Introducere în sistemul de operare Android

LAURA RUSE
VLAD TRAISTĂ-POPESCU

Universitatea Politehnica din București

Cuprins

1 Introducere **1**

1.1 Arhitectura Android-ului . . . 1
1.1.1 Popularitate . . . 1
1.1.2 PHA . . . 1
1.1.3 Arhitectura sistemului de operare . . . 2
1.1.4 Kernel-ul Linux . . . 3
1.1.5 Dalvik . . . 4
1.1.6 Crearea unui APK . . . 4
1.1.7 ART . . . 4
1.1.8 Biblioteci native . . . 5
1.1.9 Application Framework . . . 6
1.2 Dezvoltarea unei aplicații Android . . . 6
1.2.1 Componentele unei aplicații . . . 6
1.2.2 Activitatea . . . 6
1.2.3 Serviciul . . . 7
1.2.4 Broadcast Receiver . . . 7
1.2.5 Content Provider . . . 7
1.2.6 Intent . . . 7
1.2.7 Binder . . . 8
1.3 Mecanisme de securitate . . . 8
1.3.1 Sandboxing . . . 8
1.3.2 Permisiunile de acces asupra fișierelor . . . 9
1.3.3 UID partajat . . . 9
1.3.4 Permisiunile aplicației . . . 9
1.3.5 Semnarea aplicațiilor . . . 10
1.4 Bibliografie . . . 10

2 Android SDK **11**

2.1 Aplicații . . . 11
2.1.1 Fișier Manifest . . . 11
2.1.2 Permisiuni . . . 11
2.1.3 Resurse . . . 13
2.1.4 Layout . . . 13
2.1.5 Drawables . . . 14
2.2 Activități . . . 14
2.2.1 Declararea Activității în Manifest . . . 15
2.2.2 Ciclul de viață al unei Activități . . . 15
2.2.3 Salvarea și restaurarea stării unei Activități . . . 17
2.2.4 Fragmente . . . 17
2.2.5 Ciclul de viață al unui Fragment . . . 19

2.2.6 Interfața cu utilizatorul . . . 19
2.3 Servicii . . . 19
2.3.1 Declararea Serviciilor în Manifest . . . 21
2.3.2 Tipuri de Servicii . . . 21
2.3.3 Ciclul de viață al unui Serviciu . . . 22
2.4 Intent-uri . . . 24
2.4.1 Cazuri de utilizare a Intent-urilor . . . 24
2.4.2 Tipuri de Intent-uri . . . 24
2.4.3 Intent-uri implicite . . . 25
2.5 Broadcast Receivers . . . 26
2.5.1 Tipuri de mesaje Broadcast . . . 26
2.5.2 Declararea unui Broadcast Receiver în Manifest . . . 27
2.5.3 Implementarea unui Broadcast Receiver . . . 27
2.6 Content Providers . . . 27
2.6.1 Content URIs . . . 28
2.6.2 Operații asupra unui Content Provider . . . 28
2.7 Utilitare . . . 29
2.7.1 Android Studio . . . 29
2.7.2 SDK Manager . . . 29
2.7.3 AVD Manager . . . 29
2.7.4 Emulatorul . . . 30
2.7.5 ADB . . . 31
2.8 Bibliografie . . . 31

3 Internele sistemului de operare Android 32
3.1 Achitectura Android-ului . . . 32
3.2 Kernel-ul Linux . . . 33
3.2.1 Wakelocks . . . 33
3.2.2 Low Memory Killer . . . 34
3.2.3 Anonymous Shared Memory . . . 35
3.2.4 Alarm . . . 35
3.2.5 Paranoid Networking . . . 36
3.3 Binder . . . 37
3.3.1 Istoric . . . 37
3.3.2 RPC . . . 37
3.3.3 Android Binder . . . 38
3.4 Framework-ul Android . . . 39
3.4.1 Serviciile de sistem . . . 39
3.4.2 Dalvik . . . 39
3.4.3 ART . . . 40
3.4.4 Zygote . . . 40
3.4.5 Xposed . . . 41
3.4.6 Logd . . . 42
3.5 Manageri . . . 43
3.5.1 Service Manager . . . 43
3.5.2 Activity Manager . . . 43
3.5.3 Package Manager . . . 44
3.5.4 Power Manager . . . 44
3.6 Bibliografie . . . 45

4 Conectivitate **46**
4.1 Multithreading . . . 46
4.1.1 Thread-ul principal de UI al aplicației . . . 46
4.1.2 AsyncTask . . . 47
4.2 Accesarea conținutului online . . . 48
4.2.1 Conectarea la rețea . . . 48
4.2.2 Accesarea conținutului online . . . 48
4.2.3 Conexiune HTTP . . . 49
4.3 Obținerea locației . . . 49
4.3.1 Permisiuni de obținere a locației . . . 50
4.3.2 Primirea actualizărilor de locație . . . 50
4.4 WiFi Manager . . . 51
4.4.1 Activarea și dezactivarea WiFi . . . 52
4.4.2 Scanarea rețelelor WiFi . . . 52
4.4.3 Informații despre conexiunea curentă . . . 52
4.5 Bluetooth . . . 53
4.5.1 Permisiuni Bluetooth . . . 53
4.5.2 Bluetooth API . . . 53
4.5.3 Bluetooth Low Energy API . . . 55
4.6 Near Field Communication . . . 55
4.6.1 Moduri de operare NFC . . . 56
4.6.2 Folosirea NFC într-o aplicație Android . . . 56
4.6.3 Descoperirea dispozitivelor NFC . . . 56
4.6.4 Trimiterea mesajelor NFC . . . 56
4.7 API-urile Google . . . 57
4.7.1 Google Play Services . . . 57
4.7.2 Google Maps . . . 58
4.8 Bibliografie . . . 59

5 Concluzii **61**

Listă de figuri

1.1 Evoluția cotei de piață pentru sistemele de operare mobile 2
1.2 Arhitectura Android-ului 3
1.3 Generarea unui fișier apk 5
1.4 Apelarea unui serviciu de sistem 6

2.1 Layout-uri diferite în funcție de dimensiunea ecranului 13
2.2 Ciclul de viață a unei activități 16
2.3 Procesul de salvare și restaurare a stării unei activități 18
2.4 Fragmente afișate pe dispozitive diferite 18
2.5 Ciclul de viață al unui fragment 20
2.6 Ciclul de viață al unui serviciu 23
2.7 Intent-uri implicite 25
2.8 Android Studio 29
2.9 SDK Manager 30
2.10 AVD Manager 30

3.1 Android Architecture 32
3.2 Generic RPC 38

4.1 Google Play Services 58

Listă de tabele

2.1 Asociere între versiunea de Android și nivelul de API 12

Capitolul 1

Introducere

1.1 Arhitectura Android-ului

1.1.1 Popularitate

Android este un sistem de operare open-source pentru dispozitive mobile. În 2019 a avut mai mult de 2.5 miliarde de utilizatori activi lunar și în ianuarie 2021 cea mai mare cotă de pe piața telefoanelor mobile dintre toate sistemele de operare, mai mult de 71%, în timp ce iOS a avut 27%.

Considerând toate sistemele de operare, Android are o cotă de piață de 41%, în timp ce Windows are 31%, iOS 16%, OSX 7% și Linux 0.8%.

În Figura 1.1 putem vedea statisticile reprezentate de Statista[1] cu evoluția cotei de piață de-a lungul anilor. Putem observa că în 2012 erau mai multe sisteme de operare pentru dispozitive mobile cu o cotă de piață considerabilă: Android, iOS, Symbian, Samsung, Blackberry. Până în 2021, Android și iOS au înlocuit celelalte sisteme de operare și au împreună 99% din cota de piață a sistemelor de operare pentru dispozitive mobile.

Google a făcut posibilă dezvoltarea ușoară a aplicațiilor pentru Android și publicarea acestora pe market-ul oficial de aplicații Google Play Store.

1.1.2 PHA

Potentially Harmful Applications (PHAs) sunt aplicații care ar putea fi periculoase pentru utilizatori, datele acestora sau dispozitive. Uneori aceste aplicații pot conține **malware**.

Dar de ce folosim cuvântul potential? Deoarece comportamentul malițios poate fi activat doar în anumite condiții sau contexte: de exemplu, pe o anumită versiune de Android, pe un anumit firmware, sau pe o anumită configurație de sistem.

Din 2017, Android oferă Google Play Protect, care este folosit pentru a detecta și elimina PHA-urile de pe dispozitivele Android.

[1] https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

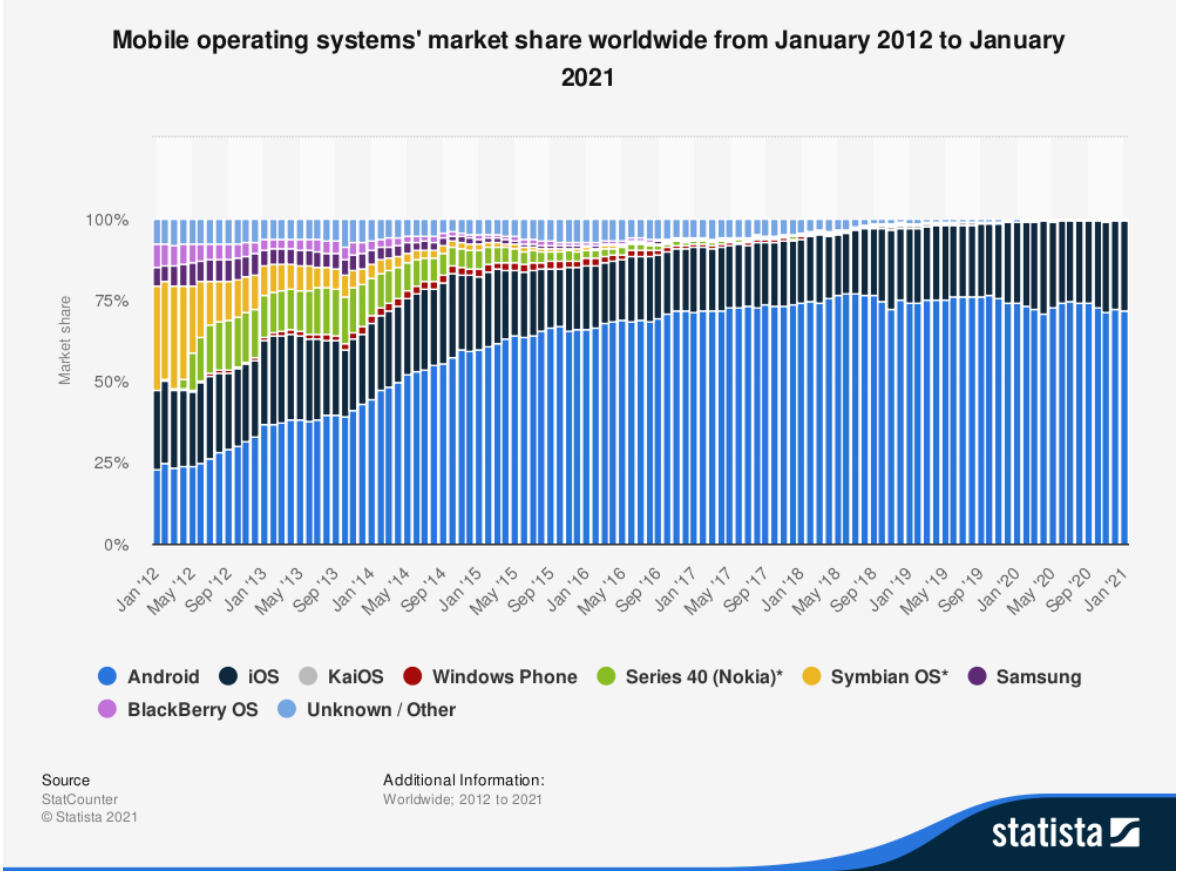

Figura 1.1: Evoluția cotei de piață pentru sistemele de operare mobile

Cele mai multe PHA-uri detectate de Play Protect sunt spyware. De exemplu, din Octombrie până în Decembrie 2020, 84.3% din toate PHA-urile au fost spyware. Se pot vedea mai multe statistici pe această pagină[2].

1.1.3 Arhitectura sistemului de operare

Arhitectura Android-ului este reprezentată în Figura 1.2. După cum se poate observa, Android-ul rulează deasupra unui kernel Linux.

Datorită proiectelor Android Mainlining Project și Android Upstreaming Project, Android-ul poate rula deasupra unui kernel vanilla recent, cu câteva modificări, numite Androidisme, de exemplu Low Memory Killer, Wakelocks, Anonymous Shared Memory, Alarms, Paranoid Networking și Binder.

Majoritatea Androidism-elor au fost deja integrate în mainline. Avantajele principale ale folosirii unui kernel Linux sunt faptul că se pot folosi mecanismele de securitate deja implementate în Linux și faptul că producătorii pot dezvolta ușor drivere de dispozitive.

Deasupra kernel-ului Linux, avem userspace-ul nativ, care include procesul `init`, câțiva daemoni nativi și sute de biblioteci native. Procesul `init` și daemonii nativi sunt diferiți față de cei din Linux-ul standard.

O mare parte din Android este implementată în Java și rulează într-o mașină virtuală numită Dalvik. Aceasta nu poate executa bytecode Java (`.class`), doar executabile

[2]https://transparencyreport.google.com/android-security/overview?hl=en

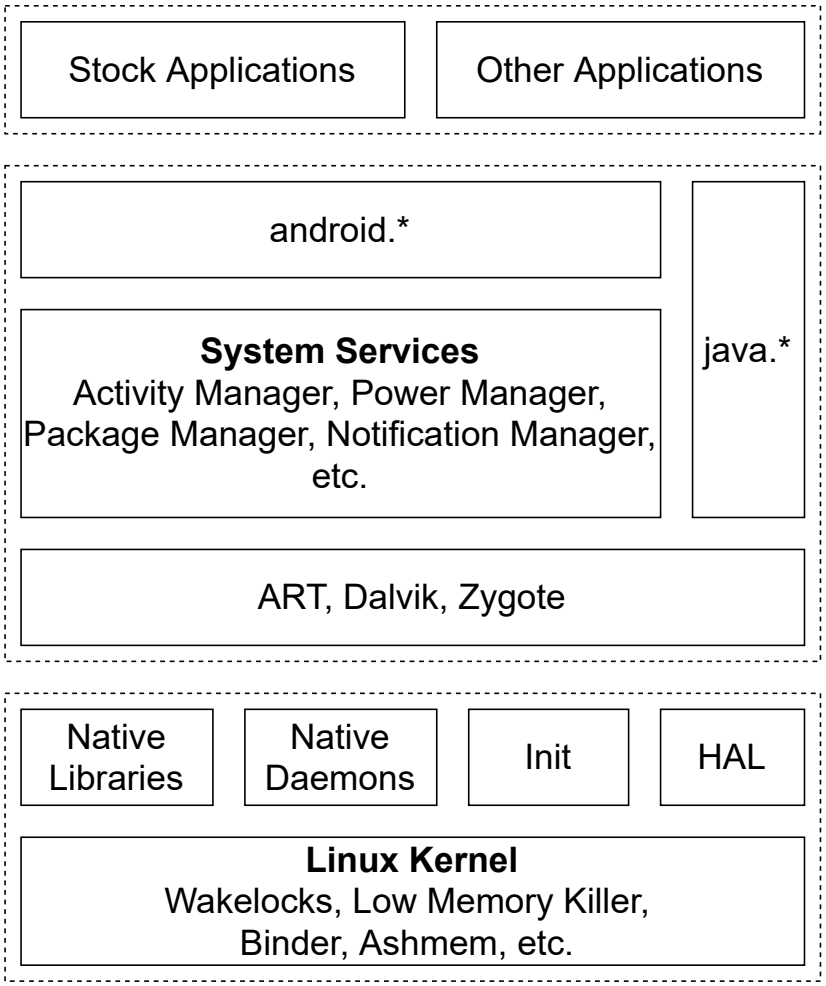

Figura 1.2: Arhitectura Android-ului

Dalvik (`.dex`). Mai recent a apărut runtime-ul ART, care este mai performant.

Bibliotecile Java runtime sunt definite în pachetele `java.*` și `javax.*`, care sunt derivate din proiectul Apache Harmony, dar anumite componente au fost înlocuite, altele extinse și îmbunătățite. Codul nativ poate fi apelat din codul Java și viceversa prin Java Native Interface (JNI).

Serviciile de sistem implementează o parte din caracteristicile fundamentale ale Android-ului: display, touch screen, telefonie, conectivitatea la rețea. O parte din aceste servicii sunt implementate în cod nativ și restul in Java. Fiecare serviciu oferă o interfață care poate fi apelată.

Bibliotecile Android Framework includ componente de bază pentru implementarea aplicațiilor Android: clase de bază pentru activități, servicii, content providers, GUI widgets, access la fișiere și la baze de date, etc. De asemenea include clase pentru interacțiunea cu hardware-ul și cu serviciile de nivel înalt.

1.1.4 Kernel-ul Linux

Kernel-ul Linux folosit de Android este unul modificat pentru a permite lucrul cu dispozitive mobile.

Android Mainlining Project si Android Upstreaming s-au ocupat de includerea acestor

modificări în mainline. Multe Androidisme au fost deja integrate, cum ar fi WakeLocks inclus in versiunea 3.5 de kernel, Low-Memory Killer din 3.10, Binder din 3.19, Alarm și Logger din 3.20.

O altă diferență față de kernelul Linux standard: kernel-ul pentru Android include doar mecanismul de suspend to memory, nu și cel de suspend to hard-disk care este folosit pe PC-uri.

1.1.5 Dalvik

Android Runtime include mașina virtuală Dalvik, care este folosită pentru a rula bytecode-ul atât al aplicațiilor, cât și al serviciilor de sistem care sunt implementate în Java.

Bibliotecile Java sunt cele din Apache Harmony Project (cu anumite modificări), nu cele de la Oracle/Sun, pentru a evita problemele de copyright și distribuție.

Mașina virtuală Dalvik a fost proiectată special pentru dispozitivele embedded care, de obicei, au puțină memorie, procesor lent, nu fac swap și sunt alimentate de la baterie.

Dalvik rulează fișiere `.dex` (vine de la Dalvik Executable). Aceste fișiere sunt cu 50% mai mici decât fișierul `.jar` conținând fișierele `.class` asociate.

Din Android 2.2, Dalvik include compilare Just-In-Time (JIT), ceea ce înseamnă că segmentele scurte din bytecode care sunt executate cel mai frecvent vor fi transformate în cod mașină nativ. Acest lucru aduce îmbunătățiri de performanță. Restul bytecode-ului va fi interpretat de Dalvik.

1.1.6 Crearea unui APK

În Figura 1.3 se poate observa cum se obține fișierul `.dex` din fișierele `.java`. Întâi se compilează fișierele `.java` în fișiere `.class` folosind compilatorul, apoi se folosește utilitarul `dx` pentru a agrega toate aceste fișiere `.class` într-un singur fișier `.dex`. Putem vedea colecția de constante, clasele și datele agregate în fișierul `.dex`.

Apoi se adaugă celelalte resurse ale aplicației și se crează fișierul `.apk` prin folosirea utilitarului `apkbuilder`. În final se semnează apk-ul folosind cheia de debug sau cheia de release.

1.1.7 ART

ART este un runtime mai avansat disponibil începând cu Android 4.4. Include compilarea de tip Ahead-Of-Time (AOT). Asta înseamnă că în momentul instalării aplicației, se va translata bytecode-ul dex în cod mașină și se va stoca executabilul pentru rulările viitoare.

Acest lucru are loc o singură dată, de aceea este mult mai eficient și, astfel, se reduce consumul de putere asociat compilării. AOT înlocuiește compilarea JIT și interpretarea în mașina virtuală Dalvik.

Dezavantajul este ca se va ocupa mai mult spațiu de stocare cu acele executabile. Alt dezavantaj este faptul că instalarea poate dura mai mult ca înainte.

