# *Progetto di Sistemi Mobili LM*

# *- KeepInQ -*

Relazione di Danilo Dignani

2012

# *Sommario*



---

# *Introduzione*

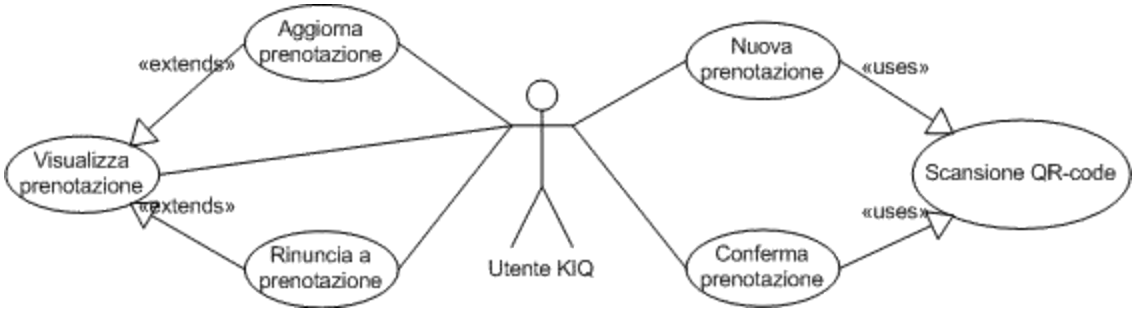
Lo scopo del progetto è quello di realizzare un'applicazione per dispositivi Android che permetta all'utente di prenotarsi in qualsiasi sistema con code (uffici, sportelli, locali …) e di ricevere aggiornamenti per quanto riguarda la situazione della propria posizione, così da potersi liberamente allontanare dalla coda fisica e farvi poi ritorno quando sta per arrivare il proprio turno.

L'utente *KeepInQ* si reca presso la coda in cui intende prenotarsi e procede a creare una nuova prenotazione scansionando un codice QR lì presente, operazione demandata all'applicazione open-source chiamata ZXing (http://code.google.com/p/zxing/) che, in aggiunta, permette all'utente di scaricare l'applicazione stessa nel caso non fosse già presente nel proprio dispositivo. Una volta scansionato il codice, si utilizzano le informazioni contenute per contattare il server e completare l'operazione di prenotazione. Da questo momento l'utente più allontanarsi dalla coda.

Durante questo periodo di assenza, *KeepInQ* contatterà periodicamente il server in cui ci si è prenotati e aggiornerà le informazioni riguardanti quella coda. Il periodo di aggiornamento della prenotazione potrà essere scelto tra alcuni valori standard oppure si potrà lasciar scegliere al dispositivo in base alla quantità di batteria residua (meno frequenti se quasi scarica, dato che non ha senso avere informazioni "fresche" se non si possono utilizzare affatto…). L'aggiornamento automatico della

prenotazione può essere anche disabilitato, lasciando così il compito all'utente di eseguire manualmente il *refresh*, magari regolandosi con il proprio piano tariffario per le connessione *mobile*.

Quando sta per avvicinarsi il turno dell'utente allora si riceverà una notifica sul dispositivo che lo invita a fare ritorno, così da poter confermare la propria prenotazione scansionando nuovamente il codice QR. Se l'utente volesse cancellarsi dalla prenotazione manderà una semplice notifica al server che ne provvederà alla rimozione, e comunque l'utente perderà il posto in coda se non avrà confermato la prenotazione entro un certo time-out dal proprio turno.



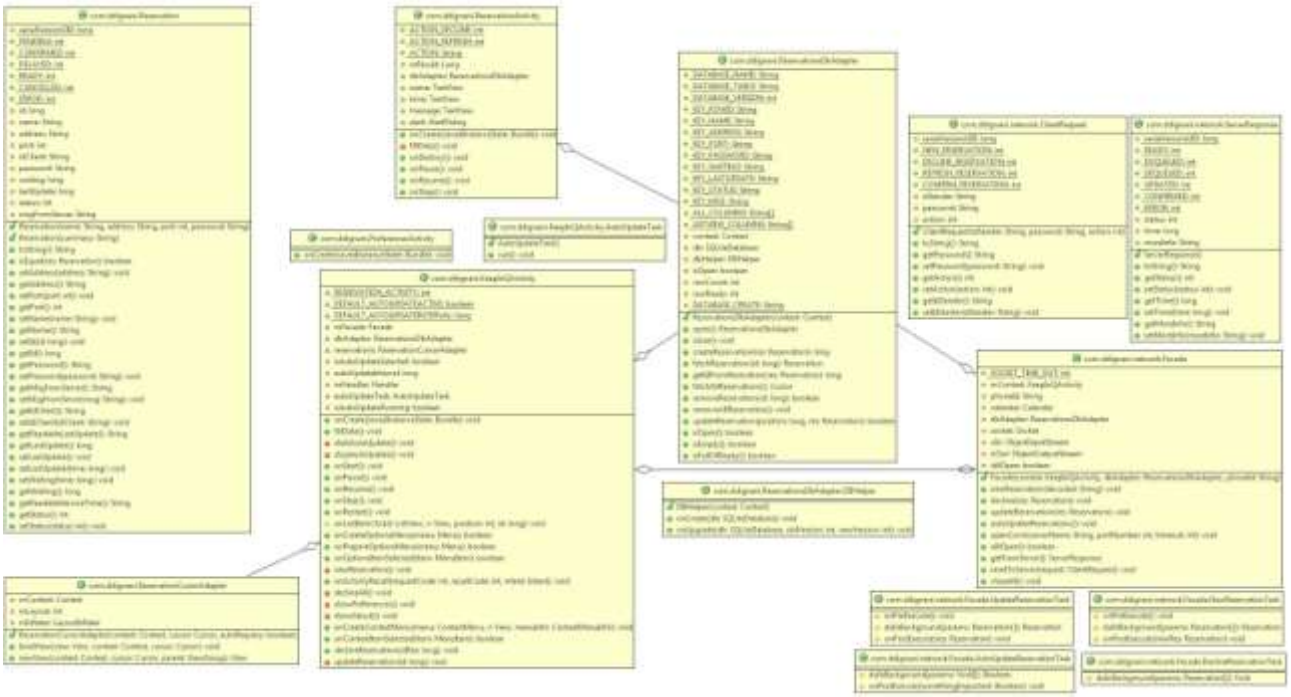

Il fatto di dover scansionare il codice QR dovrebbe servire a scoraggiare comportamenti malevoli da parte dell'utente, accorgimento senza il quale si potrebbero verificare prenotazioni (o conferme di prenotazione) a distanza. Sfortunatamente questo metodo non è sufficiente, dato che basterebbe fare una copia digitale o meno del codice e così utilizzarlo a sproposito. Quindi per "imporre" la presenza fisica dell'utente nei pressi della coda è stato aggiunto il requisito di utilizzare la localizzazione dei dispositivi Android per effettuare un controllo di coordinate GPS tra la posizione attuale del dispositivo e le coordinate incluse nel QR-code.

I requisiti del dispositivo per poter utilizzare l'applicazione KeepInQ sono:

- Piattaforma minima: Android 1.5 (*Cupcake*)
- Piattaforma Target: Android 2.3.3 (*Gingerbread*)
- Disponibilità di una fotocamera e del sensore GPS
- Possibilità di connettersi ad Internet

Per lo sviluppo dell'applicazione è stato utilizzato Eclipse v.3.7 (Indigo) con l'aggiunta del plugin Android Developtment Tool (ADT), Android SDK versione 20 la quale richiede almeno un JDK 6. Lo sviluppo iniziale è stato testato sui dispositivi virtuali (Android Virtual Devices) forniti come *add-on* all'SDK ed in seguito, per testare l'utilizzo della scansione del codice QR e della localizzazione, si è utilizzato in dispositivo mobile LG modello P500 con installata la versione 2.3.3 di Android.

# *Architettura del Client*



La classe *KeepInQActivity* è la *Activity* principale che viene mostrata all'utente. Si presenta come una lista, inizialmente vuota, delle prenotazioni registrate. E' possibile interagire con ognuna di queste tramite un menù contestuale oppure, in caso di più registrazioni, usare dei comandi 'collettivi' che permettono di cancellare o aggiornare manualmente tutte le prenotazioni.

La *Activity* lancia la classe *PreferencesActivity* quando l'utente sceglie dal menu di cambiare le impostazioni e lancia la classe *ReservationActivity* quando l'utente tocca una delle prenotazioni registrate, così da vederne i dettagli. Per la visualizzazione delle prenotazioni si usa un *ReservationCursorAdapter* (estensione di *CursorAdapter*) che, utilizzando le informazioni lette da una query sul dadabase, associa queste informazioni ad una *View* che sarà la singola linea mostrata nella *ListActivity.*

*KeepInQActivity* fa uso della classe *ReservationsDBAdapter* la quale memorizza in maniera persistente le prenotazioni registrate, utilizzando la classe *DbHelper* per l'apertura e la gestione del database.

