EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

Programozáselmélet és Szoftvertechnológiai Tanszék

Drive testing Androiddal

Sike Sándor
ELTE IK Programozáselmélet és Szoftvertechnológiai Tanszék

Specker Zsolt
ELTE IK Programtervező Informatikus BSc

Simonyi Tibor
Ericsson Magyarország Kft.

Budapest, 2014

Tartalomjegyzék

1. fejezet

Bevezetés

1.1. Probléma leírása

A Drive testing egyfajta mérési módszere és értékelése a mobiltelefon hálózatok teljesítményének, kapacitásának, lefedettségének és minőségének. Ez a teszt fontos szerepet kap a telefonhálózatokat üzemeltető cégek - operátorok - hálózat tervezési, üzemeltetési stratégiáiban és nem utolsó sorban a hálózati hibák felderítésében. A drive test tipikusan az alábbi adatokat gyűjti össze a hálózatról:

- Jel intenzitás, erősség
- Adat fel- és letöltési sebesség
- Elutasított és blokkolt hívások
- Handover információk (ez egyes tornyok, cellák közötti váltás)
- Szomszédos cella információk

A manapság használt megoldások nagyon költségesek a drága eszközök és szakemberek miatt. A mérés során a tesztelő - aki általában egy mérnök - egy speciálisan felszerelt autóval kivonul a megadott területre, ahol a tesztet el kell végeznie, majd bekonfigurálja az eszközöket a méréshez. Egy egyszerűbb felszerelés látható a jobb oldali képen. A mérés típusától függően lehet, hogy bonyolultabb és költségesebb eszközökre, felszerelésekre van szükség. Az autóval végigjárja a kijelölt útvonalat - akár többször is - miközben a mérési eredményeket rögzíti a

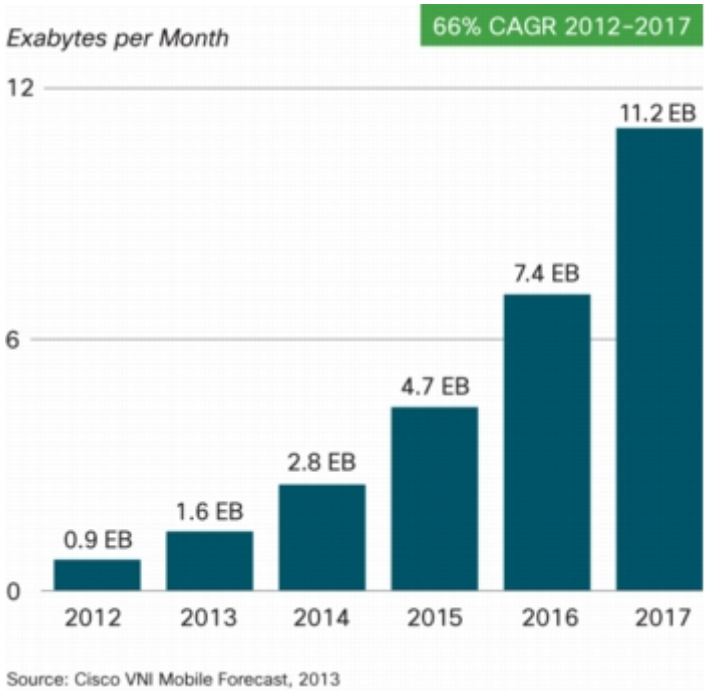
1.1.1. ábra. Egy Drive Test felszerelés

számítógép egy speciálisan erre a célra fejlesztett szoftver segítségével. A mérés végeztével kiértékeli az eredményeket a program segítségével. Látható, hogy sok költséges, speciális eszközre van szükség a mérések elvégzéséhez.

1.2. Szakdolgozat célja

A szakdolgozatom célja egy ”Drive testing" alkalmazás megvalósítása, egy kis költségvetésű Android operációs rendszerű mobilkészülék segítségével. Nagy előnye ennek a megoldásnak a meglévő technológiával szemben, hogy a teszt elvégzéséhez elég egy mobiltelefon készülék és nem kell drága felszerelést és mérnököt alkalmazni. Az Android platform lehetőséget nyújt arra, hogy az alkalmazás információkat gyűjtsön a telefonról, valamint a mobilhálózatról. Ezen információk segítségével egy olyan program készíthető, amely képes részben vagy teljesen kiváltani a jelenleg használt technikát. Segítséget nyújt a telefonhálózat problémáinak felderítésében és a fejlesztendő területek feltérképezésében.

Az Android platformot választottam, mivel széles körben elérhető és a [Wikipedia] adatai szerint ezt használják a legtöbben a világon. Így akár a felhasználók által gyűjtött teszt adatokat is felhasználhatják a telefon szolgáltatók, operátorok, mivel a program használata egyszerű, nem igényel szaktudást. Csak az adatok értelmezéséhez, feldolgozásához kell szakember.



1.2.1. ábra. Mobil adatforgalom 2012-2017

A szakdolgozatom leginkább a mobil adathívások tesztelésével és elemzésével foglalkozik, mivel a jövőben - 4G-s hálózatok esetén is - ez a terület különösen fontos szerepet fog játszani a mobilkommunikációban. Kimutatások szerint egyre növekszik a mobilhálózatok adatforgalma és 2017-re elérheti a havi 11,2 exabytenyi mennyiséget. A mobilkészülékek és hálózatok fejlődésével teret hódítanak a különféle szolgáltatások, mint a videó telefonálás, melyek növelik az adatforgalmat és szükségessé teszik a gyors és

megbízható adatszolgáltatást. A felhasználók gyakrabban használják ezeket a szolgáltatásokat, mint a hagyományos hanghívást. Így a szolgáltatóknak is alkalmazkodniuk kell a megváltozott igényekhez, biztosítaniuk kell a nagyobb lefedettséget és megbízható, gyors adatátvitelt.

2. fejezet

Szükséges ismeretek

2.1. Mobilhálózatok bemutatása

Mobilhálózatok kialakulása

Már a második világháborúban képesek voltak rádiótelefonos kapcsolatot kiépíteni katonai használatra. Kézi rádió adó-vevők már az 1940-es évektől elérhetőek voltak, pár telefon társaságnak volt gépkocsiban is használható mobil telefon készüléke. A korai készülékek testesek voltak és sok áramot fogyasztottak, a hálózat pedig csak néhány szimultán beszélgetést tudott csak kiszolgálni. Az egyesült államokbeli Bell laboratórium mérnökei olyan rendszeren dolgoztak, ami képessé teszi a mobil felhasználókat hívások kezdeményezésére és fogadására akár az autójukból. Az AT&T vállalat 1947-ben megalkotta az első mobiltelefon hálózatot a Mobile Telephon Service-t (MTS). 5,000 ügyféle volt és körülbelül heti 30 000 hívást bonyolítottak rajta. A cella technológia bevezetése lehetővé tette a frekvenciák sokszori újrahasznosítását kis szomszédos területeken, melyeket alacsony teljesítményű adók segítségével fedtek le. Ez lehetővé tette gazdaságilag megvalósítható módon a mobiltelefonok széles körben való elterjedését.

2.1.1. ábra. Első mobil telefonok egyike

A mobil telefonok fejlődése jól nyomon követhető az egymást követő generációkon, a korai "0G" szolgáltatásokon keresztül az első generációs (1G) analóg cellás hálózatok és második generációs (2G) digitális mobil hálózatokon át a har-

madik generációs (3G) szélessávú adat szolgáltatásokig. Napjainkban fejlődnek és egyre inkább elterjednek a negyedik generációs (4G) native-IP hálózatok.

Cellák koncepciója

1947-ben Douglas H. Ring és W. Rae Young a Bell laboratórium mérnökei javasolták a hatszögletű cellák használatát a mobil hálózatokban. Akkoriban ez még nem volt megvalósítható csupán két évtizeddel később, mikor Richard H. Frenkiel, Joel S. Engel és Philip T. Porter mérnökök kiterjesztették a korábbi javaslatot és egy sokkal részletesebb tervet dolgoztak ki. Porter volt, aki először javasolta, hogy az adótornyok használják a már jól ismert irányított antennákat, annak érdekében, hogy redukálják az interferenciát és növeljék a csatornák újrahasznosíthatóságát.

2.1.2. ábra. Többirányú mobilhálózati antenna

Ezekben a korai időkben a mobilkészüléknek az adótorony által lefedett körzetben kellett maradnia, ahhoz hogy szolgáltatás folyamatos maradjon, vagyis a szolgáltatás nem volt folytonos, átjárható az egyes cellák között. A koncepciók, mint a frekvencia újrahasználás és az átadás (handoff vagy handover) számos új mobiltelefon hálózati koncepciók alapját képezték. 1970-ben Amos E. Joel Bell Laboratórium mérnöke feltalálta a három oldalú antennát, hogy segítse a hívások átadásának folyamatát egyik cellából a másikba.

2.1.1. Mobilhálózatokban gyakran használt kifejezések leírása

Hozzáférési Technológiák

FDMA (Frequency Division Multiple Access) a leggyakoribb analóg rendszer. Ez a technika a spektrum frekvenciatartományokra való feldarabolásán alapszik és egy frekvenciasávot társít egy felhasználóhoz. Egy csatornát egy időben csak egy előfizető használhat, így egy csatorna lezárul a beszélgetés idejére más felhasználók elől, amíg le nem teszik, vagy át nem adódik a beszélgetés egy másik csatornának. A "full-duplex" FDMA átvitelnek két

csatornára van szüksége, egy az adatok küldésére és egy másik a fogadására. FDMA technikát az első generációs analóg rendszerekben használták.

TDMA (Time Division Multiple Access) nagyobb kapacitásra volt képes, mert az egyes frekvenciákat időszeletekre (time slot) osztotta. TDMA rendszerben a felhasználók az egész frekvenciatartományt használhatták, több felhasználó osztozott ugyanazon a frekvencia csatornán amiből egy-egy időszeletet kaptak. TDMA a második generációs mobil hálózatok domináns technológiája.

CDMA (Code Division Multiple Access) szórt spektrumú technológián alapul. Titkosított adat átvitelt is lehetővé tette, ezért régóta használják katonai célokra. CDMA megnövelte a spektrum teljesítő képességét azzal, hogy a felhasználók az összes csatornát használhatják egy időben. Az adások az egész hullámsávot kihasználják, és minden hang vagy adat híváshoz egy egyedi kód tartozik, ami megkülönböztethetővé teszi egymástól a hívásokat.

Mobil hálózati szabványok

PSTN (Public Switched Telephone Network) nyilvános vonalkapcsolt telefonhálózat. Telefon vezetékek, optikai kábelek, mikrohullámú átviteli állomások, kommunikációs műholdak és tengeralatti telefon kábelek kötik össze a kapcsolási központokat, hogy bármely telefon készülékkel felvehessük a kapcsolatot a világon. Eredetileg a hálózat analóg, vezetékes telefon rendszer volt, PSTN majdnem teljesen digitális, mobil és vezetékes telefonokat is tartalmaz. Egyetlen globális címtartománya van a telefonszámoknak, melyet a E.163 és E.164 szabványok írnak le. Az összekapcsolt hálózatok és az egységes számozásnak köszönhetően bármely telefonról hívható bármely másik készülék.

GSM (Global System for Mobile communication) ezt a szabványt az ETSI (European Telecommunications Standards Institute) a második generációs (2G) mobilhálózati protokollok leírására. A GSM sztenderd váltotta le az első generációs (1G) analóg hálózatot egy új, digitális, teljes kétirányú kommunikációt lehetővé tevő, vonalkapcsolt hálózat bevezetésével. Ezt később kiegészítették azzal, hogy belevették az adatkommunikációt is a szabványba, először vonalkapcsolt átvitellel, majd csomagkapcsolt átvitellel: GPRS (General Packet Radio Services) és EDGE (Enhanced Data rates for GSM Evolution vagy EGPRS).

GPRS (General Packet Radio Service) egy csomagkapcsolt szolgáltatás 2G és 3G kommunikációs rendszerekben. A GPRS-t az ETSI szabványosította, napjainkban a 3GPP (3rd Generation Partnership Project) tartja karban a szabványt. A GPRS szolgáltatás jellemzője a változó teljesítmény és késés, melyek mértéke a rendszert egyidejűleg használók számától függ. A vonal kapcsolt rendszerrel ellentétben, ahol szolgáltatás minősége (Quality of Service) biztosított a kapcsolat ideje alatt. 2G-s rendszerben a GPRS által biztosított adatráta 56–114 kbit/s. A 2G és GPRS technológiák kombinációját néha 2.5G-nek szokták hívni, ami a 2G és a 3G mobil technológiák között helyezkedik el. Mérsékelt sebességű adatátvitelt biztosít a GSM rendszerben, az addig kihasználatlan több csatornás idő osztásos (TDMA: Time division multiple acces) technológia segítségével.

EDGE (Enhanced Data rates for GSM Evolution), más néven EGPRS (Enhanced GPRS) mobiltelefon technológia, ami nagyobb adatátviteli sebességet tesz lehetővé. EDGE a 2003 év elején került bevezetésre a GSM hálózatokban. A 3GPP szabványosította és GSM család részévé vált. Kifinomult kódolási eljárások bevezetésének köszönhetően megnőtt a csatornánként elérhető bitráta, háromszoros kapacitás és teljesítmény beli növekedést eredményezett a hagyományos GSM/GPRS kapcsolathoz képest. A továbbfejlesztett EDGE kisebb késést és több mint kétszeres teljesítmény növekedést biztosít. A legnagyobb bitráta elérheti a 1Mbit/s-ot, átlagosan pedig a 400kbit/s-ot.

W-CDMA (Wideband Code Division Multiple Access) egy 3G mobilhálózati rádiós interfész sztenderd. A leggyakrabban használt tagja az Universal Mobile Telecommunications System (UMTS) családnak és néha az UMTS szinonimájaként használják. A DS-CDMA csatorna hozzáférési és FDD duplex módszereket alkalmazza, hogy nagyobb sebességre legyen képes és több felhasználót tudjon kiszolgálni a TDMA és TDD (time division duplex) rendszerekhez képest. Ugyanazt a központi hálózatot használja, mint a 2G GSM hálózat. Code Division Multiple Access kommunikációs hálózat sok cég közreműködésével jött létre, de a CDMA (inkább W-CDMA) cellás telefon hálózat fejlesztésében a Qualcomm dominált. Qualcomm volt az első cég, akinek sikerült alkalmazható és költséghatékony megoldást találnia: ez volt a korai IS-95 rádióinterfésze sztenderd, melyből a CDMA2000 (IS-856/IS-2000) fejlődött ki. CDMA2000 hálózatok széles körben elterjedtek különösen Amerikában. Azonban az eltérő követelmények és felépítés megakadályozta a globális elterjedését. A meglévő rádiós interfészekkel való inkompatibilitás és a nagy költséggel járó fejleszthetőség ellenére a W-

CDMA egy meghatározó szabvánnyá vált. A W-CDMA egy pár 5 MHz széles rádió csatornát használ, míg a CDMA2000 egy vagy több pár 1.25 MHz széles rádió csatornát. Habár a W-CDMA szintén CDMA átviteli technikát alkalmaz, akár a CDMA2000, a W-CDMA nem egyszerűen egy széles sávú verziója a CDMA2000-nak. Mérnöki szempontból a W-CDMA különböző optimalizálási lehetőséget nyújt költség, kapacitás és teljesítmény között. Specifikációk egész halmazát tartalmazza, melyek részletesen definiálják a protokollokat, a kommunikációt a készülékek és a tornyok között, hogyan kell modulálni a szignálokat és azt is hogyan kell az adatokat strukturálni.

UMTS (The Universal Mobile Telecommunications System) harmadik generációs mobil hálózati rendszer. A 3GPP (3rd Generation Partnership Project) fejlesztette ki és tartja karban. W-CDMA hozzáférési technológiát használ a jobb hálózati kihasználtság és sávszélesség miatt. UMTS specifikál egy teljes hálózati rendszert, melybe beletartozik a hozzáférési hálózat (UMTS Terrestrial Radio Access Network, UTRAN), a központi hálózat (core network) és a felhasználók SIM általi azonosítása. Az EDGE és a CDMA2000 ellentétben az UMTS új bázis állomást és frekvencia allokációs módszert igényel. Az UMTS által támogatott maximális adat átviteli ráta 42 Mbit/s, ha HSPA+ alkalmaznak a hálózatban és 7.2 Mbit/s HSDPA használat esetén. Ezek a sebességek szignifikánsan gyorsabbak, mint a GSM 9.6 kbit/s-os hibajavításos vonalkapcsolt adatátvitele, vagy a CDMAOne csatornák 14.4 kbit/s rátája. 2006 óta UMTS hálózatokat számos országban feljavították a High Speed Downlink Packet Access (**HSDPA**) technikával, melyet néha 3.5G néven emlegetnek. HSDPA segítségével a letöltési sebesség elérheti a 21 Mbit/s-ot. A feltöltési sebesség javítását támogató technológia a High-Speed Uplink Packet Access (**HSUPA**). Az első UMTS hálózatot 2002-ben indították be, nagy hangsúlyt fektetve az olyan mobil alkalmazásokra, mint a mobil TV és videóhívás. A nagy adatátviteli sebességet gyakran Internet elérésre használják.

LTE (Long-Term Evolution) 4G LTE, a legújabb sztenderd a mobiltelefonok és adat terminálok közötti nagy sebességű vezeték nélküli adatkommunikációra. GSM/EDGE és UMTS/HSPA hálózati technológiákon alapszik, de nagyobb kapacitást és sebességet nyújt különböző rádiós interfészek együttes alkalmazásával. A világ első publikus LTE szolgáltatója a TeliaSonera Osloban és Stockholmban. Várhatóan az LTE lesz az első igazán globális mobiltelefon sztenderd, habár a országonként különböző frekvenciasávok használata miatt csak a többsávos telefonok lesznek képesek kihasználni minden országban az LTE szolgáltatást. Míg a korábbi technológiák 28

Mbit/s elméleti maximumot támogatták, az LTE 326 Mbit átvételére lesz képes másodpercenként, 20 MHz széles frekvenciatartományon. Az LTE sztenderd csak IP hálózatokat támogatja. A GSM, UMTS és CDMA2000 hagyományos vonalkapcsolt hálózatok, így az LTE-vel való együttműködés miatt át kell tervezni a hálózatokat. Erre több fél megoldást dolgoztak ki:

VoLTE (Voice Over LTE) megközelítés IP Multimedia Subsystem (IMS) hálózaton alapszik. Ezzel a megoldással a hangszolgáltatás adatfolyamként továbbítódik az LTE állomáson keresztül. Ez azt jelenti, hogy nem kell módosítani a hagyományos vonalkapcsolt hálózatot, képes lesz a 4G-s hívást adatcsomagként továbbítani. Másik megközelítés a **CSFB** (Circuit Switched Fallback). Eben az esetben az LTE csak adatcsomag szolgáltatást nyújt és a hanghívások vonalkapcsolt hálózatban dolgozódnak fel. Ezt a megoldást használva az operátoroknak csak az MSC-t kell frissíteniük IMS telepítés helyett. Hátránya a hosszabb hívás kiépítési idő. A jobb minőségű hang biztosításához a 3GPP megköveteli legalább az AMR-NB kodek (narrow band) használatát, de a javasolt beszéd kodek VoLTE-hoz az Adaptive Multi-Rate Wideband, avagy a HD Voice.

Mobil hálózat elemei

BSC (Base Station Controller) több tíz esetleg több száz BTS-t irányít. A BSC rádió csatornák kiosztásáért felelős, mérési eredményeket fogad a mobil készülékektől és a BTS-ek közötti handover-eket kezeli. BSC fő funkciója összekötőként viselkedik a BTS-ek és a Mobile Switching Center (MSC) között. Általában a hálózatokban gyakorta úgy strukturálják, hogy számos BSC-t elosztanak a régiókban közel a BTS-ekhez, melyek egy nagy központosított MSC site-tal állnak kapcsolatban. MSC-ket és Service GPRS Support Node-okat (SGSN) is kiszolgál (abban az esetben ha GPRS-t használ).

BSS (Base Station Subsystem) része a hagyományos mobilhálózatnak (2G, GSM) és a kommunikációt biztosítja a mobiltelefon és a hálózati kapcsolóalrendszer között. A BSS transzkódolást végez, rádió csatornákat rendel a mobiltelefonokhoz, lapozást kezel, és még sok más feladatot ellát a rádiós hálózatban.

BTS (Base Transceiver Station) tartalmazza a rádió szignálok átviteléhez és fogadásához szükséges berendezéseket, antennákat, valamint a bázisállomással (BSC) történő kommunikáció dekódoláshoz és titkosításhoz szükséges eszközöket. Valójában nem más, mint egy pikócella sok rádió adó-

vevővel (TRX), amik lehetővé teszik különböző frekvenciák kiszolgálását. Egy BTS-t a "szülő" bázis állomás (BSC) vezérel.

MS (Mobile Station) azaz maga a mobil készülék, amivel a felhasználó hívást kezdeményezhet és fogadhat. Ez a terminológia 2G rendszerekre vonatkozik, mint a GSM. 3G rendszerekeben a mobile station helyett "user equipment"-ként (UE) hivatkoznak rá.

