Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Távoli eljáráshívás android platformon

Bozóki Szilárd
BSc mérnök informatikus hallgató

Kari TDK konferencia, 2011.

Konzulens: Dr. Goldschmidt Balázs, Irányítástechnika és Informatika Tanszék

Tartalomjegyzék

Ábrajegyzék

1 Bevező

1.1 A technológia szülte új világ

A technológia nagymértékben átszabta a világunkat az által, hogy a számítógépeket olyan mértékben sikerült lekicsinyíteni, hogy egy telefonban is elférjenek. Ma már a legtöbb ember telefonja képes integrált multimédiás szolgáltatásokat nyújtani.

1.1.1 Rövid történet

A személyi számítógépekhez hasonló karriert futottak be az okos telefonok. Az okos telefonok a 2000-es évek elején jelentek meg. A Palm 2001-ben dobta piacra ki a Kyocera6035 eszközt, ami az első olyan telefon volt, ami a telefonáláson túl extra funkciókat biztosított, akár egy PDA. A RIM 2002-ben adta ki az első BlackBerry készüléket, ami Az Egyesült Államokban jelentős sikereket ért el. A készülékek azonban csak egy szűk réteg, főleg az üzletemberek számára voltak elérhetők a magas árfekvésük miatt, illetve az átlagos felhasználónak nem nyújtottak megfelelő szolgáltatásokat. Az évek során a készülékek egyre több funkciót láttak el és egyre olcsóbbá váltak. Több nagyobb cég saját megoldással állt elő, mint például a Nokia, Microsoft, Ericsson, HP, Samsung, Palm. Nagyobb áttörést az Apple ért el 2007-ben az iPhone-nal. A készülék elsőként a világon olyan érintőképernyővel rendelkezett, ami egyszerre több ujjat is képes követni. A készülék multimédiás alkalmazásai nagy népszerűséget vívtak ki maguknak. 2008-ban a készülék árát 200$ alá vitték, így egyre nagyobb tömeg számára vált elérhetővé. Ugyanebben az évben az Apple megalapította az App Store virtuális alkalmazás áruházat, ami egy nagyon nyereséges üzletté nőtte ki magát. Ezen keresztül a fejlesztők kisebb nagyobb alkalmazásai kényelmesen eljutnak a felhasználókhoz, akik a legkülönbözőbb dolgokat igénylik és veszik meg.

A technológia odáig fejlődött, hogy hétköznapi felhasználásra, mint például e-mailek olvasása, filmek, képek, elektronikus könyvek olvasására bőven elég számítási kapacitás fér egy mobil eszközbe. Az egyetlen korlátot a kijelző mérete jelenti. Erre a kérdésre először megint az Apple lépett és 2010. Január 27.-én bemutatta az iPadet, ami egy nagyobb kijelzővel ellátott készülék, mely lefedi a hétköznapi igények széles spektrumát.

1.1.2 Statisztikák

Ahhoz, hogy tiszta képet kapjunk az okostelefonok jelentőségéről, nézzük meg a következő statisztikákat.

Forgalmazó	2010, 4. negyedév szállított mennyiség	2010, 4. negyedév piaci részesedés	2009, 4. negyedév szállított mennyiség	2009, 4. negyedév piaci részesedés	éves növekedés
HP	17,955	19.5%	18,115	20.2%	-0.9%
Dell	11,140	12.1%	10,686	11.9%	4.2%
Acer Group	9,775	10.6%	11,505	12.8%	-15.0%
Lenovo	9,551	10.4%	7,888	8.8%	21.1%
Toshiba	5,347	5.8%	4,768	5.3%	12.1%
többi vállalat	38,308	41.6%	36,687	40.9%	4.4%
Összesen	92,075	100.0%	89,649	100.0%	2.7%

1. ábra Az 5 legnagyobb PC forgalmazó eladási statisztikái

Az 1. ábra az 5 legnagyobb PC gyártó 2009-es és 2010-es utolsó negyedévéről kiadott eladási statisztikákat tartalmazza.[1]

