Mobile Computing

Practical Assignment #1 / Native Design and Development

Android applications and services for ticketing/cafeteria

Ticket / Cafeteria Ordering System

1. Scenario

A music and event theater (for instance like http://www.casadamusica.com) wants to provide to its customers an integrated system for an easy acquisition and validation of tickets and cafeteria vouchers to be used on premises. Each customer is previously registered in the system (together with a payment mean, like a credit card). The customer account in the company server maintains a record of the tickets, cafeteria orders, and vouchers gained by each customer.

The theater

A customer Android app allows him to consult the next few shows happening in the theater and to buy tickets for them, using the company remote server and an internet connection. When the event happens, the customer tickets are validated from the customer smartphone in a validation terminal.

Sometimes, vouchers with free gifts and discounts are offered to the customer, to be redeemed in the local cafeteria. To accelerate the cafeteria operation, a customer can compose an order in his app and add some vouchers to be redeemed with it. He can then approach a cafeteria terminal and transmit his order. The vouchers will be taken into consideration and the resulting price will be

automatically paid. In this way the cafeteria attendants only need to take care of supplying the ordered products.

Cafeteria and ticket terminals

When a customer runs the application for the first time (and only once) he should register with the system, supplying some personal data (at least a name and tax number (NIF)) and his credit card data (type, number, validity). This information should be kept in the server data base. Then, customers can perform all the operations allowed by the app.

