AN.ON Documentation

Christoph Dähne, Sebastian Kurfürst

May 5, 2010

Contents

1 Task description **5**

2 Overview and Terminology **7**
2.1 Outline of structure . . . 7
2.1.1 Communication layer . . . 7
2.1.2 User roles . . . 8
2.1.3 Project dependencies . . . 8
2.2 Important Classes and Packages . . . 9
2.2.1 ANONCardApplet . . . 9
2.2.2 GUI . . . 9
2.2.3 ISmartCard . . . 9
2.2.4 Proxy . . . 10
2.2.5 Commands . . . 10

3 Getting started **11**
3.1 Requirements . . . 11
3.1.1 Subsystems . . . 11
3.1.2 Libraries . . . 11
3.2 Set up workspace . . . 12
3.2.1 SVN directories . . . 12
3.2.2 Eclipse . . . 12
3.3 jUnit tests . . . 13
3.3.1 ANONCardTests . . . 13
3.3.2 Run Applet in Emulator . . . 13
3.4 Install applet . . . 13
3.5 Run applet . . . 13

4 How to write new ANONCardApplet methods **15**
4.1 Single APDU request . . . 15
4.2 Extended Response . . . 17
4.3 Multi APDU request . . . 17

5 Implementation Details **19**
5.1 GUI and Model . . . 19
5.2 ANONCardApplet . . . 21
5.2.1 Flow of a request . . . 21
5.2.2 Configuration and transactions . . . 22
5.3 Virtual data channels . . . 23

5.3.1 DataSendChannel 23
5.3.2 DataReceiveChannel 23

6 Outline and open topics **25**
6.1 Communication with timeserver 25
6.2 Signatures of seed 25
6.3 Optimizes applet memory allocation 25
6.4 Optimize applet performance 25
6.5 PIN input via card reader 26
6.6 Automated testing of an installed applet 26

Chapter 1

Task description

For the JAP service of the TU Dresden we programmed an applet for Java smart cards. JAP provides anonymity and unobservability to its users while surfing the internet. But it needs to do some data retention. This data must be accessible quickly on federal requests for some given time, but naturally be inaccessible after that time period as well as for unauthorized requests from inside and outside.
Therefore the data is encrypted with a symmetric key, which again is encrypted asymmetrically. In order to receive the key for a particular date and Mix this encrypted information must be sent to the ANONCardApplet along with a configurable number of valid operator logins. The card checks the current date and returns the decrypted symmetric key logging the event, time and date, and involved operators.
Our task was to write this applet and parts of the MixConfig tool, which we completed partially. See 6 for more details.

Chapter 2

Overview and Terminology

2.1 Outline of structure

Some layers are explained in more details in 5 or 4.