Forgalmazó	2010, 4. negyedév szállított mennyiség	2010, 4. negyedév piaci részesedés	2009, 4. negyedév szállított mennyiség	2009, 4. negyedév piaci részesedés	éves növekedés
Nokia	28,3	28.0%	20,8	38.6%	36.1%
Apple	16,2	16.1%	8,7	16.1%	86.2%
Research In Motion	14,6	14.5%	10,7	19.9%	36.4%
Samsung	9,7	9.6%	1,8	3.3%	438.9%
HTC	8,6	8.5%	2,4	4.5%	258.3%
többi vállalat	23,5	23.3%	9,5	17.6%	147.4%
Összesen	**100,9**	100.0%	**53,9**	100.0%	**87.2%**

2. ábra Az 5 legnagyobb okostelefon forgalmazó eladási statisztikái

Az 2. ábra az 5 legnagyobb okostelefon forgalmazó 2009-es és 2010-es utolsó negyedévéről kiadott eladási statisztikákat tartalmazza. [2]

Látható, hogy 2009-ben még PC-ből adtak el többet, de 2010-re az okos telefonok átvették a vezetést. Ráadásul, míg a pc-piac 2,7 %-ot nőtt, addig az okostelefonpiac 87,2%-ot.

1.2 Az új szereplő: Android

Az Android egy ingyenes operációs rendszer, ami mobil eszközökön fut. Az Open Handset Alliance fejleszti, ami több nagyobb informatikai vállalatot tömörít és a Google vezeti.

1.2.1 Open Handset Alliance

Az Open Handset Alliance-t a Google hozta létre 2007-ben. Célja a nyílt szabványok fejlesztése mobil eszközökre.

1.2.1.1 Open Handset Alliance Tagok

A tagok különböző iparágak nagyobb cégei. A teljesség igénye nélkül áll itt egy felsorolás a nagyobb tagokról.

Vezető: Google

Készülékgyártók: Huawei, Samsung, HTC, Asus, Motorola, Toshiba, Sony Ericsson, Ericsson, LG, Acer, Garmin

Félvezetőgyártók: Intel, nVidia, Marvell, SiRF, ARM, Atheros, Broadcom

Operátorok: T-Mobile, Vodafone

Jelenleg 70-nél is több tagja van, de a számuk gyakran változik. Látható, hogy a szövetség, több iparág elég komoly támogatásával bír, ez a sikerének egy kulcsa.

1.2.1.2 A tagok motivációi

A nagyobb gyártóknak azért éri meg beszállni ebbe az üzletbe, mert egy saját operációs rendszer kifejlesztése jóval költségesebb lenne, ami hatalmas hátrány jelentene az árversenyben a piacon.

A kisebb gyártóknak azért éri meg, mert az ingyenes operációs rendszerrel és olcsó, nyomott áru készülékekkel piacra tudnak kerülni, bevezetni saját márkanevüket.

1.2.1.3 A Google érdekei

A Googlenek azért éri meg, mert így egy új szolgáltatással bővült a portfóliója. Belépett a mobil piacra. Létrehozta a fizetős Android Marketet, ezáltal az elsődleges szoftver eladóvá lépett elő az Androidot futtató készülékek piacán. Továbbá, növelte korábbi webes szolgáltatásainak a forgalmát azáltal, hogy integrálta őket az Androidba.

1.3 A legnagyobb konkurens piaci szereplők

1.3.1 Nokia

Nagy múltú távközlési eszközöket gyártó vállalat. Az új multimédiás igényekhez nem alkalmazkodott elég gyorsan, így részesedése az elmúlt években folyamatosan csökkent. A Symbian operációs rendszerben érdekelt, amely nem váltotta be a hozzáfűzött reményeket.

1.3.2 Microsoft

Informatikai nagyvállalat. Windows Phone operációs rendszerével ért el mérsékelt piaci sikereket. Szerződést kötött a Nokiával, így a jövőben Windows Phone 8 fog futni a Nokia készülékeinek jelentős részén.

