Two Factor Authentication

Problem statement

Standard VPN authentication mostly consists of a username and password combination. This means that anyone knowing this combination could get immediate access to the system. To make a stronger authentication process, often a second way of identity validation would be introduced. Examples are an additional PIN-code, fingerprint validation, RSA based tokens, etc.

We would like to enable Two Factor Authentication by means of a smartphone. Whenever the user signs on to his VPN account, he will receive a notification on his smartphone device. Only responding to this unique notification will give him eventually access to the system.

A second option – for those not possessing a smartphone device – would be a SMS containing a specific code being send to the mobile phone of user during his VPN authentication process. Besides the username and password combination, also the received code has to be entered to ensure a successful sign-on.

Architecture

Common setups in current VPN implementations consist of a VPN client accessing a VPN server. This server normally resides within a DMZ. For the identification validation the VPN server will query a Network Policy Server (NPS). The communication between the VPN and NPS server will normally be based on the Remote Authentication Dial In User Service (RADIUS) protocol (see Figure 1: typical VPN implementation).

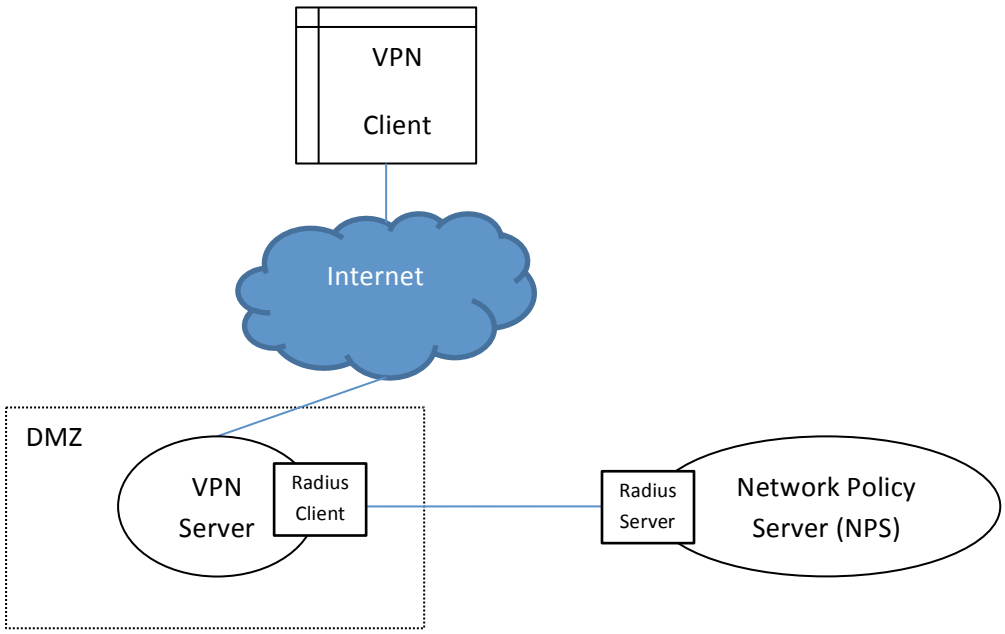


Figure 1: typical VPN implementation

This architecture has to be extended to supply a Two Factor Authentication sign-on based on an additional smartphone acknowledgement. Therefore we introduce three new components (see

Figure 2: extended VPN architecture to support TFA):