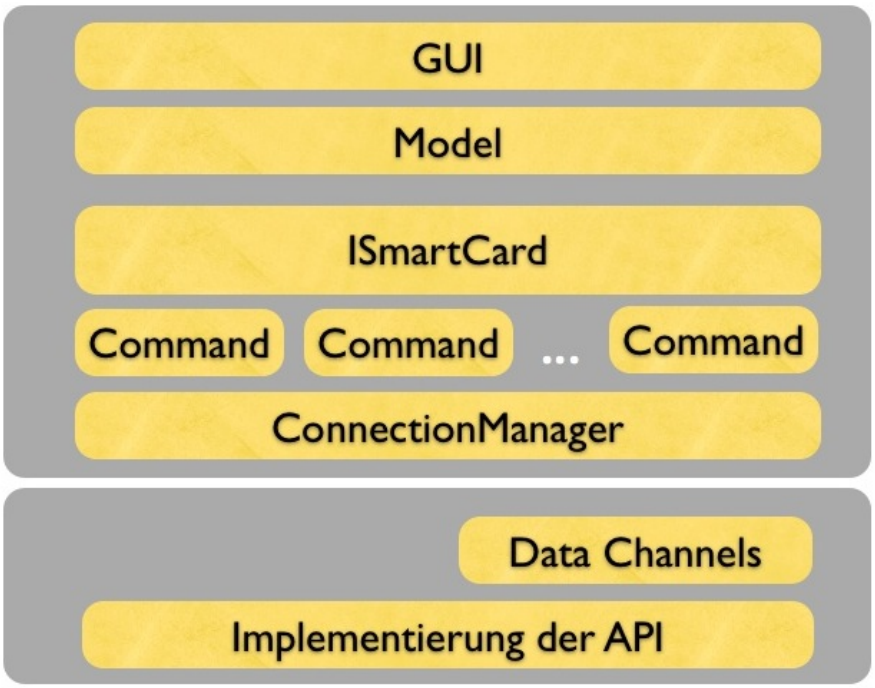

Figure 2.1: program layers

2.1.1 Communication layer

As can be seen in Figure 2.1 no direct communication with the smart card is needed. Using the ISmartCard and the implementing class SmartCardProxy all

method calls are forwarded to the smart card and the arguments are transformed into byte streams, the return values back to "normal" data.

2.1.2 User roles

The ANONCardApplet knows two types of users with different capabilities as shown in Figure 2.2. Note that one administrator or operator alone cannot necessarily do anything. A configurable minimum number of administrators or operators must be present. Some of the features can be switched of and the Figure does not show all of them.

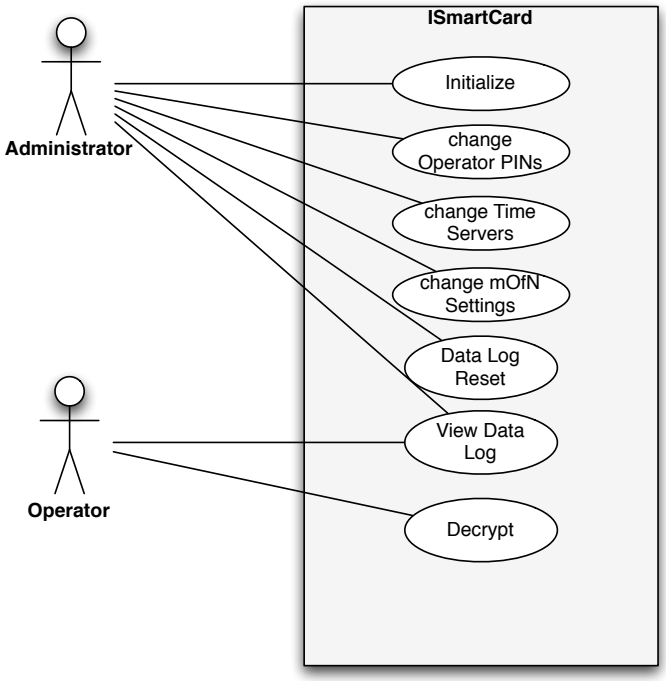

Figure 2.2: User roles

2.1.3 Project dependencies

Dependencies are shown in Figure 2.3. To avoid errors in eclipse, the ApduConstants class must be compiled. See 3.2.2 for more details.

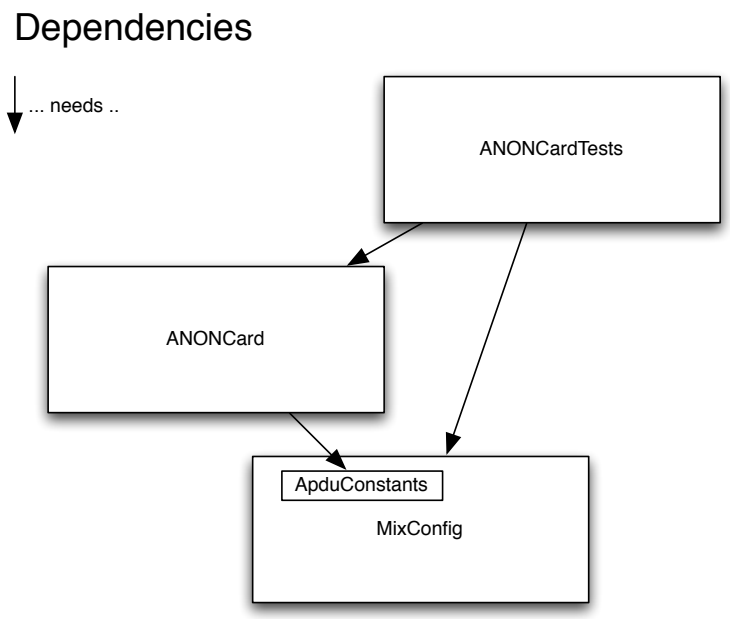

Figure 2.3: Project dependencies

2.2 Important Classes and Packages

2.2.1 ANONCardApplet

The ANONCardApplet is installed at the smart card and is the only class accessible from outside of its package. All requests will invoke a call of the method process. We do not advice to build and transmit the APDUs by hand but to use Commands and our ConnectionManager. How to do that is written in 4 for more details about the ANONCardApplet see 5.2.

