Univerzitet u Beogradu
Matematički fakultet

Master rad

Komunikacija između procesa na Android operativnom sistemu

Student:
Ilić Aleksandar

Mentor:
dr Filip Marić

Mentor:
dr Filip Marić
Matematički fakultet Univerziteta u Beogradu

Članovi komisije:
Dr DušanTošič
Matematički fakultet Univerziteta u Beogradu

Dr Miroslav Marić
Matematički fakultet Univerziteta u Beogradu

Datum odbrane:
_____

Sadržaj

Apstrakt

Platforme za mobilne uređaje su nešto što se trenutno najbrže razvija u računarskom svetu, tako da je većina inovacija postaje deo ove platforme. Centralni deo mobilnih platformi čine operativni sistemi. Pored ostalih uloga koje imaju operativni sistemi za normalno funkcionisanje jednog uređaja, veoma važna uloga koju operativni sistemi obezbeđuju je komunikacija između procesa (*eng. Inter Process Communitation, IPC*). Jezgro operativnog sitema Linux nudi različite mehanizme komunikacije, poput datoteka, soketa, signala, redova poruka, semafora, deljene memorije itd. Zasnovan na Linux operativnom sistemu, Android operativni sistem je jedan od vodećih operativnih sistema za mobilne platforme. Andorid operativni sistem nudi svoj mehanizam komunikacije kroz sloj koji se nalazi iznad Linux jezgra koji se naziva *Binder* okvir. Smatra se da je to najbitniji deo Android operativnog sistema jer povezuje sve njegove komponente.

Cilj ovog rada detaljno izlaganje međuprocesne komunikacije na Android operativnom sistemu, kao i to zašto je ona efikasan model za razmenu poruka između procesa. Rad se sastoji iz dva dela.

U prvom delu je dat opis Android operativnog sistema sa detaljno opisanim teorijskim i implementacionim aspektima *Binder* okvira. Prikazani su razni nivoi komunikacije od najnižeg do najapstraktnijeg, njihove dobre i loše strane. U drugom delu je kao artefakt ovog rada izložena aplikacija “Kule Hanoja” kroz koju su prikazane razne implementacije inter-procesne komunikacije karakteristične za *Binder* okvir.

Rečnik pojmova

Intent – Predstavlja poruku kojom se može zatražiti neka akcija od osnovnih Android komponenta koje imaju filter za tu poruku

IPC (eng. *inter-process communication*) – Komunikacija između procesa

Kernel – Centralni deo operativnog sistema u kome su implementirane sve njegove neophodne-osnovne funkcionalnosti. On upravlja ulaznim/izlaznim zahtevima softvera i prevodi ih procesorske instrukcije koje se izvršavaju na procesoru.

Binder driver – Drajver u operativnog sistema koji je implementiran u krenelu i zadužen za IPC.

IBinder Interface – Predstavlja skup metoda koje *Binder* objekat mora da implementira

AIDL (*eng.* Android Interface Definition Language) – Jezik za definisanje operacija I*Binder* interfejsa.

Binder Token – Jedinstveni 32-bitni integer koji identifikuje *Binder* objekat širom sistema

Binder Transaction – Predstavlja pozivanje metoda na udaljenom *Binder* objektu i može uključivate slanje ili primanje podataka preko *Binder* protokola.

Binder Client – Objekat koji koristi usluge *Binder* servisa.

Parcel – Predstavlja kontejner poruka (podataka i referenci objekata) koje mogu biti poslate kroz I*Binder*. Jedinica transakcionih podataka, jedna za odlazni zahtev i druga za dolazni odgovor.

Marshalling – Proces pretvaranja „high-level“ struktura podataka u parcele, sa ciljem da podaci budu integrisani unutar transakcije.

Unmarshalling – Obrnut proces od marshalling-a, tačnije rekonstrukcija podataka iz parsela.

Proxy – Implementacija *AIDL* intefejsa koja radi (un)marshaling podataka i mapira pozive metoda u tansakcije.

Stub – Implementacija *AIDL* interfejsa koja mapira transakcije u pozive metoda *Binder* Servisa.

Context Manager – Posebna vrsta *Binder* objekata koji se koristi kao registar za druge *Binder* objekte.

Thread safe – Koncept kompjuterskog programiranja koji se primenjuje u kontekstu programa koji rade na više niti. Parče koda je thead-safe ako ono manipuliše deljenim strukturama podataka na način koji garantuje bezbedno izvršenje po više niti istovremeno.

Dalvik Virtual Mashine (DVM)- Virtuelna mašina na kojoj se izvršavaju aplikacije na Android OS

Google Play – Internet prodavnica aplikacija za Android OS

OpenGL (*eng.Open Graphics Library*)- Biblioteka za grafička iscrtavanja

API (*eng. Application Programing Interface*)- Programski interfejs

1. Uvod

Poslednjih godina pametni mobilni uređaji su postali neizostavni deo života svakog čoveka. U prilog tome govori činjenica da se do 2016 očekuje da na tržištu bude preko 2 milijarde korisnika pametnih telefona [1]. Mobilni telefoni postaju toliko moćni da sve više zamenjuju „obične" računare u životima ljudi, čemu doprinose napredne mogućnosti koje nude ovi uređaji. Sa povećanjem mogućnosti koje nude mobilni telefoni, nastaje i potreba za boljim hardverskim karakteristikama uređaja. Tako da premijum telefoni nekih proizvođača imaju dva procesora sa po četri jezgra kao i rezoluciju ekrana od čak 2560x1440 piksela.

Trenutno se na tržištu pametnih telefona mogu naći nekoliko različitih operativnih sistema. Tri najznačajnija su Windows Mobile (Microsoft Inc.), iPhone OS (Apple Inc.) i Android OS (Google Inc.). Android OS je najzastupljeniji sa udelom od 85% na svetskom tržištu [2]. Razlog tolike zastupljenosti Android OS-a je u tome sto je on otvorenog koda i zato mnogi proizvođači telefona (Samsung, HTC, LG, Sony, itd.) prave uređaje sa ovim operativnim sistemom. Neke studije predviđaju da će softver orvorenog koda nastaviti svoju dominaciju na svetskom tržištu OS za mobilne uređaje [16].

Android OS ima veoma dobru podršku za „nezavisne" aplikacije (aplikacije koje nisu napravljene od strane kompanije Google). On nudi razna proširenja kroz dodatke kao što su GPS, NFC, Google Map, Bluetooth, SMS, Content providers i mnogih drugih Google servisa koji omogućavaju pravljenje najraznovrsnijih aplikacija od strane nezavisnih programera. Kompanija Google je takođe napravila onlajn prodavnicu aplikacija (Google Play), putem koje korisnici mogu da skidaju razne aplikacije. Sa programerske tačke gledišta, pravljenje aplikacije za Android predstavlja šansu da se napravi inovativna aplikacija koja će biti dostupna više-milionskom i stalno rastućem tržištu. U junu 2015 je Google Play nalazilo preko 700000 aplikacija.

2. Android operativni sistem

U ovom poglavlju će biti predstavljen istorijat Android OS kroz verzije, njegova struktura i odlike, kao i osnovne komponente svake Android aplikacije.

2.1 Šta je Android?

Android predstavlja platformu otvorenog koda za mobilne uređaje. Android OS je prvobitno razvijen od istoimene firme Android Inc. koju je kompanija Google kupila 2005. godine. Kompanija Google je želela da Android bude otvorenog koda pa je zato inicirala stvaranje Open Handset Alliance (OHA) udruženja koje radi na razvoju Andorid OS [3]. Andorid OS je objavljen pod Appache licencom za otvoreni kôd što znači da da svako može da skine izvorni kôd Android operativnog sistema i doda svoje izmene. Ova pogodnost je zainteresovala mnoge proizvođače pametnih telefona, pre svega kompanije Sony i Motorola. Oni su razvijali svoje operativne sisteme koji su doživeli neuspeh pojavljivanjem na tržištu prvog iPhone telefona kompanije Apple Inc., tako da su oni pronašli alternativu u korišćenju Andorid OS.

2.2 Arhitektura Android OS

Android operativni sistemi su zasnovani na verziji 2.6.*, a noviji na verziji 3.1* Linux operativnog sistema, tako da se za Android OS može reći da je Linux razvijan za ARM i x86 arhitekturu. Za razliku od Linux operativnog sistema, Android nema standardni X Window System, ni standardni skup GNU biblioteka tako da nije u mogućnosti da pokreće aplikacije koje su pravljene za standardne Linux sisteme [4,5]. Android OS se sastoji iz sledećih slojeva (Slika 2.2.):

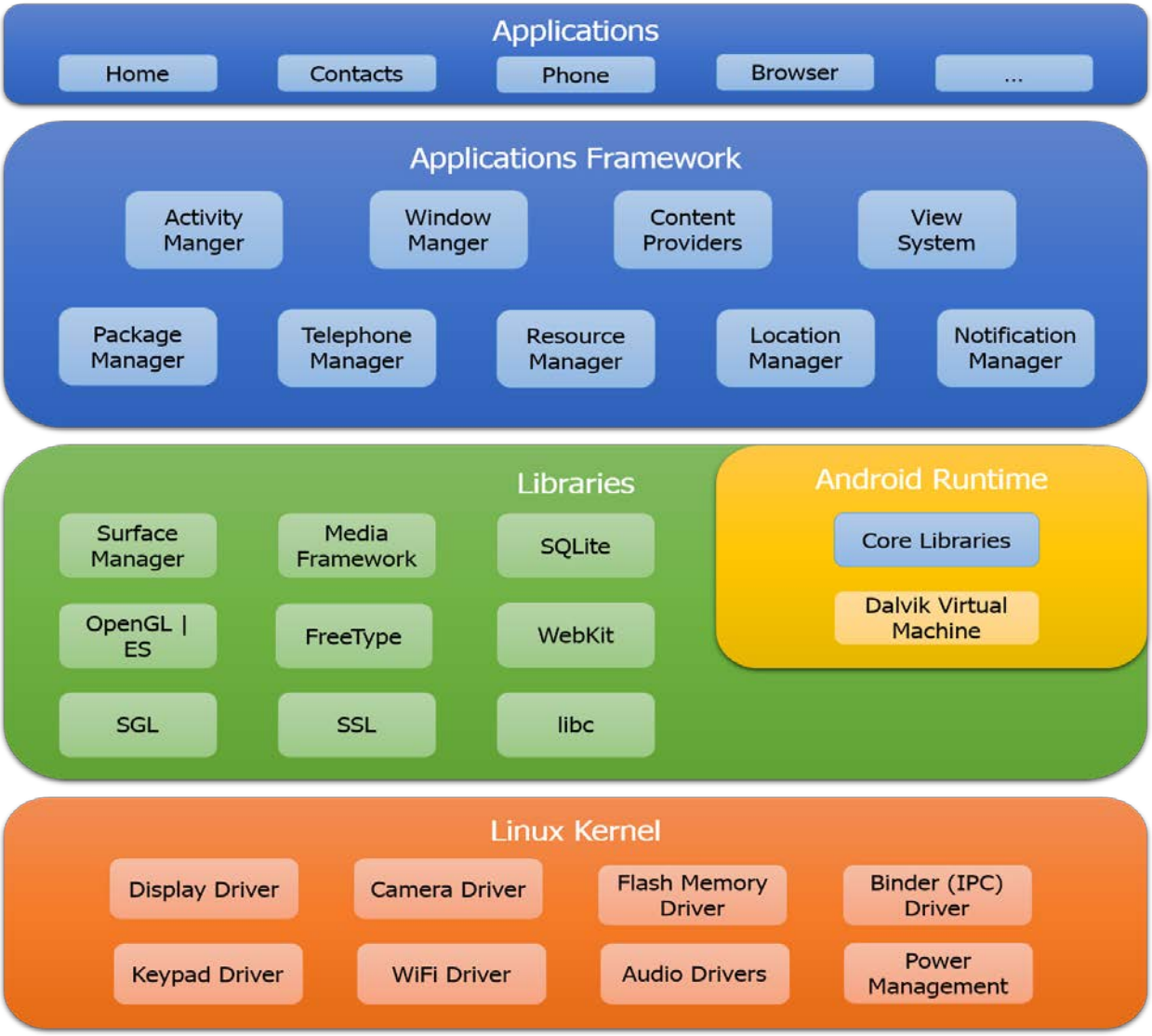
- *Aplikativni sloj* – predstavlja najviši sloj u arhitekturi Android OS i čine ga korisničke aplikacije. Korisničke aplikacije mogu biti sistemske (Imenik, Internet pregledač, SMS aplikacija, itd.) i nezavisne aplikacije. Nezavisne aplikacije predstavljaju sve one aplikacije koje nisu napisane od strane proizvođača uređaja.

- *Sloj okvir aplikacija* – predstavlja skup programskih komponenti uređaja koje koriste aplikacije. On obezbeđuje servise visokog nivoa aplikacijama preko API koji je napisan u Javi. Neke od najvažnijih komponenata ovog sloja su: *Menadžer aktivnosti* (upravljanje životnim ciklusom aplikacije), *Menadžer paketa* (sadrži informaciju o tome koje su aplikacije instalirane na uređaju), *Menadžer prozora* (upravljanje prozorima aplikacija), *Menadžer telefonije* (upravljanje pozivima), *Provajder sadržaja* (omogućavaju da više aplikacija koristi iste podatke), *Menadžer resursa* (brine o resursima aplikacija, a to mogu biti na primer razni multimedijalni fajlovi). Ovaj sloj je napisan u programskom jeziku Java.

- *Sloj biblioteka* – predstavlja skup C/C++ biblioteka. Ove biblioteke koriste razne komponente sistema i one su izložene programerima kroz *sloj okvira aplikacija.* Najbitnije biblioteke su sledeće: *Okvir medija (*podrška za mutimediju*), SQLite (*podrška za bazu podataka*), OpenGL (podrška za iscrtavanje grafičkih elemenata), sistemske C biblioteke, itd.*

- *Sloj Android izvršavanja* – je komponenta se nalazi u istom sloju kao i *Biblioteke.* Glavni deo ove komponente je *Dalvik virtuelna mašina (DVM)* do verzije 5.0 tj. *Android Runtime (ART)* u novijim verzijama. *DVM* je vrsta *Java virtuelne mašine (JVM)* koja je optimizovana za uređaje sa slabijim procesorom i manjom memorijom. Za razliku od JVM, DVM ne izvršava *.class* nego *.dex* fajlove, koji obezbeđuju veću efikasnost u radu na uređajima sa manjim resursima. Svaka aplikacija ima svoju instancu DVM i izvršava se u zasebnom procesu. Zbog performansi DVM se kreira samo jednom prilikom pokretanja operativnog sistema u sistemskom servisu koji se naziva *Zygote*. Svaka naredna instanca se dobija kloniranjem ove. Nakon verzije 5.0, kompanija Google je novom virtuelnom mašinom ART u potpunosti zamenila Dalvik. ART ima velike prednosti u odnosu na DVM. Na primer, ART podržava kompilaciju unapred (Ahead of time, AOT) i ima unapređen sakupljač otpadaka koji značajno unapređuje performanse aplikacija.

- *Linux jezgro* – predstavlja poslednji sloj arhitekture u kojem se nalaze uglavnom drajveri za različite hardverske komponente. Zato se za ovaj sloj može reći da predstavlja apstraktni sloj između hardvera i softvera. U Linux jezgru su takođe definisane sve stvari vezane za bezbednost, upravljanje procesima, memorijom i napajanjem. Jedna od najvažnijih komponenti je *Binder* drajver koji će biti detaljnije opisan u nastavku rada.



