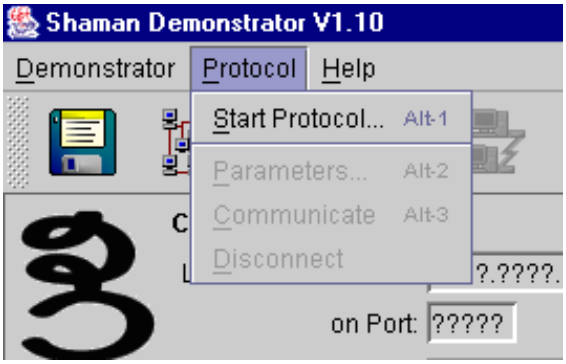


**Figure 26: Protocol selection (1)**

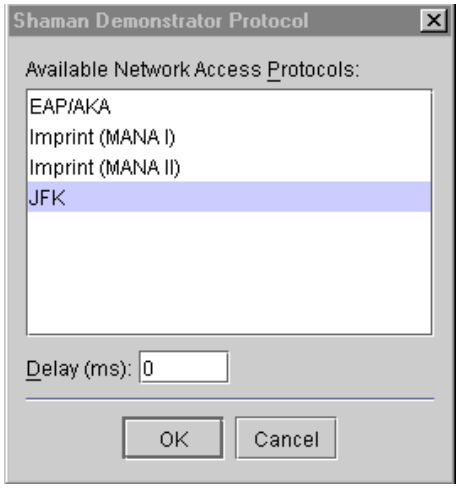


**Figure 27: Protocol selection (2).**

Once the protocol has been selected, the protocol parameters have to be specified. To this end, we first have to choose Protocol->Parameters via the main GUI window (see Figure 26). Now, the different roles of the Initiator and the Responder come into play. The corresponding Initiator and Responder GUI windows are shown in Figure 28. In the Initiator window one has to set the IP-address and port number of the Responder, in the "Responder Properties" section. In the "Security Settings" section we can set the JFK nonce length, the forward secrecy interval and the DH-group and the Certificate to use (this is used in the PSD scenario as outlined in section 3.2.1). For the JFK protocol, only the initiator is able to use the real smart card. The different options can be set in the "SmartCard Use" part. The Responder GUI is very similar. The main difference is that there is no section for the smart card and that the user has the option to select the encryption algorithm, signature algorithm etc. We have to note however that when a real smart card is used only these default settings (for the crypto settings) for the responder can be used; due to the fact that the smart card does not support other crypto-algorithms.