RNC (Radio Network Controller) az UMTS mobil hálózat (UTRAN) irányító egysége és a hozzá kapcsolt "Node B"-k kontrollálásáért felelős. BSC-nek 3G-s hálózati megfelelője, hasonló munkát végez: az üzenetek encryptálása és erőforrás kezelés. Az RNC kapcsolatban áll a Circuit Switched Core Network-hez a Media Gateway-en (MGW) keresztül és a Serving GPRS Support Node-on (SGSN) keresztül a Packet Switched Core Network-höz.

Node-B Ezt a terminust az UMTS hálózatban használják és megfelel a BTS-nek a GSM-ben. A mobil készülékek Node B-n tudnak egymással kommunikálni. Az RNC kontrollálja ezt a hálózati elemet.

VLR (Visitor Location Register) egy adatbázis azon előfizetők számára, akik beléptek az adott VLR-hez tartozó Mobile Switching Center (MSC) területére. Minden egyes bázisállomáshoz pontosan egy VLR tartozik, így az előfizető egy időben mindig csak egy VLR-ben szerepel. Az adatok vagy a mobil készülékről (MS), vagy a HLR-től érkeznek. Amint érzékel egy új MS-t az MSC, azonnal készít egy új bejegyzést a VLR-jében és frissíti a HLR-ben az előfizető helyzetét.

HLR (Home Location Register) egy központi adatbázis, ami tartalmazza az összes mobiltelefon előfizetője adatát, akik használhatják hálózatot.

M-MGW (Mobile Media Gateway) egy olyan eszköz, ami a digitális média folyamokat konvertálja (transzkódolja) a különböző telekommunikációs hálózatok között (PSTN, 2G, 3G, stb.), illetve lehetővé teszi a különféle átviteli protokollok használatát (ATM, IP, TDM) . Általában az egyes hálózatok határán helyezkedik el, hogy biztosítsa az adatok konvertálását egyik hálózatból a másikba.

SGSN (Serving GPRS Support Node) felelős a mobileszközök és kiszolgáló állomások közötti adat csomagok szállításáért. A feladatai közé tartozik az irányítás (routing), a helymeghatározás és a felhasználók azonosítása (authentication).

GGSN (Gateway GPRS Support Node) központi eleme a GPRS hálózatnak, az internet elérését biztosítja a GPRS hálózat és a külső hálózatok között. Kívülről nézve a GGSN egy router, ami elrejti a GPRS hálózat infrastruktúráját a külső hálózattól. Mikor adat csomagokat kap megvizsgálja, hogy a felhasználó aktív-e, és ha igen akkor továbbítja a csomagot az SGSN-nek, aki kiszolgálja a mobil felhasználót.

MSC (Mobil Softswitch Controller) elsődleges eleme a GSM/CDMA hálózatoknak, a hanghívások route-olásáért és az SMS, valamint más szolgáltatásokért felelős, mint a konferencia hívások, FAX, stb. Kiépíti, majd lebontja a kapcsolatokat, hívásokat, valamint kezeli a handover-eket.

IMS (IP Multimedia Subsystem) egy architektúra az IP alapú multimédiás szolgáltatásokhoz. Kezdetben a 3GPP arra fejlesztette ki, hogy terjesszék ki a GPRS-t Internet alapú szolgáltatásokkal. A könnyebb integrálhatóság miatt IETF protokollokat használnak ahol csak lehetséges, ilyen például a SIP (Session Initiation Protocol) .

Mobil hálózat szolgáltatások

SIM (Subscriber Identity Module) egyik fő jellemzője a GSM-nek a SIM bevezetése, a legtöbben SIM kártya néven ismerik. A SIM egy cserélhető okoskártya, mely az előfizető információt és telefonkönyvet tartalmazhat. Lehetővé teszi a felhasználónak, hogy információkat tároljon és egy másik telefonban is fel tudja használni azokat. Ezentúl lehetővé teszi, hogy szolgáltatót váltson a felhasználó a készülék lecserélése nélkül, mivel csak a SIM kártyát kell lecserélnie. Néhány operátor blokkolja ezt a lehetőséget a SIM lezárásával, ez esetben a készülék csak az adott szolgáltató kártyáját fogadja el.

WAP (Wireless Application Protocol): egy sztenderd technika a vezetéknélküli hálózatokon keresztül történő információ cserének. A WAP böngésző egy webböngésző a mobil készülékekhez. A WAP bevezetése előtt, a mobil szolgáltatóknak limitált lehetőségük volt az interaktív adatszolgáltatásra. Interaktivitást követelt az Internet és Web alkalmazások támogatása pl.: email kezelés, tőzsdei árak vagy a sporteredmények követése. A WAP sztenderd egy protokoll családot ír le, mely lehetővé teszi az interoperabilitást a WAP és más különböző hálózati technológiák között, mint a GSM és a CDMA.

Roaming egy általános kifejezés a mobil szolgáltatás kiterjesztésére olyan helyre, amely különbözik a szolgáltatás regisztrált helyétől. Roaming biztosítja a kapcsolatot vezetéknélküli eszköz és a hálózat között, a kapcsolat elvesztése nélkül.

Hagyományos GSM Roaming definíció szerint (GSM Association Permanent Reference Document AA.39) a cella felhasználó képes automatikusan hívást indítani és fogadni, adatokat küldeni és fogadni, vagy más szolgáltatásokhoz kapcsolódni, miközben utazik, elhagyja a hazai szolgáltatás által lefedett területet.

Handover kifejezés (vagy handoff) arra a folyamatra utal, mikor egy folyamatban lévő hívást vagy adat folyamot átviszünk az egyik csatornáról a másikra. Erre akkor van szükség, mikor a készülék átlépi a cella határát, vagyis kilép az eddigi adótorony körzetéből és egy másik veszi át a kiszolgálását. Többféle handover létezik, attól függően, hogy a telefon melyik cellát, milyen irányba lépi át. Például abban az esetben, mikor a készülék nem csak egy cella határán van, hanem egy tartomány határán is, akkor nem csak egy másik toronyhoz (BTS vagy Node B) kerül át, hanem egy másik BSC vagy RNC-hez is.

2.1.2. Mobilhálózatok fejlődése

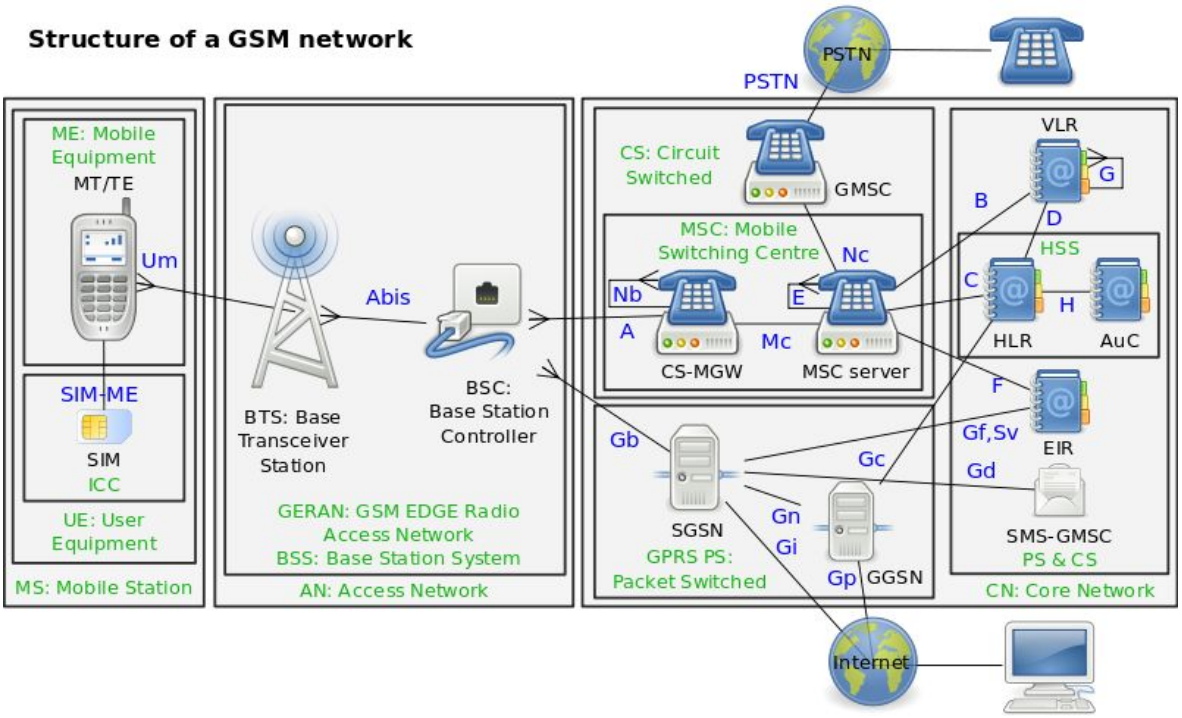
Az első rádiótelefon szolgáltatást a ’40-es évek végén vezették be Amerikában. Ez azt jelentette, hogy a mobil használók akár autóból is tudtak kapcsolódni a meglévő publikus telefonhálózathoz. 1960-ban új rendszert indított a Bell Systems, amit IMTS-nek (Improved Mobile Telephone Service) hívtak, mely számos újítást hozott, például szélesebb sávszélességet. Az első analóg cellás rendszer ezen az IMTS rendszeren alapult és a 1970-es évek elején fejlesztették ki. A rendszer "cellás" volt, mert a lefedett területeket felosztották kisebb részekre, avagy cellákra.

Első generáció

Első generációs mobil rendszer az AMPS (Advance mobile phone system), melyet először az Egyesült államokban vezették be. Ez az egyik legjobb FDMA (Frequency Division Multiple Access) technológia, mely lehetővé teszi, hogy hanghívást bonyolíthat az egész országban.

Második generáció

2G digitális cellás rendszer, melyet az 1980-as évek végén fejlesztették ki. Az új rendszer jobb minőséget és nagyobb kapacitást biztosít a felhasználóknak az analóg 1G rendszerhez képest. GSM (Global system for mobile communication) volt az első kereskedelemben működtetett digitális cellás rendszer, mely TDMA (Time division multiple acces) alapú.

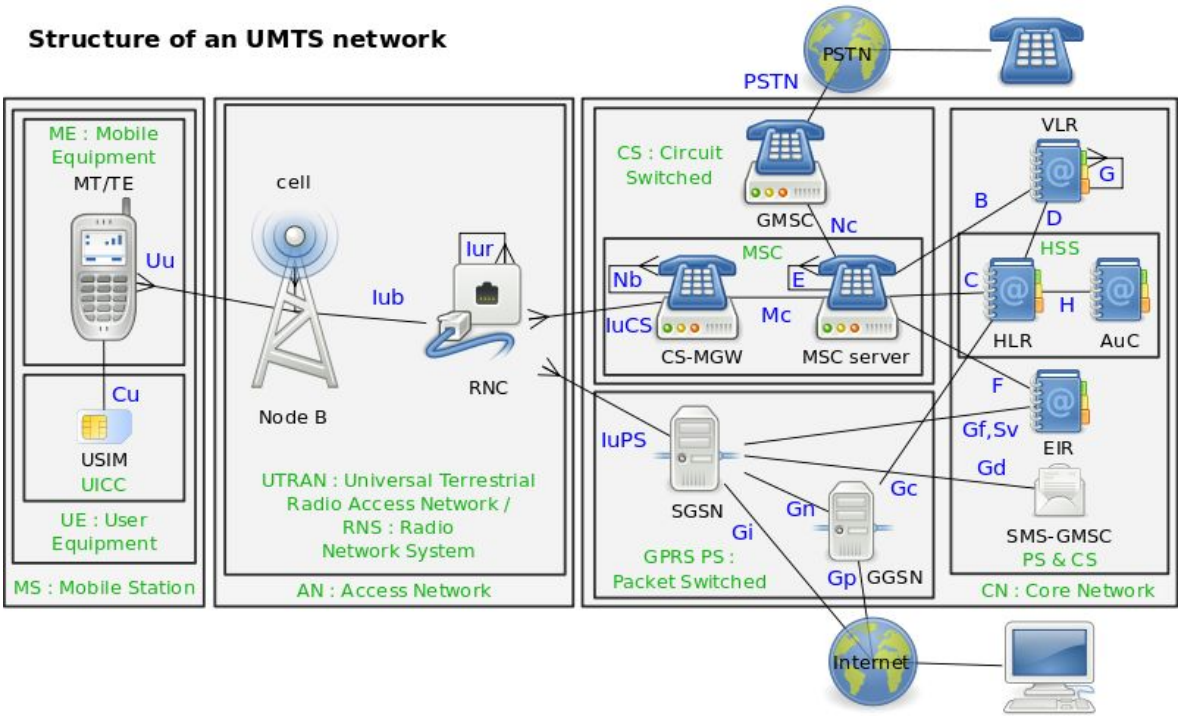


2.1.3. ábra. 2G hálózat struktúrája

A 2G technológiák biztosítanak különböző szolgáltatásokat, mint szöveges üzenet, kép üzenet és MMS (multi media messages) küldését. Az összes szöveges üzenet digitálisan titkosított, így csak a megfelelő címzett tudja fogadni és elolvasni az üzenetet. Idővel sokat fejlesztettek a rendszeren, így alakult ki a korábban már említett GPRS és az EDGE technológiák.

Harmadik generáció

3G kezdeményezés a készülék gyártóktól származott, nem pedig a telefonhálózatok operátoraitól. A fejlesztés 1996-ban kezdődött a Nippon Telephone & Telegraph (NTT) és az Ericsson vezetésével; 1997-ben az amerikai Telecommunications Industry Association (TIA) a CDMA-t választotta 3G technológiájának; 1998-ban az European Telecommunications Standards Institute (ETSI) ugyanezt tette és végül a széles sávú CDMA (W-CDMA) és a CDMA 2000 adaptálták az UMTS-hez (Universal Mobile Telecommunications System). W-CDMA és CDMA 2000 volt a két legnagyobb javaslat a 3G technológiára. A W-CDMA TDM-et (Time Division Multiplexing), ezzel ellentétben a CDMA 2000 CDM-et (Code Division Multiplexing) eljárást használ.



2.1.4. ábra. 3G hálózat struktúrája

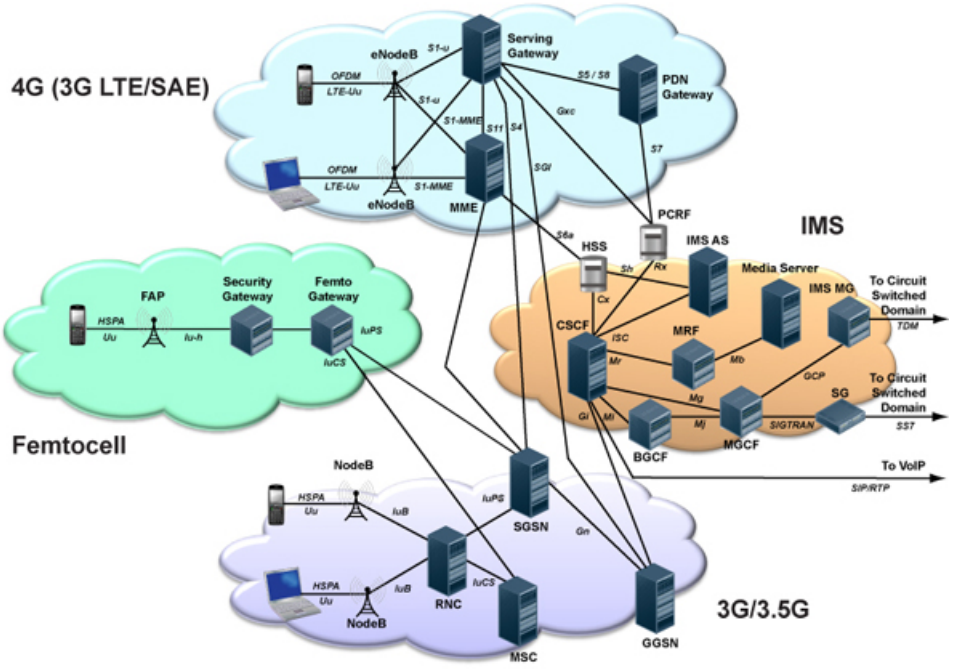
3G rendszer gyorsabb kommunikációs szolgáltatást nyújt, beleértve a hang, fax és Internet szolgáltatásokat, melyek bármikor és bárhonnan elérhetőek. Az ITU féle IMT-2000 3G sztenderd tette lehetővé az innovatív applikációk, például multimédiás szórakoztatás, információ és helymeghatározás szolgáltatások elterjedését a felhasználók között. Az első 3G hálózatot Japánban építették ki 2001-ben. 2.5G hálózatok, mint a GPRS (Global Packet Radio Service) már elérhetőek voltak Europa bizonyos országaiban. 3G technológia támogatja a 144 Kbps sávszélességet nagy sebességű mozgásnál (pl: járművek), 384 Kbps pl. egyetemek területén és 2 Mbps-ot állóhelyzetben (pl.: épületen belül). A maximális letöltési adatráta 384Kpbs, átlagosan 200 Kpbs és a feltöltés is ráta 64 Kpbs.

Negyedik generáció

Negyedik generációs (4G) mobil kommunikáció nagyobb adatátvitelre lesz képes, mint a 3G. Az adatráta tervezett mértéke 100 Mbps .

A 4G szolgáltatás biztosítani fogja a szélessávú, nagy kapacitású, nagy sebességű adatátvitelt, így lehetővé teszi többek között a jó minőségű, nagy felbontású videótelefonálást. A 2008-as ITU szabvány szerint legfeljebb 100 Mbit/s átviteli sebesség elérése lesz képes egy gyorsan mozgó készülék esetén (például: kocsiban, vonaton) és akár 1Gbit/s adatráta is elérhető egyéb esetekben. Két 4G rendszertervet dolgoztak ki, amit már kereskedelmi forgalomban is használnak,

ez a Mobile WiMAx és az LTE. Egyik megoldás sem teljesen teljesíti az ITU szabványt, ezért ezeket még ”hivatalosan” nem tartják a szakemberek negyedik generációs hálózatnak, habár az operátorok már annak hirdetik. Mindkét technológiát fejlesztik, hogy megfeleljen a 4G szabvány elvárásainak, főleg az adatátviteli sebesség kapcsán. Egyáltalán nem támogatja már a hagyományos vonalkapcsolt szolgáltatásokat, csak az internet protokollra (IP) épülőket, mint amilyen az IP telefonálás.



2.1.5. ábra. 4G és 3G hálózat architektúrája

2.1.3. A hálózatok vizsgált adatai

A lenti táblázat tartalmazza az átviteli sebességeket különféle technológiák esetén. Ezek a sebesség adatok módosulhatnak attól függően, hogy milyen rádiótechnológiát használ a telefonhálózat operátora (CDMA, FDD, stb.) Amint látható, a 2G és 3G hálózatok adatrátája egyre inkább növekszik, így lehetővé téve a nagyobb sávszélességigényű szolgáltatások bevezetését, mint a HD audió és videó szolgáltatások.

Technológia	Adatráta
GSM	9.6Kbps
GSM/GPRS	40Kbps-144Kbps
EDGE	474Kbps
UMTS (WCDMA)	384Kbps-2Mbps
CDMA-One	9.6Kbps-76.8Kpbs
CDMA 2000 1x EV-DO	384Kbps – 2.4Mbps
CDMA 2000 1x EV-DV	3.09Mbps
CDMA 2000 3x	2Mbps – 4Mbps
HSDPA	8Mbps – 10Mbps
LTE	le 1 Gbps, fel 500 Mbps.

2.1. táblázat. Technológiák és az elérhető adatráták

A hálózat további fontos tulajdonságai amik mérésre kerülnek a jelerősség és UDP tesztek esetén a jitter. A jelerősség - amit dBm-ben mérnek - indikálja a szolgáltatás minőségét. Az értéke sok paramétertől függ, többek között az adótoronytól való távolságtól, a természeti akadályoktól, amik a készülék és a torony között vannak (erdő, hegy, stb.) és az időjárástól is függhet. A telefonkészüléken 5 darab kis osztás szokta jelezni a jelerősséget. Minél több osztás aktív, annál erősebb a jel, annál jobb a szolgáltatás. Ha kicsi a jel erősség, akkor probléma léphet fel a szolgáltatások minőségében, például akadozik a hangátvitel, nagyon ingadozik az átviteli sebesség. A ritkán lakott területeken is kicsi szokott lenni a jelerősség. A mérésekből kiderülhet, hogy olyan helyen is rossz a jel minősége, ahol ezt nem is várnánk. Ilyen esetben az operátorok feladata, hogy kiderítsék ennek az okát.

A jitter a csomagok megérkezési ideje közötti különbség, ami a hálózati torlódásból, időzítési problémából vagy útvonal váltásból eredhet. Ez akkor jó, ha kicsi az érték, azaz a csomagok sűrűn követik egymást és a csomagok beérkezési idejei között nem nagy a különbség. A jitter nagysága, illetve nagy, sűrű változása okozhatja a telefonbeszélgetések közbeni hangkiesést, akadozást, illetve a videolejátszás során a képek kiesését, szaggatást. Az UDP-t használó tesztek során a jelentések tartalmazzák az aktuális jitter értéket. Ezen értékeket figyelve derülhet ki a magas jitter érték, illetve a gyakori, nagy különbségek is problémát jelezhetnek a hálózatban.

