UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Tamara D. Ivanović

IMPLEMENTACIJA UPRAVLJAČA ZA DIGITALNU TELEVIZIJU KORIŠĆENJEM PLATFORME ANDROID

master rad

Beograd, 2023.

Mentor:

dr Milena Vujošević Janičić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Filip Marić, redovni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Aleksandar Kartelj, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Mami, tati i dedi

Naslov master rada: Implementacija upravljača za digitalnu televiziju korišćenjem platforme Android

Rezime: Apstrakt rada na srpskom jeziku u odabranom pismu

Ključne reči: analiza, geometrija, algebra, logika, računarstvo, astronomija

Sadržaj

Glava 1

Uvod

Glava 2

Android

Operativni sistem Android (u nastavku OS Android) je operativni sistem zasnovan na Linuks jezgru (eng. *Linux kernel*) i pripada zajednici otvorenog koda. U ovom poglavlju biće reči o samom nastanku i razvoju ovog operativnog sistema, arhitekturi i osnovnim kompontentama. Kako je centralna tačka ovog rada aplikacija koja kontroliše set top-boks (eng. *set top-box*, skraćeno STB) uređaje koja je pisana u programskom jeziku Java biće objašnjen i odnos OS-a Android sa njima. Radi boljeg razumevanja rada aplikacije ovo poglavlje će se osvrnuti i na značaj i funkcionisanje Google API-ja, kao i na metod povezivanja dva uređaja sa OS-om Android pomoću mreže.

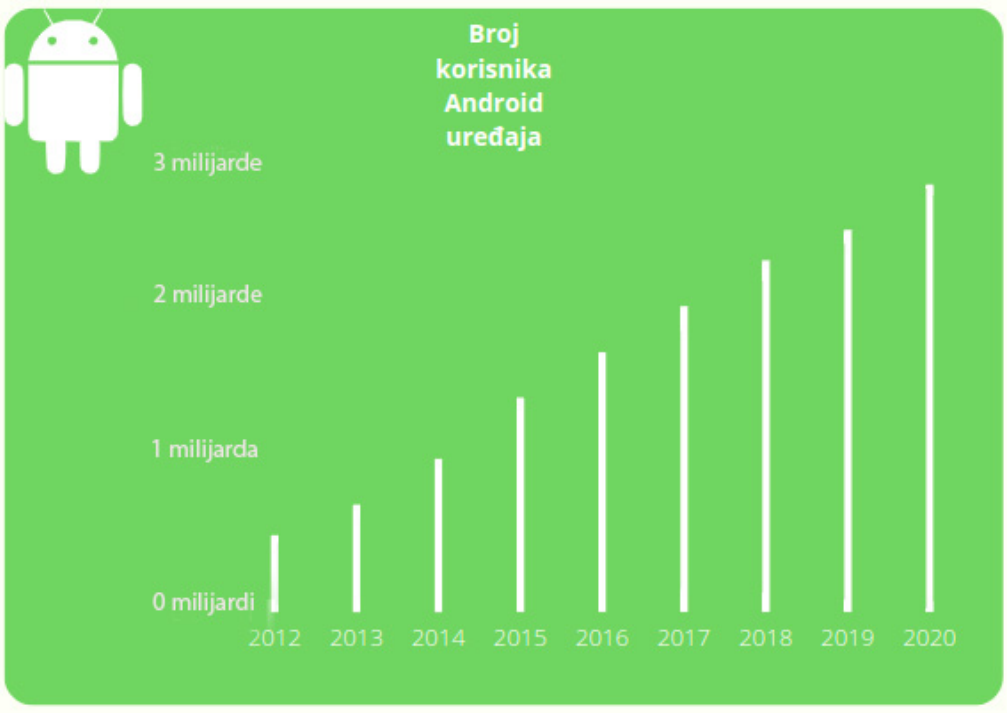
2.1 Istorijat

Endi Rubin (*Andy Rubin*) je 2003. godine sa trojicom kolega u Palo Altu osnovao kompaniju *Android Inc.* sa namerom da kreiraju platformu za kameru sa podrškom za skladištenje u oblaku. Takva ideja nije naišla na podršku investitora i cilj kompanije se preusmerio na pametne mobilne telefone, a vremenom i sve pametne uređaje. Zamisao je bila da sistem bude besplatan za korisnike, a da zarada zavisi od aplikacija i ostalih servisa. To je postalo moguće 2005. godine kada je kompanija *Google* kupila kompaniju i ostavila priliku osnivačima na čelu sa Rubinom da nastave sa razvojem ovog operativnog sistema [5].

Sam razvoj operativnog sistema i dalje traje, verzije su mnogobrojne i izlaze često, a svaka verzija donosi sa sobom znacajna poboljšanja [11, 14]. Svaka verzija je označena brojem, kao i nazivom slatkiša po ideji projektnog menadžera Rajana Gibsona. U tabeli 2.1 je prikazan pregled najznačajnijih verzija zajedno sa novitetima

koje su donele.

Na početku razvoja, OS Android je svoju primenu našao u pametnim telefonima i tabletima. Tokom godina programeri su raširili upotrebu na media plejere (za Android TV), pametne satove i naočare, kućne uređaje, automobile, kamere, konzole za igru [11]... Prema statistici [2], Kineske kompanije drže više od 55% Android tržišta. Od svih kompanija na tržištu najveći udeo imaju: *Samsung* (37.10%), *Xiaomi* (11.20%) i *Huawei* (11%). Sa rastom raznovrsnosti aplikacija koje postoje za Android uređaje, kao i broja različitih uređaja koji se koriste rastao je i broj korisnika. Statistika vezana za trend rasta broja korisnika može se videti na slici 2.1.



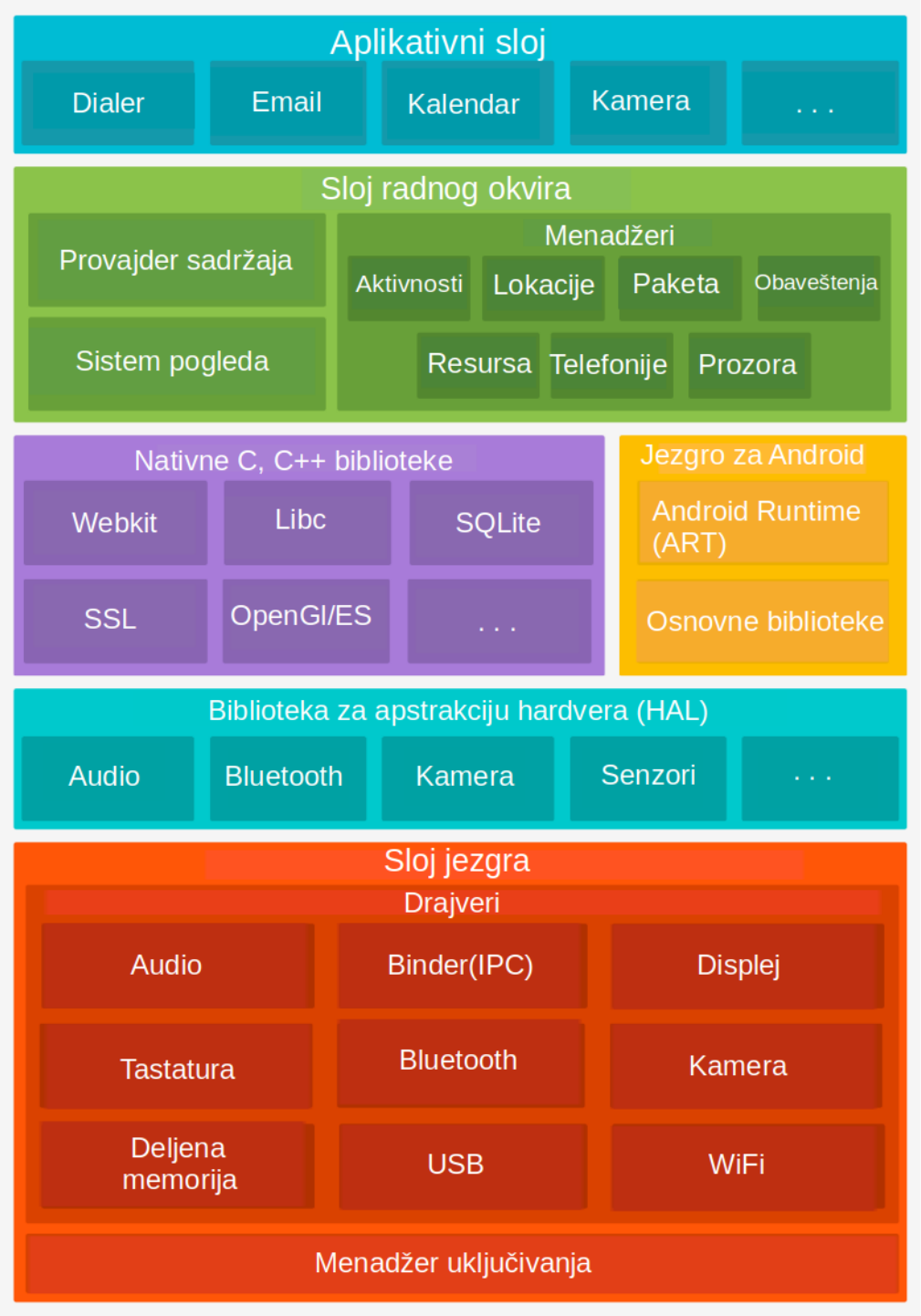
Slika 2.1: Broj korisnika tokom godina, izvor [2]

2.2 Arhitektura

Android platforma predstavlja skup komponenti kao što su OS Android, biblioteke, radni okviri, API-ji i sl. koji omogućavaju razvoj i izvršavanje Android aplikacija. Zasnovana je na Linuks jezgru pri čemu se jezgro i drajveri nalaze u prostoru jezgra (eng. *kernel space*), a nativne biblioteke u korisničkom prostoru (eng. *user space*). Sve aplikacije se izvršavaju u Java virtuelnoj mašini koja se zove *Android Runtime*

Tabela 2.1: Verzije OS-a Android

Naziv verzije	Datum objavljivanja	Najznačajniji noviteti
Android 1.0	Septembar 2008.	Podrška za kameru, internet pregledač, preuzimanje i objavljivanje aplikacija na *Android Market*-u, integrisani su *Google* servisi: *Gmail*, *Google maps*, *Google Calendar*, omogućene *Wi-Fi* i *bluetooth* bežične komunikacije
Android 1.5 — ***Cupcake***	April 2009.	Poboljšana *Bluetooth* komunikacija, tastatura sa predikcijom teksta, snimanje i gledanje snimaka
Android 2.2 — ***Froyo***	Maj 2010.	Poboljšanje brzine, implementacija *JIT*-a, instaliranje aplikacija van memorije telefona, povezivanje uređaja preko USB
Android 3.x — ***Honeycomb***	Februar 2011.	Višeprocesorska podrška, *Google Talk* video čet, '*Private browsing*', uživo prenos preko HTTP-a, *USB host API*, jednostavnije automatsko ažuriranje aplikacija preko *Android Marketa*
Android 4.1–4.3 — ***Jelly Bean***	Jul 2012.	Glasovna pretraga, *WiFi/WiFi-Direct* otkrivanje servisa, bezbedno USB debagovanje, 4K podrška, podrška za BLE (*Bluetooth Low Energy*), *WiFi scanning API*
Android 6.0 — ***Marshmallow***	Oktobar 2015.	Podrška za USB tip C, autentikacija pomoću otiska prsta, MIDI podrška
Android 8.0/8.1 — ***Oreo***	Avgust 2017.	Svetla i tamna tema, PIP (*Picture-In-Picture*) sa opcijom promene veličine, API-ji za neuronske mreže i za deljenu memoriju
Android 9 — ***Pie***	Avgust 2018.	Prikaz celog teksta i slike u obaveštenjima o porukama, dugme za gašenje može i da snimi sliku ekrana
Android 10 — ***Queen Cake***	Septembar 2019	Bolja podrška za privatnost, pristup sistemskim podešavanjima iz panela, biometrijska autentikacija unutar aplikacija
Android 11 — ***Red Velvet Cake***	Septembar 2020.	Snimak ekrana, balončići za poruke, podrška za 5G, bežično debagovanje, bolja podešavanja za dozvole
Android 12 — ***Snow Cone***	2021.	*Material You* jezik za dizajn, podrška za *AVIF*, *Android Private Compute Core*



Slika 2.2: Arhitektura OS Android, slika kreirana na osnovu [12]

(ili, skraćeno, ART), a postoje Java biblioteke koje povezuju aplikaciju sa bibliotekama napisanim u nativnom jeziku. Sama arhitektura softvera kod Androida je

slojevita i postoje četiri sloja koja se naslanjaju na sloj fizičke arhitekture. Slojevi arhitekture prikazani su na slici 2.2, a navedeni od viših ka nižim (eng. *top-down*) su [6]:

Aplikativni sloj (eng. *application layer*) predstavlja skup svih aplikacija koje se izvršavaju na Androidu. Aplikacije mogu biti sistemske, ugrađene ili korisničke. Sistemske su one koje je napisao proizvođač uređaja, ugrađene su one koje su drugi kreirali ali su unapred instalirane na uređaj i ne mogu se obrisati, a sve ostale su korisničke.