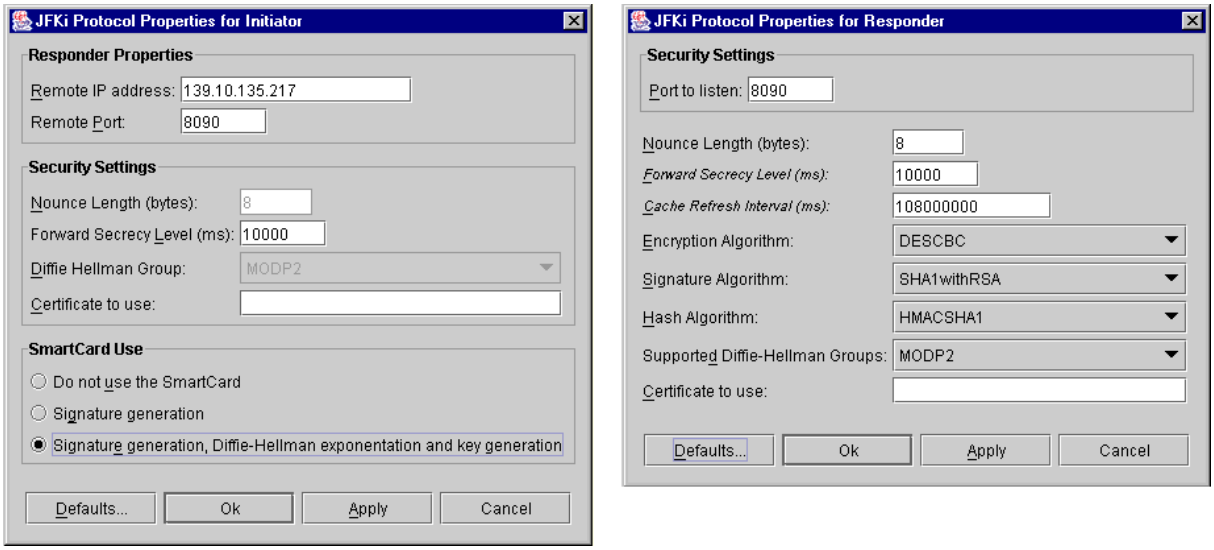


**Figure 28: Protocol parameter settings for the Initiator (left) and the Responder (right).**

Now the protocol can be launched. First, the Responder has to be started, via Protocols->Communicate (see Figure 26), this to prepare a server-socket in order for the initiator to be able to connect. Then the Initiator can be started and the protocol runs between both parties.

The result of the protocol run can be seen in Figure 29 and Figure 30. One the left side of the GUI, there is a log-window, showing the different states and steps the initiator goes through. In Figure 29 on the right side, a graphical representation of the message flow between the Initiator and the Responder is shown. Clicking on the "Message Detail" panel gives more detail about the exchanged messages; this is shown in Figure 30. In the bottom right window, the messages sent to and received from the smart card are shown. Note that here no real smart card is used and the card subsystem works in card simulation mode, as indicated in the figures. The window and the messages will look exactly the same when a real card is used.