1.3.3 Apple

Informatikai nagyvállalat. Az elmúlt években innovatív termékeivel nagymértékben megerősítette piaci részesedését. Saját zárt rendszerrel rendelkezik.

1.3.4 Research in Motion (RIM)

Nagy múltú távközlési eszközöket gyártó vállalat. A BlackBerry telefonokkal 2002-2010 között piacvezető volt. Az utóbbi években drasztikusan csökkent a piaci részesedése.

1.4 Az Android Története

Az Android operációs rendszert a Google 2005-ben vette meg, az azonos nevű cégtől. Az első Android operációs rendszert futtató telefon a HTC Dream volt 2008-ban. 2009-ben megnyílt az Android Market, ami az App Store hoz hasonlóan, egy virtuális software áruház. 2011 Februárjában megjelent az Android 3.0, amit kifejezetten nagyobb kijelzős eszközökre, tábla-gépekre optimalizáltak, így az Android belépett a táblagépek piacára.

1.5 Az Android sikerének titka

Az Android sikerének titka a nyíltság és a kicsi, de jól skálázódó hardver igény, ami lehetővé teszi, mind a színvonalas felhasználói élmény létrehozását, mind az olcsó készülékek gyártását. A kisebb gyártók olcsó készülékeket, a nagyobbak drága minőségi eszközöket gyártottak, így a piaci igények teljes skáláját lefedték.

1.6 Piaci Trendek

3. ábra Az okos telefonok piacának megoszlása

A 3. ábrán az okostelefonok piacának legnagyobb szereplőinek részesedése látható 2007-től 2011-ig.

Látható, hogy az Android 2011-re piacvezetővé lépett elő az okos telefonok piacán. [3] [4]

1.7 Miért éppen Android?

Piacvezető lett az okos telefonok piacán.

Egy tőkeerős vállalat a Google irányítja a fejlesztését, továbbá a mögötte álló szövetség egy hatalmas szakmai tudást biztosít.

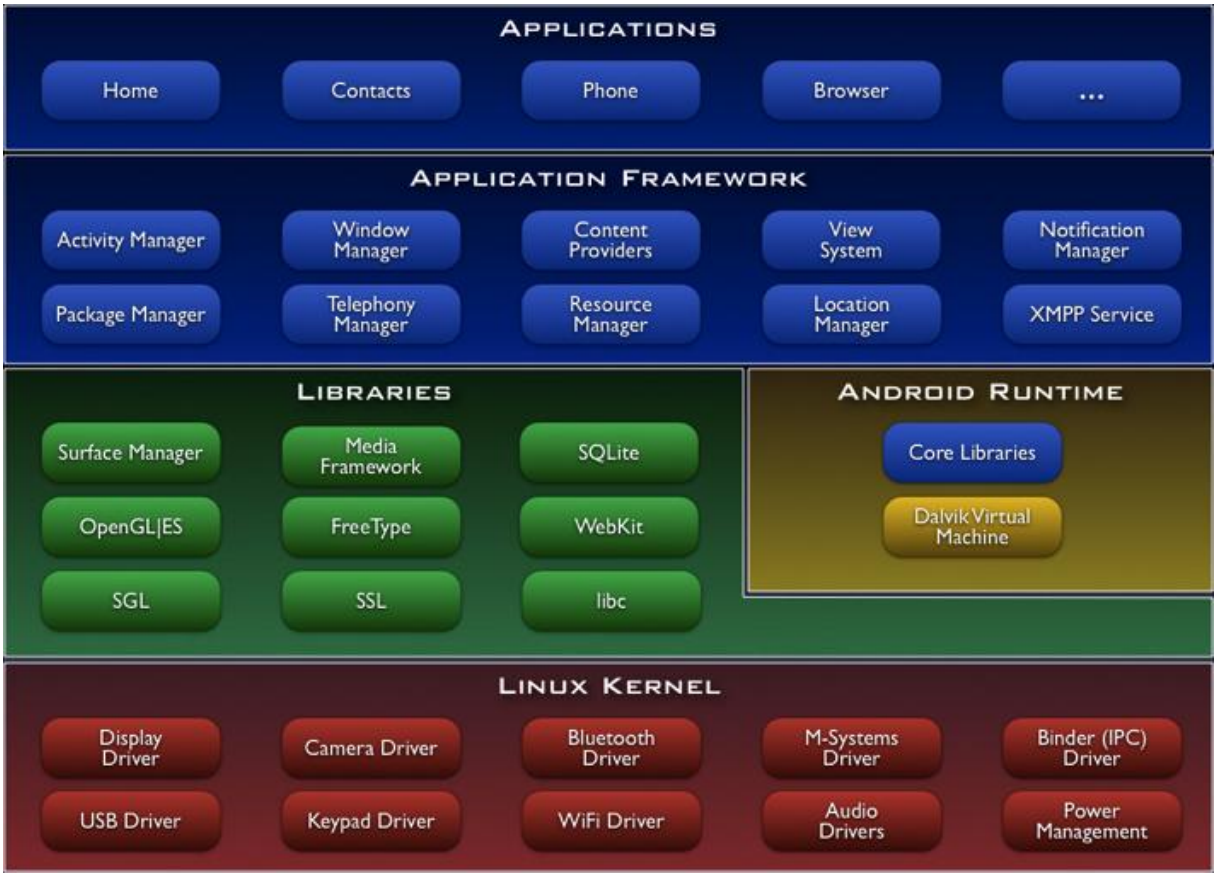
Nyílt rendszer, így nem kell drága licenc díjakat fizetni.

