

# ERA Streaming

## Assistenza da remoto tramite Streaming

Paolo Bonato  
Università degli Studi di Padova  
Email: paolo.bonato.12@gmail.com

Tommaso Padovan  
Università degli Studi di Padova  
Email: tommaso.pado@gmail

**Abstract**—Questo report descrive l'applicazione ERAStreaming e le sue fasi di sviluppo. Si tratta di una applicazione sviluppata nativamente per *smartphone* Android che permettere ottenere assistenza da remoto tramite *Streaming*. Fornisce inoltre un *hub* sociale che permette agli utenti di entrare in contatto per offrire o ricevere assistenza.

Per fornire alcuni servizi in tempo reale come la *chat* o le recensioni degli utenti ERAStreaming usa un database noSql real-time. Per quanto riguarda lo *Streaming* invece fa uso di un server dedicato con un protocollo di tipo RTMP creato appositamente dagli autori.

### I. INTRODUZIONE

Lo scopo del progetto è quello di permettere all'utenza di ricevere assistenza remota per un ventaglio quanto più possibile vasto di ambiti. Chi ha bisogno di supporto per una qualsiasi mansione avrà la possibilità di collegarsi con un tecnico competente che potrà, tramite *streaming* video, supervisionare il lavoro e dare istruzioni per portarlo a termine.

#### A. Target

Il target naturale per questo genere di applicazioni dovrebbe essere quello degli *smart-glasses*, o comunque dispositivi *wearable*. Essi infatti permettono di operare a mani libere e di avere un punto di vista preferenziale per il tutor.

Durante le fasi preliminari di questo progetto, pertanto, è stato condotto uno studio di fattibilità, riguardante in particolare i dispositivi *wearable*: essi si sono però dimostrati una tecnologia ancora non sufficientemente matura. dal punto di vista *Hardware*:

- Durata insufficiente della batteria.
- Surriscaldamento.
- Limitata potenza di calcolo, e quindi di compressione dell'immagine.
- Scarsa qualità della fotocamera.

Ma anche *Software*:

- La maggior parte dei dispositivi supporta una versione ridotta di Android KitKat 4.4,
- La maggior parte dei dispositivi non supporta le nuove API per gestire la camera, e fa utilizzo di API precedenti deprecate negli altri dispositivi.

Per questi motivi dunque si è scelto di sviluppare l'applicazione per *smartphone*, lasciando il *porting* o lo sviluppo di una *companion-app* per un momento futuro.

#### B. Motivazioni e Contesto

Il *concept* applicazione è nato per utilizzo in ambito professionale, ad esempio per diminuire il numero di trasferte per riparazione di macchinari oppure per ricevere veloce supporto in operazioni critiche in cui non c'è tempo o possibilità di attendere l'arrivo di un tecnico specializzato. In generale nel contesto dell'industria 4.0 servizi di questo genere saranno sempre più richiesti in quanto sempre più dispositivi saranno connessi alla rete e quindi mantenuti da remoto, per gli altri di certo, ci sarà necessità di una assistenza altrettanto immediata. Tale servizio, però, potrebbe prendere piede anche in ambito *consumer*: sostituire lo schermo dello *smartphone*, sostituire la batteria danneggiata di un *laptop*, riparare un piccolo elettrodomestico e molto altro sono esigenze all'ordine del giorno. Per queste motivazioni è stato deciso di intraprendere questo progetto sviluppando non solo un sistema efficiente di *streaming*, ma anche una interfaccia semplice ed immediata per consentire a tutte le fasce di utenza di accedervi. Inoltre sarà possibile in futuro salvare le sessioni di assistenza sul server creando così un database di contenuti e di *know-how* autogenerati.

### II. PROPOSTA DI SOLUZIONE

La parte di *hub* e quella di *streaming* sono state sviluppate in maniera completamente indipendente. Si è deciso di mantenerle distaccate il più possibile per due ragioni principali:

- La parte di *streaming* deve essere un prodotto indipendente: alcuni *wearable* potrebbero non avere uno schermo (e.g. i *Google Glass*) o potrebbero avere altre limitazioni.
- Le due componenti hanno esigenze computazionali molto diverse: la prima ha bisogno di indicizzare e gestire velocemente grandi liste di danti, mentre la seconda deve avere una infrastruttura per gestire lunghi *burst* e di uno spazio di dimensioni di archiviazioni maggiori.

Nelle sezioni seguenti verranno esposte nel dettaglio queste due componenti.

#### A. Social

Questa macro-componente del sistema si occupa di interfacciarsi con l'utente, raccogliere informazioni, far incontrare l'utente con il tutor adatto e passare i dati dei dispositivi da far comunicare alla macro-componente di *streaming*. Inoltre fornisce una *real-time chat* ed un sistema di *rating*.

Un aspetto centrale di questa componente è il coinvolgimento

dell'utenza. Per questo si è cercato di tenere il *client* il quanto più possibile semplice e reattivo.

La tecnologia predominante usata in questa parte è la *suite* *Firebase*.

#### 1) *Firebase*:

- **Realtime - noSQL** *Firebase Database* è il database consigliato dalle *best-practices* Android. È un database di tipo noSQL che salva i dati in formato *json*. Ha un ottima integrazione con il *framework* Android in quanto permette di salvare e leggere dati direttamente come oggetti Java.

Non solo permette, ma costringe lo sviluppatore a leggere i dati in maniera asincrona tramite *listener design pattern*. Ciò si riflette in una ottima reattività del prodotto, in quanto tutti i dati sono aggiornati in maniera istantanea non appena essi vengono modificati sul database senza bisogno di *refresh* o altre azioni da parte dell'utente.