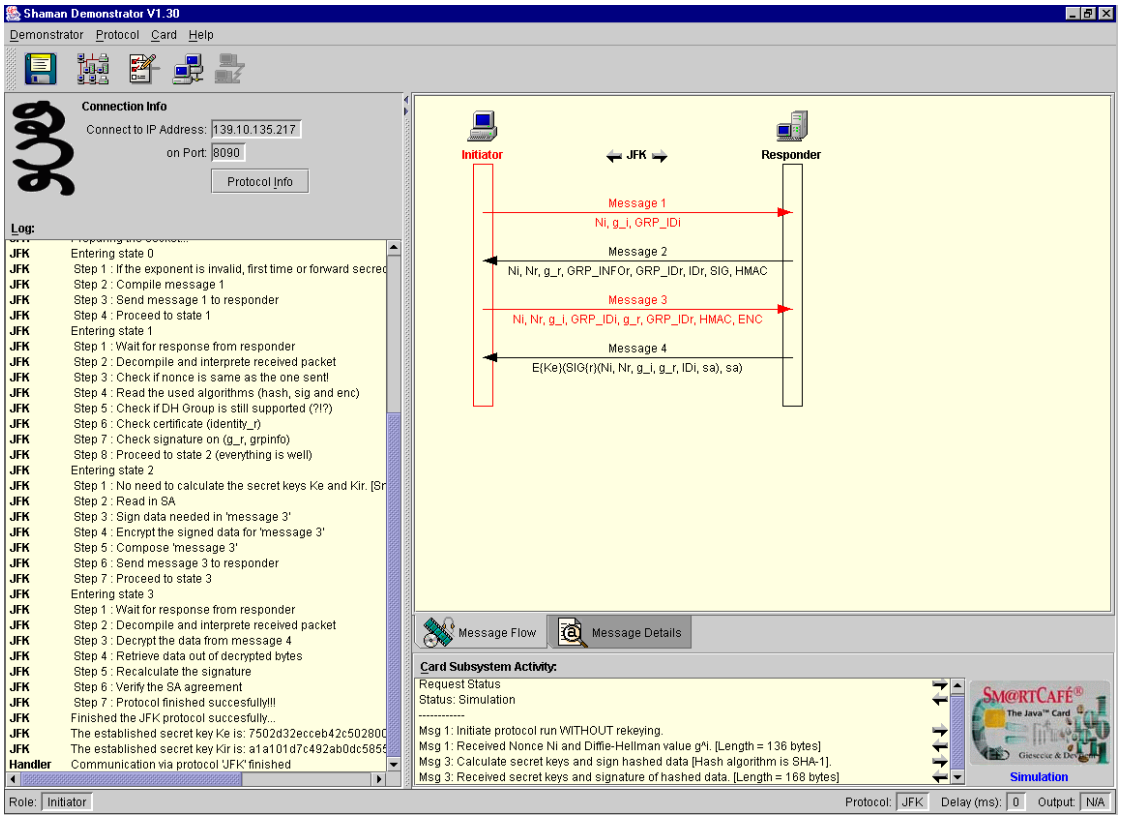


**Figure 29: Message flow for JFK protocol run.**

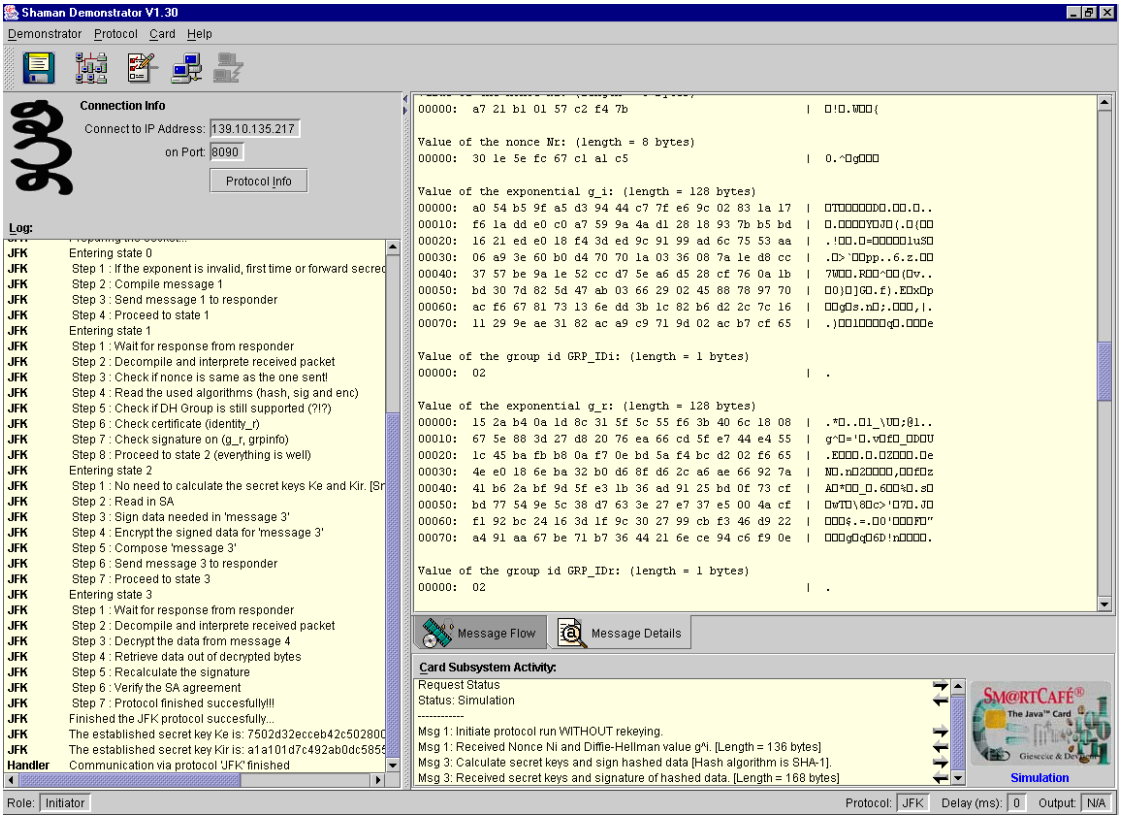


**Figure 30: Message detail for JFK protocol run.**

# 10 Annex III: SHAMAN @ Cebit

Here we include the SHAMAN leaflet introducing the IST-SHAMAN work and the SHAMAN demonstrator at the CEBIT fair.

SHAMANfinal.pdf