- JF-NPS Extension DLL
- JF-TFA Module
- JF-TFA App

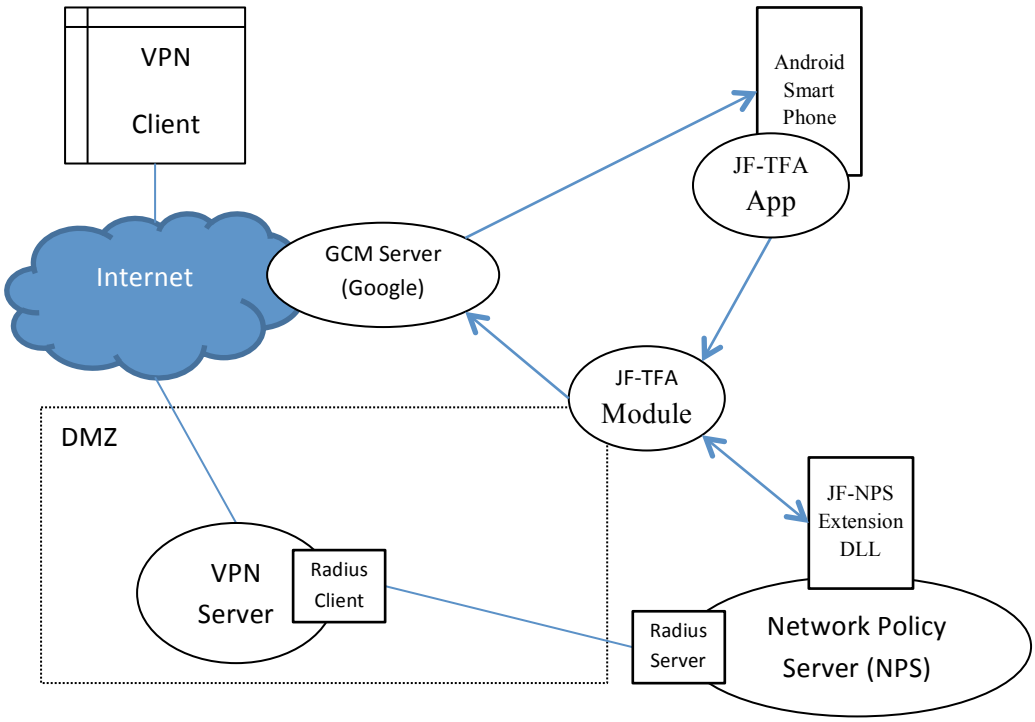


Figure 2: extended VPN architecture to support TFA

Because Android has the largest market share by far (about 70%) we will concentrate our architecture on this Google platform first.

JF-NPS Extension DLL

Microsoft's Network Policy Server executes the authentication requests from the VPN Server. It replies with the result of the authentication. To enable custom authentication steps, one or more DLLs with a predefined callback function can be registered for this service. These DLLs will be invoked before the default authentication of NPS. Dependent on the return value of the callback function the NPS continues the authentication process.

This extension model makes it possible to include additional authentication steps like our smartphone based TFA.

JF-TFA Module

We are forced to take a closer look at smartphone technology if we are going to introduce this in our TFA solution. Since we will introduce additional functionality at the smartphone itself we have to investigate its implications.

A choice between a push or pull communication model has to be made in information exchange

between two parties. Do we have both options available here and if so, what are their pros and cons.

Pull model

Within a pull model the smartphone has to query at frequent times whether a new request for authentication is available. This implies that the smartphone needs a running background application at all times to support this feature. It will be very inconvenient to a smartphone owner to first startup the authentication application before he sets up a VPN connection. However, running an application with frequent polling requests will definitely have its impact on battery consumption. Since this should be avoided to any degree, the pull model violates the low battery consumption requirement and is therefore not very suitable.

Push model

An app installed on the smartphone should get awakened if a new authentication request arrives. To accommodate this model special services have been introduced by parties like Google and Apple. After sending a 'request for notification' to the special service, as soon as possible the smartphone will be notified about this event. The smartphone owner will see a popup and can start up the app responsible for handling this notification from here. It is up to the app how to handle this event. So the app is only running shortly to minimize battery consumption.

Using the push model will make it necessary to introduce a module that will be responsible for registering the 'request for notification' at the special service and handling the authentication confirmation from the smartphone afterwards. Therefore we introduce the "JF-TFA Module".

Preferably this module would be placed within a DMZ to avoid direct access from the smartphone to a protected network environment.