Slika 2.2. Arhitektura Android operativnog sistema

2.3 Karakteristike Android OS

Open Handset Alliance ističe nekliko bitnih karakteristika Android operativnog sistema:

- **Otvorenost** – sigurno jedna od najznačajnih karakteristika, koja je doprinela tolikoj popularizaciju Android OS. Omogućava svima potpunu slobodu u razvoju novih aplikacija i njihovo postavljanje na Google Play prodavnicu, a proizvođačima mobilnih uređaja slobodno korišćenje i prilagođavanje osnovne verzije Android OS, bez plaćanja autorskih prava;

- **Sve aplikacije su ravnopravne** – Android OS dozvoljava ravnopravan pristup resursima mobilnog uređaja svim aplikacijama. Ovo korisniku daje veliku mogućnost izbora kako bi što bolje prilagodio uređaj individualnim potrebama. Aplikacije koje su bitne za normalno funkcionisanje moraju imati sitemski popis. U većini slučajeva su to aplikacije koje pravi proizvođač uređaja. Na primer, Samsung koristi svoju „pokretač" aplikaciju TouchWiz. Ova aplikacija se prva pokrene kad se upali uređaj i kroz nju se pokreću ostale aplikacije.

- **Automatsko upravljanje životnim ciklusom aplikacije** – operativni sistem ima potpunu kontrolu nad upravljanjem životnim ciklusom aplikacija, na taj način omogućava njihovo optimalno korišćenje. Iz ovoga proizilazi to da korisnik ne mora da brine o zatvaranju aplikacije pre nego što pokrene drugu.

- **Brz i jednostavan razvoj aplikacija** – ova karakteristika je pre svega omogućena veoma dobro razvijenim razvojnim okruženjem (Android Studio), kao i bogatom bazom nezavisnih biblioteka. Razvojno okruženje je takođe potpuno besplatno.

- **Visokokvalitetni grafički prikaz i zvuk** – zahvaljujući integraciji 2D vektorske i 3D OpenGL grafike, kao i kodecima za većinu audio i video formata.

- **Kompatibilnost sa većinom sadašnjeg i budućeg hardvera** – uključuje prenosivost Android aplikacija na ARM, x86 i ostale arhitekture, te prilagodljivost sistema ulaznim i izlaznim komponentama.

2.4 Verzije Android OS

Android OS se razvija prilično brzo, tako da su korisnici skoro navikli da dobiju novu verziju softvera svake godine. Svaka nova verzija dolazi sa novim uslugama koje se nude kranjem korisniku. U listi ispod su navedene verzije Android OS kao i splisak usluga koje su nudile korisnicima [12].

Verzija 1.0 (23. septembar, 2008) je uvela sledeće novine:

- skidanje i postavljanje aplikacija na *Android Market* (kasnije *Google Play*),
- podrška za kameru,
- internet pregledač (*eng. Internet browser*),
- integracija Google servisa (*Gmail, Google maps, Calendar, Hangout*),

- Medija plejer,
- podrška za Wi-Fi i bluetooth bežičnu komunikaciju.

Verzija 1.5 „*Cupcake*“ (30. april, 2009) je uvela sledeće novine:

- integracija predikcije reči prilikom kucanja na tastaturi,
- dodavanje vidžeta,
- snimanje video-zapisa u MPEG-4 i 3GP formatu,
- animacije koje se koriste da poboljšaju,
- auto-rotacija ekrana,
- mogućnost postavljanja videa na YouTube.

Verzija 1.6 „*Donut*“ (15. septembar, 2009) je uvela sledeće novine:

- dodavanje galerije slika,
- bolja integracija kamere,
- podrška za WVGA rezolucije ekrana,
- nova verzija *Android Marketa*,
- olakšana pretraga aplikacija,
- mogućnost slikanja sadržaja na ekranu (*eng. screenshot*).

Verzija 2.0/2.1 „*Eclair*“ (26. oktobar, 2009) je uvela sledeće novine:

- mogućnost dodavanja više korisničnih naloga,
- poboljšana sinhronizacija mejlova i kontakata,
- podrška za Bluetooth 2.1 bežičnu komunikaciju,
- poboljšana brzina kucanja na tastaturi,
- podrška za više veličina i rezolucija ekrana,
- poboljšane aplikacije *Google maps*.

Verzija 2.2.x „*Froyo*“ (20. maj, 2010) je uvela sledeće novine:

- poboljšano upravljanje memorijom,
- integracija podrške za jezik JavaScript u internet pregledačima,
- integracija „Could to Device Messeging“(C2DM) servisa [7],
- USB povezivanje,
- mogućnost deljenja interneta između uređaja,
- glasovno biranje i deljenje kontakata preko Bluetooth bežične konekcije,
- podrška za *Adobe Flash*,
- skidanje i postavljanje aplikacija na Google Play (tada još uvek Android Market).

Verzija 2.3.x „*Gingerbread*“ (6. decembar, 2010) je uvela sledeće novine:

- unapređene i pojednostavljene korisničkog interfejsa sa bržim odzivom,

- podrška za jako velike ekrane i rezolucije (WXGA i veće),
- unapređena podrška za SIP i VoIP internet telefoniranje,
- integracija komunikacije bliskog polja (*eng. Near Field Communication, NFC*) omogućavanja korisniku da čita etikete sa postera stikera ili oglasa [8],
- poboljšana grafika kao i obrada zvuka,
- ugrađena pordška za senzore (žiroskop, barometar).

Verzija 3.x „*Honeycomb*“ (22. februar, 2011) je uvela sledeće novine:

- Proširenja vezana za tablet uređaje,
- Podrška za procesore sa više jezgara,
- Mogućnost otvaranja više kartica u internet pretraživaču,
- Podrška za HTTP *proksi.*

Verzija 4.0.x „*Ice cream Sandwich*“ (14. novembar, 2011) je uvela sledeće novine:

- Soft dugmići za navigaciju umesto hardverskih,
- poboljšana brzina i performanse,
- poboljšana copy/paste funkcionalnost,
- *Android Beam*, lokacijska komunikacija koja omogućava deljenje kontakata, *YouTube* videa i drugog saržaja [9],
- pordška za *WebP* format slika,
- hardversko ubrzanje korisničkog interfejsa,
- *Wi-Fi direct*.

Verzija 4.3.x „*Jelly Bean*“ (24. jul, 2013) je uvela sledeće novine:

- mogućnost pravljenja više korisničkih profila,
- *Bluetooth smart ready* tehnologija, podrška za satove i drugu dodatnu opremu,
- predikcija prilikom pretraživanja kontakta,
- *OpenGL ES 3.0*,
- *Bluetooth AVRCP*, mogućnost povezivanja sa audio sistemom u autu koristeći *Bluetooth*,
- mogućnost određivanja lokacije putem bežičnog interneta,

Verzija 4.4.x „*KitKat*“ (3. septembar, 2013) je uvela sledeće novine:

- *OK Google* servis za glasovnu pretragu i upravljanje uređajem,
- neograničen broj početnih ekrana,
- podrška za pedometar,
- impresivan režim, mogućnost uklanjanja navigacionih dugmića prilikom čitanja ili igranja igrica
- *QuickOffice* integracija [10],
- *Cloud Storage* integracija [11],

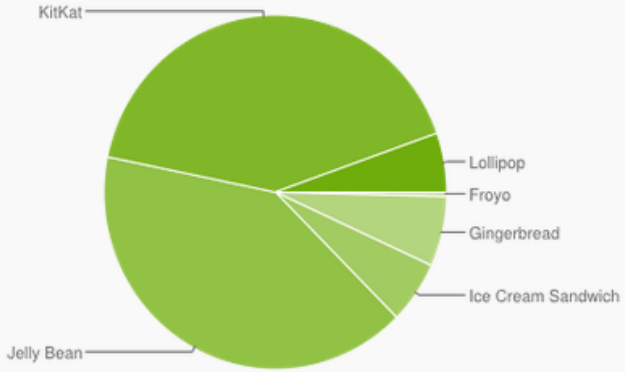
- Brži multi-tasking

Verzija 5.0.x *„Lolipop“* (25. jun, 2014) je uvela sledeće novine:

- podrška za 64-bitne procesore,
- *Android Runtime* (ART) je u potpunosti zamenio Dalvik virtuelnu mašinu,
- „Materijalni“ dizajn, restilizovani korisnički interfejs,
- Audio ulaz i izlaz putem USB,
- unapređena zaštita uređaja.

Kao pomoć programerima koji pišu aplikacije za Android uređaje, kompanija Google prikuplja podatke o verzijama Android OS na uređajima koji se aktivno koriste. Ovi podaci (Slika 2.2) se osvežavaju na svakih 7 dana [13].

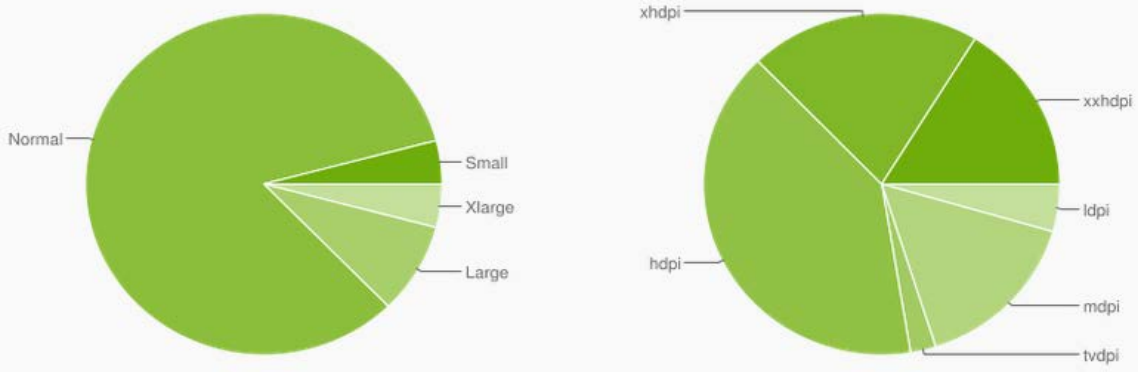
Version	Codename	API	Distribution
2.2	Froyo	8	0.4%
2.3.3 - 2.3.7	Gingerbread	10	6.4%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	5.7%
4.1.x	Jelly Bean	16	16.5%
4.2.x		17	18.6%
4.3		18	5.6%
4.4	KitKat	19	41.4%
5.0	Lollipop	21	5.0%
5.1		22	0.4%



Slika 2.2. Prikaz zastupljenost različitih verzija Android OS na svim uređajima širom sveta, dana 1.6.2015

Zbog pravljenja razumljivijeg, intuitivnijeg i lepšeg korisničkog interfejsa, kompanija Google takođe prikuplja podatke o konfiguraciji ekrana na Android uređajima (Slika 2.3)[13]. Ova konfiguracija je prikazana kao odnos veličine ekrana i gustine piksela (eng. density).

	ldpi	mdpi	tvdpi	hdpi	xhdpi	xxhdpi	Total
Small	4.1%						4.1%
Normal		7.6%	0.1%	39.5%	19.8%	16.2%	83.2%
Large	0.4%	4.8%	2.2%	0.6%	0.6%		8.6%
Xlarge		3.2%		0.3%	0.6%		4.1%
Total	4.5%	15.6%	2.3%	40.4%	21.0%	16.2%	



Slika 2.3 Prikaz zatupljenosti ekrana, grupisanih na odnosu gustine piksela i veličine ekrana, dana 1.6.2015

2.5 Osnovne komponente Android aplikacije

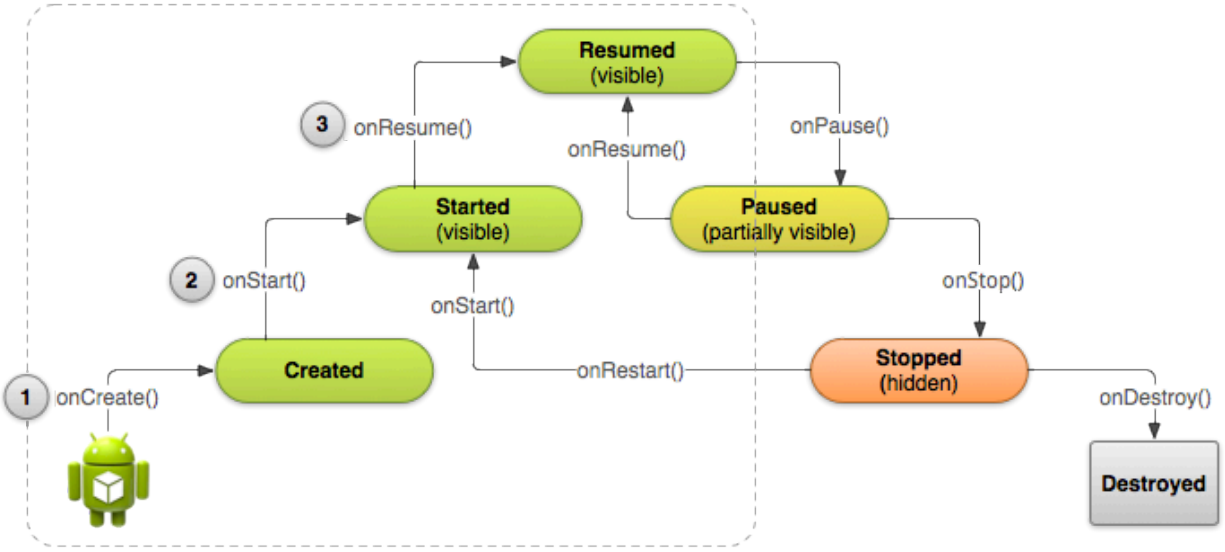
Osnovne komponente predstavljaju gradivne elemente svake Android aplikacije. Svaki od ovih elemenate ima svoju specifičnu ulogu u opisu ponašanja aplikacije. Sve komponente jedne aplikacije su definisane u njenom *AndroidManifest.xml* fajlu (više u poglavlju 2.5.1). Na Android platformi postoje četri različitih vrsta komponenti:

- *Aktivnosti (eng. Activity)* – Ova komponenta predstavlja pojedinačni ekran/prozor sa kojim korisnici mogu komunicirati kako bi nešto uradili. Jedna aplikacija može imati više aktivnosti i sve one se međusobno nezavisne (*eng. loosely coupled*) [6]. Uglavnom aplikacija ima svoju "glavnu" aktivnost koja se pokreće kada korisnik pokrene aplikaciju. Jedna aktivnost može pokrenuti neku drugu aktivnost ili pak neku drugu komponentu, na primer, servis. Za svaku aktivnost postoji životni ciklus koji definiše kako se ona kreira i kako se ona uništava (Slika 2.4). Redosled pozivanja metoda, koje definišu životni ciklus aktivnosti, je regulisan od strane Android OS.

- *Servisi (eng. Service)* – Za razliku od aktivnosti koje su vezane za neki ekran, servisi predstavljaju komponente koje se izvršavaju u pozadini. Servisi su namenjeni operacijama koje se dugo izvršavaju. Sve komponente mogu da pokrenu neki servis, i on će nastaviti da „živi" i kada korisnik napusti aplikaciju. Ostale komponente se mogu vezati za servis i komunicirati sa njim unutar procesa ili čak preko inter-procesne komunikacije (ona je detaljnije opisana u Poglavljima 3 i 4). Servis se može javiti u dva oblika, može biti pokrenut ili vezan. Pokrenut servis se pokreće metodom *startService()* koja se poziva iz neke druge komponente. Startovan servis ostaje da „živi" u pozadini čak iako je komponenta koja ga je startovala uništena. Ovakav servis se koristi za obavljanje nekih dugih operacija, kao što su skidanje nekog sadržaja sa interneta i on, uglavnom, ne vraća nikakav rezultat komponenti koja ga je pozvala. Drugi oblik servisa je vezani servis; i klijent se vezuje za servis metodom *bindService()*. Vezan servis obezbeđuje klijent-server interfejs, koji omogućava komponentama da međusobno komuniciraju. Životni vek servisa, pokrenutog na ovaj način, direktno zavisi od životnog veka komponenti sa kojima je on vezan. Ukoliko je servis vezan sa više komponenti, servis će „živeti" sve dok poslednja komponenta sa kojom je vezan ne bude uništena. Servis može biti vezan i sa komponentama iz drugog procesa preko IPC (detaljnije u Poglavljima 3 i 4). Na Android OS postoje dve klase servisa, to su *Service* i *IntentService. Service* je osnovna klasa servisa; on se izvršava na glavnoj (UI) niti i može da utiče na performanse aplikacije ukoliko se u njemu izvršavaju duge operacije. *IntentService* je servis koji se izvršava van glavne niti i on je predviđen za izvršavanje dugih operacija u pozadini.