2.2. Android platform bemutatása

Az Android fejlesztő környezet széleskörű támogatást nyújt a fejlesztőknek a telefon és a hálózat adatainak lekéréséhez, ami nagyban megkönnyíti az alkalmazás

fejlesztését. A nyílt fejlesztői környezettel az Android lehetőséget biztosít a fejlesztőknek, hogy magas színvonalú és innovatív alkalmazásokat készítsenek. A fejlesztők kihasználhatják például az adott eszköz hardver képességeit, hozzáférhetnek a felhasználó adataihoz és akár futtathatnak háttér szolgáltatásokat is.

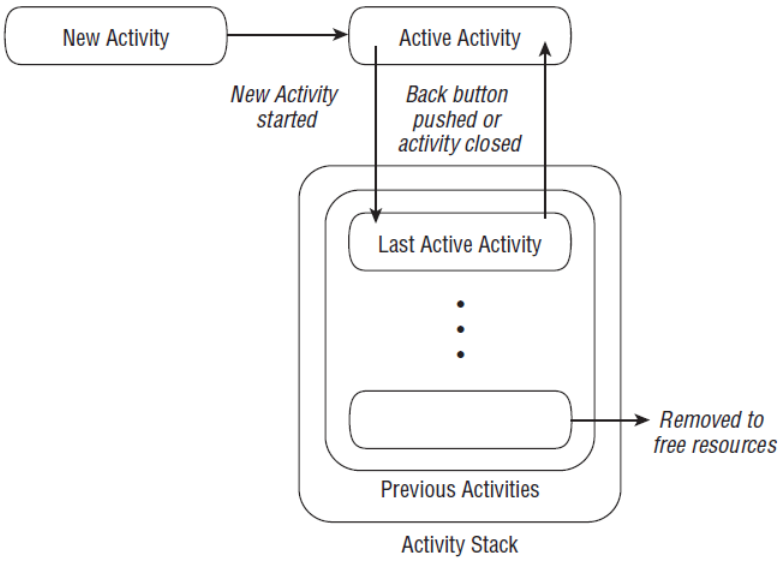
2.2.1. Android komponensek

Az Android fejlesztőkörnyezetnek több komponense létezik, mindegyiknek különböző célja és szerepe van a programban. Ezek általában egymással szoros kapcsolatban állnak, de önálló példányban léteznek és saját életciklusuk van.

Activity

Egy felhasználói felülettel rendelkező képernyőt, ablakot Activity-nek nevezünk. Például egy zenelejátszó program Activity-je lehet, ami megjeleníti a lejátszó felületet, és egy másik, ahol kiválaszthatja a felhasználó a lejátszandó zenéket. Minden alkalmazásnak van egy kitüntetett "fő” Activity-je, ha a felhasználó elindítja a programot ez az ablak jelenik meg. Habár mindegyik Activity a zenelejátszó programban együttműködve adja a teljes felhasználói élményt, mégis mindegyik független a többitől.

A hagyományos applikációkkal ellentétben az Android programoknak nincs kontrolljuk az életciklusuk felett. Így az alkalmazás komponenseinek figyelnie kell a program állapotváltozására és aszerint reagálni. A komponensek állapota befolyásolja a program állapotát, mely meghatározza a program prioritását a rendszerben. Az Activity-k úgynevezett "Activity stack”-en, azaz egy Last-In-First-Out adatszerkezetben tárolódnak. Mikor egy új Activity elindul, az előtte futó kerül a stack tetejére. Ha az előtérben lévő, aktív Activity bezáródik vagy a felhasználó megnyomja a Vissza gombot, akkor a verem tetején lévő kerül előtérbe. Ha az Activity-t megsemmisíti a rendszer, mert már nincs rá szükség, akkor felszabadítja annak erőforrásait és eltávolítja a veremből. A folyamatot az alábbi kép szemlélteti.



2.2.1. ábra. Activity-ket tartalmazó verem

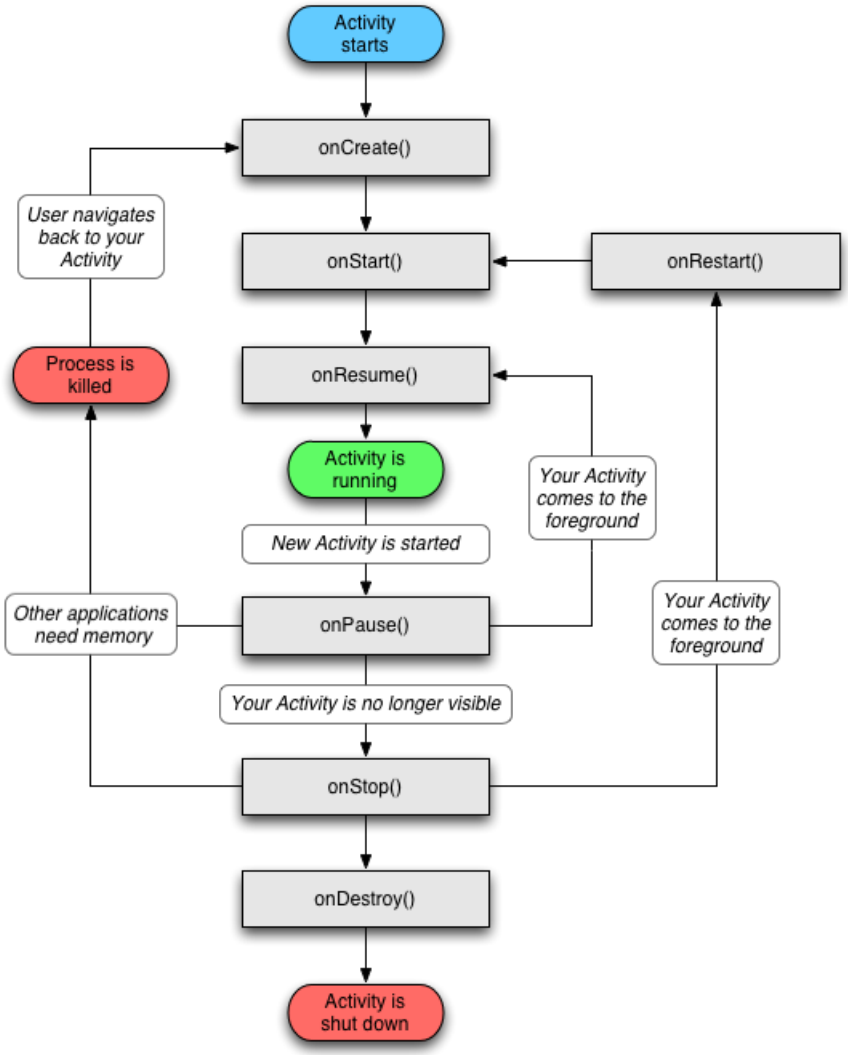
Az Activity-nek négy állapota lehet:

- **Aktív** (Active), ebben az állapotban a verem tetején lévő Activity lehet, ilyenkor az ablak előtérben van és képes fogadni a felhasználói utasításokat. Az Android rendszer mindent megtesz annak érdekében, hogy ez az Activity futhasson. Így, ha kevés az erőforrás, a memória, akkor leállítja a verem alján lévőket és felszabadítja azok erőforrását. Mikor egy másik Activity lesz az aktív, akkor az előbbi állapota felfüggesztett lesz.

- **Felfüggesztett** (Paused) állapotban az Activity látható maradhat, de nem reagál felhasználói eseményekre, nincs fókusza. Ez akkor fordulhat elő, ha az előtérben lévő ablak átlátszó vagy nem fedi le a teljes képernyőt.

- **Leállt** (Stopped), ekkor az Activity nem látható, leállt. Az erőforrások még nem szabadulnak fel és a veremből sem kerül ki. A leállt Activity-k prioritása a legkisebb, így ezeket terminálja legelőször a rendszer, ha erőforrásra van szüksége.

- **Inaktív** (Inactive) állapotba kerül, ha terminálva lett vagy mielőtt elindul. Ezek az Activity-k nincsenek a veremben és csak újraindításkor válhat aktívvá.



2.2.2. ábra. Activity életciklusa

Android rendszer biztosítja, hogy az egyes Activity-k reagálhassanak az állapotváltásra. A lenti képen láthatóak az Activity életciklusai. A téglalapok visszahívó (callback) metódusokat szimbolizálnak. Ezek felhasználhatók különféle műveletek végrehajtására az egyes átmenetek során.

- **onCreate()** Az Activity létrejöttekor hívódik meg. Itt kell az összes statikus adatot inicializálni, például nézetek létrehozása. Ez a függvény kap egy Bundle típusú paramétert, mely az Activity előző állapotát tartalmazza. Mindenképp onStart() metódus hívás követi.

- **onRestart()** egy leállított Activity újraindítása során hívódik meg. OnStart() metódus követi.

- **onStart()** az előtt hívódik, hogy az Activity láthatóvá válna a felhasználó számára. OnResume() metódus hívódik ezután, ha az Activity előtérbe kerül vagy az onStop(), ha háttérbe kerül.
- **onResume()** akkor hívódik, ha az Activity a verem tetejére kerül, mielőtt aktívvá válna és felhasználói inputot fogadna. Minden esetben onPause() hívódik utána.
- **onPause()** egy másik alkalmazás folytatása váltja ki ezt a metódushívást. Itt célszerű elmenteni az adatokat, illetve leállítani az erőforrás igényes műveleteket, mint például az animáció lejátszást.
- **onStop()** az Activity leállásakor vagy egy másik Activity aktívvá válásakor hívódik. Ezután onRestart() hívás jelzi, hogy újra aktívvá vált az Activity.
- **onDestroy()** az Activity megszűnésekor kerül meghívásra, amit vagy a rendszer vált ki, mert erőforrásra van szüksége vagy meghívták rá a *finish()* metódust.

Service

A Service (szolgáltatás) egy háttérben, párhuzamosan futó komponens. Sokáig futó műveletek végrehajtására használható és nem biztosít felhasználói felületet. Ilyen művelet lehet például a zenelejátszás, miközben a felhasználó más alkalmazást használ, anélkül, hogy a háttérben futó zenelejátszás blokkolná vagy zavarná a másik alkalmazást, illetve a felhasználót. Egy másik komponens, mint például egy Activity indíthat Service-t és interakcióba léphet vele. Egy új service-t indíthat egy komponens a *startService()* metódus hívással, ami akkor is fut, ha a hívó komponens megszűnik. A service automatikusan leáll, ha a feladatát befejezte. Ez a feladat lehet például egy fájl feltöltése. Másik működési módja a service-nek az úgynevezett kötött (bound) service. Egy komponens a *bindService()* metódussal tud összeköttetésbe kerülni ezzel a service-szel. Több komponens is kapcsolódhat egy service-hez, ha már egy komponens se kapcsolódik a szolgáltatáshoz, akkor megszűnik a service. Fontos callback metódusok, melyek implementálhatók egy service-hez:

- **onStartCommand()** akkor hívódik meg, mikor egy másik komponens meghívja a *startService()* metódust. Ha ez a metódus implementálásra kerül és befejezte a működését, akkor le kell állítani a *stopSelf()* vagy *stopService()* metódussal. Ha csak kötött szolgáltatásként szeretnénk használni, akkor nem kell implementálni ezt a metódust.

- **onBind()** metódus hívódik meg, ha egy másik komponens akar kapcsolódni a service-hez a *bindService()* függvény hívással. Ha nem akarjuk megengedni a szolgáltatáshoz való csatlakozást akkor null-t kell küldeni visszatérő paraméterként, egyébként pedig egy IBinder típusú objektumot, amin keresztül a hívó komponens kommunikálhat a szolgáltatással.

- **onCreate()** a service létrehozásakor hívódik az *onStartCommand()* előtt.

- **onDestroy()** a rendszer hívja ezt a metódust, mikor a szolgáltatást már nem használják többé és meg kell szüntetni. Itt kell felszabadítani a foglalt erőforrásokat, szálakat.

IntentService

Az IntentService a Service osztályból öröklődik és aszinkron kéréseket kezel (Intent-eken keresztül). A kliens küld egy kérést a startService(Intent) híváson keresztül és a szolgáltatás elindul, majd leállítja magát ha végrehajtotta a feladatát. Egyszerre csak egy kérést szolgál ki, ha több érkezik be, akkor egymás után lesznek kiszolgálva. Ez a módszer gyakran használt eljárás annak érdekében, hogy tehermentesítsük az applikáció fő szálát. A használatához az IntentService osztályból kell származtatni és implementálni az *onHandleIntent(Intent)* metódust. Az IntentService meg fogja kapni ezt az Intent-et, elindítja az új szálat és megállítja szolgáltatást.

Broadcast receiver

A broadcast receiver egy olyan komponens, amely válaszol a rendszerszintű üzenetekre. Számos rendszerüzenetet kezelhet például a képernyő kikapcsolásról vagy az akkumulátor töltöttségének csökkenéséről érkező üzeneteket. Az alkalmazás is kezdeményezhet ilyen üzenet küldést, például ha egy másik alkalmazásnak akar üzenetet küldeni. A service-hez hasonlóan nem rendelkezik felhasználói felülettel.

Az egyes komponenseket egy aszinkron üzenettel lehet aktiválni, amit Intent-nek neveznek. Ez kapcsol össze komponenseket futási időben, akkor is, ha a komponens egy másik applikációhoz tartozik. Az Intent közvetíthet üzenetet másik komponensnek, például, hogy jelenítsen meg egy képet vagy nyisson meg egy weboldalt.

Intent

Az intent üzenetekkel futás idejű kapcsolatot, kötést létesíthetünk Android komponensek között. Intent objektum egy adatstruktúra, ami általában a végrehajtandó művelet absztrakt leírását tartalmazza, illetve a fogadó komponens számára fontos információkat is tartalmazhat. Ezen felül tartalmaz a rendszer számára lényeges információkat például: a fogadó komponens kategóriáját.

Application

Ezen osztály segítségével lehet figyelni, karbantartani az egész applikáció állapotát. A saját application objektumunkat az AndroidManifest.xml fájl <application> tagja írja le, ami azt jelenti, hogy az osztály példánya létrejön az applikáció indulása elején. Általában nincs szükség erre az objektumra. A többi Android objektumból elérhető az Application objektum a *getApplication()* metódus segítségével.

Ha szükség van egy globális Context objektumra (például BroadcastReceiver objektum beregisztrálásához), itt található a *getApplicationContext()* függvény, mely visszaadja ezt a Context objektumot.

2.2.2. Futási környezet és konfiguráció

Az Android SDK tartalmaz egy alap könyvtárat, mely megvalósítja a Java programozási nyelv túlnyomó részét.

Minden alkalmazás saját szálon fut és saját Dalvik virtuális gép példánnyal rendelkezik, így egymástól elkülönítve tudnak futni az egyes alkalmazások. A Dalvik VM (virtuális gép) a Linux kernelre támaszkodik az alacsony szintű funkcionalitás használata során, mint például a szálkezelés. A Linux továbbá biztonsági, memória és hálózat kezelési szolgáltatásokat is nyújt.

Az Android operációs rendszer megvalósítja a *legkisebb jogosultság elvét,* így az alkalmazás csak a számára szükséges komponensekhez fér hozzá. Egy alkalmazás engedélyt kérhet arra, hogy komponensekhez, adatokhoz férhessen hozzá például: névjegyzék, tárolók (SD kártya), kamera, stb. Minden engedélyt a felhasználónak kell jóváhagynia telepítéskor.

AndroidManifest.xml

Az Android rendszer számára szükséges konfigurációs paramétereket, illetve az alkalmazás által használt komponensek deklarációját tartalmazza. Tartalmazza

továbbá a felhasználói jogosultságok listáját, deklarálja az alkalmazás futtatásához szükséges minimum Android API szint számát, illetve egyéb hardver és szoftver paramétereket. Azok a komponensek, amik nincsenek deklarálva ebben a fájlban, nem láthatóak a rendszer számára. Ez a fájl az alkalmazás gyökérkönyvtárában foglal helyet.

Így néz ki egy Activity deklarációja a manifest fájlban:

```
<manifest xmlns:android="http://schemas... >
 <application android:name=".DriveTestApp" ...>
   <activity android:name=".TestActivity"
     android:label="@string/title_activity_drive_test" ...>
    <activity android:name=".MainActivity">
     <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER"/>
     </intent-filter>
   </activity>
    ...
 </application>
</manifest>
```

Az <activity> elemben az android:name paraméter tartalmazza az Activity osztályból származó osztály nevét, melyet az alkalmazásban használunk. Az android:label paraméterben megadhatjuk a felhasználó számára megjelenő nevet.

A fenti példában látható <intent-filter> rész utasítja a rendszert, hogy a MainActivity nevű acvtivity objektumot töltse be az alkalmazás elindulása után.

Komponensek deklarációja a manifest fájlban a következők lehetnek:

- <activity> elemek az Activity-ból származó osztályokhoz
- <service> elemek a Service illetve IntentService típusú osztályokhoz
- <receiver> elemek a BroadcastReceiver osztályokhoz
- <provider> elemek a tartalomszolgáltató osztályokhoz

Adattárolás

Az Android számos lehetőséget kínál az adatok perzisztens tárolására. Ezek lehetnek megosztott preferenciák (Shared preference), mely primitív adatokat tárol kulcs-érték párok alapján, külső illetve belső tároló, illetve SQLite adatbázis.

Megosztott preferenciák segítségével tárolhatjuk el egyszerűen az alkalmazás, illetve az egyes Activity objektumok állapotát, mivel az adatok megmaradnak az alkalmazás megszűnése után is. Kulcs-érték párok segítségével el tudjuk menteni az objektumok változóinak értékét, majd visszaolvasni, ha szükséges. Az eltárolt attribútumok float, int, long, boolean és string típusúak lehetnek. A *getSharedPreferences()* metódus segítségével névvel ellátott preferenciát kapunk, melyet első paraméterként kell megadnunk, és arra hivatkozva bárhonnan el tudjuk érni az alkalmzásból. A *getPreferences()* függvény segítségével az Activity objektumokhoz rendelhetünk egy preferenciát, amiben az adott tevékenység objektum adatait tárolhatjuk.

Belső tár segítségével olyan fájlokat hozhatunk létre a telefonkészülék belső, nem hordozható memóriájában, amikhez csak az alkalmazás férhet hozzá és annak eltávolításakor ezek a fájlok automatikusan törlődnek.

A **külső tár** lehet a telefonhoz csatolt SD kártya, azaz hordozható memória egység, melyen olyan fájlokat tárolhatunk, melyekhez a felhasználó is hozzáférhet.

Android alkalmazásokban SQLite **adatbázisok**at hozhatunk létre, melyek név szerint minden komponensből elérhetőek, de más applikációból már nem. Az SQLite támogatja a hagyományos relációs adatbázis funkciókat, mint az SQL szintaxis és tranzakciók. A futás során az adatbázisnak kevés memóriára van szüksége (kb. 250 KByte). Az adatbázisműveletek fájl műveleteket igényelnek, amik lassúak lehetnek, ezért aszinkron módon kell azokat végrehajtani. Adatbázis kezelés megvalósításához szükség lesz egy osztályra, mely az *SQLiteOpenHelper* osztályból származik. Ennek az *onCreate()* függvényét kell felüldefiniálni az adatbázist létrehozó implementációval, melyet SQL paranccsal adhatunk meg. Az *onUpgrade()* metódus felüldefiniálásával megadhatjuk hogy a programunk milyen utasításokat hajtson végre, ha a tábla korábbi verziója már létezik az adott eszközön. A helper osztályunk *getReadableDatabase()* és a *getWriteableDatabase()* függvényeinek segítségével kaphatunk egy SQLiteDatabase objektumot, melyen keresztül hozzáférhetünk az adatbázis táblákhoz és módosíthatjuk azokat. SQLiteDatabase *query()* és *rawquary()* metódusaival futtathatunk lekérdezéseket. Minden lekérdezés egy Cursor objektumot ad vissza, mely segítségével manőverezhetünk a lekérdezés eredményének sorai között.

```
public List<DbData> queryAll() {
List<DbData> dataList = new ArrayList<DbData>();
Cursor cursor = db.rawQuery("SELECT * FROM "+DB_TABLE , null);
if (cursor != null) {
    cursor.moveToFirst();
while (cursor.isAfterLast() == false) {
    DbData data = cursorToData(cursor);
    dataList.add(data);
    cursor.moveToNext();
    }
cursor.close();
}
return dataList;
}
```

2.2.3. Hálózati protokollok a programban

UDP (User Datagram Protocol): kapcsolat nélküli, megbízhatatlan csomagtovábbítást nyújtó protokoll. Portszámot és checksum-ot is tartalmaz egy csomag. A csomagok nem biztos, hogy ugyanabban a sorrendben érkeznek meg a fogadóhoz, mint amilyen sorrendben el lettek küldve. A küldő nem kap visszaigazolást a csomag megérkezéséről. Ez a protokoll nagyon elterjedt a videojátékoknál és a telekommunikációban.

TCP (Transmission Control Protocol): megbízható de ezért lassabb is, mint például az UDP. Folyamként vagy fájlként kezeli a kapcsolatokat. A csomagok megfelelő sorrendjéről, illetve az elveszett csomagok újra küldéséről is gondoskodik. A csomagok pufferrelésével eléri, hogy az átviteli sebesség is közel állandó legyen az adat küldése, továbbítása során.