It is obvious that there has to be some kind of communication between "JF-NPS Extension DLL" and the "JF-TFA Module". A request for smartphone authentication will always be initiated from the "JF-NPS Extension DLL". Besides that, there might be a firewall between the two – and the preferable way of setting up communications is from inside to outside the firewall – the communication channel could best be setup from the "JF-NPS Extension DLL".

JF-TFA App

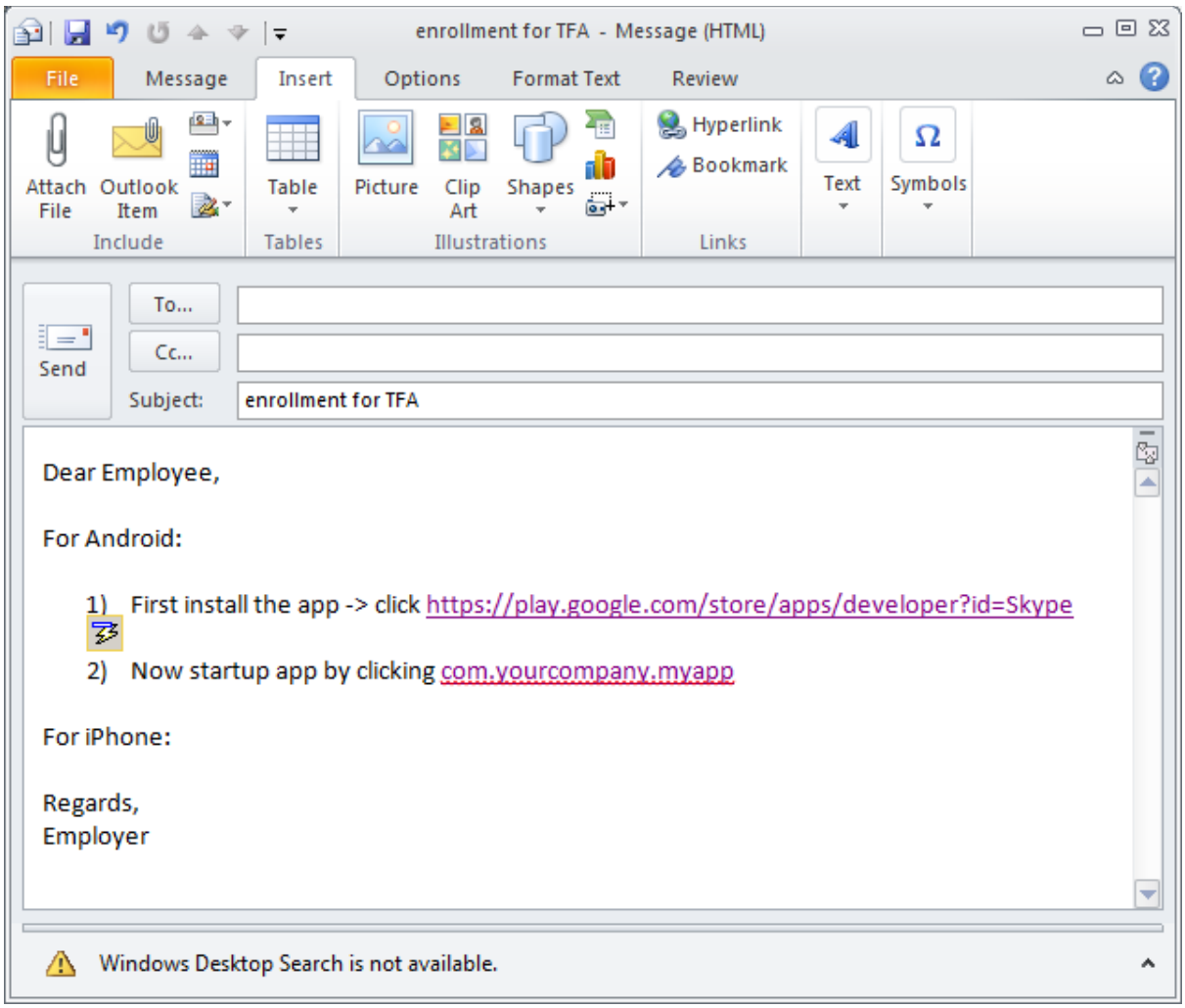
The second factor in our Two Factor Authentication will be the confirmation by a user from his smartphone when his mobile receives an authentication request during sign-on.