Sloj radnog okvira (eng. *frameworks layer*) predstavlja sloj koji omogućava da se premoste razlike između aplikativnog i nativnog sloja i koji određuje ograničenja koja moraju da se poštuju pri razvoju Android aplikacija. Ovaj sloj je napisan u programskom jeziku Java, a pomoću interfejsa JNI (skraćeno od eng. *Java Native Interface*) komunicira sa nativnim slojem. Najbitniji elementi sloja mogu se videti na slici 2.2, a neki od njih su: menadžer aktivnosti koji upravlja životnim ciklusom aplikacije, menadžer paketa koji poseduje informacije o svim instaliranim aplikacijama na uređaju, menadžer lokacije koji pronalazi geografsku lokaciju uređaja i menadžer telefonije koji omogućava pristup sadržajima telefonije kao što su brojevi telefona.

Izvršni sloj (eng. *runtime layer*) napisan je u nativnom jeziku (C ili C++), sastoji se od nativnih biblioteka, biblioteka za apstrakciju hardvera (*HAL*) i izvršnog okruženja za Android (eng. *Android runtime*) u koji spadaju osnovne bibilioteke i ART. Do verzije 5.0 ART nije postojao već je korišćena *Dalvik virtuelna mašina (skraćeno DVM)* [10].

Sloj jezgra (eng. *kernel layer*) predstavlja sloj između hardvera i softvera koji poseduje sve drajvere koji su potrebni za hardverske komponente. Takođe, u ovom sloju je moguće pronaći sve vezano za upravljanje memorijom, procesima i uključivanjem, kao i o bezbednosti.

2.3 Komponente Android aplikacije

Pod Android aplikacijom se smatra bilo koja aplikacija koja se pokreće na uređaju sa OS-om Android. Programiranje ovih aplikacija je moguće u mnogim pro-

gramskim jezicima, dok se zvaničnim smatraju programski jezici Java i Kotlin [3]. U nastavku će biti reči o programiranju u programskom jeziku Java.

Kreiranje Android aplikacija ne bi bilo moguće bez njenih osnovnih komponenti. Svaka komponenta ima svoje karakteristike, slučajeve upotrebe kao i funkcije koje vrši. Moguće je i poželjno kombinovati ih u aplikaciji. Svaka komponenta koje se kreira u aplikaciji mora da se navede u datoteci *AndroidManifest.xml*. Četiri osnovne komponente su:

1. Aktivnosti (eng. *activity*)
2. Servisi (eng. *service*)
3. Prijemnici (eng. *broadcast receiver*)
4. Provajderi sadržaja (eng. *content provider*)

Aktivnosti

Aktivnosti predstavljaju jedan prikaz grafičkog korisničkog interfejsa (eng. *Graphical User Interface*) na ekranu. Ne postoji ograničeni broj aktivnosti koje jedna aplikacija može imati, takođe moguće je da postoje aplikacije bez aktivnosti. Za razliku od mnogih programskih jezika gde pokretanje aplikacije počinje pozivom metoda *main()* i uvek od istog mesta, Android aplikacije ne moraju uvek započinjati na istom mestu. Uglavnom Android aplikacije imaju jedan početni ekran koji se naziva *Main Activity* i koji se pokreće pri pokretanju aplikacije i više dodatnih koji su logički povezani sa početnim. Logika iza koje stoji ovaj koncept je da je korisniku omogućeno da pokrene različite delove aplikacije u zavisnosti od trenutnih potreba. Jedan primer koji ovo ilustruje je kada korisnik klikne na obaveštenje aplikacije za dostavu hrane da je hrana koju je naručio u putu, aplikacija će otvoriti aktivnost koja prikazuje mapu za praćenje, a ne početnu stranu za izbor restorana.

Svaka aktivnost ima četiri osnovna stanja:

1. Trajanje (eng. *running*)
2. Mirovanje (eng. *paused*)
3. Zaustavljeno (eng. *stopped*)
4. Uništeno (eng. *destroyed*)

Pri implementaciji svaka aktivnost mora da ima svoje ime i da nasleđuje klasu *Activity*. Ova klasa pruža metode koji prate osnovna stanja životnog ciklusa aktivnosti: *onCreate(), onStart(), onPause(), onResume(), onStop(), onDestroy() i onRestart()* [1]. Ove metode je potrebno predefinisati (eng. *override*) kako bi se definisala ponašanja aktivnosti za svaku promenu njenog stanja.

Metod *onCreate()* je metod u kojem se nalazi logika koja je potrebna da se izvrši pri prvom pokretanju aktivnosti. U njemu je potrebno uraditi sve inicijalizacije osnovnih komponenti aktivnosti kao i inicijalizaciju statičkih promenljivih, stavljanje podataka u liste... Iz ovog metoda mora se pozvati metod *setContentView()* koji određuje prikaz grafičkog korisničkog interfejsa. Nakon izvršavanja *onCreate()* metoda uvek se poziva metod *onStart()*.

Metod *onStart()* vodi računa o svemu što je potrebno da aktivnost bude vidljiva korisniku. Aplikacija ovde priprema aktivnost da bude prikazana korisniku. Može se registrovati prijemnik da osluškuje promene koje bi izmenile grafički korisnički interfejs. Metodi koji prate *onStart()* su *onResume()* ili *onStop()*.

Metod *onPause()* se poziva u trenutku kada se primeti da korisnik više neće koristiti tu aktivnost. S obzirom da se njeno izvršavanje dešava u momentu kada je aktivnost još uvek vidljiva korisniku sve što se izvršava u metodi mora biti brzo jer sledeća aktivnost neće biti nastavljena dok se metod ne završi. Ovde treba prekinuti sve što nije potrebno da se izvršava kada je aktivnost u stanju mirovanja. Ovaj metod prate metodi *onResume()* ukoliko se fokus vrati na ovu aktivnost ili *onStop()* ukoliko je aktivnost nevidljiva korisniku.

Metod *onResume()* se poziva kada aktivnost počinje da ima interakciju sa korisnikom.

Metod *onStop()* se poziva kada god aktivnost više nije vidljiva korisniku što može biti zbog toga što je pokrenuta nova aktivnost ili jer se trenutna aktivnost uništava. Neki od čestih primera kada se koristi implementacija ove metode su osvežavanje korisničkog interfejsa i zaustavljanje animacija ili muzike. Ukoliko se aktivnost vraća interakciji sa korisnikom pozvaće se metod *onRestart()*, u suprotnom metod *onDestroy()*.

Metod *onDestroy()* je poslednji poziv i može se desiti iz dva razloga. Prvi, jer se aktivnost završava. Drugi, da se privremeno gasi aktivnost radi čuvanja memorijskog prostora. Koji se razlog desio može se saznati pomoću metode *isFinishing()*.

Metod *onRestart()* se poziva nakon što je aktivnost stopirana, a pre njenog ponovnog prikaza. Tu možemo uraditi eventualne ponovne inicijalizacije ili neke

izmene korisničkog interfejsa pre nego što bude ponovo pozvan metod *onStart()*.

Servisi

Servis je komponenta koja izvršava svoje zadatke u pozadini i najčešće se koriste za zadatke koji se dugo izvršavaju i koji bi usporili aplikaciju ako bi se izvršavali na glavnoj niti. Servisi nemaju grafički korisnički interfejs, ali mogu da komuniciraju sa ostalim komponentama [4]. U zavisnosti od tipa zadatka koji se očekuje da servis izvrši, kao i dužine trajanja izvršavanja razlikuju se tri vrste servisa:

Pozadinski (eng. *background*) servisi ne obaveštavaju korisnika ni na koji način o zadacima koje izvršavaju zbog toga što za njihovo izvršavanje nije potrebna nikakva interakcija sa korisnikom. Primer je sinhronizovanje podataka u unapred određeno vreme.

Vidljivi (eng. *foreground*) su servisi za koje korisnici znaju da se izvršavaju tako što servis pomoću obaveštenja obaveštava korisnika o svom izvršavanju. Korisniku se daje mogućnost da pauzira ili u potpunosti zaustavi proces koji se izvršava. Primer ovog servisa je preuzimanje datoteka.

Vezani (eng. *bound*) servisi se izvršavaju kada je neka komponenta aplikacije povezana sa servisom, odnosno dokle god postoji neka komponenta kojoj je potrebno izvršavanje zadataka koje dati servis izvršava.

Na osnovu životnog ciklusa servisa razlikujemo pokrenute (eng. *started*) servise i povezane (eng. *bounded*). Kod pokrenutih servis se inicijalizuje pozivom *startService()* metode, a zaustavlja kada komponenta pozove metod *stopService()* ili ukoliko sam servis pozove metod *stopSelf()*. Povezani se mogu doživeti kao klijent-server struktura zato što komponente mogu da šalju zahteve servisu, kao i da dohvataju rezultate. U trenutku kada neka komponenta pozove metod *bindService()* i time se poveže sa servisom servis se smatra povezanim, a tek kada se sve komponente aplikacije koje su bile povezane sa njim oslobode pozivom *unbindService()* servis prestaje sa radom. Svi navedeni metodi su iz klase *Service* koju je neophodno da svaki servis nasledi pri implementaciji.

Prijemnici

Prijemnici služe da osluškuju sistemska obaveštenja kao i obaveštenja od strane drugih aplikacija na uređaju ili drugih delova iste aplikacije. Da bi mogao da izvršava svoju funkciju potrebno je da prijemnik bude registrovan da osluškuje određene namere (eng. *intent*). Moguće je da jedan prijemnik osluškuje više različitih namera i u zavisnosti od namere da izvršava različite operacije. Neki od primera upotrebe sistemskih prijemnika su prijemnik za procenat baterije, prijemnik za alarm i prijemnik za SMS poruke [14].

Provajderi sadržaja

Provajderi sadržaja obezbeđuju skladištenje podataka aplikacije. Pored samog skladištenja njihova uloga je i da omoguće drugim aplikacijama da pristupe sadržaju ukoliko imaju prava za to. Samo skladištenje je moguće da bude pomoću *SQLite* baza podataka, datoteka ili na mreži. Sa strane implementacije aplikacija koja želi da deli svoje podatke mora da koristi klasu *ContentProvider* i kreira interfejs prema tim podacima. Druga aplikacija da bi mogla da koristi te podatke mora da napravi instancu objekta klase *ContentResolver* sa svim metodama koje prva aplikacija poseduje.