Ticket and voucher representations

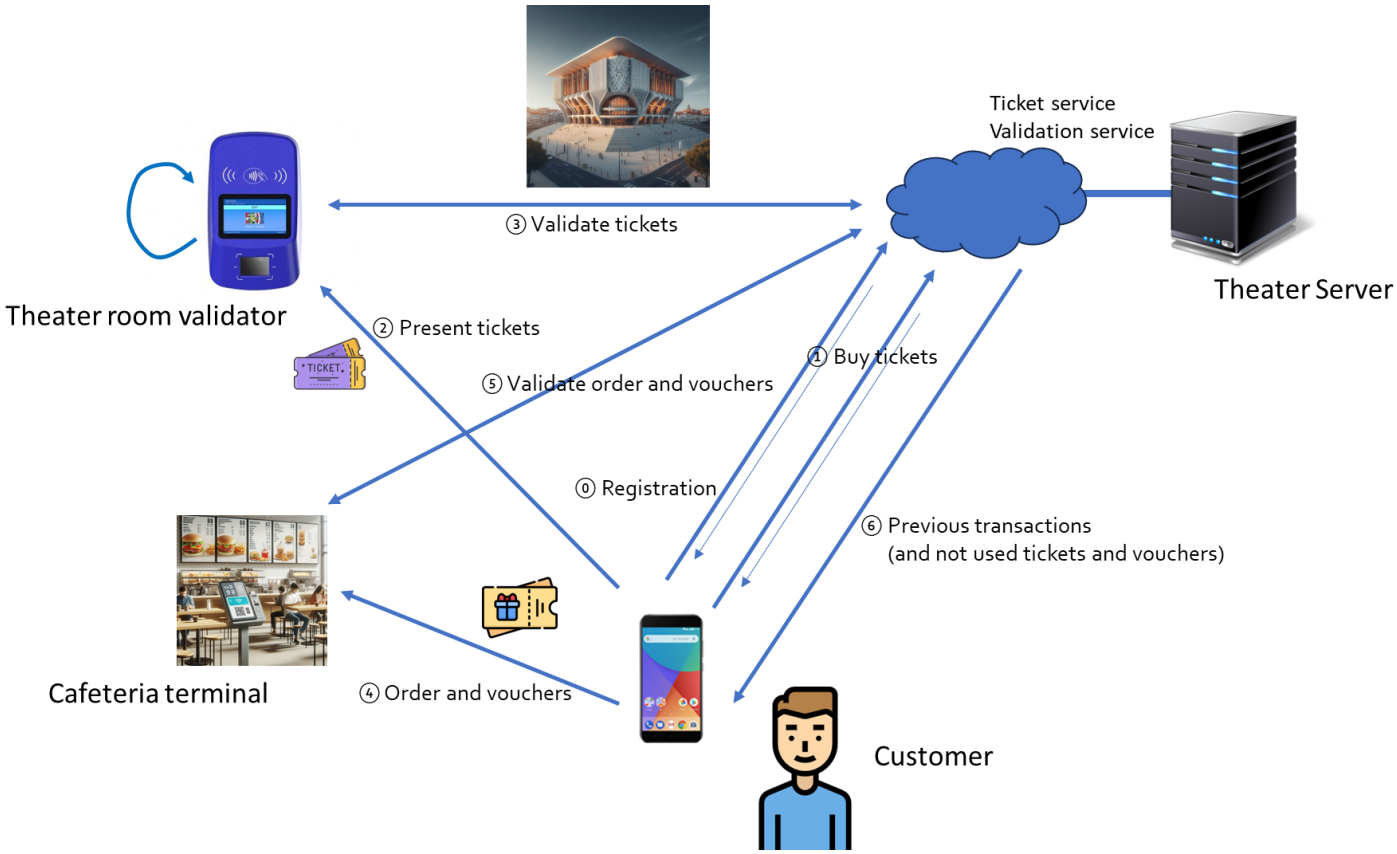
2. System applications

This ticket and payment system is composed by 4 applications, namely:

1. The remote service (a REST service) located on a theater server (it can be divided into several groups of operations: register customers, sell tickets, validate tickets, emit and validate vouchers and consult transactions).
2. The Customer App, allowing him to register himself in the system, consult shows, buy tickets, select and validate tickets at the entrance gates, compose an order to the cafeteria, including selecting and use gift vouchers, and consult past transactions.
3. The ticket validation terminal (at the entrance gates) runs an Android application capable of reading tickets from the customer app and validate them with the server presenting a validation result (indicating which tickets were validated and which were not).
4. The cafeteria order terminal runs also an Android application receiving from the customer app the ordered quantities of available products (for simplification let's assume that the available products are only: **coffee**, **soda drink**, **popcorn** and **sandwich**) and at most **two vouchers** used in the price calculation. After validation

of the vouchers the terminal should show an **order number**, **products ordered**, **vouchers accepted** and final **price to pay**. The customer then collects the products in another place at the cafeteria, when the order number is called.

3. Operations and interactions



Applications interactions

The operations and interactions between these software pieces should be, at least, the following:

0. **Registration** – the first operation the customer app should do is to register the customer in the service. The customer should supply his name and NIF and credit card information (type, number, validity).

   Also, the app should generate a cryptographic RSA key pair (512 bits in length, to keep it simple), and transmit the **public key** to the server (in some format). If the operation succeeds, a **unique** 'user id' (a **uuid** value with 16 bytes) should be generated and returned to the app. This 'user id' and generated **private key** should be stored locally by the app (the values are never shown to the user and the key should be in a protected *keystore*). The server takes note of his information in some database.

1. **Consult the next performances and buy tickets** – the customer can see the next performances to be presented in the theater (say the next four). Some info about each should be presented, with a picture, including the performance dates (assume just one session per day), and prices (that info should come from the server database). An option to buy a given number of tickets (maximum: 4) should be also present.

   A purchase is done transmitting the performance date and the pretended number of tickets, together with the registered 'user id', all signed by the private key of the user. A **local user authentication** should also be performed (preferably biometric, if available).

After showing the intended tickets to buy, its total price, and getting the customer confirmation, they are bought and assumed paid using the customer credit card.

Tickets are now emitted (with also unique IDs per place (another **uuid**)) and transmitted to the customer app. Both the server, and the app, store and take note of the tickets. For each bought ticket a cafeteria voucher is also generated (also with a unique ID (**uuid**)) for a free coffee or popcorn, randomly selected. These vouchers are also transmitted to the app and locally stored. Also, whenever the total paid value to the company by this customer attains a new multiple of say €200.00, another kind of voucher is created, giving a 5% discount in a cafeteria order.

Use the REST protocol for this interaction.

Ticket representations should contain the name of the show, the date of performance, its unique ID and a place in the room (don't care about the capacity of the room). Graphical representations can be simpler, but always show date and seat place. A voucher should have a type code (free coffee or popcorn or the 5% discount voucher) and its unique ID. They can be graphically represented with different colors, for easy identification by the customer.

2. **Present tickets** – in this interaction a set of a maximum of 4 tickets can be transmitted to the validation terminal. The user must choose which tickets, from the available in his device, he wants to validate. The validation app should be configured to accept tickets for a single show (on some date). The transmitted info must contain the **user id**, the **number of tickets**, the **tickets' IDs** and the **event date**. The validation terminal verifies locally the date, sends the tickets information to the server, and should show in a very visible way if all the tickets were validated or not. All the presented tickets in the customer app change now to a '**used**' state, but are not deleted. In this way the customer can see their used tickets (with their seat numbers) but not use them again.