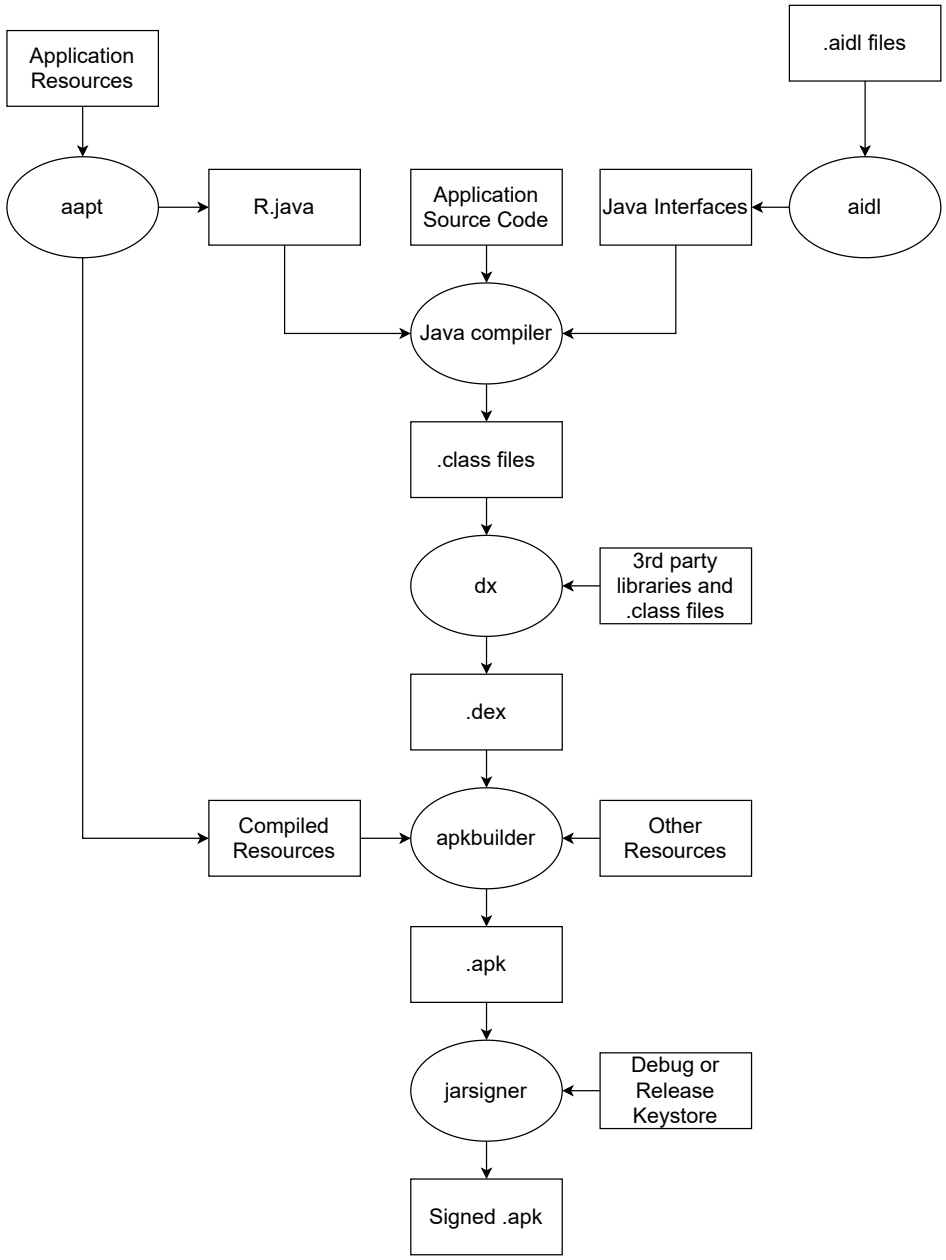

Figura 1.3: Generarea unui fișier apk

În plus, ART oferă mecanisme îmbunătățite de alocare de memorie, garbage collection, debugging și profiling de nivel înalt al aplicațiilor.

1.1.8 Biblioteci native

Tot deasupra kernel-ului avem și bibliotecile native. Cea mai importantă este bioniC, biblioteca de C a Android-ului, un înlocuitor pentru glibc. Are licență BSD și dimensiune mult mai mică decât glibc.

SQLite este o bibliotecă pentru gestiunea bazelor de date SQL. În prezent nu se mai folosește atât de mult, deoarece există multe soluții de stocare a datelor în cloud.

OpenGL ES este versiunea de OpenGL pentru dispozitive embedded. OpenGL este o interfață software standard pentru hardware-ul care face procesări grafice 3D.

WebKit este o bibliotecă pentru afișarea paginilor web, folosită de multe sisteme de operare pentru dispozitive mobile (și nu numai): Android, Apple iOS, Blackberry, Tizen.

SSL include implementarea protocoalelor de securitate folosite pentru a securiza comunicația peste Internet.

1.1.9 Application Framework

Application Framework include servicii și manageri folosiți de către aplicații, de exemplu: Telephony Manager pentru apeluri telefonice, Location Manager pentru obținerea locației, Activity Manager pentru gestiunea activităților în aplicații, Package Manager pentru gestiunea pachetelor de aplicații, Notification Manager pentru generarea și gestiunea notificărilor.

Tot la acest nivel sunt implementați și Content Providers de sistem (impliciți): contacte, calendar, dicționar, media store, setări, etc.

Atunci când o aplicație vrea să acceseze un serviciu de sistem (Figura 1.4), va apela o metodă din API-ul public care, în spate, apelează un stub de RPC care comunică prin Binder cu serviciul de sistem din framework. De asemenea, observăm că aplicația poate să apeleze bibliotecile native prin JNI.

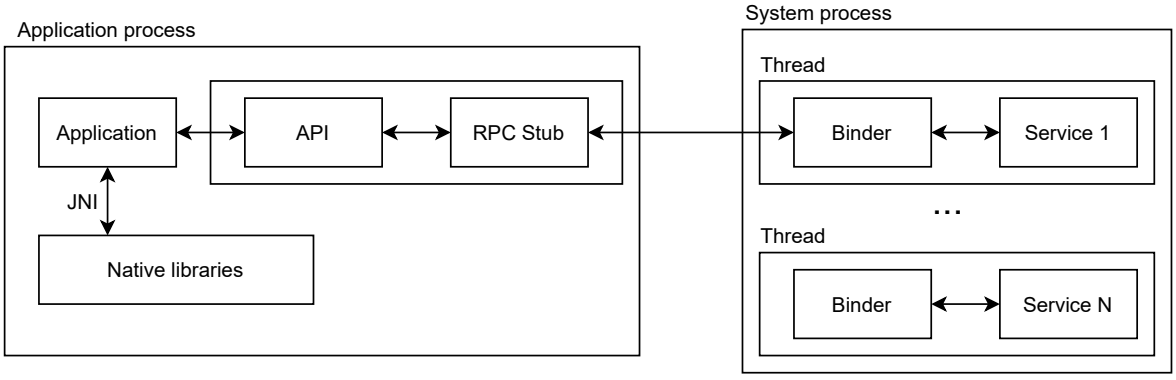

Figura 1.4: Apelarea unui serviciu de sistem

1.2 Dezvoltarea unei aplicații Android

1.2.1 Componentele unei aplicații

O aplicație Android are 4 tipuri de componente - activități (pentru interacțiunea cu utilizatorul), servicii, broadcast receivers și content providers (care rulează în background fără a interacționa direct cu utilizatorul).

1.2.2 Activitatea

Activitatea reprezintă o interfață cu utilizatorul a unei aplicații. Este similară cu o fereastră din interfețele grafice window-based. Însă, o activitate nu poate fi minimizată, maximizată sau redimensionată, ea ocupă aproape tot spațiul vizual de pe ecran.

Utilizatorul poate interacționa cu o singură activitate la un moment dat. Dintr-o activitate se poate porni o altă activitate și așa mai departe. Astfel se creează o stivă de activități.

Similar cu navigarea folosind un browser, se poate apăsa pe butonul back pentru a reveni la activitatea precedentă, dar nu există buton de forward.

Odată scoasă din stivă, o activitate este distrusă și nu se mai poate reveni la ea, dar se poate intra din nou în ea din interfața grafică.

O activitate poate fi pornită folosind un Intent, din Laucher, sau din altă activitate.

1.2.3 Serviciul

Un Serviciu este o componentă a aplicației, care execută operații în background și nu are interfață grafică. El rulează, de obicei, în același proces cu aplicația, dar poate fi configurat să ruleze într-un proces separat.

Un serviciu poate face procesări pentru aplicația curentă sau poate oferi servicii altor aplicații. Comunicarea cu un serviciu se face întotdeauna prin Binder.

1.2.4 Broadcast Receiver

Un Broadcast Receiver este folosit pentru a primi mesaje broadcast care conțin anunțuri sau avertizări precum: descărcarea bateriei, închiderea ecranului (display-ului), un reboot al telefonului.

Unele anunțuri sunt globale (în tot sistemul) și altele sunt locale (doar în aplicația curentă). Mesajele de broadcast sunt livrate folosind Intent-uri.

Din aplicația noastră putem alege ce broadcast-uri să primim și să le tratăm folosind intent filters. Un broadcast receiver rulează (este activ) doar atunci când primește un mesaj de broadcast.

1.2.5 Content Provider

Content Providers sunt folosiți pentru a stoca, gestiona și partaja datele unei aplicații. Dacă o aplicație vrea să pună la dispoziția altor aplicații un set de date va trebui să folosească un content provider.

Content Providers folosesc baze de date relaționale (SQLite) sau fișiere pentru stocarea datelor. Se vor folosi URI-uri pentru identificarea provider-ului și a tabelei.

Content Resolver-ul (obiect client) va folosi acest URI pentru a trimite query-uri către provider (obiect server). Content provider-ul este activ doar atunci când primește o cerere.

1.2.6 Intent

Intent-urile sunt similare semnalelor din Linux. Ele sunt folosite pentru a transmite un mesaj, pentru a determina execuția unei acțiuni.

Ele sunt folosite cu următoarele scopuri principale: pentru pornirea activităților, pentru pornirea serviciilor și binding-ul (legarea) la servicii și pentru trimiterea mesajelor de broadcast către receivers. Intent-urile sunt gestionate și livrate de către sistem.

Un intent are două componente principale: acțiunea ce trebuie executată și datele cu care se operează (de exemplu un URI).

Există două tipuri de Intent-uri: explicite și implicite. Cele explicite sunt trimise direct către un receiver, iar pentru cele implicite sistemul va găsi cea mai potrivită aplicație care poate să facă acțiunea respectivă.

1.2.7 Binder

Binder-ul este un mecanism RPC (Remote Procedure Call) prin care se face invocarea obiectelor remote. Se folosește pentru comunicarea între două componente din același proces sau din procese diferite (Figura 1.4).

Datele vor fi serializate (în obiecte de tip Parcel) și apoi trimise prin Binder de la o entitate la alta. Apelul este sincron, astfel prima componentă se blochează până când primește un răspuns de la a doua componentă.

1.3 Mecanisme de securitate

1.3.1 Sandboxing

Android-ul rulează peste un kernel Linux, prin urmare folosește mecanismele de securitate ale Linux-ului.

De exemplu, Linux izolează procesele și resursele fiecărui utilizator. Un user nu poate accesa fișierele altui utilizator (dacă nu are permisiunile necesare). Fiecare proces va rula cu UID/GID-ul utilizatorului care l-a pornit (în afară de situația în care sunt setați biții SUID sau SGID).

Android folosește acest mecanism de securitate, dar cu alt scop: de a izola aplicațiile una de alta. Sandboxing-ul în Android este bazat pe acest mecanism de securitate.

Sandboxing-ul pe Android este un mecanism de securitate care are la bază folosirea UID-urilor.

Atunci când o aplicație este instalată, primește un UID unic în sistem (care este identificatorul aplicației). Aplicația va rula într-un proces cu acel UID. În plus, va avea un director dedicat, în care doar acel UID are permisiuni de read write execute. Astfel se obține un sandbox la nivel de proces și la nivel de fișier.

Acest sandbox este implementat la nivelul kernel-ului, folosind mecanismele Unix standard în legătură cu procese, UID și permisiuni pe fișiere.

Daemon-ii și serviciile de sistem primesc un UID care este definit într-un fișier header (`android_filesystem_config.h`). Utilizatorul root are UID 0. Prin aplicarea principiului celui mai mic privilegiu, foarte puțini daemoni rulează ca root.

Utilizatorul system are UID 1000. Acesta este un utilizator cu privilegii speciale, dar limitate, nu este echivalent cu root.

Serviciile de sistem încep de la UID 1000 în sus. Aplicațiile normale au UID-uri începând cu 10000 (asignate dinamic, la instalare).

1.3.2 Permisiunile de acces asupra fișierelor

Fiecare aplicație are un director de date dedicat (în care poate stoca baza de date, imagini, sau alte fișiere) și permisiuni de read/write/execute pe acele fișiere, doar pentru acel UID/GID.

Alte aplicații nu pot citi aceste fișiere deoarece nu vor avea permisiunile necesare (nu există drepturi pentru others).

Înainte de Android 4.2 se puteau folosi flag-urile `MODE_WORLD_READABLE` și `MODE_WORLD_WRITEABLE` pentru a oferi acces de citire și scriere pe anumite fișiere altor aplicații. Dar aceste flag-uri sunt deprecated din Android 4.2 și nu este recomandată partajarea directă a fișierelor.

1.3.3 UID partajat

În anumite cazuri speciale, o aplicație poate fi instalată cu același UID ca altă aplicație, reprezentând **Shared UID**. În acest caz, cele două aplicații pot partaja direct fișiere și pot rula chiar și în același proces.

Cel mai frecvent, shared UID este folosit de aplicațiile de sistem, pentru a partaja resursele mai ușor (ex. system UI și lockscreen). În general nu este recomandată folosirea shared UID pentru aplicațiile care nu sunt de sistem deoarece ar putea introduce vulnerabilități.

Pentru a folosi shared UID, aplicațiile trebuie semnate cu aceeași cheie și trebuie folosit atributul `sharedUserId` în fișierul Manifest. Shared UIDs sunt deprecated din Android 10 (API 29).

1.3.4 Permisiunile aplicației

Un punct central al arhitecturii de securitate a Android-ului este faptul că nicio aplicație nu are în mod implicit permisiunea de efectua operații care să afecteze alte aplicații sau sistemul.

În Android, o permisiune este un string ce semnifică abilitatea de a efectua o anumită operație în afara sandbox-ului. Aplicațiile vor cere permisiuni prin specificarea lor în fișierul `AndroidManifest.xml`.

Până la Android 6, permisiunile erau oferite la instalare și nu puteau fi modificate sau revocate mai târziu. Singura metodă de a revoca permisiunile era dezinstalarea aplicației.

De la Android 6, permisiunile sunt cerute de la utilizator în timpul rulării. În plus, utilizatorul poate revoca ulterior în mod individual orice permisiune din Settings.

Permisiunile sunt verificate (enforced) la diferite niveluri, depinzând de tipul lor. Atunci când o aplicație dorește sa acceseze resurse de nivel scăzut, cum ar fi device files, permisiunile sunt verificate de către kernel-ul Linux, prin verificarea UID și GID al aplicației apelante în relație cu proprietarul resursei și biți de acces.

Când o aplicație dorește să acceseze componente de nivel înalt al Android-ului, permisiunile sunt verificate de către Android OS sau de către o componentă specifică (sau amândouă).

1.3.5 Semnarea aplicațiilor

Toate aplicațiile Android ar trebui să fie semnate de către dezvoltator. Metoda de semnare a arhivelor APK se bazează pe semnarea arhivelor JAR.

Acest mecanism de securitate ne asigură faptul că atunci când actualizăm o aplicație, noua versiune vine de la același dezvoltator. Un alt dezvoltator nu poate actualiza aplicația deoarece nu poate produce aceeași semnătură. Acest lucru este numit: **same origin policy**.

Aplicațiile de sistem sunt semnate cu cheia de platformă. Dacă sunt semnate cu aceeași cheie atunci pot partaja resurse și chiar rula în același proces. Cheile de platformă sunt generate și gestionate de entitatea care compilează Androidul (producători, carriers, Google, utilizatori).

1.4 Bibliografie

- https://gs.statcounter.com/os-market-share/mobile/worldwide (Accesat: Mai 2021)
- https://gs.statcounter.com/os-market-share (Accesat: Mai 2021)
- https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/ (Accesat: Mai 2021)
- https://developers.google.com/android/play-protect/potentially-harmful-applications (Accesat: Mai 2021)
- https://transparencyreport.google.com/android-security/overview?hl=en (Accesat: Mai 2021)
- Karim Yaghmour. 2013. Embedded Android: Porting, Extending, and Customizing. O'Reilly Media, Inc. (Chapter 2)
- http://elinux.org/Android_Kernel_Features (Accesat: Mai 2021)
- https://developer.android.com/guide/components/activities/intro-activities (Accesat: Mai 2021)
- https://developer.android.com/guide/components/services (Accesat: Mai 2021)
- https://developer.android.com/guide/components/broadcasts (Accesat: Mai 2021)
- https://developer.android.com/guide/topics/providers/content-provider-basics (Accesat: Mai 2021)
- https://developer.android.com/guide/components/intents-filters (Accesat: Mai 2021)
- Nikolay Elenkov. 2014. Android Security Internals: An In-Depth Guide to Android's Security Architecture. No Starch Press.
- Joshua J. Drake, Zach Lanier, Collin Mulliner, Pau Oliva Fora, Stephen A. Ridley, and Georg Wicherski. 2014. Android Hacker's Handbook. Wiley Publishing.

Capitolul 2

Android SDK

2.1 Aplicații

2.1.1 Fișier Manifest

Fiecare aplicație trebuie să aibă un fișier `AndroidManifest.xml` în directorul rădăcină al aplicației. În acest fișier sunt descrise componentele aplicației și resursele de care aceasta are nevoie ca să ruleze.

În primul rând include numele aplicației și al pachetul Java. Numele pachetului este considerat un identificator unic în sistem. Nu se pot instala două aplicații cu același nume de pachet. De exemplu, dacă avem o aplicație cu numele de pachet `com.myapp` instalată de telefon și încercăm să instalăm o alta aplicație cu același nume de pachet, vom primi o eroare de instalare.

De asemenea, descrie toate componentele aplicației - activități, servicii, broadcast receivers, content providers. Pentru fiecare, se specifică clasa și capabilitățile (ce Intent-uri pot primi).

Aici este specificată și activitatea principală, cea care va porni atunci când dăm click pe iconița din home screen.

Sunt declarate permisiunile de care aplicația are nevoie pentru a apela anumite părți protejate din API sau a interacționa cu alte aplicații. De asemenea sunt specificate permisiunile de care are nevoie altă aplicație pentru a interacționa cu aplicația noastră.

Sunt specificate bibliotecile care vor fi folosite de către aplicație. Nu în ultimul rând, se specifică nivelul de API minim și țintă. De exemplu, dacă vrem ca aplicația noastră să fie compilată pentru Android 11, dar vrem să fie compatibilă și cu Android 9, vom folosi target API 30 și minimum API 28. Fiecare versiune de Android are asociat un anume nivel de API, după cum putem vedea in Tabela 2.1.

2.1.2 Permisiuni

Fiecare aplicație are nevoie de un set de permisiuni pentru a efectua anumite operații/apeluri de API sau pentru a avea acces la anumite date sau resurse.

Acest mecanism de securitate oferă protecție prin sandboxing. Asta înseamnă că

Nume	Versiune	Nivel API
Android11	11	API 30
Android10	10	API 29
Pie	9	API 28
Oreo	8.1.0	API 27
Oreo	8.0.0	API 26
Nougat	7.1	API 25
Nougat	7.0	API 24
Marshmallow	6.0	API 23
Lollipop	5.1	API 22
Lollipop	5.0	API 21
KitKat	4.4 - 4.4.4	API 19
Jelly Bean	4.3.x	API 18
Jelly Bean	4.2.x	API 17
Jelly Bean	4.1.x	API 16
Ice Cream Sandwich	4.0.3 - 4.0.4	API 15
Ice Cream Sandwich	4.0.1 - 4.0.2	API 14
Honeycomb	3.2.x	API 13
Honeycomb	3.1	API 12
Honeycomb	3.0	API 11
Gingerbread	2.3.3 - 2.3.7	API 10
Gingerbread	2.3 - 2.3.2	API 9
Froyo	2.2.x	API 8
Eclair	2.1	API 7
Eclair	2.0.1	API 6
Eclair	2.0	API 5
Donut	1.6	API 4
Cupcake	1.5	API 3
-	1.1	API 2
-	1.0	API 1

Tabelul 2.1: Asociere între versiunea de Android și nivelul de API

aplicațiile trebuie să declare capabilitățile de care au nevoie ca să funcționeze. Astfel, o aplicație nu poate efectua anumite operații fără să primească permisiunile necesare.

Permisiunile sunt specificate în fișierul Manifest folosind tag-ul `<uses-permission>`. Dacă o aplicație vrea să acceseze Internetul atunci va avea nevoie de permisiunea Internet. Astfel aplicația va putea folosi WiFi sau date mobile pentru a accesa resurse din Internet.

```
<uses-permission android:name="android.permission.INTERNET" />
```

De asemenea, putem controla cine accesează componentele aplicației noastre: pentru a porni o activitate, a porni și a face bind pe un serviciu, pentru a trimite un mesaj de broadcast, pentru a accesa datele din content provider.

În continuare este prezentat un exemplu de permisiune necesară pentru a porni o activitate. Aceasta este o permisiune custom, creată de dezvoltator.

```
<activity android:name=".ExampleActivity"
          android.permission="com.example.perm.START">
```

```
[...]
</activity>
```

Permisiunile standard nu sunt suficiente pentru lucrul cu content providers. De aceea putem avea permisiuni specifice de citire și scriere pentru URI-uri.

2.1.3 Resurse

În directorul `res/` sunt organizate resursele de genul imagini, string-uri și layout-uri. Fiecare tip de resursă se află într-un subdirector cu nume specific, de exemplu drawable, layout, values, menu, xml, etc. În aceste subdirectoare avem resursele default.

Drawables sunt imagini, layouts sunt fișiere xml care descriu cum sunt plasate elementele UI pe ecran. Values includ string-uri, menus includ descrieri ale meniurilor aplicației, xml includ alte fișiere xml folosite de aplicație.

Diferite tipuri de dispozitive Android pot avea nevoie de resurse diferite. De exemplu, pe un dispozitiv cu un ecran mai mare (tabletă) vom face un layout diferit pentru a profita de spațiul disponibil.

Pe un dispozitiv cu o setare de limbă diferită, putem avea alte valori pentru string-uri. La rulare, aplicația va folosi resursele asociate cu o anumită configurație (de exemplu limba română și ecran hdpi).

Pentru a oferi resurse alternative, vom avea subdirectoare pentru fiecare configurare alternativă. Numele subdirectorului va fi nume resursa - nume configurație (de exemplu, drawable-hdpi pentru highdensity screens - 240dpi). Pentru fiecare nume de resursă se va genera un ID unic în directorul `gen/`.