Namere

Namera (eng. *intent*) predstavlja objekat koji slanjem poruke zahteva da drugi deo aplikacije ili druga aplikacija izvrši neku akciju. Najčešće su tri upotrebe namera: pokretanje aktivnosti, pokretanje servisa i slanje poruka (eng. *broadcast*). Implementacija se vrši pomoću klase *Intent* i potrebno je kreirati novi objekat [11].

AndroidManifest.xml

Glavna datoteka bez kog nijedna Android aplikacija ne može da postoji je *AndroidManifest.xml*. Pomoću ove XML datoteke OS Android i njegovi alati za izgradnju aplikacija (eng. *build tools*) dobijaju sve potrebne informacije za instalaciju i pokretanje aplikacije. Kôd počinje navođenjem verzije XML-a i enkodiranja, nakon čega sledi etiketa (eng. *tag*) *<manifest>* u okviru kog se piše ceo kôd. Obavezni deo koda je etiketa *<application>*. Pregled osnovnih elemenata i njihovih opisa može se videti

u kodu 2.1. Više informacija o elementnima i njihovim opcijama može se pronaći na vebu [13]

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
  <!-- Navodi se za svaku dozvolu koju aplikacija zahteva -->
    <uses-permission android:name="string"/>
    <!-- Definise bezednosnu dozvolu za ogranicavanje pristupa
   funkcijama ili komponentama-->
    <permission />
    <!-- Definise kompatibilnost aplikacije sa verzijama Androida
   -->
    <uses-sdk/>
    <!--Definise hardverske i softverske karakteristike koje
   aplikacija zahteva  -->
    <uses-configuration />
    <!-- Deklarise aplikaciju i sve njene elemente-->
    <application>
    <!-- Definise aktivnost -->
        <activity>
        <!-- Definise tipove namera koje aktivnost podrzava -->
            <intent-filter> . . . </intent-filter>
            <!-- Par ime vrednost za dodatne podatke koji se
   dodeljuju roditeljskoj komponenti-->
            <meta-data />
        </activity>
    <!-- Alias za aktivnost, moze imati svoje drugacije postavke u
   odnosu na aktivnost -->
        <activity-alias> . . . </activity-alias>
    <!-- Deklarise servis i njenove intent-filter i meta-data -->
        <service> . . . </service>
    <!-- Deklarise prijemnik-->
        <receiver> . . . </receiver>
    <!-- Definise provajdera sadrzaja-->
        <provider> . . . </provider>
    <!-- Definise deljenu biblioteku sa kojom aplikacija mora biti
   vezana. -->
        <uses-library />
    </application>
</manifest>
```

Listing 2.1: Primer AndroidManifest.xml, izvor: [13]

2.4 Android i STB

STB uređaji su namenjeni za pružanje digitalnih televizijskih usluga korisnicima, a koristeći OS Android, ovi uređaji mogu da pruže mnogo više funkcionalnosti. Kako OS Android pripada zajednici otvorenog koda proizvođači STB uređaja mogu lako prilagoditi sistem svojim potrebama. Takođe moguće je koristiti *Google* prodavnicu čime se broj aplikacija koje se mogu koristiti na uređajima znatno povećava. Pored ovoga moguće je pokretati svoje aplikacije koje će raditi samostalno ili u interakciji sa drugim instaliranim aplikacijama. Sigurnost aplikacija koje se kreiraju za STB uređaje je u stalnom porastu s obzirom da nove verzije OS Android donose sa sobom veću stabilnost i bezbednost, a nove verzije često izlaze.

2.5 Android i Java

2.6 Google API

Kompanija *Google* pruža skupove pravila i protokola, odnosno *Google API*-je kako bi programeri mogli da obezbede interakciju svojih aplikacija sa *Google* servisima i resursima. Zahvaljujući tome aplikacije imaju mogućnosti da pristupe podacima, funkcionalnostima i drugim resursima koje *Google* nudi. Postoji više kategorija *Google API-ja* od kojih su najpoznatiji: *Google Cloud API*, *Google Maps API*, *YouTube API* i *Google Ads API*. Spisak svih dostupnih *API*-ja sa detaljnijm opisima i uslovima korišćenja mogu se pronaći na linku: spisak *Google API*-ja. Svaki *API* služi za pružanje pristupa specifičnim funkcionalnostima i resursima kompanije. Kako bi se koristili potrebno je registrovati se na njihovom sajtu, a zatim i dobiti *API* ključ koji služi za autentifikaciju i autorizaciju zahteva. Zavisno od *API*-ja, neki zahtevi mogu biti besplatni, dok drugi mogu imati troškove u zavisnosti od količine korišćenja.

Google Cloud API-ji omogućavaju interakciju sa *Google Cloud Platform* servisima odnosno funkcionalnostima računarstva u oblaku, uključujući *Google Cloud Storage*, *Google Cloud Functions*, *Google Compute Engine* i mnoge druge. Jedan specifičan servis je *Speech-to-Text API*, odnosno govor u tekst. Servis pomoću neuronskih mreža i mašinskog učenja pretvara audio zapis u tekst. Podržava više od 120 jezika. Za sve dostupne jezike podržani osnovni model prepoznavanja i model prepoznavanja komandi i pretrage (eng. *command and search model*). Neki jezici imaju podršku i za još neke modele kao što su poboljšani poziv (eng. *enhanced*

audio call) i poboljšani video (eng. *enhanced video*). Primeri upotrebe su prepoznavanje komandi u realnom vremenu, generisanje titlova i transkripcija audio zapisa. Upravljanje *API* ključevima, naplatom usluga, *API* servisima koji su omogućeni, protokom korišćenja, kao i projektima za koje se koriste se vrše preko konzole (eng. *Google Cloud Console*). Kako bi se neki servis koristio potrebno je ispuniti sledeće korake:

1. Kreiranje projekta u konzoli ili odabir već postojećeg.
2. Omogućavanje željenog *API*-ja u konzoli ukoliko već nije omogućen.
3. Kreiranje servisnog računa u delu *Credentials* koji se koriste za autentifikaciju kada aplikacija komunicira sa *Google* servisima. U ovom koraku se generiše i privatni ključ koji se u *JSON* formatu čuva na uređaju.
4. Na računaru je potrebno instalirati Google Cloud SDK koji omogućava upravljanje resursima na *Google Cloud*-u putem komandne linije.
5. U komandnoj liniji je potrebno pokrenuti komandu

   ```
   gcloud auth activate-service-account --key-file=PUTANJA_DO_KLJUCA
   ```

Nakon ovih podešavanja moguće je koristiti odabrani *API* u svojoj aplikaciji.

Izgled grafičkog interfejsa konzole, kao i gde se nalaze prethodno navedena podešavanja u konzoli se mogu videti na linku: *Google Cloud* konzola.

Upotreba *Google Cloud API*-ja može biti malo složenija u kombinaciji sa programskim jezikom *Java* u poređenju sa nekim drugim jezicima. Iz razloga zato što programski jezik *Java* zahteva dodatne korake za generisanje klijentskih biblioteka korišćenjem *gRPC* (eng. *Google Remote Procedure Call*) i proto fajlova (eng. *Protocol Buffers*). O ovome će biti više reči u poglavljima vezanim za implementaciju aplikacije.

Kako bi aplikacija mogla da koristi metode iz *API*-ja potrebno je ostvariti konekciju sa serverom. Kreira se instanca *ManagedChannel* koja je deo *Java gRPC* biblioteke, kojoj se prosleđuju adresa i port *gRPC* servera, kao i privatan ključ raspakovan iz JSON datoteke. Moguće je dodati i presretače (eng. *interceptor*) koji pre svakog zahteva mogu da modifikuju zahtev ili odgovor. Za potrebe autentifikacije svakog zahteva putem privatnog ključa ranije generisanog koristi se *ClientAuthInterceptor*.

Glava 3

Implementacija aplikacije

U ovom poglavlju će biti opisana implementacija aplikacije *Daljinski za digitalnu televiziju*. Prvo će biti navedene i objašnjene biblioteke i softveri koje je potrebno instalirati da bi mogla da se kreira aplikacija . Nakon toga će biti opisan rad aplikacije koji uključuje način instalacije, pokretanje i korišćenje aplikacije. Zatim će biti opisana struktura projekta sa detaljnim opisom koda.

3.1 Potrebne instalacije

3.2 Opis rada aplikacije

U ovom delu će biti objašnjeni koraci za instaliranje aplikacije za korišćenje, kao i za testiranje tokom implementiranja. Zatim će biti prikazani pokretanje i upotreba aplikacije.

Instalacija

Instaliranje Android aplikacija na mobilne telefone je moguće na više načina. Najjednostavniji nacin je preuzimanje aplikacije iz *Google Play* prodavnice. Ako aplikacija nije dostupna preko prodavnice, moze se instalirati preuzimanjem datoteke u formatu **APK** (eng. *Android Package*). **APK**predstavlja format datoteka koji OS Android koristi za instaliranje i distribuciju aplikacija. Ova datoteka se kreira u *build* direktorijumu projekta nakon što se u *Android Studio*-u odabere opcija *build*. Za aplikaciju *Daljinski za digitalnu televiziju* instalaciona datoteka se može preuzeti

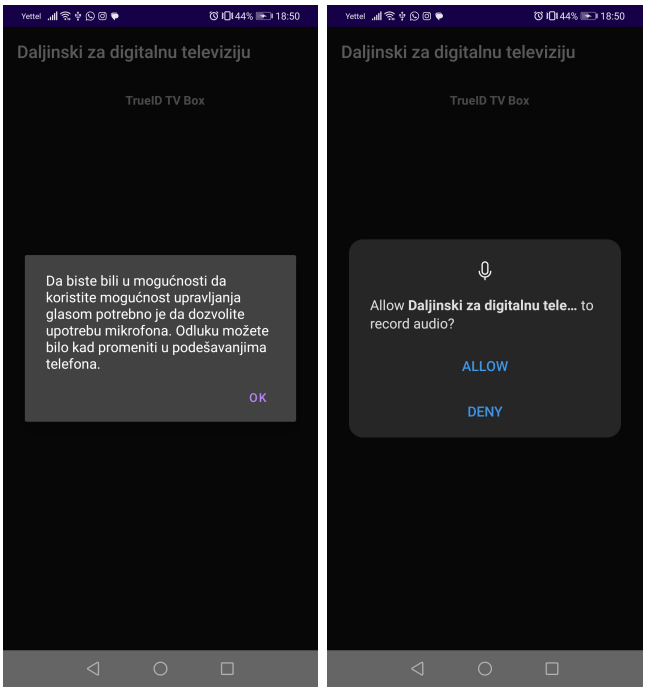
na narednom linku: daljinski.apk. Klikom na preuzetu datoteku, koja se može naći u direktorijumu gde se čuvaju preuzete datoteke, pokreće se instalacija aplikacije.

Za potrebe testiranja aplikacije tokom implementacije najčešće se koriste dva načina instaliranja. Za njih je potrebno omogućiti opcije programera (eng. *developer options*) na mobilnom uređaju i da se uređaj poveže pomoću USB kabla sa računarom na kom se nalazi kôd. Prva mogućnost pokretanja je da se u *Android Studio*-u pritisne dugme *Run*. Druga opcija je da na računaru postoji instaliran **adb** (eng. *Android Debug Bridge*) i u terminalu da se pokrene komanda `adb install app-debug.apk`.