Az Intel 2011 Szeptemberében bejelentette, hogy hivatalosan támogatni fogja az Androidot, így az x86-os architektúra is megnyílik az Android előtt.

A jövő következő nagy dobása lehet az okos tévé, aminek egyik úttörője a Google TV, ami Android operációs rendszert futtat, már elérhető.

2 Technológiai ismertető

2.1 Android Architektúra



4. ábra Android architektúra

A 4. ábrán az Android architektúrájának vázlata látható. [5]

2.1.1 Kernel

Az Android alapja egy monolitikus linux kernel. A megfelelő drivereket a készülék gyártónak kell elkészítenie. A mobil eszközökön oly kritikus teljesítményszabályozás is itt található.

2.1.2 Programkönyvtárak és Szolgáltatások

A kernel fölött futnak a programkönyvtárak és a szolgáltatások, mint például SSL, libc vagy SQLite. Az itt található komponensek natív módon futnak.

2.1.3 Android futtatókörnyezet

Az Android futtatókörnyezet függ a programkönyvtáraktól és a kerneltől is.

Benne található a **Dalvik virtuális gép**. **Fontos, hogy ez nem kompatibilis a Sun virtuális gépével**, az utasításkészlete teljesen más, ezért más bináris programokat futtat. A Java az Androidnak csak a programozási nyelve. Ezzel a virtuális géppel lesz hardver független Android.

2.1.4 Android keretrendszer

Az eddigi rétegeket teljesen elfedi a Java-ban írt keretrendszer azáltal, hogy az alacsonyabb szinten lévő komponensekhez Java csomagolót ad. Az Android operációs rendszer

szolgáltatásainak jelentős része is itt található Java-ban megírva, mint például az Activity Manager, ami a felhasználói felülettel rendelkező alkalmazások futtatását felügyeli.

2.2 Elosztott szoftvertervezés

Elosztott szoftvertervezés esetén kulcskérdés, hogy hogyan oldjuk meg az adatok, metódus hívások továbbítását.

2.2.1 Saját protokoll

Implementálhatunk saját protokollt, ami közvetlenül ráépül a hálózati erőforrásokra, például TCP/IP socketekre.

Előnye, hogy kicsi az erőforrás igénye és biztosan megfelel az elvárásainknak.