2.2.2 GUI

The GUI is implemented in MixConfig, inside the package mixconfig.panels.smartcard. The GUI itself (InstallationWizard.groovy) is backed by a model (InstallationWizardFormModel.groovy). This means all validation and data checking happens in the Model, and is displayed in the GUI. Additionally, the Model can save itself to the card, and load its data from the smart card (via the ISmartCard interface, explained below). More details can be found at 5.1.

2.2.3 ISmartCard

We found it useful to use the smart card with the ANONCardApplet installed on it as a plain Java object, without any need to encode or decode data. All communication is wrapped in this interface.

2.2.4 Proxy

The SmartCardProxy is our implementation of the ISmartCard. It encodes arguments, builds commands, gives them to the connection manager and decodes its responses. Also some local prechecking of arguments is done to safe some performance. Some methods are still not implemented and performance can be increased as well (see 6).

2.2.5 Commands

APDUs, since they are basically byte streams, strongly lack readability. We build Commands instead, encapsulating different parts of an APDU into documented arguments with readable names.

Chapter 3

Getting started

3.1 Requirements

3.1.1 Subsystems

For development the following is required. The stated version numbers except for the JCDK might not matter, but this configuration has worked fine on a Ubuntu system.

- JDK 1.5 (Java Development Kit)
- AspectJ
- Eclipse Galileo
- JCDK 2.2.1 (Java Card Development Kit) on a card that supports garbage collection
- Subversive for Eclipse

3.1.2 Libraries

Some external libraries are automatically downloaded by the ant build (init-libs, init-jondo, init-groovy) of the MixConfig project:

- Bouncy Castle APIs
- Groovy 1.7.0
- Miglayout 3.7.2
- APDU libraries of the JCDK 2.2.1 (Java Card Development Kit)
- API of the JCDK 2.2.1

3.2 Set up workspace

As mentioned before, this work consists of several eclipse project, each having its own SVN repository:

- ANONCard/
- ANONCardDoc/
- ANONCardTests/
- MixConfig/

3.2.1 SVN directories

All folders can be checked out in https://anon.inf.tu-dresden.de/svn/JavaCardStudents/. We advice to finish reading this section before doing that.

ANONCard

This project contains the applet, that is first deployed and then runs on the Java smart card. Although it is written in kind of Java, note that the smart card usually supports only a small subset of Java.

ANONCardDoc

This project contains this documentation written in LaTeX, but naturally the Java source files contain the Javadoc.

ANONCardTests

All jUnit tests belong to this location. They can be run with ant.

MixConfig

This is the configuration tool for the Mixes, which includes the configuration panel for the smart cards.

3.2.2 Eclipse

To set up your Eclipse workspace do the following:

1. Start Eclipse and install Subversive
2. Check out ANONCard/trunk, ANONCardTests/trunk, MixConfig/trunk and ANONCardDoc
3. Run the following targets from MixConfig/build.xml with ant:
   - init-libs
   - init-jondo
   - init-groovy
   - compile

If you get some error message while performing init-jondo concerning an unknown fingerprint, open a terminal and enter:

```
svn list https://svn.jondos.de/svnpub/JonDonym/trunk
```

3.3 jUnit tests

3.3.1 ANONCardTests

The directory tests contains all jUnit test cases. They are in the same package as the unit under test. In order to test the ANONCardApplet we found it necessary to access private fields and emulate the behaviour of the smart cards operating system. The provided API library is incapable doing that.
Therefore we manipulate and extend classes using AspectJ. All AspectJ files can be found in ANONCardTests/tests/aspects. Note that altering those files might cause the jUnit tests to fail.

To run the test suite just run the build.xml with ant. The default target is runall. In order to run single test (i.e. ANONCardAppletTest), just start the target (ANONCardAppletTest). The tests are **not recompiled automatically**, when run one by one.