This function will be implemented in a so-called app that has to be installed on the smartphone device. For Android this means an application written in Java using Google's GCM services. At user confirmation, after a GCM notification from the "JF-TFA Module" has been received, the app could send a regular HTTP message to the "JF-TFA Module" with its response.

JF-TFA Enrollment

Possible choices to enroll the smartphone device for TFA are:

1. url/username/password to be filled in app enrollment wizard
2. e-mail containing link to app and all needed information: e-mail starts app and passes enclosed url



A typical scenario for this choice will look like:

a) User executes a VPN authentication
b) The "JF-TFA Module" manages enrolled devices and if not enrolled yet
c) Send e-mail with app url + info to registered e-mail address of authenticating user

3. Smartphone managing software

JF-TFA Implementation

Scenario's

ARS = Authentication Request Service (e.g. a Radius server)

APS = Authentication Provider Service (e.g. Google+ Sign-in)

SAS=Smart Authentication Service ("JF-TFA Module")

SNS = Smartphone Notification Service (e.g. Google Cloud Messaging)

Device = Smartphone (e.g. Android device)

Scenario 1: User has not been enrolled yet

This means that the smartphone device of the user has not been registered at the APS service while the user has been administered to use TFA.

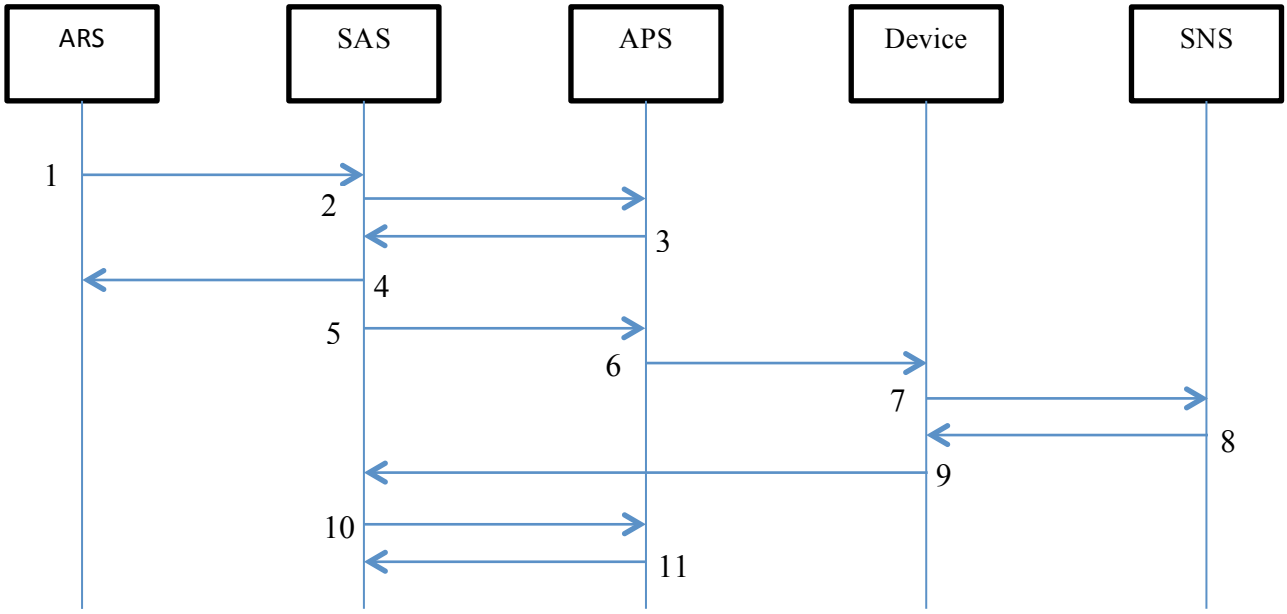


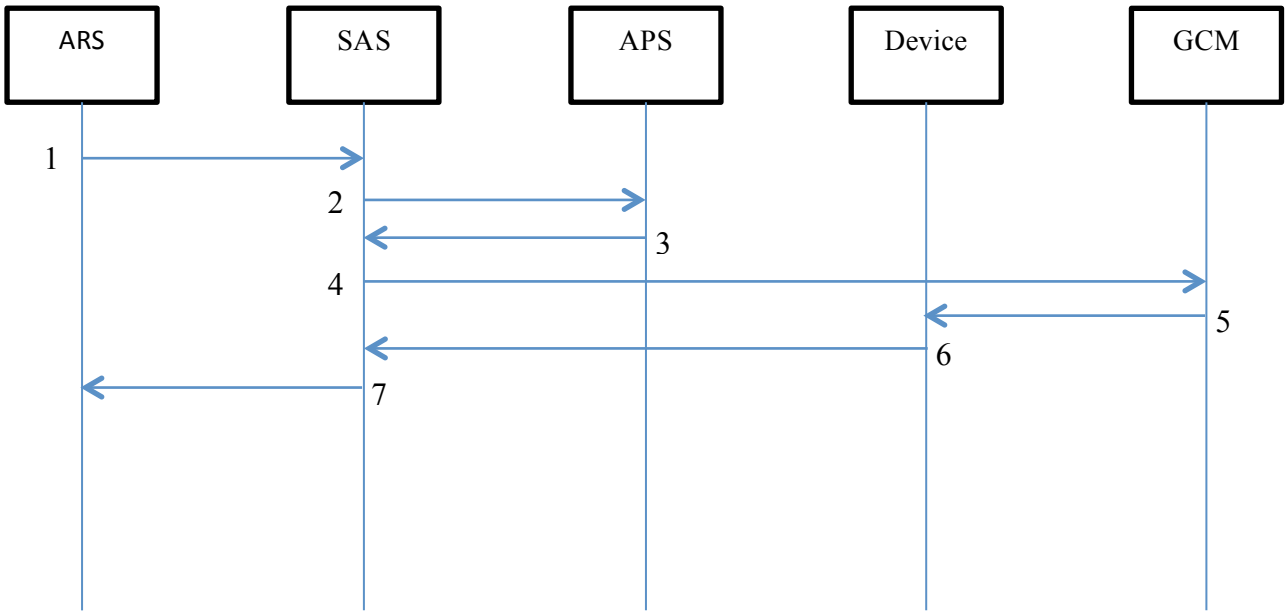
Figure 3: Authentication request NOT enrolled

1	Authentication request (second factor) for username/password
2	UserInfo request for username
3	Enrollment status (not enrolled yet) + additional userinfo like e-mail address of user
4	Login status (OK/NOK) dependent of enrollment window
5	Request to send enrollment e-mail to user
6	Send (using smtp) enrollment e-mail to user
7	Register device at SNS service (using info from e-mail)
8	Receives registration-id (device-id) from SNS service

9	Send registration-id to SAS (using info from e-mail)
10	Register registration-id/service (GCM|APNS|…) for user (on success Enrollment status = enrolled)
11	Returns register response