Pokretanje i korišćenje

Nakon instalacije aplikacije prilikom prvog pokretanja prikazuje se ekran prikazan na slici **??** koji obaveštava korisnika da je potrebno dozvoliti korišćenje mikrofona. Nakon toga se prikazuje sistemsko obaveštenje o traženju dozvole sa opcijama da korisnik da dopuštenje ili ga odbije. Ovo obaveštenje može se videti na slici **??**. Ukoliko se ne da dopuštenje moguće je dati ga naknadno u podešavanjima telefona. U slučaju da ovo nije prvo pokretanje aplikacije ova dva obaveštenja neće biti prikazana. Aplikacija započinje pretragu uređaja kao na slici 3.2. Kako je aplikacija napravljena u saradnji sa jednim stranim klijentom prikazaće se samo uređaji koji imaju instaliranu njihovu aplikaciju, a nalaze se na istoj mreži. Pretraga je ograničena na 15 sekundi. Nakon isteka vremena ukoliko se ne pronađe nijedan uređaj korisnik dobija adekvatno obaveštenje sa izborom da li da se zatvori aplikacija ili ponovo pokuša traženje. Ovo je prikazano na slici 3.3. Svi uređaji koji su pronađeni se ispisuju na ekranu kao na slici 3.4 i moguće je kliknuti na bilo koji od njih. Pritiskom na naziv odgovarajućeg uređaja iskazuje se želja da bude izvršeno uparivanje sa tim uređajem. Na uređaju sa kojim se pokušava uparivanje se prikazuje četvorocifren broj kao na slici 3.6. Polje za unos tog broja se prikazuje u aplikaciji kao na slici 3.5. Nakon uspešnog unosa prikazaog na slici 3.7 uređaji se uparuju, a uspešno uparivanje potvrđuje i prikaz daljinskog upravljača. Podržane funkcionalnosti su prikazane na slici 3.11.

Dalje korišćenje aplikacije je isto kao i korišćenje fizičkog daljinskog upravljača. Pri svakom pritisku dugmeta će korisnik osetiti blagu vibraciju što ujedno obaveštava i da je dugme pritisnuto. U slučaju kada nije data dozvola za korišćenje mikrofona nije moguće zadavati komande glasom i tada su dugmići za mikrofone onemogućeni i precrtani. U suprotnom korisnik može neometano da ih koristi.



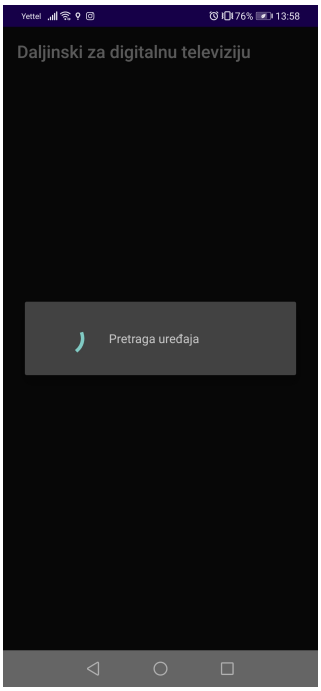
Slika 3.1: Snimci ekrana: A. obaveštenje o traženju dozvole, B prikaz sistemskog obaveštenja

Levo dugme za mikrofon koje je tamnije boje koristi *Google Cloud API*. Pri pritisku dugmeta na donjem delu ekrana pojaviće se poruka da je snimanje započeto i dugme će biti onemogućeno dokle god mikrofon sluša. Nakon pet sekundi prikazaće se poruka da je snimanje završeno, dugme će biti omogućeno i ukoliko je prepoznata komanda ona će se izvršiti.

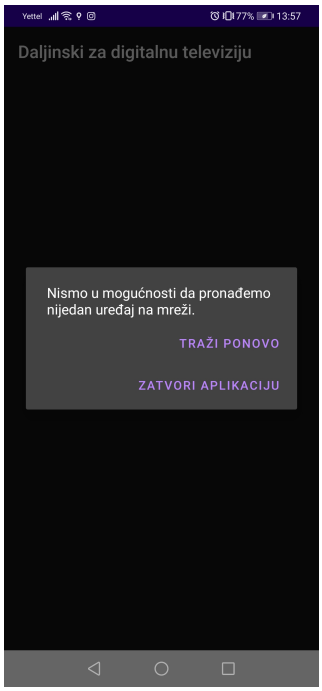
Desno dugme koje je svetlije boje pokreće mikrofon generisan na standardni način koji obezbeđuje *Google*. Pritiskom na dugme se pojavljuje polje generisano od strane *Google*-a kao na slici 3.8. Izgled tog polja pri neuspešnom slušanju je prikazan na slici 3.9, a pri uspešnom na slici 3.10. Pri uspešnom slušanju kao i u prethodnom slučaju izvršiće se zadata komanda.

Komande koje su podržane na ovaj način će biti izlistane u nastavku rada.

Ukoliko korisnik želi da prekine konekciju sa uređajem dovoljno je da pritisne dugme za otkazivanje konekcije koje ga vraća na početni ekran aplikacije. Tada će ponovo biti ivršena pretraga i izlistani pronađeni uređaji. Izlazak iz aplikacije bez prekida konekcije omogućava da korisnik ostane povezan sa uređajem i da pri sledećem pokretanju aplikacije odmah može da koristi sve funkcionalnosti bez ponovnog



Slika 3.2: Snimak ekrana, pretraga uređaja



Slika 3.3: Snimak ekrana, nisu pronađeni uređaji

povezivanja.

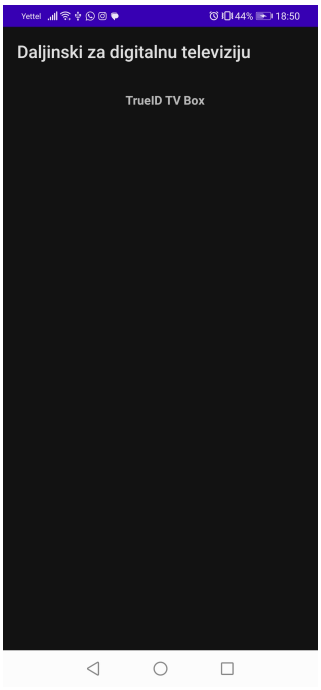
3.3 Struktura projekta

Kôd svake Android aplikacije je podeljen u dva direktorijuma: **app** i **src**. Osnovnu strukturu **app** direktorijuma bilo koje Android aplikacije čine poddirektorijumi:

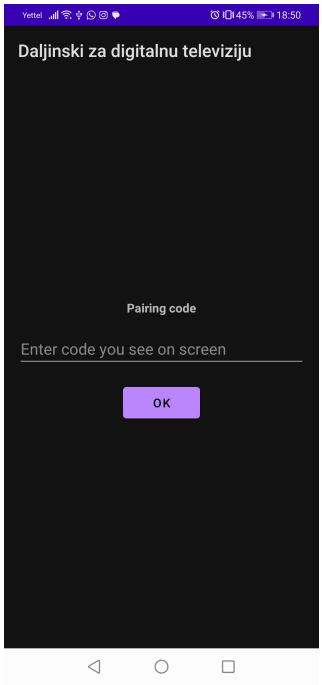
- *build* sa izvršnom verzijom koda,
- *libs* sa eksternim bibliotekama odnosno bibliotekama koje nisu deo Androida, Java ili Kotlin programskih jezika i
- *src* sa izvornim kodom.

Direktorijum **src** se uglavnom sastoji od sledećih poddiretkorijuma:

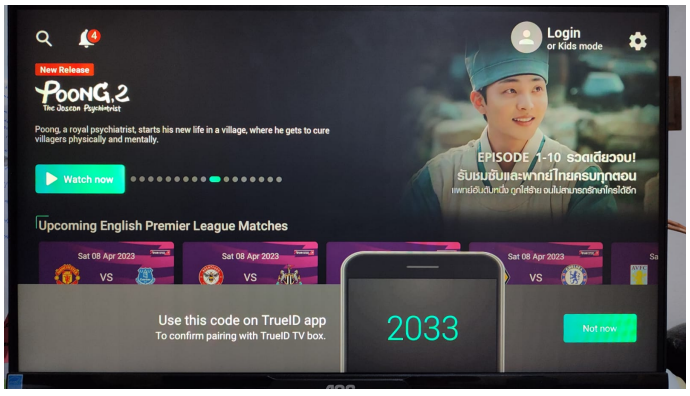
- *AndroidTest* u kom se smeštaju svi testovi koje je potrebno pokrenuti na Android uređaju,
- *test* u kom se smeštaju svi testovi za testiranje jedinica koda (eng. *unit tests*)



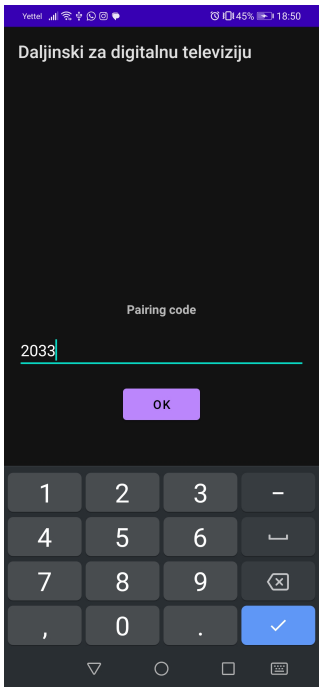
Slika 3.4: Snimak ekrana, pronađeni uređaji



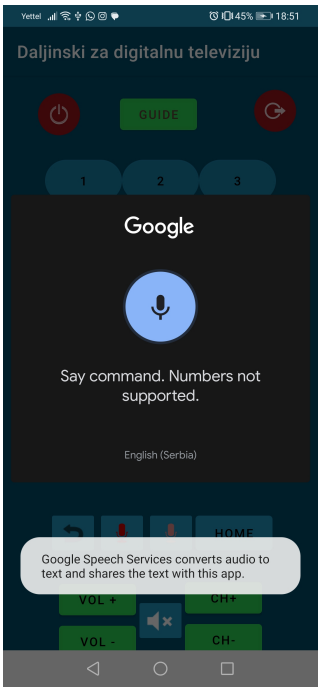
Slika 3.5: Snimak ekrana, polje za unos koda



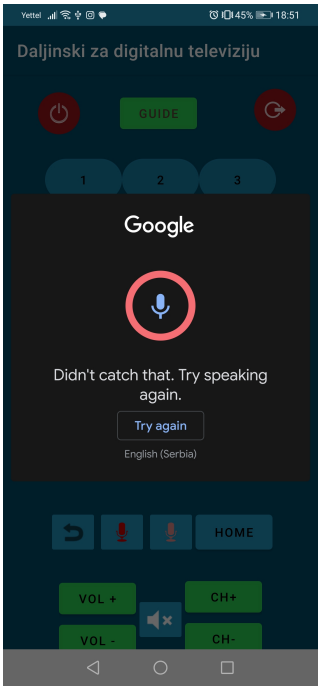
Slika 3.6: Snimak ekrana, kôd za uparivanje na uređaju



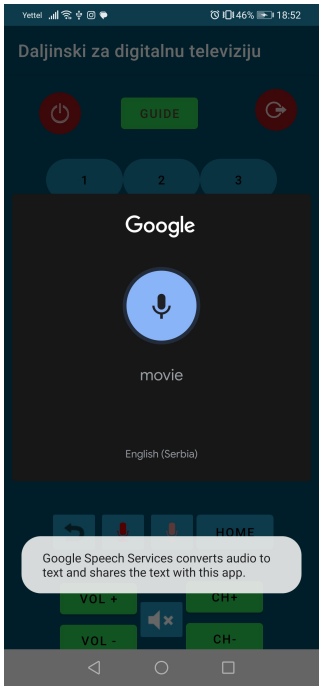
Slika 3.7: Snimak ekrana, polje za unos koda



Slika 3.8: Snimak ekrana, Google generisano polje za slušanje

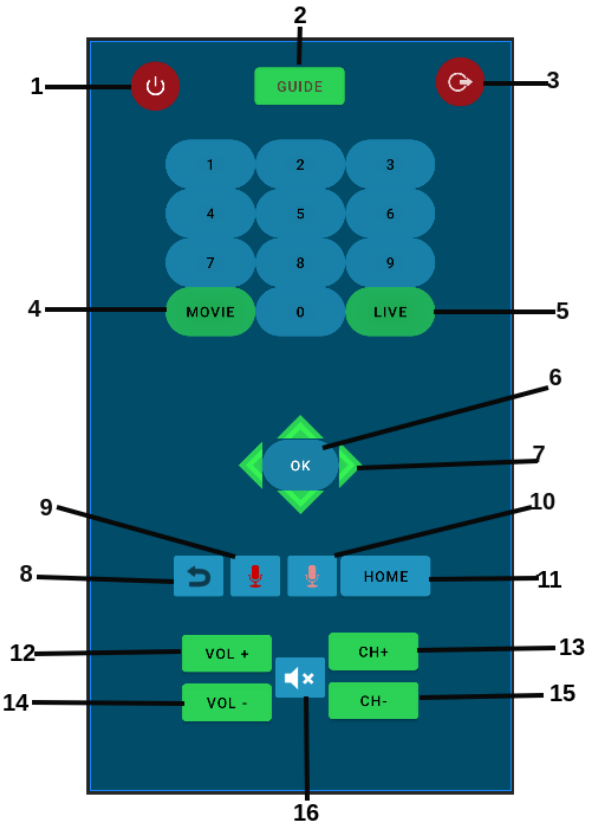


Slika 3.9: Snimak ekrana, Google generisano polje za slušanje prikaz neuspeha



Slika 3.10: Snimak ekrana, Google generisano polje za slušanje prikaz uspeha

1. Uključivanje / Isključivanje stb-a
2. Otvaranje elektronskog vodiča kroz program
3. Prekidanje konekcije
4. Prikaz dostupnih filmova
5. Puštanje kanala uživo
6. OK
7. Srelice za kretanje po aplikaciji na stb-u
8. Nazad na prethodni ekran na stb-u
9. Mikrofon pomoću Google Cloud API-ja
10. Mikrofon na standardni način Google-a
11. Povratak na početnu stranu stb aplikacije
12. Pojačavanje zvuka
13. Sledeći kanal
14. Smanjivanje zvuka
15. Prethodni kanal
16. Gašenje zvuka



Slika 3.11: Snimak ekrana, opis komandi

- *main* u kom se nalazi ceo izvorni kôd.

Projekat implementacije upravljača za digitalnu televiziju je smešten u direktorijumu pod nazivom *RemoteControlApp*. Srž aplikacije se nalazi u **main** poddirektorijumu, a za ovu aplikaciju ovaj direktorijum ima sledeću strukturu:

- *AndroidMainifest.xml* — datoteka koji opisuje glavne postavke aplikacije,
- *res* — direktorijum sa svim XML datotekama koje čine korisnički interfejs (eng. *user interface*) aplikacije,
- *java* — direktorijum sa svim klasama i interfejsima aplikacije
- *proto* — direktorijum sa *.proto* datotekama koje su neophodne za korišćenje *Google Cloud API*-ja.

3.4 Upravljanje procesom izdgradnje i određivanje zavisnosti aplikacije

Važne datoteke pri postavljanju projekta su ***build.gradle*** datoteke koje služe da upravljaju procesom izgradnje i odrede zavisnosti (eng. *dependency*) aplikacije. Ove datoteke se generišu automatski pri kreiranju projekta, ali je moguće dodati sve dodatne zavisnosti koje budu potrebne. Postoje dva tipa ovih datoteka — na nivou projekta i na nivou modula. Datoteku na nivou projekta čini skup pravila koji važi za ceo projekat, dok datoteke na nivou modula čine pravila za modul u kom se nalazi datoteka. Svaka ova datoteka se sastoji od vise blokova koji grupišu unutar sebe pravila, odnosno opcije koje se primenjuju.

Za uspešno korišćenje *Google Cloud* servisa u aplikaciji, neophodno je uvesti podršku za obradu *Protobuf* (skraćeno od eng. *Protocol Buffers*) datoteka. *Protobuf* je interfejs za definiciju jezika (eng. *Interface Definition Language*, skraćeno IDL) koji definiše strukturu podataka i programski interfejs. [9] Omogućava usaglašeost pri deljenju struktuiranih podataka između različitih sistemskih platformi i jezika programiranja. Neophodno je unutar bloka *plugins* dodati dodatak `id 'com.google.protobuf'` kako bi se obezbedila adekvatna podrška. Takođe u okviru bloka *protobuf*, se dodaju opcije za korišćenje protobuf-a. Tačne zavisnosti koje je potrebno dodati biće navedene kod opisa implementacije glasovnih komandi gde će biti i objašnjena povezanost *Google cloud* servisa i *protobuf*-a.

3.5 Potrebne dozvole i informacije za pokretanje aplikacije

Informacije potrebne za pokretanje i instalaciju aplikacije čine datoteku *AndroidManifest.xml*. U ovoj datoteci su definisane sve dozvole koje aplikacija zahteva od uređaja sa kog se pokreće aplikacija: pristup internetu, stanju bežične mreže (eng. *WiFi*), stanju telefona, vibracija i snimanje audio sadržaja. Pregled ovih dozvola je dat u listingu 3.1. Kao što je navedeno u delu 2.3 glavna etiketa koja mora postojati je za aplikaciju i u okviru nje su postavljene vrednosti naziva aplikacije, slika kojom je aplikacija predstavljena, ciljani API nivo kao i dve aktivnosti koje se pojavljuju. Prva aktivnost je *ChooseConnection*, koja je odabir uređaja sa kojim će se korisnik povezati i ona je obeležena kao glavna aktivnost. Druga aktivnost je *RemoteView*

koja čini prikaz daljinskog upravljača.

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE
    "/>
<uses-permission android:name="android.permission.READ_PHONE_STATE"
    />
<uses-permission android:name="android.permission.VIBRATE" />
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
```