   Note: If you have two Android real devices supporting NFC, that should be the communication channel between customer app and validation app. If you are using a single real device and an emulator, or two devices without NFC, you should use a QR-code to transmit the information (**use a byte representation** to minimize the message size). If you are using two Android emulators the only available channel is a socket and TCP/IP. (See https://developer.android.com/studio/run/emulator-networking). Choose an appropriate solution for this communication channel.

3. **Validate tickets** – in this interaction the **ticket IDs** and the **user id** are sent to the company server for validation (together with the user signature). The server should verify the existence of the tickets, belonging to that user, and that they were not yet used. The server changes internally the tickets state before sending back the validation result. Use the REST protocol for this interaction.

4. **Make a cafeteria order** – From a local menu of products, each customer can compose a cafeteria order. If he owns vouchers he can add a maximum of 2 adequate vouchers to the order. At the cafeteria terminal the app transmits this information (**user id**, **ordered products** and **vouchers**), signed. After validation with the server, the terminal should clearly show an **order number** (integer), the **products in the order**, **accepted vouchers** and the **total price** charged in the credit card (taking into account the accepted vouchers) with the customer NIF. All the vouchers presented by the customer (accepted or not) are deleted from the customer local data. All the info transmitted to the cafeteria terminal is signed with the user private key before sending, and validated in the server.

5. **Validate vouchers and pay an order** – The order containing the **vouchers**, the **ordered products** and the **user id**, with **signature**, are sent to the server. The server verifies the signature and if the vouchers are valid, and if they apply to the ordered products (**only one 5% voucher is accepted in each order**) calculates the final price, using the accepted vouchers. Those are also marked in the server as used. If the total value expended by the customer so far attains a new multiple of €200.00 a new 5% discount voucher is emitted in the server (but not transmitted back to the terminal and customer), and assigned to the customer. The result information (validation and total price, including accepted vouchers) is transmitted back to the order terminal application.

6. **Consult transactions** – At any moment, and using an internet connection, the customer can consult his passed transactions (well, at least a few of them) concerning tickets and cafeteria orders, and have a receipt. The receipt information is only shown on the customer app and should contain at least the customer name, his NIF, number and acquired goods, date, and value paid.

   Whenever this operation is performed **all unused** tickets and vouchers assigned to this customer in the server are transmitted back, and totally **replace** the local info existent (for tickets and vouchers). In this way used tickets are deleted from the customer app, allowing the customer to recover some voucher transmitted by mistake in a previous order, or not yet transmitted, and get rid of used ones, if they are still there.

4. Signatures

All the data originated in the customer app and to be transmitted to the server should have a cryptographic signature with the user private key. To verify a signature, the server should use the customer public key which was transmitted at the registration phase. The key pair is generated by the customer app (once) and the private key is kept stored there. For the signature keys use the "RSA" algorithm and a length of 512 bits (this is only for illustrative purposes; nowadays RSA key sizes should be at least 2048 bits, or even 4096 or more), and for the signature algorithm use "SHA256WithRSA". This produces a signature of 64 bytes, which is the shortest, using standard algorithms with the Android key store.

5. Communications

All the communications in all the operations, except operation 2. and 4., are done using the internet and the http protocol (in a REST service), over Wi-Fi or over the phone operator network. The communication between the customer app and terminals should use NFC or a QR-code and camera.

If you have two physical Android phones supporting NFC use them. If not, use the QR-code technique. The QR-code technique can also be used between a physical phone and an emulator. QR-codes can only represent a small number of bytes, so the information coded should be the minimum possible (use binary values, not strings for numbers; in the Android or Java side you can use the *ByteBuffer* class for easy manipulation of such values in a byte array).

With one real phone and an emulator, the emulator should be the user phone presenting the QR-code and the real phone can capture it with the camera.

If you don't have any Android physical phone available, you can use a TCP/IP connection between two different emulators.

6. Design and development

You should design and implement the set of applications capable of comply with the described functions and demonstrate its use. The applications should have a comfortable and easy to use

interface. You **can** add any functionalities considered convenient, and fill any gaps not detailed in this specification with your own requirements.

You should also write a report, describing the **architecture**, **data schemas** (***server database, a ticket, an order, a voucher***), **included features**, **navigation map**, and performed scenario tests. The applications how to use should also be included in the report, presenting significative screen capture sequences.