La classe *Facade* è quella che si occupa delle operazioni 'lente' ovvero ogni comunicazione col server viene eseguita utilizzando gli oggetti *AsincTask* messi a disposizione dal framework, i quali lavorano in un thread differente da quello della *activity* così da non bloccare l'interazione con l'utente. Questi taski inviano un oggetto *ClientRequest* opportunamente configurato e attendono un *ServerResponse* che viene interpretato a seconda dell'azione che si sta eseguendo. Al termine dell'esecuzione gli *AsincTask*

possono eseguire delle operazione nel thread della GUI, in questo caso aggiornano la vista delle prenotazioni.

L'*AutoUpdater* implementa la classe java *Runnable* e, tramite l'oggetto Handler fornito dal framework, si auto-innesca in base alle opzioni specificate nella *PreferencesAcitivty*. Il suo scopo è quello di invocare sulla *Facade* un metodo che aggiorna tutte le prenotazioni e ne memorizza lo stato nel db.

E' un servizio che ovviamente funziona anche quando l'applicazione non è visibile e, se c'è qualcosa di importante che richiede l'attenzione dell'utente (ad esempio si sta avvicinando il turno in una prenotazione o c'è stato un qualche errore), viene presentata una *Notification* nella status bar del dispositivo. Quando l'utente apre la notifica viene rimandato alla *ListActivity* iniziale in cui può controllare lo stato delle sue prenotazioni.

## *L'interazione con l'utente*

*KeepInQActivity* è la *activity* che viene visualizzata quando si accede all'applicazione. E' un'estensione della classe *ListActivity*, ovvero permette di visualizzare al suo interno gli elementi disposti in una lista verticale. In seguito si vedrà che offre anche altre caratteristiche molto comode per poter gestire questa lista, specialmente se si tratta di attingere dati da un database. Il layout di questa activity è specificato nel file xml presente nel percorso */res/layout/main.xml*. Questo file oltre a specificare la porzione di schermo da riservare alla lista e al pulsante principale (valori specificati in DP, ovvero pixel indipendenti dalla densità dello schermo) specifica anche cosa visualizzare in caso di lista vuota, nel nostro caso un semplice testo di notifica che informa di non essere accodati in nessuna coda.

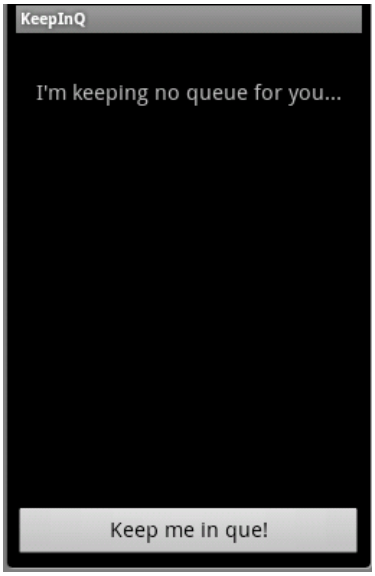
File *main.xml*.

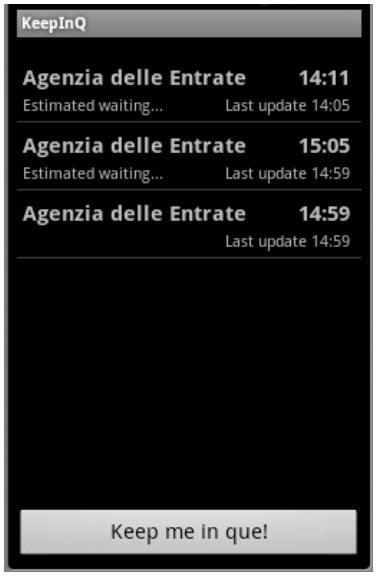
Qui si specifica la porzione di schermo che viene riservata alla lista delle prenotazioni.

Qui invece si specifica cosa mostrare in caso di lista vuota (*empty*), ovvero mostrare la stringa *no_reserv.*

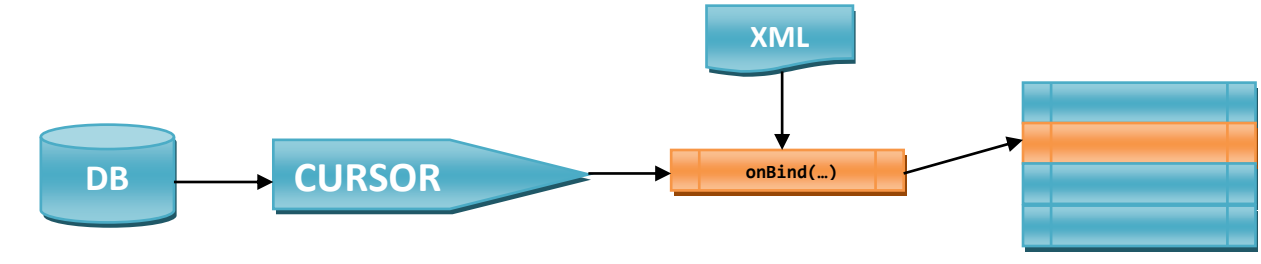
In fondo a tutto viene collocato il pulsante per una nuova prenotazione.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:gravity="bottom|right"
      android:orientation="vertical" >

      <ListView
            android:id="@android:id/list"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:layout_above="@+id/new_reserv"
            android:layout_alignParentTop="true"
            android:paddingBottom="24dp"
            android:paddingTop="16dp" />

      <TextView
            android:id="@android:id/empty"
            android:layout_width="match_parent"
            android:gravity="center_horizontal"
            android:layout_height="wrap_content"
            android:layout_gravity="center_horizontal"
            android:paddingTop="40dp"
            android:text="@string/no_reserv"
            android:textSize="20sp"
            />

      <Button
            android:id="@+id/new_reserv"
            android:layout_width="match_parent"
            android:layout_height="48dp"
            android:layout_alignParentBottom="true"
            android:text="@string/new_reserv"
            android:textSize="20sp" />