În Figura 2.1 este reprezentat un exemplu de folosire a layout-urilor diferite pentru dimensiuni de ecran diferite, pentru a folosi mai bine spațiul disponibil.

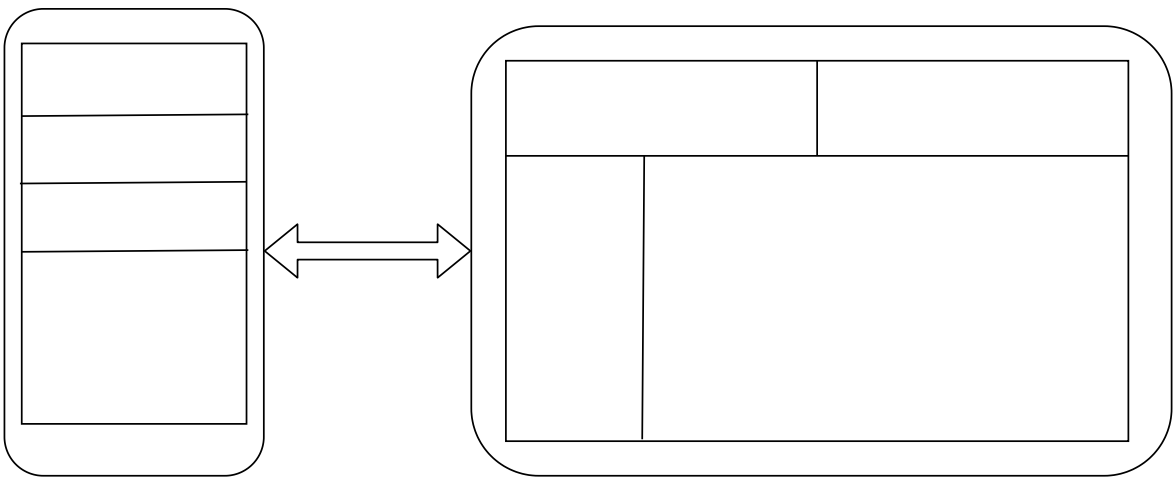

Figura 2.1: Layout-uri diferite în funcție de dimensiunea ecranului

2.1.4 Layout

Layout este o resursă inclusă în `res/layouts/` care descrie interfața cu utilizatorul pentru o activitate sau o parte a interfeței. Practic un layout va conține elemente de UI:

butoane, liste, câmpuri de text, etc.

Un layout este inclus într-un fișier de tipul `res/layout/filename.xml`, iar filename va fi ID-ul resursei respective. Acel layout poate fi referit în cod folosind `R.layout.filename`. `R.java` este generat la compilare, poate fi găsit în folderul `gen/`, și include toate ID-urile resurselor. Nu este permisă editarea manuală a fișierului `R.java` deoarece este generat automat pe baza resurselor.

Fișierele xml conținând layout-uri pot fi editate direct sau cu alte tool-uri, de exemplu Android Studio.

Acesta este un exemplu de layout pentru o activitate. Se consideră un fișier XML: `res/layout/main_activity.xml`. Acesta conține un Linear Layout care include un câmp de text și un buton.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res
/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, I am a Button" />
</LinearLayout>
```

În continuare se află codul care aplică acest layout activității.

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_activity);
}
```

2.1.5 Drawables

Drawables sunt resurse din `res/drawables/`, elemente ce pot fi afișate pe ecran. Acestea pot fi imagini sau fișiere xml.

Fișierele xml sunt folosite pentru a descrie cum va arăta un element de UI atunci când se acționează asupra lui (de exemplu când este apăsat). Aceste xml-uri se referă la imagini. De exemplu putem avea o imagine și când este apăsată să fie înlocuită cu altă imagine.

Acest lucru îmbunătățește calitatea experienței utilizatorului prin feedback-ul vizual atunci când utilizatorul interacționează cu interfața grafică.

2.2 Activități

O activitate este o componentă a aplicației care reprezintă o fereastră în care este descrisă interfața cu utilizatorul. Acesta va interacționa direct doar cu activitatea.

Pentru ca o activitate să fie afișată avem nevoie de un layout care să descrie elementele de UI și așezarea lor pe ecran.

Interfața grafică poate fi schimbată doar din thread-ul looper al activității. De aceea este bine să nu facem procesări intensiv computaționale sau blocante pe acest thread. Este de preferat să se facă aceste operații în thread-uri diferite. O regulă importantă este să nu accesăm rețeaua în thread-ul de UI, trebuie creat un thread separat pentru operațiile cu rețeaua.

O aplicație poate include mai multe activități din care doar una este cea principală care va fi pornită atunci când dăm click pe iconița din launcher (home screen).

Putem porni o activitate din altă activitate, cea veche va fi pusă pe stop iar cea nouă va fi pornită. Astfel avem o stivă de activități numită și **backstack**.

Atunci când apăsăm pe butonul back, activitatea curentă va fi distrusă și cea precedentă va fi repornită.

2.2.1 Declararea Activității în Manifest

Pentru a folosi o nouă activitate în aplicație, trebuie să fie declarată în fișierul Manifest. Mai exact, trebuie adăugat un element `<activity>` în cadrul elementului `<application>`.

Elementul `<activity>` poate să includă multe atribute, dar `android:name` este singurul obligatoriu. Acesta este și numele clasei care implementează funcționalitatea activității.

```
<manifest ... >
  <application ... >
      <activity android:name=".ExampleActivity" />
      ...
  </application ... >
  ...
</manifest >
```

2.2.2 Ciclul de viață al unei Activități

În Figura 2.2 este o reprezentare detaliată a ciclului de viață al unei activități. Metodele pe care le vedem aici sunt callback-uri și sunt apelate de către sistem (de către serviciul `ActivityManager`), nu de către aplicația în sine.

Când o activitate este pornită, se apelează întâi `onCreate()`, care este punctul de intrare în activitate, apoi activitatea intră în starea Created. În acest moment, activitatea își obține layout-ul, dar nu îl desenează pe ecran.

Apoi `onStart()` este apelat, care va desena layout-ul pe ecran iar activitatea va intra în starea Started, în care este vizibilă pe ecran.

Apoi `onResume()` este apelat, și activitatea trece în starea Resumed, în care este vizibilă și acceptă input-ul utilizatorului.

Atunci când primim o notificare sau apare un dialog box, `onPause()` este apelat, iar activitatea va trece în starea Paused, în care este parțial vizibilă și utilizatorul nu poate

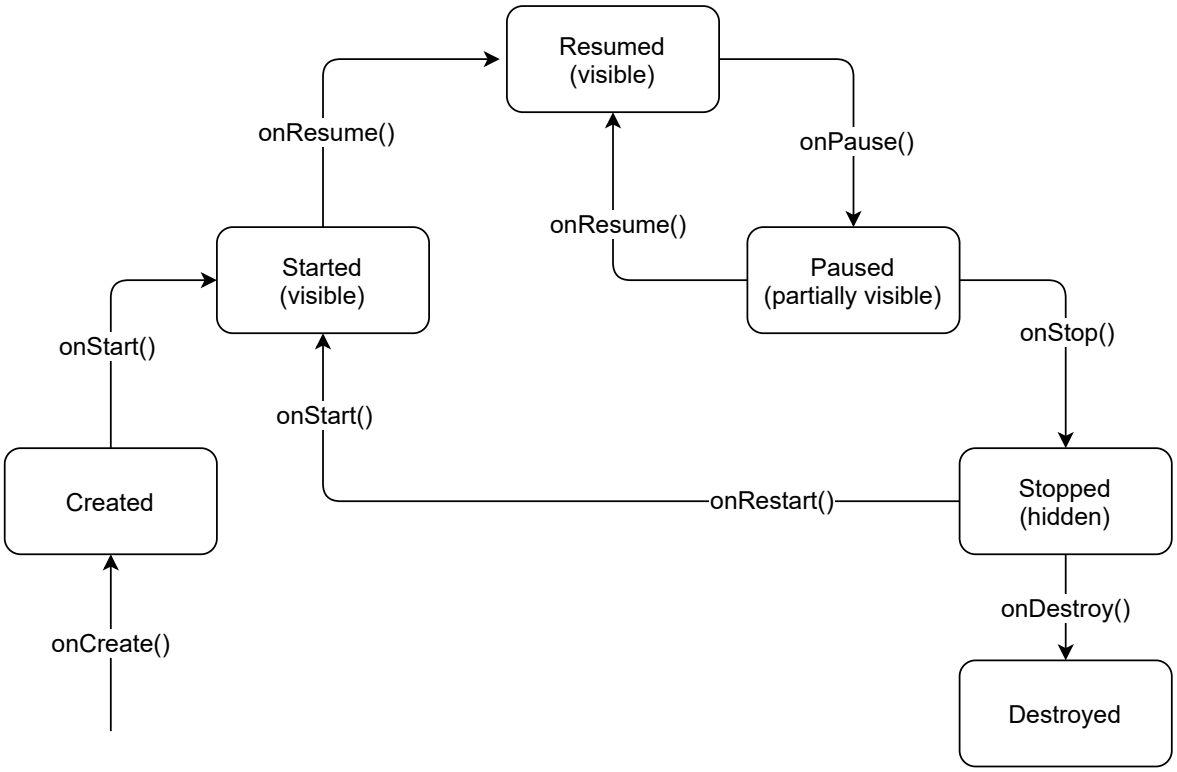

Figura 2.2: Ciclul de viață a unei activități

interacționa cu ea. Dacă închidem notificarea/dialogul, `onResume()` este apelat și activitatea intră în starea Resumed. Layout-ul nu este redesenat, activitatea are focus din nou și primește input-ul utilizatorului.

Atunci când din activitatea A pornim activitatea B, se va apela `onPause()` și `onStop()` din A, iar activitatea A va rămâne în starea Stopped în backstack în timp ce activitatea B își începe ciclul de viață. Atunci când activitatea este în foreground și apăsăm back, această activitate va fi distrusă și apoi se vor apela `onRestart()`, `onStart()` și `onResume()` pentru activitatea A. De ce trebuie să se treacă din nou prin starea Started? Deoarece este posibil să fie necesară redesenarea activității.

Atunci când rulează activitatea A și se apasă back, se vor apela `onPause()`, `onStop()` și `onDestroy()`, pentru a distruge acea activitate și a reveni la cea anterioară.

În starea Stopped/Paused activitatea poate fi omorâtă de Low Memory Killer pentru că altă aplicație cu prioritate mai mare are nevoie de memorie. În acest caz, atunci când utilizatorul va intra din nou în activitate, se va apela `onCreate()`, `onStart()`, `onResume()`, și activitatea este recreată de la zero.

În continuare este prezentat scheletul de cod al unei activități care include toate metodele ciclului de viață.

```
public class ExampleActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Activitatea este creată.
    }
    @Override
```

```
    protected void onStart() {
        super.onStart();
        // Activitatea urmează să devină vizibilă.
    }
    @Override
    protected void onResume() {
        super.onResume();
        // Activitatea a devenit vizibilă și se poate interacționa cu ea.
    }
    @Override
    protected void onPause() {
        super.onPause();
        // Activitatea este pusă pe pauză.
        //Altă activitate/dialog box are vizibilitate si focus.
    }
    @Override
    protected void onStop() {
        super.onStop();
        // Activitatea nu mai este vizibilă, este pusă pe stop.
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // Activitatea urmează să fie distrusă.
    }
}
```

2.2.3 Salvarea și restaurarea stării unei Activități

Activitățile pot fi omorâte de către sistem dacă este nevoie de memorie pentru alte aplicații mai prioritare.

În acel moment se pierde starea activității. Avem posibilitatea de a salva această stare, incluzând primitive, obiecte, UI, prin callback-ul `onSaveInstanceState()`. În acest callback trebuie să salvăm toate informațiile într-un singur Bundle.

Starea activității va putea fi restaurată în `onCreate()` sau în `onRestoreInstanceState()` care primesc ca argument Bundle-ul respectiv.

Totuși, este recomandat să salvăm starea activității doar atunci când este foarte important să facem asta, pentru a nu ocupa memoria inutil.

Atunci când se omoară procesul se vor omorî și thread-urile din activitate. E bine ca acestea să fie oprite înainte ca procesul să fie omorât mai ales dacă se fac operații critice de genul scris în fișiere. De aceea este recomandat ca în `onPause()` să semnalăm thread-urilor să se oprească.

În Figura 2.3 este prezentat procesul de salvare a stării și restaurare. Observăm ca nu trebuie să facem această restaurare dacă procesul nu este omorât pentru că starea activității se păstrează în Paused și Stopped.

2.2.4 Fragmente

Fragmentele sunt elemente de UI conținute într-o activitate. Sunt ca un fel de activități mai mici în cadrul unei activități, și au propriul ciclu de viață.

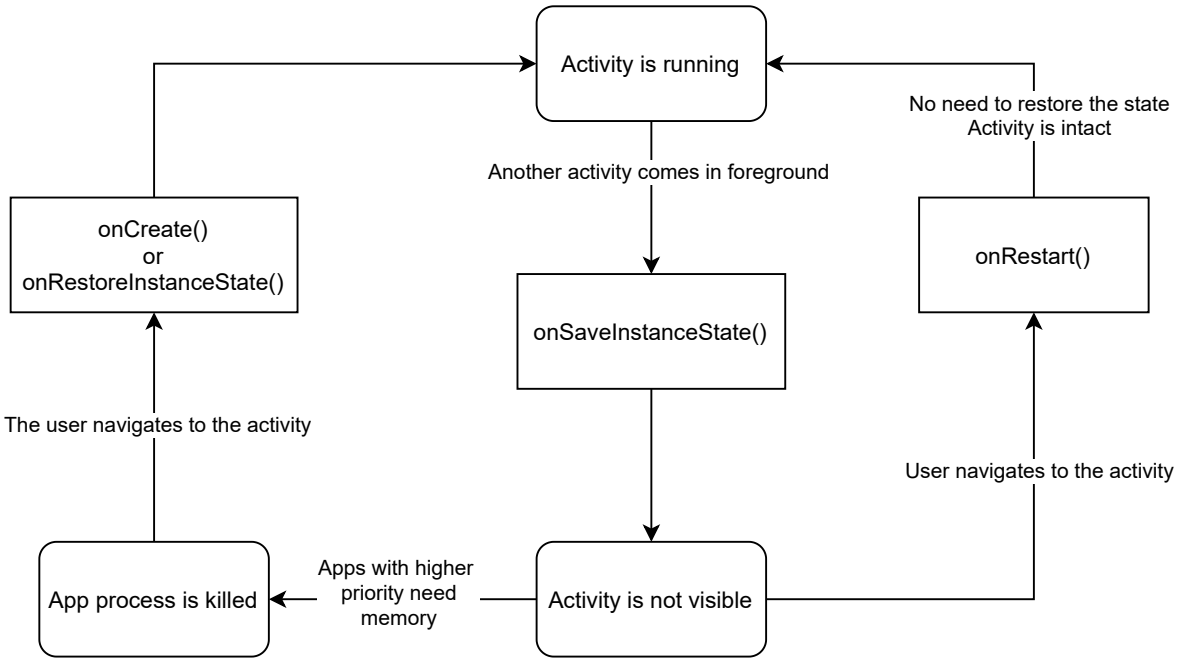

Figura 2.3: Procesul de salvare și restaurare a stării unei activități

Ele pot fi combinate în diferite moduri pentru a construi UI-ul în funcție de layout. De exemplu, pe un telefon și o tabletă, pot fi rearanjate în moduri diferite, dar codul fragmentelor rămâne același. De exemplu, dacă suntem în mod landscape pe o tabletă vom avea o aranjare diferită a fragmentelor decât în mod portret pe un telefon. În plus, fragmentele pot fi refolosite și în alte activități.

În Figura 2.4 este reprezentat un exemplu cu două fragmente afișate pe dispozitive diferite. Prin selectarea unui element din fragmentul A, se actualizează fragmentul B. Pe o tabletă putem combina cele două fragmente în activitatea A, iar pe un telefon nu va fi suficient spațiu pentru ambele fragmente, deci activitatea A va conține fragmentul A și activitatea B va conține fragmentul B.

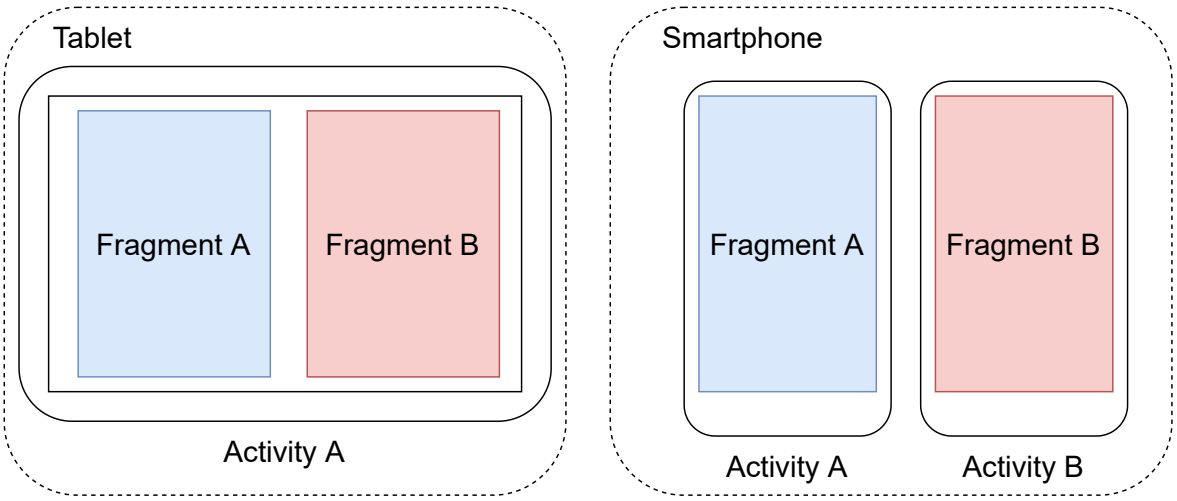

Figura 2.4: Fragmente afișate pe dispozitive diferite

2.2.5 Ciclul de viață al unui Fragment

Ciclul de viață al unui fragment este direct afectat de ciclul de viață al activității părinte. Când activitatea este pe pauza, fragmentele vor fi pe pauza. Când activitatea este distrusă, fragmentele vor fi distruse. Atunci când o activitate este în starea Resumed, se pot adăuga și scoate fragmente. Aceste operații se numesc tranzacții cu fragmente.

Există un backstack pentru fragmente asociat cu activitatea. Fiecare intrare în backstack conține câte o tranzacție efectuată. Atunci când apăsăm back, se va anula ultima tranzacție.

Un fragment nu va primi layout-ul în `onCreate()`, ci în callback-ul `onCreateView()` (Figura 2.5). Această metodă este apelată atunci când fragmentul știe la ce activitate este atașată și cât spațiu ocupă în cadrul acelei activități. Distrugerea fragmentului va avea loc în `onDestroyView()`, iar aici trebuie să fie oprite și thread-urile.

2.2.6 Interfața cu utilizatorul

Interfața cu utilizatorul este realizată printr-o ierarhie de view-uri (derivate din clasa View: Button, TextView, Checkbox, etc).

Fiecare View controlează un spațiu dreptunghiular în activitate și oferă interacțiune cu utilizatorul. Exemple de View-uri: butoane, liste, imagini, căsuțe de text, etc.

Pentru interacțiunea cu aceste obiecte există callback-uri în care putem specifica ce acțiuni să se execute, de exemplu dacă este apăsat un buton (se apelează callback-ul `onClick()`).

Un ViewGroup va putea conține mai multe obiecte View și alte ViewGroup-uri. Pentru a crea View-uri mai complexe se pot extinde clasele actuale.

De asemenea, pentru a afișa tipuri de date complexe se pot folosi adaptori precum: ArrayAdapter, ListAdapter, CursorAdapter, etc.

2.3 Servicii

Un serviciu este o componentă a unei aplicații care poate execută operații care durează mult, în background, și care nu oferă interfață cu utilizatorul.

El va continua să ruleze chiar dacă utilizatorul a deschis o altă aplicație care apare în foreground. De exemplu, o aplicație de muzică ar trebui să poată reda muzică chiar și atunci când aplicația este în background.

Un serviciu poate face operații cu rețeaua, operații I/O pe fișiere, poate interacționa cu un content provider, etc.

Un serviciu va rula, în mod implicit, pe thread-ul principal al procesului - nu se va crea un nou thread și nu va rula în alt proces decât dacă specificăm în mod explicit. Prin urmare, dacă vrem să facem operații CPU intensive sau operații blocante, e bine să creăm un nou thread pentru serviciu.

Un serviciu poate fi pornit folosind un Intent din aplicația curentă sau din altă aplicație. Dacă vrem să blocam accesul altor aplicații la serviciul nostru, trebuie sa-l declarăm privat în fișierul Manifest.