Hátránya, hogy teljesen egyedi, így minden esetben újra meg kell írni, újra meg kell oldani a nagyon hasonló problémákat, mint például az átküldendő adatszerkezet létrehozása, bájtokba szerializálása. Ez a többletmunka megnöveli a hibázás esélyét is.

2.2.2 Keretrendszer

Használhatunk egy keretrendszert, ami elfedi a hálózat alacsonyszintű kezelését, és egy jól definiált, egyszerűsített felületet nyújt felénk.

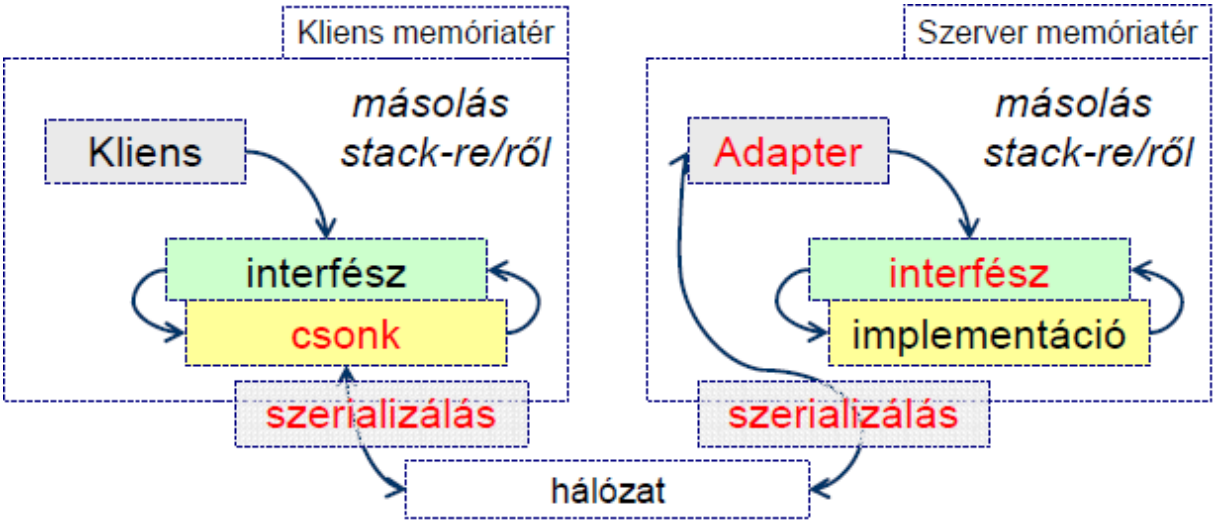
Előnye, hogy magas szinten tudjuk kezelni az elosztottságot, például bájtok helyett objektumokat tudunk átküldeni. Tud egyéb szolgáltatásokat is nyújtani, mint titkosítás, azonosítás.

Hátránya, hogy erőforrás igényes és, hogy nem biztos, hogy támogatja a kívánt platformot, vagy kommunikációs megoldást.

2.3 Távoli Eljáráshívás, Remote Procedure Call

A távoli eljáráshívás egy processzek közötti kommunikációt megvalósító technológia, ami lehetővé teszi olyan eljárások meghívását, amik más címtartományban vannak, tipikusan más gépen is, amit egy hálózaton keresztül ér el.

2.3.1 Működés



5. ábra RPC

Az 5. ábrán az RPC magas szintű vázlata látható.

A csonk feladata, hogy a kliens oldalon proxyként implementálja a megadott interfészt. Metódushívás esetén a paramétereket elküldi a hálózaton.

A szerializálás feladata a paraméterek átalakítása hálózaton elküldhető formába, illetve vissza alakítása.

Az adapter feladata, hogy a vett paraméterekkel továbbhívjon a megadott interfészt implementáló objektumhoz, majd a visszatérési értékét visszaküldje a hálózaton.

2.3.2 Új Problémák

A távoli eljáráshívás új, korábban nem jelentkező problémákat vet fel.

2.3.2.1 Memóriakezelés

A memóriakezelési problémák gyökere a különböző címtér. Primitív típusok esetén nincs probléma, másolatuk minden érintett címtérben lesz.