</RelativeLayout>
```




Ogni riga della lista fa riferimento ad un altro file xml di layout (*/res/layout/reserv_row.xml*) il quale indica quali campi di testo devono essere visualizzati e la loro posizione. Il contenuto di questi campi di testo viene automaticamente attinto dall'oggetto *ReservationCursorAdapter*, estensione della classe *CursorAdapter*. Ogni volta che all'utente viene presentata questa activity, vengono lette tutte le eventuali prenotazioni presenti nel database tramite una query e vengono memorizzate in un oggetto Cursor. Il ReservationCursorAdapter utilizza questo Cursor come archivio di dati e specifica quale colonna della query deve essere associata ad ogni campo di testo. In questo modo è possibile modificare la quantità di informazioni da visualizzare per ogni riga semplicemente modificandone il layout nel file xml e aggiustando il *binding* di queste informazioni con gli oggetti nel layout.



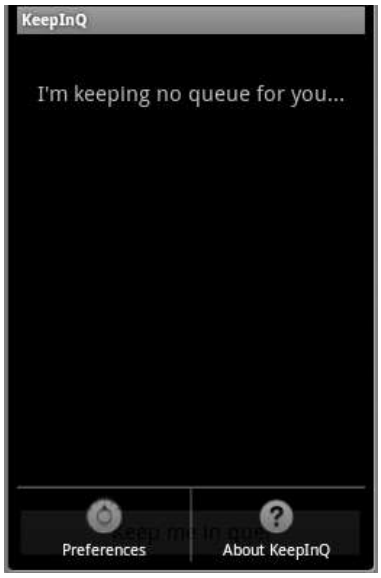

Come detto in precedenza, se nessuna prenotazione è stata registrata allora il cursor sarà vuoto e non avverrà il binding, visualizzando al posto della lista il testo di *nessuna prenotazione*.

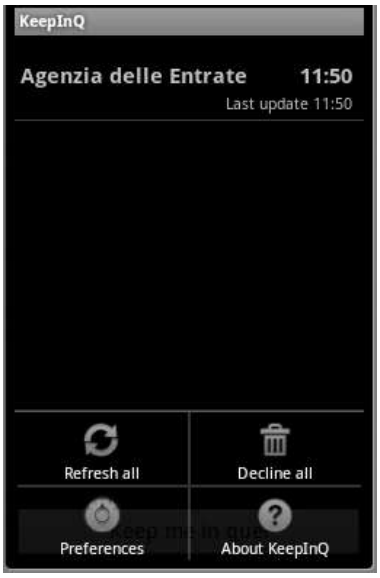
L'utente può interagire con ognuna prenotazione in due modi: può accedere ad una visualizzazione più dettagliata della prenotazione applicando un tocco alla riga interessata oppure con un tocco lungo può far comparire un menù contestuale con le azioni più frequenti da applicare a quella prenotazione. In quest'ultimo caso l'applicazione offrirà la possibilità di Aggiornare manualmente la *reservation*, di

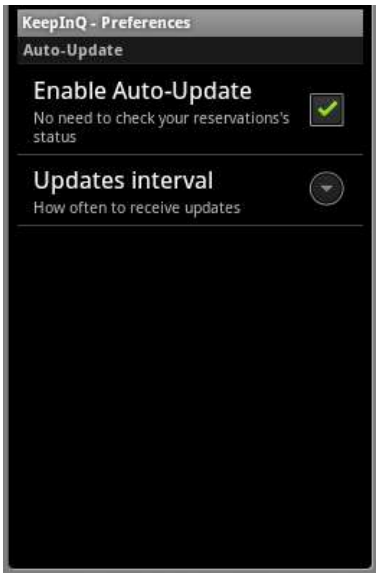
eliminarla oppure di confermarla, sempre che ci siano le condizioni per poterlo fare. Utilizzando il comando `registerForContextMenu` è possibile registrare la lista per ricevere un tocco-lungo e mostrare un menù contestuale e con le funzioni `onCreateContextMenu` e `onContextItemSelected`, offerte anche esse dall'architettura, è possibile specificare rispettivamente come deve essere il menù (utilizzando il file xml di layout presente nel percorso */res/layout/context_menu.xml*) e che comportamento bisogna associare ad ognuna delle voci visualizzate.

Per capire con quale riga della lista (o più praticamente, con quale prenotazione) si vuole interagire, si fa uso di una caratteristica dell'accoppiata *ListActivity-CursorAdapter*: si utilizza un valore di identificazione della riga che risulta essere esattamente il valore della chiave primaria corrispondente a quella prenotazione all'interno del database! Ovvero nel momento in cui viene creata una nuova prenotazione nel database, se si utilizza come chiave primaria la colonna "`_id`", il *CursorAdapter*, e di conseguenza la *ListActivity*, utilizza questo valore per identificare la riga nella lista. In questo modo quando l'utente interagisce con una di esse è possibile utilizzare questo identificatore nella clausola *where* della query sul database così da recuperare tutte le informazioni necessarie riguardanti quella prenotazione, senza dover estrapolare informazioni identificanti dalla riga toccata, informazioni che potrebbero anche non essere visualizzate (ad esempio se per identificare in maniera univoca una prenotazione si usasse l'indirizzo IP del server, questa informazione potrebbe non essere recuperabile "a mano" dalla view della riga dato che potrebbe non essere visibile in quanto poco interessante per l'utente…). Sia quindi che si tratti di tocco o di tocco-lungo viene utilizzato l'identificatore della riga come parametro da passare alla ReservationActivity oppure alle funzioni che descrivono il comportamento delle voci del menù contestuale.


Ultimo aspetto della parte di interazione con l'utente è quello che riguarda la schermata delle preferenze, a cui è possibile accedervi utilizzando il tasto delle opzioni sul dispositivo. La *PreferencesActivity* così avviata permette all'utente di scegliere se fare uso o meno della funzionalità di auto-aggiornamento delle prenotazioni e in tal caso può specificare con che frequenza tali aggiornamenti debbano avvenire. Questa activity è un'estensione della classe *PreferenceActivity* dell'architettura Android ed ha proprio lo scopo di memorizzare in maniera persistente le impostazioni scelte dall'utente, anche se l'applicazione dovesse terminare o venisse distrutta dal sistema. Queste impostazioni saranno poi recuperate dalla KeepInQActivity in fase di creazione tramite un *PreferenceManager* il quale permette anche di associare dei valori di default nel caso di primissimo avvio dell'applicazione.



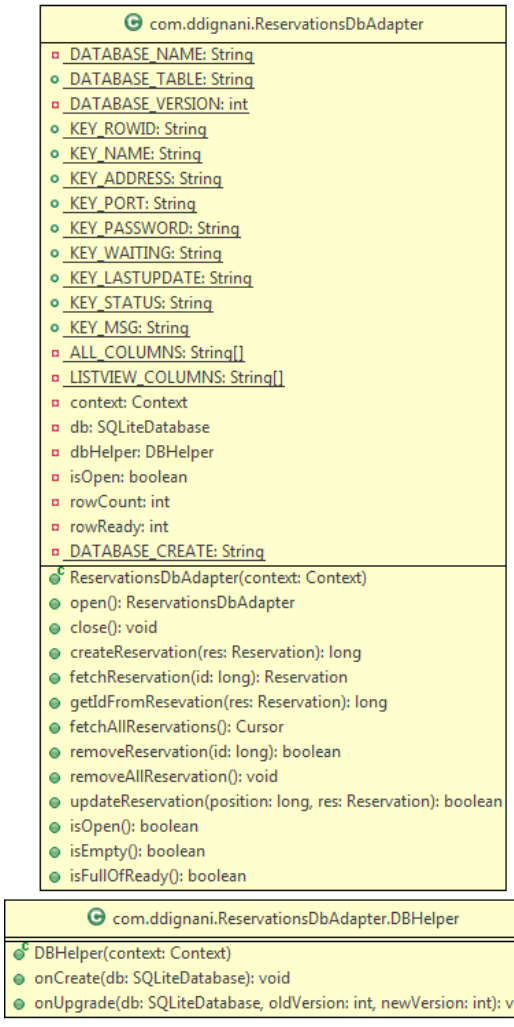

Nel caso sia presente almeno una prenotazione registrata, il menu delle opzioni mostra anche due comandi aggiuntivi che permettono delle operazioni collettive che riguardano l'aggiornamento manuale o l'eliminazione di tutte le *reservations* con un solo tocco. Questa duplice forma del menù è possibile ottenerla tramite la funzione `onPrepareOptionsMenu` la quale viene invocata un attimo prima ogniqualvolta il menù viene mostrato. Se il database delle prenotazioni risulta essere vuoto allora non viene data visibilità al gruppo di icone delle operazioni collettive così da non appesantire inutilmente l'interfaccia con dei pulsanti inutili.

## *Persistenza dei dati*

Come accennato precedentemente, tutte le informazioni riguardanti le code a cui ci si prenota devono essere memorizzate in maniera persistente nel database SQLite che il dispositivo Android mette a disposizione, questo perché basterebbe un semplice cambio di applicazione per perdere tutte le prenotazioni! In aggiunta il sistema stesso, quando ne ha bisogno, può liberare della memoria terminando applicazioni *in background*, ovvero non visualizzate in quel momento dall'utente, e addirittura, in situazioni estreme, potrebbe anche distruggere l'applicazione corrente per poter garantire funzionalità vitali per il dispositivo (che malgrado tutto continua ad essere un telefonino). Tenere informazioni importanti per il funzionamento dell'app in memoria è quindi impensabile.

L'utilizzo del database interno risulta essere un'operazione veloce, affidabile ed efficiente, e tutte le operazioni che prevedono un uso di query in linguaggio pseudo-sql sono state affidate all'oggetto *ReservationsDbAdapter*. Questa classe a sua volta fa uso di un oggetto *DBH*elper che estende *SQLiteOpenHelper* offerto dall'architettura per le operazioni di apertura del database o di creazione nel caso non dovesse esistere. *ReservationsDbAdapter* mette a disposizione principalmente 6 funzioni:

- una per inserire nel database una nuova *reservation*,
- due per restituire tutte o una singola *reservation* (nel primo caso viene restituito un *Cursor* che verrà utilizzato per popolare la *KeepInQActivity*),
- altre due per rimuovere tutto o una singola *reservation* (nel primo caso è più performante fare un *drop* dell'intera tabella e crearne una nuova, piuttosto che cancellare tutte le righe una ad una),
- una per aggiornare una *reservation*.




Il termine *Reservation* si riferisce all'omonima classe di comodo che permette di memorizzare in via temporanea tutte le informazioni che riguardano una prenotazione quali nome, indirizzo IP e porta del server, eventuale password di accesso, tempo di attesa, ultimo aggiornamento, stato della prenotazione ed un eventuale messaggio ricevuto dal server (utilizzabile anche per comunicazioni di 'emergenza' come pesanti rallentamenti, o peggio blocchi del sistema, dovuti a problemi tecnici…). Anche se l'oggetto in sé non vi viene serializzato il suo utilizzo risulta comodo quando si tratta di estrarre ed inserire prenotazioni nel database oppure per una qualche manipolazione delle informazioni

contenute. Ad esempio quando si tratta di aggiornare una prenotazione si invoca `fetchReservation(id)` sul *DbAdapter* utilizzando il valore di `id` ottenuto dalla *ListActivity*, si ottiene un oggetto di tipo *Reservation*, se ne modificano i campi da aggiornare e si utilizzano gli stessi `id` e `reservation` per invocare la `updateReservation(id, reservation)`.

Le query vengono invocate su un oggetto `db` di tipo *SQLiteDatabase* che offre funzioni per gestire un database SQLite quali l'inserimento, la modifica o l'esecuzione di comandi sql. Riprendendo l'esempio di sopra si può capirne il funzionamento guardando il codice delle due funzioni citate:

```
public Reservation fetchReservation(long id){
        open();
        Cursor cursor = db.query(DATABASE_TABLE, ALL_COLUMNS, KEY_ROWID +"="+ id, null, null, null, null);
        Reservation res = null;
        if(cursor.moveToFirst()){ //se cursor è vuoto ritorna False
                res = new Reservation(cursor.getString(cursor.getColumnIndex(KEY_NAME)),
                        cursor.getString(cursor.getColumnIndex(KEY_ADDRESS)),
                        cursor.getInt(cursor.getColumnIndex(KEY_PORT)),
                        cursor.getString(cursor.getColumnIndex(KEY_PASSWORD)));
                        res.setStatus(cursor.getInt(cursor.getColumnIndex(KEY_STATUS)));
                        res.setWaiting(cursor.getLong(cursor.getColumnIndex(KEY_WAITING)));
                        res.setLastUpdate(cursor.getLong(cursor.getColumnIndex(KEY_LASTUPDATE)));
                        res.setMsgFromServer(cursor.getString(cursor.getColumnIndex(KEY_MSG)));
        }
        close();
        return res;
}
```

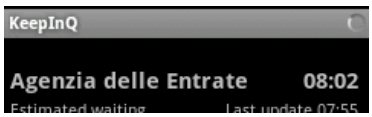
```
public boolean updateReservation(long id, Reservation res){
        open();
        ContentValues args = new ContentValues();
        args.put(KEY_STATUS, res.getStatus());
        args.put(KEY_WAITING, res.getWaiting());
        args.put(KEY_LASTUPDATE, res.getLastUpdate());
        args.put(KEY_MSG, res.getMsgFromServer());
        String where = KEY_ROWID + "=" + id;
        if(res.getStatus() == Reservation.READY)
                rowReady ++;
        int i = db.update(DATABASE_TABLE, args, where, null);
        close();
        return i > 0;
}
```

*ReservationsDbAdapter* tiene conto anche del numero delle prenotazioni registrate e anche di quelle che risultano già in stato di *READY*. Questo accorgimento è stato preso per evitare che il sistema di aggiornamento automatico delle prenotazioni rimanga attivo anche quando tutte le prenotazioni non necessitano più di un aggiornamento perché l'utente è stato già informato che il suo turno è arrivato o perché sono già scadute (cioè l'utente non ha fatto ritorno alla coda). In questo modo, sapendo che tutte le prenotazioni sono Ready, è possibile disabilitare l'autoupdater (vedere più sotto) risparmiando così risorse di sistema. Se anche una sola delle varie *reservations* non è ancora in stato Ready allora l'autoupdater rimarrà attivo e controllerà solo quella prenotazione.

## *Comunicazioni con il server ed esecuzioni in background*

Stabilire una connessione con un server remoto è un'operazione indubbiamente lunga. Utilizzando poi connessioni di tipo *mobile* il tempo di attesa potrebbe protrarsi anche per qualche decina di secondi e tenere bloccata un'applicazione per un tempo così lungo non è affatto una buona pratica. Android in particolar modo ha una politica piuttosto ferrea a riguardo: se il *thread* dove viene gestita la GUI non risponde per più di 5 secondi viene mostrato all'utente un messaggio ANR (Application Not Responding) chiedendo se si intende terminare l'applicazione o aspettare ancora. Vista la natura dell'applicazione si è reso necessario dividere la parte che si occupa dell'interazione con l'utente da quella che si occupa dell'interazione con il server. L'architettura Android mette a disposizione alcune soluzioni a questo problema, ognuna con i propri pregi e difetti. La scelta per affrontare questa questione è ricaduta sull'utilizzo degli *AsyncTask*.

Gli AsyncTask sono degli oggetti che permettono eseguire un operazione onerosa (come le operazioni di networking) in un thread che non sia quello principale, ovvero l'unico e solo dove dovrebbero avvenire le interazioni con la GUI e quindi con l'utente. Una volta terminata l'esecuzione onerosa è possibile far eseguire all'AsyncTask  delle operazioni nel thread principale così da poter interagire con l'utente, magari notificando l'esito di azione o modificando la vista che sta visualizzando. In aggiunta è possibile eseguire operazioni sul thread GUI prima, durante e dopo l'esecuzione! Un AsyncTask infatti mette a disposizione 4 funzioni che rappresentano 4 step nell'esecuzione del thread oneroso, ovvero quello che avviene prima, l'esecuzione dell'operazione lenta, cosa avviene dopo e, se necessario, come mostrare i progressi dell'azione.

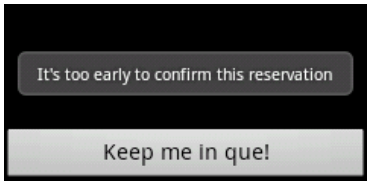



Nel caso di KeepInQ, ad esempio, quando inizia una operazione lenta (`onPreExecute()`) viene mostrato sulla barra del titolo dell'app un 'circoletto' rotante che simboleggia un'azione che sta avvenendo in background (`doInBackground()`) (il termine preciso è *Indeterminate progressbar*, ad indicare che non ha una durata precisa) e al termine dell'operazione (`onPostExecute()`) la *progressbar* viene nascosta all'utente e viene aggiornata la lista delle prenotazioni. In questo modo l'utente è consapevole del fatto che alla azione di toccare un tasto è corrisposta una effettiva risposta da parte dell'app, senza oscurare la GUI con messaggi di attesa.

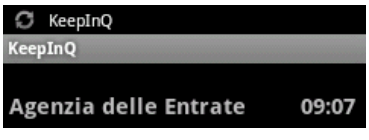
Tutte le operazioni di comunicazione col server sono state implementate nella classe *Facade*, prendendo spunto per il nome dal pattern di sviluppo e anche la sua funzione è simile: nascondere la complessità dei meccanismi dietro a delle semplici funzioni che verranno invocate dalla *KeepInQActivity*. Al suo interno infatti prendono posto sia gli *AsyncTask* per connessione al server sia le operazioni sul database.. La comunicazione avviene tramite lo scambio di due oggetti chiamati `ClientRequest` e

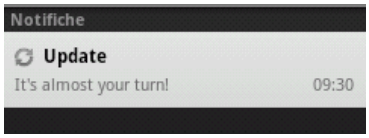
`ServerResponse` i quali rappresentano appunto la richiesta da parte del client da inviare al server e la corrispettiva risposta ricevuta.


Ogni volta che viene richiesta un'operazione ti interazione con il server, l'oggetto *Facade* si occupa di stabilire la connessione (socket TCP) utilizzando le informazioni lette in precedenza dal codice QR o ricavate dal database, crea ed invia un oggetto *ClientRequest* in cui memorizza il codice IMEI del dispositivo come identificativo unico dell'utente, l'eventuale password e il tipo di azione da eseguire. Il server controllerà se l'operazione da eseguire è logicamente fattibile (ad esempio non si può aggiungere una doppia prenotazione per quell'utente oppure non si può confermare una prenotazione prima di un certo periodo all'arrivo del turno) e, una volta eseguita, risponderà con una *ServerResponse* in cui specificherà lo stato dell'azione/prenotazione, il tempo di attesa residuo in coda e un facoltativo messaggio di comunicazione per l'utente. La *Facade* in base alla risposta ricevuta dal server dovrà informare l'utente sull'esito dell'azione eseguita e quindi aggiornerà la lista delle prenotazioni in caso di esito positivo oppure mostrerà un *Toast* in cui si spiega che cosa sia andato storto, sia a livello locale che remoto (utilizzando il messaggio di errore inviato dal server).




Quando è attiva l'opzione dell'aggiornamento automatico delle prenotazioni, la *Facade* deve utilizzare un metodo alternativo per comunicare con l'utente. Infatti molto probabilmente una volta che l'utente si è accodato in una coda inizierà ad utilizzare il dispositivo per altri scopi o comunque non rimarrà tutto il tempo a controllare la schermata di KeepInQ in attesa di aggiornamenti. Per questo quando si verifica un evento che richiede l'attenzione dell'utente la *Facade* lancerà una notifica nella barra delle notifiche, ad esempio quando si sta avvicinando il turno in una coda o magari c'è stato un errore qualsiasi. Una volta selezionata dal menu *drop-down* l'utente viene rimandato alla schermata principale di KeepInQ così da poter controllare quale prenotazione richiede la sua attenzione.



Nel seguente stralcio di codice è mostrato l'esempio dell'esecuzione di un aggiornamento manuale:

```
public void updateReservation(long id) {
        new UpdateReservationTask().execute(id);
}


private class UpdateReservationTask extends AsyncTask <Long, Void, Reservation>{

protected void onPreExecute(){
        // Mostra la progress-bar nel titolo
        mContext.setProgressBarIndeterminateVisibility(true);
}

@Override
protected Reservation doInBackground(Long... params) {
        Reservation res = dbAdapter.fetchReservation(params[0]);
        if(res != null){
                //Stabilisce la connessione
                openConn(res.getAddress(), res.getPort(), SOCKET_TIME_OUT);
                if(isItOpen()){
                        ClientRequest request = new ClientRequest(phoneId, res.getPassword(),
                                                                        ClientRequest.REFRESH_RESERVATION);
                        ServerResponse response = new ServerResponse();
                        try{
                                sendToServer(request);
                                response = getFromServer();
                        }catch (Exception e) {
                                response.setStatus(ServerResponse.ERROR);
                        }