Listing 3.1: Odobrenja definisana u datoteci *AndroidManifest.xml*

3.6 Resursi aplikacije

Direktorijum u kom se smeštaju svi resursi aplikacije se naziva *res*. Resursi koji se mogu čuvati su slike, planovi (eng. *layout*), stringovi, stilovi itd. Moguće je čuvati iste resurse u različitim dimenzijama kako bi u zavisnosti od dimenzija i podešavanja uređaja bili odabrani odgovarajući resursi. Direktorijum *drawable* skladišti slike. Za potrebe projekta formirane su i sačuvane slike za strelicu koja je predstavljena trouglom, ovalno dugme i dugme za prekid konekcije.

Vrednosti, boje i dimenzije za različite resurse se skladište u direktorijumu *values*. Datoteka *colors.xml* sadrži boje korišćene za kreiranje interfejsa. Stringovi potrebni za dizajn korisničkog interfejsa su sačuvani u *strings.xml*. Kolor shema, odnosno tema aplikacije je definisana u *themes/themes.xml* odakle se može uočiti da boje koje se koriste za korisnički interfejs su nijanse plave boje.

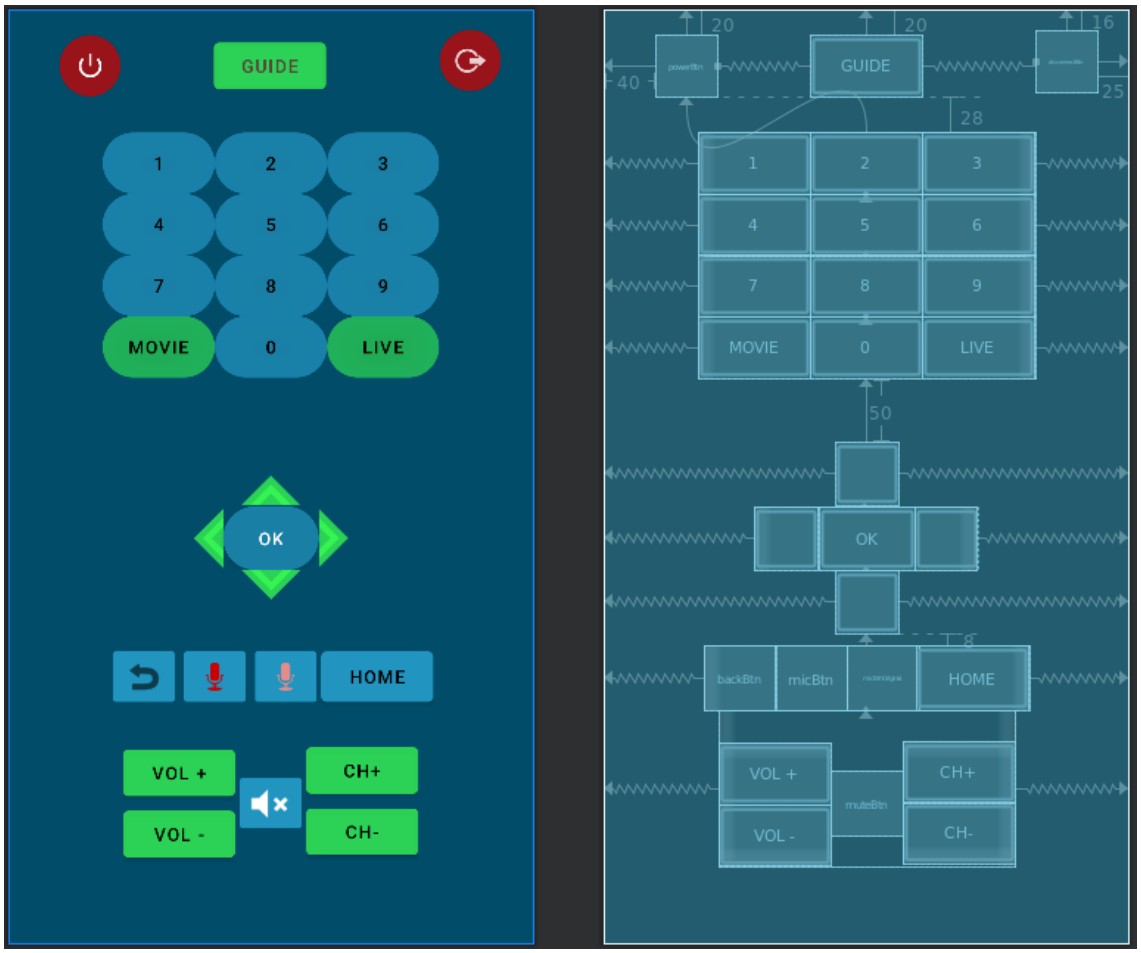
Svaki ekran sa kojim se korisnik može susresti mora imati korisnički interfejs koji ima svoj plan opisan unutar XML datoteke. Skup svih planova je smešten unutar direktorijuma *layout* koji kod ove aplikacije ima sledeće XML datoteke:

activity_choose_stb predstavlja plan ekrana za odabir konekcije, odnosno stb uređaja sa kojim korisnik želi da se poveže. Ovaj plan je kreiran pomoću klase *RelativeLayout*. Unutar *RelativeLayout*-a nalaze se tri komponente koje se mogu prikazati ili skloniti u zavisnosti od potrebe. Prva komponenta je *RecyclerView* sa id-jem *@+id/rv_boxes_list* sa podešenom vidljivošću na *gone* što znači da je inicijalno skriven. Njegova vidljivost se menja u kodu u trenutku kada se pronađu uređaji na mreži tako što se postavlja na *visible*, odnosno da je vidljiv. Sledeća komponenta je *ProgressBar* koji može poslužiti prilikom učitavanja. Poslednja komponenta je *RelativeLayout* sa id-jem