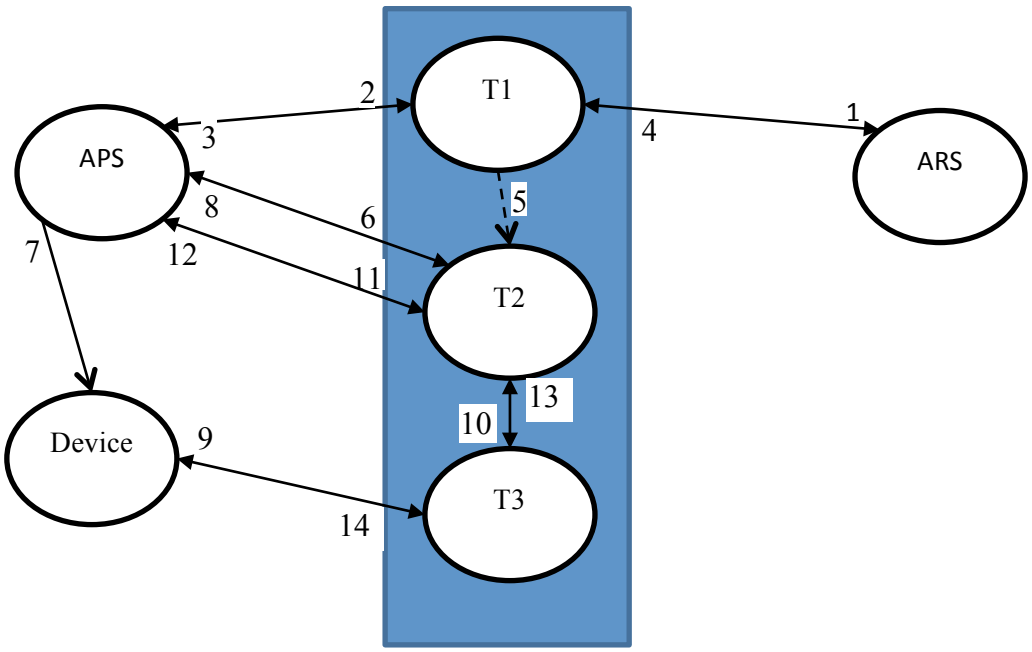
Scenario 2: User has been enrolled

This means that the smartphone device of the user has been registered at the APS service before this authentication request needs to be processed.



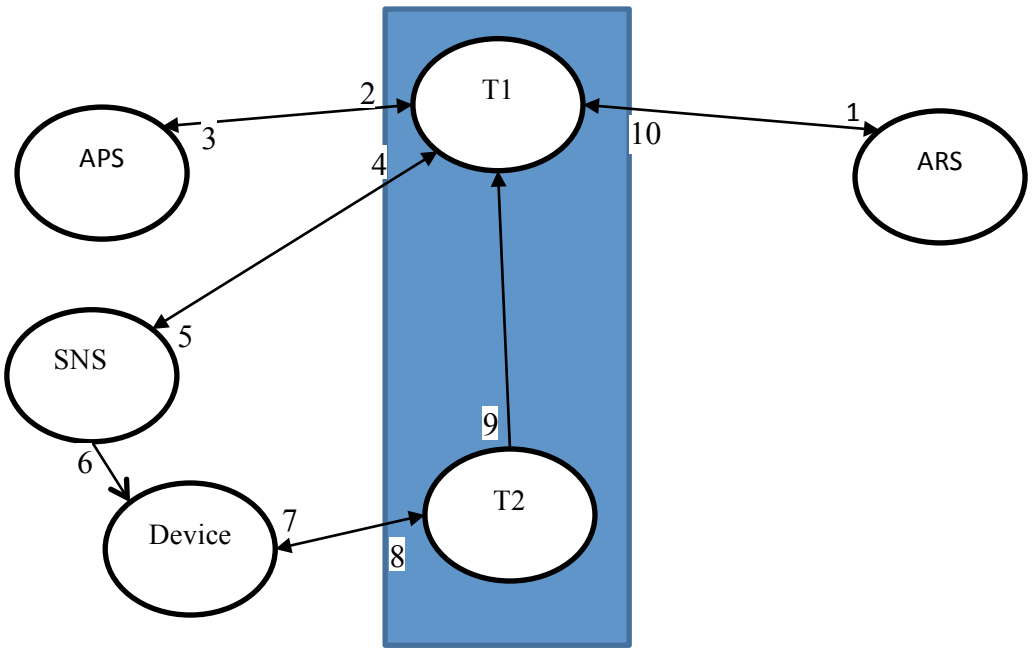
1	Authentication request (second factor) for username/password
2	UserInfo request for username
3	Enrollment status (enrolled) + additional userinfo like device-id of user
4	Register a notification for device at GCM service
5	Device receives the notification from the GCM service
6	Send a confirmation request
7	Returns on authentication request with confirmation response (approved|cancelled)