                        if(response.getStatus() != ServerResponse.ERROR){
                                res.setWaiting(response.getTime() - (System.currentTimeMillis() -
                                                                                res.getLastUpdate()));
                                res.setStatus(Reservation.PENDING);
                                res.setMsgFromServer(response.getMoreInfo());
                                // Il server dice che bisogna prepararsi
                                if(response.getStatus() == ServerResponse.READY){
                                        res.setStatus(Reservation.READY);
                                        res.setMsgFromServer(response.getMoreInfo());
                                }
                        }else{
                                res.setStatus(Reservation.ERROR);
                                res.setMsgFromServer(response.getMoreInfo());
                        }
                        res.setLastUpdate();
                        closeAll();
                }else{
                        res.setStatus(Reservation.ERROR);
                        res.setMsgFromServer("No internet connection available");
                }
        }
        return res;
}

protected void onPostExecute(Reservation res){
        // Nasconde la progress-bar nel titolo
        mContext.setProgressBarIndeterminateVisibility(false);

        if(res.getStatus() != Reservation.ERROR){
                long id = dbAdapter.getIdFromResevation(res);
                dbAdapter.updateReservation(id, res);
                mContext.fillData();
        }else
                //Toast per dire che c'è stato qualche errore a contattare il server
                Toast.makeText(mContext, res.getMsgFromServer(), Toast.LENGTH_LONG).show();
}
}
```

Essendo un aggiornamento manuale l'utente sta già visualizzando la schermata delle prenotazioni, quindi non occorre lanciare una notifica quando necessario, ma sarà sufficiente aggiornare la lista o, al più, mostrare un *Toast* in caso di errori.

## *L'autoupdater*

Se la relativa opzione è abilitata e se ci sono delle prenotazioni in sospeso, l'applicazione KeepInQ si occupa di controllarne automaticamente lo stato ad una frequenza scelta dall'utente dalla schermata delle preferenze. Il numero di aggiornamenti da effettuare è un aspetto che ha un impatto considerevole sul consumo della batteria dato che per ogni prenotazione è necessario stabilire una connessione ed inviare e ricevere dei dati, a cui si può aggiungere anche l'eventuale necessità di dover risvegliare il componente radio del dispositivo che magari era in uno stato "di riposo" perché inutilizzato. Per questo motivo è stata data la possibilità di scegliere la frequenza degli aggiornamenti così da scegliere il compromesso più adeguato tra freschezza delle informazioni e consumi energetici. Se invece l'utente non volesse interessarsi della batteria utilizzata può scegliere che sia il dispositivo ad adeguare il numero di aggiornamenti in base alla carica residua, diminuendone la frequenza non appena questa si riduca sotto una certa soglia. Per fare ciò si è specificato nel file *manifest.xml* dell'applicazione che si vuole ricevere una notifica sia quando la batteria scende sotto un livello di carica sia quando si trova in uno stato di normalità.

```
<receiver android:name=".BatteryStatusReceiver">
    <intent-filter>
        <action android:name="android.intent.action.ACTION_BATTERY_LOW" />
        <action android:name="android.intent.action.ACTION_BATTERY_OKAY" />
    </intent-filter>
</receiver>
```

Nella classe *KeepInQActivity* è stato creato un *BroadCastReceiver* chiamato *BatteryStatusReciver* il quale, quando il sistema invia il broadcast sullo stato della batteria, riceve questa notifica e cambia il parametro di frequenza di aggiornamenti.

```
public class BatteryStatusReceiver extends BroadcastReceiver{
        @Override
        public void onReceive(Context context, Intent intent) {
                if(intent.getAction() == "ACTION_BATTERY_LOW"){
                        autoUpdateIntervalOnBattery = AUTOUPDATEINTERVAL_ON_LOW_BAT;
                }
                if(intent.getAction() == "ACTION_BATTERY_OKAY"){
                        autoUpdateIntervalOnBattery = AUTOUPDATEINTERVAL_ON_OKAY_BAT;
                }
        }
}
```

La variabile `autoUpdateIntervalOnBattery` viene letta dal sistema solo quando l'utente sceglie di eseguire gli aggiornamenti basandosi sulla batteria, infatti ogni volta che viene invocato il metodo `onStart()` della classe *KeepInQActivity* si effettua questo controllo:

```
isAutoUpdateSelected = preferences.getBoolean("autoUpdateCheckBox", DEFAULT_AUTOUPDATEACTIVE);
if(isAutoUpdateSelected){
        autoUpdateInterval = Long.parseLong(preferences.getString("updates_interval", "-1"));
        if(autoUpdateInterval == -1){
                autoUpdateInterval = autoUpdateIntervalOnBattery;
        }
}
```

L'avvio dell'autoupdater viene affidato alla funzione `fillData()` presente in *KeepInQActivity* la quale, come suggerisce il nome, si occupa di popolare la lista delle prenotazioni registrate. Questo metodo viene invocato ogni volta che l'utente ritorna alla schermata principale (sia che provenga dalle altre activity di KeepInQ, sia che ritorni dopo aver utilizzato il dispositivo per altri scopi) e anche ogni volta che l'oggetto *Facade* termina una comunicazione con il server, così da poter mostrare subito il risultato di quella azione. Proprio per quest'ultimo caso è bene che si controlli lo stato dell'autoupdater e poter così deciderne la sua effettiva utilità o meno: se non ci sono *Reservation* (per esempio l'utente le ha annullate tutte?) o se sono già tutte in stato *Ready* allora non occorre eseguire aggiornamenti automatici, come pure se l'utente attiva o meno l'autoupdater dalla schermata delle preferenze è bene che si applichi questa sua volontà appena possibile.

```
if(isAutoUpdateRunning) //Sta andando
        if(!isAutoUpdateSelected) // Non dovrebbe andare
                stopAutoUpdater();
        else
                // Dovrebbe andare ma se il db è vuoto o ci sono solo Reservation già Ready allora no
                if(dbAdapter.isEmpty() || dbAdapter.isFullOfReady())
                        stopAutoUpdater();
else    //Non sta andando
        if(isAutoUpdateSelected) //Dovrebbe andare
                if(!dbAdapter.isEmpty() && !dbAdapter.isFullOfReady()) // Occorre davvero che vada?
                        startAutoUpdater();
```

Le funzioni `startAutoUpdater()` e `stopAutoUpdater()` permettono rispettivamente di aggiungere e rimuovere ad un oggetto *Handler* l'esecuzione del task dell'auto-updater. Una volta aggiunto il task questo verrà eseguito dall'*Handler* con un preciso ritardo e si innescherà da solo per venir eseguito nuovamente con lo stesso ritardo (sempre che nel frattempo non sia cambiata la frequenza degli aggiornamenti). L'esecuzione del task prevede che venga invocato il metodo `autoUpdateReservations()` della Facade il quale si occuperà di contattare i server delle prenotazioni registrate e di aggiornarne lo stato. Come detto in precedenza, qualora dovesse verificarsi che sia richiesta l'attenzione dell'utente, verrà inviata una notifica al sistema che la visualizzerà nella barra delle notifiche e rimanderà l'utente alla schermata principale delle prenotazioni.

```
private void startAutoUpdater(){
    mHandler.postDelayed(autoUpdateTask, autoUpdateInterval);
        isAutoUpdateRunning = true;
}

private void stopAutoUpdater(){
        mHandler.removeCallbacks(autoUpdateTask);
        isAutoUpdateRunning = false;
}


public class AutoUpdateTask implements Runnable {
        @Override
        public void run() {
                mFacade.autoUpdateReservations();
                // riavvio l'updater
                mHandler.postDelayed(this, autoUpdateInterval);
        }
}
```

```
private class AutoUpdateReservationTask extends AsyncTask <Void, Void, Integer>{
        @Override
        protected Integer doInBackground(Void... params) {
        final int NOTHING = 0;
        final int ALMOST_READY = 1;
        final int ERRORS = -1;

        int somethingImportant = NOTHING;

        Cursor cursor = dbAdapter.fetchAllReservations();
        if(!cursor.moveToFirst()){
                //il database è vuoto
                cancel(true);
        }
        do{
                long id = cursor.getLong(cursor.getColumnIndex(ReservationsDbAdapter.KEY_ROWID));
                Reservation res = dbAdapter.fetchReservation(id);
                if(res.getStatus() != Reservation.READY){ // Se è già Ready non la controllo ancora
                        //Stabilisce la connessione
                        openConn(res.getAddress(), res.getPort(), SOCKET_TIME_OUT);
                        if(isItOpen()){
                                ClientRequest request = new ClientRequest(phoneId, res.getPassword(),
                                                                        ClientRequest.REFRESH_RESERVATION);
                                ServerResponse response = new ServerResponse();
                                try{
                                        sendToServer(request);
                                        response = getFromServer();
                                }catch (Exception e) {
                                response.setStatus(ServerResponse.ERROR);
                        }

                        if(response.getStatus() != ServerResponse.ERROR){
                                res.setWaiting(response.getTime() - (System.currentTimeMillis() -
                                                                        res.getLastUpdate()));
                                res.setStatus(Reservation.PENDING);
                                res.setMsgFromServer(response.getMoreInfo());
                                // Il server dice di prepararsi o manca meno del prossimo autoUpdate
                                if(response.getStatus() == ServerResponse.READY) ||
                                        response.getTime() <= mContext.getNextUpdateInterval()){