- *Prijemnici (eng. Broadcast reveicer)* – Osnovna namena ove komponente je da prihvata *Intent* objekte (detaljnije o *Intent* objektima u Poglavlju 3.1) koji se šalju širom sistema poslate metodom *sendBroadcast().* Prijemnici mogu biti registrovani dinamički u ostalim komponentama ili statički u *AndroidManifest.xml* fajlu. Prilikom registrovanja prijemnika definiše se i *IntentFilter* objekat kako bi on dobijao samo određene *Intent* objekte.

- *Provajderi sadržaja (eng. Content provider)* – Provajderi sadržaja služe kako bi obezbedili pristup centralnom repozitorijumu podataka. Primarna namena ove komponente je u tome da omogući deljenje podataka između aplikacija/procesa. Da bi podaci neke aplikacije bili dostupni nekoj drugoj, potrebno je da ta aplikacija definiše interfejs prema svojim podacima. Taj interfejs se definiše uz pomoć *ContentProvider* klase. Sa druge strane, klijentska aplikacija, koja koristi te podatke, mora da instancira objekat klase *ContentResolver* koji će sadržati iste metode kao i *ContentProvider* objekat aplikacije čijim podacima želimo da pristupimo. *ContentResolver* metode obezbeđuju „KPAO" (kreiraj, povrati, ažuriraj i obriši) operacije nad deljenim podacima.



Slika 2.4 Prikaz životnog ciklusa aplikacija

2.5.1. AndroidManifest.xml

Svaka aplikacija mora da poseduje svoj AndroidManifest.xml fajl u korenom folderu projekta. Ovaj XML sadrži sve neophodne informacije o aplikaciji koje su potrebne Android operativnom sistemu kako bi bez problema upravljao njome. Ove informacije sistem mora da poseduje pre početka izvršavanja aplikacije, tako da se podaci iz AndroidManifest-a učitavaju prilikom njene instalacije na sistem (*Slika 2.5.1*). AndroidManifest.xml fajl sadrži sledeće informacije o aplikaciji:

- ime paketa aplikacije,
- broj verzije aplikacije koji se koristi da bi Android OS proverio da li postoji novija verzija aplikacije na Google Play-u,
- ime verzije aplikacije je string vrednost, koja se uglavnom koristi za prikazivanje broja verzije korisniku,
- *android:minSdkVersion* atribut *<uses-sdk>* elementa određuje minimalnu verziju operatnivnog sistema na kojoj aplikacija može raditi,
- u ovom fajlu se definiše logo aplikacije koja će se prikazati u glavnom meniju uređaja, kao i ime aplikacije,
- osnovne komonente kao i filteri za *Intent* objekte koji ih pokreću,
- definiše se komponenta koja će se prva pokrenuti prilikom startovanja aplikacije,
- sve dozvole (*eng. permission*) koje aplikacija mora da ima da bi nesmetano radila i pristupala određenim resursima,
- lista biblioteka koje aplikacija koristi.

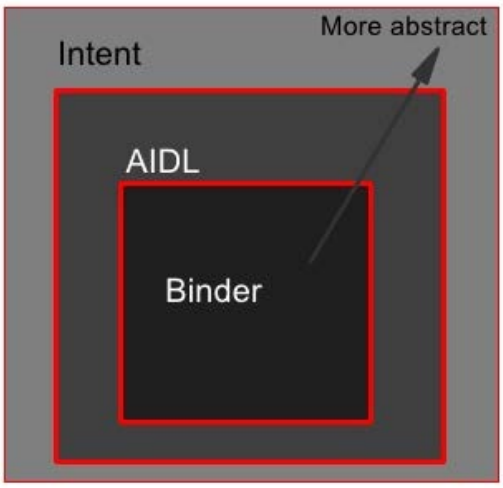
```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
  <uses-permission />
  <permission />
  <permission-group />
  <uses-sdk />
  <uses-configuration />
  <uses-feature />
  <compatible-screens />
  <application>
    <activity>
      <intent-filter>
        <action />
        <category />
        <data />
      </intent-filter>
      <meta-data />
    </activity>

    <activity-alias>
      <intent-filter> . . . </intent-filter>
      <meta-data />
    </activity-alias>

    <service>
      <intent-filter> . . . </intent-filter>
      <meta-data/>
    </service>

    <receiver>
      <intent-filter> . . . </intent-filter>
      <meta-data />
    </receiver>

    <provider>
      <grant-uri-permission />
      <meta-data />
      <path-permission />
    </provider>

    <uses-library />
  </application>
</manifest>
```

Slika 2.5.1 Prikaz strukture AndroidManifest.xml fajla

3. Implementacija mehanizama inter-procesne komunikacije karateristične za *Binder* okvir

Android OS karakteriše upotreba inter-procesne komunikacije (IPC) u okviru aplikacije kao i između aplikacija. Android OS svaku aplikaciju pokreće u zasebnom procesu (*eng.* sandbox) sa jedinstvenim sistemskim identifikatorom. Svaka aplikacija ima svoju zasebnu instancu Dalvik VM. Na ovaj način aplikacije su u potpunosti nezavisne jedne od drugih sa sopstvenim adresnim prostorom. Ovakva implementacija nam omogućava, pre svega, sigurnost i bezbednost jer jedan proces ne može direktno upravljati memorijom drugog procesa. Na Linux operativnom sistemu to je postignuto mehanizmom virtuelne memorije (*virtual memory mechanism*) gde se procesu dodeljuje linearni i neprekidni deo virtuelne memorije koju operativni sistem mapira u fizičku memoriju. Na ovaj način izolacija procesa osigurava zaštitu memorije svakog procesa. Međutim, u mnogim slučajevima komunikacija između procesa je neophodna. Zato operativni sistem mora da obezbedi mehanizme za inter-procesnu komunikaciju.

Android OS ne podržava sve *System V* inter-procesne mehanizme koje postoje na Linux OS [14]. Razlog tome je što su ovi mehanizmi skloni curenju resursa, kada proces „zaboravi“ da ukloni resurse kad ga operativni sistem skloni iz memorije. Android operativni sistem sadrži jedan oblik inter-procesne komunikacije koji je karakterističan samo za njega, a to je *Binder* okvir. IPC na Android operativnom sistemu se većinom odvija preko *Binder* okvira i u maloj meri na soketima koji se koriste na najnižem nivou.



Slika 3.1.1 Prikaz nivoa apstrakcije kod IPC

Inter-procesna komunikacija u *Binder* okvir je implementirana kao odnos klijent-servis. Na Android platformi postoje tri mehanizma, tj. implementacije inter-procesne komunikacije koja se odvija preko *Binder* okvira, a to su: *Intent* mehanizam, *Messenger* mehanizam i mehanizam baziran na jeziku za definisanje Andorid interfejsa. *Messenger* i *AIDL* implementacije rade sa vezanim (*eng. bound*) servisima, dok *Intent* objekti rade sa pokrenutim (*eng. started*) servisima.

3.1 *Intent* mehanizam za IPC

Intent mehanizam predstavljaju najabstrktiniji nivo IPC na Android OS. *Intent* mehanizam je u potpunosti nezavistan od implementacije aplikacija. Na Android operativnom sistemu nikad ne kažemo da želimo da pošaljemo mejl nekom određenom aplikacijom; umesto toga mi samo kažemo da želimo da pošaljemo mejl. Android OS će se sam pobrinuti za to, jer on već unapred zna koje aplikacije služe za slanje mejla na osnovu filtera za *Intent* objekte koji su definisani u AndroidManifest.xml fajlu tih aplikacija. Intenti služe za asinhronu komunikaciju izmedju komponenti i zbog svoje jednostavnosti su najčešće korišćeni oblik IPC.

Intenti se sastoje iz dva dela *akcije i podataka.* Pod apstraknim se misli na to da komponente, koje treba da izvrše akciju, ne moraju biti definisane u *Intent* objektu. *Intent* objekti se mogu zamisliti kao poruke koje se šalju širom operativnog sistema kako bi izvršile odredjenu akciju na nekoj koponenti koja ima odgovarajući filter za tu akciju. Osnovne komponente svake aplikacije imaju definisine filtere u *AndroidManifest.xml* fajlu te aplikacije. Filter jedne komponente sardže akcije koje podržava ta komponenta. Kad se pošalje intent, Android OS pronalazi sve komponente koje sadrže filter za tu akciju. Ukoliko postoji više komponenti sa tim filterom, Android OS prepušta korisniku da izabere željenu komponentu. Postoje dve vrste *Intent* objekata e*ksplicitni* i *implicitni. Eksplicitni* gađaju određenu komponentu po imenu i ograničen je samo na komponente unutar jedne aplikacije. S obzirom na to da se komponente jedne aplikacije mogu nalaziti na različitim procesima, i eksplicitni intenti imaju svoju ulogu u IPC. Sa druge strane *implicitni,* pokušavaju da pronađu komponente po akciji i nisu ograničeni na komponete jedne aplikacije.

Deo koda koji se nalazi na slici 3.1.2 pokazuje koliko je *Intent* mehanizam za inter-procesnu komunikaciju zapravo jednostavan za korišćenje.

```
//slanje intenata koji u sebi sadrži string Broadcast receiver-u
Intent intent = new Intent();
intent.setAction("com.primer.akcija");
intent.putExtra("MusicList", msg);
context.sendBroadcast(intent);
```

Slika 3.1.2 Primer slanja *Intent* objekta

Na slici 3.1.3 AndroidManifest.xml fajlu je definasan *Prijemnik* komponenta „MusicListReceiver“ koji ima filtere za akciju „com.primer.akcija".

```
<!--Broadcast receiver koji u sebi sadrži filter za
„com.primer.akcija" akciju-->

<receiver android:name="MusicListReceiver">
  <intent-filter>
    <action android:name=" com.primer.akcija"/>
    <category android:name="android.intent.category.DEFAULT"/>
  </intent-filter>
</receiver>
```

Slika 3.1.3 Primer definisanja filtera u AndroidManifest.xml fajlu

IPC zasnovana na *Intent* mehanizmu je asinhrona i Android OS ne garantuje vreme za koje će *Intent* objekat pokrenuti neku komponentu. Svaki *Intent* objekat sadži:

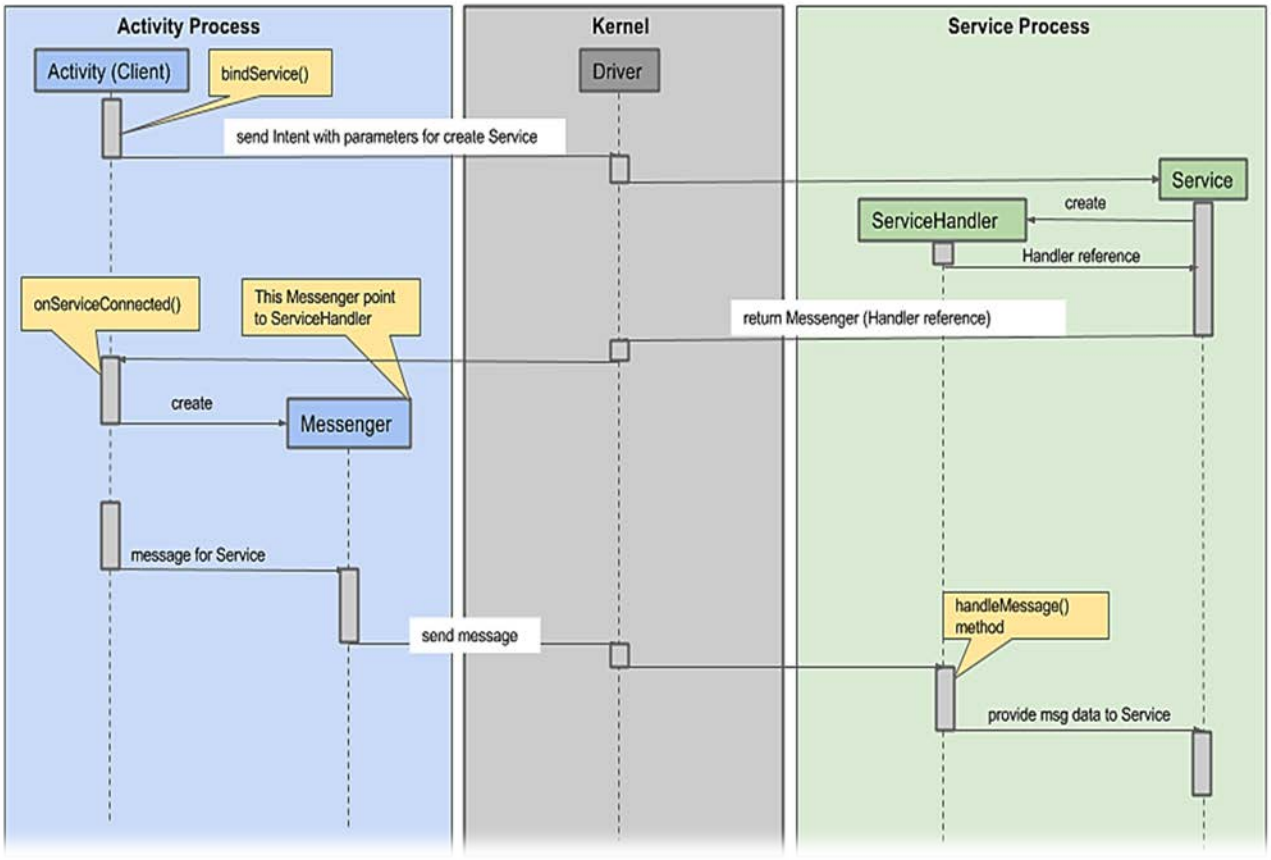
- *Ime komponente* – Ova informacija je neobavezna jer implicitni *Intent* objekti ne sadrže ovu informaciju, pa Android OS određuje koja komponenta treba da bude pokrenuta na osnovu ostalih informacija.
- *Akciju* – String koji precizira koja akcija treba da se izvrši. Android OS sadrži spisak predefinisanih sistemskih akcija kojima se pozivaju sistemski servisi, dok za svoje komponente programer sam definiše akcije koje će ih pokrenuti u filteru tih komponenti.
- *Podatke* – Uglavnom služi za slanje lokacije podataka. Osim lokacije vazno je naglasiti tip podataka kako bi Andriod OS našao najbolju komponentu za određene podatke.
- *Kategorija* – Neobavezan string koji sadrži dodatnu informaciju o tome koju komponentu treba da pokrene *Intent* objekat.
- *Flagove* – Mogu se posmatrati kao meta podaci *Intent* objekata.

Intent mehanizam kao najabstraktniji nivo IPC je najlakši za korišćenje, ali on zbog toga ima svoja ograničenja. Zbog toga što je ansinhrona, ova mehanizam IPC nije dobro prilagođen situacijama gde se traži brz odziv. S obzirom da je aplikacioni programski interfejs slabo definisan, ovaj IPC mehanizam je sklon greškama u toku izvršavanja.

3.2 *Messenger* mehanizam za IPC

Messenger mehanizam IPC predstavlja srednji nivo apstrakcije u okviru *Binder* okvira. On predstavlja kompromis između *Intent* i *AIDL* mehanizma. Njegove dobre strane su da ne traži toliko detaljnu implementaciju kao što definisanje interfejsa kod *AIDL* mehanizma i malo je efikasniji od *Intent* mehanizma kad se traži brži odaziv. Njegove loše strane su da nije tako jednostavan kao IPC mehanizam baziran na *Intent* objekatima i nije toliko brz kao *AIDL* implementacija.

U odnosu na *Intent* mehanizam, gde ne moramo da znamo ništa o komponenti sa kojom komuniciramo, ovde komponete ipak moraju da razmene *Messenger* objekte preko kojih ce komunicirati i razmenjivati poruke. Na slici 3.2.1 je prikazan način na koji komuniciraju klijent i servis preko *Messenger* objekata.