Összetett és Pointer típusok esetén deep copy jön létre minden esetben. Saját létrehozó és felszabadító függvények szükségesek, amik által megoldható, hogy a különböző címterekben-szinkronban legyenek az objektumok.

2.3.2.2 Hálózati hibák

A hálózati hibák gyökere a nem megbízható hálózat.

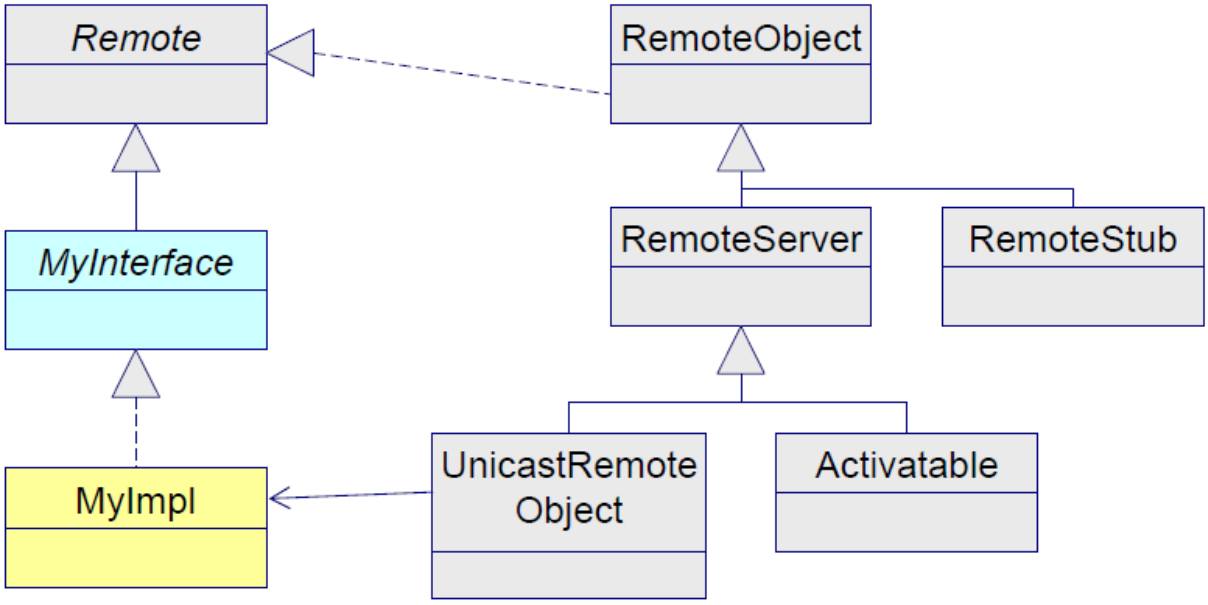
Nem triviális kérdés, hogy hálózati hiba esetén jelezzük a hibát, vagy újra megpróbáljuk a távoli eljáráshívást. Utóbbi esetben újabb probléma, hogyan védjük ki a többszöri végrehajtást.

2.4 Remote Method Invocation

A RemoteMethod Invocation a Java RPC implementációja.

Az RMI a Java szerializálást használja.

2.4.1 Szerver oldali architektúra



6. ábra RMI Szerver oldali architektúra

A 6. ábrán az RMI szerver oldali architektúrája látható.

2.4.1.1 Remote

A **java.rmi.Remote** egy jelölő interfész, aminek a leszármazott interfészeinek a metódusai lesznek távolról elérhetőek.

Megkötés, hogy minden távolról elérhető metódusnak jeleznie kell, hogy **java.rmi.RemoteException**t dobhat, ami a távoli eljáráshívás során keletkező hibák ősosztálya. További megkötés, hogy a távolról elérhető metódusok paraméteri és visszatérési értékei csak primitív, szerializálható vagy **RemoteStub** típusúak lehetnek.