                                        res.setStatus(Reservation.READY);
                                        somethingImportant = ALMOST_READY;
                                }
                        }else{
                                res.setStatus(Reservation.ERROR);
                                res.setMsgFromServer(response.getMoreInfo());
                                somethingImportant = ERRORS;
                        }
                        res.setLastUpdate();
                        closeAll();
                }

                //Aggiorno il db
                dbAdapter.updateReservation(id, res);
                }
        }while(cursor.moveToNext());

return somethingImportant;
}
```

Le funzioni `sendToServer()` e `getFromServer()` usano degli ObjectOutpup/InputStream per inviare e ricevere gli oggetti serializzabili *ClientRequest* e *ServerResponse*. Nel caso vada tutto bene con la trasmissione di questi dati allora si riceve dal server il tempo di attesa al quale viene sottratto il tempo che si è già atteso, ovvero la quantità di tempo tra l'istante attuale e quello dell'ultimo aggiornamento.

L'autoupdater deve tenere conto di un aspetto importante: il momento in cui l'utente deve cominciare a fare ritorno alla coda per confermare la prenotazione. Per questo motivo quando riceve dal

server il tempo di attesa rimanente viene effettuato un controllo e, se il server ha risposto che la prenotazione è *Ready* oppure se l'attesa è inferiore al tempo che trascorrerà fino al prossimo aggiornamento automatico, allora quella prenotazione viene considerata come *Ready*. Facendo altrimenti potrebbe capitare la situazione in cui un aggiornamento automatico cada oltre la scadenza di validità della prenotazione e l'utente si ritroverebbe con poco tempo per ritornare alla coda o addirittura con una prenotazione già scaduta.

Nel caso siano presenti più prenotazioni e si presenti la situazione che contemporaneamente ce ne siano una in stato di *Ready* e un'altra invece in stato *Errors* allora il valore della variabile `somethingImportant` assumerebbe il valore `ALMOST_READY` o `ERRORS` a seconda di quale delle due si presenti per ultima. In realtà questo non è un vero e proprio problema dato che in entrambe i casi si richiede l'attenzione dell'utente e quindi, al momento di controllare la lista delle prenotazioni, potrà farsi un'idea più precisa della situazione.

Le notifiche vengono lanciate utilizzando la funzione `sendNotification` :

```
private void sendNotification(CharSequence ticker, CharSequence title,  CharSequence message) {
        // notifica
        int icon = R.drawable.ic_menu_refresh;

        Intent notificationIntent = new Intent(mContext, KeepInQActivity.class);
        notificationIntent.setAction(Intent.ACTION_MAIN);
        notificationIntent.addCategory(Intent.CATEGORY_LAUNCHER);
        PendingIntent pendingIntent = PendingIntent.getActivity(mContext, 0, notificationIntent, 0);

        Notification notification = new Notification(icon, ticker, System.currentTimeMillis());
        notification.setLatestEventInfo(mContext, title, message, pendingIntent);
        notification.flags |= Notification.FLAG_AUTO_CANCEL;
        notification.defaults |= Notification.DEFAULT_SOUND;


        // ;-)
        long DOT = 100, DASH = DOT * 3, GAP = DOT, LETTERGAP= DOT* 3;
        long vibe [] = new long [] {0, DASH, GAP, DOT, GAP, DASH, LETTERGAP, DOT, GAP, DOT, LETTERGAP, DASH,
                                    GAP, DASH, GAP, DOT, GAP, DASH};
        notification.vibrate = vibe;
        NotificationManager nm = (NotificationManager)mContext.getSystemService("notification");
        nm.notify(0, notification);
}
```

Per dare un tocco di personalizzazione alla notifica, le è stata associata una vibrazione che riproduce in codice Morse le lettere KIQ, ispirandosi alla suoneria Nokia per la ricezione degli SMS…

## *Scansione del codice QR*

Come accennato all'inizio, la prenotazione in una coda e la sua successiva conferma deve avvenire scansionando il codice QR presente in loco. Per la scansione si è scelto di utilizzare il progetto open-source ZXing (http://code.google.com/p/zxing/) la quale è un ottima applicazione per la scansione di qualsiasi tipo di codice a barre, anche quelli 2D come i codici QR. Vengono messe a disposizione dello sviluppatore dei semplici comandi per avviare la scansione e, nel caso l'applicazione non si installata nel dispositivo, viene chiesto all'utente se intende scaricarla.

Per avviare la scansione basta invocare l'*intent* in questo modo:

```
IntentIntegrator integrator = new IntentIntegrator(this);
integrator.initiateScan(IntentIntegrator.QR_CODE_TYPES);
```

e il risultato sarà recuperato all'interno della funzione `onActivityResult` che viene invocata ogni volta che si ritorna alla *activity* dopo aver gestito un *intent*:

```
public void onActivityResult(int requestCode, int resultCode, Intent intent) {
        […]
        IntentResult scanResult = IntentIntegrator.parseActivityResult(requestCode, resultCode, intent);
        if (scanResult != null) {
                String contents = scanResult.getContents();
        […]
        if (contents != null) {
                String serverInfo = intent.getStringExtra("SCAN_RESULT");
                mFacade.newReservation(serverInfo);
        }
```

Il testo codificato, per essere correttamente interpretato dall'applicazione, dovrà rispettare la sintassi *<Nome della prenotazione>*|*<indirizzo IP del server>*|*<porta del server>*[|*<eventuale password>*] altrimenti verrà segnalato all'utente che il codice non è ben formattato e non si può procedere oltre.

A questi semplici passaggi però occorre integrare anche la capacità di controllare tramite la localizzazione GPS se il dispositivo si trovi effettivamente nei pressi del codice che si sta scansionando. Prima di fare questo però occorre controllare che il sensore sia attivo (il fatto che sia presente è scontato perché il file *manifest.xml* dell'applicazione specifica espressamente che è richiesta la disponibilità del GPS: `<uses-feature android:name="android.hardware.location.gps" />`). Nel caso non fosse attivo allora si chiede all'utente se intende che gli venga mostrata la schermata delle impostazioni delle connessioni del dispositivo da cui potrà accendere il sensore. Qualora non accettasse o comunque tornasse indietro senza attivarlo allora non si potrà procedere con la scansione del codice.

```
private void newReservation() {
  locationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
  boolean gpsEnabled = locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER);
    if(!gpsEnabled){
        // Mostro Dialog che chieda all'utente di andare alla schermata delle connessioni
        // Se clicca Sì
        […]
        // Alla schermata delle impostazioni di connessione
        Intent settingsIntent = new Intent(Settings.ACTION_LOCATION_SOURCE_SETTINGS);
        startActivityForResult(settingsIntent,GPS_ACTIVITY);
    }else{
        // Procede con la scansione
        IntentIntegrator integrator = new IntentIntegrator(this);
        integrator.initiateScan(IntentIntegrator.QR_CODE_TYPES);
    }
}
```

Al ritorno dalla *activity* delle connessioni viene, come da prassi, invocata la funzione `onActivityResult` nella quale si dovrà distinguere da quale situazione l'utente sta ritornando (nel caso di KeepInQ questa funzione viene chiamata quando l'utente ritorna dalla *activity* della singola prenotazione, dalla schermata delle connessioni e dalla scansione del codice QR) e sarà necessario controllare nuovamente se il sensore GPS sia stato attivato o meno. In caso positivo si procede con la scansione.