@+id/rl_pairing_container koji je inicijalno sakriven. On sadrži tekstualno polje tipa *TextView* sa uputstvom korisniku da unese kôd koji vidi na ekranu uređaja sa kojim pokušava uparivanje, polje za unos teksta tipa *EditText* u koji se unosi kôd i dugme za potvrdu unosa. Ceo prikaz za uparivanje postaće vidljiv kada korisnik odabere uređaj sa kojim želi da se poveže. Prikaz ovih elemenata je pokriven u opisu rada aplikacije.

remote_control_scene predstavlja izgled daljinskog upravljača. Sve komponente su unutar *ConstraintLayout*-a koji omogućava da udaljenost komponenti bude predstavljena ograničenjima (eng. *constraint*). Ovakvo predstavljanje kompontenti olakšava prilagođavanje prikaza ekrana na uređajima različitih dimenzija. Korisnički interfejs ovog plana kao i shematski plan zajedno sa svim ograničenjima se može videti na slici 3.12. Dugmići za kontrolu uključenosti uređaja, prekid konekcije sa povezanim uređajem, vraćanje nazad, gašenje zvuka i mikrofoni su objekti tipa *ImageButton*. Svi ostali dugmići su kreirani pomoću klase *Button*. Nad svim dugmićima je postavljena opcija da se na pritisak dugmeta pozove metoda *onClick()*. O akcijama koje se dese prilikom pritiska dugmeta i poziva metode *onClick* biće reči u nastavku poglavlja. U prvom nizu su dugmići za paljenje/gašenje uređaja, zatim dugme *GUIDE* koje prikazuje elektronski vodič kroz program (eng. *Electronic Program Guide (EPG)*) i na kraju dugme za prekid konekcije između uređaja. Dugmići za brojeve su podeljeni u četiri horizontalna *LinearLayout*-a sa po tri dugmeta. Poslednji od njih sadrži i dugme *MOVIE* koje prikazuje obabir filmova za iznajmljivanje i dugme *LIVE* koje prebacuje ekran na sadržaj koji je uživo na tv-u. Centralni deo ekrana zauzima dugme za potvrdu izbora *OK* i strelice za levo, desno, gore i dole. Strelice su kreirane samostalno i u implementaciji je zato potrebno dodati `android:background="@drawable/arrow"`. Dugmići za povratak na prethodno prikazano, mikrofoni i *HOME* - prikaz početnog ekrana su grupisani u jedan *LinearLayout*. Na samom dnu ekrana se nalazi još jedan *LinearLayout* koji se sastoji od dva vertikalna *LinearLayout*-a, za menjanje kanala i podešavanje jačine zvuka. Između njih je dugme za totalno gašenje zvuka.

stb_view je jedan *RelativeLayout* koji čini samo jedan *TextView*. On se ubacuje unutar *RecyclerView*-a kada je potrebno prikazazi pronađene uređaje. Predstavlja ime jednog uređaja koji je pronađen. Inicijalno nema nikakav tekst.



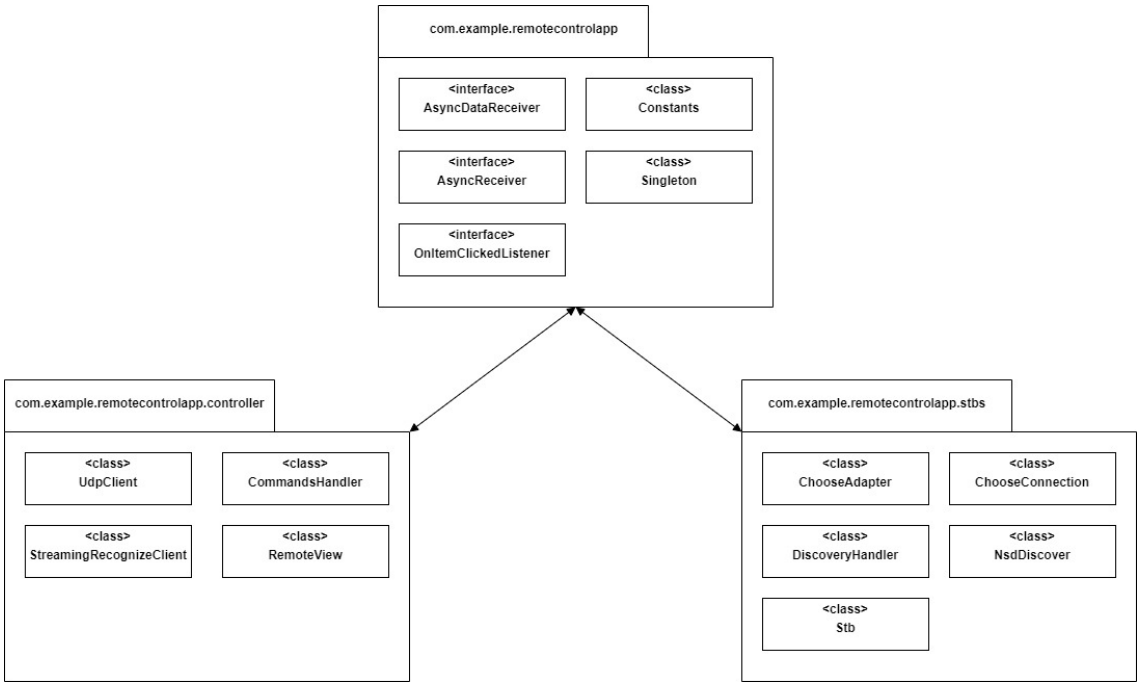
Slika 3.12: Korisnički interfejs i shematski plan daljinskog upravljača

3.7 Struktura direktorijuma java

Za funkcionisanje aplikacije neophodno je da se omogući pronalaženje uređaja, a zatim i da se manipuliše sa odabranim uređajem. Kako bi ovo sve radilo ispravno potrebna je međusobna interakcija između klasa, kao i podela koda u adekvatne pakete i klase prema funkcionalnostima koje obezbeđuju. Iz tih razloga unutar glavnog paketa aplikacije kôd je podeljen u dva paketa. Paket **stbs** sadrži klase koje se bave pronalaženjem i upravljanjem STB uređajem. Klase koje se bave upravljanjem komandama i komunikacijom sa uređajem su smeštene u paket **controller**. Dijagram paketa je prikazan na slici 3.13

Klase unutar paketa **stbs** su:

- *ChooseAdapter* koja se koristi za prikaz liste pronađenih uređaja,



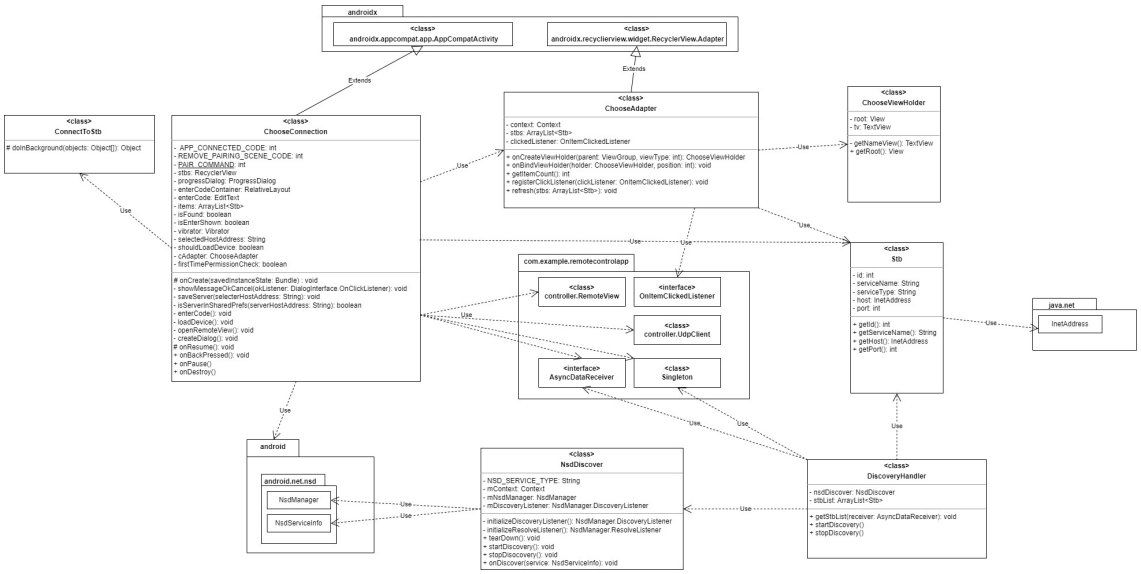
Slika 3.13: Dijagram paketa

- *ChooseConnection* koja se koristi za odabir i povezivanje sa izabranim uređajem,
- *DiscoveryHandler* se koristi za pronalaženje dostupnih uređaja na mreži,
- *NsdDiscover* koja se koristi za korišćenje NSD (eng. *Network Service Discovery*) mehanizma za pronalaženje uređaja i
- *Stb* koja predstavlja jedan uređaj i sadrži informacije o njemu.

Dijagram klasa ovog paketa se može videti na slici 3.14.

Klase unutar paketa **controller** su:

- *CommandsHandler* koja je odgovorna za obradu i slanje komandi na uređaj,
- *RemoteView* koja se bavi prikazom daljinskog upravljača i interakcijom korisnika sa aplikacijom,
- *UdpClient* koja omogućava komunikaciju putem UDP protokola i
- *StreamingRecognizeClient* koje se koristi za obradu audio podataka i prepoznavanje govora pomoću *Google Cloud API*-ja.



Slika 3.14: Dijagram klasa za paket stbs

Komunikacija i izgled ovih klasa su prikazani na dijagramu 3.15.

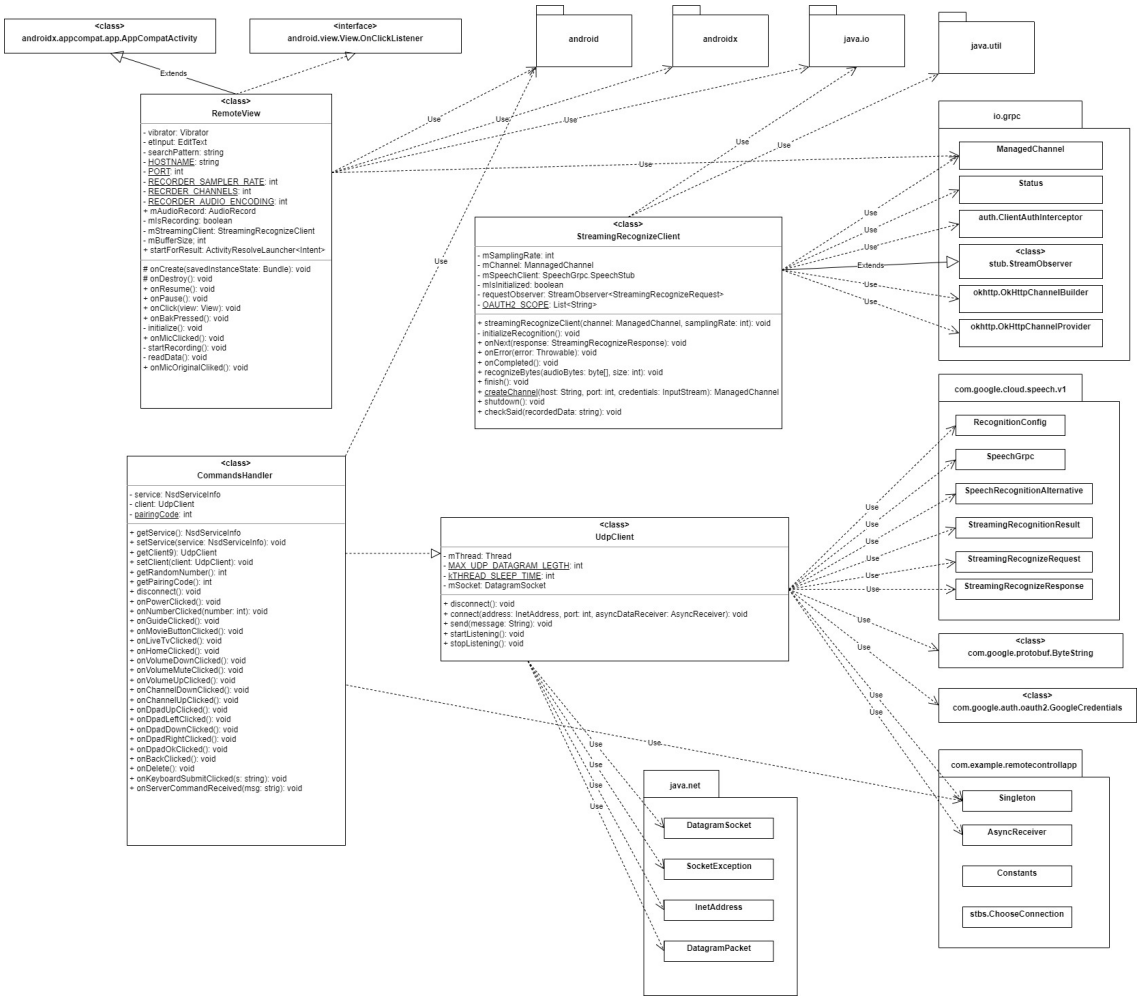
Pored ovih klasa u glavnom paketu aplikacjie nalaze se sledeće klase i interfejsi:

- *AsyncDataReceiver* interfejs služi za primanje asinhronih podataka,
- *AsyncReceiver* interfejs služi za primanje asinhronih podataka,
- *OnItemClickedListener* interfejs služi za obradu klikova na elemente liste,
- *Singleton* klasa se koristi za implementaciju Singlton (eng. *Singleton*) šablona i
- *Constants* klasa koja skladišti sve konstante potrebne u razvoju aplikacije.

NJihov dijagram klasa je prikazan na slici 3.16.

3.8 Implementacija glavnih funkcionalnosti aplikacije

Kao što je navedeno u delu 3.1 postoje koraci koji su potrebni da se ispune kako bi se aplikacija koristila za upravljanje uređajem za digitalnu televiju. U nastavku će biti prikazani delovi koda i objašnjenja kako implementirati pretragu uređaja na

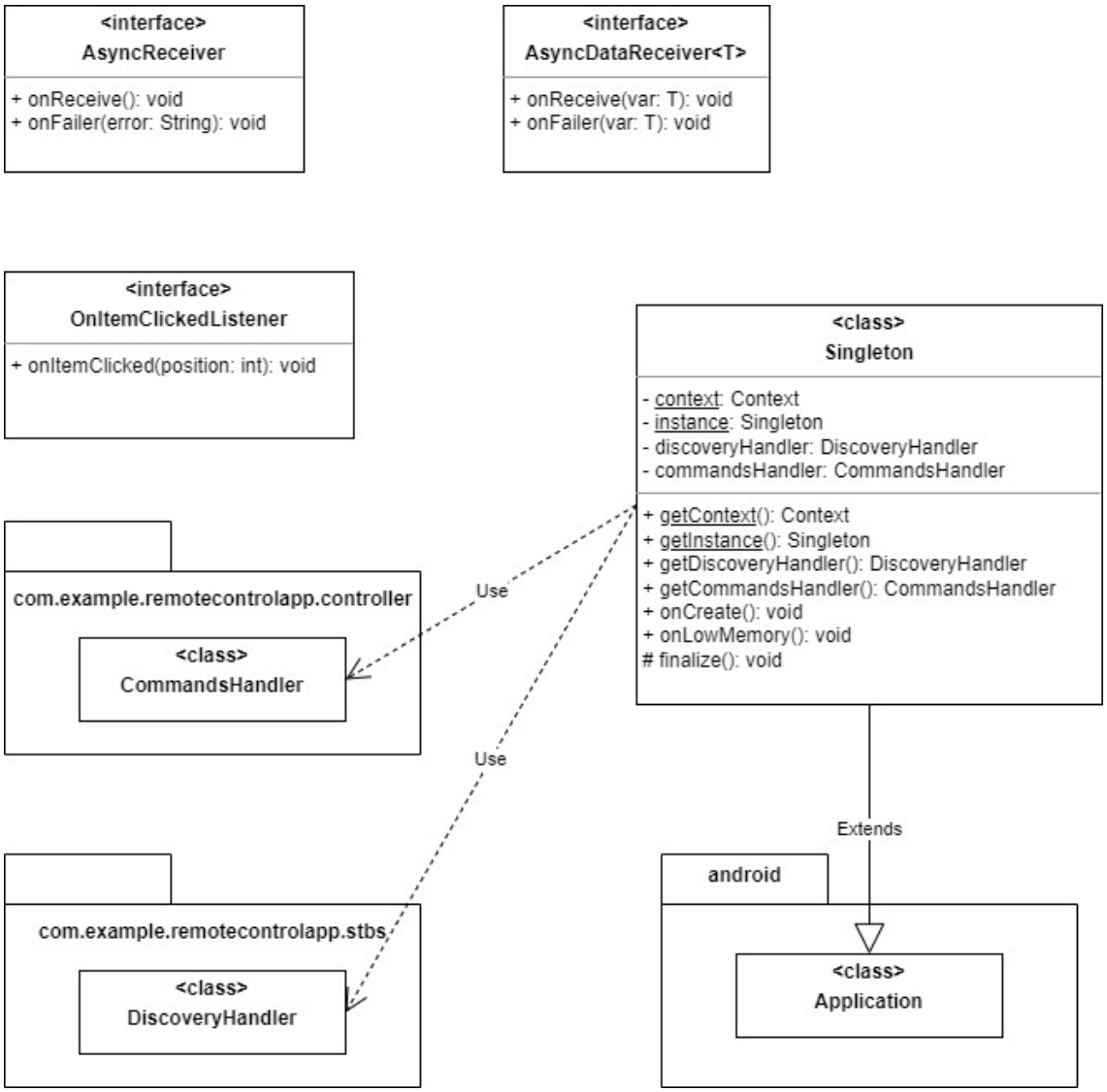


Slika 3.15: Dijagram klasa za paket controller

mreži, povezivanje sa odabranim uređajem kao i komunikaciju i slanje komandi na uređaj. Poseban osvrt će biti na implementaciji zadavanja glasovnih komandi.

Implementacija pretrage uređaja

Za pretragu uređaja koji se nalaze na istoj mreži koristi se klase iz paketa `android.net.nsd`. Otkrivanje mrežnih servisa (eng. *Network Service Discovery (NSD)*) [8] obezbeđuje klase koje pružaju usluge pronalaska svih servisa koje pružaju uređaji na lokalnoj mreži. Potrebno je kreirati jednu klasu u kojoj će se koristiti mogućnosti za otkrivanje koje pruža ova biblioteka i koja je nazvana *NsdDiscover*. Pored ove klase potrebno je napraviti klasu *DiscoveryHandler* koja koristi prethodno kreirane metode u svrhu upravljanja pretragom i rezultatima.



Slika 3.16: Dijagram klasa za glavni paket aplikacije

Unutar klase *NsdDiscover* se inicijalizuju osluškivač pretrage (eng. *discovery listener*) i osluškivač rezultata (eng. *resolve listener*). Oba predstavljaju instance klasa iz menadžera za otkrivanje mrežnih servisa (eng. *NsdManager*) i imaju svoje predefinisane metode koje je potrebno prepisati. U listingu 3.2 su prikazani koraci za inicijalizaciju ova dva osluškivača.

```
1 NsdManager.DiscoveryListener listener = new NsdManager.
    DiscoveryListener() {
2
3     // Poziva se cim se zapocne pretraga i dohvataju se informacije
      o lokalnoj mrezi
```

```
4     public void onDiscoveryStarted(String regType) {
5         WifiManager wifiMgr = (WifiManager)              mContext.
    getSystemService(Context.WIFI_SERVICE);
6         WifiInfo wifiInfo = wifiMgr.getConnectionInfo();
7         String ipAddress = wifiInfo.toString();
8     }
9
10    //Definisu se akcije koje se izvrsavaju kada se pronade zadati
    servis
11    public void onServiceFound(NsdServiceInfo service) {
12      String type = service.getServiceType();
13      //Ako se tip servisa podudara sa servisom koji se trazi, kao
    i sa imenom poziva se metod. Potrebno je obraditi i slucajeve
    ako se ne poklapaju.
14      mNsdManager.resolveService(service, initializeResolveListener
    ());
15    }
16    // Kada servis na mrezi vise nije dostupan poziva se ova metoda
    .
17    public void onServiceLost(NsdServiceInfo service) {}
18
19    // Kada se zaustavi pretraga poziva se ova metoda.
20    public void onDiscoveryStopped(String serviceType) {}
21
22    // Ukoliko nakon nekog vremena nije bilo moguce da se pokrene
    otkrivanje ovaj metod ce biti pozvan.
23    public void onStartDiscoveryFailed(String serviceType, int
    errorCode) {
24      mNsdManager.stopServiceDiscovery(this);
25    }
26
27    // Ukoliko nakon nekog vremena nije bilo moguce da se zaustavi
    otkrivanje ovaj metod ce biti pozvan.
28    public void onStopDiscoveryFailed(String serviceType, int
    errorCode) {
29      mNsdManager.stopServiceDiscovery(this);
30    }
31 }
32
33  NsdManager.ResolveListener listener = new NsdManager.
    ResolveListener() {
34    //Ukoliko je razresavanje bilo neuspesno
```

```
35   public void onResolveFailed(NsdServiceInfo nsdServiceInfo, int
      errorCode) {}
36
37   //Ukoliko je razresavanje bilo uspesno
38   public void onServiceResolved(NsdServiceInfo nsdServiceInfo) {}
39 }
```

Listing 3.2: Inicijalizacija osluškivača pretrage i rezultata

Potrebno je definisati i jednu funkciju povratnog poziva (eng. *callback*) `public void onDiscover(` koja će biti prepisana u klasi *DiscoveryHandler* prilikom kreiranja instance klase *NsdDiscover*. Klasa *DiscoveryHandler* je kreirana da bi pružila logiku koja će se izvršavati za pretragu. Ovo je prikazano u listingu 3.3 putem definicije funckije koja kreira listu STB uređaja `getStbList`.

```
1  public void getStbList(final AsyncDataReceiver receiver){
2          //Potrebno je obezbediti da pretraga nije u toku
3          nsdDiscover.stopDiscovery();
4
5          nsdDiscover = new NsdDiscover(Singleton.getContext()){
6              public void onDiscover(NsdServiceInfo service) {
7                  //Sva logika vezana za pronadjene uredjaje
8                  //se smesta unutar ove funkcije:
9                  //kreiranja nove instance stb uredjaja,
10                 //dodavanje uredjaja u listu,
11                 //provera da vec nije isti uredjaj u listi...
12         };
13         nsdDiscover.startDiscovery();
14     }
```