HTTP (Hypertext Transfer Protocol): a web hálózati protokollja, fájlok és más adatok - azaz erőforrások - továbbítására használják. Ezek az erőforrások lehetnek HTML fájlok, képek, lekérdezési eredmények, stb. Használhat TCP és UDP protokollokat is. HTTP a kliens-szerver modellt használja, vagyis a HTTP kliens megnyitja a kapcsolatot és küld egy kérést a HTTP szervernek. Amint a szerver visszaküldte a választ lezárja a kapcsolatot. A HTTP üzenet felépítését tekintve egy kérés vagy válasz sorral kezdődik pl.: ”GET /path/to/file/index.html HTTP/1.0” vagy ”HTTP/1.0 200 OK”. A GET

HTTP metódussal kérhetünk el egy erőforrást a szervertől, további metódusok még a POST és a HEAD. Ez után az erőforrás neve, címe szerepel majd a HTTP verzió. Válasz esetében csak a HTTP verzió és a státusz kód szerepel. Ezek után szerepelhetnek paraméterek ”PARAMÉTER: érték” alakban. Itt szerepelhet a szerver, illetve a kliens számra fontos információk, mint például az erőforrás módosításának ideje: ”Last-Modified: Fri, 31 Dec 1999 23:59:59 GMT”. Válasz üzenetek esetén a kért erőforrás az üzenet végén szerepel. Ebben az esetben a Paraméterek között szerepel az üzenet hossza és típusa.

HTTP* egy specializált HTTP protokoll, melyet ebben a programban használok az Android applikáció és a szerveren futó alkalmazás kommunikációjában. A kommunikációt mindenképpen a mobil alkalmazás kezdeményezi egy "GET” üzenet küldésével, mely ilyen adatokat tartalmaz: GET / HTTP*/1.0\n REPORTPERIOD: <jelentésküldés ideje>\nMODE: <DL vagy UL>\n CONNECTION: <TCP vagy UDP>\n Unit: <1, 2 vagy 3>\n BUFFERSIZE: <buffer méret>\n. Ez az üzenet tartalmazza a teszt paramétereit, amiket a felhasználó adott meg. A szerver oldali program ezek alapján indítja el a megfelelő küldő vagy fogadó szálat. A program a teszt leállításakor egy ”STOP” üzenetet küld a szervernek, mely ilyen alakú: "STOP / HTTP*/1.0\n". Ennek hatására a szerver leállítja a teszt szálat.

3. fejezet

Felhasználói dokumentáció

3.1. Program bemutatása

A Drive Testing applikáció egy Android készüléken futtatható program, mely a telefon hálózatról tud különféle statisztikai adatokat gyűjteni. Ehhez tartozik egy szerver oldali program is, mellyel a mobil alkalmazás a mérések során aktívan kommunikál. Az alkalmazással futtatható tesztek adatokat gyűjtenek a telefonhálózatról és a készülékről a szerver programmal történő kommunikáció során, melyek fontos információkkal szolgálhatnak a telefonhálózatot üzemeltető operátoroknak. A tesztelő akár menet közben, autóban ülve tudja a teszteket futtatni. A mérések során az adatok grafikus térképen is megtekinthetőek és a futtatás végeztével az eredmények ki is exportálhatóak az adatbázisból CSV formátumban egy tetszőleges fájlba.

3.1.1. Program használatának feltételei

Hardver feltételek

Az alkalmazás futtatásához szükséges egy legalább 4.0.4 verziójú Android operációs rendszerű telefonkészülék. Korábbi verziójú rendszeren lehetséges, hogy egyes funkciók nem működnek megfelelően, illetve el se indul a program.

A mobilkészüléken be kell kapcsolni a mobilhálózaton keresztüli adatelérést (Beállítások -> Vezeték nélküli és mobilhálózatok -> Mobilhálózatok -> Adatok engedélyezve jelölőnégyzetet kell bepipálni), illetve a GPS pozíció elérését a Helyszolgáltatások menüpontban a GPS-műholdak jelölőnégyzet segítségével aktiválhatjuk.

Szükséges továbbá egy asztali számítógép, melyen található JVM (Java virtuális gép) a szerver oldali program futtatásához. Ennek internet hozzáférést kell

biztosítani.

Szoftver feltételek:

A szerver oldali program fordításához és futtatásához szükséges a JDK (Java Developer Kit) legalább 1.6-os verziója, ami itt elérhető: [JDK]

Valamint az Apache ant, amit innen tölthetünk le: [ant].

A feltelepített Drive Test programban be kell állítani a szerver oldali számítógép IP címét, ha router illetve egyéb más hálózati eszköz is érintett a kommunikációban, úgy biztosítani kell, hogy az applikációk kommunikálni tudjanak a 4500-as porton keresztül. Továbbá a programok használhatják még az 5000, illetve a 5500 és 5600 közötti portokat a kommunikáció során.

Az alábbi lista tartalmazza a jogokat (permission), amit a mobilalkalmazás használ:

access coarse location: az alkalmazásnak szüksége van erre ahhoz, hogy megjeleníthessük a készülék helyzetét a térképen és mérhessük a jelerősséget. Ezt a hálózati pozíciót gyorsan el tudja nekünk küldeni az adott hálózati eszköz, de elég durva becslést ad a telefon helyére vonatkozólag, körülbelül 100 méteres pontossággal. Ezt akkor használjuk, amikor a GPS szolgáltatás nem elérhető, például beltérben.

access fine location (GPS): a szabadban megfelelően erős a GPS műhold jele ahhoz, hogy pontosan meghatározzuk a telefon helyzetét, ehhez a szolgáltatáshoz szükséges ez a jogosultság.

internet, access network state, read phone state: a hálózat állapotának figyelésével kapunk értesítést a jel erősség változásról. Internet hozzáférés szükséges a szerverrel való kommunikációhoz és a térkép frissítéséhez. A mérések során a telefon és a hálózat fontos paraméterei is rögzítésre kerülnek, kivéve a kritikus, azonosításra alkalmas paraméterek, mint például az IMEI/ESN szám, telefon szám, SIM széria szám vagy a MAC cím. Ezeket nem rögzíti az alkalmazás, csupán megjeleníti.

read and write external storage: külső tár írása és olvasása szükséges a jelentések, logok kiírásához. A felhesználó innen tudja lementeni a mérési eredményeket USB-n keresztül.

3.2. Program használatának bemutatása

3.2.1. Program telepítése

Szerver oldali program futtatása

A program fordításához és futtatásához szükséges legalább a JDK 1.6 verziója. A **HttpServer** mappában található ant *build.xml* segítségével könnyedén fordíthatjuk és futtathatjuk az alkalmazást:

- Egy tetszőleg terminálban menjünk a HttpServer könyvtárba.
- Ezután az Apache ant program segítségével tudjuk fordítani az alkalmazást. Ezt a lépést csak egyszer kell végrehajtani, későbbiekben elég lesz csak a futtatásnál leírtakat elvégezni. A terminálban az *ant* parancsot kell kiadni és a program fordítása elkezdődik.
- Majd a sikeres fordítás végeztével az *ant run* paranccsal futtathatjuk a programot.
- A program indulása után ezt az üzenetet fogja látni:

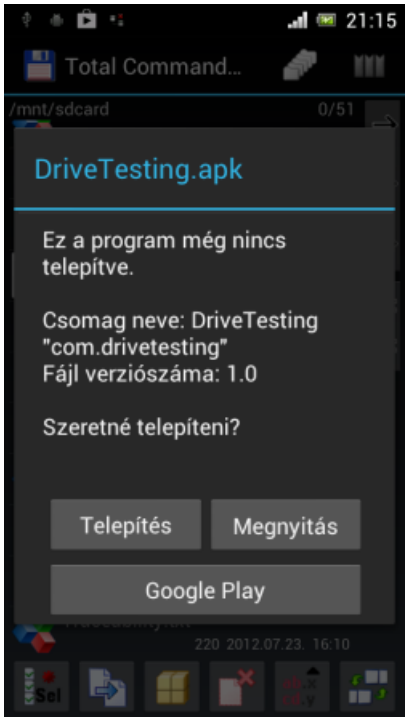
```
d:\HttpServer>ant run
Buildfile: d:\HttpServer\build.xml
run:
[java] Log file: d:\HttpServer\server.log
[java] Log fileWriter created
[java] HTTP_Server: Waiting for connection on port:4500
[java]
```

Telepítés a mobileszközre

A telepítéshez szükséges a DriveTesting.apk fájl és hogy a mobilkészülék csatlakoztatva legyen a számítógéphez és hozzáférésünk legyen a telefon SD kártyájához. A telepítés menete:

1. A külső program telepítéséhez kell egy fájlkezelő program, mint a Total Commander vagy az Apps Installer nevű program, amivel tudunk manuálisan telepíteni.

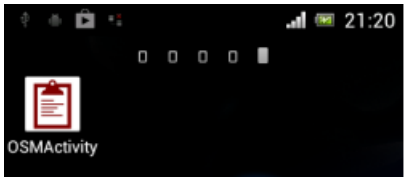
2. Engedélyezni kell a telefonon az ismeretlen forrásból való telepítést. Ehhez menjünk a ”Beállítások” menüben, ott válasszuk az ”Biztonság" részt. Majd pipáljuk be az “Ismeretlen források” opciót. Ezek után már képesek leszünk külső programokat is telepíteni.



3.2.1. ábra. Telepítés

3. Kapcsoljuk össze a telefont és a számítógépet USB kábellel. A telefonon az értesítési területen válasszuk ki az USB részt, és csatlakoztassuk a memóriakártyát. Ekkor a telefon leválasztja az SD kártyát, és lehetővé teszi hogy a számítógépünkkel elérjük azt.

4. Másoljuk fel a programot egy tetszőleges mappába, majd válasszuk le az SD kártyát a PC-ről, és csatlakoztassuk a telefonon. Indítsuk el a fájlkezelő programot, és keressük meg a megfelelő fájlt. Válasszuk ki az apk fájlt, ekkor feljön egy opció, hogy megnyitjuk a fájlt, vagy telepítjük. Mi most az utóbbit válasszuk, az ablak a bal oldali képen látható. A következő ablakban átnézhetjük a program legfontosabb tulajdonságait, majd az ”Telepítés” gombra kattintva telepíthetjük.



3.2.2. ábra. Program ikonja

5. A sikeres telepítést követően az alkalmazások között megjelenik a program ikonja, mellyel futtathatjuk azt. Az ikon a bal alsó képen látható.

6. Az indítást követően meg kell adni a programnak a szerver gép publikus IP címét.

3.2.2. Menü áttekintése

A felhasználó a menüpontok segítségével navigálhat az egyes ablakok között. Összesen öt ablakot tartalmaz az alkalmazás. Az applikáció minden ablakában négy menüelem kap helyet.

Széles telefon kijelző esetén - mint amilyennel egy tablet rendelkezik - az összes menüpont megjelenik a képernyő tetején az aktuális ablak nevével együtt.

3.2.3. ábra. Menü tablet-en (4.7”, 1280 * 720)

Keskenyebb kijelző esetén a menüpontoknak csak az ikonja jelenik meg.

3.2.4. ábra. Menü kis kijelzőn a készülék alján jelenik meg (4”, 480*800 felbontás)

Ikonok listája:

 - Main ablak  - Test ablak  - Térkép ablak

 - Export ablak  - Beállítások

3.2.5. ábra. Ikonok

3.2.3. Ablakok leírása

Main (fő) ablak

Az alkalmazás fő ablaka, ez jelenik meg az indítás után. A megjelenített lista két fő részre bontható: az első tartalmazza a telefon adatait (Phone Data), ezek nagy része nem változik, fontos paraméter a jel erősség érték (signal strength), ami a rádió hálózati torony által küldött jel erőssége. A második része a hálózati adatokat tartalmazza (Network Data).

3.2.6. ábra. Main ablak

Ezen adatok nagy része a telefon, illetve a hálózat aktuális állapotától függenek, mint például a hálózat típusa (Network Type), ami lehet UMTS, EDGE, stb. Ez a telefon helyétől függ és persze az adott területen elérhető mobilhálózat szolgáltatásoktól. A mérések szempontjából fontos adatok a már említett szignál erősség, hálózat típusa, ezeken kívül az MCC, MNC, LAC és a CID, ezen értékek változása esetén a képernyő automatikusan frissül a képernyőn is. Utóbbiak azonosítják a telefonoperátort az adott országra vonatkozóan, és meghatározzák a mobilkészülék helyét, azaz, hogy az operátor melyik cellájában tartózkodik. MNC (Mobile Network Code) és az MCC (Mobile Country Code) együtt egyértelműen meghatározza az operátort az adott országban, LAC (Location Area Code) a hálózat szolgáltatójához tartozó területi azonosító, CID (Cel ID) cella azonosító. A lista automatikusan frissül, ha megváltozik egy érték. A felhasználó az ujja segítségével fel és le görgetheti a kijelzőn a táblázatot.

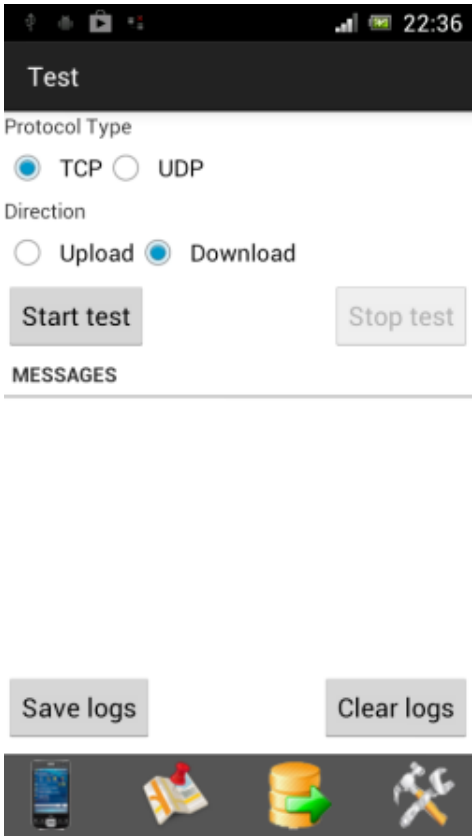
Test (teszt) ablak

Az alkalmazás fő része, ahonnan a felhasználó a teszteket tudja indítani. Egyszerre csak egy mérés futhat, és nem lehet menet közben megváltoztatni a beállításokat. A beállítások a következőek lehetnek: feltöltés vagy letöltés teszt, azon belül UDP vagy TCP protokollok használatát lehet beállítani rádiógombok segítségével. A megadott értékek, beállítások elmentődnek és az ablakok közti váltáskor megőrződnek, illetve visszatöltődnek, ha a felhasználó visszatér ehhez az ablakhoz.

A ”Start test” feliratú gombra kattintva elkezdődik a teszt futtatás. Az előbb említett gomb beszürkül, ilyenkor nem fogadja a felhasználói eseményeket. A ”Stop test” feliratú gomb kivilágosodik és fogadja a kattintás eseményét, ezzel lehet leállítani a futó tesztet. A teszt automatikusan leáll ha nem sikerül elindítani a tesztelést valamilyen beállítási vagy hálózati oknál fogva.

A teszt futtatásához szükséges paramétereket, a távoli szerver IP címét és a jelentés gyakoriságát módosíthatjuk a Settings (Beállitások) ablaknál.

A teszt futása során a felhasználó előre megadott intervallumonként jelentést kap, ami tartalmazza a mérések aktuális eredményét: hálózat típusát, a fel- és letöltési légsebességet, UDP teszt esetén a jittert és a fogadott és az elveszett csomagok számát. Ezt a riportot a képernyő közepén található lista tartalmazza. Ezt a listát törölheti a felhasználó, a ”Clear logs” feliratú gombra kattintva. Az esetleges kapcsolati hibákról is ezen keresztül értesül a felhasználó. Ezen hibák a ”Hibaüzenetek” fejezetben lesznek bővebben kifejtve.



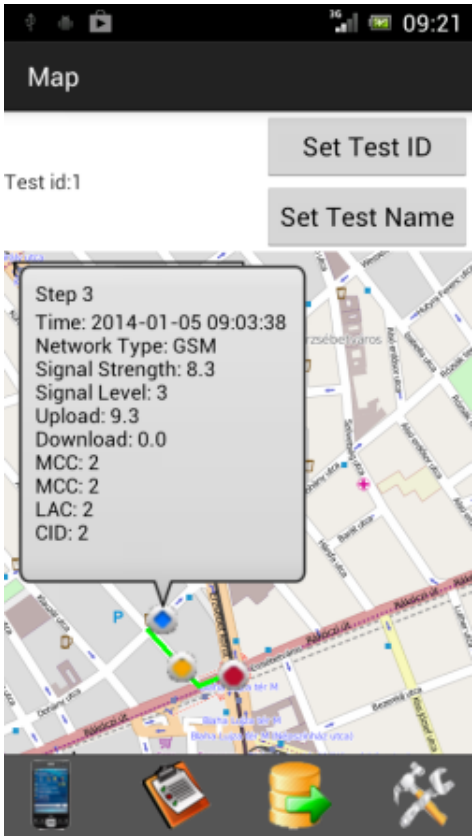
3.2.7. ábra. Test ablak

Az aktuális tartalma ennek a log listának elmenthető a ”Save logs” gomb segítségével. Az elmentett fájl neve az aktuális időbélyegből fog generálódni, például: 201311031130.txt. A megadott beállításoknak megfelelően és a tárolók elérhetőségétől függően a külső vagy belső tárra fog elmentésre kerülni az applikáció könyvtárán belül.

Map (térkép) ablak

A térkép megjelenítéséhez szükséges mobil internet kapcsolat és GPS adat használat engedélyezés a pontos pozíció meghatározására. A felhasználó kiválaszthatja, hogy melyik teszt eredményeit szeretné látni a térképen megjelenítve. Erre szolgál a két oldalsó gomb: a "Set Test ID” és a ”Set Test Name” gombok. A ”Set Test ID” feliratú gombbal a felhasználó egy listából kiválaszthatja, hogy melyik teszt futást szeretné látni, annak azonosítója alapján. A ”Set Test Name” gomb segítségével a teszt elnevezése alapján adhatja meg a látni kívánt tesztet.

Ha a felhasználó az ”ALL” -t választja - akár test id-ként, akár test névként - a térképen az összes pont megjelenik és össze lesznek kötve úttal, akkor is ha valójában nem egy méréshez tartoztak. Ez megkönnyíti a pontok megkeresését a térképen.



3.2.8. ábra. Térkép ablak

Ha éppen fut egy teszt, amikor ez az aktív képernyő, akkor a futó tesztre vonatkozó adatokat láthatja a felhasználó. Újabb jelentések érkezésekor a térkép frissül és az új mérések is láthatóak lesznek.

A térképen a színes pontok mérési pontokat jelölnek, a jelentés rögzítésének idejében az adott helyen volt a mobil készülék. Az egyes pontokra kattintva részletes információt kap a felhasználó egy felugró ablak segítségével. Ez tartalmazza többek között a jelerősséget, a le és feltöltési sebességet és a cella információkat.

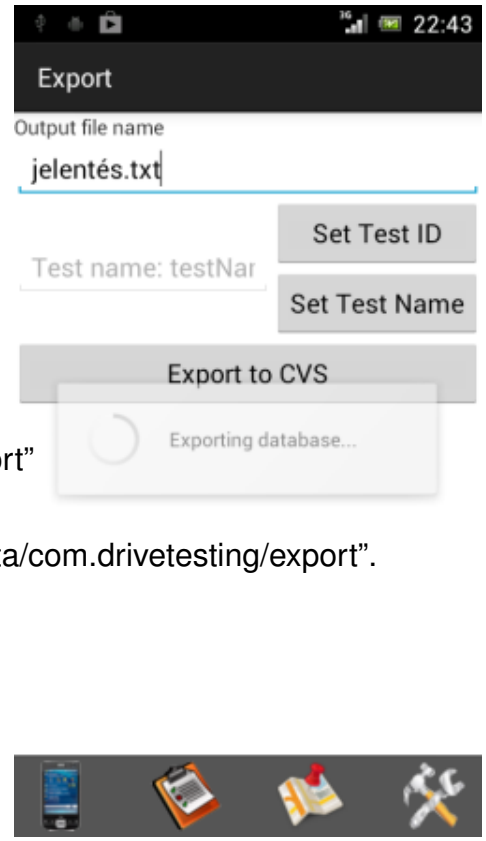
Export ablak

Az ablak tetején található mezőbe írható be a kimeneti fájl neve. Ezt mindenképpen ki kell tölteni.

A program által generált teszteredmények fájlba menthetőek CSV formátumban. A CSV fájl első sora a fejléc: id, test_id, test_name, time, lat, lon, signal_strength, up_speed, down_speed, jitter, lost_packet, sum_packet, mcc, mnc, lac, cid, rate, network_type mezőkkel, ahol az id az adatbázis rekord azonosítója, a test_id a teszt futtatáskor generált teszt azonosító, test_name a teszt neve, amit a felhasználó a beállítások között adott meg, time a bejegyzés ideje, lat avagy latitude szélességi koordináta, a lon azaz longitude a hosszúsági koordinátája a mérés helyének, up_speed és a down_speed a fel- és letöltési sebesség, jitter-nek az UDP protokollt használó tesztek esetén van jelentősége és a csomagok közti késést mutatja, a lost_packet és sum_packet szintén UDP teszt esetén jelentős, az elveszett és az eddigi összes csomag számát jelentik, és a network type pedig a hálózat típusa (UMTS, EDGE, stb.), végül a rate a fel- és letöltési sebesség mértékegységét jelölik (Mbits, Kbits, stb.). Az ez után lévő bejegyzések az adatbázis sorainak megfelelő adatok.