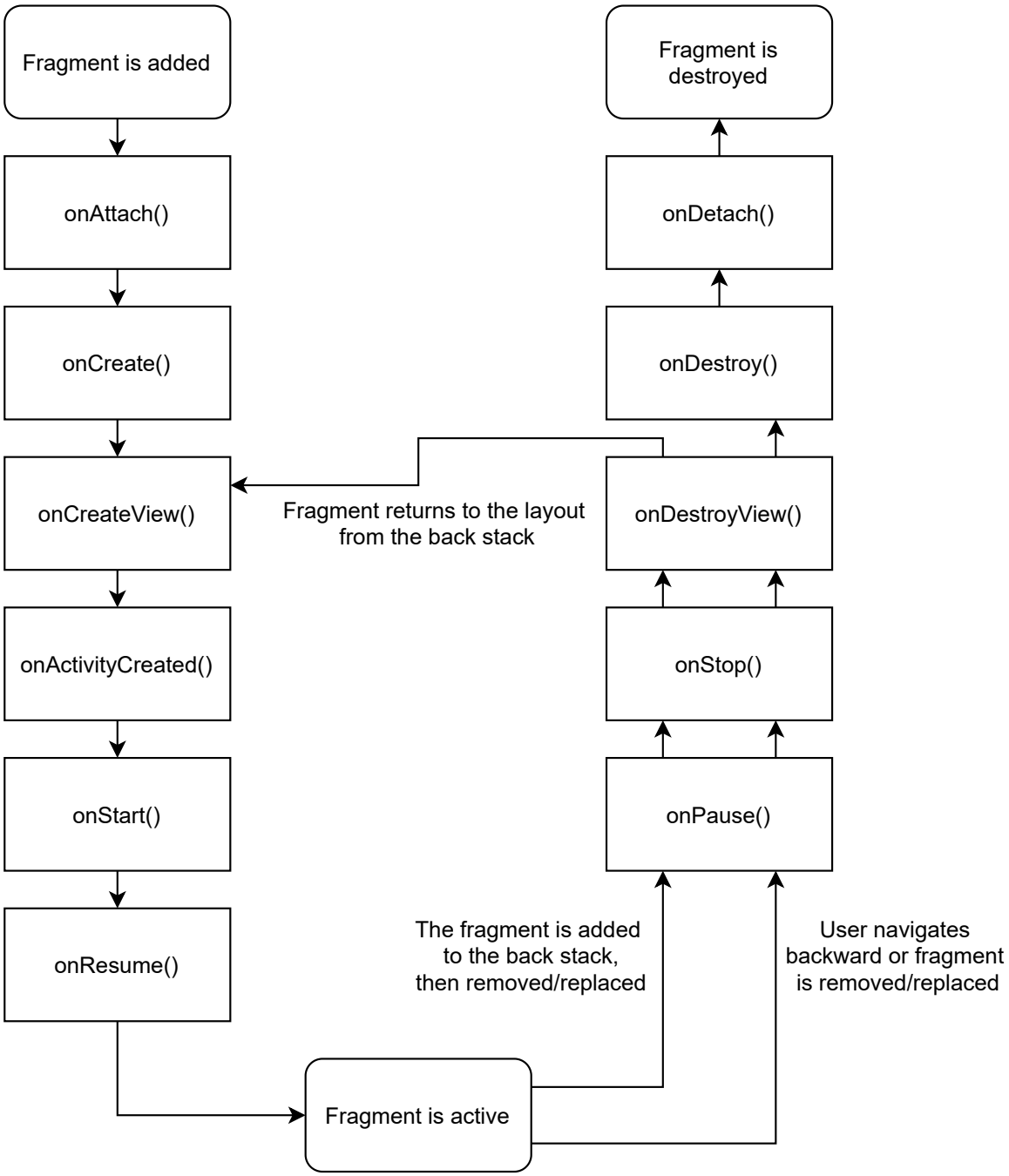

Figura 2.5: Ciclul de viață al unui fragment

2.3.1 Declararea Serviciilor în Manifest

Serviciile trebuie declarate în fișierul Manifest. În continuare este prezentat un exemplu: se adaugă un element `<service>` în cadrul elementului `<application>`.

```
<manifest ... >
  ...
  <application ... >
      <service android:name=".ExampleService"
               android:exported="false" />
      ...
  </application>
</manifest>
```

Se pot folosi multe atribute, de exemplu permisiunea de care are nevoie altă aplicație pentru a porni acest serviciu. De asemenea se poate configura serviciul să ruleze într-un proces separat. Atributul `android:name` este singurul obligatoriu. Putem face serviciul privat folosind atributul `android:exported` setat pe false.

2.3.2 Tipuri de Servicii

Există două tipuri de servicii: Started și Bound.

Un serviciu este **Started** atunci când o componentă a unei aplicații apelează `startService()`. El va rula indefinit, chiar dacă componenta care l-a pornit este distrusă. Un astfel de serviciu face o anumită operație apoi se oprește, fără a returna vreun rezultat spre componenta care l-a pornit.

Un serviciu este **Bound** atunci când o componentă se leagă la el folosind funcția `bindService()`. Un astfel de serviciu va oferi o interfață client-server în care se vor putea trimite cereri și primi răspunsuri. AIDL va putea fi folosit pentru a genera interfața folosită între client și serviciu. Acest serviciu va rula doar până când componenta care l-a pornit face unbind.

Android Interface Definition Language (AIDL) este un limbaj de descriere a interfețelor care permite definirea interfeței între client și serviciu, pentru ca aceștia să comunice folosind un IPC (interprocess communication). Obiectele transmise trebuie descompuse în primitive care sunt înțelese de către sistemul de operare. Android-ul se va ocupa de serializarea și deserializarea datelor transmise.

Următoarea situație necesită atenție din partea programatorului: dacă se face bind la un serviciu dintr-o activitate, și activitatea este pusa în starea Stopped, atunci când utilizatorul se întoarce la acea activitate, serviciul poate fi `null` pentru că sistemul a avut nevoie de memorie și a omorât serviciul. De aceea este recomandat să se verifice dacă serviciul este `null` pentru a evita `NullPointerException`.

Pot exista mai multe componente care fac bind pe un serviciu. În acest caz, serviciul va fi distrus abia după ce toate fac unbind.

Un serviciu poate fi Started și Bound în același timp.

2.3.3 Ciclul de viață al unui Serviciu

În Figura 2.6 este reprezentat ciclul de viață al unui serviciu în cele două cazuri (Started și Bound).

În primul caz, când o componentă apelează `startService()`, se rulează `onCreate()`, apoi `onStartCommand()`, apoi serviciul rulează.

După aceea, serviciul se poate opri cu `stopSelf()` sau poate fi oprit de către altă componentă cu `stopService()`. Înainte să fie omorât serviciul se va apela `onDestroy()` - aici ar trebui să aibă loc clean-up-ul.

În al doilea caz, un serviciu este pornit cu `bindService()`, se rulează `onCreate()`, `onBind()`, iar apoi clientul este legat la serviciu și poate comunica cu el. După ce toți clienții fac unbind, se apelează `onUnbind()` și apoi `onDestroy()`.

Un serviciu poate fi Started și Bound în același timp. Chiar dacă serviciul a fost pornit folosind `startService()`, poate primi apeluri la `onBind()` (atunci când clienții apelează `bindService()`).

În continuare este reprezentat scheletul de cod pentru un serviciu, incluzând toate callback-urile:

```
public class ExampleService extends Service {
    int mStartMode;       // indică ce trebuie făcut în cazul
                         // în care serviciul este omorât
    IBinder mBinder;      // interfața pentru clienții
                         // care se leagă la serviciu
    boolean mAllowRebind; // indică dacă rebind-ul este permis

    @Override
    public void onCreate() {
        // Serviciul este creat
    }
    @Override
    public int onStartCommand(Intent intent, int flags,
                                                int startId) {
        // Serviciul este pornit în urma unui apel startService()
        return mStartMode;
    }
    @Override
    public IBinder onBind(Intent intent) {
        // Un client s-a legat la serviciu folosind apelul bindService()
        return mBinder;
    }
    @Override
    public boolean onUnbind(Intent intent) {
        // Toți clienții au făcut unbind folosind unbindService()
        return mAllowRebind;
    }
    @Override
    public void onRebind(Intent intent) {
        // Un client se leagă la serviciu folosind bindService()
        // după ce a apelat deja unbindService()
    }
    @Override
    public void onDestroy() {
        // Serviciul nu mai este folosit și este distrus
    }
```

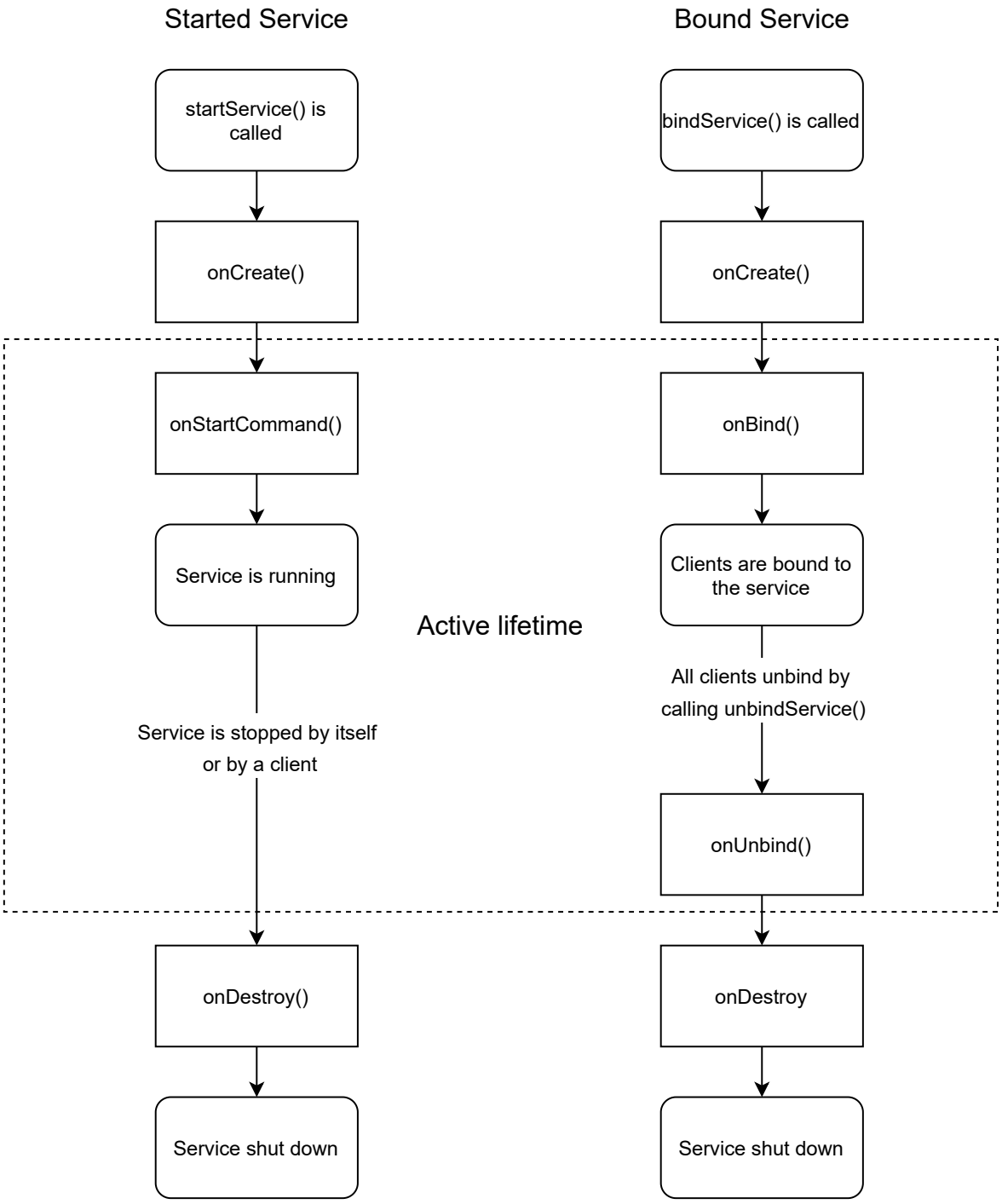

Figura 2.6: Ciclul de viață al unui serviciu

```
}
```

2.4 Intent-uri

Un intent este folosit pentru a trimite un mesaj sau a determina execuția unei acțiuni.

Un intent are trei componente principale: numele componentei destinație, acțiunea care trebuie efectuată și datele folosite în acțiune (de exemplu, sună la un anumit număr de telefon).

Pentru a specifica ce intent-uri acceptă o anumită componentă, se va declara un `<intent-filter>` în Manifest, iar acțiunea și tipul de date se vor specifica în tag-urile `<action>` și `<data>` incluse în intent-filter.

2.4.1 Cazuri de utilizare a Intent-urilor

Un intent poate fi folosit în următoarele cazuri: pentru a porni o activitate, pentru a porni sau a face bind pe un serviciu, și pentru livrarea unui mesaj de broadcast.

O activitate poate fi pornită prin folosirea unui intent la apelarea metodei `startActivity()`. Intent-ul include informații despre activitatea ce trebuie pornită și datele necesare.

De asemenea, o activitate poate fi pornită prin pasarea unui intent la apelarea metodei `startActivityForResult()`. Rezultatul va fi primit printr-un intent separat.

Un serviciu poate fi pornit prin pasarea unui intent la apelarea metodei `startService()`. Intent-ul include informațiile necesare despre serviciul țintă și date.

De asemenea, se poate face bind la un serviciu prin pasarea unui intent la metoda `bindService()`.

Un mesaj de broadcast poate fi trimis prin folosirea unui intent la apelarea uneia dintre metodele: `sendBroadcast()`, `sendOrderedBroadcast()`.

2.4.2 Tipuri de Intent-uri

Există două tipuri de intent-uri: explicite și implicite.

La cele explicite se specifică componenta destinație, mai exact numele clasei. De obicei se folosește pentru a porni o activitate sau un serviciu în cadrul aceleiași aplicații.

Nu este nevoie să specificăm un intent-filter daca vrem să folosim intent-uri explicite, acestea vor fi livrate chiar dacă nu este declarat un intent-filter.

Intent-urile implicite nu specifică numele componentei țintă, ci doar acțiunea de executat.

Sistemul Android va căuta cea mai potrivită componentă care poate executa acea acțiune. Va face asta prin căutarea unei potriviri între intent-filter-ele din fișierele Manifest și intent-ul respectiv.

Dacă se găsesc mai multe, utilizatorul va fi întrebat ce aplicație vrea să folosească (de exemplu pentru citirea unui fișier pdf).

Deci în cazul acesta avem nevoie clară de intent-filters declarate în Manifest.

2.4.3 Intent-uri implicite

În Figura 2.7 vedem cum sunt livrate intent-urile implicite. Activitatea A crează un obiect Intent cu acțiunea asociată și îl dă ca argument metodei `startActivity()`. Sistemul caută toate intent-filters care se potrivesc cu acțiunea respectivă. Atunci când se găsește activitatea potrivită, se apelează `onCreate()` și apoi se trimite obiectul Intent activității respective.

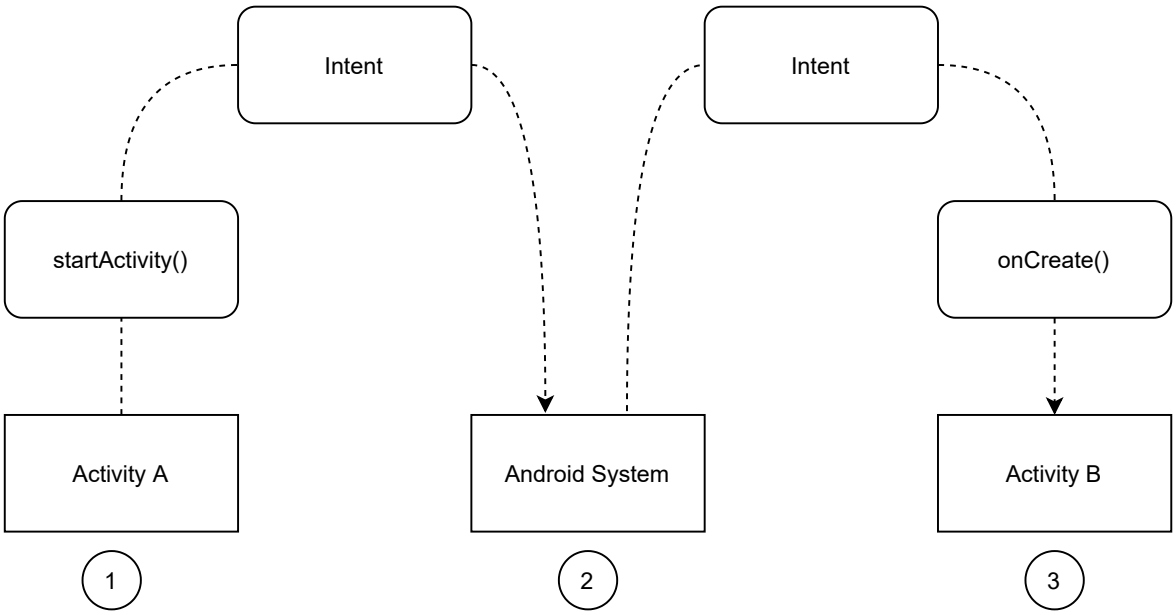

Figura 2.7: Intent-uri implicite

Acesta este un exemplu de intent implicit. Presupunem că avem de partajat anumite date. Creăm un intent, configurăm acțiunea `ACTION_SEND`, adăugam mesajul text (tipul de date este text) și apelăm metoda `startActivity()` cu acel intent.

Dacă nu există o activitate care va primi acest tip de intent, pot apărea erori în aplicație. De aceea este recomandat să verificăm că există o aplicație care poate primi acel intent înainte să-l trimitem, prin metoda `resolveActivity()`. Dacă rezultatul este diferit de `null`, atunci există cel puțin o activitate care poate primi acest timp de intent.

```
// Se crează un Intent ce conține un mesaj de tip text
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
sendIntent.setType("text/plain");

// Se verifică dacă există o activitate care să poată primi Intent-ul
if (sendIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(sendIntent);
}
```

Acesta este un exemplu de declarație a unei activități în fișierul Manifest, care include un intent-filter cu acțiunea SEND și tipul de date text. Acest lucru specifică faptul că activitatea poate primi intent-ul trimis în codul prezentat anterior.

```
<activity android:name=".ExampleActivity">
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
    </intent-filter>
</activity>
```

2.5 Broadcast Receivers

Un broadcast receiver este o componentă care tratează mesajele de broadcast care sunt trimise în sistem. Mesajele de broadcast sunt notificări sau anunțuri.

Majoritatea mesajelor de broadcast sunt generate chiar de către sistem, de exemplu: când bateria este pe terminate, când s-a stins ecranul, a fost primit un SMS, etc.

Dar aplicațiile pot genera și ele mesaje de broadcast pentru a face diverse anunțuri către aplicații.

Deși acești receivers nu au interfață grafică, ei pot genera notificări în status bar pentru a anunța utilizatorul.

Un receiver de obicei nu face procesări elaborate, deoarece întotdeauna rulează pe thread-ul de UI. El va lăsa alte componente, de exemplu un serviciu, să facă operațiile mai complexe, care sunt determinate de către eveniment.

Orice mesaj de broadcast este trimis folosind un Intent (cu `sendBroadcast()` sau `sendOrderedBroadcast()`).

Dacă vrem să folosim mesaje de broadcast doar în aplicația noastră atunci este mai sigur să folosim `LocalBroadcastManager`, deoarece este mai eficient (nu se duce prin sistem), datele nu ies din aplicație, iar o altă aplicație nu poate să trimită acele mesaje (nu avem breșe de securitate).

Un receiver se poate declara static folosind tag-ul `<receiver>` în Manifest sau dinamic folosind metoda `Context.registerReceiver()`. Este recomandată folosirea tag-ului `<receiver>` pentru a păstra codul curat.

2.5.1 Tipuri de mesaje Broadcast

Putem avea două tipuri de mesaje de broadcast:

- Normale
- Ordonate

Mesajele de broadcast normale sunt complet asincrone. Receiver-ii pentru acel mesaj rulează într-o ordine nedefinită, posibil chiar în același timp. Trimiterea unui mesaj de broadcast normal se face prin pasarea unui intent funcției `sendBroadcast()`.

Mesajele de broadcast ordonate sunt livrate la câte un receiver odată (receiveri ordonați). Trimiterea unui mesaj de broadcast ordonat se face prin pasarea unui intent funcției `sendOrderedBroadcast()`.

Receiver-ul se execută și apoi poate alege să propage rezultatele la următorul receiver sau să nu trimită broadcast-ul mai departe la următorul receiver (abort broadcast).

Ordinea receiver-ilor este determinată folosind `android:priority` în intent-filter-ul din codul receiver-ului.

2.5.2 Declararea unui Broadcast Receiver în Manifest

Acesta este un exemplu de declarare a unui broadcast receiver în fișierul Manifest. Include un intent filter pentru evenimentul `BOOT_COMPLETED`. Pentru acest lucru, trebuie cerută permisiunea `RECEIVE_BOOT_COMPLETED`.

```
<manifest ... >
  <uses-permission android:name=
                "android.permission.RECEIVE_BOOT_COMPLETED" />
  <application ... >
    <receiver android:name="ExampleReceiver" >
      <intent-filter>
        <action android:name=
                "android.intent.action.BOOT_COMPLETED" />
      </intent-filter>
    </receiver>
      ...
  </application ... >
  ...
</manifest >
```

2.5.3 Implementarea unui Broadcast Receiver

Acesta este un exemplu de implementare al receiver-ului. Callback-ul `onReceive()` va fi apelat atunci când evenimentul `BOOT_COMPLETED` are loc. În callback este pornit un serviciu.

```
public class ExampleReceiver extends BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent) {
                Intent intent = new Intent(context,
                                        ExampleService.class);
                context.startService(intent);
        }
}
```

2.6 Content Providers

Un content provider este o componentă care oferă acces la un set de date. De obicei o aplicație oferă acces altei aplicații la datele sale folosind un provider.

Există content providers de sistem (cei default), de exemplu: contacts, dictionary, calendar, settings, etc.