Slika 3.2.1 Prikaz Messenger IPC između klijenta i servisa

Odnos *Activity-Servis* možemo posmatrati kao odnos klijent-servis, gde je klijent pretplaćen na servis. Klijent može da šalje zahteve sevisu i od njega dobijati odgovore. *Messenger* je vezan za Handler tako da svi zahtevi koji dolaze od klijenata izvršavaju sekvencijalno na jednoj niti, dok kod *AIDL* se sve izvršava konkurentno na *Binder* nitima. Ovo je glavni razlog zašto je *AIDL* najbrži IPC mehanizam. Zbog toga što se zahtevi izvršavaju sekvencijalno, ne moramo da pravimo više-nitno bezbedan (*eng. thread-safe*) servis. Nekoliko stvari se mora uraditi ukoliko želimo komunikaciju preko *Messenger* objekta:

- Servis mora da implementira *Handler* objekat koji prima povratne pozive (*eng. callbacks*) za svaki poziv od klijenta
- Handler se koristi za pravljenje *Messenger* objekta unutar servisa
- *Messenger* pravi *IBinder* koji servis vraća klijentu iz metode *onBind()*
- Klijent koristi I*Binder* objekat preko *getBinder()* metode i u njegovoj implementaciji konekcije sa servisom da instancira *Messenger* objekat koji će biti referenca na Handler koji je u servisu
- Ovaj *Messenger* objekat klijent koristi kako bi slao poruke servisu metodom *send()*
- Servis će primati poruke u svojoj *Handler.handlemessage()* metodi
- Klijent takođe može definisati svoj *Messenger* objekat koji će poslati servisu ukoliko ima potrebe da servis šalje poruke klijentu
- Klijent se povezuje na servis *bindService()* metodom.

Kôd koji se nalazi ispod opisuje implementaciju *MessengerService* objekta koji koristi *Messenger* mehanizam za IPC.

```
public class MessengerService extends Service {

  private String TAG = "MessengerService";

  // Message.what je kod poruke definisan od strane korisnika
  // kako bi primalac(servis) poruke znao da identifikuje poruku.

  static final int MSG_SAY_HELLO = 1;

  // Messenger objekat koji se prosleduje klijentu i koji on
  // koristi za slanje poruka IncamingHandler-u

  Messenger mMessenger = new Messenger(new IncamingHandler());

  // Handler za dolazeće poruke
  class IncomingHandler extends Handler {
    @Override
    public void handleMessage(Message msg) {
      switch (msg.what) {
        case MSG_SAY_HELLO:
```

```
            Bundle bundle = msg.getData();
            String hello = (String) bundle.get("hello");
            Toast.makeText(getApplicationContext(),
                    hello,Toast.LENGTH_SHORT)
                    .show();
            break;
          default:
            super.handleMessage(msg);
        }
      }
    }
    public MessengerService() {
    }

    @Override
    public void onCreate() {
      Log.d(TAG, "onCreate called");
    }
     /*
          Metoda koja vraća interfejs Messenger-a i preko
          koga klijent šalje poruke servisu.
    */
    @Override
    public IBinder onBind(Intent intent) {
      Log.d(TAG, "onBind done");
      return mMessenger.getBinder();
    }

    @Override
    public boolean onUnbind(Intent intent) {
      return false;
    }
}
```

Slika 3.2.2 Primer implementacije *Messenger* mehanizma IPC kod servisa

Sa druge strane nam je potreban klijent, u ovom slučaju *Activity,* koji samo treba da kreira *Messenger* objekat na osnovu *IBinder* objekta. *IBinder* objekat se dobija iz *ServisConnection* objekta. Poruka se kreira korišćenjem *Messege.obtain(..)* metoda. Nakon inicijalizacije *Messenger* objekta, klijent šalje poruke metodom *send()*. Klijent se povezuje na Servis metodom *bindService(...)*.

```
boolean isBound = false;
Messenger mMessenger;

private ServiceConnection serviceConnection = new ServiceConnection() {
  @Override
  public void onServiceConnected(ComponentName name, IBinder service) {
    isBound = true;
```

```
    // Kreiranje Messenger objekta
    mMessenger = new Messenger(service);
    // Kreiranje poruke MSG_SAY_HELLO je "what" vrednost poruke
  Message msg = Message.obtain(null, MessengerService.MSG_SAY_HELLO, 0, 0);
    // Kreiranje Bundle sa podacima
    Bundle bundle = new Bundle();
    bundle.putString("hello", "world");
    // dodavanje Bundle-a poruci
    msg.setData(bundle);
    // Slanje poruke
    try {
      mMessenger.send(msg);
    } catch (RemoteException e) {
      e.printStackTrace();
    }
  }


  @Override
  public void onServiceDisconnected(ComponentName name) {
    mMessenger = null;
    isBound = false;
  }
};

@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);
  Intent intent = new Intent(this, MessengerService.class);
  bindService(intent, serviceConnection, Context.BIND_AUTO_CREATE);
}
```

Slika 3.2.3 Primer implementacije Messenger mehanizma IPC kod klijenta

Prikazana implementacija se odnosi samo za komunikaciju u jednom smeru (*eng. One-way*), u ovom slučaju za slanje poruka iz klijenta ka servisu. Ukoliko želimo razmenu poruka u oba smera (*eng. Two-way*), definišemo još jedan Messenger objekat, u ovom slučaju na strani klijenta i pošaljemo ga servisu. Messenger postaje dosta komplikovan za korišćenje sa povećanjem broja usluga koje može da izvrši servis.

3.3 Mehanizam IPC zasnovan na jeziku za definisanje Android interfejsa (AIDL)

Android Interface Definition Language predstavlja jezik za definisanje interfejsa. Service koristi ovaj interfejs definisan na ovaj način kako bi izložio svoje mogućnosti (metode) klijentu, koje on kasnije koristi kako bi komunicirao direktno sa njim. Međuprocesna komunikacija bazirna na *AIDL* daje daleko bolje performanse od prethodno dva gore navedena mehanizma za IPC. Zamišljeno je da se *AIDL* mehanizam koristi onda kada je rezmena poruka između dva procesa veoma učestala. *AIDL* sintaksa je veoma slična sintaksi Java interfejsa. *AIDL* definiše metode koje je implementirao servis i koje klijent može jednostavno da koristi direktno pozivanjem tih metoda kao i kod svakog drugog objekta (Slika 3.3.1). Na ovaj način se podiže nivo apstrakcije IPC koji je u duhu objektno-orjentisanog programiranja.

```
/** Primer definisanja AIDL interfejsa */
interface IRemoteService {
  /** Lista metoda koje servis podržava. */
  int metod1();
  void metod2(int anInt, boolean aBoolean, float aFloat);
  oneway void metod3(CustomClass aClass);
}
```

Slika 3.3.1 *AIDL* interfejs

Način povezivanja klijenta i servisa je veoma sličan kao kod *Messenger* mehanizma. I u ovom slučaju se iz servisa prosleđuje generički *IBinder* interfejs, s tim što sada nije *Messenger* objekat koji implementira taj interfejs, već je to *Proxy* objekat koji ima definisane metode Service objekta. Klient pretvara (*eng. cast*) generički *IBinder* u *Proxy* objekat statičkom metodom *asInterface()* koja je definisana u okviru *Stub* objekta (Slika 3.3.2).

```
IRemoteService mIRemoteService;
private ServiceConnection mConnection = new ServiceConnection() {
  // Povratni poziv koji će se desiti kad se konekcija
  // sa servisom uspostavi
  public void onServiceConnected(ComponentName className, IBinder service)
        {
    // cast-ovanje Binder objekta
    mIRemoteService = IRemoteService.Stub.asInterface(service);
  }

  // povratni poziv koji će se desiti kada se konekcija sa serverom prekine
  public void onServiceDisconnected(ComponentName className) {
```

```
    mIRemoteService = null;
  }
};
```

Slika 3.3.2 Primer implementacije AIDL mehanizma IPC kod klijenta

Sa druge strane, *Service* klasi u *onBind()* metodi vraćamo *Stub* objekat tog servisa (Slika 3.3.3).

```
public class RemoteService extends Service {
  @Override
  public void onCreate() {
    super.onCreate();
  }

  @Override
  public IBinder onBind(Intent intent) {
    // Vraćanje interfejsa
    return mBinder;
  }

  private final IRemoteService.Stub mBinder = new IRemoteService.Stub() {
    int metod1(){...}
    void metod2(int anInt, boolean aBoolean, float aFloat){...}
    void metod3(CustomClass aClass){...}
   }

  };
}
```

Slika 3.3.3 Primer implementacije AIDL mehanizma IPC kod servisa

Proxy klasa je privatna klasa u okvuru *Stub* klase i ona sadrži sve metode koje su definisane *AIDL* interfejsom koji implementira *Stub* klasa. Tačnije, *Proxy* klasa služi tome da izloži Stub klijentu. Android build-tool je deo Android SDK koji generiše Java interfejse od *AIDL* Interfejsa u **/gen** folderu i u njima generiše Stub i Proxy klase. *Binder* okvir koristi ove klase kao bi omogućio da IPC između servisa i klijenta.

```
package com.example.app;

public interface IRemoteService extends android.os.IInterface
{
public static abstract class Stub extends android.os.Binder
implements com.example.app.IRemoteService {
…
public static com.example.app.IRemoteService asInterface(
```

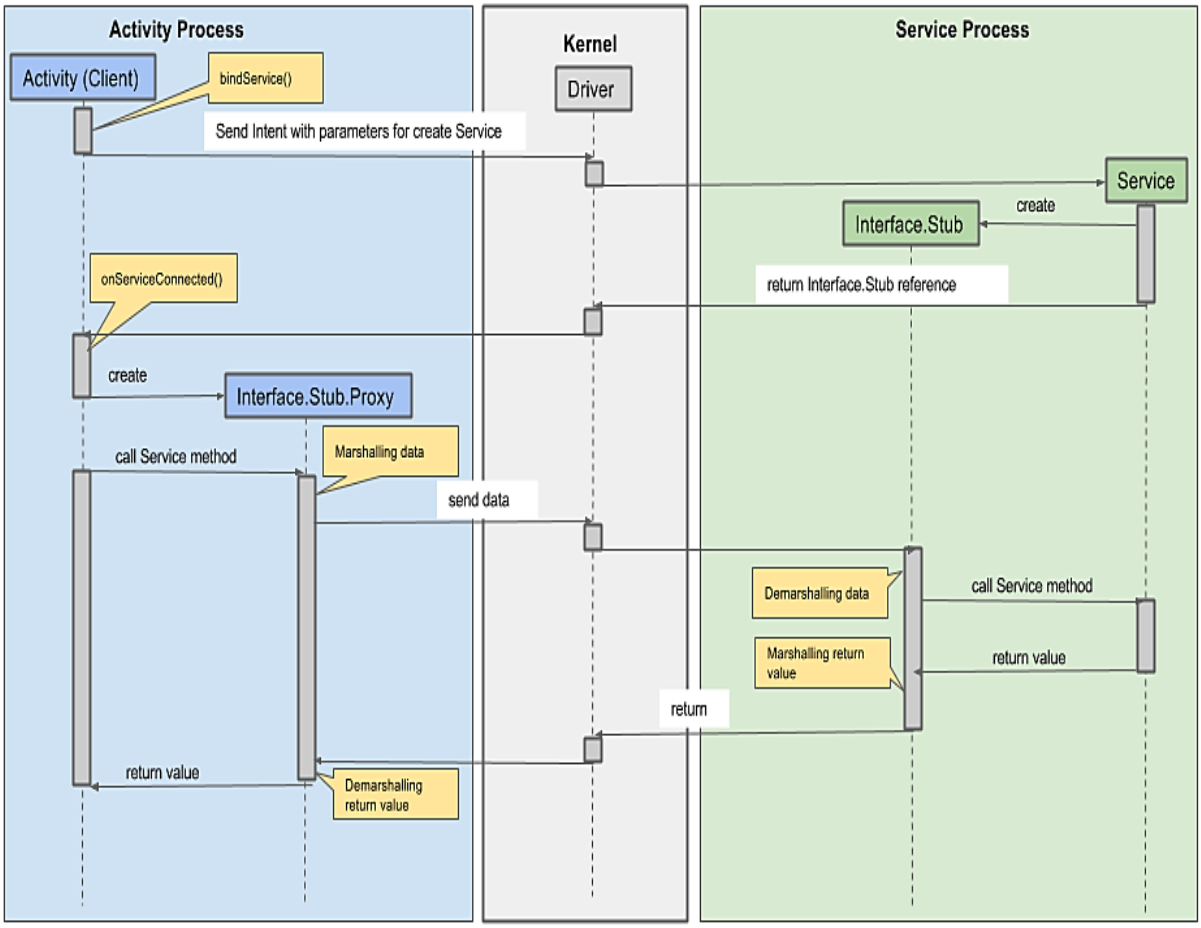
```
android.os.IBinder obj) {
…
return new com.example.app.IRemoteService.Stub.Proxy(obj);
}
…
public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel
reply, int
flags) throws android.os.RemoteException {
...
}

private static class Proxy implements com.example.app.IRemoteService {
private android.os.IBinder mRemote;
…
public void metod1() throws android.os.RemoteException {
...
}
public void metod2(int anInt, boolean aBoolean, float aFloat) throws
android.os.RemoteException {
...
}
public void metod3(CustomClass aClass) throws android.os.RemoteException {
...
}
}
}
```

Slika 3.3.4 Auto-generisana *Stub* klasa

AIDL mehanizam treba koristiti ukoliko postoji potreba da klijenti iz raznih aplikacija komuniciraju sa servisom na više niti. To znači da u ovoj implementaciji nikad ne možemo podrazumevati nit na kojoj će se servis izvršavati. *Binder* drajver dodeljuje nit na kojoj će se servis izvršiti iz sistemskog „bazena“ niti, tako da za dva kijenta on dodeljuje dve niti. Iz ovog razloga servis mora biti više-nitno bezbedan i to programer mora da implementira ukoliko želi da koristi *AIDL*. Takođe, zbog toga što ovaj mehanizam podržava rad sa više niti on daje nabolje performanse, za razliku od *Messenger* mehanizma gde servis može da obradi samo jednu poruku u datom trenutku.

Metode koje su deklarisane u *AIDL* interfejsu mogu biti asinhrone(„One-way“) i sinhrone („Two-way“). Metodu deklarišemo kao asinhronu koristeći ključnu reč „oneway“. Na slici 3.3.1 metod3(), metoda je deklarisana kao asinhrona. Sve metode u *AIDL* interfejsu su sinhrone ukoliko programer ne definiše drugačije. Ovo znači da će poziv metode *Proxy* objekta blokirati nit na klijentu sve dok servis ne vrati odgovor. Ukoliko programer definiše *Proxy* metode kao asinhrone i u ovom slučaju metoda ne vraća nikakvu vrednost, tako da poziv, metode neće blokirati nit na kojoj se izvršava klijentski kod. U ovom slučaju imamo problem što ne dobijamo nikakav odgovor od strane servisa. Dvosmernu asinhronu komunikaciju je moguće implementirati tako što ćemo deklarisati dva interfejsa koji implementiraju „One-way“ pozive.

Ovo se postiže tako što, koristeći „one-way“ pozive, mi pošaljemo servisu objekat preko kojeg će on obavestiti klijenta kada bude odradio određenu radnju. Na slici 3.3.5 je prikazana *AIDL* implementacija inter-procesne komunikacije između klijenta i servisa.



Slika 3.2.1 Prikaz *AIDL* IPC između klijenta i servisa

4. *Binder* okvir

Binder okvir je prvobitno razvijen kao *OpenBinder* od strane *Be Inc.* i *Palm Inc.* pod vođstvom Dianne Hackborn. *OpenBinder* je zamišljen kao komponenta operativnog sistema sa ciljem da obezbedi viši nivo apstrakcije sistemskih servisa u odnosu na tradicionalne operativne sisteme. *OpenBinder* je trebao da omogući vezivanje sa finkcijama i podacima iz jednog okruženja u drugo. *OpenBinder* implementacija radi pod Linux-om tako da on proširuje njegove postojeće mehanizme za IPC. *Binder* okvir predstavlja prilagođeni *OpenBinder* za Android operativni sistem.