```
public void onActivityResult(int requestCode, int resultCode, Intent intent) {
  // L'utente è ritornato dalla ReservationActivity
  if(requestCode == RESERVATION_ACTIVITY){
        […]
  }else if(requestCode == GPS_ACTIVITY){
        locationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
        boolean nowEnabled = locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER);
        if(!nowEnabled){
          // evidentemente l'utente è tornato senza abilitare il gps...
          Toast.makeText(KeepInQActivity.this, "GPS is still disabled: cant' proceed",
                                                                    Toast.LENGTH_LONG).show();
        }else{
                IntentIntegrator integrator = new IntentIntegrator(this);
                integrator.initiateScan(IntentIntegrator.QR_CODE_TYPES);
        }
   }else{
        //L'utente ha scansionato il codice
        IntentResult scanResult = IntentIntegrator.parseActivityResult(requestCode, resultCode, intent);
        if (scanResult != null) {
                String contents = scanResult.getContents();
                // Manipolazione testo decodificato
                […]
        }
```

Il codice QR quindi non conterrà solamente le indicazioni per contattare il server associato, ma anche le coordinate in latitudine e longitudine della posizione fisica della coda. La sintassi completa del testo da codificare sarà

*<Nome della prenotazione>***|***<indirizzo IP del server>***|***<porta del server>*[**|***<eventuale password>*] **@***lat***,***lon*

dove *lat* e *lon* devono rispettare il formato *gradi***:**minuti**:**secondi di arco (sono accettate comunque anche coordinate che si limitano a specificare solo gradi o gradi e minuti a scapito dell'accuratezza).

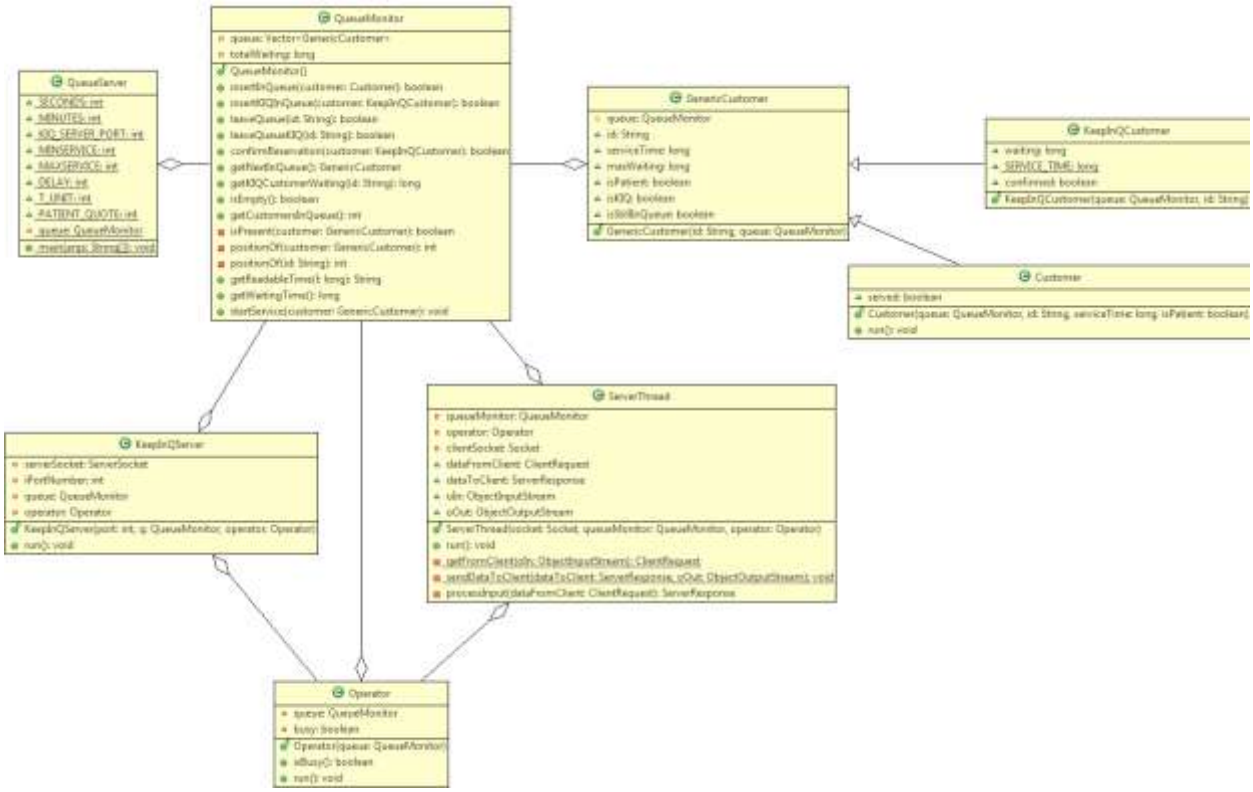
Il testo decodificato viene quindi diviso in una parte relativa al server e una relativa alla localizzazione, poi si procede a controllare la vicinanza del dispositivo a queste coordinate.

In Android per poter usufruire delle capacità di localizzazione dei dispositivi occorre utilizzare il servizio *LocationManager* il quale mette a disposizione una serie di API per poter determinare la posizione del dispositivo. Una volta istanziato si ha la possibilità di scegliere tra una serie di *LocationProvider* quello che rispetta al meglio una serie di criteri specificati dallo sviluppatore, quali ad esempio l'accuratezza, il consumo energetico o un eventuale costo per il servizio. I *LocationProvider* si occupano di memorizzare periodicamente la posizione del dispositivo quindi, una volta scelto quello più adatto alle esigenze, è possibile interrogarlo per ottenere l'ultima posizione nota e inseguito controllarne la distanza dalle coordinate decodificate dal codice QR.

```
private boolean isCloseToQR(String lat, String lon) {

   Criteria criteria = new Criteria();
   criteria.setAccuracy(Criteria.ACCURACY_FINE);
   criteria.setAltitudeRequired(false);
   criteria.setBearingRequired(false);
   criteria.setCostAllowed(false);

   String provider = locationManager.getBestProvider(criteria, false);
   Location devicePosition = locationManager.getLastKnownLocation(provider);

   float[] results = new float[1];
   Location.distanceBetween(devicePosition.getLatitude(), devicePosition.getLongitude(),
                                          Location.convert(lat),Location.convert(lon), results);
   return results[0] < DISTANCE_MAX;
}
```

La distanza calcolata tra la posizione del dispositivo e le coordinate decodificate viene memorizzata in un vettore di tipo `float` di cui viene preso in considerazione solo il primo valore (gli altri due sono la direzione iniziale e finale per il percorso più breve tra i due punti) che viene confrontato con un parametro di distanza massima. Se la disuguaglianza è rispettata allora l'utente è abilitato a contattare il server e quindi ad accodarsi o confermare la prenotazione.

# *Architettura del Server*



*(Nel diagramma non sono presenti le classi ClientRequest e ServerResponse che sono le stesse utilizzate dal client.)*

Per poter testare l'applicazione si è creato un server che cercasse in qualche modo di simulare una situazione di reale utilizzo di KeepInQ. La classe *QueueMonitor* rappresenta la coda del sistema a cui si possono presentare dei *GenericCustomer* che possono essere sia dei normali *Customer* sia dei *KeepInQCustomer*. Un oggetto *Operator* simula il comportamento di un addetto ad uno sportello, ovvero estrae un cliente dalla coda e inizia il servizio richiesto dal cliente oppure, se si tratta di un cliente KIQ, rimane impegnato con questi fino a quando non riceve un input da *console*.

Inizialmente la classe *QueueServer* crea dei *GenericCustomer* assegnando loro in maniera casuale un tempo di servizio compreso tra un tempo minimo e massimo specificabili da riga di comando all'avvio del server; in aggiunta, con una probabilità anch'essa assegnabile, stabilisce se si tratta di un cliente paziente oppure se dopo un certo periodo di attesa questi decida di lasciare la coda. In quest'ultimo caso sarà l'operatore ad accorgersi che il cliente non è più in coda e passerà quindi al cliente successivo.

```
//Simula l'arrivo di altri clienti
while(true){
        long serviceTime = (new Random().nextInt(maxService - minService) + minService) * t_unit;
        boolean isPatient = (new Random().nextInt(100) > PATIENT_QUOTE);
        customer = new Customer(queue, "Customer " + ((i>9)?i:" "+i), serviceTime, isPatient);
        customer.start();
        queue.insertInQueue(customer);
        i++;
        if(i > 7) //I primi 8 clienti arrivano subito: la coda è già formata
                sleep(DELAY * T_UNIT);
        }
}
```

Prima di simulare l'arrivo dei clienti viene anche avviato l'operatore e la classe *KeepInQServer* che si occuperà di ricevere le connessioni dei dispositivi KIQ e la cui struttura è piuttosto semplice: rimane in attesa di una connessione TCP su una porta specificata dal *QueueServer* e avvia un oggetto *ServerThread* quando questa avviene. Sarà quest'ultima classe ad occuparsi di gestire i comandi inviati dal client KIQ ed apportare le eventuali modifiche alla coda del sistema.

Per un *KeepInQCustomer* il tempo totale di attesa in coda prima di poter essere servito è dato dalla somma di tutti i tempi di servizio dei clienti accodati prima di lui, senza tenere conto del tempo di servizio dell'utente che l'operatore sta effettivamente servendo, quindi si può dire che effettivamente è il tempo di attesa prima di essere il primo della fila. Per i *Customer* questo dato ovviamente non ha rilevanza. Ha invece importanza sapere se sono stati serviti prima che scada il time-out della loro 'pazienza', ovvero se l'operatore ha iniziato il loro servizio. In caso negativo il *Customer* segnalerà che se ne sta andando, ma non si provvederà a rimuoverlo fisicamente dalla coda, un po' come quando in uno scenario reale ci si prenota con un bigliettino numerato e non ci si presenta quando viene chiamato il proprio numero: se non c'è risposta alla chiamata, si passa al numero successivo.