Pentru a accesa un provider, aplicația noastră are nevoie de permisiuni specifice (care sunt declarate în fișierul Manifest) de read sau write pe datele respective. De exemplu, dacă vrem să citim date din dicționar, ne trebuie permisiunea `READ_USER_DICTIONARY`.

Există 2 metode de a stoca datele:

1. Folosind fișiere, de exemplu audio, video, poze
2. Folosind date structurate sub forma de tabele cu linii și coloane (de exemplu baza de date sau array-uri). De obicei se folosește SQLite pentru stocarea datelor.

Se folosește un provider dacă vrem să accesăm datele unei aplicații din altă aplicație, îi vom numi provider și client.

Aplicația care deține datele va include provider-ul iar aplicația care va accesa datele va include clientul.

Este nevoie de un obiect client numit ContentResolver. Prin metodele sale vom putea crea, accesa, actualiza și șterge date.

Atunci când apelăm metodele sale, se vor apela metodele cu același nume din ContentProvider.

2.6.1 Content URIs

Pentru a identifica datele se vor folosi Content URI-uri.

Un URI are două componente: authority este numele provider-ului, și path este numele tabelei. Un exemplu de URI este `content://user_dictionary/words`. Providerul este user_dictionary iar tabela este words.

Astfel, ContentResolver-ul va citi acest URI și va identifica provider-ul. Va căuta într-o tabelă de sistem cu toți provider-ii și va obține acces la acest provider.

Resolver-ul va realiza un query către provider pentru acel path.

Provider-ul va folosi path-ul pentru a identifica tabela și a face operația cerută pe aceea tabelă.

2.6.2 Operații asupra unui Content Provider

Acestea sunt câteva exemple de operații care pot fi efectuate asupra Content Provider-ului numit user dictionary. Putem vedea aici 3 operații: query, insert și update. Pentru fiecare operație dăm Content URI-ul tabelei Words ca argument. Projection specifică coloanele care vor fi incluse pentru fiecare rând selectat. Putem specifica criteriile de selecție pentru rânduri. În final putem specifica ordinea de sortare a rândurilor returnate.

```
mCursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,
    mProjection,
    mSelectionClause,
    mSelectionArgs,
    mSortOrder);
[...]
mNewUri = getContentResolver().insert(
```

```
    UserDictionary.Words.CONTENT_URI,
    mNewValues);
[...]
mRowsUpdated = getContentResolver().update(
    UserDictionary.Words.CONTENT_URI,
    mUpdateValues,
    mSelectionClause,
    mSelectionArgs);
```

2.7 Utilitare

În continuare vor fi prezentate câteva utilitare, folosite în implementarea aplicațiilor Android.

2.7.1 Android Studio

Android Studio (Figura 2.8) este IDE-ul oficial pentru dezvoltarea aplicațiilor Android. Include un sistem de build bazat pe Gradle.

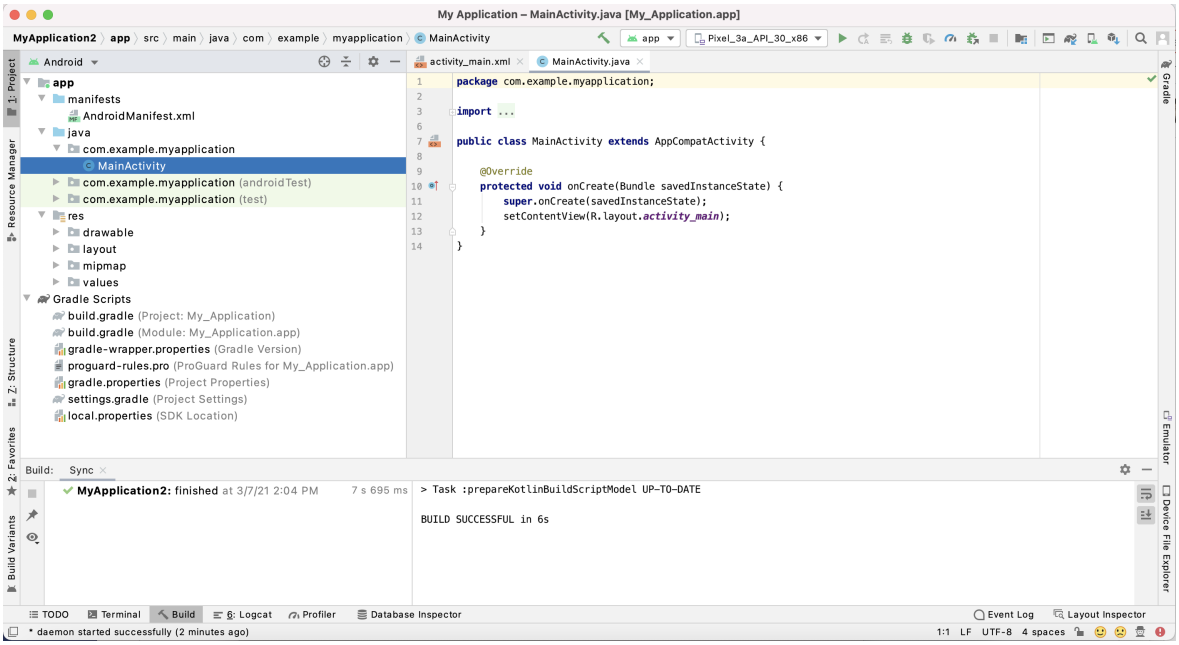

Figura 2.8: Android Studio

2.7.2 SDK Manager

Android SDK Manager (Figura 2.9) este folosit pentru descărcarea și gestionarea pachetelor SDK, a exemplelor, a imaginilor de sistem pentru emulator și a tool-urilor (platform și build tools). Putem alege să instalăm sau să dezinstalăm diferite pachete.

2.7.3 AVD Manager

Prin Android Virtual Device (AVD) Manager (Figura 2.10) putem crea și gestiona dispozitive virtuale care vor fi folosite de către emulator.

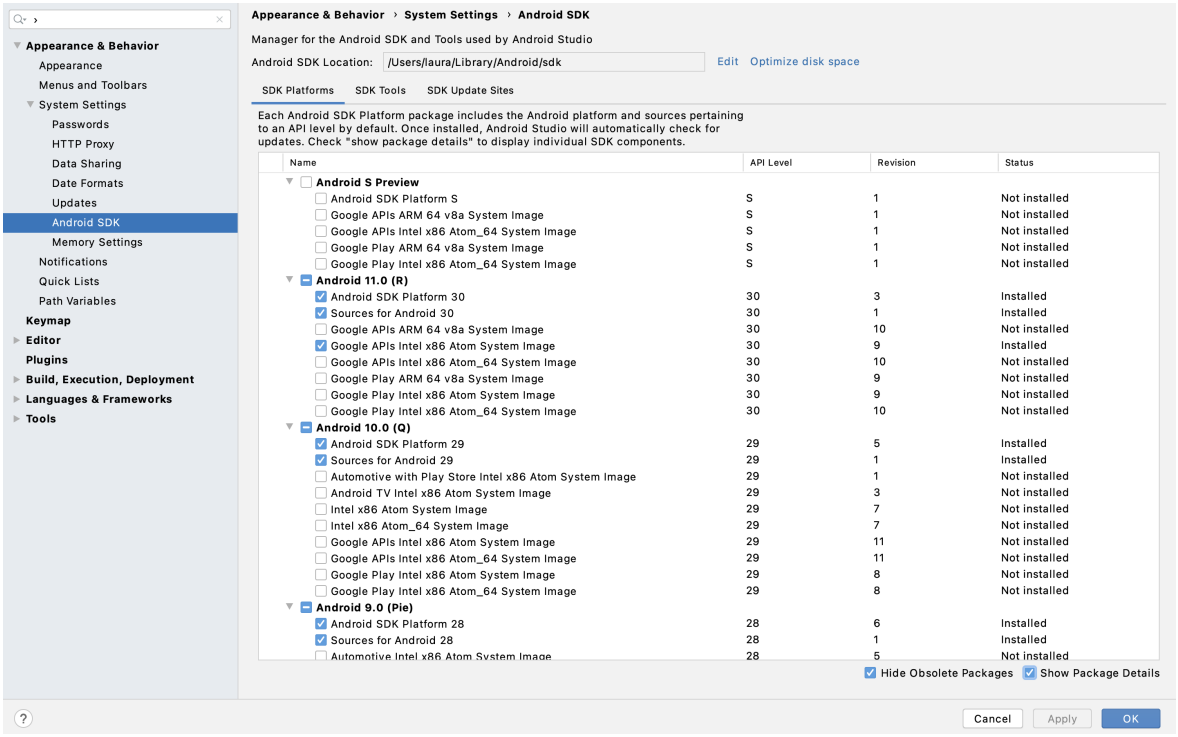

Figura 2.9: SDK Manager

Emulatorul va rula dispozitive virtuale și astfel se pot testa aplicațiile pe calculator, fără să fie nevoie de un dispozitiv fizic.

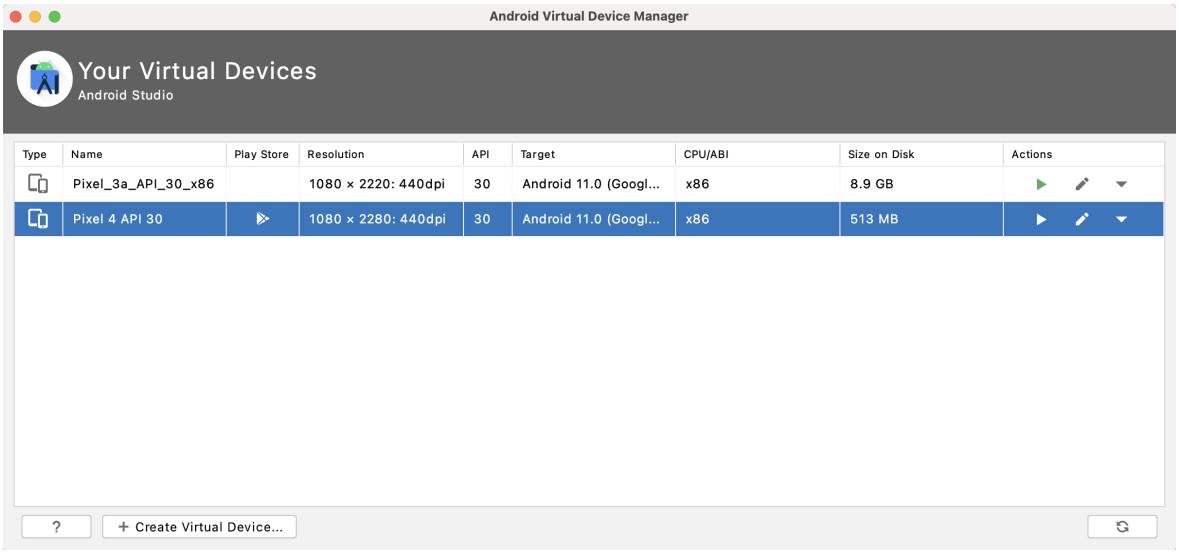

Figura 2.10: AVD Manager

2.7.4 Emulatorul

Emulatorul de Android se bazează pe QEMU. Acesta oferă posibilitatea de lucru cu mai multe componente: display, tastatură, rețea, GPS, audio, radio, etc.

Pentru a crește viteza de virtualizare este nevoie de o imagine de Android de x86 și de KVM pe Linux și de HAXM pe Windows (pentru că nu se poate modifica kernel-ul).

Astfel, nu este nevoie ca instrucțiunile să fie translatate, ci pot rula direct pe procesorul PC-ului.

În plus, în locul folosirii unui GPU virtual, se poate folosi GPU-ul PC-ului și astfel vom obține o performanță mai bună.

2.7.5 ADB

Android Debug Bridge (ADB) este folosit pentru comunicarea cu dispozitivul real sau virtual. ADB are 3 componente principale: un client care rulează pe PC, un server care rulează pe PC și un daemon care rulează pe dispozitivul real sau virtual.

Atunci când dăm în terminal `adb` și o comandă, aceasta va fi trimisă serverului, care o va executa comunicând cu daemon-ul de pe dispozitiv.

Se pot face următoarele acțiuni:

- Se pot copia fișiere de pe PC pe telefon și invers - `adb push`, `adb pull`
- Se pot instala aplicații pe telefon - `adb install`
- Se pot afișa mesaje de debug - `adb logcat`
- Se poate obține un shell pe dispozitiv - `adb shell`

2.8 Bibliografie

- http://developer.android.com/guide/topics/manifest/manifest-intro.html (Accesat: Mai 2021)
- http://developer.android.com/guide/topics/resources/overview.html (Accesat: Mai 2021)
- https://developer.android.com/guide/components/activities/intro-activities (Accesat: Mai 2021)
- https://developer.android.com/guide/components/services (Accesat: Mai 2021)
- https://developer.android.com/guide/components/broadcasts (Accesat: Mai 2021)
- https://developer.android.com/guide/topics/providers/content-provider-basics (Accesat: Mai 2021)
- https://developer.android.com/guide/components/intents-filters (Accesat: Mai 2021)
- https://developer.android.com/studio/command-line/index.html (Accesat: Mai 2021)

Capitolul 3

Internele sistemului de operare Android

3.1 Achitectura Android-ului

Arhitectura Android-ului este reprezentată în Figura 3.1.

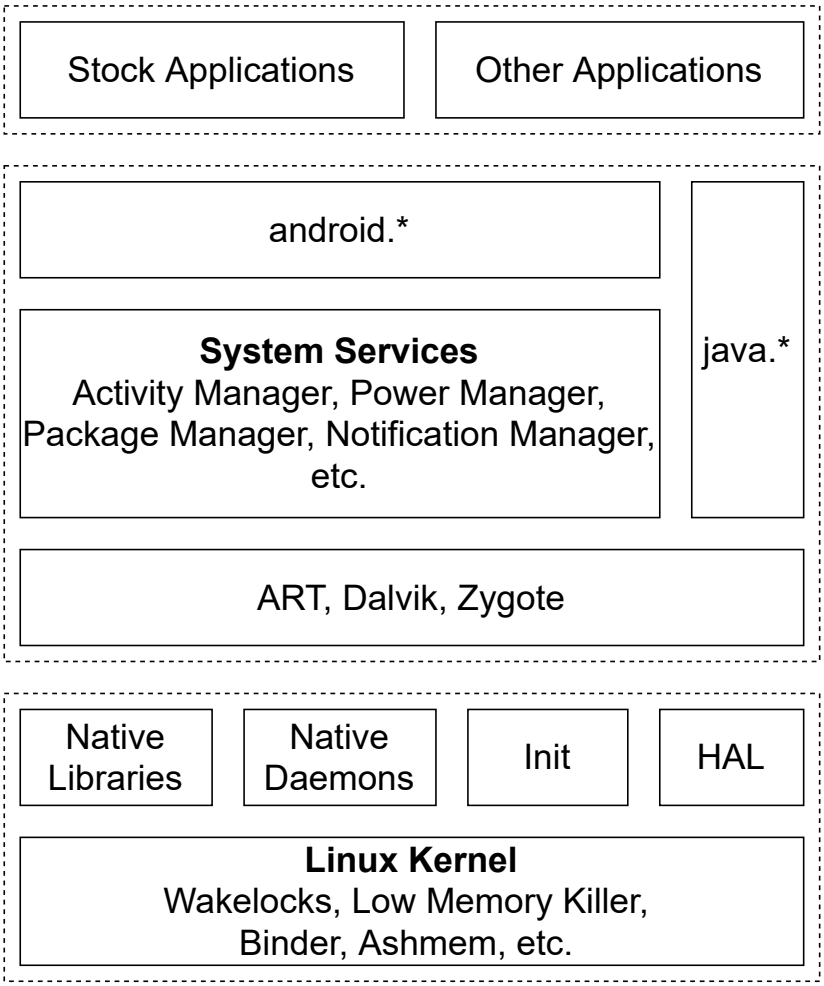

Figura 3.1: Android Architecture

Acesta rulează deasupra unui kernel de Linux vanilla, cu câteva modificări numite

Androidisme (în mod informal).

Userspace-ul nativ include procesul init, câțiva daemoni nativi, câteva sute de biblioteci native și Hardware Abstraction Layer (HAL).

Bibliotecile native sunt implementate in C/C++ și funcționalitatea lor este expusă aplicațiilor prin API-ul framework-ului de Java (Java framework APIs).

O mare parte din Android este implementată în Java și până la Android 5.0, runtime-ul implicit a fost Dalvik. De la Android 5.0, a fost integrat un runtime mai performant, numit ART.

Bibliotecile de runtime Java (Java runtime libraries) sunt definite în pachetele `java.*` și `javax.*`, și sunt derivate din proiectul Apache Harmony (nu din Oracle/SUN, pentru a evita problemele de copyright și distribuție). Codul nativ poate fi apelat din Java prin Java Native Interface (JNI).

O mare parte din funcționalitățile Android-ului este implementată în serviciile de system: display, touch, telefonie, conectivitate la rețea, etc. O parte dintre aceste servicii este implementată cod nativ, și cealaltă parte în Java. Fiecare serviciu oferă o interfață care poate fi apelată din alte componente.

Bibliotecile Android Framework, includ clase pentru construirea aplicațiilor Android.

Deasupra, avem aplicațiile stock, cele care vin odată cu telefonul. De asemenea, avem aplicațiile instalate de către utilizator.

3.2 Kernel-ul Linux

Majoritatea distribuțiilor de Linux iau kernel-ul vanilla (din Linux Kernel Archives) și ii aplică propriile patch-uri pentru a rezolva bug-uri, a îmbunătăți performanța și a-i adăuga funcționalități specifice.

Android-ul face același lucru, ia kernel-ul vanilla și ii aplică câteva sute de patch-uri care aduc îmbunătățiri, rezolvări de bug-uri și funcționalități specifice pentru dispozitivele pe care rulează Android-ul.

Android Mainlining Project și Android Upstreaming Project s-au ocupat cu includerea acestor Androidisme in mainline. O parte din ele au fost deja integrate în mainline.

În continuare, vor fi prezentate cele mai importante dintre aceste Androidisme: Wakelocks, Low Memory Killer, Binder, Anonymous Shared Memory, Alarm, Paranoid Networking.

3.2.1 Wakelocks

În cazul desktop-urilor și laptop-urilor, în general, utilizatorul decide când sistemul va intra în sleep și când se va trezi. În contrast, kernel-ul Androidizat va intra în sleep de fiecare dată când este posibil, cât mai des posibil.

De aceea este nevoie de un mecanism care să forțeze sistemul să nu intre în sleep atunci când se fac procesări importante sau atunci când se așteaptă input-ul utilizatorului.

Acest mecanism este implementat folosind wakelock-uri și rolul lor este de a menține sistemul treaz. O aplicație care vrea să ruleze fără ca sistemul să intre în sleep, va trebui să obțină un astfel de wakelock.

În general dezvoltatorii nu au nevoie să ceară în mod explicit un wakelock, deoarece abstracțiile pe care le folosesc în aplicațiile lor vor gestiona în mod automat obiectele wakelock.

Totuși, Aplicațiile de Android pot cere wakelock-uri în mod explicit de la serviciul numit Power Manager. În schimb, driverele de dispozitiv vor trebui să apeleze direct primitivele wakelock din kernel, pentru a obține și elibera un wakelock.

În versiunea 3.5 de Linux, în 2012, un echivalent al wakelock-urilor și mecanismul corelat de early suspend au fost introduse în mainline. Înlocuitorul lui early suspend este numit autosleep iar wakelock-urile sunt înlocuite de noul flag pentru `epoll()` numite `EPOLLWAKEUP`.

3.2.2 Low Memory Killer

Kernel-ul de Linux include Out of Memory killer (OOM), care se activează dacă nu mai există memorie disponibilă.

În Android este foarte importantă tratarea situațiilor în care există puțină memorie disponibilă. De aceea s-a introdus Low Memory Killer (LLM) care va rula înainte de OOM Killer-ul default din kernel.

Scopul lui LMK este de a preveni activarea OOM killer-ului, prin omorârea proceselor care au componente care nu au fost folosite de mult timp și nu au prioritate mare. Astfel, sistemul nu va ajunge în situația în care să rămână fără memorie.

LMK a fost inclus în mainline din Linux 3.10. El se bazează pe mecanismul de ajustări OOM prin care putem avea priorități diferite pentru procese diferite.

Aceste ajustări vor permite userspace-ului să controleze o parte din politicile de omorâre a proceselor din kernel. Politicile de userspace sunt aplicate de către procesul `init` la pornirea sistemului și pot fi reajustate și aplicate la runtime prin serviciul Activity Manager.

Ajustările OOM sunt de la -17 la 15 iar o valoare mai mare înseamnă că acel proces este mai probabil să fie omorât de sistem dacă va rămâne fără memorie.

Android-ul va acorda un nivel de ajustare fiecărui tip de proces, în funcție de componentele care rulează și configurează LMK să aplice anumite limite (threshold-uri) pentru fiecare tip de proces. Aici avem câteva tipuri de procese:

- Foreground app - aplicație în foreground
- Visible app - aplicație vizibilă dar care nu este în foreground
- Secondary server - serviciu
- Hidden app - aplicație ascunsă, dar folosită de către o aplicație care rulează
- Content provider - componentă care oferă acces la date
- Empty app - aplicație care nu este activă

Procesele vor fi omorâte atunci când acele limite au fost atinse. Și astfel LMK va intra în acțiune înaintea OOM killer-ului, pentru că nu se va ajunge să se termine memoria.