Internal SAS architecture



The above diagram depicts the needed threads and inter-process-communication for a user that has not been enrolled yet. Authentication will not wait for a successful enrollment. Both process steps will be executed separately.

1) ARS sends 'authenticate' request to SAS (starts T1)
2) T1 sends 'authenticate' request to APS
3) APS replies on 'authenticate' request. If OK, userInfo has been filled
4) T1 replies on 'authenticate' request.
5) If msg 3 equals OK, a thread (T2) will be started to handle the enrollment
6) T2 sends a 'sendmail' request to APS for registered e-mail address
7) APS sends an e-mail message to the passed recipient
8) APS replies on 'sendmail' request to T2. If not OK, thread T2 will be closed
9) App on device sends 'register' request with supplied arguments to SAS (starts T3)
10) T3 forwards 'register' request to T2
11) T2 send 'register' request to APS
12) APS replies on 'register' request
13) T2 replies on 'register' request to T3
14) T3 replies on 'register' request to device



The above diagram depicts the message-flow for a user that has been enrolled yet.

1) ARS sends 'authenticate' request to SAS (starts T1)
2) T1 sends 'authenticate' request to APS
3) APS replies on 'authenticate' request. If OK, userInfo has been filled
4) T1 sends a request for 'notification registration' to the SNS
5) SNS replies on 'notification registration'
6) SNS forward the notification to the device
7) App on device sends 'confirm' request to SAS (starts T2)
8) T2 replies on 'confirm' request
9) T2 forwards 'confirm' request to T1
10) T1 replies on 'authenticate' request from ARS

Json messages to SAS

register

On authentication a register process will be started if the user has not been enrolled yet. In this case an e-mail with a unique code and the api-key will be sent to the device. From this e-mail the app can be downloaded and started. At startup of the app (from the e-mail), the api-key and code will be passed through. The device will register itself at GCM and subsequently submit a "register" request to the SAS service.

```
Request =
{
        "function":"register",          // fixed
        "requestId" : "<request-id>", // id for this request
        "userInfo" :
        {
                "serviceType" : "[GCM|APNS]",          // GCM for Android, APNS for iOS
```

```
                    “serviceNumber” : ”<service-number>”, // unique service/project number as assigned by service
                    “deviceId” : “<device-id>”,     // assigned id by service
                    “notificationId” : “<notification-id>”,     // send to smartphone before
                    “registerCode” : ”<code>”,     // generated unique code send to smartphone device before
          }
}
Reply =
{
          “requestId” : “<request-id>”, // id from corresponding request
          “result” : ”<result-code>”,     // 0 is OK
          “resultText” : ”<result-text>”
}
```

authenticate

```
Request =
{
          “function”:”authenticate”,     // fixed
           “requestId” : “<request-id>”,// id for this request
           “username” : ”<username>”,// username to authenticate
          “password” : “<password>”,  // password (may be empty if first phase authentication has been done)
}
Reply =
{
          “requestId” : “<request-id>”, // id from corresponding request
          “result” : ”<result-code>”,     // 0 is OK
          “resultText” : ”<result-text>”
}
```