4.1 Zašto *Binder* okvir?

Binder okvir, na prvom mestu, obezbeđuje *sigurnost* i izolaciju svake aplikacije, zato što se svaka od njih pokreće u zasebnom procesu i na sopstvenoj instanci virtuelne mašine. Upravljenje memorije je na Android OS malo izmenjeno u odnosu na Linux OS i prilagođeno uređajima sa ograničenim resursima (mobilni telefoni, tableti itd.). Jedna od glavnih razlika je u tome da Android OS može da „ubije“ nepotrebne procese i oslobodi resurse (uglavnom RAM memoriju) koje je taj proces držao. Android OS održava klasifikaciju procesa i oni imaju različite prioritete. Procesi u Android mogu biti (redosled po prioritetu): *prednji, vidljivi, servisni, procesi u pozadini i prazni procesi.* Ukoliko proces višeg prioriteta traži resurse, a svi resursi uređaja su zauzeti, Android OS osobađa resurse koje drže procesi najnižeg prioriteta. Operativni sitemi koji imaju ovakvo upravljanje memorijom uređaja se nazivaju *„low-memory killer*“ (brisanje nepotrebnih procesa) operativni sistemi. *Binder* okvir je veoma dobaro rešenje za ovakva okruženja, zato što on obezbeđuje da ukoliko se neki proces skloni iz memorije to ne utiče na funkcionisanje ostalih procesa koji su od njega zavisni. Shodno tome se za *Binder* okvir može reći da doprinosi *stabilnosti* operativnog sistema.

4.2 Karakteristike *Binder* okvira

Binder okvir implementira inter-procesnu komunikaciju u duhu objekno orjentisanih (OO) jezika. To znači da programer poziv metoda I*Binder* interfejsa koristi isto kao i pozive metoda objekata u OO jezicima. Programer ne mora da brine u kom procesu se nalazi servis

kome pristupa iz klijenta, jer ga on koristi isto kao lokani objekat. Sa druge strane, izlaganje servisa drugim aplikacijama (procesima) se radi implementiranjem *AIDL* interfejsa u servisu. *Binder* okvir poseduje sledeće karakteristike :

- Identifikacija pošaljioca zahteva (poruke) kod primaoca preko jedinstvenog identifikacionog broja ili po broju procesa. Ovo je važno iz bezbednosnih razloga. Zato što servis može videti da li određeni klijent ima dozvolu (*eng. permission*) za određene akcije.
- Sposobnost da šalje fajl dekriptore (*eng. File descriptors*) između procesa. Bitno jer ukoliko želimo da slušamo neku pesmu na plejeru koji se nalazi na drugom procesu ne moramo slati ceo fajl pesme.
- Jedinstveno mapiranje objekta širom OS. Ukoliko se referenca nekog objekta pošalje nekom drugom procesu, a zatim taj proces prosledi nekom trećem, procesu sve reference će pokazivati na isti objekat u sistemu.
- *Binder* objkat za jedan servis je jedinstven tako da se on može koristiti kao bezbedonosni token prilikom autorizacije
- Prethodno navedeni „one-way“ i „two-way“ pozivi metoda
- Brojanje referenci koje pokazuju na *IBinder* objekat (*eng. Reference counting*)
- Mehanizam obaveštavanja svih referenci koje pokazuju na *IBinder* objekat ukoliko je on uklonjen iz memorije (*eng. Death notification*). Ovo je veoma bitno zato sto svaki proces može biti prekinut u svakom trenutku.

4.2 *Parcel*, *Marshalling* i *Unmarshalling*

Jenda od najvažnijih karakteristika inter-procesnih mehanizama je ta na koji način i koliko brzo transportuju podatke između procesa. *Binder* okvir ima svoj tip podataka koje može da transportuje širom operativnog sistema. To su Parcel objekti. Proces pakovanja podataka iz Java formata u *Parcel* format se naziva *marshaling* a proces raspakivanja se naziva *unmarshaling.* Pakovanje osnovnih Java tipova podataka (*Array, String, int, boolean, double, short,itd.)* je definisano u okviru *Binder* okvira. Ukoliko želimo da šaljemo objekte klasa širom sistema, taj objekat mora da implementira *Parcelable* interfejs. U okviru ovog interfejsa su deklarisane dve metode koje objekat mora da implementira. Prva je *writeToParcel(Parcel parcel, int flag)* u kojoj se definiše *marshaling* i druga *createFromParcel(Parcel pracel)* u kojoj se definiše *unmarshaling.* Na slici 4.2.1 je prikazana implementacija *Pracelable* interfejsa za klasu Student.

```
import android.os.Parcel;
import android.os.Parcelable;

public class Student implements Parcelable{

  String mIme;
  int mGodine;
  String mAdresa;

  @Override
  public int describeContents() {
    // TODO Auto-generated method stub
    return 0;
  }

  /**
  * Snimanje podataka studenta u Parcel objekat
  **/
  @Override
  public void writeToParcel(Parcel dest, int flags) {
    dest.writeString(mIme);
    dest.writeInt(mGodine);
    dest.writeString(mAdresa);
  }

  /**
  * A konstruktor za inicijalizaciju Student objekta
  **/
  public Student(String sName, int sAge, String sAddress){
    this.mIme = sName;
    this.mGodine = sAge;
    this.mAdresa = sAddress;
  }

  /**
  * Privatni konstruktor za pravljenje Student objekata iz Parcel objekta.
  * Konstruktor se poziva u createFromParcel(Parcel source) metodi
  **/
  private Student(Parcel in){
    this.mIme = in.readString();
    this.mGodine = in.readInt();
    this.mAdresa = in.readString();
  }

  public static final Parcelable.Creator<Student> CREATOR = new
Parcelable.Creator<Student>() {

    @Override
    public Student createFromParcel(Parcel source) {
      return new Student(source);
    }
```

```
    @Override
    public Student[] newArray(int size) {
      return new Student[size];
    }
  };
}
```

Slika 4.2.1 Primer implementacije Parcelable interfejsa u okvuru Student klase

Implementacija transporta podataka preko *Parcel* objekata daje neverovatno dobre performanse u poređenju sa klasicnom serijalizacijom objekata. U poređenju sa Java serijalizacijom, *Pacel* objekti su duplo brži. Razlog je to što kod Java serijalizacije *marshaling* radi Java Virtual Mashine (JVM) koristeći refleksiju na osnovu tipova, dok kod Parcelable svaki objekat definiše sopstveni mehanizam za *marshaling*.

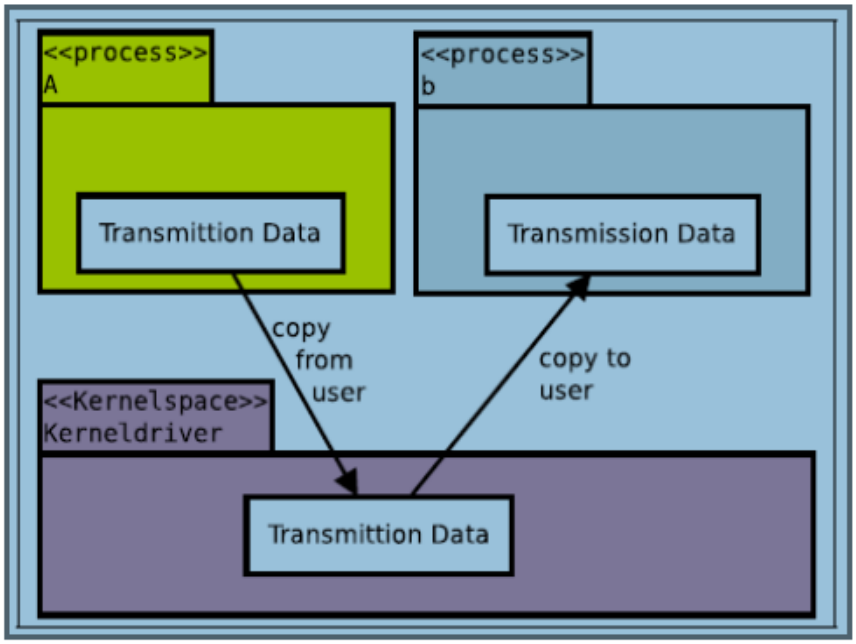
4.3 Model komunikacije

Model komunikacije na Android OS se zasniva na principu servis-klijent. Prilikom definisanja servisa on je nevidljiv klijentu. Android OS obezbeđuje mehanizam registracije svih servisa koji žele da izlože svoje usluge izvan svojih procesa. Na Android OS postoji *sistemski servis* u kome se registruju ostali servisi. Android OS ima svoje uobičajene servise koji su već registrovani prilikom startovanja OS-a i kojima klijent može da pristupi, i to su : *Location Service, Notification Service, Package Service i dr.* Svim uobičajenim (*eng. default*) servisima se pristupa preko njihovih Manager objekata. Ovi objekti ne predstavljaju ništa drugo nego *Proxy* tih servisa. Sa druge strane, servisi koje definiše programer se ne registruju u *sistemski servis* prilikom njihovg pravljenja, već prilikom prvog slanja njegovog *Binder* objekta izvan granica procesa. *Binder* drajver će zapamtiti ovaj *Binder* objekat i napraviti svoju referencu na njega koju će kasnije prosleđivati svim klijentima zainteresovanim za ovaj servis.

4.4 *Binder* drajver

Binder okvir,koji je implementiran od strane *Binder* drajvera i koji je deo Linux kernela, je proširen od strane Google programera koji su ga prilagodili svojim potrebama. *Binder* drajver koristi više-nitni sistem kako bi maksimizovao iskorišćenost više-jezgarnih procesora. *Binder* drajver je napisan u C/C++, što je još jedan od razloga zašto su u *Binder* okviru uvedeni *Pracel* objekti. *Pracel* objekti su razumljivi i C/C++ baš kao i Java-i. Svako slanje poruke između procesa na nivou kernela se odvija putem transakcija. Svaka transakcija sadrži neke podatke. Ukoliko želimo da podatke pošaljemo nekom udaljenom procesu, *Binder* driver će iskopirati te podatke

iz memorijskog prostora lokalnog procesa u memorijski prostor kernela, a zatim u memorijski prostor udaljenog procesa (Slika 4.4.1).



Slika 4.4.1 Slanje podataka između procesa

Veličina podataka koja može biti poslata jednom transakcijom je 1MB. Ovakva mehanizam nije dobar za slanje velikih količina podataka tako da *Binder* drajver takođe omogućava korišćenje i deljene memorije (*eng. Shared memory*) kako bi načinio podatke dostupnim svim procesima bez kopiranje u memoriju svakog procesa. Na slici 4.4.2 je prikazana struktura *Binder* objekta na najnižem nivou.

```
struct binder_write_read {
signed long write_size; /* veličina bajtova za upis */
signed long write_consumed; /* bajtovi za upis */
unsigned long write_buffer;
signed long read_size; /* veličina pročitanih bajtova */
signed long read_consumed; /* pročitani bajtovi */
unsigned long read_buffer;
};
```

Slika 4.4.2 Implementacija *Binder* objekta u kernelu

4.5 Ograničenja *Binder* okvira

Binder okvir ima sledeća ograničenja:

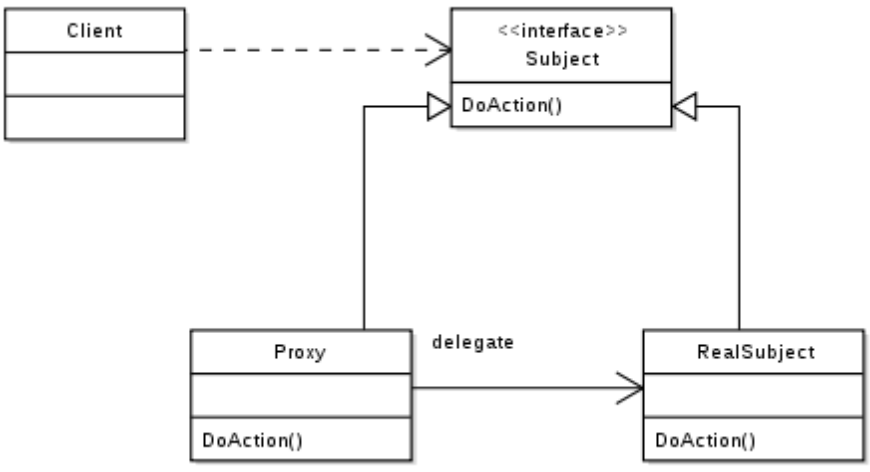
- Dozvoljava maksimalno 15 niti po procesu
- Postojanje samo jednog *Service Manager* objekta (sistemskog servisa)
- Ograničavanje veličine slanja podataka na 1 MB po transakciji.

4.6 Karateristični uzorci dizajna *Binder* okvira

Za *Binder* okvir su karakteristična tri dizajn paterna i to su: *Proxy*, *Mediator* i *Bridge*.

4.6.1 *Proxy* dizajn uzorak

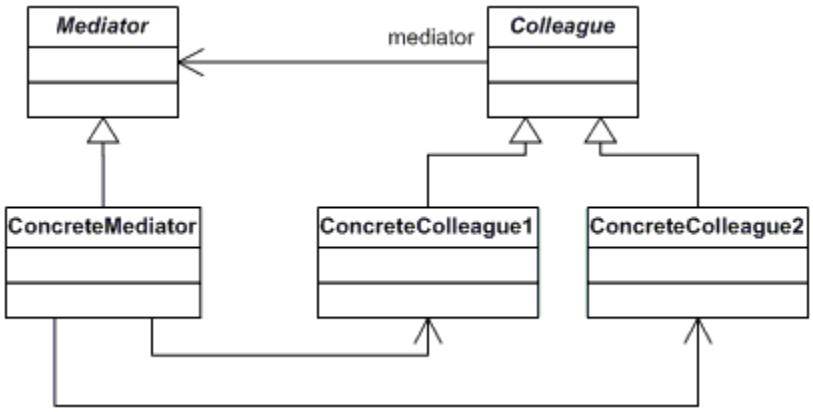
Proxy u svom najopštijem obliku predstavlja klasu koja služi kao interfejs za neki drugi objekat. *Proxy* objekat može biti interfejs ka svemu: internet konekciju, velike objekte u memoriji, fajlove ili ka bilo kojem drugom resursu koji je veoma skupo ili nemoguće duplirati u nekom sistemu. U našem slučaju *IBinder* objekat implementira *Proxy* dizajn uzorak, koji koristi klijentska strana kako bi pristupila implementaciji servisa, bez obzira na to u kom se adresnom prostoru on nalazi. Na slici 4.6.1 je prikazan *UML* dijagram *Proxy* dizajn uzorka.



Slika 4.6.1 UML dijagram *Proxy* dizajn uzorka

4.6.2 *Mediator* dizajn uzorak

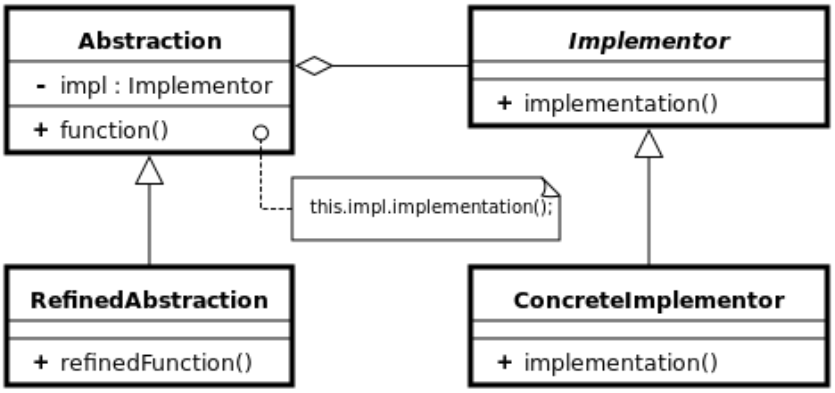
Mediator dizajn uzorak definiše objekat koji opisuje način na koji skup objekata komunicira između sebe. Obično je program sačinjen od mnogo klasa. Sa povećanjem broja klasa problem komunikacije između njih postaje sve veći. *Mediator* dizajn uzorkom se enkapsulira komunikacija između objekata na jednom mestu, u Mediator objektu. Na ovja način objekti ne komuniciraju direktno nego preko posrednika. Ovo dosta smanjuje zavisnost u komunikaciji između objekata. Primer implementacije ovog uzorka može biti puštanje muzike korišćenjem sistemskog MediaPlayer servisa. Na slici 4.6.2 je prikazan UML dijagram *Mediator* dizajn uzorka.