New tests can be added to the according package and the build.xml file needs to be extended.

3.3.2 Run Applet in Emulator

If you want to run the applet in the emulator you need to change the constructor a bit. Just look inside. Then open two terminals, go to ANONCardTests and run:

```
ant emulate
```

```
ant runMixconfigEmulated
```

3.4 Install applet

Go to the folder ANONCard and run:

```
ant deploy
```

3.5 Run applet

Go to the folder MixConfig and run:

```
ant run-wizard
```

Chapter 4

How to write new ANONCardApplet methods

The smart card can only communicate by APDUs. With this implementation, the size of an APDU is limited to 127 bytes. Thus in theory all method calls with arguments bigger than 121 bytes ($127 - 6 = 121$) are split into multiple APDUs. The same applies to the responses, which can only contain 99 bytes each. This value appeared in practice and we are not sure where it comes from.

4.1 Single APDU request

We now will write an example method, which already exists: getVersion(). It needs no arguments and returns two bytes, indicating the ANONCardApplets version.

First an unique instruction code has to be added to ApduConstants in mixconfig.tools.dataretention.smartcard.

```
public class ApduConstants {
  ...
  public final static byte
  INSTRUCTION_GET_VERSION = (byte) 0x20;
  ...
}
```

Next we need a new command which must extend AbstractCommand. The source of AbstractCommand is very short and easy to read. So take a look inside.

```
public class GetVersionCommand extends AbstractCommand {
  public GetVersionCommand() {
    instruction =
    ApduConstants.INSTRUCTION_GET_VERSION;
    expectedReturnDataLength = 0x0F;
  }
```

```
  public String getVersion() {
    return (int)rawResponse[0] +
    "." + (int)rawResponse[1];
  }
}
```

Now we can use the GetVersionCommand in the SmartCardProxy in mixconfig.tools.dataretention.smartcard. Don't forget to add the new method in the ISmartCard interface.

```
public class SmartCardProxy implements ISmartCard {
  ...
  @Override
  public String getVersion() throws ErrorCodeException {
    GetVersionCommand command = new GetVersionCommand();
    connectionManager.processCommand(command);
    return command.getVersion();
  }
  ...
}
```

Last thing to do is to add the new method actually to the ANONCardApplet. This requires two steps:

1. Write the method

2. "Register" it in the callInstructionByCode method

```
public class ANONCardApplet extends ... {
  ...
  private void callInstructionByCode(...) {
    switch (instructionCode) {
      case ApduConstants.INSTRUCTION_GET_VERSION:
        getVersion(apdu);
        break;
      ...
      default:
        ISOException.throwIt(...);
        break;
    }
  }
  ...
  protected void getVersion(APDU apdu) {
    sendReply(apdu, new byte[] {
      APPLET_VERSION_MAJOR, APPLET_VERSION_MINOR
    });
  }
  ...
}
```

4.2 Extended Response

If you want to receive responses longer than 99 bytes, you need to use the DataSendChannel (see 5.3.1) of the ANONCardApplet. The response is then automatically split into multiple APDUs.
The first two steps are the same:

1. Choose an unique instruction code (see 4.1)
2. Create a command class (see 4.1)

As an example, we consider the method getLogEntry which returns the i-th entry of the ANONCardApplets log.
Now take a look inside the SmartCardProxy:

```
public class SmartCardProxy implements ... {
  ...
  @Override
  public List<LogEntry> getLog() {
    ...
    for (int i = 0; i < getLogSizeCommand.getSize(); i++) {
      GetLogEntryCommand getLogEntryCommand = new GetLogEntryCommand(i);
      try {
        connectionManager.processCommand(getLogEntryCommand);
        int logEntryLength = getLogEntryCommand.getLength();
        byte[] response = getExtendedResponse(logEntryLength);
        log.add(new LogEntry(...));
      } catch (ErrorCodeException e) {...}
    }
  ...
  }
}
```

The GetLogEntryCommand returns the length of the data to receive in bytes and afterwards the data packages can be polled with getExtendedResponse(logEntryLength). The changes to be made on the ANONCardApplet are quite the same as before (see 4.1). Although the DataSendChannel needs to be initialized.

```
protected void getLogEntry(APDU apdu) {
  LogEntry logEntry = log.getLogEntry(...);
  if (logEntry != null) {
    dataSendChannel.open(logEntry.getData());
    sendReply(apdu, dataSendChannel.lengthAsBytes());
  } else {
    ISOException.throwIt(ApduConstants.EXCEPTION_ILLEGAL_ARGUMENT);
  }
}
```