Lehetséges kiválasztani egy teszt ID-t, vagy teszt nevet, melyet ki akarunk exportálni, de akár az összes elemet is kiexportálhatjuk (ALL). Ezt a ”Set Test ID” vagy a

”Set Test Name” nevű gombra kattintva tehetjük meg. Rákattintva a felugró ablakból választhatjuk ki a megfelelő teszt azonosítót. Fontos megjegyezni, hogy csak a program adatbázisában található teszt azonosítók közül választhat a felhasználó. Ha még nem futtatott tesztet, akkor nem lesz bejegyzés és így exportálható adat sem. Továbbiakban meg kell adnunk a fájl nevét az ”Output file name” szövegmezőben, mely a külső memória egységen lesz letárolva ”sdcard/Android/com.drivetesting/files/export” illetve, ha ez nem lehetséges akkor az applikáció könyvtárában ”data/data/com.drivetesting/export”.



3.2.9. ábra. Export ablak

Az ”Export To CVS” feliratú gombra kattintva elkezdődik az adatok kimentése a megadott fájlba és megjelenik egy ablak ”Exporting database...” felirattal. A dialógus ablakban forgó ikon jelzi, hogy még tart az exportálási folyamat.

A művelet befejeztével az ablak automatikusan bezáródik és egy felugró üzenetablakban megjelenik, hogy sikeres volt-e a művelet vagy sem. Sikeres exportálás esetén a fájl elérési útja is megjelenik.

Setting (beállítások) ablak

Az itt megjelenő paraméterek segítségével konfigurálható a teszt futtatás, illetve a program működése. Ezek a paraméterek a program bezárása után is megmaradnak, nem kell a felhasználónak újra beállítania ezeket.

Teszt futtatásra vonatkozó paraméterek:

- Server IP address - szerver publikus IP címe
- Buffer size - egy-egy átküldött adatmennyiség mérete
- Report period - jelentés gyakoriságának ideje
- Test Name - teszt neve

- Units - sebesség és a fogadott adatok mennyiségének mértékegységét állíthatóak be

3.2.4. Program használata és teszt futtatás

A fő ablakban is nyomon lehet követni a jel erősség, hálózat típusának és a cella információk változását, de ilyenkor nem kerül eltárolásra a változás, így nem lehet visszakövetni sem. A teszt ablakban van lehetőség tesztek futtatására. Választhatunk UDP és TCP protokollok, fel- és letöltés között a rádiógombok segítségével. A teszt futtatása előtt győződjön meg róla, hogy a szerver oldali program a HttpServer fut és elérhető a szerver és a fentebb említett portok szabadok. A szerver publikus IP címét be kell állítani a Beállítások menüpontban. A teszt futtatásához a ”Start test” gombra kell kattintani. Ezután elindul az adatok fel vagy letöltése a beállításoknak megfelelően. A teszt addig fut amíg le nem állítják a ”Stop test” feliratú gombbal. Az előre megadott időközönként (report intervallum) a program megvizsgálja és kiírja az aktuális értékeket az adatbázisba és megjelennek a ”Messages” fejlécű listában a teszt ablakban. Ezek a bejegyzések tartalmazzák a különböző mérési eredményeket. A ”Save logs” gomb segítségével elmenthető a Messages üzenet doboz aktuális tartalma. Az elmentett fájl neve és elérési útja egy felugróablakban lesz látható. A térképen látható az épp futó teszt és méréseknek megfelelően frissülnek az adatok rajta. Lehetőség van korábbi tesztek eredményeit is megjeleníteni a térképen teszt azonosító vagy teszt név alapján. Az export ablakban bármely tesztet ki lehet exportálni CSV formátumban az adatbázisból név vagy azonosító alapján. A fájl nevét is megadhatja a felhasználó és az export gomb megnyomásával elindul az exportálási folyamat. Végül egy felugró ablakban jelenik meg az elmentett fájl neve és elérési útja, illetve hiba esetén a hibaüzenet.

3.2.5. Hibaüzenetek

Hibaüzenetek a teszt futtatás során

- GPS aktiválása szükséges a teszt futtatáshoz, ha ezt elmulasztja a felhasználó, akkor egy felugró ablak tájékoztatja erről, amikor a tesztet elindítaná a következő üzenettel: "GPS is not enabled. Do you want to go to settings menu?". A ”Yes” gombra kattintva megjelenik telefon GPS-sel kapcsolatos beállításai, ahol aktiválni tudja a felhasználó ezt a szolgáltatást. A beállítás után a vissza gombbal a felhasználó visszatérhet a program tesztfuttatási ablakába.

- A mobilinternet aktiválása is szükséges a teszt futtatásához, ha ezt elmulasztja a felhasználó, akkor egy felugró ablak tájékoztatja erről, amikor a tesztet elindítaná a következő üzenettel: "Network is not enabled. Do you want to go to settings menu?". A ”Yes” gombra kattintva megjelennek a telefon hálózattal kapcsolatos beállításai, ahol aktiválni tudja a felhasználó ezt a szolgáltatást. A beállítás után a vissza gombbal a felhasználó visszatérhet a program tesztfuttatási ablakába.

- Hibásan megadott szerver IP cím esetén a teszt futása hamar leáll és a következő üzenet olvasható a Teszt ablak üzenetei között: "Error: Cannot connect to server! IP: <megadott IP cím> port: 4500”. Ilyen esetben a felhasználónak ellenőriznie kell a beállítások között a megadott IP cím helyességét, illetve, hogy elérhető-e egy külső hálózati eszközről. Ha a szerver nem közvetlenül kapcsolódik az internethez, akkor a hálózati eszközökben a PortForwarding funkció használatát és beállításait is érdemes ellenőrizni.

- Foglalt teszt port esetén a "Could not connect to server! Test port: <port szám>" üzenetet láthatja a felhasználó. Ez akkor történhet meg, ha egy másik program már használja az adott portot, ebben az esetben próbálkozzon új tesztet indítani, a teszt megpróbál egy másik portot használni. A szerver gépen az 5500 és 5600 közötti portokat használhatja teszt futtatásra a program. Minden teszt indításkor egy új portot fog választani a program.

4. fejezet

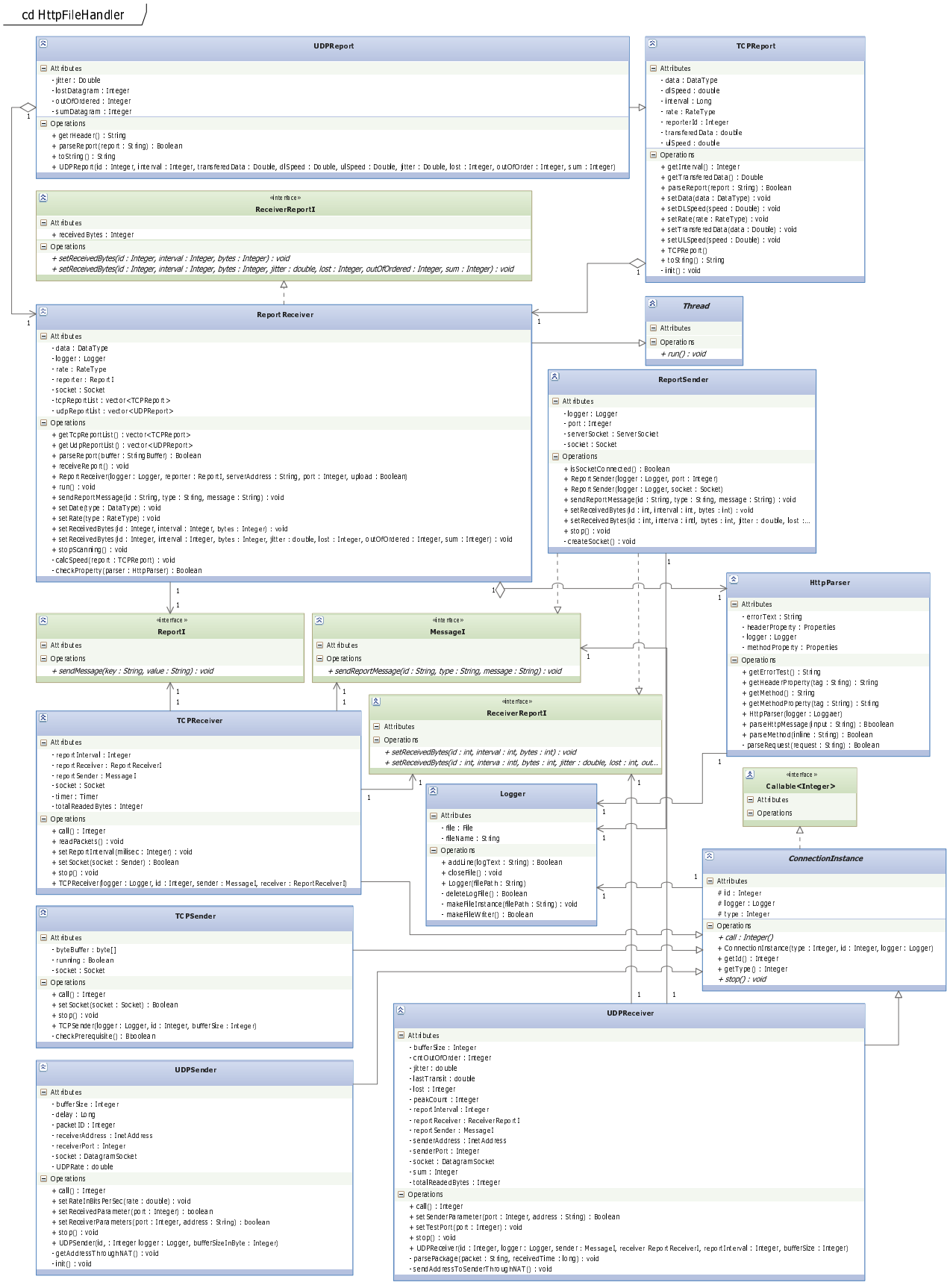
Fejlesztői dokumentáció

4.1. A program architektúrája

A szerveroldali és a mobil alkalmazás is használja a *HttpFileHandler.jar* fájlt, ami tartalmazza a közösen használt osztályokat. Ezen osztályok nagy része a tesztek futtatásában játszik szerepet.

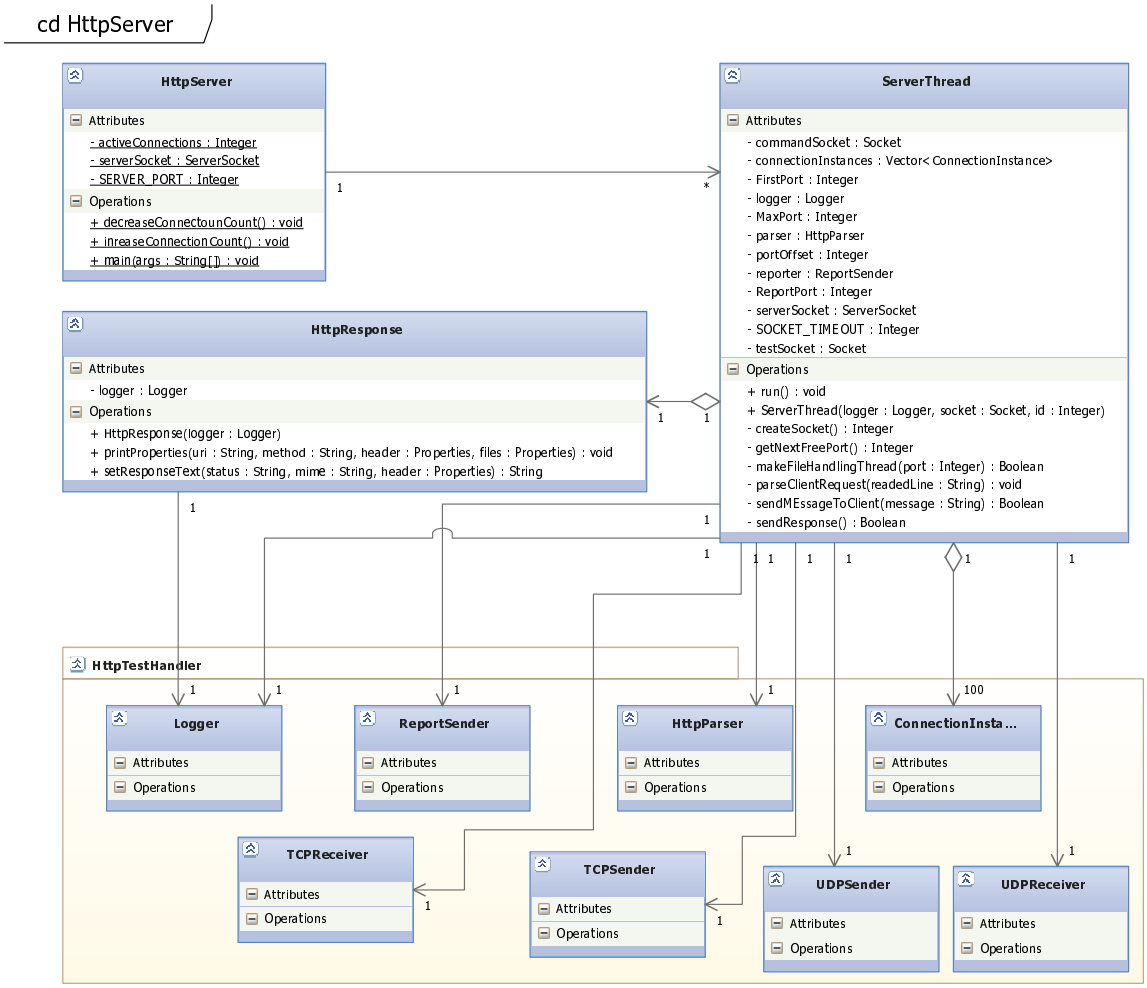
4.1.1. Közös osztályok architektúrája

4.1.1. ábra. HttpTestHandler osztályok



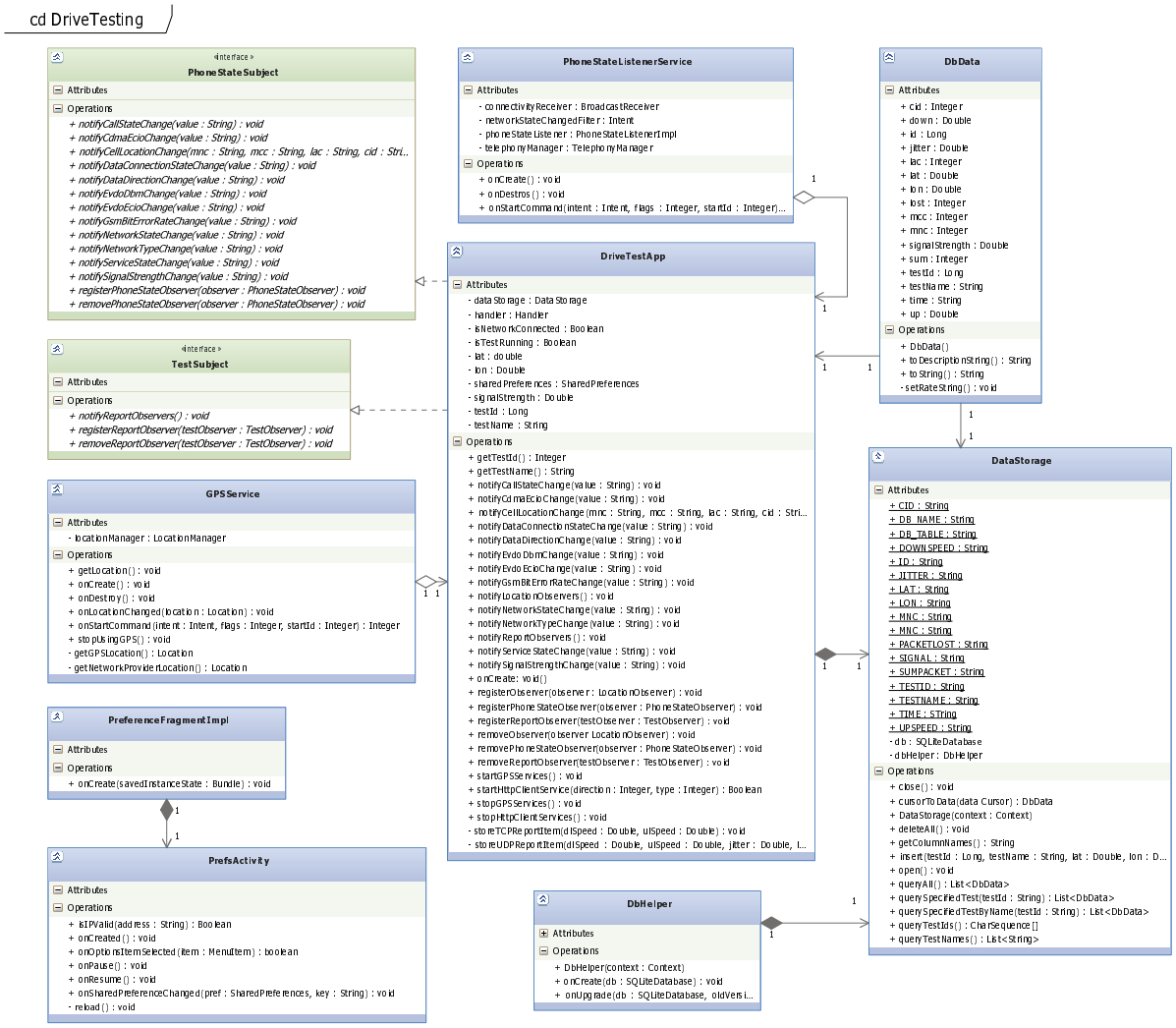
4.1.2. Szerver oldal

4.1.2. ábra. HttpServer osztályai

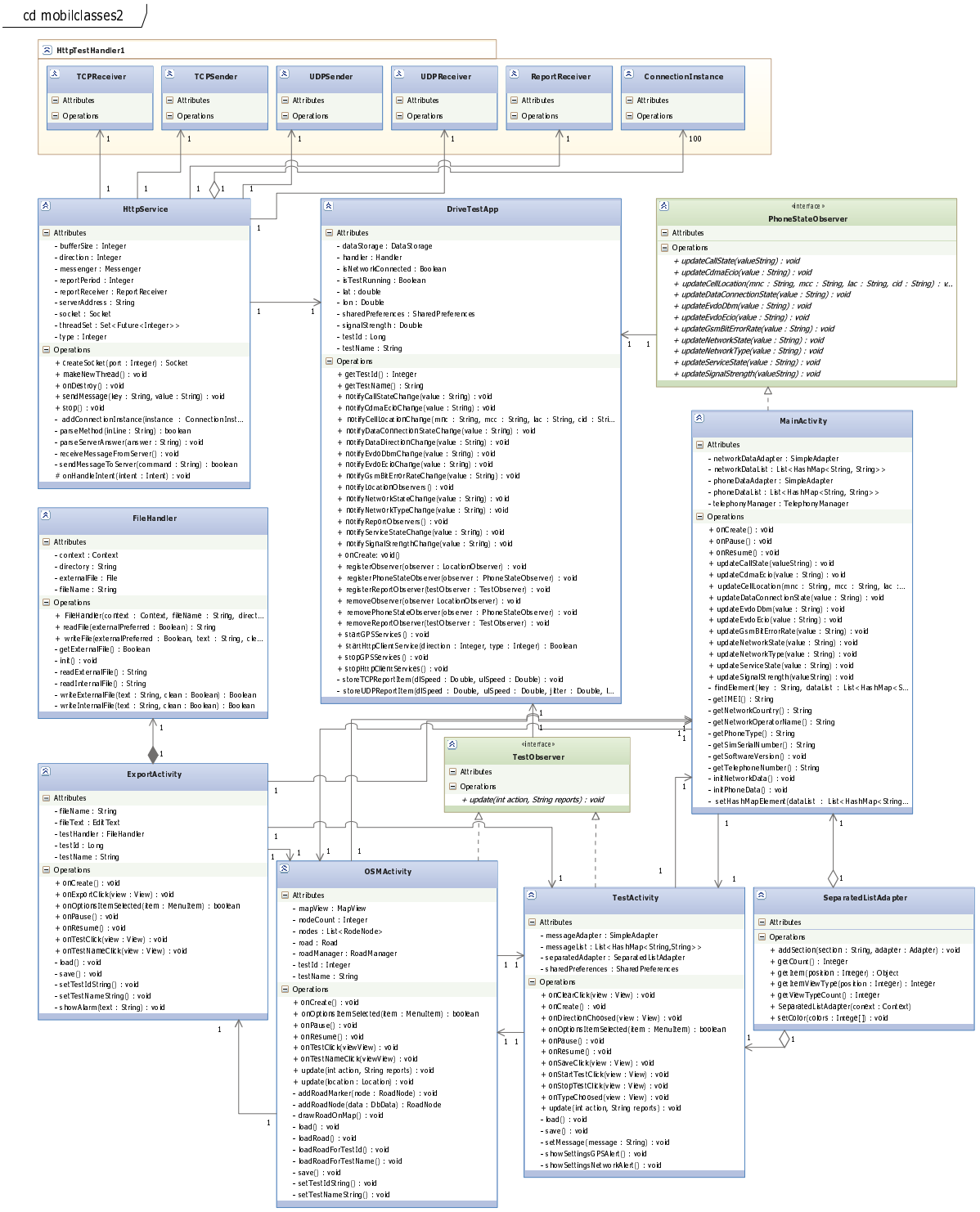


4.1.3. Kliens oldal

4.1.3. ábra. DriveTestApp osztályai 1. rész



4.1.4. ábra. DriveTestApp osztályai 2. rész



4.1.4. Adatbázis leírása

Az applikáció adatbázisa a mobileszközön ebben a fájlban tárolódik:
data/data/com.drivetesting/databases/drive_test_db_v1.
Az adatbázis, melyet drive_test_db_v1 hívnak egy test_data nevű adatbázis táblát tartalmaz. Ezek az attribútumai a táblának:

- "id" - integer típusú érték, a tábla elsődleges kulcsa, automatikusan implementálódik
- "test_id" - integer típusú érték, a teszt azonosítója
- "test_name" - varchar(100) típusú érték, a teszt nevét tartalmazza, amit a felhasználó a beállításoknál megadott
- "time" - varchar(100) típusú érték, a mérés pontos idejét tárolja
- "lat" - varchar(15) típusú érték, a mobilkészülék pozíciójának szélességi értékét tárolja
- "lon" - varchar(15) típusú érték, a mobilkészülék pozíciójának hosszúsági értékét tárolja
- "signal_strenght" - varchar(15) típusú érték, a jel erősség értékét tárolja dBm-ben
- "signal_level" - varchar(15) típusú érték, a jel erősség besorolását tartalmazza (0-4 közötti értéket vehet fel)
- "up_speed" - varchar(15) típusú érték, a feltöltési sebességet tárolja a rate által meghatározott mértékegységben
- "down_speed" - varchar(15) típusú érték, a letöltési sebességet tárolja a rate által meghatározott mértékegységben
- "jitter" - varchar(15) típusú érték, UDP tesztek esetén a jitter értékét tárolja
- "lost_packet" - integer típusú érték, UDP tesztek esetén az elveszett csomagok számát tárolja
- "sum_packet" - integer típusú érték, UDP tesztek esetén az eddig elküldött csomagok számát tárolja
- "mcc” - integer típusú érték, MCC azonosítója
- "mnc" - integer típusú érték, MNC azonosítója

- "lac" - integer típusú érték, LAC azonosítója
- "cid" - integer típusú érték, cella azonosítója
- "network_type" - varchar(15) típusú érték, a mobil hálózat típusát tartalmazza
- "rate" - integer típusú érték, az átviteli sebesség mértékegység azonosítóját tartalmazza

4.1.5. Felhasználó felület, navigálás az ablakok között

Felhasználói felület leírása

Az Android rendszerben az Activity objektumok rendelkeznek felhasználói felülettel, melyek leírását objektumonként külön-külön kell deklarálni XML fájlokban. Ezek a fájlok az alkalmazás könyvtárán belül a *"res/layout"* könyvtárban találhatóak. Egy XML elem neve ebben az esetben egy Java objektum nevét tükrözi, tehát ha az XML elem egy <Button> elem, akkor az alkalmazás egy Button (gomb) objektumot fog létrehozni. Amikor betöltődik egy Activity objektum, akkor a neki megfelelő felület leíró XML fájlból a rendszer megfelelően létrehozza futási időben és inicializálja ezeket az objektumokat.