3.2.3 Anonymous Shared Memory

Anonymous Shared Memory (Ashmem) este un mecanism Inter-process Communication (IPC) de memorie partajată. Este bazat pe fișiere și contoare de referință.

Echipa Android a evitat folosirea IPC-urilor de tip System V deoarece acestea pot duce la leak-uri de resurse în kernel și astfel pot permite aplicațiilor malițioase sau misbehaving să afecteze sistemul.

Ashmem este similar cu memoria partajată POSIX (Portable Operating System Interface) dar prezintă următoarele diferențe: Ashmem folosește un contor de referință pentru a distruge zonele de memorie atunci când toate procesele care au referit acele zone s-au terminat. De asemenea mecanismul va micșora zonele de memorie mapate atunci când sistemul are nevoie de mai multă memorie.

Dacă o zonă este pinned, ea nu va putea fi micșorată, de aceea trebuie să fie unpinned înainte.

Partajarea memoriei se face astfel: primul proces creează o zonă de memorie partajată folosind ashmem, apoi folosește Binder-ul pentru a partaja descriptorul de fișier asociat cu alte procese.

Multe dintre componentele system server-ului se bazează pe ashmem, prin interfața `IMemory`, nu direct (exemple: Surface Flinger și Audio Flinger).

Driver-ul pentru ashmem a fost inclus în staging tree din versiunea 3.3, dar nu a fost inclus în mainline.

3.2.4 Alarm

Driver-ul alarm se bazează pe funcționalitățile Real-Time Clock (RTC) și High Resolution Timers (HRT).

`setitimer()` este un apel de sistem care determină generarea unui semnal atunci când expiră timpul. Acest apel de sistem se bazează, printre altele și pe timer-ul ITIMER_REAL care folosește HRT-ul din kernel.

Totuși, acesta nu funcționează când sistemul este suspendat. Astfel, aplicația va primi semnalul abia când dispozitivul se va trezi.

În afară de HRT mai există și driver-ul RTC accesibil prin `/dev/rtc`, cu care se comunică prin apeluri `ioctl()`. Dacă folosim RTC, alerta va fi generată chiar dacă sistemul este suspendat deoarece dispozitivul RTC va rămâne activ chiar dacă restul sistemului este suspendat.

Driver-ul pentru Android va combina ambele mecanisme. În mod implicit, driver-ul folosește HRT pentru a genera alerte, dar atunci când sistemul este pe cale de a se suspenda, se programează RTC pentru a trezi sistemul la momentul potrivit.

Astfel, orice aplicație din spațiul utilizator poate folosi acest driver pentru a genera o alertă indiferent dacă sistemul este treaz sau suspendat în acel moment.

Driver-ul va putea fi accesat din spațiul utilizator prin `/dev/alarm` care este un dispozitiv de tip caracter. Va permite configurarea alarmelor și a ceasului prin apeluri `ioctl()`.

Multe componente cheie din AOSP se bazează pe acest driver. De exemplu, clasa SystemClock se bazează pe el pentru a obține și seta ceasul. De asemenea AlarmManager îl folosește pentru a oferi servicii de alertă aplicațiilor.

Atât driver-ul cât și AlarmManager vor folosi Wakelock-uri pentru a păstra consistența între alarme și restul sistemului. De exemplu, atunci când se generează o alarmă, aplicația va putea face operațiile necesare (va deține wakelock-ul) înainte ca sistemul să intre din nou în suspend.

Acest driver a fost inclus în mainline din versiunea 3.20.

3.2.5 Paranoid Networking

În general, pe Linux, toate procesele pot crea socketi și accesa rețeaua. În Android, din motive de securitate, trebuie să decidem care aplicații vor avea acces la rețea.

Mecanismul Paranoid Networking restricționează accesul la rețea în funcție de grupul din care face parte procesul apelant.

Pentru a putea crea socket-uri `AF_INET` și `AF_INET6`, un proces trebuie să aibă GID-ul suplimentar `AID_INET`. Dacă aplicația are permisiunea `android.permission.INTERNET` atunci va avea GID-ul suplimentar `AID_INET`.

Pentru a crea socket-uri INET raw, un proces trebuie să aibă GID-ul suplimentar `AID_NET_RAW`.

Apartenența la grupul `AID_NET_ADMIN` garantează capabilitatea `CAP_NET_ADMIN`, care permite configurarea interfețelor de rețea și a tabelelor de rutare.

Apartenența la grupul `AID_NET_BT` permite crearea socket-urilor Bluetooth (SCO, RFCOMM, L2CAP). Dacă aplicația are permisiunea `android.permission.BLUETOOTH`, atunci va avea grupul suplimentar `AID_NET_BT`.

Grupul `AID_BT_ADMIN` are abilitatea de a gestiona conexiunile Bluetooth. Dacă aplicația are permisiunea `android.permission.BLUETOOTH_ADMIN`, atunci va avea grupul suplimentar `AID_BT_ADMIN`.

Asocierile între permisiuni și GID-uri se află în fișierul `/etc/permission/platform.xml` de pe dispozitiv. În continuare este prezentat un fragment al acestui fișier, unde apar mapările între permisiuni și GID-uri.

```
<permission name="android.permission.BLUETOOTH_ADMIN" >
        <group gid="net_bt_admin" />
</permission>
<permission name="android.permission.BLUETOOTH" >
        <group gid="net_bt" />
</permission>
<permission name="android.permission.INTERNET" >
```

```
        <group gid="inet" />
</permission>
<permission name="android.permission.NET_ADMIN" >
        <group gid="net_admin" />
</permission>
```

3.3 Binder

3.3.1 Istoric

Binder-ul include un mecanism de Remote Procedure Call (RPC). A fost prima oară dezvoltat în cadrul BeOS care apoi a fost cumpărat de către Palm. Apoi a fost pus la dispoziția dezvoltatorilor prin OpenBinder.

Câțiva dintre dezvoltatorii care au lucrat la OpenBinder s-au mutat în echipa Android de la Google.

Astfel Binder-ul de Android a fost inspirat din OpenBinder (oferă aceeași funcționalitate), dar nu deriva din codul acestuia, ci a fost scris de la zero.

Astfel se poate folosi documentația de OpenBinder pentru a înțelege mecanismul general, deoarece nu prea există documentație despre Binder-ul de Android, lucru valabil pentru toate componentele interne Android-ului.

Driver-ul Binder a fost inclus în mainline din versiunea 3.19.

3.3.2 RPC

RPC este un mecanism de comunicare între procese aflate la distanță. Cele două procese pot fi pe aceeași mașină sau pe mașini fizice diferite.

Un prim principiu de funcționare RPC este următorul: ambele procese (client și server) trebuie să folosească un “contract” prestabilit între cele două entități. Acest contract poartă numele de interfață de tip stub și se află atât în client, cât și în server (Figura 3.2).

Aceasta interfață conține semnăturile funcțiilor care pot fi apelate de către client din server-ul aflat la distanță.

Interfața trebuie sa fie sincronizată atât pe client, cât și pe server. În caz contrar, spre exemplu, dacă un client adaugă o funcție în interfața sa stub de care serverul nu știe, un apel al acestei funcții se va solda cu o excepție. În schimb dacă serverul va adăuga o funcție în interfața sa stub si clientul nu știe de ea, nu se va întâmpla nimic pentru că clientul nu are cum să cheme o funcție de care nu știe.

Odată ce a fost stabilit acest contract, se mai pune problema formatului în care sunt transmise datele prin RPC pentru că clientul și serverul pot fi implementați în orice limbaj de programare (client de Python care invocă o funcție în server Java sau client de C care invocă o funcție în server Haskell). De aceea trebuie definit un format standard în care să fie transmise datele, un format agnostic de limbajul de programare în care au fost implementați serverul și clientul.

Exemple de mecanisme RPC: XML-RPC care transmite datele în format XML, JSON-RPC care transmite datele în format JSON, gRPC care folosește propriul format

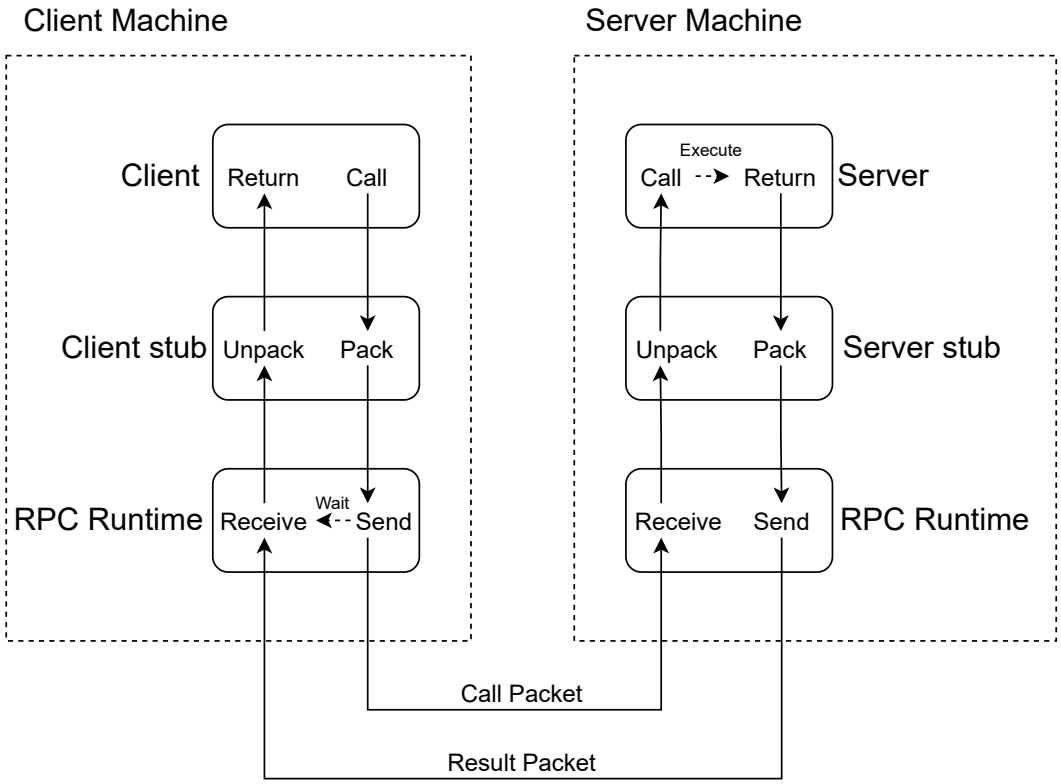

Figura 3.2: Generic RPC

standard pentru mesaje, numit protobuf.

Când un proces dorește să folosească RPC pentru a comunica cu un proces, se va folosi de interfața de tip stub. Aceasta interfață va serializa datele în formatul corespunzător mecanismului de RPC. Când serverul va primi datele prin RPC, interfața sa stub va deserializa datele (Figura 3.2).

3.3.3 Android Binder

Binder-ul extinde funcționalitatea sistemului prin invocarea obiectelor la distanță.

În loc de a crea un nou daemon pentru fiecare serviciu, se poate folosi un obiect remote implementat într-un anumit limbaj de programare (C, Java) care se află în același proces cu alte servicii sau într-un proces separat.

Este nevoie doar de o interfață și o referință la aceasta pentru a putea invoca metode remote.

Binder-ul este o componentă centrală a arhitecturii Android-ului. El este folosit pentru a comunica cu serviciile de sistem și cu serviciile aplicațiilor.

Dezvoltatorii de aplicații nu vor folosi niciodată Binder-ul direct, ci vor folosi interfețe și stub-uri generate de utilitarul `aidl`. Chiar și atunci când comunică cu serviciile de sistem, API-ul public va apela în spate stub-uri care comunică prin Binder cu serviciile.

O bună parte din Binder este implementată în kernel într-un driver. Acest driver este accesibil printr-un dispozitiv caracter `/dev/binder`. Componentele care comunică prin Binder vor putea schimba date serializate prin apeluri `ioctl()`.

In Android, atunci când un proces dorește să interacționeze cu alt proces, va iniția o tranzacție de Binder. Aceasta tranzacție nu va fi trimisă direct către driver-ul de Binder, ci va fi trimisă către o interfață client de tip stub (`IBinder`).

Această interfață va serializa datele necesare și le va transmite către driver-ul de Binder prin intermediul unui apel `ioctl()`. Acesta este un apel de sistem folosit pentru a interacționa cu dispozitive/drivere I/O, dar poate fi folosit și pentru a trece din user space în kernel space.

Driver-ul de Binder face niște verificări de securitate la nivel de kernel precum: are voie procesul cu UID-ul UID_SURSA să acceseze procesul cu UID-ul UID_DESTINATIE?

Odată ce aceste verificări au fost realizate cu succes, driver-ul va aloca o zona de memorie în spațiul de adresă al procesului destinație în care să fie scrise datele. După ce datele au fost scrise, procesul destinație va oferi un răspuns către procesul sursă.

Un aspect important de menționat este ca tranzacția prin Binder este blocantă: sursa va fi deblocată abia atunci când va primi răspunsul de la destinație.

3.4 Framework-ul Android

Framework-ul Android rulează deasupra userspace-ului nativ și include o multitudine de componente: pachetele `android.*`, serviciile de sistem și runtime-ul Android.

Din punct de vedere al surselor, codul ce include framework-ul se află în directorul `frameworks/` din AOSP. Framework-ul include câteva componente cheie, de bază: Service Manager, Zygote și Dalvik/ART.

3.4.1 Serviciile de sistem

Serviciile de sistem din Android formează ceea ce am putea numi un sistem de operare orientat pe obiecte deasupra kernel-ului Linux.

Principala componentă este System Server care rulează în procesul cu același nume: `system_server`, și conține un număr mare de servicii scrise în Java și 2 servicii scrise în C/C++.

Aici avem implementate în Java: PowerManager, ActivityManager, PackageManager, LocationManager, etc. Cele implementate în C/C++ sunt SurfaceFlinger și SensorService.

Un alt set de servicii este inclus în Media Service care rulează în procesul cu același nume: `mediaservice`. Aceste servicii sunt scrise în C/C++ și se ocupă de audio, video și camera. De exemplu: AudioFlinger, MediaPlayerService, CameraService.

3.4.2 Dalvik

Dalvik este runtime-ul implicit folosit în Android înainte de versiunea 5.0. Este o mașină virtuală de Java, optimizată special pentru dispozitivele mobile, și prin urmare, are un footprint de memorie mult mai scăzut.

Lucrează cu fișiere `.dex`, care sunt cu 50% mai mici decât fișierele `.jar` care conțin aceleași clase.

Spre deosebire de mașina virtuală Java originală care este bazată pe stivă, Dalvik are o arhitectură bazată pe registre. Astfel, va lucra cu un număr finit de registre care stochează o valoare întreagă pozitivă.

Dalvik nu poate rula bytecode Java original. Dar acesta poate fi convertit în bytecode Dalvik folosind comanda `dx` care construiește fișierul `.dex`.

Bytecode-ul Java lucrează cu instrucțiuni pe 8 biți specifice pentru stivă. Asta înseamnă că variabilele locale trebuie copiate pe stivă înainte de a fi folosite. În schimb Dalvik lucrează cu instrucțiuni pe 16 biți și accesează direct variabilele locale. Rezultă mai puține instrucțiuni și o viteză de rulare mai mare.

Începând cu Android 2.2, Dalvik include compilatorul Just-in-Time (JIT) pentru ARM, x86 și MIPS. Acesta traduce secțiuni de bytecode Dalvik (care sunt rulate cel mai frecvent) în instrucțiuni mașină care vor rula nativ pe CPU-ul dispozitivului în loc să fie interpretate instrucțiune cu instrucțiune de către mașina virtuală. Aceste secțiuni se numesc trace-uri. Restul bytecode-ului va fi interpretat de mașina virtuală Dalvik.

Această conversie este făcută o singură dată și apoi stocată pentru rulările viitoare. Astfel, aplicațiile se vor încărca un mai greu prima oară, dar apoi vor rula mult mai repede. Execuția nativă a trace-urilor aduce îmbunătățiri de performanță.

3.4.3 ART

Android Runtime (ART) este disponibil din Android 4.4, dar a devenit runtime-ul implicit din Android 5.0.

ART vine să înlocuiască Dalvik, aducând o serie de îmbunătățiri. La fel ca și Dalvik, lucrează cu fișiere .dex, din motive de compatibilitate.

ART oferă compilare Ahead-of-Time (AoT), ceea ce poate îmbunătăți performanța aplicațiilor. La instalare, ART folosește utilitarul `dex2oat` pentru a transforma fișierul dex într-un executabil pentru dispozitivul respectiv.

Astfel, întreaga aplicație va rula în mod nativ direct pe procesorul dispozitivului. Prin urmare, AOT înlocuiește compilarea JIT și interpretarea Dalvik. În plus, face o verificare mai strictă a aplicațiilor la instalare.

Dezavantajele AOT sunt următoarele: executabilele ocupă spațiu de stocare adițional, și instalarea aplicațiilor durează mai mult.

ART oferă un proces de garbage collection mult mai eficient. Oferă suport pentru un sampling profiler dedicat care generează informații precise despre execuția aplicațiilor fără a afecta performanța. Oferă un număr de opțiuni noi de debugging, mai ales legate de monitorizare și garbage collection. În plus, ART va genera mai multe detalii și informații legate de context atunci când are loc o excepție pe parcursul rulării.

3.4.4 Zygote

Zygote este un daemon folosit pentru a porni aplicații și este activ doar în momentul în care trebuie pornită o aplicație. Este practic părintele tuturor aplicațiilor în Android.

La început, Zygote va pre-încărca toate clasele Java și resursele de care aplicațiile ar putea avea nevoie, în RAM, pentru a realiza o pornire mai rapidă a aplicațiilor.

Apoi Zygote va asculta pentru a primi conexiuni pe socket-ul `/dev/socket/zygote`. Pe acest socket primește cereri de pornire a aplicațiilor. Atunci când primește o astfel de cerere, își va face fork și va porni aplicația în noul proces.

Avantajul de a avea toate aplicațiile fork-uite din Zygote este faptul că avem toate clasele și resursele încărcate în memorie și aplicațiile își pot începe direct execuția.

Acest lucru se întâmplă datorită mecanismului Copy on Write (CoW) din kernel-ul Linux. Atunci când se crează un nou proces cu fork, el este o copie a părintelui și va avea mapate paginile de memorie fără a fi necesară copierea acestora. Doar atunci când se scrie într-o pagină, ea va fi întâi copiată.

Clasele și resursele folosite de aplicații nu sunt modificate niciodată. Prin urmare, toate aplicațiile vor avea acces la cele încărcate inițial de Zygote fără a fi necesară o copie. Deci va fi folosită o singură variantă a claselor și resurselor din RAM.

Există un singur proces pe care Zygote îl pornește în mod explicit (fără nici o cerere) - `system_server` - este primul proces pornit de către Zygote.

Dacă dăm comanda `ps`, se poate observa că PID-ul lui Zygote este PPID-ul tuturor aplicațiilor, dar și al procesului `system_server`.

În continuare se poate observa rezultatul rulării comenzii `ps`. Procesul `init` are PID-ul 1, el pornește procesul `zygote` care are PID-ul 3279 în acest caz și care mai departe crează celelalte aplicații și servicii (unele de sistem - care au userul system pe prima coloană). Se poate observa și procesul `system_server` care este creat explicit de către Zygote.

```
hero2lte:/ # ps
USER      PID   PPID  NAME
root      1     0     /init
root      3279  1     zygote
system    3689  3279  system_server
system    5063  3279  com.samsung.android.radiobasedlocation
u0_a10    5090  3279  com.samsung.android.providers.context
advmodem  5117  3279  com.samsung.android.networkdiagnostic
u0_a99    5271  3279  com.samsung.android.widgetapp.briefing
u0_a45    5287  3279  com.samsung.android.service.peoplestripe
u0_a4     5313  3279  com.samsung.android.app.aodservice
u0_a128   5922  3279  com.samsung.android.sdk.handwriting
u0_a6     6178  3279  com.samsung.android.contacts
system    6927  3279  com.samsung.ucs.agent.boot
u0_a108   6939  3279  com.samsung.ucs.agent.ese
u0_a37    12229 3279  com.samsung.klmsagent
system    24833 3279  com.samsung.android.lool
system    25118 3279  com.samsung.android.securitylogagent
system    25354 3279  com.samsung.android.sm.provider
```

3.4.5 Xposed

Aducem în discuție framework-ul Xposed care este un framework open source. Acesta permite dezvoltatorului să introducă pre/post hooks în funcțiile oferite de middleware-ul Android, funcții care, de obicei, sunt private și nu sunt expuse către dezvoltatorii de aplicații. Pentru a putea fi instalat acest framework, este necesar ca telefonul sa fie rooted.

Xposed reușește să introducă pre/post hooks în funcțiile din middleware în felul următor: extinde fișierul `/system/bin/app_process` cu un fișier JAR custom care permite hooking-ul de funcții. Acest fișier `/system/bin/app_process` este folosit de către procesul Zygote pentru a încărca toate classpath-urile de Android si de Java, ceea ce înseamnă ca va încărca și JAR-ul custom de Xposed.

Xposed oferă posibilitatea de a defini preHook pentru o anumită funcție ceea ce înseamnă că dezvoltatorul are posibilitatea de a modifica anumite argumente care să fie transmise către funcție.

Xposed oferă posibilitatea de a defini postHook pentru o anumită funcție ceea ce înseamnă că dezvoltatorul are posibilitatea de a modifica rezultatul oferit de acea funcție. Spre exemplu, putem modifica rezultatul oferit de către metoda din middleware `getLocation()` sau putem modifica rezultatul returnat de către funcția care întoarce lista de contacte.