4.3 Multi APDU request

Sending extended method arguments to the ANONCardApplet can be done using the DataReceiveChannel (5.3.2) of the ANONCard. The steps again are very much the same as in 4.1, except three details:

1. No Command class is needed
2. SmartCardProxy and
3. ANONCardApplet method are slightly different

We never needed to receive response data after sending an extended argument request to the ANONCardApplet, but it is possible to implement, i.e. with few modifications to the AbstractCommand.
As an example, we show how to reset the log. The extended argument data is a list of valid administrator $(name, PIN)$ pairs.

In the SmartCardProxy you just use the method doDataTransmission, which will initialize and use the ReceiveDataChannel of the ANONCard.

```
public class SmartCardProxy implements ... {
  ...
  @Override
  public void resetLog(Map<String, String> administrators) throws ... {
    checkAdministratorMap(administrators);
    doDataTransmission(
      ApduConstants.INSTRUCTION_RESET_ANONCARDAPPLET_LOG,
      Helpers.packUsers(administrators)
    );
  }
  ...
}
```

To receive the transmitted data in the ANONCardApplet you can check, weather there is any in the channel and access it using the dataReceiveChannel field. The method call to resetLog is only invoked if the data transmission is completed. (The additional checking if the transmission has completed is done, because this method supports both: single and multiple APDU requests.)

```
public class ANONCardApplet extends ... {
  ...
  protected void resetLog(APDU apdu) {
    ...
    if (dataReceiveChannel.isTransmissionComplete()) {
      authenticated = checkAdminPermission(
          dataReceiveChannel.getData(), (short) 0,
          dataReceiveChannel.length()
      );
      names = getNamesFromBytes(...);
      dataReceiveChannel.close();
    }
    ...
  }
  ...
}
```

Chapter 5

Implementation Details

5.1 GUI and Model

The GUI to configure the smart card is implemented in two parts: The GUI itself, and the underlying model which stores the data. As building a GUI with standard Java is very cumbersome, we decided to use Groovy as programming language (which is then compiled to Java byte code), and an open source layout manager called *MigLayout*[1]. As groovy has support for Domain Specific Languages, it is very easy to build the GUI declaratively. Following is an excerpt showing this.

```
def frame = swingBuilder.frame(
   title:'Installation Wizard',
   size:[300,300],
   pack: true,
   visible: true
  ) {
  panel(layout: new MigLayout("fill", "grow", "top")) {
  errorMessages = label(text: "<html>", constraints: 'wrap')
  tabbedPane(constraints:"") {
    panel(title: 'Administrator Settings', layout: new MigLayout()) {
      label(text: "needed number of administrators")
      textFieldConnectedToModel(
         targetProperty: 'neededNumberOfAdministrators',
         dataType: 'int',
         constraints: "wrap"
      )
      checkBoxConnectedToModel(
         text: '... can delete/create other admins...',
         targetProperty: "allowedToModifyAdministrators",
         constraints: "span"
      )
      userEditingAreaConnectedToModel(
         role: 'Administrator',
         targetProperty: 'administrators'
```

[1] http://www.miglayout.com/

```
      )
    }
    ... [more tabs] ...
  }
}
```

Here, the GUI hierarchy can be directly seen from the code. Additionally, we created some special helper methods which are used to bind the form to the underlying model, all methods ending with *connectedToModel. There, one could set the targetProperty, which is the target property of the model the form element should be bound to.

Now have a look at the corresponding model (InstallationWizardFormModel): here, two things are very important:

- Every model class needs to have the @Bindable annotation on top of it, as the GUI needs to install some property change listeners.
- All *lists* in the model need to be of the type ObservableList, as all lists need property change notification support as well.

Figure 5.1 shows how the model and the user interface works together, and how the listeners work: Simple form fields work the most straightforward way. On form initialization, the data is loaded from the model and displayed in the form. Additionally, a change listener to the form field is added, which changes the model on every value change (and this, in turn, triggers validation – see below).