Példa egy egyszerű függőleges elrendezés egy szövegmezővel és egy gombbal:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas..."
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView android:id="@+id/textView1"
              android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="" />
    <Button android:id="@+id/button1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
</LinearLayout>
```

Ezeket a leíró fájlokat könnyedén kézzel is szerkeszthetjük vagy akár az Eclipse fejlesztőeszközbe ágyazott grafikus szerkesztőfelület segítségével is. Különféle Layout objektumok segítségével meghatározhatjuk az egyes elemek (gombok,

szövegmezők, stb.) elhelyezkedését. A felhasználóval való interakciókra különféle elemek állnak rendelkezésre pl.: gombok, beviteli mezők, dátumválasztók, stb. ezeket widget-eknek nevezzük. A felhasználói felületet (UI - user interface) leíró XML fájlokban hozzá tudunk rendelni különféle elemekhez visszahívó (callback) metódusokat. Ezeket az XML elemek android:on<valami> (pl.: android:onClick=<metódus_név>) paraméterében tudjuk meghatározni. A felhasználói felületet leíró fájlt használó Activity osztályban kell definiálni az XML elemekhez tartozó visszahívó metódusokat, melyek meghívódnak, ha az adott eseményt kiváltja egy felhasználói esemény.

Menü leírása

A menüket külön XML leíró fájl(ok)ban definiálhatjuk, melyekben meg kell határozni az egyes menüpontokat. Ezen leíró fájlok a program könyvtárán belül a ”res/menu” mappában találhatóak. A felhasználó a menün keresztül navigálhat a user interface-szel rendelkező Activity-k között és elérheti a beállításokat. Az Activity osztályokhoz az onCreateoptionMenu metódus megvalósításával tudunk menüt rendelni. Itt kell megadni, hogy mi történjen, ha a felhasználó az egyes menüpontokra kattint. Az Android rendszer automatikusan kirajzolja a menüt a képernyőre. A DriveTest mobilapplikáció minden Activity osztályból származó osztálya rendelkezik grafikus interfésszel és tartalmazza, felüldefiniálja a *onCreateOptionsMenu* és a *onOptionsItemSelected* metódusokat. Ezek segítségével lehet megjeleníteni a menüt és reagálni a menüt érintő felhasználói eseményekre. Az Android 3.0 verziójától a menü egy ActionBar-nak nevezett felületen jelenik meg, a menu.xml fájlban leírtaknak megfelelően. A leíró fájlban a menüelemek *showAsAction* tulajdonságának állításával lehet befolyásolni azok megjelenését: ha ”always” akkor mindig látható az adott elem, ha ”ifRoom”, akkor csak akkor jelenik meg az ActionBar-ban, ha elfér a telefon kijelzőjén. Az ”always” opció esetén akkor is kirajzolódik a menüelem, ha nem fér ki az ActionBar-ban, ilyenkor a képernyő alján jelenik meg, ahogy az ActionBar bevezetése előtt is történt.

4.2. A program megvalósítása

4.2.1. Döntések a megvalósítás során

A fejlesztést megelőző kutatómunka során kiderült, hogy az android SDK nem támogatja számos hálózati berendezés és készülék információ elérését (biztonsági okokból), így nem lehet egyszerűen az Android SDK segítségével úgy hívást kezdeményezni, hogy a hívás végeztével a hívásról adatokat kaphasson a program.

Ehhez alacsony szinten, kernel szinten kellene hozzá férni az adatokhoz. Így a programban csak adathívásokra korlátozódik a tesztelés a tervezett hangívások helyett.

A program Java programozási nyelven készül, mert az Android is erre épül és könnyebb a szerverrel közös részek miatt mind a három részt Java alapokon megvalósítani. Ráadásul a Java készen ad sok programozást segítő könyvtárat, többek között a hálózati kommunikációhoz. Sajnos ebben az esetben a Java.net csomagban lévő osztályok egy nagy részét elfedik a hálózati kommunikációnak, így mikor egy Socket outputstream-jén keresztül egy üzenetet küldök a másik félnek, akkor csak bízni tudok benne, hogy a program valóban egyből kiküldi az üzenetet és nem végez semmiféle tömörítési, egyszerűsítési eljárást a csomagon, ami meghamisíthatná a mérési eredményeket és így a tesztek eredményeit is.

A hálózati forgalom mérésére, figyelésére használtam volna a TrafficStats osztályt, ez egy Android API által biztosított osztály, mely a hálózati forgalomról nyújt információkat a /proc/uid_stat/<uid> könyvtárból. Ez tartalmazza a TCP vagy UDP protokollokon küldött és fogadott byte-ok és csomagok számát. Ha a fájlok nem léteznek aTrafficStats nem tud hálózati statisztikával szolgálni. A getUidTxBytes() és a getUIDRxBytes() jelentések csak a TCP forgalmat tartalmazzák az UDP-re vonatkozót nem. Így az UDP forgalomról az alkalmazás nem kap semmilyen adatot. Ezen és hasonló hibák miatt, nem tudtam érdemben használni a küldött és fogadott csomagok mérésére. Ezért saját csomag figyelő és számoló algoritmust kellett írjak.

4.2.2. Osztályok bemutatása

Közös osztályok

A közösen használt és a teszteléssel összefüggő osztályokat a HttpTestHandler projekt tartalmazza. Ezen osztályok és interfészek tisztán Java alapúak, nem tartalmaznak Android specifikus megoldásokat, ezért használhatóak a szerver oldali programban is.

ConnectionInstance megvalósítja a Java *Callable* interfészt. Ez szolgál ősosztályként a TCP és UDP sender és receiver osztályoknak.

HttpParser a HTTP* üzenetek feldolgozását végzi. A *parseHttpMessage* metódusnak kell átadni az üzenetet és a függvény true (igaz) értékkel tér vissza, ha sikeres volt a feldolgozás. Egyébként az üzenet szintaktikai vagy szemantikai hibás, amitől nem tudja értelmezni a paraméterként kapott szöve-

get. A feldolgozott értékek Java *Properties* típusú attribútumaiban tárolódnak. A *methodProperty* tartalmazza az üzenet típusát, egy paramétert, ami a puffer méretét adja meg vagy egy azonosítót és a HTTP verziót. A *headerProperty* pedig az opcionális paramétereket tartalmazza, ezek a tesztek paramétereit írja le: a ”MODE” az adatforgalom irányát határozza meg, hogy a szerver küldi-e az adatokat (ez a letöltés, ”DL”), vagy a mobiltelefon (ez a feltöltés, ”UL”). A másik paraméter a ”CONNECTION” a kapcsolat típusát határozza meg, miszerint UDP, vagy TCP protokollt fog használni a teszt.

Logger osztály segítségével különféle üzeneteket menthetünk el egy fájlba. A programban főleg a hibaüzenetek eltárolására használom, ami a hiba keresést könnyíti meg. Az üzenet a sztenderd kimenetre is kiíratható a *addLineAndPrint* metódus segítségével. Ez utóbbit szerver oldali komponensben használható ki, mert ott a parancssorban is olvashatóak ezek az üzenetek. A log fájl nevét a konstruktorban adhatjuk meg, alapértelmezetten ”log.txt" névvel fog létrejönni a fájl a bináris fájl mellett. Az addLine metódus hívásával az átadott szöveg ki fog íródni a fájlba és az elejére egy időbélyeg kerül.

MessageI egy interfész, ami egyetlen sendReportMessage függvény fejlécét tartalmazza. Ennek segítségével tud a TCPSender és UDPSender osztály jelentéseket küldeni - a ReportSender osztályon keresztül - a teszt és a csomagok állapotáról a ReportReceiver osztály egy példányának.

ReceiverReportI a ReportReceiver osztály interfésze, melyet jelentések küldésére használ. Külön metódus tartozik a TCP és UDP teszthez kapcsolódó jelentéshez.

ReportI interfészt megvalósítja a HttpService osztály és segítségével tud a ReportReceiver osztály példánya üzenetet továbbítani, illetve küldeni a Test Activity felé a felhasználónak. A *sendMessage* metódussal lehet üzenetet küldeni.

ReportReceiver osztály példánya a mobilkészüléken fut - egy külön szálon - és fogadja a jelentéseket, riportokat, majd továbbítja a HttpService példánynak, ami továbbítja a TestActivity-nek az Application objektumon keresztül. Így láthatóvá válik és folyamatosan frissül a TestActivity üzenetlistája. A TCP- és UDPReceiver osztályoktól kapott értékek a *ReceiverReportI* interfészen keresztül a *setReceivedBytes* metódus segítségével továbbítódnak, ha a receiver (fogadó) osztálypéldányok a mobil alkalmazáson futnak -

azaz a mobilkészülék a fogadó -, így a mobil letöltési sebességét vizsgáljuk. Egyébként a szerver által fogadott csomagokról a ReportSender osztálytól kap üzenetet, amit a *receiveReport* függvényben található Scanner objektum vár és olvas be. A *parseReport* és *checkProperty* metódusok értelmezik a kapott üzenetet és küldenek jelentést a HttpService-nek a ReportI interfészen keresztül, ahogy a korábban említett *setReceivedBytes* függvény. A sebesség adatok is itt kerülnek kiszámolásra a *calcSpeed* függvényben.

ReportSender a szerver oldali programban fut az osztály egy példánya, külön szálon. A megadott porton létrehoz egy ServerSocket-et, ahova a kliens oldalon futó ReportReceiver osztálypéldánya tud csatlakozni. Ezen keresztül kap értesítést a kliens a szerver oldali csomagok érkezéséről, a teszt állapotáról. A *sendReportMessage* függvény segítségével küld HTTP üzenetet, ami tartalmazza a jelentés paramétereit.

TCPReceiver fogadja a TCP típusú teszt során a TCP csomagokat és számolja a kapott üzeneteket, azok méretét és erről jelentést küld előre megadott időintervallumonként, ezt a *setReportInterval* függvény segítségével lehet megadni. A *TimerTask* osztály segítségével lehet meghatározni, hogy milyen művelet hajtódjon végre az időzítő lejártakor. Az időzítést a Timer osztály *scheduleAtFixedRate* metódusával lehet megadni. Az osztály konstruktora képes MessageI és ReceiverReportI interfészeket fogadni attól függően, hogy szerver oldalon fut - így a ReportSenderen keresztül tud jelentést küldeni -, vagy a mobil oldalon, amikor a ReportReceiver osztály példányát használja. A *setSocket* segítségével állítható be a socket, amin keresztül érkeznek majd az adatok, üzenetek. A *readPackets* végzi az üzenetek figyelését és kiolvassa az InputStream objektum segítségével a kapott adatokat. A *totalReadedBytes* változó tartalmazza az eddig fogadott adatok mennyiségét byte-ban. A ConnectionInstance osztály leszármazottjaként tartalmaz egy stop metódust is, ami a teszt leállításakor hívódik meg és leállítja az időzítőt és a socket-et.

TCPReport a TCP teszt mérési eredményeinek eltárolására szolgál. Tartalmazza: a reporter objektum azonosítóját, a jelentés intervallumát, az átvitt adatokat és a le- vagy feltöltési sebességet. tartalmaz egy parseReport függvényt, mely egy String objektumot elemez és feltölti a kiolvasott értékekkel a riport paramétereket. A jelentés paraméterek a *toString* metódus felüldefiniálásával szöveggé konvertálhatóak.

TCPSender a ConnectionInstance osztályból származik és a konstruktorában megadott méretű puffert tölt fel véletlenszerű adatokkal. A puffer adatát

folyamatosan küldi a *setSocket* függvényen keresztül megadott socket-en keresztül egy OutputStream objektum segítségével. A küldést addig folytatja, amíg a *stop* metódusát meghívva le nem állítjuk, ami lezárja a socket objektumot.

UDPReceiver fogadja a UDP típusú teszt során a csomagokat és számolja a kapott üzenetek méretét, figyeli a csomagok sorrendjét és az elveszett csomagokat is számolja. Erről jelentést küld előre megadott idő intervallumonként, ezt a *reportInterval* paraméter lehet megadni az osztály konstruktorában. Az UDP protokollt használó tesztek során nem épül ki tartós kapcsolat a kliens és a szerver között, ezért nem kap Socket objektumot, hanem maga az UDP teszt osztály építi ki magának a kapcsolatot. Az UDP protokoll történő üzenetküldéshez a Java DatagramSocket és DatagramPacket osztályait használom. A mobilhálózatokban előforduló NAT-olás miatt a kommunikációt minden esetben a mobil oldali osztálynak kell kezdeményeznie, így nyitva egy portot, amin keresztül ezután a szerver képes üzenetet küldeni a mobilkészüléknek. A mobil oldalon való futtatáskor a *setSenderParameters* metódust használom, ami megkapja a server IP címét, a teszt által használni kívánt port számot és létrehoz egy alapértelmezett Datagramsocketet. Ezt a szál *start* metódusának meghívása elött kell meghívni. A *start* meghívása után az osztály *call* függvényébe kerül át a vezérlés, ami leellenőrzi, hogy be van-e állítva a *senderAddress* paraméter - vagyis a szerver IP címe -, ha igen, akkor meghívja a *sendAddressToSenderThroughNAT* függvényt. Ez arra szolgál, hogy küld egy DatagramPacketet a szerver címével és port számával a DatagramSoceket-en keresztül és vár egy válasz üzenetet. A riportolás a TCPReceiver-hez hasonlóan történik, annyi különbséggel, hogy itt több adatot küld át. Ezután a call metódus folyamatosan vár bufferSize méretü DatagramPacketeket. Számolja ezek sorszámát és fiegyeli az elveszett illetve rossz sorrendbe érkező csomagokat és számolja az aktuális jitter értéket a *parsePackage* függvényben. A jitter érték számítás képletét az RFC 1889, Real Time Protocol (RTP) dokumentum tartalmazza. A *lost* paraméter tartalmazza az elveszett, a *cntOutofOrder* attribútum a felcserélődött csomagok számát tárolja, a *sum* pedig az összes fogadott csomagszámát.

UDPReport a TCPReport leszármazottja, mely tartalmazza a UDP tesztek sajátos paramétereit: az elveszett csomagok számát, a felcserélődött csomagok számát, az össz csomag számot és a jittert. Felüldefiniálja az ősosztályának metódusait, a *parseReport* és a *toString* függvényeket. Először meghívódik az ősosztály függvénye, majd az UDP specifikus attribútumok

kezelése történik meg.

UDPSender az UDPReceiver-hez hasonlóan megtalálható a *setReceiverParameter* metódus, amelyből két példány is megtalálható különféle paraméterlistával. A mobil oldalon csak a port számot adjuk meg, ekkor a receiverAddress paraméter null értéket fog tartalmazni, ami azt jelenti, hogy a *getAddressThroughNAT* metódus fog meghívódni a *call* metódus futása során. A *getAddressThroughNAT* fogadja a mobilkészüléktől az üzenetet és a beérkező DatagramPacket-ből kiolvassa a feladó címét és a portot, ahova majd az üzeneteket fogja küldeni. UDP protokoll nem garantálja a stabil adatátvitelisebességet sem, így ez megadható *setRateInBitsPerSec* segítségével. Az *initDelay* segítségével kiszámolódik, hogy mennyi időt kell várni a két csomag elküldése között ahhoz, hogy egységes adatrátát tudjon produkálni a teszt. A csomagküldés után ebből az időből levonódik a csomag elkészítésének ideje (paketization time) és ennyit vár a következő csomag küldése előtt, a JavaThread osztályának *sleep* metódusának segítségével várakoztathatjuk a szál futását. Minden csomag tartalmazza a csomag azonosítóját és elküldésének idejét.

Utility osztály csak statikus függvényeket tartalmaz. Ezeknek kisegítő szerepe van. A *decodePercent* metódust a HTTP üzenet feldolgozásánál használom, a *fillStringBuffer* metódusok, pedig az átküldendő üzenetek elkészítésében játszanak szerepet. Véletlenszerű karakterekkel töltenek fel egy előre megadott méretű tömböt.

Szerver oldal osztályai

HttpServer a szerver oldali alkalmazás fő osztálya. Ez tartalmazza a *main* metódust, ami az alkalmazás belépési pontja. Az indulása után létrehoz egy *ServerSocket* objektumot és az előre definiált 4500 porton vár a kliens (mobilegység) csatlakozására. A ServerSocket accept metódusa blokkolja a futást, amíg nem kapcsolódik a kliens, amint ez megtörtént létrehoz egy új szálat (Thread), amiben a ServerThread osztály egy példánya fog futni. Jelen állapotában csak egy mobilt tud kiszolgálni az osztály, de könnyedén módosítható, hogy több eszközt is ki tudjon szolgálni.

ServerThread osztály szolgálja ki a klienstől érkező kéréseket. A HttpServertől kapott socket-en keresztül fognak érkezni a klienstől a kérések, ezek kezelésére kell egy Scanner objektum, amit a konstruktorban hozok létre a socket *getInputStream()* metódusának segítségével. A válaszok küldésére egy PrintWriter objektumot használok. A Java*Thread* (szál) osztályának

leszármazottjaként megörökli a *run* metódust, amit felüldefiniál. Az ott leírt parancsok futnak a szál objektumon meghívott *start* függvény után. Az osztály a konstruktorában hívja meg saját magára a start függvényt. A kliens és a szerver HTTP*/1.0 protokol alapján kommunikálnak egymással. A kliens kezdeményezi a kommunikációt, így a *run* metódus kezdetén megvárom a kliens üzenetét. Minden üzenetnek ”\nEND"-del kell végződnie, addig fűzi össze soronként a klienstől kapott üzeneteket. Ezután a *HttpParser* osztály *parseHttpMessage* függvényének segítségével értelmezi a program az üzenetet. Két féle üzenetet fogad a szerver: az egyik a ”GET” a másik pedig a ”STOP”, minden más esetben hibaüzenetet küld vissza a kliensnek válaszul. Az válaszüzenetet a *sendResponse* metódus segítségével küldi ki az applikáció. GET üzenet esetén létrehozunk egy ServerSocketet-et a következő szabad portszámmal - ez 5500 és 5600 között vehet fel értéket- és 5 másodpercig vár, hogy a kliens oldalon indított teszt csatlakozzon hozzá, ezután a *makeTestHandlingThread* metódus ellenőrzi a kapott paramétereket és küld választ a kliensnek, mely tartalmazza a test port számát. Hiba esetén error üzenetet küld vissza a kliensnek. A kliens csatlakozása után inicializálódik a ReportSender objektum, későbbiekben ezen keresztül tud jelentést küldeni a tesztről az applikáció. Végül a beállításoknak megfelelően létrejön és inicializálódik a megfelelő teszt osztály, amit egy *ExecutorService* objektumban tárolunk, mely segítségével visszajelzést kaphatunk az aszinkron futó szálak befejeződéséről és visszatérési értékükről. A ExecutorService *pool* metódusának hívásakor a szál futása is elkezdődik. ”STOP” típusú üzenet akkor érkezik, mikor a felhasználó leállítja a tesztfuttatást. Ilyenkor az összes futó szál *stop* metódusa meghívódik és leáll a futásuk, beleértve a *report* osztályt is.

Mobil alkalmazás osztályai