Slika 4.6.2 UML dijagram *Mediator* dizajn uzorka

4.6.3 *Bridge* dizajn uzorak

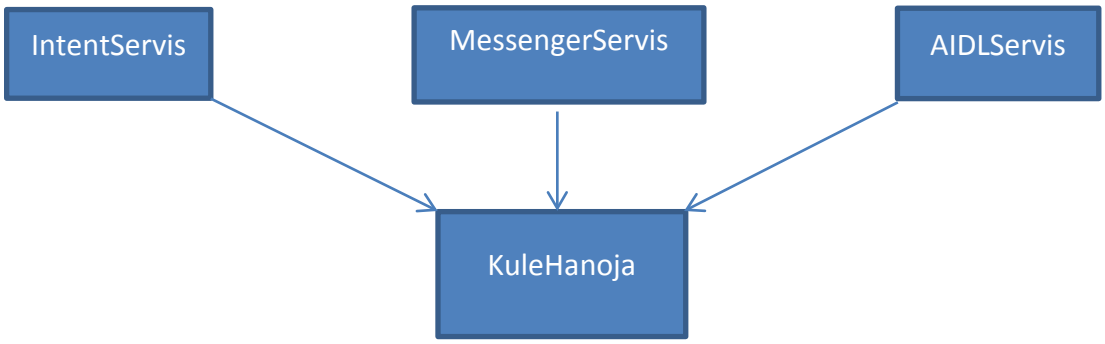
Prilikom razvoja softvera *Bridge* dizajn uzorak ima za cilj da razdvoji apstrakciju od implementacije, tako da one budu nezavisne. Ovo se u praksi postiže grupisanjem klasa pod jedan interfejs, a celokupna logika se umesto korišćenja klasa implementira kroz intefejs. U našem slučaju svi servisi implementiraju *IBinder* interfejs. Na slici 4.6.3 je prikzan UML dijagram *Bridge* dizajn uzorka.



Slika 4.6.3 UML dijagram *Bridge* dizajn uzorka

5. Implementacija aplikacije „Kule Hanoja“

U ovom poglavlju će biti opisan praktični deo inter-procesne komunikacije. On će biti prikazan kroz 4 aplikacije. Tri aplikacije u sebi sadržati samo servis koji će raditi istu stvar. Svaki od servisa će iplementirati mehanizam IPC baziran na *Binder* okviru na načine koji su opisani u poglavlju 3 (*Intent* mehanizam, *Mesenger* mehanizam i *AIDL* mehanizam). Servisi će generisati poruke i slati ih klijentu. Ideja je da servisi generišu pokrete u problemu „Kule Hanoja“ za *n* elemenata. Za n elemenata će biti izgenerisano 2^n-1 pokreta, pa će kroz povećavanje broja *n* biti prikazana vremena za koje se poruke razmene. Na ovaj način će biti prikazane razlike u brzini rezmene podataka u zavisnosti od načina na koji mehanizm inter-procesne komunikacije implementiramo. Četvrta aplikacija je klijent koji može da komunicira sa sva tri servisa (Slika 5.1).



Slika 5.1.1 Način komunikacije između aplikacija

5.1 Problem Kula Hanoja

Ovaj problem tačnije zadatak je po prvi put sastavio francuski matematčar Lucas Edvard. Osim toga, takođe postoji legenda o ovom problemu. Ona kaže da su u hramu Brahma u Hanoju postojala tri stuba. Na levom stubu su stajala 64 diska različite veličine koncentrično poređani od najvećeg na dnu do najmanjeg na vrhu stuba. Monasi su imali zadatak da prebace sve diskove sa prvog stuba na treći koristeći drugi stub, ukoliko ima potrebe, ali uz jedno pravilo, nikad se veći disk ne sme naći na manjem.

Matematički je dokazano da je minimalan broj pokreta potreban da se N diskova prebaci sa prvog stuba na treći 2^n-1. U ovom slučaju za 64 diskova minimalan broj pokreta bi bio 18,446,744,073,709,551,615.

5.2 Opis aplikacija

Prva aplikacija „Kule Hanoja“ se sastoji iz četri dela. U prvom delu je implementirana igra, koja dozvljava korisniku da izabere broj diskova (od 1 do 6). Na slici 5.2. je prikazan grafički interfejs ove igre.

Slika 5.2.1 Grafički interfejs aplikacije „Kule Hanoja“

Na slici 5.2.2 je prikazana implementacija grafičkog iterfejsa putem XML jezika. Na Android OS postoje predefinisani XML elementi sa svojim takođe predefinisanim atributima.

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:gravity="center_vertical"
  android:padding="@dimen/activity_vertical_margin"
  tools:context="com.masterrad.aleksandarsilic.kulehanoja.IgraActivity">

  <LinearLayout
    android:id="@+id/sticks_layout"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:gravity="center"
    android:orientation="horizontal"
    android:paddingLeft="12dp"
    android:paddingRight="12dp"
    android:paddingTop="150dp">

    <TextView
      android:layout_width="0dp"
      android:layout_height="200dp"
          android:layout_marginLeft="40dp"
      android:layout_marginRight="40dp"
      android:layout_weight="0.3"
      android:background="#2255FF" />

    <TextView
      android:layout_width="0dp"
      android:layout_height="200dp"
      android:layout_marginLeft="40dp"
      android:layout_marginRight="40dp"
      android:layout_weight="0.3"
      android:background="#2255FF" />

    <TextView
      android:layout_width="0dp"
      android:layout_height="200dp"
      android:layout_marginLeft="40dp"
      android:layout_marginRight="40dp"
      android:layout_weight="0.3"
      android:background="#2255FF" />
  </LinearLayout>
  <LinearLayout
    android:id="@+id/disks_layout"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:gravity="center"
    android:orientation="horizontal"
    android:paddingLeft="12dp"
    android:paddingRight="12dp"
```

```
    android:paddingTop="150dp">

    <!-Prvi stap-->
    <LinearLayout
      android:id="@+id/sticks_A_layout"
      android:layout_width="0dp"
      android:layout_height="200dp"
      android:layout_marginLeft="2dp"
      android:layout_marginRight="2dp"
      android:layout_weight="0.3"
      android:alpha="1"
      android:orientation="vertical">

      <View
        android:layout_width="wrap_content"
        android:layout_height="0dp"
        android:layout_weight="1" />
    </LinearLayout>
    <!-Drugi stap-->
    <LinearLayout
      android:id="@+id/sticks_B_layout"
      android:layout_width="0dp"
      android:layout_height="200dp"
      android:layout_marginLeft="2dp"
      android:layout_marginRight="2dp"
      android:layout_weight="0.3"
      android:alpha="1"
      android:orientation="vertical">

      <View
        android:layout_width="wrap_content"
        android:layout_height="0dp"
        android:layout_weight="1" />
    </LinearLayout>
    <!-Treći stap-->
    <LinearLayout
      android:id="@+id/sticks_C_layout"
      android:layout_width="0dp"
      android:layout_height="200dp"
      android:layout_marginLeft="2dp"
      android:layout_marginRight="2dp"
      android:layout_weight="0.3"
      android:alpha="1"
      android:orientation="vertical">

      <View
        android:layout_width="wrap_content"
        android:layout_height="0dp"
        android:layout_weight="1" />
    </LinearLayout>
  </LinearLayout>

  <TextView
```

```
    android:layout_width="match_parent"
    android:layout_height="5dp"
    android:layout_marginTop="350dp"
    android:background="#2255FF" />

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:paddingRight="15dp"
      android:paddingTop="10dp"
      android:text="Izaberite broj diskova:"
      android:textSize="18sp" />

    <Spinner
      android:id="@+id/broj_diskova_spinner"
      android:layout_width="50dp"
      android:layout_height="wrap_content"
      android:entries="@array/spinner_items" />

    <LinearLayout
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:orientation="vertical">

      <Button
        android:id="@+id/startuj_igru_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/start_button_title" />

      <Button
        android:id="@+id/resetuj_igru_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:enabled="false"
        android:textSize="14sp"
        android:text="@string/reset_button_title" />
    </LinearLayout>

  </LinearLayout>
</FrameLayout>
```

Slika 5.2.2 Implementacija grafičkog interfejsa za aplikaciju „Kule Honoija“ u XML

Preostala tri dela aplikacije predstavljaju klijente za udaljene servise (*Intent* servis, *Messenger* servis i *AIDL* sevis). Svaki od klijenta se povezuje na svoj udaljeni servis i komunicira sa njim na određen način. Svaki od klijenta može da pošalje udaljenom servisu broj diskova za koje servis treba da izračuna pokrete da bi se oni prebacili sa prvog štapa na poslednji.

Implementacija klijent-servis komunikacije, koja se odvija putem *Intent* mehanizma, prikazana je u kodu na slikama 5.2.3 i 5.2.4. Klijent na početku komunikacije pokreće servis putem *Intent* objekta koji u sebi sadrži informaciju o broju diskova. Kad se servis startuje, on kreće sa izračunavanjem pokreta. Servis za svaki pokret šalje klijentu *Intent* objekat. Sa druge strane, klijent u sebi sadrži registrovan *Prijemnik (*opisan u poglavlju 2.5) koji prima *Intent* objekte koje šalje servis i pamti vreme kada su oni stigli.

```
public class IntentServisActivity extends ActionBarActivity {

  private Button mRunButton;
  private TextView mLogTextView;
  private Context mContext;
  private BroadcastReceiver mInentServiceReceiver;
  private EditText mBrojDiskova;
  private Intent mService1Intent;
  private int mMaksimalanBrojPokreta;
  DateFormat formatter = new SimpleDateFormat("mm:ss:SSS");
  private int mDestetinaPokreta;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_intent_service);
    mRunButton = (Button) findViewById(R.id.run_button);
    mRunButton.setEnabled(true);
    mLogTextView = (TextView) findViewById(R.id.message_log_textView);
    mBrojDiskova = (EditText) findViewById(R.id.broj_diskova);
    mContext = this;
    getSupportActionBar().setTitle(R.string.title_intent_service_client);
    mService1Intent = new Intent(getString(R.string.SERVICE_1_INTENT));
  // Definisanje akcije prilikom klika na dugme koje pokreće izračunavanje
    mRunButton.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
        if (mBrojDiskova.getText().toString().trim().equals("")) {
          new AlertDialog.Builder(mContext).setMessage
        ("Morate unteti broj diskova pre nego sto
                pokrenete izvrsavanje").show();

        }else {
        // Pokretanje servisa slanjem Intent objekta
          int n = Integer.decode(mBrojDiskova.getText().toString()).intValue();
          mService1Intent.putExtra("brojDiskova", n);
          startService(mService1Intent);
          mMaksimalanBrojPokreta = (int)Math.pow(2 * 1.0, n * 1.0)-1;
          mDestetinaPokreta = mMaksimalanBrojPokreta / 10;
          mRunButton.setEnabled(false);
        }
}});
  // Prijemnik za Intent objekte. Svaki primnjeni Intent objekat predstavnja jedan
```

```
  // pokret u igri
    mInentServiceReceiver = new BroadcastReceiver() {
      int brojPokreta;
      long pocentoVreme;

      @Override
      public void onReceive(Context context, Intent intent) {
        brojPokreta = intent.getIntExtra("brojPokreta", 0);
        if(brojPokreta==1){
          java.util.Date date = new java.util.Date();
          pocentoVreme=date.getTime();
          mLogTextView.setText("");
        }

        if (mMaksimalanBrojPokreta >=brojPokreta) {
          if (brojPokreta % mDestetinaPokreta == 0) {
            java.util.Date date = new java.util.Date();
            String message ="Proteklo vreme za " + brojPokreta
            + " pokreta je "
             + formatter.format(date.getTime() - pocentoVreme)
            + "\n"+mLogTextView.getText();
            mLogTextView.setText(message + "\n");
            }
        } else {
          java.util.Date date = new java.util.Date();
          mLogTextView.setText("Vreme potrebno da se "
          + mBrojDiskova.getText()
          + " diskova, prebaci \nsa stapa A na C je "
          + formatter.format(date.getTime() - pocentoVreme)
          + " i tom prilikom se izvrsi " + (int) mMaksimalanBrojPokreta
          + " pokreta\n" + mLogTextView.getText());
          mRunButton.setEnabled(true);
        }
      }
    };
    IntentFilter intentFilter = new IntentFilter("com.master.rad.KILJENT");
  // Registrovanje prijemnika
    registerReceiver(mInentServiceReceiver, intentFilter);
  }
}
```

Slika 5.2.3 Implementacija klijenta koji komunicira sa udaljenim servisom putem *Intent* objekata

```
public class ServiceIntent extends Service {
  Context mContext;
  int mBrojDiskova;
  Intent mKlijentIntent;
  int maksimalanBrojPokreta;

  @Override
  public IBinder onBind(Intent intent) {
    return null;
  }

  @Override
  public void onCreate() {
    super.onCreate();
    mContext = this;
    mKlijentIntent = new Intent("com.master.rad.KILJENT");
  }
```

```
  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
    super.onStartCommand(intent, flags, startId);
    mBrojDiskova = intent.getIntExtra("brojDiskova", 0);
         maksimalanBrojPokreta = (int) (Math.pow(2 * 1.0, mBrojDiskova * 1.0)-1);
   //Posebna nit u servisu na kojoj se izvrsava algoritam
    Thread t = new Thread(new Runnable() {
      int brojPokreta = 0;

      @Override
      public void run() {
        doTowers(mBrojDiskova, 'A', 'B', 'C');
      }
      private void doTowers(int topN, char from, char inter, char to) {
        if (topN == 1) {
                  mKlijentIntent.putExtra("brojPokreta", brojPokreta);
          mContext.sendBroadcast(mKlijentIntent);
                 return;
        } else {
          doTowers(topN - 1, from, to, inter);
                  mKlijentIntent.putExtra("brojPokreta", brojPokreta);
          mContext.sendBroadcast(mKlijentIntent);
          doTowers(topN - 1, inter, from, to);
        }
      }
    });
    t.start();
    return START_NOT_STICKY;
  }
```

Slika 5.2.4 Implementacija udaljenog servisa koji komunicira sa klijentom putem *Intent* objekta

Implementacija klijent-servis komunikacije, koja se odvija korišćenjem *Messenger* objekta, prikazana je u kodu na slikama 5.2.5 i 5.2.6. Servis implementira svoj *Messenger* objekat čiju referencu šalje klijentu kada on inicira vezivanje na ovaj servis. Klijent se vezuje na servis metodom *bindService(...),* čiji je jedan od argumenata *ServiceConection* povratni poziv. Putem povratnog poziva klijent dobija referencu na *Messenger* objekat, koji se nalazi u servisu. *Messenger* objekat servisa podržava dva različita poziva ka servisu, jedan za registraciju klijenata i drugi za pokretanje izračunavanja pokreta za N diskova problema „Kule Hanoja“. Prilikom registracije klijenta u servis, klijent šalje referencu na svoj *Messenger* objekat koji podržava jedan poziv ka klijentu i to je vraćanje pokreta. Detaljnije o implementaciji *Messenger* mehanizma za IPC možete videti u poglavlju 3.2.