Ce face de fapt Xposed în spate cu funcțiile de hooking și funcția originală poate fi sumarizat astfel:

```
envelope_function():
        preHook();
        original_function();
        postHook();}
```

Peste Xposed sunt construite o mulțime de aplicații, printre care și XPrivacy, o aplicație cu ajutorul căreia un utilizator poate sa configureze în mod granular rezultatul oferit atunci când sunt accesate anumite resurse precum lista de contacte sau locația.

Practic, XPrivacy returnează informații false pentru anumite apeluri și reușește acest lucru folosindu-se de apelurile de preHook si postHook oferite de Xposed.

3.4.6 Logd

Din Android 5.0, este inclus un nou mecanism de logging bazat pe daemon-ul `logd`. Acesta are comportamentul unui logger centralizat de user-space.

Logd adresează dezavantajele folosirii bufferelor circulare, folosite de mecanismul Logger anterior (dimensiunea mică și nevoia de a fi rezidente în memorie).

Logd poate fi integrat cu SELinux, prin înregistrarea lui ca `auditd`, pentru a primi mesajele SELinux din kernel prin socket-ul netlink.

Logd folosește 4 socket-uri:

- `/dev/socket/logd`, care este socketul ce oferă interfața de control
- `/dev/socket/logdw`, care un socket write-only folosit doar pentru scrierea logurilor
- `/dev/socket/logdr`, care este un socket read-only, folosit doar pentru citirea logurilor
- un socket netlink fără nume folosit pentru integrarea logd cu SELinux

Socket-urile nu sunt accesate în mod direct, ci doar prin biblioteca `liblog`. Aplicațiile folosesc clasa `Log` (sau `EventLog`) pentru scrierea logurilor, care în schimb apelează

biblioteca nativă `liblog` prin JNI, care deschide socket-ul `/dev/socket/logdw` pentru a scrie mesajul.

Pentru citirea logurilor, `logcat` va apela biblioteca `liblog` care se va conecta la `/dev/socket/logdr` (prin `LogReader`).

3.5 Manageri

3.5.1 Service Manager

O componentă importantă în framework este Service Manager. Acesta este responsabil cu identificarea serviciilor de sistem (operația numită lookup). Este un fel de Pagini Aurii a tuturor serviciilor de sistem.

Un serviciu care nu s-a înregistrat la Service Manager nu poate fi accesat. Orice serviciu care se vrea a fi accesibil, trebuie să se înregistreze întâi la Service Manager.

Acest manager este pornit de către procesul `init` înaintea oricărui alt serviciu. Când pornește, accesează `/dev/binder` și folosește un apel `ioctl()` pentru a se face pe sine Context Manager-ul Binder-ului.

De ce face acest lucru? Pentru a deveni obiectul magic (Binder ID 0). Orice proces care va comunica cu Binder ID 0 (obiectul magic, sau magic Binder) va comunica de fapt cu Service Manager prin Binder.

Fiecare serviciu de sistem se va înregistra la Service Manager (tot printr-un apel prin Binder). Manager-ul va păstra o listă de servicii disponibile.

Atunci când o aplicație vrea să comunice cu un serviciu de sistem, va cere de la Service Manager, prin `getSystemService()`, un handle către acel serviciu apoi va apela metodele serviciului folosind acel handle (tot prin Binder, deoarece toate operațiile se fac prin Binder).

Acest mecanism este valabil doar pentru serviciile de sistem, el nu va fi folosit atunci când se accesează serviciul unei aplicații (apelul va trece direct prin Binder fără a fi căutat în Service Manager).

De asemenea, managerul este folosit de către o serie de utilitare, cum este `dumpsys`, care face dump la statusul unui serviciu sau a tuturor serviciilor de sistem. Întâi va cere o listă cu toate serviciile, apoi va cere handle-ul pentru fiecare și va apela funcția `dump` din serviciul respectiv.

3.5.2 Activity Manager

În continuare vom aborda trei manageri importanți: Activity Manager, Package Manager și Power Manager.

Activity Manager este unul dintre cele mai importante servicii din System Server și este responsabil cu gestiunea ciclului de viață al activităților.

Acest serviciu va porni activități și servicii în aplicații, va obține content provideri și va face broadcast la intenturi.

Dialog-ul Application Not Responing (ANR) este generat tot de Activity Manager. De asemenea, este implicat în multe task-uri adiacente - verifică permisiuni, ajută la calculul ajustărilor OOM pentru Low Memory Killer, și face task management.

Activity Manager este cel care pornește Launcher-ul cu un intent de tipul `CATEGORY_HOME`.

Să vedem ce se întâmplă atunci când o aplicație este pornită din Launcher: Se apelează callback-ul `onClick()` din Launcher. În acest callback se va apela prin Binder metoda `startActivity()` din Activity Manager. Serviciul va apela metoda `startViaZygote()` care va deschide o conexiune prin socket cu Zygote și îi va cere să pornească un nou proces cu activitatea respectivă.

În linia de comandă avem utilitarul `am` pentru a da comenzi către Activity Manager. Putem porni o activitate, un serviciu, trimite un intent, porni profilling-ul sau putem face debugging.

3.5.3 Package Manager

Package Manager este acel serviciu care gestionează fișierele de tip apk. Mai exact, el oferă posibilitatea instalării, dezinstalării și actualizării de pachete.

Package Manager lucrează cu câteva fișiere din `/data/system`, cele mai importante fiind: `packages.xml` care conține toate permisiunile și informații despre pachetele instalate și `packages.list` care conține toate pachetele instalate, UID-ul lor și directorul de date.

Acest manager rulează în cadrul `system_server` dar se folosește de `installd` pentru a efectua majoritatea operațiilor (deoarece `installd` are permisiuni de root).

Tot Package Manager ajută la rezolvarea intent-urilor, mai exact la identificarea componentei care trebuie să primească intent-ul. El va primi o cerere de rezolvare și va folosi informația din fișierul Manifest pentru a identifica componenta cea mai potrivită.

În linia de comandă, avem utilitarul `pm` pentru a da comenzi către Package Manager. Vom putea lista pachetele instalate, vizualiza permisiunile cerute de o aplicație, instala, dezinstala pachete, afișa folderul aplicației, dezactiva un pachet, și altele.

3.5.4 Power Manager

Power Manager este responsabil cu controlul consumului de putere al dispozitivului.

Aici este locul din AOSP unde sunt gestionate wakelock-urile. Serviciul include clasa `WakeLock` cu metodele aferente `aquire` și `release`. Aplicațiile vor cere Wakelock-uri de la Power Manager.

Gestiunea consumului de putere este implementată în kernel, dar toate apelurile trebuie să treacă mai întâi prin Package Manager.

Printre operațiile pe care le poate face acest serviciu este forțarea dispozitivului să intre în sleep și configurarea luminozității ecranului.

3.6 Bibliografie

- Karim Yaghmour. 2013. Embedded Android: Porting, Extending, and Customizing. O'Reilly Media, Inc. (Chapter 2).
- Joshua J. Drake, Zach Lanier, Collin Mulliner, Pau Oliva Fora, Stephen A. Ridley, and Georg Wicherski. 2014. Android Hacker's Handbook. Wiley Publishing (Chapter 2).
- https://source.android.com/devices/tech/dalvik/ (Accesat: Mai 2021)

Capitolul 4

Conectivitate

4.1 Multithreading

4.1.1 Thread-ul principal de UI al aplicației

În aplicațiile Android, thread-ul de UI (User Interface) este cel principal. Acest thread este foarte important deoarece este responsabil cu afișarea, desenarea și actualizarea elementelor de UI, și cu gestionarea/livrarea evenimentelor de UI (interacțiunea utilizatorului cu aplicația).

Dacă alt thread încearcă să actualizeze elementele de UI, va primi `CalledFromWrongThreadException`.

În plus, serviciile și Broadcast Receivers rulează, în mod implicit, pe thread-ul de UI.

Pentru a efectua operații intensive computațional sau blocante (de exemplu operații cu rețeaua sau baza de date), thread-ul principal va fi blocat și evenimentele de UI nu vor fi livrate. Astfel utilizatorul nu va putea să interacționeze în mod normal cu aplicația și va primi mesajul Application Not Responding (ANR) după câteva secunde.

Există două reguli simple legate de thread-ul principal de UI:

- Thread-ul de UI nu trebuie blocat cu operații intensive sau blocante
- API-ul de UI trebuie apelat doar de pe thread-ul principal de UI

Prin urmare, dacă vrem să efectuăm operații intensive computațional sau blocante, trebuie create noi thread-uri (“worker” / “background” threads).

Pentru a face asta, avem mai multe opțiuni, de exemplu de a crea o instanță `Thread` și a apela `start()`, sau de a implementa interfața `Runnable`. Totuși, atunci când folosim clasele `Thread` sau `Runnable`, trebuie să trimitem datele înapoi la thread-ul de UI în mod manual folosind un `Handler`.

Clasele Thread și Runnable sunt clase simple, dar sunt baza unor clase mai puternice din Android, cum ar fi `AsyncTask`, `IntentService` și `HandlerThread`. De asemenea, sunt baza pentru `ThreadPoolExecutor`, care gestionează thread-urile și cozile de task-uri.

4.1.2 AsyncTask

`AsyncTask` este o clasă care permite efectuarea operațiilor asincrone pe un thread separat. Practic, execută operații folosind un worker thread și apoi publică rezultatele înapoi pe thread-ul de UI. Deci nu avem nevoie să gestionăm thread-uri sau să folosim handlere pentru a trimite datele înapoi la thread-ul de UI.

Această clasă include o metodă care rulează pe worker thread și mai multe metode care rulează pe UI thread, care pot fi folosite pentru a publica rezultatele.

Metoda `doInBackground()` din clasa AsyncTask rulează pe worker thread, în timp ce metodele `onPreExecute()`, `onPostExecute()`, și `onProgressUpdate()` rulează pe thread-ul de UI.

`doInBackground()` întoarce o valoare care este trimisă ca argument metodei `onPostExecute()`, după ce task-ul a fost finalizat. `onProgressUpdate()` va fi executat atunci când apelăm `publishProgress()` din `doInBackground()`, în orice moment, în timp ce rulează task-ul.

Task-ul este pornit atunci când apelăm `execute()`. Task-ul poate fi anulat în orice moment, din orice thread, prin folosirea metodei `cancel()`.

În continuare este prezentat un exemplu de AsyncTask, folosit pentru download-area mai multor fișiere.

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
     protected Long doInBackground(URL... urls) {
         int count = urls.length;
         long totalSize = 0;
         for (int i = 0; i < count; i++) {
             totalSize += Downloader.downloadFile(urls[i]);
             publishProgress((int) ((i / (float) count) * 100));
             if (isCancelled()) break;
         }
         return totalSize;
     }
     protected void onProgressUpdate(Integer... progress) {
         setProgressPercent(progress[0]);
     }
     protected void onPostExecute(Long result) {
         showDialog("Downloaded␣" + result + "␣bytes");
     }
}
```

Clasa din exemplu include: metoda `doInBackground()` care execută task-ul pe worker thread, metoda `onPostExecute()`, care primește rezultatul returnat de `doInBackground()` și rulează pe UI thread. Exemplul afișează un dialog cu rezultatul. Metoda `onProgressUpdate()` care rulează pe UI thread și este executată când este apelată metoda `publishProgress()` de pe worker thread.

În exemplul următor este prezentată modalitatea prin care se pornește AsyncTask-ul implementat mai sus.

```
new DownloadFilesTask().execute(url1, url2, url3);
```

Clasa AsyncTask a fost marcată ca deprecated începând cu Android 11 (API 30).

4.2 Accesarea conținutului online

4.2.1 Conectarea la rețea

În această secțiune se va prezenta modul de a dezvolta aplicații care se conectează la rețea. În primul rând, ca să efectuăm operații de rețea, avem nevoie să cerem următoarele permisiuni: `ACCESS_NETWORK_STATE` - care permite aplicației să verifice starea rețelei; `INTERNET` - care permite aplicației să acceseze resurse din Internet.

În continuare este prezentat un exemplu de specificare a acestor permisiuni în fișierul Manifest. De asemenea, trebuie cerute în mod explicit la runtime.

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

Pentru a nu afecta experiența utilizatorului, operațiile de rețea trebuie făcute pe un thread separat de cel de UI. `AsyncTask-ul` este un mod simplu de a efectua aceste operații pe un worker thread.

Înainte de a realiza un transfer prin rețea, trebuie verificată starea conexiunii la rețea. Pentru a face asta, trebuie obținută o referință către `ConnectivityManager`, și apoi apelată metoda `getActiveNetworkInfo()`.

Această metodă întoarce un obiect `NetworkInfo`, care poate fi folosit pentru apelarea metodei `isConnected()` pentru verificarea conexiunii la rețea.

```
public void getRemoteData(View view) {
    [...]
    ConnectivityManager connMgr = (ConnectivityManager)
                        getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
    if (networkInfo != null && networkInfo.isConnected()) {
        // dispozitivul este conectat la rețea
        // se pot obține datele
    } else {
        // se afișează o eroare
    }
    [...]
}
```

4.2.2 Accesarea conținutului online

Putem accesa conținutul online prin mai multe metode. O opțiune este folosirea socket-ilor Java, care pot fi ușor de folosit dacă se implementează un protocol simplu. Trebuie gestionate toate mesajele în mod manual, ceea ce poate deveni complicat pentru un protocol complex.

O altă opțiune este folosirea clasei `URLConnection`, care poate gestiona în mod automat mesajele de nivel aplicație pentru diferite protocoale, cum ar fi FTP, HTTP și HTTPS (`file://, ftp://, http://, https://`).

Această clasă este folosită pentru conectarea la un URL specific, cu scopul de a citi sau a scrie date. În primul rând trebuie creat un nou obiect URL (de exemplu: `new`

URL("ftp://example.com")), și apoi trebuie obținut obiectul URLConnection prin folosirea metodei openConnection() a obiectului URL.

4.2.3 Conexiune HTTP

Atunci când lucrăm cu HTTP sau HTTPS, putem folosi anumite clase mai specifice. Cele recomandate pentru gestionarea mesajelor HTTP și HTTPS sunt HttpURLConnection și HttpsURLConnection.

Se pot efectua operații GET pentru a downloada datele. De asemenea, oferă suport transparent pentru IPv6. Dacă stația are atât adresă IPv4 cât și adresă IPv6, atunci va încerca să se conecteze la fiecare adresă, până când o conexiune este realizată.

În primul rând trebuie creat un obiect URL, apoi apelat openConnection() pe acel URL, și făcut cast la HttpURLConnection. Apoi putem trimite o cerere printr-un OutputStream (getOutputStream()) și va primi datele prin InputStream (getInputStream()). În mod implicit HttpURLConnection folosește metoda GET. Pentru a folosi metoda POST, trebuie apelat setDoOutput(true).

Pentru alte metode folosiți setRequestMethod(String). De asemenea, se pot gestiona cookie-urile prin folosirea CookieManager și HttpCookie.

În continuare este prezentat un exemplu de folosire a HttpURLConnection. Întâi, un obiect URL este creat, apoi HttpURLConnection este obținut, și de asemenea InputStream pentru acea conexiune.

```
URL url = new URL("http://www.google.ro/");
HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection
    ();
try {
        InputStream in = new BufferedInputStream(urlConnection.
           getInputStream());
        readStream(in);
} finally {
        urlConnection.disconnect();
}
```

4.3 Obținerea locației

Pe Android, informațiile despre locație sunt obținute din diferite surse: prin GPS, prin rețeaua mobilă, prin rețeaua WiFi sau prin alte aplicații.

În comparație cu informațiile primite de la WiFi/mobile, GPS este mai precis, dar funcționează cel mai bine în afara clădirilor, consumă mai multă putere și returnează un răspuns mai lent.

Locația obținută de la rețeaua WiFi sau mobilă este mai puțin precisă, dar funcționează atât în interiorul cât și în exteriorul clădirilor, consumă mai puțină putere, și oferă un răspuns mai rapid. Într-o aplicație se pot folosi ambele metodele sau doar una dintre ele.

4.3.1 Permisiuni de obținere a locației

Pentru a folosi GPS_PROVIDER, trebuie declarată și cerută permisiunea `ACCESS_FINE_LOCATION`. În plus, trebuie declarată caracteristica hardware `android.hardware.location.gps`, pentru GPS_PROVIDER.

Pentru a folosi NETWORK_PROVIDER (pentru WiFi/mobile), trebuie declarată și cerută permisiunea `ACCESS_COARSE_LOCATION`. De asemenea, trebuie declarată caracteristica `android.hardware.location.network`, pentru folosirea NETWORK_PROVIDER.

Atunci când folosim ambele metode, trebuie declarată și cerută doar permisiunea `ACCESS_FINE_LOCATION`, deoarece permite ambii provideri. Totuși, trebuie specificate ambele caracteristici hardware.

Acestea sunt exemple de declarare a permisiunilor și caracteristicilor hardware pentru cei doi provideri în fișierul Manifest.

```
<manifest ... >
    <uses-permission android:name="android.permission.
       ACCESS_FINE_LOCATION" />
    ...
    <uses-feature android:name="android.hardware.location.gps" />
    ...
</manifest>
```

```
<manifest ... >
    <uses-permission android:name="android.permission.
       ACCESS_COARSE_LOCATION" />
    ...
    <uses-feature android:name="android.hardware.location.network" />
    ...
</manifest>
```

4.3.2 Primirea actualizărilor de locație

Într-o aplicație Android, locația poate fi obținută de la `LocationManager` prin folosirea unor callback-uri pentru primirea actualizărilor de locație.

În primul rând, trebuie obținută o referință la `LocationManager`, care este un serviciu de sistem. Actualizarea locației este primită folosind `LocationListener`.

Deci trebuie implementat un `LocationListener` care include callback-uri care vor fi apelate de către `LocationManager`. În final, trebuie înregistrat listener-ul la `LocationManager` pentru a primi actualizările.

În continuare este prezentat un exemplu de implementare.

```
LocationManager locationManager =
                                (LocationManager) this.getSystemService(
                                Context.LOCATION_SERVICE);
[..]
LocationListener locationListener = new LocationListener() {
    public void onLocationChanged(Location location) {
      makeUseOfNewLocation(location);
    }
```

```
    public void onStatusChanged(String provider,
                                        int status, Bundle extras) {}

    public void onProviderEnabled(String provider) {}

    public void onProviderDisabled(String provider) {}
  };
[..]
locationManager.requestLocationUpdates(
                                        LocationManager.NETWORK_PROVIDER,
                                        0, 0, locationListener);
```

Putem vedea că `LocationManager` este obținut folosind `getSystemService()`. În continuare este o implementare de `LocationListener`, care include callback-ul `onLocationChanged()` care va fi apelat atunci când `LocationManager` determină o nouă locație.

În final, se observă că listener-ul este înregistrat la serviciu prin folosirea metodei `requestLocationUpdates()`, și specificarea tipului de provider, aici `NETWORK_PROVIDER`.

În exemplul următor sunt prezentate câteva operații adiționale care pot fi făcute cu `LocationManager`. În primul rând, putem specifica provider-ul ales: Network sau Gps. Atunci când se apelează `requestLocationUpdates()` pe acel provider, va începe să asculte pentru a primi actualizări de locație.

```
String locationProvider = LocationManager.NETWORK_PROVIDER;
// Sau pentru folosirea provider-ului GPS:
// String locationProvider = LocationManager.GPS_PROVIDER;

locationManager.requestLocationUpdates(locationProvider, 0, 0,
    locationListener);
```

De obicei durează o perioadă de timp până când este primită o nouă actualizare. Pentru a obține un răspuns mai rapid, se poate cere ultima locație cunoscută (o locație cache-uită). În timp ce se așteaptă o nouă actualizare, se poate apela metoda `getLastKnownLocation()` a clasei `LocationManager`.

```
String locationProvider = LocationManager.NETWORK_PROVIDER;

Location lastKnownLocation = locationManager.getLastKnownLocation(
    locationProvider);
```

În final, se poate opri ascultarea actualizărilor:

```
locationManager.removeUpdates(locationListener);
```

4.4 WiFi Manager

`WifiManager` este un serviciu de sistem pentru gestiunea conexiunilor WiFi. Este folosit pentru a monitoriza, configura și gestiona conexiunile WiFi. De asemenea se poate scana pentru determinarea rețelelor WiFi disponibile.

Pentru a gestiona conexiunea WiFi este nevoie să fie specificate în Manifest și să fie cerute de la utilizator următoarele permisiuni: `ACCESS_WIFI_STATE`, și `CHANGE_WIFI_STATE`.

4.4.1 Activarea și dezactivarea WiFi

În această secțiune vor fi prezentate câteva operații ce pot fi efectuate folosind `WifiManager`. În primul rând, se poate activa și dezactiva WiFi. Pentru asta este nevoie să fie obținută o referință către serviciul `WifiManager`. Apoi se folosește acea referință pentru a apela `setWiFiEnabled` cu parametrii `true` sau `false`.

4.4.2 Scanarea rețelelor WiFi

Pentru a determina rețelele WiFi disponibile, trebuie obținută o referință către serviciul `WifiManager`. Trebuie implementat un Broadcast Receiver care va fi apelat atunci când rezultatele scanării sunt disponibile. În receiver, se vor cere în mod explicit rezultatele scanării de la WifiManager.

Acest receiver trebuie înregistrat cu acțiunea `WifiManager.SCAN_RESULTS_AVAILABLE_ACTION`. În final trebuie pornită scanarea. Când este finalizată, se va apela receiver-ul și se vor citi rezultatele.

În continuare, este prezentat un exemplu de implementare pentru scanarea rețelelor WiFi:

```
WifiManager wifiManager = (WifiManager)
                        this.getSystemService(Context.WIFI_SERVICE);
[..]
class WifiScanReceiver extends BroadcastReceiver {
   public void onReceive(Context c, Intent intent) {
         List<ScanResult> wifiScanList = wifiManager.getScanResults();
         String data = wifiScanList.get(0).toString();
   }
}
[..]
WifiScanReceiver wifiReceiver = new WifiScanReceiver();
registerReceiver(wifiReceiver, new IntentFilter(
                        WifiManager.SCAN_RESULTS_AVAILABLE_ACTION));
wifiManager.startScan();
```

4.4.3 Informații despre conexiunea curentă

Se pot obține diverse informații despre conexiunea activă curentă de WiFi. De exemplu, dacă se obține un obiect `WifiInfo` de la `WifiManager`, se poate determina SSID-ul, frecvența, viteza legăturii, și altele.

Dacă obținem un obiect `DhcpInfo` de la `WifiManager`, se poate determina adresa IP, masca, gateway, servere DNS, etc.