DataStorage az adatbázis kezelésére szolgáló osztály. Ezen osztály metódusai segítségével hajthatunk végre módosításkot, lekérdezéseket az adatbázisban. Az open metódus meghívásával létrejön a tényleges adatbázis, ezután lehet új rekordokat felvinni és módosítani azokat az adatbázisban. Ezt a metódust csak egyszer kell meghívni az applikáció indulásakor. A *db* változó reprezentálja az SQLite adatbázist, és az *open* metódus meghívásakor kap értéket a DbHelper *getWritableDatabase* metódusának segítségével. Az *insert* metódus segítségével tudunk egy új elemet, rekordot felvinni az adatbázisba, paraméterlistájában a rekord összes paraméterét meg kell adni. A különféle query (lekérdezés) kezdetű metódusokkal lekérdezéseket futtat-

hatunk, melyek visszatérési értéke egy String vagy DbData típusú objektumokat tartalmazó lista. A *queryAll* metódussal az adattábla összes elemét lekérdezhetjük, a *querySpecifiedTest és a querySpecifiedTestByName* metódussal egy meghatározott azonosítójú, illetve nevű elemet kérdezhetünk le. A cursorToData nevű private függvény segítségével alakítja át az osztály a lekérdezés eredményeit - amik Cursor típusú elemek - DbData típusúvá. Így egységesen és könnyen kezelhetőek lesznek a lekérdezések kimenetei.

DbHelper osztály az Android SQLiteOpenHelper osztályából származik, mely segítségével lehet létrehozni az adatbázist. A programban az *onCreate* metódus lefutásakor egy SQL művelet segítségével hozzuk létre a megfelelő attribútumokkal rendelkező adatbázistáblát. Az onUpgrade metódus segítségével lehet egy korábbi verziójú adatbázistáblát frissíteni egy újabbra, aminek például több vagy kevesebb attribútuma van. A program ilyen esetben csak törli a korábbit és létrehozza az újat. Ezek a műveletek tranzakciókban hajtódnak végre, ami biztosítja, hogy egy hiba esetén a program visszaáll a korábbi állapotra, táblára, így biztosítva az adatok sértetlenségét.

DbData struktúra tartalmazza egy adatbázis elem paramétereit. Konstruktorában ezek inicializálódnak. Az egyes paraméterek publikus hozzáférésűek, így az osztályon kívülről szabadon állíthatóak ezek értékei. A *toString* metódus szöveges kimenetet állít elő az attribútumokból. A *toDescriptionString* pedig a OSMActivityben használt mérési pontokon megjelenő szöveget generálja ki az adott parméterekből.

DriveTestApp az Android Application osztály leszármazottja és a LocationSubject, a PhoneStateSubject és a TestSubject interfészt valósítja meg. Az interfészek az Observer tervezési mintának megfelelő elemek, így az Observer (megfigyelő) objektumok beregisztrálhatják magukat és a releváns adat(ok) változásáról értesítést kapnak. Ezen osztály objektuma indul el legelőször a program indítása során. Azon paraméterek tárolódnak itt, melyeket több Activity is használ, általában az információ áramlást, cserét segítik ezek a tárolt adatok. Itt történik az adatbázis létrehozása és írása és azokat az adatokat, melyek nem a mérésből származnak (hálózat típusa, jel erőssége, stb) is itt tárolódnak el. Továbbá eltárolódik a mobilinternet és a GPS használatának be- vagy kikapcsolt állapota, ami a teszt futtatása előtt ellenőrzésre kerül. Ezen információkat a megfelelő szolgáltatások adják, a PhoneStateListenerService a mobilkészülék adatainak és a telefon állapotának változásairól értesít, míg a GPSService szolgáltatás a GPS pozíció értékét frissíti. Ezen szolgáltatásokat az Application objektum indítja el és

a hozzájuk tartozó adatokat, attribútumokat is ez kezeli. A DriveTestApp objektum értesül a változásokról a szolgáltatásoktól és értesíti a beregisztrált megfigyelő objektumokat. Itt található a prefs változó is mely egy SharedPreferences típusú változó. Ez tárolja a Beállítások menüben beállított értékeket és a prefrences.xml tartalmazza a beállítható paramétereket és a menü leírását is. Tartalmaz egy Handler típusú változót mely fogadja a HttpService-től érkező üzeneteket. Ezen keresztül továbbítódnak a jelentések a tesztek futása során és itt íródnak ki adatbázisba a jelentések a *storeUDPReportItem*, illetve a *storeTCPReportItem* metódusok segítségével. A *startHttpClientService* függvény indítja el a HttpService szolgáltatást, ami fontos szerepet játszik a tesztek futtatásában. A szolgáltatás indításakor átadja a szükséges paramétereket egy Intent objektumba ágyazva. A szolgáltatás indítása a *startService* metódus segítségével történik. A teszt futását és a szolgáltatást is a *stopHttpClientService* meghívásával lehet leállítani. Tartalmaz továbbá lekérdező metódusokat, melyek a DataStorage típusú változón kerülnek végrehajtásra.

Export Activity A program által generált teszt eredények fájlba menthetőek CSV formátumban ennek az Activity-nek a segítségével. A ”Set Test Id” gombra kattintva a *onTestClick* visszahívó metódus hívódik meg, ami megjeleníti a test azonosító választó felugróablakot. A test id-kat az adatbázisból kérdezi le a program majd egy AlarmDialog-on keresztűl megjeleníti azokat. A ”Export to CVS” feliratú gombra kattintva a *onExportClick* metódus hajtódik végre, ahol leellenőrzi a program a beállított paramétereket, ha valamelyik nincs beállítva, akkor hibát jelez AlarmDialog segítségével. Az adatok a FileHandler osztály segítségével íródnak ki a külső vagy belső tárra. Ha elérhető (írásra is) a külső tár, akkor oda fog kerülni a fájl, különben a belső tárra. Az exportálás a fájlművelet miatt aszinkron folyamat, ami az AsyncTask osztály segítségével lett megvalósítva. Ez az aszinkron osztály a ExportToCVS, ez felelős a fájl kiírásáért, ami a *doInBackground* metódusban lett megvalósítva, itt kerül meghívásra a FileHandler osztály példányának megfelelő kiíró függvénye.

FileHandler osztály tartalmazza a fájl kezelési függvényeket, amik segítik az írás és olvasási műveleteket a külső vagy belső tárokon. A paraméterek segítségével megadható a célkönyvtár és a fájl neve ami tartalmazni fogja a kiírt infomrációkat vagy ahonnan az olvasás fog történni. Az interfészen megadható a külső vagy belső tárat szeretnénk használni és a program automatikusan megvizsgálja, hogy elérhető-e a külső tár vagy sem. Utóbbi esetben a program mindenképpen a belső tárra fog írni és onnan próbál meg olvasni

is. Ezt a logikát a *writeFile* és a *readFile* metódusok tartalmazzák, maga a külső tár elérhetőségének vizsgálata az *init* függvényben történik meg.

GPSService szolgáltatás az Android Service osztályból öröklődik és megvalósítja a LocationListener interfészt. A szolgáltatás indításakor az Application objektumban beállítódik, hogy fut a GPS szolgáltatás, majd az *onStartCommand* metóduson keresztül meghívódik a *getLocation* függvény, ahol beregisztrálódik a GPSService osztály, mint LocationListener és így kap értesítést, ha változik a GPS pozíció, illetve, ha a szolgáltatás álllapotának változásáról is. Ehhez meghívom a LocationManager Android osztály *requestLocationUpdates* metódusát és utolsó paraméterként átadom a GPSService példányra mutató this pointert. A GPS koordináta változásakor a *onLocationChanged* visszahívó metódussal kap értesítést a szolgáltatás. Ekkor a kapott Location paramétert továbbítja a DriveTestApp objektum felé. Az osztály megszűnésekor leiratkozik a pozíció frissítési szolgáltatásról.

HttpService Az alkalamazás egyik kulcs osztálya, nagy szerepe van a teszt futtatás során. A *IntentService* osztályból származik, mely olyan szolgáltatás, ami csak addig fut amíg az *onHandleInstance* metódusában leírtak végrehajtódnak. Ez az osztály kezeli a szerveroldali ServerThread osztályhoz hasonlóan a teszt futtatásában aktívan részt vevő osztályok példányosítását és a szálkezelést is. A Messenger osztály segítségével tud kommunikálni a TestActivity példányával a *sendMessage* függvény segítségével tud annak üzenetet küldeni. Ezt a Messenger objektumot az *onHandleInstance* metódusban kapja meg az osztály az Intent típusú paraméteren keresztül, akárcsak a többi teszt futtatáshoz szükséges paramétert mint az adatáramlás iránya, protokoll típusa és a buffer mérete. Ebben a fügvényben csatlakozik a mobilkészülék a szerverhez az előre definiált 4500-as porton keresztül. Ez a program futása során csak egyszer történik meg. Ezután meghívódik a *makeNewThread* nevű függvény, ahol elkezdődik a kommunikáció a serverrel HTTP*/1.0 típusú üzenetek segítségével. Itt beszéli meg a készülék a szerverrel a teszt paramétereit, majd létrejönnek a teszt futtatáshoz szükséges TCP vagy UDP osztály és elindul egy külön szálon a reportReceiver is, ami a jelentéseket fogadja. A *sendMessageToServer* metódussal tud üzenetet küldeni a szerver felé, a *receiveMessageFromServer* függvényben pedig vár egy üzenetet a szervertől és értelmezi azt a *parseServerAnswer* metódus segítségével. Az onDestroy eseményt a szolgáltatás leállítása váltja ki, ekkor a *stop* metódusának segítségével leállítódnak a teszt szálak is.

MainActivity Az alkalmazás fő (main) activity objektuma, vagyis ez az objek-

tum töltődik be az alkalmazás indulásakor és az ehhez tartozó felület fog megjelenni a képernyőn. A képernyőn megjelenő paraméter lista két fő részből tevődik össze. A telefon adataiból - ezek nagyrészt statikus adatok -, melyeket egyszer kérdez le az osztály a TelephonyManager segítségével, ezek a phoneDataList változóban tárolódnak. A másik rész pedig a mobil hálózatra vonatkozó információk, melyek dinamikusan változhatnak, ezen információkat a networkDataList változó tartalmazza. Ezek értékeit a PhonestateListenerService-től kapja az osztály. Ehhez megvalósítja a PhoneStateObserver interfészt és beregisztrálja magát az application objektumba, mint observer (megfigyelő) objektum: "((DriveTestApp)getApplication()).registerPhoneStateObserver(this);" A SimpleAdapter segítségével lehet táblázatos formába rendezni az adatokat, ehhez szükséges egy layout xml fájl, ami leírja az elrendezést. A két listához tartozó adapter neve phoneDataAdapter és networkDataAdapter. A *setHashMapElement* metódus használatával adhatunk hozzá elemet a listákhoz, illetve módosíthatjuk egy bejegyzés értékét. A módosítást követően az adapter objektum *notifyDataSetChanged* metódusával frissíthetjük a kijelzőn megjelenő listát. A SeparatedListAdapter segédosztály segítségével jeleníthetjük meg ezeket a listákat fejléccel, összefűzve. Ennek segítségével rendelhetünk különböző színt a lista soraihoz. Az osztály *OnCreate* metódusában inicializálódnak a lista attribútumok az *initPhoneData*, illetve az *initNetworkData* függvények segítségével. A TelephonyManager segítségével kiolvassuk a megfelelő értéket az állandó paraméterekhez, a változó értékekhez "-" jel íródik be, mely rögtön megváltozik, ha az adott paramétert érintő változás következik be. Az osztály OnPause esemény bekövetkeztekor a removePhoneStateObserver metódus segítségével kitörölteti magát a PhoneStateObeserverek listájáról. Ez az esemény akkor következik, be mikor a felhasználó másik activity-re vált át vagy bezárja az applikációt. Az OnResume esemény következik be, amint újra ez az activity kerül előtérbe. Iyenkor beregisztrálódik újra a PhoneStateObeserverek listájára és értesítést kap az értékek változásokról.

OSMActivity főfeladata a térkép és azon a kiválasztott mérési pontok megjelenítése. A TestObserver interfészt megvalósítja, mivel így kap értesítést az újabb teszt bejegyzések érkezéséről, amit lekér az adatbázisból a *queryLastInsertedRow* függvény segítségével és megjelenít a térképen. Ehhez az *update* metódust kellett felüldefiniálni. A térkép megjelenítéséhez és kezeléséhez egy külső könyvtárat használok osmdroid-ot. Ez egy szabadon felhasnzálható könyvtár, mely a Google Maps-hoz hasonló szolgáltatásokat nyújt ingyenesen. Ennek segítségével tudom megjeleníteni a térképet

és rajta az útvonalat és a jelentéseket reprezentáló elemeket. Az onCreate eseménykor inicializálódnak a paraméterek, mint például a mapView, ami a térképnézetet reprezentálja és a controller változó, ami IMapController típusú és többek között ezen keresztül lehet a nagyitást (zoom) állítani. Az onPause eseménykor elmentésre kerül az aktuális pozíció, a nagyítás értéke és a beállított teszt azonosító, illetve teszt név is. Ezek az információk az onResume eseménykor, vagyis mikor újra aktív lesz ez az ablak, visszatöltődnek és beállítódnak. A teszt azonosítót a ”Set Test Id” feliratú gombra kattintva állíthatja be a felhasználó, ennek megnyomásakor a *onTestClick* függvény fog meghívódni. Ha a teszt nevét választja ki a felhasználó, akkor a ”Set Test Name” feliratú gombra kattintva a *onTestNameClick* metódus fog végrehajtódni. A szövges mezőben megjelenik a kiválasztott érték, a megadott teszt azonosító vagy a teszt neve. A teszt név több teszt futást is tartalmazhat, különféle teszt azonosítójú teszteket foghat össze. Ilyenkor a térképen a különbözö tesztek végpontjai is össze lesznek kötve egymással. A *loadRoad* metódus hívódik meg egy teszt betöltésekor és egy új elem hozzáadásakor is. Az itt szereplő nodeA és nodeB változók két egymást követő mérési pontot jelölnek a térképen. Ezek RoadNode típusú objektumok, ami tartalmazza a pont helyét a térképen (szélességi és hosszúsági fokot) és szövegesen a mérési adatokat. A *addRoadNode* metódus segítségével készít a program RoadNode típusú objektumokat az egyes adatbázis rekordokból. A noOfPoints változó tartalmazza, hogy eddig összesen hány darab pont van betöltve a térképen, illetve a nodeCount változó meghatározza hogy még hány pontot kell betölteni. Maga az út rajzolás és számolás kötelezően AsyncTask típusú objektumban kell hogy történjen, hogy ne akadályozza a fő szál futását, így ezt az UpdateRoadTask belső osztály végzi. A roadManager *getRoad* metódusa kiszámítja az utat a két megadott végpont között és visszatér egy Road típusú elemmel. Az *drawRoadOnMap* metódus rajzolja a térképre a kiszámolt útszakaszt a PathOverlay típus segítségével, ami a RoadManager *buildRoadOverlay* függvényének eredménye. Itt korrigálódik a mérési pont helye, hogy illeszkedjen az útra és kirajzolódik a mérési pontot reprezentáló jel is. A jel színe a jel erősségétől függően 5 féle lehet. Ha van még kirajzolandó pont, akkor újra meghívódik a loadRoad függvény, egyéb esetben pedig a setOverlays hívódik meg, ami az utolsó betöltött pontot helyezi a középpontba és garantálja hogy az egyes rétegek megfelelő sorrendben kerüljenek kirajzolásra, végül újra rajzoltatja a térképet az *invalidate* metódus hívással.

PhoneStateListenerService Az Android API által nyújtott dinamikus, időben vál-

tozó telefon és hálózati információk lekérdezésére vonatkozó szolgáltatásokat valósítom meg itt. A *connectivityReceiver* egy BroadcastReceiver, minek segítségével a ConnectivityManager üzeneteit megkapja az alkalmazás és így figyelhető a mobil internet szolgáltatás elérhetősége. A változásról értesül az application objektum is. A készülék és a hálózat adatairól a TelephonyManager segítségével kaphatunk információkat. PhoneStateListenerImpl segédosztály megvalósítja a PhoneStateListener interfészt, így kap értesítést az applikáció az adatkapcsolat (data connection) állapotáról és a pozícióhoz kapcsoló MCC, MNC, LAC és CID paraméterekről is. Ezek az értékek az application objektumban lesznek letárolva. A PhoneStateListener a ConnectivityReceiverrel egyetemben a service konstruktorába kerül beregisztrálásra. A jelerősség értéket osztályozom a GSM sztenderdnek megfelelően (TS 27.007, 8.5 fejezet) ASU (Arbitrary Strength Unit) értékek 0-tól 3-ig terjednek, ahol a 31 a kiváló minőséget jelöli, ezen értékek átszámíthatóak dBm-be és a jel erősség paraméter ezt az átszámolt értéket fogja tartalmazni.

PhoneStateObserver interfész tartalmazza a telefon adatainak változásához tartozó értesítési, frissítési metódusok fejléceit.

PhoneStateSubject interfész tartlamazza a PhoneStateObserver-hez kapcsolódó értesítési és regisztrálási függvények fejléceit.

PrefsActivity egy egyszerű Activity, ami a beállítások megjelenítéséért felelős. Tartalmazza a menüket a többi Activity-hez hasonlóan, melyek segítségével elérhető a többi Activity is. A *onSharedPreferenceChanged* metódus segítségével értesül az osztály a beállítások változásáról, így ellenőrizhető a megadott adatok helyessége. Az IP cím megadása esetén használom ezt a metódust, ha nem illeszekedik a megadott mintára a bevitt cím, akkor egy előre megadott, alapértelmezett IP címet álítok be, ami jól formázott és egy Toast üzenet segítségével értesítem a felhasználót a hibáról.

TestActivity a felhasználó itt állíthatja be a tesztek paramétereit és elindíthatja vagy leállíthatja teszt futtatását.

TestObserver interfészen keresztül egy osztály a tesztek során érkező jelentésekről értesülhet szöveges formában.

TestSubject interfész a TestObserver-ek ki- és beregisztálásához szükséges metódusok fejlécét tartalmazza.

4.2.3. Osztályok egymás közti kommunikációja teszt futtatás során

Az alábbi szekvencia diagrammon egy TCP teszt futása során használt üzenet és jelentés küldések láthatóak. Ez a teszt letöltési (Download) üzemmódban fut vagyis a szerver küldi az adatokat a TCPSender osztályon keresztül és a mobileszköz fogadja azokat a TCPReceiver segítségével.

1. Felhasnzáló megnyomja a ”Start Test” gombot a TestActivity-ben és elindul a GSP helyszolgáltatás és a HttpService osztály példányosul és egy külön szálon, a háttérben futni kezd.

2. Előre megadott porton keresztül és a beállításoknál meghatározott IP címen elérhető HttpServer objektumhoz kapcsolódik egy Socket-en keresztül.