Listing 3.3: Metod klase DiscoveryHandler za pretragu uređaja

Implementacija povezivanja sa odabranim uređajem

Nakon uspešnog pronalaska uređaja na mreži i prikaza liste na ekranu korisnika potrebno je obezbediti da se korisnik klikom na odabrani uređaj poveže sa istim. Za izvršenje ovog zadatka prvenstveno je potrebno obezbediti klasu koja omogućava komunikaciju korišćenjem UDP protokola (eng. *User Datagram Protocol*) iz razloga što servis koji se za potrebe ove aplikacije poržava UDP protokol. Glavne funkcionalnosti za UDP komunikaciju su prikazane ulistingu 3.4.

```
1  public class UdpClient {
```

```
2
3   private DatagramSocket mSocket;
4
5   public void disconnect(){
6    //Zatvaranje mSocket-a ukoliko postoji
7   }
8
9  public void connect(final InetAddress address, final int port,
      final AsyncReceiver asyncDataReceiver){
10   disconnect();
11
12    new Thread(() -> {
13     try {
14      mSocket = new DatagramSocket(0);
15    } catch (SocketException e) {
16      e.printStackTrace();
17    }
18    mSocket.connect(address, port);
19
20    if (mSocket.isConnected()) {
21     asyncDataReceiver.onReceive();
22    } else {
23     asyncDataReceiver.onFailed("Socket not connected");
24    }
25    }).start();
26  }
27
28  //Funkcija koja salje komande na uredjaj
29  public void send(final String message) {
30  //...
31  }
32
33  public void startListening() {
34   stopListening();
35
36   mThread = new Thread(() -> {
37    while (!Thread.currentThread().isInterrupted()) {
38
39    byte[] buf = new byte[MAX_UDP_DATAGRAM_LEN];
40    final DatagramPacket pack = new DatagramPacket(buf, buf.length)
      ;
41
```

```
42      if (mSocket != null) {
43       if (mSocket.isConnected()) {
44         try {
45           // Primanje poruke od uredjaja
46           mSocket.receive(pack);
47           String msg = new String(pack.getData(), pack.getOffset(),
     pack.getLength());
48           msg = msg.trim();
49           Singleton.getInstance().getCommandsHandler().
     onServerCommandReceived(msg);
50         } catch (IOException e) {
51            e.printStackTrace();
52         }
53      }
54    }
55      //...
56    });
57    mThread.start();
58  }
59  public void stopListening() {
60         //Prekid rada niti
61      }
62 }
```

Listing 3.4: Klasa UdpClient

Takođe, kreirana je i jedna unutrašnja klasa koja izvršava asinhroni zadatak u pozadini. `doInBackground` je funkcija koja pomoću klijenta za UDP protokol povezuje sa uređajem na osnovu informacija koje prethodno dobila o njemu. U slučaju da uređaj uparen sa korisnikovim mobilnim uređajem potrebno je poslati komandu za uparivanje o čemu će biti više reči u nastavku. Implementacija ove unutrašnje klase nazvane *ConnectToStb* se nalazi u listingu 3.5.

```
1 private class ConnectToStb extends AsyncTask {
2  @Override
3  protected Object doInBackground(Object[] objects) {
4   //Pozicija u listi pronadjenih uredjaja
5   int position = (int) objects[0];
6   final UdpClient client = new UdpClient();
7   final InetAddress inetAddress = items.get(position).getHost();
8   client.connect(inetAddress, items.get(position).getPort(), new
     AsyncReceiver() {
9    @Override
```

```
10    public void onReceive() {
11     //Cuvanje instance klijenta sa kojim se povezujemo
12     Singleton.getInstance().getCommandsHandler().setClient(client);
13     //Ukoliko nemamo sacuvanu konekciju sa datim uredjajem
14     if (!isServerInSharedPrefs(inetAddress.getHostAddress())) {
15       //Slanje komande za povezivanje
16       Singleton.getInstance().getCommandsHandler().getClient()
17                .send(PAIR_COMMAND + "@" + Singleton.getInstance().
    getCommandsHandler()
18                .getRandomNumber());
19       //Potrebno je otvoriti dijalog za unos i obradu koda
20     } else {
21       //Ukoliko je sacuvana konekcija sa datim klijentom dovoljno
22       //je poslati kod uredjaju da je aplikacija povezana sa njim
23       //i pozvati funkciju koja menja trenutnu aktivnost za
24       //aktivnost sa planom daljinskog upravljaca
25     }
26     //UDP klijent osluskuje poruke koje se salju
27     client.startListening();
28    }
29   // ...
30 }
```

Listing 3.5: Klasa ConnectToStb

Implementacija komunikacije sa uređajem i zadavanja komandi

Komunikacija aplikacije sa STB uređajem je implementirana putem UDP protokola kao što je prikazano u prethodnom poglavlju. Za slanje komandi implementirana je funkcija `send(String message)` koja kreira novu nit u kojoj bajtove poruke pakuje u jedan *DatagramPacket*[7] i preko soketa šalje povezanom uređaju. Preduslov da ova funckija radi je da je soket povezan, odnosno da je komunikacija ostvarena za zadatu mrežu i port. Implementacija funkcije je zadata u listingu 3.6.

```
1 public void send(final String message) {
2   new Thread(() -> {
3    byte[] buf = message.getBytes();
4    DatagramPacket p = new DatagramPacket(buf, buf.length);
5
6    try {
```

```
7       if (mSocket.isConnected()) {
8           mSocket.send(p);
9           Log.d(TAG, "[send] sending data to stb - mSocket is
        connected");
10      } else {
11          Log.d(TAG, "[send] sending data to stb - mSOCKET not
        connected");
12      }
13    } catch (IOException e) {
14          Log.e(TAG, "[send] " + e.getMessage());
15          e.printStackTrace();
16      }
17    }).start();
18  }
```

Listing 3.6: Funkcija za slanje komandi preko UDP protokola

Poruka koja se šalje predstavlja konstantu za klik dugmeta na daljinskom upravljaču. U Androidu događaj tastature (eng. *Key Event*) je događaj kada korisnik pritisne taster na tastaturi ili nekom od podržanih uređaja. Svaki od tastera ima svoju numeričku vrednost, odnosno kôd tastature (eng. *Key Code*). Oni zajedno olakšavaju interakciju sa uređajima koji koriste neki od podržanih uređaja. Nije potrebno znati vrednosti ovih konstanti napamet jer im se može pristupiti uključivanjem paketa `android.view.KeyEvent`.

Može se napraviti klasa koja upravlja komandama sa imenom *CommandsHandler*. I u njoj implementirati funkciju za svako dugme koje je moguće pritisnuti. Kada se u glavnoj klasi pritisne dugme na primer za odlazak na početni ekran potrebno je pozvati funkciju na sledeći način:

```
Singleton.getInstance().getCommandsHandler().onHomeClicked();
```

S obzirom da je kreirana Singleton instanca aplikacije potrebno je dohvatiti tu instancu i nad njom instancu klase koja upravlja komandama i zatim i funkciju koja obrađuje traženi zahtev. Linija koda koja se izvršava pri ovom pozivu iz klase *CommandsHandler* je

```
getClient().send(String.valueOf(android.view.KeyEventKEYCODE_HOME));
```

Standardni način implementacije prepoznavanja govora

Kao standardni način za prepoznavanje glasovnih komandi koji je obezbeđen od strane *Google*-a se smatra upotreba klase *Recognizer Intent*. Ova klasa je deo *Speech*

Recognizer API-ja ugrađenog u Android, a sve metode koje su definisane u njemu je potrebno izvršavati na glavnoj niti (eng. *Main Thread*). Da bi se pokrenuo proces prepooznavanja govora kreira se namera sa akcijom `RecognizerIntent.ACTION_RECOGNIZE_SPEECH`. Ova naredba pokreće aktivnost koja sluša korisnikov govor i prepoznaje ga. *Recognizer Intent* pruža korisne opcije kojima se može precizirati kako sistem za prepoznavanje govora treba da se ponaša i kako komunicira sa korisnikom. Neke od opcija su:

1. **EXTRA_LANGUAGE_MODEL** koja se koristi za odabir modela jezika za prepoznavanje govora. Jedan primer je **LANGUAGE_MODEL_FREE_FORM** koji se preporučuje za prepoznavanje slobodnog stila govora.
2. **EXTRA_PROMPT** koja omogućava definisanje poruke koja će se prikazati korisniku prilikom slušanja.
3. **EXTRA_MAX_RESULTS** koja omogućava ograničavanje maksimalnog broja rezulata koje će vratiti.

Namera koja je kreirana se koristi u paru sa klasom *ActivityResultLauncher*. Instanca se registruje u kodu pozivanjem metode *registerForActivityResult* koja kao argumente prima *ActivityResultContract* koji definiše ulazne i izlazne tipove i funkciju povratnog poziva koja prima izlaz. Pozivanjem metoda *launch* sa argumentom definisane namere se pokreće pretraga i po završetku aktivira funkcija povratnog poziva koja obrađuje rezultat.

Prepoznavanje govora pomoću Google računarstva u oblaku

Poređenje dostupnih rešenja

Glava 4

Zaključak

Bibliografija

[1] Rick Boyer. *Android 9 Development Cookbook, Third Edition.* Packt, 2018.

[2] BusinessOfApps David Curry. Android Statistics (2022), 2022. on-line at: `https://www.businessofapps.com/data/android-statistics/`.

[3] Kotlin Foundation. Kotlin programski jezik. on-line at: `https://kotlinlang.org/docs/android-overview.html`.

[4] Erik Hellman. *Android Programming, Pushing the Limits.* Wiley, 2014.

[5] Darren Cummings Iggy Krajci. *Android on x86, An Introduction to Optimiying for Intel Architecture.* Apress, 2013.

[6] Nemanja Lukić Ištvan Papp. *Projektovanje i arhitekture softverskih sistema: Sistemi zasnovani na Androidu.* FTN Izdavaštvo, 2015.

[7] Google LLC. Datagram Packet. on-line at: `https://developer.android.com/reference/java/net/DatagramPacket`.

[8] Google LLC. Network Service Discovery. on-line at: `https://developer.android.com/training/connect-devices-wirelessly/nsd`.

[9] Google LLC. Protocol Buffers. on-line at: `https://cloud.google.com/apis/design/proto3`.

[10] Google LLC. Dalvik VM, 2020. on-line at: `https://source.android.com/devices/tech/dalvik`.

[11] Google LLC. Android Developers, 2022. on-line at: `https://developer.android.com/`.

[12] Google LLC. Android Developers Arhitektura, 2022. on-line at: `https://developer.android.com/guide/platform`.

[13] MIT. Android Manifest. on-line at: `https://stuff.mit.edu/afs/sipb/project/android/docs/guide/topics/manifest/manifest-intro.html`.

[14] Miodrag Živković. *Razvoj mobilnih aplikacija, Android Java programiranje*. Univerzitet Singidunum, 2020.

Biografija autora

Vuk Stefanović Karadžić (*Tršić, 26. oktobar/6. novembar 1787. — Beč, 7. februar 1864.*) bio je srpski filolog, reformator srpskog jezika, sakupljač narodnih umotvorina i pisac prvog rečnika srpskog jezika. Vuk je najznačajnija ličnost srpske književnosti prve polovine XIX veka. Stekao je i nekoliko počasnih mastera. Učestvovao je u Prvom srpskom ustanku kao pisar i činovnik u Negotinskoj krajini, a nakon sloma ustanka preselio se u Beč, 1813. godine. Tu je upoznao Jerneja Kopitara, cenzora slovenskih knjiga, na čiji je podsticaj krenuo u prikupljanje srpskih narodnih pesama, reformu ćirilice i borbu za uvođenje narodnog jezika u srpsku književnost. Vukovim reformama u srpski jezik je uveden fonetski pravopis, a srpski jezik je potisnuo slavenosrpski jezik koji je u to vreme bio jezik obrazovanih ljudi. Tako se kao najvažnije godine Vukove reforme ističu 1818., 1836., 1839., 1847. i 1852.