For compound properties, like lists of users or files, it is a bit more complex. If the button to add a field is pressed, a new element is simply added to the array in the model. This triggers a callback which then refreshes the user interface, displaying the new element. The editing of the individual values works as described above. Removing an element works the same way as just described.

Additionally, validation constraints can be defined in the model as follows: All constraints are listed in a Map with the name "constraints". The key for each constraint is the field to which this constraint applies (or GLOBAL if the constraints are not tied to a single field), and the value for each constraint is an anonymous function which checks the constraint, and returns an array of error messages if the validation fails, or null if there was no validation error.

All this can be seen in the following code snippet:

```
@Bindable
class InstallationWizardFormModel extends AbstractFormModel {
  protected def constraints = [
    neededNumberOfAdministrators: {
      if (this.neededNumberOfAdministrators > this.administrators.size()) {
        return ["<b>Needed number of administrators</b> must be
          smaller or equal than the total number of administrators!"]
      }
      return null
    },
    ... more constraints follow here ...
```

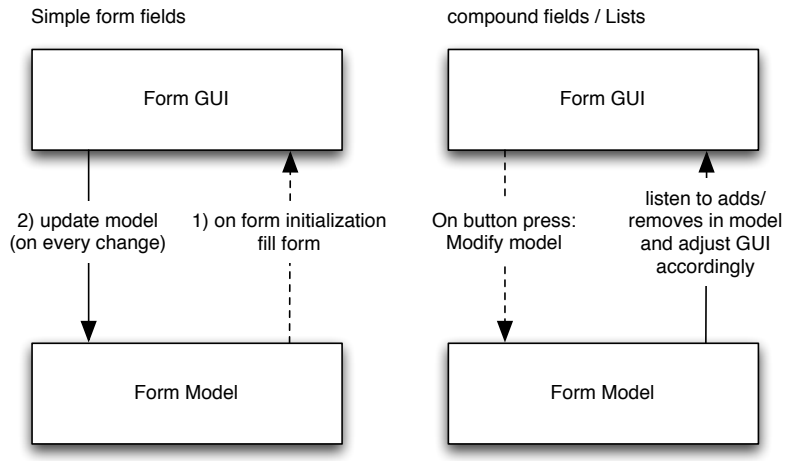

Figure 5.1: Form fields

```
  ]
}
```

Now, only one question remains: How is the model saved to the smart card, and restored from the smart card? For this, the model has the method saveToCard(), which connects to the card, and commits all changes atomically, i.e. inside a transaction. For more details on the implementation of the transaction, see section 5.2.2. For loading the data from the card, the method loadFromCard() is used.

5.2 ANONCardApplet

The ANONCardApplet holds the encryption keys and checks permissions on requests. Almost every possible instruction code has its own protected method. Short examples can be found in 4. This section only explains parts of the applet, which cannot be documented well enough using Javadoc. You need to read the Javadoc! The applet overrides some methods from its superclass. We advice to read its documentation, too.

5.2.1 Flow of a request

In Figure 5.2 you can see, how a APDU request flows throw the ANONCardApplet. This only holds for single APDU requests (see 4.1). Not shown in this overview the DataReceiveChannel (see 5.3.2) is closed, too. In the case of an extended APDU request (see 4.3) the method calls are slightly different as can be seen in Figure 5.3. The first APDU opens the DataReceiveChannel (see 5.3.2) and already determines the instruction code for the method call (myOperation) which is executed after all argument data has been transmitted. All

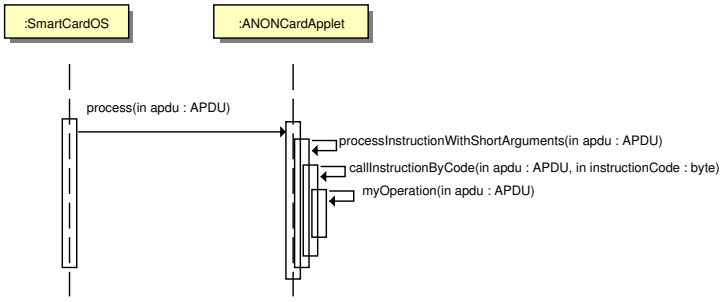

Figure 5.2: Method calls in the ANONCardApplet on a single APDU request

but the last APDU deliver pieces of data to the data channel and the last one additionally invokes the method call. Note, that the data channel is not closed automatically, but will be as soon as the next APDU arrives. Naturally it can be closed "by hand" in myOperation.