```
public class Customer extends GenericCustomer {
        […]

        public void run(){
                // simula il cliente che spazientito abbandona la fila prima di venir servito...
                if(!isPatient){
                        sleep(maxWaiting);
                        if(!served)
                                queue.leaveQueue(id);
                }
        }
}
```

```
public class QueueMonitor {
        […]
        public boolean leaveQueue(String id) {
                int pos = positionOf(id);
                if(pos >= 0){
                        queue.elementAt(pos).isStillInQueue = false;
                        return true;
                }
                return false;
        }
}
```

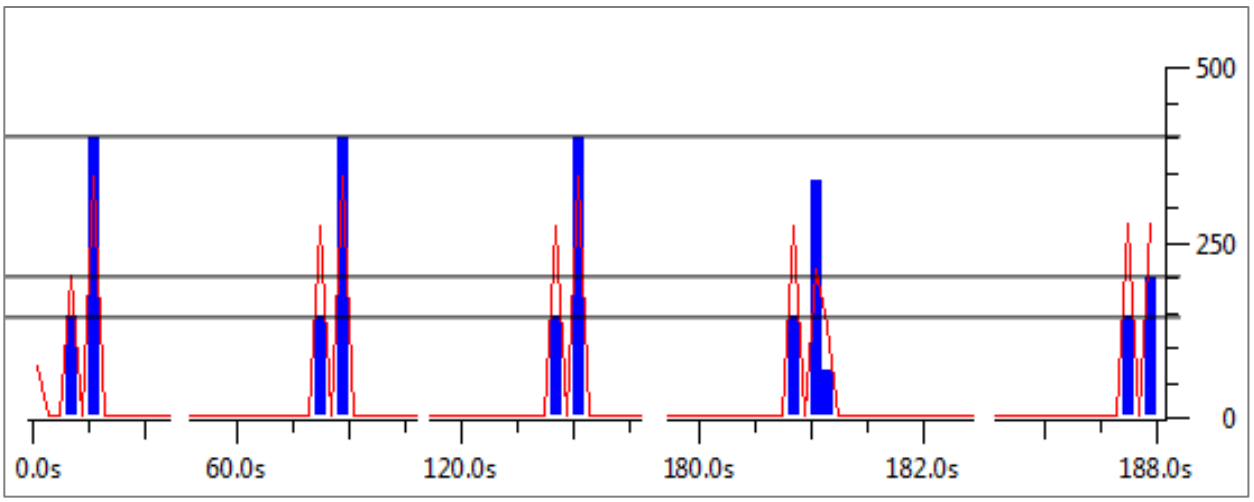
Quando l'*Operator* noterà che la variabile `isStillInQueue` è di valore falso allora passerà direttamente al cliente successivo.

# *Test di performance*

Per eseguire i test dell'applicazione è stato utilizzato un dispositivo LG modello P500 connesso tramite una LAN Wifi al server in esecuzione su un notebook. Per poter catturare e analizzare lo scambio di dati tra client e server si è scelto di utilizzare il software WireShark versione 1.6.8.

Il programma è stato configurato per catturare lo scambio di pacchetti sulla porta 8888 ed è stato eseguito un test di un ciclo tipico dell'applicazione KeepInQ, ovvero prenotazione, attesa con auto-update attivato e conferma della prenotazione non appena il server comunichi la disponibilità. Per velocizzare il tutto è stato impostato il server con un tempo di servizio dei clienti tra i 30 e i 50 secondi e con già 7 clienti presenti in coda, mentre il client con un auto-update ogni minuto.

Dal traffico tra i due endpoint durante il test è stato estrapolato il seguente grafico in cui è indicato in ascissa il tempo in secondi e in ordinata i Bytes inviati rispettivamente dal client (barra blu) e dal server (linea rossa). Gli *spikes* nel grafico indicano nell'ordine lo scambio di dati durante una prenotazione, 3 aggiornamenti automatici e la conferma della prenotazione.




Ogni comunicazione da parte del client pesa quindi circa 500bytes per un totale di 6 pacchetti mentre ogni risposta da parte del server pesa poco più di 600bytes in 7 pacchetti. Una singola conversazione quindi tra client e server richiede un totale di circa 1,2kbytes, ad eccezione del caso in cui il client chiede di venir cancellato dalla coda perché il server non invia nessun messaggio di conferma oltre ai pacchetti per il TCP.

Questi valori possono essere confermati guardando il riassunto dei dati catturati durante il test:

IPv4 Endpoints

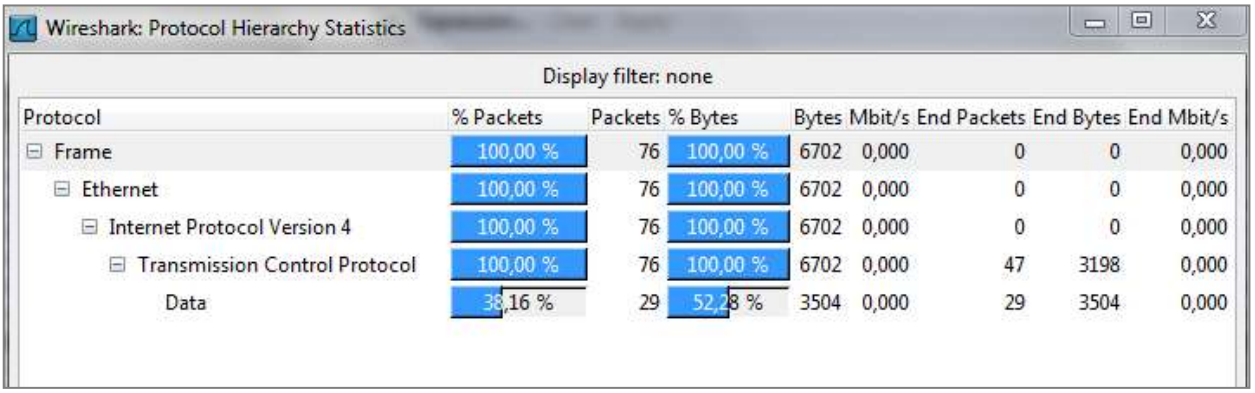
| Address | Packets | Bytes | Tx Packets | Tx Bytes | Rx Packets | Rx Bytes |
|---|---|---|---|---|---|---|
| 192.168.0.139 | 76 | 6 702 | 41 | 3 642 | 35 | 3 060 |
| 192.168.0.169 | 76 | 6 702 | 35 | 3 060 | 41 | 3 642 |

.139: IP del client

.169: IP del server

Il client ha inviato un totale di 3642 bytes in 41 pacchetti, ogni pacchetto pesa circa 88 bytes e quindi ogni comunicazione di 6 pacchetti pesa 528 bytes. Lo stesso ragionamento è valido per il server.

Quanti di questi dati però sono effettivamente informazioni per l'applicazione e quanti invece sono dati di controllo per il TCP?




| Protocol | % Packets | Packets | % Bytes | Bytes | Mbit/s | End Packets | End Bytes | End Mbit/s |
|---|---|---|---|---|---|---|---|---|
| Frame | 100,00 % | 76 | 100,00 % | 6702 | 0,000 | 0 | 0 | 0,000 |
| Ethernet | 100,00 % | 76 | 100,00 % | 6702 | 0,000 | 0 | 0 | 0,000 |
| Internet Protocol Version 4 | 100,00 % | 76 | 100,00 % | 6702 | 0,000 | 0 | 0 | 0,000 |
| Transmission Control Protocol | 100,00 % | 76 | 100,00 % | 6702 | 0,000 | 47 | 3198 | 0,000 |
| Data | 38,16 % | 29 | 52,28 % | 3504 | 0,000 | 29 | 3504 | 0,000 |

Da questo grafico offerto da Wireshark si può notare che solo il 38% dei pacchetti inviati è effettivamente di dati e che corrispondono al 52% del carico totale di bytes trasmessi. Si potrebbe pensare quindi di provare ad alleggerire le comunicazioni utilizzando il protocollo UDP, ma il fatto che non ci sia garanzia della consegna dei pacchetti richiederebbe un controllo molto più sofisticato per evitare incoerenze tra lo stato delle prenotazioni presenti nel client e quelle residenti nel server (ad esempio se non venisse notificato correttamente al server che si sta abbandonando una coda, nell'eventualità che l'utente voglia accodarsi nuovamente, si dovrà attendere che scada la precedente prenotazione, dato che al server risulta ancora presente mentre nel client la prenotazione è stata rimossa dal database). In conclusione l'utilizzo del TCP comporta forse uno 'spreco' di dati trasmessi ma aiuta a non creare situazioni spiacevoli che farebbero alterare l'utente.

Il principale consumo di batteria del dispositivo durante l'utilizzo di KeepInQ riguarda le comunicazioni con il server. Consigliano gli sviluppatori di Android di cercare raggruppare le trasmissioni tra di loro, così da sfruttare con un'unica accensione del componente radio più trasferimenti dati contemporaneamente. Sarebbe anche buona prassi stabilire che tipo di connessione dati è al momento attiva nel dispositivo e cercare di scaricare più dati se questa connessione richiede un costo maggiore, così da aumentare il rapporto dati scaricati - consumo energetico.

Nel caso di KeepInQ queste accortezze non hanno molto significato dato che le connessioni devono essere per loro natura sporadiche e la quantità di dati trasmessa è più o meno la stessa ad ogni comunicazione. Si sarebbe potuto implementare una modalità di aggiornamento automatico che, in base al tempo di attesa comunicato dal server al momento della registrazione, eseguisse un numero di update prefissato (ad esempio 4 update per ogni prenotazione), ma si è preferito lasciare la libertà all'utente di scegliere quanti aggiornamenti in modo da adattarli a suo piacere a seconda delle prenotazioni registrate. In ogni caso, sarebbe una modalità di facile implementazione nell'applicazione.