2.4.1.2 RemoteObject
A **RemoteObject** a távoli objektumok ősosztálya.

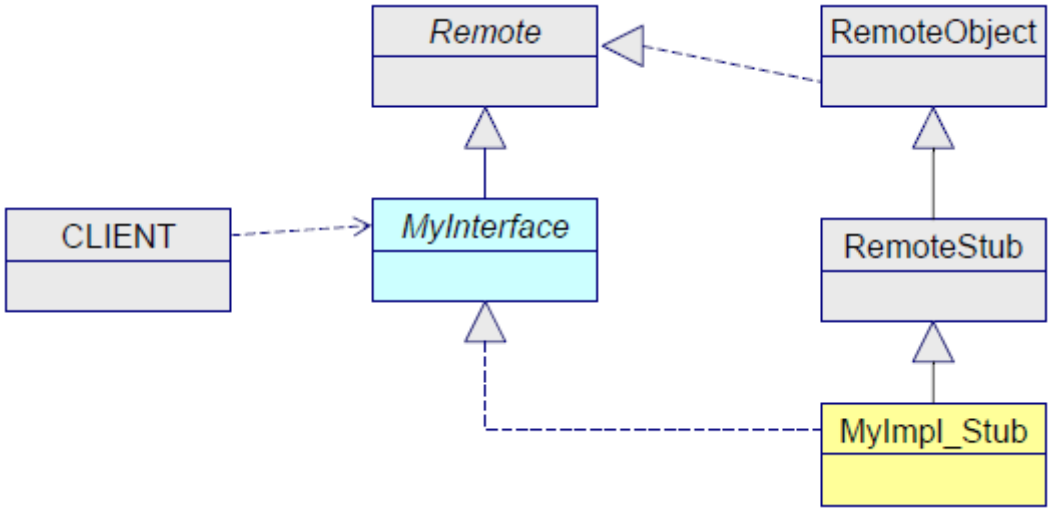
2.4.1.3 RemoteStub
A **RemoteStub** osztály referál egy **RemoteObject** távoli objektumra. A kliens oldalon van proxyként viselkedő implementációja.

2.4.1.4 RemoteServer
A **RemoteServer** osztály a távoli objektumok létrehozását és exportálását végzi.

2.4.1.5 UniCastRemoteObject
Az **UnicastRemoteObject** osztály a **RemoteServerből** származik feladata a **RemoteObjectek** exportálása, távolról elérhetővé tétele és a hozzátartozó **RemoteStubok** visszaadása.

2.4.1.6 Activatable
Az **Activatable** osztály a perzisztens elérésű **RemoteObjectek** őse. A szerver újraindítása után is elérhetőek a **RemoteStubjai**.

2.4.2 Kliens Oldal



7. ábra Kliens oldali RMI architektúra

A 7. ábra az RMI kliens oldali architektúráját mutatja.

A kliens oldalon jelenik meg a **RemoteStub** leszármazottja, ami proxyként viselkedik a szerver oldali implementáció felé.

2.5 Common Object Request Broker Architecture

A Corba egy elosztott objektumorientált architektúra, amit az Object Management Group hozott létre.