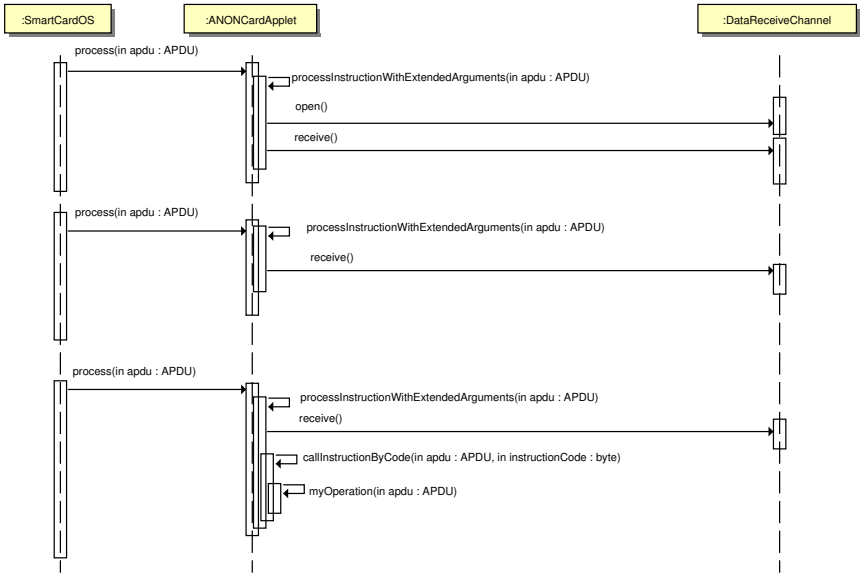

Figure 5.3: Method calls in the ANONCardApplet on an extended APDU request

5.2.2 Configuration and transactions

The settings are saved in the ANONCardApplet:configuration object. Since users can be removed and added changes on the configuration have to be atomic and complete. To achieve that, we use transactions (see Figure 5.4). To change the configuration you must follow these steps:

1. Start a transaction
2. Change settings

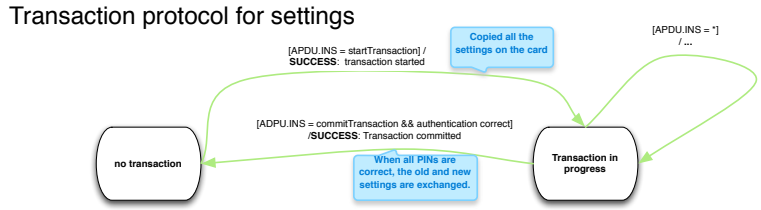

Figure 5.4: Transaction protocol to change settings

3. Commit the transaction with **old** administrator logins

The configuration objects are never initialized after the installation of the applet. See its constructor for more details.

5.3 Virtual data channels

Some argument or response data extends the size of one APDUs (see 4.2 and 4.3) and need to be split into multiple ones. Since the channels are placed on the smart card, the channel to send data to the smart card that it must receive is called DataReceiveChannel and vice versa.

5.3.1 DataSendChannel

The protocol is very simple. When a response is too long, the DataSendChannel needs to be opened and the data can be polled one package at a time using the GetNextDataPackageCommand. Each data package is transmitted exactly once and the channel is closed automatically, when all data has been sent. See 4.2 for an example.

5.3.2 DataReceiveChannel

For extended arguments the DataReceiveChannel is to be used. See 4.3 for an example. The protocol is shown in Figure 5.5. The DataReceiveChannel was once named DataTransmissionChannel, the name of the instruction code constants stayed the same, but are also defined in the ApduConstants class.