notify

```
Request =
{
          “function”:”notify”,// fixed
           “requestId” : “<request-id>”,// id for this request
          “serviceType” : ”[GCM|APNS]”,          // as previously registered
          “serviceNumber” : ”<service-number>”,  // unique service/project number as assigned by service
          “apiKey” : ”<api-key>”,          // unique application key as assigned by service
          “deviceId” : “<device-id>”,     // as previously registered
}
Reply =
{
          “requestId” : “<request-id>”, // id from corresponding request
          “result” : ”<result-code>”,     // 0 is OK
          “resultText” : ”<result-text>”,
          “confirmation” : ”[approved|cancelled]” // as contained by confirmation msg from device
}
```

confirm

```
Request =
{
          “function”:”confirm”,          // fixed
           “requestId” : “<request-id>”,// id for this request
           “deviceId” : “<device-id>”,     // as previously registered
          “notificationId” : “<notification-id>”,     // as previously submitted to service
          “confirmation” : ”[approved|cancelled]” // reflecting user choice
}
Reply =
{
```

```
        "requestId" : "<request-id>", // id from corresponding request
        "result" : "<result-code>",     // 0 is OK
        "resultText" : "<result-text>"
}
```

Json messages to Authentication Server

register

```
Request =
{
        "function":"register",          // fixed
         "requestId" : "<request-id>",// id for this request
         "username" : "<username>",// username to authenticate
        "password" : "<password>",  // password
        "userInfo" :
        {
                "serviceType" : "[GCM|APNS]",          // as received from device registration
                "serviceNumber" : "<service-number>",  // as previously received from authentication reply
                "apiKey" : "<api-key>",         // as previously received from authentication reply
                "deviceId" : "<device-id>"      // as received from device registration
        }
}
Reply =
{
        "requestId" : "<request-id>", // id from corresponding request
        "result" : "<result-code>",     // 0 is OK
        "resultText" : "<result-text>"
}
```

authenticate

```
Request =
{
        "function":"authenticate",     // fixed
         "requestId" : "<request-id>",// id for this request
         "username" : "<username>",// username to authenticate
        "password" : "<password>",  // password
}
Reply =
{
        "requestId" : "<request-id>", // id from corresponding request
        "result" : "<result-code>",     // 0 is OK
        "resultText" : "<result-text>",
        "userInfo:"
        {
                "serviceType" : "[GCM|APNS]",          // as previously registered
                "serviceNumber" : "<service-number>",          // as previously registered
                 "apiKey" : "<api-key>",        // as previously registered
                "deviceId" : "<device-id>",    // as previously registered
                "email" : "<e-mail address>", // e-mail address as registered for user
                "serverURL" : "<server url>", // URL to SAS server to be used by device
                "appURL" : "<app url>",                 // URL to startup App from e-mail
                "storeURL" : "<store url>"      // URL to download App from store
        }
}
```

sendmail

```
Request =
```

```
{
        “function”:”sendmail”,          // fixed
         “requestId” : “<request-id>”,// id for this request
         “username” : ”<username>”,// username to authenticate
        “password” : “<password>”,  // password
        “email” : “<email-address>”,  // email address recipient
        “subject” : “<subject>”,                    // subject of the e-mail
        “text” : “<text>”                   // textual content of the e-mail (html format)
}
Reply =
{
        “requestId” : “<request-id>”, // id from corresponding request
        “result” : ”<result-code>”,     // 0 is OK
        “resultText” : ”<result-text>”
}
```

Google Cloud Messaging

Google Cloud Messaging will be used to submit notifications to Android devices. This GCM service is setup and maintained by Google. It’s free, but a Google Account is needed to make use of it.

The URL https://code.google.com/apis/console is the place to specify your own GCM project. See http:// http://developer.android.com/google/gcm/gs.html for an introduction about how to create a GCM project.

After the GCM project has been created the following attributes are important:

1) Project Number from the overview screen. This number is needed by the Android device to register itself at the service
2) API Key from the API Access screen. This key is needed by the server process to register a notification at the GCM service.
3) Host: ‘android.googleapis.com’, port: ‘443’, URL: ‘/gcm/send’ are the attributes at where the notification request has to be put.