- **Serverless** Questa *suite* incoraggia una architettura *serverless*: ovvero non necessita di un server fisico e statico. Il database e qualsiasi altra infrastruttura sono *hostati* dai server Google e la quantità di spazio disco, banda, CPU etc. viene modificata dinamicamente a seconda del carico a cui l'applicazione è sottoposta.

2) *Struttura generale*: L'applicazione è stata realizzata nativamente per Android. L'aspetto centrale è la figura dell'utente che ha il suo profilo con nome, cognome, email, foto, biografia e lista di competenze. La caratterizzazione dell'utente potrà essere facilmente incrementata nel futuro perché è stata implementata con la massima flessibilità.

Ogni utente inoltre può cercare nel database altri profili in base alle competenze necessarie, ma anche per nome o email. Una volta trovato il tutor cercato è possibile aggiungerlo alla propria rete di contatti: una volta che la richiesta viene accettata sarà possibile accordarsi tramite chat e poi far partire una sessione di supporto video. Ovviamente è possibile anche rimuovere un contatto indesiderato.

Ogni utente può modificare il proprio profilo in ogni momento; per migliorare la qualità dei profili disponibili al primo login viene chiesto di fornire alcuni dati personali.

Per mantenere una buona qualità delle sessioni di assistenza, è messa a disposizione una funzione di recensione degli utenti.

3) *Realtime chat*: La chat in tempo reale è uno dei più importanti requisiti di questa parte, deve essere semplice, immediata e fruibile. La soluzione proposta usa nativamente *Firebase* sfruttando la sua natura real-time.

Ad ogni coppia di utenti viene assegnato un codice univoco di una stanza di chat che non è altro che un nodo particolare dell'albero *json* presente nel database a cui entrambi hanno permessi di lettura e scrittura. Per inviare un nuovo messaggio è sufficiente quindi che il client faccia la *push* di un nuovo messaggio sul database. Per visualizzare la lista dei messaggi invece basta seguire ancora una volta il *listener design pattern* e registrare entrambi gli utenti alla lista degli ascoltatori di una particolare stanza di chat, il sistema Android

quindi provvederà automaticamente ed in tempo reale a tenere aggiornata la pagina con l'elenco dei messaggi.

4) *Rete dei contatti*: La relazione di "amicizia" o di "contatto" tra coppie di utenti è un classico *use-case* per i database di tipo relazionale; in un database di tipo noSQL apre una problematica molto diversa. *Firebase* non fornisce un servizio di database relazionale, quindi è necessario usare una struttura diversa e inevitabilmente introdurre ridondanza.

Si prenda in considerazione il problema di trovare tutti gli ID degli "amici" di un determinato utente: se il database fosse strutturato con un nodo *json* che contiene coppie di utenti tra loro "amici" (come in un classico SQL) per ottenere tale lista sarebbe necessario un tempo lineare nella dimensione del database, mentre aggiungendo ad ogni utente la lista degli id dei propri amici è possibile ottenere questa lista in  $O(1)$ .

Per gestire in maniera controllata questa ridondanza ed i problemi di sicurezza che ne conseguono è necessario del codice lato server. *Firebase* fornisce una funzionalità chiamata *Firebase Functions* che permette esattamente di fare questo; sempre in un contesto *serverless* è possibile caricare del codice *nodeJS* che viene eseguito in risposta ad eventi legati al database oppure in risposta a delle richieste *GET* ad un determinato *end-point http*.

5) *Notifiche*: Le notifiche sono un altro aspetto molto importante per questo genere di applicazioni. Ad esempio devono essere mandate all'utente quando riceve un nuovo messaggio di chat, oppure quando ha una nuova richiesta di amicizia e così via.

Ancora una volta per la gestione delle notifiche push è stato scelto un servizio Google denominato *Firebase Cloud Messaging* o FCM. Ad ogni utente *Firebase* viene associato automaticamente un token identificativo, esso può essere usato per inviare *push message*. Inoltre gruppi di utenti possono sottoscrivere a *topic* e possono ricevere notifiche push riguardo ai loro argomenti di interesse.

Nel prodotto presentato questa funzione è stata implementata unendo *Firebase Database*, *Firebase Functions* e il *Cloud Messaging*. Non è indicato che un utente abbia accesso al token FCM di un altro, perché creerebbe grossi problemi di sicurezza; pertanto l'invio di notifiche push da parte di un utente *A* verso un altro utente *B* (come nel caso di una richiesta di amicizia) è stato strutturato come segue:

- **Push request**: L'utente *A* aggiunge un nodo (contenente tutti i dati della notifica) al database in una tabella speciale ad esempio */pushQueues/friendRequests*. È una semplice scrittura su database, *A* a questo punto non deve fare altro.
- **Sanity check**: Il nodo sopra citato è "speciale" in quando ha una *Firebase Function* registrata come *observer*. Questa funzione ad ogni nuova scrittura in *pushQueues* legge i dati della richiesta di notifica push, verifica che *A* abbia i corretti permessi per mandare una notifica a *B* e in caso affermativo legge nel database l'FCM-token di *B*.
- **Push notification**: Se la funzione del passo precedente termina con esito positivo allora viene chiamato

il servizio FCM, che si occupa di inviare un *Firebase Message* al corretto dispositivo (o gruppo di dispositivi).

- **Ricezione:** Il client installato sul dispositivo dell'utente  $\mathcal{B}$  riceve il messaggio tramite un *service* e notifica l'utente in base alle sue impostazioni sulle notifiche.

*B. Streaming*

### III. CONCLUSIONI

*A. Risultati*

*B. Sviluppi futuri*

### ACKNOWLEDGMENT

The authors would like to thank...

### REFERENCES

- [1] H. Kopka and P. W. Daly, *A Guide to L<sup>A</sup>T<sub>E</sub>X*, 3rd ed. Harlow, England: Addison-Wesley, 1999.