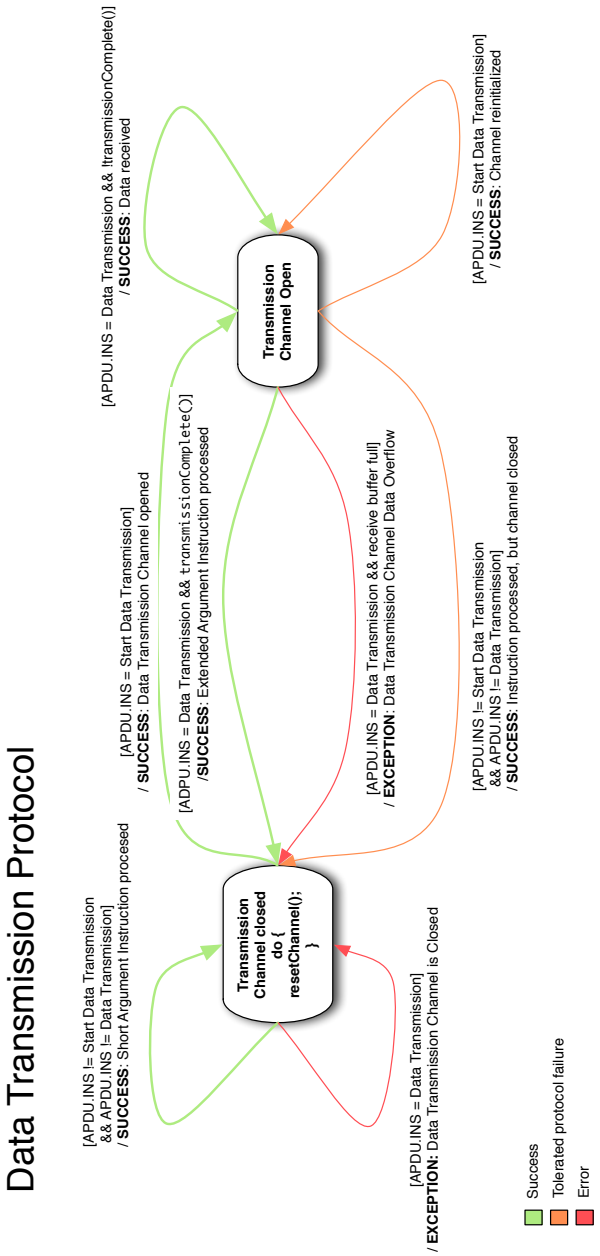

Figure 5.5: Protocol of the DataReceiveChannel - APDU.INS is the APDU byte containing the instruction code

Chapter 6

Outline and open topics

6.1 Communication with timeserver

The ANONCardApplet is supposed to check, whether a symmetric key must be still delivered or whether the time period for data retention already has passed. While decrypting the applet can determine the day of creation of the key. As todays date, the applet uses system time of the MixConfig tool, which is quite insecure. Instead the card should communicate with several timeservers using the attached PC as a proxy in a secure way.

6.2 Signatures of seed

Saved symmetric keys are an asymmetrically encrypted tuple of $(Kr, date)$ where Kr is some kind of random seed for the symmetric key key. In order to improve security it would be nice to sign it additionally alike $(Kr, date, sign(hash(Kr, date)))$ and implement that in a way that the date neither can be altered nor a new tuple can be created with the same Kr and another $date'$ which would result in the same key.

6.3 Optimizes applet memory allocation

The current implementation is not designed for low memory consumption. The only slight optimization is done for logging. We neither use the transient memory of the smart card, which would make sense i.e. for the data channels. A lot of optimization can be done here, too.
Additionally it would be very nice to not need garbage collection. To achieve that the only memory allocation, for objects as well as for plain data types, has to be done only during the installation of the applet.

6.4 Optimize applet performance

The applet neither is design for high performance. The applet code can be optimized for sure and the current SmartCardProxy i.e. lacks caching.

6.5 PIN input via card reader

To increase security further PINs can be typed in on some smart card readers instead the PC. It was optional for our task to make it possible. Currently PINs are entered always via the PC. We can imagine to add the possibility to configure an applet that it enforces the use of such devices and blocks access if the card reader has no input field.

6.6 Automated testing of an installed applet

The test suit so far compiles the ANONCardApplet and other components using a standard Java compiler. This allows developers to use nice features like jUnit, debugging with breakpoints and fast testing on home computers without smart card readers. On the other side, some classes needs to be manipulated with aspects and the behaviour of the smart cards OS might differ from the expected behaviour simulated using the aspects. Also the encryption, a vital function of the card, can be tested poorly.
Therefore it would be a great improvement to have test, that communicate with a real ANONCardApplet installed on a not-emulated, real smart card. The tests must be possible to be run automatically and to be extended.