```
public class MessengerActivity extends ActionBarActivity {

  private Button mRunButton;
  private TextView mLogTextView;
  private EditText mBrojDiskova;
  private Context mContext;
  private HandlerThread handlerThread =new HandlerThread("MessengerClient");

  Messenger mMessenger;
  private int mMaksimalanBrojPokreta=0;
  private int mDestetinaPokreta = 1;
  private Messenger mService=null;
```

```
  // Objekat kojim se inicira konekcija sa udaljenim Messenger servisom
  private ServiceConnection mConnection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
      mService=new Messenger(service);
      try {
        Message msg =
essage.obtain(null,Constants.MESSENGER_SERVICE_REGISTRUJ_KLIJENTA);
        msg.replyTo = mMessenger;
        mService.send(msg);
      } catch (RemoteException e) {
      }
      mRunButton.setEnabled(true);
    }

    @Override
    public void onServiceDisconnected(ComponentName name) {
      mService=null;
      mRunButton.setEnabled(false);
    }
  };

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    handlerThread.start();
    mMessenger = new Messenger(new IncomingHandler(handlerThread));
    setContentView(R.layout.activity_messenger_service);
    mRunButton = (Button) findViewById(R.id.run_button);
    mRunButton.setEnabled(false);
    mLogTextView = (TextView) findViewById(R.id.message_log_textView);
    mBrojDiskova = (EditText) findViewById(R.id.broj_diskova);
    mContext = this;
    getSupportActionBar().setTitle(R.string.title_messeger_service_klient);
    // Definisanje akcije prilikom klika na dugme koje pokreće izračunavanje
    mRunButton.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {

        if (mBrojDiskova.getText().toString().trim().equals("")) {
          new AlertDialog.Builder(mContext)
                  .setMessage("Morate unteti broj diskova pre nego sto pokrenete
                   izvrsavanje").show()
        }else {
         int n = Integer.decode(mBrojDiskova.getText().toString()).intValue();
          try {
            mService.send(Message.obtain(null,
              Constants.MESSENGER_SERVICE_POKRENI_IZVRSAVANJE, n, 0));
          } catch (RemoteException e) {

          }
          mMaksimalanBrojPokreta = (int) Math.pow(2 * 1.0, n * 1.0) - 1;
          mDestetinaPokreta = mMaksimalanBrojPokreta / 10;
          mRunButton.setEnabled(false);
        }
      }
     });

  getApplicationContext().bindService(
  new Intent("com.messenger.service.ACTION"), mConnection, Context.BIND_AUTO_CREATE);
  }
  class IncomingHandler extends Handler {
```

```
    public IncomingHandler(HandlerThread handlerThread){
      super(handlerThread.getLooper());
    }

    long pocentoVreme;
    DateFormat formatter = new SimpleDateFormat("mm:ss:SSS");
    @Override
    public void handleMessage(Message msg) {

      switch (msg.what) {
        case Constants.MESSENGER_KLIJENT_PORUKA_POKRET:
          final int brojPokreta=msg.arg1;
          if(brojPokreta==1){
            java.util.Date date = new java.util.Date();
            pocentoVreme=date.getTime();
            runOnUiThread(new Runnable() {
              @Override
              public void run() {
                mLogTextView.setText("");
              }
            });
          }

          if (mMaksimalanBrojPokreta >= brojPokreta) {
            if (brojPokreta % mDestetinaPokreta == 0) {
                final java.util.Date date = new java.util.Date();
                final String message ="Proteklo vreme za " +brojPokreta
                            + " pokreta je "
               + formatter.format(date.getTime() - pocentoVreme)
                            + "\n"+mLogTextView.getText();

              runOnUiThread(new Runnable() {
                @Override
                public void run() {
                  mLogTextView.setText(message + "\n");
                }
              });
            }
          } else {
            final java.util.Date date = new java.util.Date();
   runOnUiThread(new Runnable() {
      @Override
      public void run() {
    mLogTextView.setText("Vreme potrebno da se " + mBrojDiskova.getText() +
            " diskova, prebaci \nsa stapa A na C je " +
                    formatter.format(date.getTime() - pocentoVreme)
                    + " i tom prilikom se izvrsi " + (int) mMaksimalanBrojPokreta
                    + " pokreta\n" + mLogTextView.getText());
            mRunButton.setEnabled(true);
              }
            });
          }
          break;
        default:
          super.handleMessage(msg);
      }
    }
  }
}
```

Slika 5.2.5 Implementacija *Messenger* klijenta

```
public class MessengerService extends Service {
  int mMaksimalanBrojPokreta=0;
  HandlerThread handlerThread = new HandlerThread("MessengerService");
  ArrayList<Messenger> mClients = new ArrayList<Messenger>();
  private Messenger mMessenger;

  @Override
  public void onCreate() {
    super.onCreate();
    handlerThread.start();
   // Kreiranje Messenger objekta preko koga se odvija razmena porka
    mMessenger= new Messenger(new IncomingHandler(handlerThread));
  }
  // Definisanje Handler objekta koji obrađuje poruke koje se šalju Messenger objektu
  private class IncomingHandler extends Handler{
    IncomingHandler(HandlerThread handlerThread){
      super(handlerThread.getLooper());
    }
    public int brojPokreta=0;
    @Override
    public void handleMessage(Message msg) {
      switch (msg.what){
        case Constants.MESSENGER_SERVICE_REGISTRUJ_KLIJENTA:
          mClients.add(msg.replyTo);
          break;
        case Constants.MESSENGER_SERVICE_POKRENI_IZVRSAVANJE:
          int brojDiskova = msg.arg1;
          brojPokreta=0;
        MaksimalanBrojPokreta = (int) (Math.pow(2 * 1.0, brojDiskova * 1.0)-1);
          doTowers(brojDiskova, 'A', 'B', 'C');
          break;
        default:
          super.handleMessage(msg);
      }
    }
    private void doTowers(int topN, char from, char inter, char to) {
      for(Messenger client : mClients){

    try {
          if (topN == 1) {
       System.out.println("Disk 1 from " + from + " to " + to);
                  return;
        } else {
          doTowers(topN - 1, from, to, inter);
              brojPokreta++;
        mClients.get(0).send(Message.obtain
            (null,Constants.MESSENGER_KLIJENT_PORUKA_POKRET, brojPokreta, 0));
          doTowers(topN - 1, inter, from, to);
        }
        } catch (RemoteException e) {
          mClients.remove(client);
        }
      }}}
  // Metoda koja se pozva kad se klijent poveze na servis
  // Metoda vraća Binder Messenger objekta
 @Override
  public IBinder onBind(Intent intent) {
    return mMessenger.getBinder();
  }
}
```

Slika 5.2.6 Implementacija udaljenog servisa koji komunicira sa klijentom putem *Messenger* objekta

Implementacija klijent-servis komunikacije, koja se odvija korišćenjem *AIDL* interfejsa, prikazana je u kodu na slikama 5.2.9 i 5.2.10. Klijent implementira svoj *AIDL* interfejs, koji je definisan na slici 5.2.7, u sebi sadrži samo jednu metodu koja vraća pokret klijentu. Interfejs koji implementira servis je prikazan na slici 5.2.8. Ovaj interfejs ima definisane dve metode, jedna za registraciju klijenata i druga za pokretanje izračunavanja pokreta za N diskova problema „Kule Hanoja“. Uspostavljanje komunikacije između klijenta i servisa se radi isto kao kod implementacije *Messenger* mehanizama. Jedna razlika je u tome što se sada ne razmenjuju *Messenger* objekti nego objekti koji implementiraju *AIDL* inrefejse. Detaljnije o implementaciji *AIDL* mehanizma za IPC možete videti u poglavlju 3.3.

```
interface IAidlClient {
        void vratiPokret(int brojPokreta);
}
```

Slika 5.2.7 Definisanje *AIDL* interfejsa ka klijentu

```
interface IAidlService {
        void registrujKlijenta(IAidlClient klijent);
        void pokreniIzvrsavanje(int brojDiskova);
}
```

Slika 5.2.8 Definisanje *AIDL* interfejsa ka servisu

```
import com.example.ailic.aidlservice.IAidlService.Stub;

public class AIDLService extends Service {
  int mBrojPokreta = 0;
  int mMaksimalanBrojPokreta = 0;

  public AIDLService() {
  }
  // Definisanje klijent objekta u servisu
  IAidlClient mKlijent;

  @Override
  public IBinder onBind(Intent intent) {
   // Kreiranje Binder servisa iz Stub klase koja se automatski generiše
   // prilikom  definisanja interfejsa servisa
    return new IAidlService.Stub() {

     @Override
     public void registrujKlijenta(IAidlClient klijent) throws RemoteException
          {
        mKlijent = klijent;
      }
     @Override
     public void pokreniIzvrsavanje(final int brojDiskova) throws RemoteException
         {
```

```
     mBrojPokreta = 0;
     mMaksimalanBrojPokreta = (int) (Math.pow(2 * 1.0, brojDiskova * 1.0) - 1);
        new Thread(new Runnable() {
          @Override
          public void run() {
            doTowers(brojDiskova, 'A', 'B', 'C');
          }
        }).start();
      }
    };
  }
  @Override
  public boolean onUnbind(Intent intent) {
    mKlijent = null;
    return super.onUnbind(intent);
  }

  private void doTowers(int topN, char from, char inter, char to) {
      try {
            if (topN == 1) {
                  return;
                } else {
                  doTowers(topN - 1, from, to, inter);
                  brojPokreta++;
                  mKlijent.vratiPokret(mBrojPokreta);
                  doTowers(topN - 1, inter, from, to);
                }
        } catch (RemoteException e) {
         mKlijent=null;
        }
      }
  }
}}
```

Slika 5.2.9 Implementacija udaljenog servisa koji komunicira sa klijentom putem *AIDL* interfejsa

```
public class AIDLActivity extends ActionBarActivity {


  private Button mRunButton;
  private TextView mLogTextView;
  private EditText mBrojDiskova;
  private Context mContext;
  private int mMaksimalanBrojPokreta = 0;
  private int mDestetinaPokreta = 1;


  private IAidlService mService = null;
// Objekat kojim se inicija konekcija sa udaljenim Messenger servisom
  private ServiceConnection mConnection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
      mService = IAidlService.Stub.asInterface(service);
      try {
        mService.registrujKlijenta(mCallback);
      } catch (RemoteException e) {
        e.printStackTrace();
      }
      mRunButton.setEnabled(true);
    }
```

```
    @Override
    public void onServiceDisconnected(ComponentName name) {
      mService = null;
      mRunButton.setEnabled(false);
    }
  };
// Definisanje klijenta na osnovu Stub klase koja se automatski generise
// na osnovu AIDL interfejsa klijeta

  private IAidlClient mCallback = new IAidlClient.Stub() {
    long pocentoVreme;
    DateFormat formatter = new SimpleDateFormat("mm:ss:SSS");

    @Override
    public void vratiPokret(final int brojPokreta) throws RemoteException {

      if(brojPokreta==1){
        java.util.Date date = new java.util.Date();
        pocentoVreme=date.getTime();
        runOnUiThread(new Runnable() {
          @Override
          public void run() {
            mLogTextView.setText("");
          }
        });
      }

      if (mMaksimalanBrojPokreta >= brojPokreta) {
        if (brojPokreta % mDestetinaPokreta == 0) {
          final java.util.Date date = new java.util.Date();
          final String message ="Proteklo vreme za " + brojPokreta + "
                   pokreta je "+ formatter.format(date.getTime() - pocentoVreme)
                 + "\n"+mLogTextView.getText();
          runOnUiThread(new Runnable() {
            @Override
            public void run() {
              mLogTextView.setText(message + "\n");
            }
          });
          }
      } else {
        final java.util.Date date = new java.util.Date();
        runOnUiThread(new Runnable() {
          @Override
          public void run() {
            mLogTextView.setText("Vreme potrebno da se " +
                         mBrojDiskova.getText()
            + " diskova, prebaci \nsa stapa A na C je "
                      + formatter.format(date.getTime() - pocentoVreme)
            + " i tom prilikom se izvrsi " + (int) mMaksimalanBrojPokreta
                      + " pokreta\n" + mLogTextView.getText());
            mRunButton.setEnabled(true);
          }
        });
      }
    }
  };

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_aidl_service);
```

```
    mRunButton = (Button) findViewById(R.id.run_button);
    mRunButton.setEnabled(false);
    mLogTextView = (TextView) findViewById(R.id.message_log_textView);
    mBrojDiskova = (EditText) findViewById(R.id.broj_diskova);
    mContext = this;
    getSupportActionBar().setTitle(R.string.title_activity_aidl);
    mRunButton.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {

        if (mBrojDiskova.getText().toString().trim().equals("")) {
          new AlertDialog.Builder(mContext).setMessage("Morate unteti broj diskova pre
nego sto pokrenete izvrsavanje").show();
        } else {
          int n = Integer.decode(mBrojDiskova.getText().toString()).intValue();
          try {
            mService.pokreniIzvrsavanje(n);
          } catch (RemoteException e) {
            e.printStackTrace();
          }
          mMaksimalanBrojPokreta = (int) Math.pow(2 * 1.0, n * 1.0) - 1;
          mDestetinaPokreta = mMaksimalanBrojPokreta / 10;
          mRunButton.setEnabled(false);
        }
      }
    });
    getApplicationContext().bindService(new Intent("com.aidl.service.ACTION"),
mConnection, Context.BIND_AUTO_CREATE);
  }
  @Override
  protected void onDestroy() {
    super.onDestroy();
    if (mService != null) {
      getApplicationContext().unbindService(mConnection);
    }
  }
```

Slika 5.2.10 Implementacija *AIDL* klijenta

5.3 Uređaji korišćeni za testiranje

Brzina komunikacije je testirana na tri urđaja koji imaju različit hardver i verzije Android operativnog sistema na sebi. To su sledeći modeli telefona: HTC Desire S, Sony Xperia J i Samsung Galaxy S4.

HTC Desire S uređaj ima sledeće specifikacije:

Procesor	**Procesor** : Scorpion processor, Adreno 205 GPU, Qualcomm MSM8255 Snapdragon
	Takt : 1000 MHz
Memorija	**ROM** : 1126 MB
	RAM : 768 MB
Operativni sistem	**Vrsta** : Android
	Verzija : v2.3
	Podržani formati : aac, wav, mp3, wma, DivX, XviD, H.263, H.264, wmv,

Sony Xperia J uređaj ima sledeće specifikacije:

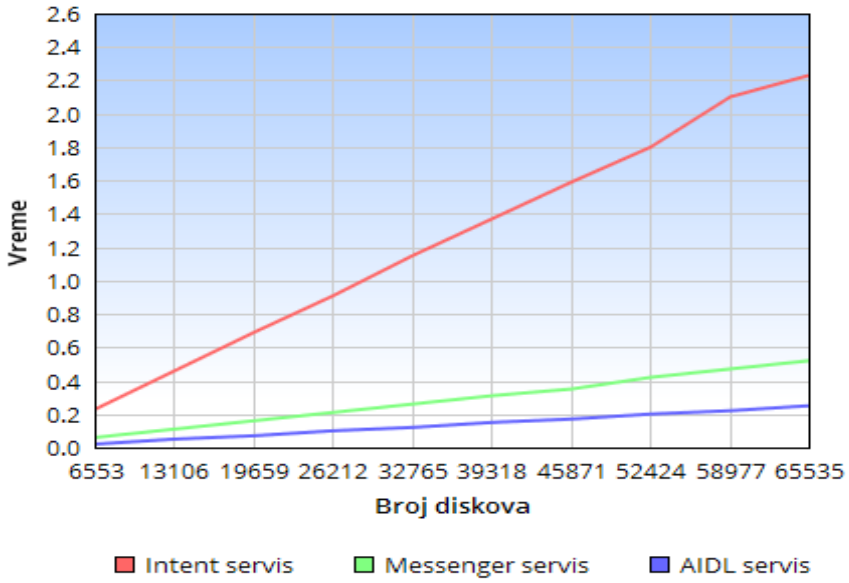
Procesor	**Procesor** : Cortex-A5
	Takt : 1000 MHz
Memorija	**RAM** : 512 MB
Operativni sistem	**Vrsta** : Android
	Verzija : v4.0
	Podržani formati : aac, wav, mp3, wma, DivX, XviD, H.263, H.264, wmv,

Samsung Galaxy S4 ima sledeće specifikacije:

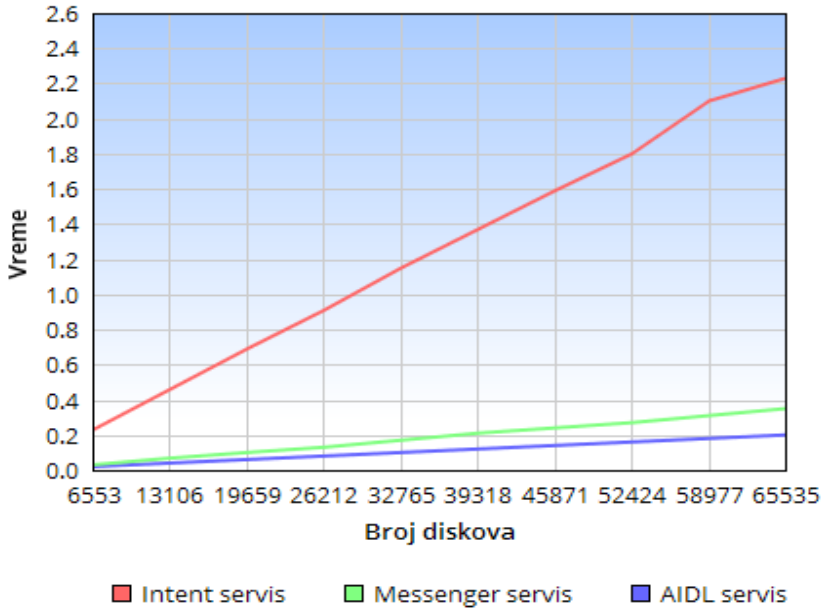
Procesor	**Procesor** : Quad-core Cortex-A15 & quad-core Cortex-A7
	Takt : 1600 i 1200 MHz
Memorija	**RAM** : 2048 MB
Operativni sistem	**Vrsta** : Android
	Verzija : v4.4.2
	Podržani formati : aac, wav, mp3, mp4, wma, DivX, XviD, H.263, H.264, wmv,

5.4 Rezultati testiranje brzine komunikacije

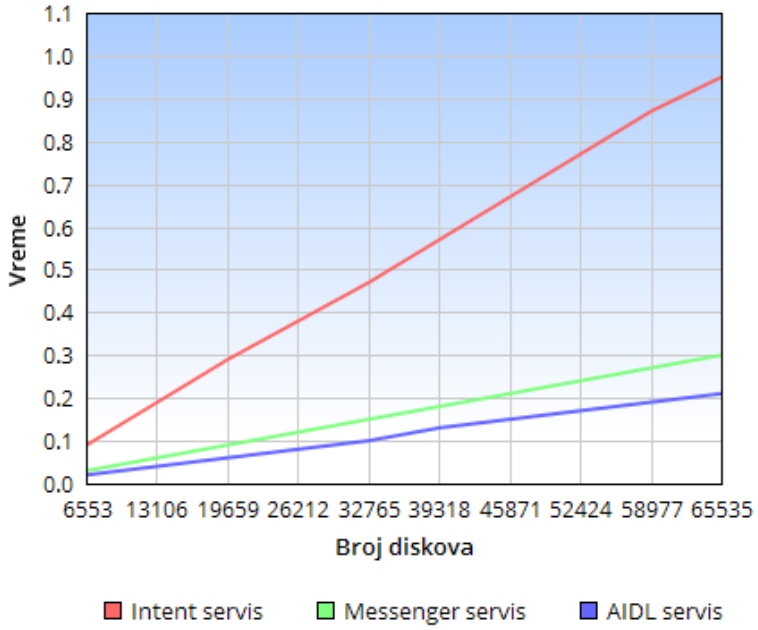
Prilikom tesitranja brzine komunikacije između servisa i klijenta biće mereno vreme za koje oni razmene određeni broj poruka, tačnije onoliko poruka koliko je potrebno za premeštanje N diskova sa prvog na poslednji štap. Uređaji koji su korišćeni prilikom testiranja su opisani u poglavlju 5.4. Na slikama 5.3.1, 5.3.2 i 5.3.3 su prikazane razlike u brzinama u odnosu na to koji je mehanizam za implementaciju inter-procesne komunikacije izabran.



Slika 5.3.1 Uporedni prikaz brzine slanja poruka na uređaju HTC Desire S



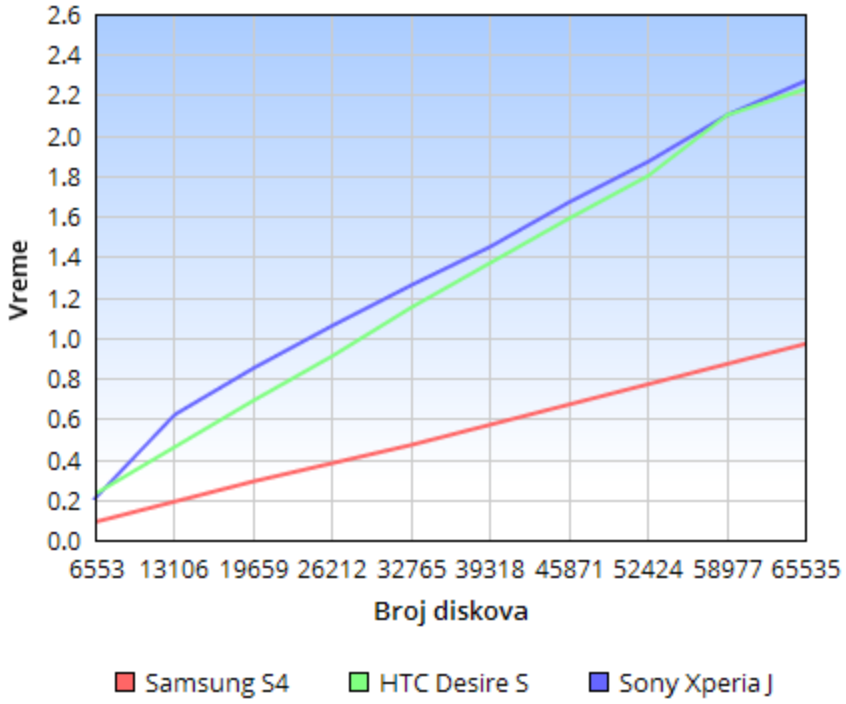
Slika 5.3.2 Uporedni prikaz brzine slanja poruka na uređaju Sony Xperia J



Slika 5.3.3 Uporedni prikaz brzine slanja poruka na uređaju Samsung S4

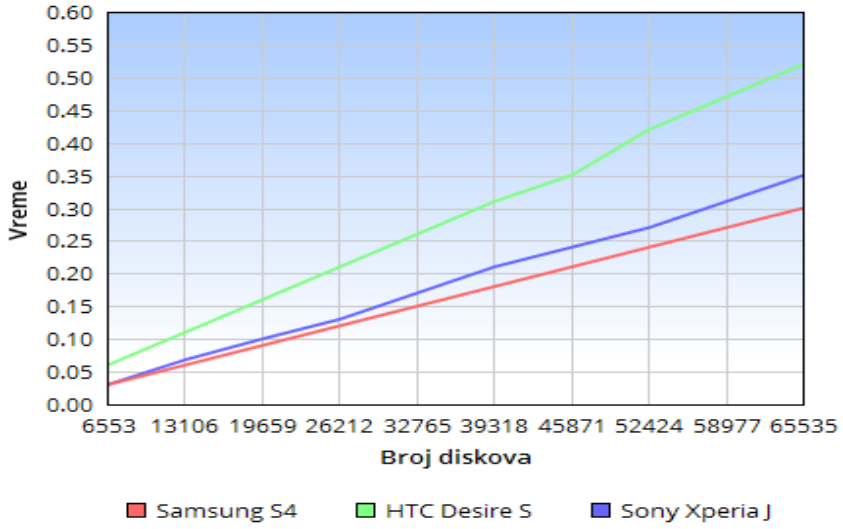
Kada analiziramo prethodne tri slike, možemo uočiti veliku sličnost u izgledu grafika. To nam govori da nezavisno od specifikacije uređaja, u zavisnosti od toga koji mehanizam IPC upotrebimo, možemo dobiti dobre ili loše performanse. Test je rađen za 16 diskova tačnije 65535 pokreta (poruka). Takođe, iz prethodnih slika, nezavisno od uređaja, možemo videti da *AIDL* i *Messenger* implementacija daju daleko bolje rezultate u odnosu na *Intent* implementaciju.

Na slikama 5.3.4, 5.3.5 i 5.3.6 su prikazani razlike u brzini određenog mehanizma inter-procesne komunikacije u zavisnosti od uređaja na kom je iplementiran.



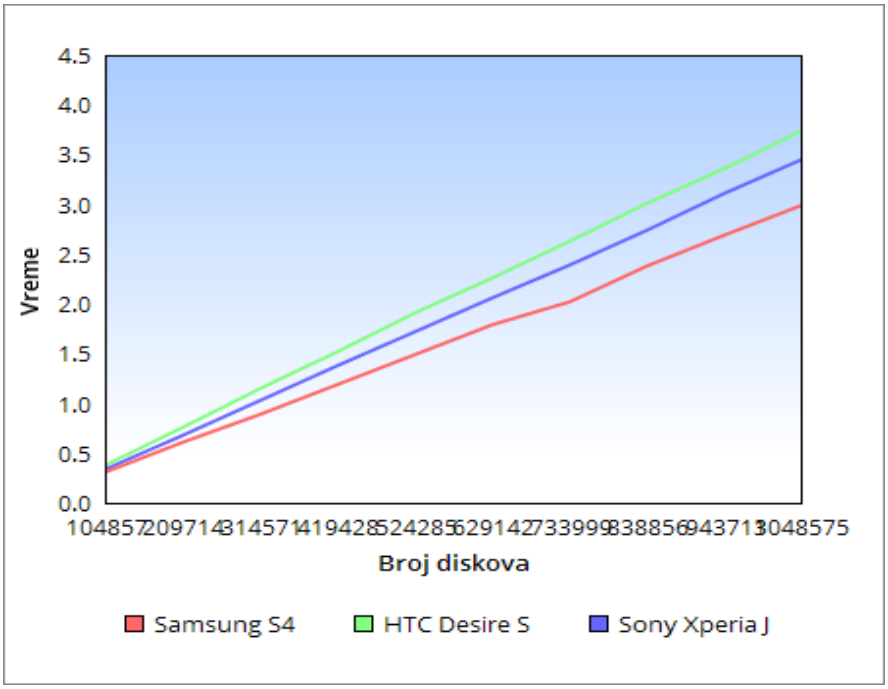
Slika 5.3.4 Brzina *Intent* mehanizama IPC na tri različita uređaja

Ukoliko analiziramo grafik na slici 5.3.4, mozemo videti da uređaj Samsung S4 daje najbolje performanse ali ne treba zaboraviti činjenicu da on ima i najbolji hardver. HTC Desire S i Sony Xperia J su približno istih hardverskih karakteristika pa zato i slični rezultati ne čude mnogo.



Slika 5.3.5 Brzina *Messenger* mehanizama IPC na tri različita uređaja

Ukoliko analiziramo grafik sa slike 5.3.5, možemo uočiti da je razlika u performansama između HTC Desire S i Sony Xperia J dosta veća u odnosu na *Intent* implementaciju IPC prikazanoj na slici 5.3.4. Razlog tome je razlika u verzijama Android operativnog sistema koji se nalaze na njima. Naime Sony Xperia J ima verziju 4.0 Android OS koja je dosta poboljšala rad sa više niti. *Messenger* objekat je implementiran na sopstvenoj niti, dok se u *Intent* mehanizmu svi *Intent* objekti šalju sa glavne niti.



Slika 5.3.6 Brzina *AIDL* mehanizama IPC na tri različita uređaja

Prilikom testiranja *AIDL* mehanizma bilo je potrebno podići broj poruka koje se šalju. Analizom slike 5.3.6 možemo videti da je *AIDL* implementacija veoma brza na svim uređajima. Razmena nešto više od milion poruka ne traje duže od 3,5 sekundi. Vidimo da Samsung S4 daje najbolje rezultate, što je i očekivano, dok za njim slede Sony Xperia J i HTC Desire S uređaju. Samsung S4 uređaj poseduje dva procesora sa po četri jezgra, tako da bi on postigao mnogo bolje rezultate u odnosu na ostala dva uređaja ukoliko bi bilo više klijenata. Podsećanja radi u *AIDL* mehanizmu dva različita klijenta mogu dobiti sopstvenu instancu servisa koji se vrte na različitim nitima operativnog sistema. Kod ostalih mehanizama servis će se uvek pokretati na jednoj niti.

6. Zaključak

Kroz ovaj rad je detaljno prikazani mehanizami inter-procesne komunikacije na Android operativnom sistemu koji je zasnovan na *Binder* okviru. Uz mnoštvo primera su prikazane implementacije tri mehanizma, i to su: *Intent* mehanizam, *Messenger* mehanizam i mehanizam zasnovan na jeziku za definisanje Android interfejsa. Pored samog upoznavanja sa mehanizmima inter-procesne komunikacije, u radu su takođe izložene prednosti i mane određenih implementacija u onosu na ostale. Ovo je veoma važno zbog toga što mnogi čak i iskusni Android programeri isključivo koriste *Intent* mehanizam za inter-procesnu komunikaciju, zanemaruju značaj ostalih mehanizama. Razlog tome je njegova jednostavnost prilikom korišćenja. Međutim, kada se dođe do trenutka da je potrebna neka optimizacija u inter-procesnoj komunikaciji *Messenger i AIDL* mehanizmi mogu dati višestruko bolje rezultate. Ova dva mehanizma još više dobijaju na značaju sa obzirom na to koliko brzo napreduju hardverske karekteristike mobilnih uređaja. Naime premijum modeli skoro svih proizvođača sadrže dva procesora sa po dva ili više jezgra. Sa povećanjem broja jezgra *AIDL* mehanizamom se veoma ubrzava inter-procesna komunikacija jer on podržava rad sa više niti u isto vreme.

U poslednjem poglavlju rada je opisana aplikacija „Kule Hanoja“ gde se mogu videti implementacije svih ovih mehanizama. U radu je takođe prikazana statistika o brzini inter-procesne komunikacije u razmeni 2^n-1 poruka u zavisnosti od hradverskih karakteritika uređaja i implementiranog mehanizma inter-procesne komunikacije.

7. Reference

[1]http://www.emarketer.com/Article/2-Billion-Consumers-Worldwide-Smartphones-by-2016/ac
[2] http://www.idc.com/prodserv/smartphone-os-market-share.jsp
[3] http://www.openhandsetalliance.com/android_overview.html
[4] http://www.gnu.org/software/libc/
[5] https://en.wikipedia.org/wiki/X_Window_System
[6] http://en.wikipedia.org/wiki/Loose_coupling
[7] https://developers.google.com/android/c2dm/?hl=en
[8] https://en.wikipedia.org/wiki/Near_field_communication
[9] https://en.wikipedia.org/wiki/Android_Beam
[10] https://support.google.com/quickoffice/answer/2986862?hl=en
[11]https://cloud.google.com/storage/?utm_source=google&utm_medium=cpc&utm_campaign=2015-q1-cloud-emea-storage-skws-freetrial-en&gclid=CjwKEAjws7OwBRCn2Ome5tPP8gESJAAfopWs_n-_pQG_Fsx9iJiZLguTFG5PWnyGDrbvm0Z2SxsepxoC-bbw_wcB
[12] https://en.wikipedia.org/wiki/Android_version_history
[13] https://developer.android.com/about/dashboards/index.html
[14] http://www.tldp.org/LDP/lpg/node21.html
[15] Professional Android 2 Application Development, by Reto Meier
Publisher: Wiley Publishing,Inc.
[16]http://www.statista.com/statistics/272307/market-share-forecast-for-smartphone-operating-systems/