3. A kapcsolat létrejötte után a HttpServer példány létrehoz egy ServerThread példányt egy új szálon. Ez szolgálja majd ki a telefontól érkező teszt kéréseket.

4. A HttpService példány elküldi a ServerThread-nek a teszt beállításait és a receiveMessageFromServer metódusban aktívan vár a szerver válaszára. A sendResponse-ban küldött válaszban megkapja a szerver által használt teszt portját, ahova majd csatlakozni tud.

5. A ServerThread létrehozza a ReportSender osztályt, ami szükség esetén továbbítja a jelentéseket a mobilkészülék felé.

6. HttpServer létrehozza a ReportReceiver egy példányát és csatlakozik a ReportSenderhez egy előre definiált porton.

7. A tesztnek megfelelő osztály létrejön a szerver oldalon, jelen esetben a TCPSender egy példánya, ami aszinkron módon egy párhuzamos szálon fog futni.

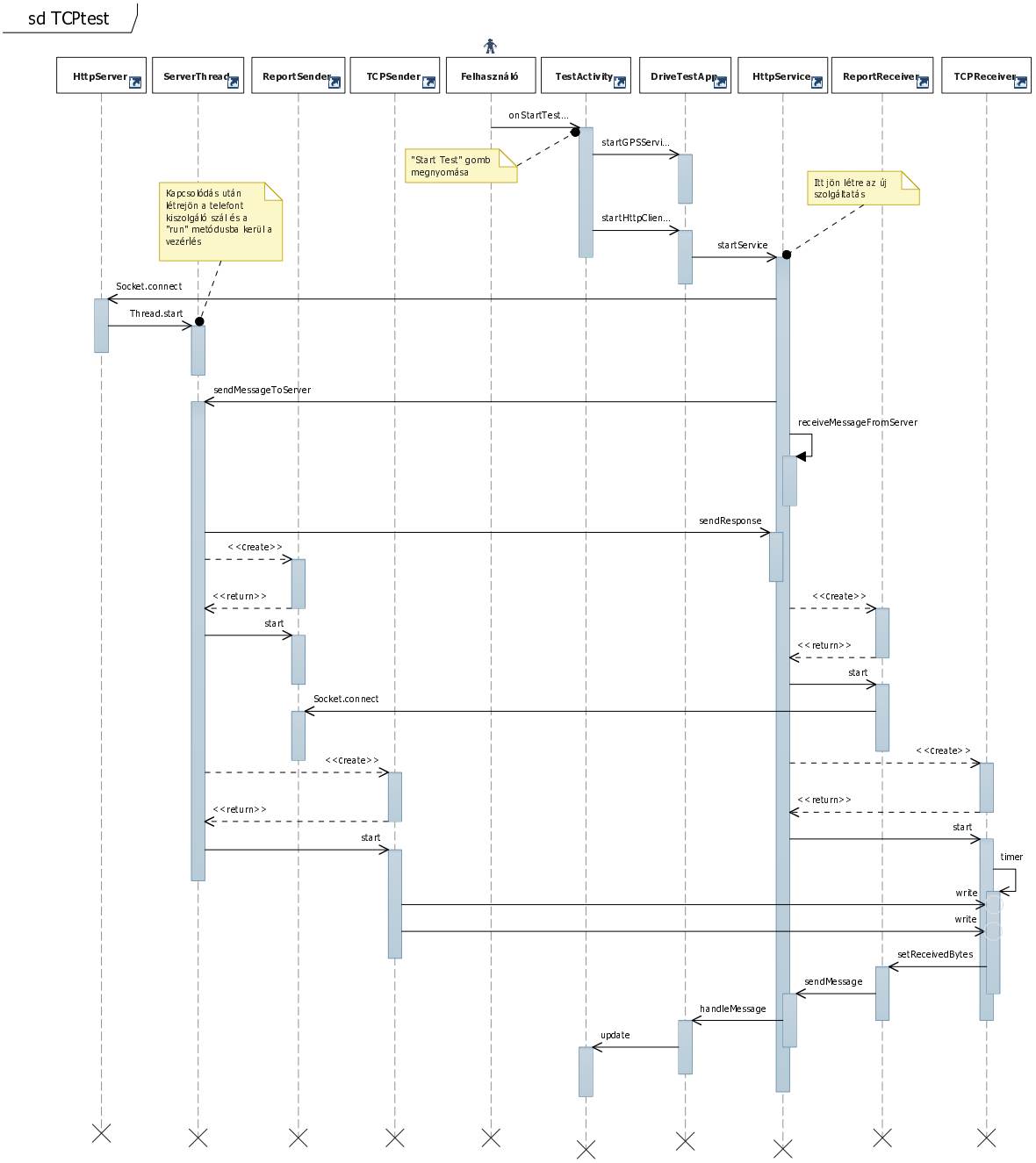
8. A TCPSender folyamatosan küldi a csomagokat a TCPReceiver felé, ami előre beállított intervallumonként a ReportReceiver-en keresztül jelentést küld egy Timer segítségével. Ez a jelentés az adatforgalomról tartalmaz adatokat, például az átviteli sebességet és az átvitt adat mennyiségét tartalmazza. A felhasználó a TestActivity Messages ablakában láthatja ezeket a jelentéseket, mindig a legfrissebb kerül a lista tetejére.

9. A ReportReceiver továbbítja a DriveTestApp application objektumnak az üzenetet egy Handler objektum segítségével. Itt a jelentések elmentődnek

az adatbázisban és értesítésre kerülnek a TesztObserver interfészt megvalósítható osztályok példányainak az interfész update metódusával. A jelentések listája az application objektumban tárolódik.

10. A TestActivity Message ablakában megjelenik a frissített üzenetek listája, amit láthat a felhasználó.

4.2.1. ábra. TCP teszt futása



4.3. Tesztelés

4.3.1. Statikus kódelemzés

A "Lint" program a Bell Laboratories terméke, az AndroidLint, része az Android SDK-nak. Ez az eszköz megvizsgálja a kódot, és javításokat ajánl. Lehetnek esetek amikor a program téved és az adott körülmények között az adott kód helyes, amire a Lint azt javasolja, hogy javítsd ki, így természetesen nem helyettesíti a hozzáértő fejlesztőt.

Egyik nagyon gyakori hiba, hogy egy Toast létrehozásakor a *makeText* metódust használják, és elfelejtik meghívni a *show* metódust; a toast létrejön, de soha nem fog megjelenni! A hagyományos fordító nem tudja elkapni ezt a hibát, de az Android Lint képes erre, és ez csak egy a sok közül. Eclipse alól nagyon egyszerű és kényelmes a használata. Sok lehetséges hibaforrást megtalált és a fejlesztés során törekedtem rá hogy alacsonyan tartsam a Lint hibajelzések számát.

4.3.2. Unit teszt

Az Android tesztek JUnit-on alapszanak. Hagyományos JUnit teszteket is használtam a HttpServer és HttpTestHandler projektek unit tesztjeinek írásakor, ezek nem használják az Android SDK-t. Az Android tesztesetekkel az Android komponensek megfelelő viselkedését tesztelem ezek a teszt osztályok az AndroidTestCase-ből öröklődnek, melyek a JUnit keretrendszeren túl Android specifikus setup, teardown, és egyéb segéd metódusokat tartalmaznak. A JUnit-beli Assert osztályt használva jeleníthetjük meg a teszt eredményeket és jelezhetjük a hibákat. Ezek a metódusok összehasonlítják kapott eredményeket a tesztben várt eredményekkel, ha nem egyeznek meg akkor kivételt generálnak, mely esetben a teszt hibát fog jelezni.

DriveTesting applikáció

Külön tesztprojektet készítettem a DriveTesting applikáció teszteléséhez *DriveTestingTest* néven.

DataStorageTest az adatbázis tábla létrehozását és az adatok beszúrását teszteli. Az insert (beillesztés) nem csak az adatbázis műveletet teszteli, hanem az adatbázis tábla attribútumainak helyes sorrendjét és típusát is. Ezenkívül a lekérdezéseket is teszteli, amik fontos szerepet játszanak a tesztelés folyamán, főként az exportálás és a térképes megjelenítés folyamataiban.

MainActivityUnitTest a két HashMap lista, vagyis a phoneDataList és a networkDataList alapértelmezett beállítását ellenőrzi. Továbba a PhoneStateObserver interfész által biztosított *update* függvények implementációjának helyességét is teszteli.

SeparatedListTest a SeparatedListAdapter osztály metódusait teszteli. Az Android BaseAdapter osztály metódusait felüldefiniálja és lehetőséget biztosít egy fejléces, kétszínű lista létrehozására. A teszt ezeket a funkciókat teszteli.

HttpTestHandler

HttpParserTest a HTTP*/1.0 típusú üzenetek feldolgozását végző osztályt teszteli, amit mind a szerver, mind a kliens oldal használ. A teszt során különböző bementi paraméterek, üzenetek esetén vizsgálom a feldolgozó (parser) osztály belső állapotát.

LoggerTest a log fájl készítést és kezelést teszteli. A log fájl tartalmazza a program futása során bekövetkező, annak működésére vonatkozó üzeneteket. Ezek megkönnyíthetik az esetleges hibák felderítését és kijavítását is.

ReportReceiverTest a HTTP*/1.0 üzenetek fogadását végző osztályt ellenőrzi. A küldött üzenetet feldolgozza és belső változóiban tárolja az utolsó üzenet értékeit, ezeket lekérdezem a teszt során és ellenőrzöm, hogy a megfelelő értékek szerepelnek-e az egyes változókban.

ReportSenderTest a HTTP*/1.0 üzenetek küldésével foglalkozó osztály tesztelését végzi. A sendReportMessage működését vizsgálom, hogy a megadott üzenetet milyen formában küldi el.

TCPReport a TCP jelentések feldolgozását végző osztály metódusainak tesztelésért felelős. Annak *parseReport* metódusán keresztüli adatbevitel a szöveggé alakító *toString* metódus segítségével ellenőrizhető.

UDPReport a TCP verzióhoz hasonlóan hajtódik végre a tesztelés. A parseReportMetódusa van felüldefiniálva, hogy az UDP jelentés specifikus tagjait is feldolgozza, ez kerül tesztelésre.

HttpServer

HttpResponseTest a HttpResponse osztályt teszteli, ami egy HTTP*/1.0 típusú válaszüzenetet generál a szerver oldalon. Különféle bemeneti paraméterek

esetén megvizsgálja a kimeneti szöveget és ezzel az elküldendő üzenetet hitelesíti.

4.3.3. Komponens teszt

Android ActivityInstrumentationTestCase2 osztály segítségével létrehozhatóak olyan tesztek, melyek az egyes Android komponensek életciklusát befolyásolják. Kiválthatóak az egyes rendszerszintű események, így tesztelhetőek a komponensek egymásra hatása is. Például egy Activity objektum életciklusa kezdetekor meghívódik az objektum *onCreate* metódusa, amit az *onResume* követ. Egy másik applikáció elindulása, vagy egy másik Activity-re váltás hatására, az *onPause* metódus hívódik meg. Az Android keretrendszer nem biztosít lehetőséget ezen metódusok, események közvetlen meghívására, az Instrumentation-ok használatával ez mégis lehetővé válik. A tesztek futásuk során az Android emulátort használják.

ExportActivityTest az ExportActivity osztályt teszteli. A menük közti navigálást és az Activity-k közti váltást és a felületen elhelyezett elemek működését teszteli.

MainActivityInstrumentTest a MainActivityt teszteli és annak fontos adatainak alapértelmezett beállításait. Megvizsgálja a phoneDataList és networkDataList lista elemek értéket, ellenőrzi a menük közti navigálást és az Activity-k közti váltást és a felületen elhelyezett elemek működését teszteli.

TestActivityTest a TestActivity osztályt tezteli az grafikus interfész helyes beállítását és megvizsgálja az alapértelmezett értékeket. A menük közti navigálást és az Activity-k közti váltást és a felületen elhelyezett elemek működését teszteli.

4.3.4. Rendszer teszt

Teszt során a szerver és kliens programok együttes tesztelését végeztem Android emulátorral és mobilkészülékkel egyaránt - készülék típusa SonyEricsson Arc S, az Android verzió 4.0.4. Szerverként minden esetben az asztali számítógépem szolgált, mely segítségével tudtam adatforgalmat generálni a mobilkészülék és a számítógép között. A számítógépem egy router-en keresztül kapcsolódik az Internethez, melyen be kellett állítani a port továbbítást, hogy tudjon kommunikálni a mobilkészülékkel. A tesztek során számos hibát felderítettem a

programban, melyeket alacsonyabb szintű tesztekkel nem volt lehetséges megtalálni, mert ezek csak a két komponens (szerver és kliens) együttes jelenléte és kommunikációja esetén lehetett felderíteni. Ilyen probléma volt a mobilkészülék NAT-olt hálózatban történő elérése. Biztonsági szempontok miatt nem lehet közvetlenül elérni a készülékeket publikus IP címen keresztül, így minden esetben a készüléknek kell kezdeményezni a kommunikációt, mely folyamat során nyit egy portot, amin keresztül a külső program tud válaszolni a mobilon futó programnak. Ez főleg UDP protokoll esetén volt probléma, ahol nincs állandó kapcsolat kiépítve, csak egy-egy üzenet elküldésének idejére. A tesztelések során valós környezteben is ki lett próbálva az applikáció, miközben aktívan kommunikált a szerverrel. A mérések grafikus megejelenítésekor a GPS jel pontatlansága gondot okozhat. Ennek minősége telefonkészüléktől és a terület műhold lefedettségétől is függ. Vannak területek, ahol centiméterre pontos értéket tud adni, máshol több méteres eltérés is jelentkezhet a valós pozícióhoz képest. Ezért használtam a térképes megejelenítésnél az OSMDroid útvonal generálási metódusát, ami útvonalra illeszti a GPS koordinátákat, így kis eltérés mellet is jó eredményeket tud mutatni a program. A tesztek során mind a kétféle protokollt és teszt módot kiprobáltam kombinálva egymással. Közben a mért adatokat, azok közül is a fel- és letöltési sebességet a Speedtest nevü programmal ellenőriztem, mely megméri, hogy az adott helyen milyen a sávszélességgel rendelkezik a hálózat. Az emulátoron különböző Android verziók melett eltérő felbontású, képarányú készülékkel is teszteltem a programot.

4.3.5. Tesztelés során talált hibák

A rendszer tesztelés során kiderült, hogy a HttpService osztályban hibásan lett implementálva a szerverhez kapcsolódás, így minden teszt futtatáskor újra próbált csatlakozni a szerverhez. Ez azt eredményezte, hogy mindig az első teszt portot (5500) akarta használni, ami a többszöri kapcsolódást megakadályozta, mert a port még foglalt volt. Ebben az esetben a megoldás az volt, hogy a szerverhez csatlakozott Socket objektumot eltárolja a program és csak akkor hoz létre újat ha ez a Socket objektum még nem létezik, így a szerverrel való kapcsolat megmarad és új teszt indítása esetén a tesztportok egyesével léptetődnek.

A program tervezése során elkövetett hiba volt, hogy a TCP és UDP segédolsztályok maguk kezelték a Socket objektumokat és a csatlakozást. Ebben az esetben a tesztelés során kiderült, hogy nem lehet tudni, a kapcsolati hibák okát és a program egy hibás futás végén is úgy tért vissza mintha egy sikeres futásnak lett volna vége. Ebben az esetben a megoldás az volt, hogy a socket objektumokat egy szintel feljebb a HttpServer vagy a HttpService osztályok végzik és a létrejött

Socket objektumokat a TCP és UDP osztályoknak továbbítják, és így a hibákról is értesül a felhasználó.

UDP protokollt használó tesztek esetén a hálózati kommunikáció során nem használhatóak a Socket objektumok, a kapcsolat nem építhető ki a szerver oldal felől. A mobil telefon IP címe nem publikus, általában NAT-olva van az operátorok hálózata, egyrészt biztonsági szempontok miatt, másrészt a kis címtartomány miatt. Ez azt eredményezte, hogy minden esetben a mobil készüléknek kell kezdeményeznie a kommunikációt, ezzel lehetővé téve hogy a szerver üzenetetket tudjon küldeni a készülék felé. A tesztek során világossá vált, hogy a TCP esethez hasonló megoldást kell találnom, így keletkeztek a *sendAddressToSenderThroughNAT* és *getAddressThroughNAT* metódusok, amik kiküszöblik a NAT által keletkezett problémát.

4.4. Továbbfejlesztési lehetőségek

A szerver jelenleg csak egy mobilkészüléket képes egyidőben kiszolgálni, ugyanis a ServerThread osztály mindig az előre megadott ServerPorton próbál Socket-et nyitni. Ha már foglalt a port akkor a következő Socket kreálása hibát fog jelezni és a kapcsolat nem tud kiépülni a második készülék és a szerver között.

A teszt jelenleg nem képes hanghívásokat kezelni, sem indítani, sem fogadni. Ezek használatához kernel szinten kéne módosításokat végezni, hogy a szükséges információk eljuthassanak az applikációhoz.

Továbbá a jelenlegi megoldással egyszerre csak egy teszt futtatható, megoldható lenne kis módosítással több teszt párhuzamos futtatása, de ha túl sok fut párhuzamosan, akkor befolyásolhatják egymás mért értékeit és így a teszt eredményét is.

5. fejezet

Összefoglalás

A szakdolgozatom elkészítésével sikerült betekintést nyerni az Android platformra való fejelsztés lépéseibe, valamint a mobilhálózatok bonyolúlt és szerteágazó világába. Első lépésben az Android platform került ismertetésre, mely a bemutatott alkalmazás implementálása mellett számos új lehetőségeket rejt. Mára már kiforrot mind a platform, mind a fejlesztői környezet, melyel nagyon könnyen fejleszthetünk alkalmazásokat. Bár sok lehetőséget nyújt a különféle applikációk fejlesztésében, a dolgozatom elkészítése során nem találtam módot arra, hogy a hivatalos Android SDK segítségével a hanghívásokat is tesztelhessem. Kutatómunkám során arra a következtetésre jutottam, hogy a szükséges információk alacsony szinten kerülnek letárolásra, illetve kezelésre, melyhez nem elég az Android SDK megváltoztatása sem.

Továbbá ismertettem a mobilhálózatokat és ezek generációit. Ezen leírásokból is kitűnik az egyre nagyobb mértékű és egyre inkább nagyobb hangsúlyt kapó mobiladatforgalom világában az enyém és az ehhez hasonló dignosztikai programoknak nagy szerepe van és lesz is. Olyan alkalmazást valósítottam meg, mellyel a telefonhálózatok operátorai költséghatékonyan megkönyíthetik a munkájukat. Természetesen még nagyon sok elemmel ki lehetne egészíteni a programot, ami még hatékonyabbá és többoldaluvá tehetné a programot, például a program tovább bővithető az Android nyújtotta új szolgáltatásokkal, illetve alacsony szintű fejlesztésekkel a telefonkészülékkel olyan hanghívások is tesztelhetőek, amit a dolgozatom nem támogat, így még gazdagabbá, hatékonyabbá lehet tenni tesztelést.

6. fejezet

Köszönetnyilvánítás

Szeretnék mindenkinek köszönetet modnani, aki közvetlen vagy közvetett módon segítette a dolgozat elkészülését.

Köszönettel tartozom témavezetőmnek Sike Sándornak, hogy lehetőséget biztosított munkám sikeres elvégzéséhez és dolgozatom megírásához. Köszönöm a segítőkész támogatását és dolgzatom alapos áttekintését.

Köszönöm Dósa István kollégámnak és Simonyi Tibor külső konzulensemnek a hasznos tanácsokat és támogatásukat a dolgozatom elkészítéséhez.

Végül szertném megköszönni családom türelmét és segíségét.

7. fejezet

Irodalomjegyzék

- Android, http://developer.android.com (2014.021.06)
- [ant] ant letöltése, http://ant.apache.org/manual/install.html (2014.021.06)
- [JDK] JDK letöltése, http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html (2014.021.06)
- Második generációs mobilhálózatok, http://en.wikipedia.org/wiki/2G (2014.021.06)
- Harmadik generációs mobilhálózatok, http://en.wikipedia.org/wiki/3G (2014.021.06)
- Negyedik generációs mobilhálózatok, http://en.wikipedia.org/wiki/4G (2014.021.06)
- HTTP 1.0 leírása, http://www.jmarshall.com/easy/http/ (2014.021.06)
- HTTP 1.0 szabvány, http://www.rfc-editor.org/rfc/rfc1945.txt (2014.021.06)
- Mobil jelerősségének számítása, http://en.wikipedia.org/wiki/Mobile_phone_signal (2014.021.06)
- [Wikipedia] mobil operációsrendszerek, http://en.wikipedia.org/wiki/Mobile_operating_sys (2014.021.06)

A. Függelék

Használt eszközök a fejlesztés során

- **Android SDK** 15. verzióját használom, a program ennél magasabb verziószámú Andorodi készülékeken is fut, de alacsonyabb verziójuak már nem támogatottak. Az SDK tartalmazza az Android specifikus könyvtárakat, amik használatával Android eszközön futtatható appplikációk készíthetőek.

- **Eclipse** (Indigo) egy IDE, integrált fejelsztő környezet. Többnyire Java-ban íródott. Applikációk fejlesztésére használják, a plugin-ek használatával és fejlesztésével több féle programozási nyelven történő fejlesztést is támogathat. Ezen eszköz támogatja az Android platformra történő fejlesztést is, így ezt hasnzáltam a dolgozatom elkészítéséhez. Könnyen integrálható bele egy plug-in, aminek segítségével futtathatunk Android emulátrot, ami hibakeresés során nagyon hasznos eszköz.

- **Git** egy nyílt forráskódú, elosztott verziókezelő szoftver, vagy másképpen egy szoftver forráskód kezelő rendszer, amely a sebességre helyezi a hangsúlyt. A Git-et eredetileg Linus Torvalds fejlesztette ki a Linux kernel fejlesztéséhez. Minden Git munkamásolat egy teljes értékű repository teljes verziótörténettel és teljes revíziókövetési lehetőséggel, amely nem függ a hálózat elérésétől vagy központi szervertől.

- **GitHub** (www.github.com) egy Git-es fejlesztéseket támogató weboldal, mely ingyenes tárterületet biztosít a felhasználóknak. A dolgozatom készítése során itt is tároltam a adatokat és a forráskodókat is.

- **LyX** egy nyílt forráskódú LaTeX dokumentum szerkesztő. Leegyszerűsíti és felgyorsítja a dokumentációt és sok közös LaTeX elem használatát, ugyanak-

kor engedi, hogy a felhasnzálók TEX kódot is használhassanak, ahol szükséges. LYX weboldala: http://www.lyx.org.

- **Visual Studio 2010 Ultimate** egy professzionális fejlesztő eszköz, mely nagyon sok programozási nyelven történő fejelsztést támogat. Ezen kívül széleskörűen támogatja a fejlesztés többi fázisát, aspektusát is így különféle UML diagrammok is készíthetőek a segítségével. ennek köszönhetően ezzel készítettem el a dolgozatban szereplő osztály és szekvencia diagrammokat.