```
WifiManager wifiManager = (WifiManager)
                this.getSystemService(Context.WIFI_SERVICE);

WifiInfo wifiInfo = wifiManager.getConnectionInfo();
```

```
Log.v(LOG_TAG, "SSID:␣" + wifiInfo.getSSID() + ",␣"
                "Frequency:␣" + wifiInfo.getFrequency() + ",␣"
                "Link␣sped:␣" + wifiInfo.getLinkSpeed());


DhcpInfo dhcpInfo = wifiManager.getDhcpInfo();
Log.v(LOG_TAG, "DHCP␣Info:␣"+dhcpInfo.toString());
```

4.5 Bluetooth

Framework-ul Android oferă un API pentru controlul adaptorului de Bluetooth (BT):

- Se poate activa sau dezactiva BT
- Se poate face dispozitivul detectabil prin BT (detectable)
- Se pot determina alte dispozitive detectabile din jur
- Se pot asocia două dispozitivele (pair)
- Se pot trimite date la alte dispozitive
- Se pot primi date de la alte dispozitive
- Se pot gestiona conexiuni multiple

4.5.1 Permisiuni Bluetooth

Pentru a folosi BT într-o aplicație, trebuie declarată în Manifest și cerută de la utilizator permisiunea `android.permission.BLUETOOTH`. Aceasta este necesară pentru operațiile de bază, cum ar fi conectarea la dispozitivele asociate, trimiterea la alt dispozitiv, primirea datelor de la alt dispozitiv.

Permisiunea `android.permission.BLUETOOTH_ADMIN` este necesară pentru schimbarea setărilor BT (de exemplu: activarea/dezactivarea adaptorului, setarea să fie detectabil), inițierea descoperirii de dispozitive, asocierea cu alte dispozitive (cu aprobarea utilizatorului). Dacă se folosește această permisiune trebuie neapărat cerută și permisiunea `android.permission.BLUETOOTH`.

Permisiunea `android.permission.BLUETOOTH_PRIVILEDGED` poate fi cerută atunci când este necesară asocierea cu alt dispozitiv fără aprobarea utilizatorului. Totuși această permisiune nu este disponibilă pentru aplicațiile third-party.

4.5.2 Bluetooth API

Clasa `BluetoothAdapter` reprezintă adaptorul BT. Pentru a obține o instanță a acestei clase, trebuie apelată metoda statică `getDefaultAdapter()`. Dacă această metodă întoarce `null`, atunci dispozitivul nu are suport pentru BT.

Clasa poate fi folosită pentru efectuarea oricărei operații cu BT: descoperirea dispozitivelor, listarea dispozitivelor asociate, obținerea unei instanțe de BluetoothDevice (pe baza unei adrese MAC cunoscute).

După obținerea unui obiect adaptor, trebuie verificat dacă BT este activat. Aceasta se face folosind metoda `isEnabled()`. Dacă nu este activat se poate activa prin trimiterea unui Intent.

De asemenea, se poate folosi obiectul adaptor pentru a obține un `BluetoothServerSocket` pentru a asculta cererile primite.

Clasa `BluetoothDevice` reprezintă dispozitivul remote. Metoda `getBondedDevices()` a clasei `BluetoothAdapter` întoarce un set de obiecte `BluetoothDevice`, care reprezintă o listă de dispozitive asociate.

Se poate folosi obiectul `BluetoothDevice` pentru a obține informații despre dispozitiv (nume, adresă, clasă și statusul de asociere).

Pentru inițierea unei conexiuni cu un dispozitiv remote (din perspectiva unui client), se obține un `BluetoothSocket` de la `BluetoothDevice` prin apelarea metodei `createRfcommSocketToServiceRecord(UUID)`.

În continuare este prezentat un exemplu de implementare care include: obținerea adaptorului, activarea BT dacă nu este deja activ (folosind un Intent), și listarea dispozitivelor asociate (nume și adresa MAC).

```
BluetoothAdapter mBluetoothAdapter =
                                BluetoothAdapter.getDefaultAdapter();
if (mBluetoothAdapter == null) {
    // Dispozitivul nu are suport pentru BT
}
[..]
if (!mBluetoothAdapter.isEnabled()) {
    Intent enableBtIntent = new Intent(
                BluetoothAdapter.ACTION_REQUEST_ENABLE);
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
}
[..]
Set<BluetoothDevice> pairedDevices =
                                mBluetoothAdapter.getBondedDevices();
if (pairedDevices.size() > 0) {
    for (BluetoothDevice device : pairedDevices) {
        String deviceName = device.getName();
        String deviceHardwareAddress = device.getAddress();
    }
}
```

Un obiect `BluetoothSocket` poate fi folosit pentru inițierea unei conexiuni cu un dispozitiv remote. Pentru aceasta este nevoie să se apeleze metoda `connect()`.

Este similar cu un socket TCP și va fi folosit pentru trimiterea și primirea datelor prin folosirea `InputStream` și `OutputStream`. Pentru obținerea lor trebuie apelate `getInputStream()` și `getOutputStream()`.

Un `BluetoothServerSocket` poate fi obținut de la `BluetoothAdapter` și poate fi folosit pentru ascultarea conexiunilor primite (ca un socket TCP de tip server).

Pentru a aștepta conexiuni, trebuie apelată metoda `accept()`. Apelul se va bloca până când o nouă conexiune este acceptată și dacă se întoarce cu succes va returna un obiect `BluetoothSocket`.

4.5.3 Bluetooth Low Energy API

Bluetooth Low Energy (BLE) a fost proiectat special ca să consume mai puțină energie decât BT standard. Pentru a folosi BLE, în fișierul Manifest (pe lângă permisiunile menționate deja), trebuie specificată caracteristica hardware `android.hardware.bluetooth_le`.

De asemenea, la runtime, trebuie verificat dacă BLE este suportat de dispozitiv folosind metoda `hasSystemFeature()`.

Listarea dispozitivelor BLE se face folosind metoda `startLeScan` a lui `BluetoothAdapter`. Trebuie implementat `BluetoothAdapter.LeScanCallback` pentru a primi răspunsul scanării. Mai exact trebuie suprascrisă metoda `onLeScan()` a lui `LeScanCallback` pentru a primi rezultatul scanării.

Rezultatul scanării include RSSI-ul care este folosit pentru a determina distanța aproximativă până la celălalt dispozitiv. Alte informații incluse sunt: tipul dispozitivului, identificatorul acestuia și alte atribute.

În continuare este prezentat un exemplu de implementare.

```
private LeDeviceListAdapter mLeDeviceListAdapter;
...
private BluetoothAdapter.LeScanCallback mLeScanCallback =
        new BluetoothAdapter.LeScanCallback() {
    @Override
    public void onLeScan(final BluetoothDevice device, int rssi,
                         byte[] scanRecord) {
        runOnUiThread(new Runnable() {
           @Override
           public void run() {
               mLeDeviceListAdapter.addDevice(device);
               mLeDeviceListAdapter.notifyDataSetChanged();
           }
       });
   }
};
```

Este implementat `LeScanCallback` și se suprascrie metoda `onLeScan`. În această metodă se adaugă dispozitivul descoperit pe o lista din UI și se afișează pe ecran.

În continuare este implementat modul de pornire și oprire a scanării:

```
mBluetoothAdapter.startLeScan(mLeScanCallback);
...
mBluetoothAdapter.stopLeScan(mLeScanCallback);
```

4.6 Near Field Communication

Near Field Communication (NFC) este o tehnologie wireless ce funcționează la o distanță foarte mică (4 cm). Poate fi folosită pentru trimiterea mesajelor scurte de la un tag NFC la un dispozitiv mobil sau între două dispozitive. Datele transmise au un anumit format numit NFC Data Exchange Format (NDEF).

4.6.1 Moduri de operare NFC

Dispozitivele Android cu hardware NFC au de obicei 3 moduri de operare:

- Modul reader/writer - care poate fi folosit pentru a citi sau scrie pe taguri NFC
- Modul P2P - care este folosit pentru a trimite mesaje NFC către alt dispozitiv mobil
- Modul card emulation - care este folosit atunci când dispozitivul mobil se comportă ca un card NFC, de exemplu atunci când folosim telefonul la un terminal POS NFC.

4.6.2 Folosirea NFC într-o aplicație Android

În primul rând, trebuie specificată în Manifest și cerută explicit la runtime permisiunea `android.permission.NFC`. De asemenea trebuie folosit un nivel de API minim 10, pentru a include API-ul NFC: `<uses-sdk android:minSdkVersion="10"/>`.

În plus, trebuie specificată caracteristica hardware `android.hardware.nfc` în fișierul Manifest. Și la runtime trebuie verificat dacă hardware-ul NFC este disponibil pe dispozitiv: dacă `NfcManager.getDefaultAdapter()` nu întoarce `null`.

4.6.3 Descoperirea dispozitivelor NFC

Pentru a primi un Intent atunci când un tag NFC specific este scanat, trebuie specificat un `IntentFilter` cu acțiunea `android.nfc.action.NDEF_DISCOVERED`. Apoi când se primește Intent-ul, se va verifica dacă acțiunea este `NfcAdapter.ACTION_NDEF_DISCOVERED`. În final, datele pot fi obțiune din Intent prin `intent.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES)`.

Urmează un exemplu pentru obținerea mesajelor primite prin NFC:

```
@Override
protected void onNewIntent(Intent intent) {
    super.onNewIntent(intent);
    ...
    if (intent != null &&
        NfcAdapter.ACTION_NDEF_DISCOVERED.equals(intent.getAction())){
        Parcelable[] rawMessages = intent.getParcelableArrayExtra
                    (NfcAdapter.EXTRA_NDEF_MESSAGES);
        if (rawMessages != null) {
            NdefMessage[] messages = new NdefMessage[rawMessages.length];
            for (int i = 0; i < rawMessages.length; i++) {
                messages[i] = (NdefMessage) rawMessages[i];
            }
            // Se vor procesa mesajele
            ...
        }
    }
}
```

4.6.4 Trimiterea mesajelor NFC

Pentru trimiterea mesajelor NFC trebuie parcurși următorii pași:

- Trebuie creată o activitate care implementează `NfcAdapter.CreateNdefMessageCallback`.
- În `onCreate()`, trebuie obținută o instanță de `NfcAdapter`
- Trebuie configurată activitatea curentă să fie responsabilă cu gestiunea callback-urilor de la adaptor folosind `NfcAdapter.setNdefPushMessageCallback()`.
- Trebuie suprascris callback-ul `createNdefMessage()` care este apelat atunci când un tag NFC este descoperit.
- În acest callback se va crea mesajul pe care vrem să-l trimitem tag-ului NFC.
- În plus, vom folosi callback-ul `onNdefPushComplete()` pentru a notifica UI-ul că un mesaj a fost trimis.

În continuare este prezentat un exemplu de implementare a unei activități care trimite un mesaj NFC atunci când descoperă un nou tag:

```
public class Beam extends Activity implements CreateNdefMessageCallback {
    NfcAdapter mNfcAdapter;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        [...]
        mNfcAdapter = NfcAdapter.getDefaultAdapter(this);
        if (mNfcAdapter == null) {
            finish();
            return;
        }
        mNfcAdapter.setNdefPushMessageCallback(this, this);
    }

    @Override
    public NdefMessage createNdefMessage(NfcEvent event) {
        String text = ("Beam␣me␣up,␣Android!\n\n" +
                "Beam␣Time:␣" + System.currentTimeMillis());
        NdefMessage msg = new NdefMessage(
                new NdefRecord[] { createMime(
                        "application/vnd.com.example.android.beam",
                        text.getBytes())
        });
        return msg;
    }
[...]
```

4.7 API-urile Google

Serviciile puse la dispoziție de Google, cum ar fi Maps, Drive, etc., nu pot fi accesate prin API-ul public al Android-ului.

4.7.1 Google Play Services

Pentru a le folosi trebuie să avem Google Play Services (Figura 4.1) instalat și alte biblioteci proprietare. Google Play Services include o serie de servicii individuale și

poate fi instalat folosind Google Play. Bibliotecile client sunt obținute prin SDK Manager și pot fi incluse în aplicații.

Aplicația include biblioteca client, care comunică cu Google Play Services, care la rândul ei comunică cu serviciile externe oferite de Google.

Google Play Services este actualizat prin Google Play, de aceea actualizările serviciilor nu sunt dependente de actualizările imaginilor de sistem de la producător.

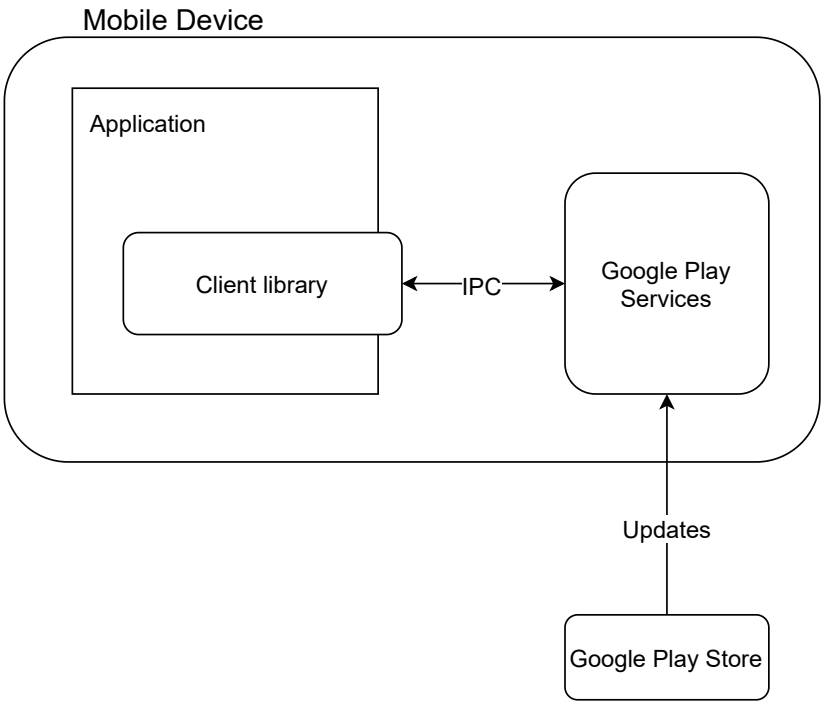

Figura 4.1: Google Play Services

4.7.2 Google Maps

Un exemplu de serviciu Google este Maps. Se poate folosi API-ul Maps în aplicații pentru a accesa serverele Google Maps, a downloada hărți și a gestiona interacțiunea utilizatorului cu hărțile. API-ul permite aplicațiilor să adauge informații pe o hartă, cum ar fi markers, overlays, polylines și polygones.

Pașii care trebuie urmați pentru folosirea Google Maps într-o aplicație sunt:

- Trebuie adăugată cheia Google Maps API ca meta-data în fișierul Manifest.
- Această cheie trebuie obținută prin înregistrarea aplicației la Google API Console
- Trebuie cerute următoarele permisiuni în fișierul Manifest:
  - `android.permission.INTERNET` - necesară pentru conectarea la serverele Google prin Internet și download-area bucăților de hartă
  - `android.permission.ACCESS_NETWORK_STATE` - pentru verificarea conexiunii la rețea înainte de a solicita datele
  - `android.permission.WRITE_EXTERNAL_STORAGE` - pentru a stoca hărțile pe sdcard

- `android.permission.ACCESS_COARSE_LOCATION` - pentru a obține locația prin WiFi/mobile.
- `android.permission.ACCESS_FINE_LOCATION` - este necesară pentru obținerea locației precise prin GPS, dar și dacă se folosește în plus WiFi/mobile.

Apoi este nevoie de a implementa un Fragment în Activitatea care va include harta. Numele fragmentului din layout trebuie să fie exact `com.google.android.gms.maps.MapFragment`.

În activitate trebuie obținută o instanță a acestui fragment și făcut cast la `MapFragment`. Pentru desenarea hărții, trebuie implementată interfața `OnMapReadyCallback` (activitatea să implementeze interfața) și apoi apelată metoda `MapFragment.getMapAsync(OnMapReadyCallback)`

Urmează un exemplu de implementare de activitate care include o hartă folosind Google Maps:

```
public class MapsActivity extends FragmentActivity implements
    OnMapReadyCallback {

    private GoogleMap mMap;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_maps);
        SupportMapFragment mapFragment = (SupportMapFragment)
                    getSupportFragmentManager().
                    findFragmentById(R.id.map);
        mapFragment.getMapAsync(this);
    }

    @Override
    public void onMapReady(GoogleMap googleMap) {
        mMap = googleMap;

        [..]
    }
}
```

4.8 Bibliografie

- https://developer.android.com/guide/components/processes-and-threads.html (Accesat: Mai 2021)
- https://developer.android.com/guide/background (Accesat: Mai 2021)
- https://developer.android.com/training/basics/network-ops/connecting.html (Accesat: Mai 2021)
- https://developer.android.com/reference/android/os/AsyncTask.html (Accesat: Mai 2021)

- https://developer.android.com/training/location/request-updates (Accesat: Mai 2021)
- https://developer.android.com/guide/topics/connectivity/bluetooth.html (Accesat: Mai 2021)
- https://developer.android.com/guide/topics/connectivity/bluetooth-le.html (Accesat: Mai 2021)
- https://developer.android.com/guide/topics/connectivity/nfc/index.html (Accesat: Mai 2021)
- https://developers.google.com/maps/documentation/android-sdk/overview (Accesat: Mai 2021)

Capitolul 5

Concluzii

Android-ul câștigă din ce în ce mai multă popularitate datorită faptului că este ușor de folosit și se pot instala un număr mare de aplicații gratuite de pe Google Play Store. Din punct de vedere al programatorilor, este facilă dezvoltarea noilor aplicații pornind de la documentația și exemplele oferite de Google.

În această carte este prezentată o introducere în Android. Pentru început este descrisă sumar arhitectura Android-ului, cu componentele de bază, de la kernel-ul de Linux, userspace-ul nativ, runtime-ul Android, serviciile de sistem, API-ul de dezvoltare al aplicațiilor, până la aplicațiile stock și third-party. Aceste aplicații sunt dezvoltate folosind patru elemente principale: activități, servicii, broadcast receivers și content providers. Pentru a controla accesul unei aplicații la componentele unei alte aplicații, Android oferă mecanismele de sandboxing și de permisiuni.

În continuare, este descris Android SDK cu ajutorul căruia dezvoltatorii pot sa creeze aplicații Android. Structura de bază a unui proiect Android este centrată în jurul fișierului Manifest care precizează toate elementele de bază care sunt definite în cadrul aplicației: componente, permisiuni. Prima componentă descrisă este activitatea care reprezintă cea mai importantă componentă dintr-o aplicație pentru că este singura componentă vizuală și interactivă cu utilizatorul.

A doua componentă este reprezentată de servicii care au rolul de a efectua operațiile intensiv computaționale sau de lungă durată în background. A treia componenta este broadcast receiver-ul care are rolul de a primi mesajele de broadcast emise de către sistem sau de către alte aplicații și de a efectua acțiuni pe baza acestora. Ultima componentă este content provider-ul care reprezintă, în esența, o interfață care oferă acces la date organizate în mod structurat precum o bază de date.

Sistemul de operare Android are la bază un kernel de Linux vanilla peste care s-au adăugat funcționalități specifice pentru dispozitive embedded mobile precum: wakelocks, low memory killer sau paranoid networking. Aceste funcționalități specifice poartă numele de androidisme.

Binder-ul este componenta centrală care stă la baza framework-ului de securitate din Android: asigura comunicarea între oricare două componente din aplicații diferite. Binder-ul sta si la baza comunicării cu serviciile de sistem. Acestea sunt procese care oferă funcționalități critice pentru un dispozitiv Android: găsirea tuturor serviciilor de sistem (responsabilitate a Service Manager), controlul ciclului de viață al unei activități

sau verificări de permisiuni (responsabilități deținute de Activity Manager), instalarea sau dezinstalarea de pachete efectuate de către Package Manager.

O bună parte din aplicațiile Android au nevoie de stabilirea unei conexiuni de rețea prin HTTP(S). Această conexiune poate fi stabilită prin diferite tehnologii precum: WiFi, date mobile (4G, 5G), Bluetooth sau NFC. Accesarea conținutului online trebuie, mereu, efectuată în cadrul unui worker thread diferit de thread-ul de UI (principal) al aplicației. Worker thread-ul poate să ofere informații către thread-ul principal prin mai multe feluri: AsyncTask, Handler.