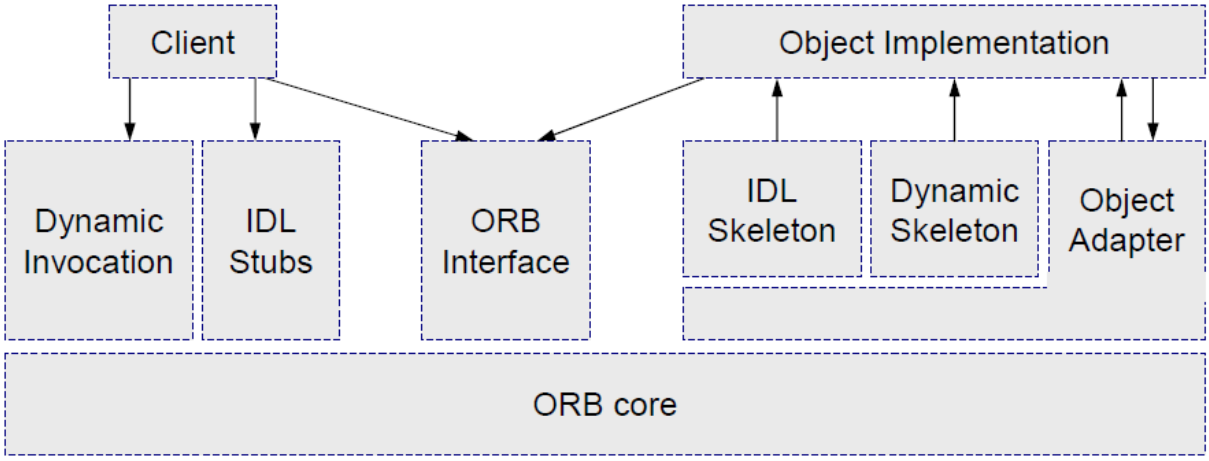
A Corba célja, hogy egy platform és nyelv független keretet biztosítson az elosztott rendszerek számára. Ezt az elosztott problémák megoldásainak szabványosításával érte el.

Rögzítette az adatátvitelt, a végpontok elérését és az implementációs osztályok csatlakozásának módját.

Megadott egy absztrakt protokollt a hálózat kezelésére, a General Inter-ORB Protocol-t.

Létrehozott egy közös nyelvet, az Interface Description Language-t, ami rögzítette az interfészek leírásához szükséges alapfogalmakat. A legtöbb natív nyelvre létezik az IDL-nek szabványos fordítója.

2.5.1 Corba Architektúra



8. ábra a Corba architektúra

Az 8. ábrán a Corba architektúrája látható.

A kliens oldalon az **IDL Stub** az **ORB Interface** előre legenerált platform specifikus implementációja, aminek feladata, hogy a paramétereket szabványos formátumba csomagolja a hálózaton való elküldéshez. A **Dynamic Invocation** feladata ugyanez, de itt nincs előre létrehozott osztály, hanem futási időben jön létre minden.

A szerver oldalon az **IDL Skeleton** az **ORB Interface** előre legenerált platform specifikus implementációja, aminek feladata, hogy a paramétereket fogadja, kicsomagolja a szabványos formátumból és platform specifikussá alakítsa. A **Dynamic Skeleton** feladata ugyanez, de itt nincs előre létrehozott osztály, hanem futási időben jön létre minden.

Az **Object Adapter** feladata az implementációs osztályok kezelése, a távoli metódushívások irányítása.

Az **ORB Core** felelős a kérések és a válaszok hálózati szállításáért.

3 Advanced Java Remote Procedure Call

Az általam létrehozott keretrendszer az Advanced Java Remote Procedure Call, röviden AJAR.

3.1 A probléma

Az Android főleg a kliens szerver architektúrát támogatja. Weblapokat kérhetünk le http kérésekkel, vagy Web szolgáltatásokat hívhatunk meg http feletti JSON vagy SOAP üzenetekkel, de ezek nem alkalmasak peer to peer kommunikációra, ugyanis http esetén a névfeloldás miatt problémába ütközünk, mert nem tudnánk elég gyorsan frissíteni a dns rekordokat, web szolgáltatások esetén meg nincs elég erőforrás egy mobil eszközben a keretrendszer futtatásához.

Az Android magas szinten nem támogatja sem a mobil eszközök közötti kommunikációt, sem a távoli eljárást megvalósító technológiákat, mint az RMI vagy a CORBA.

3.2 Megoldás

3.2.1 Követelmények a keretrendszerrel kapcsolatban

A keretrendszer létrehozásának első lépésében meghatároztam a követelményeket.

3.2.1.1 Android kompatibilitás és Java

A keretrendszer fusson Android platformon és legyen Java a nyelve az egyszerű használat miatt.

3.2.1.2 Dinamikus működés

A keretrendszer dinamikusan kezelje az elosztottságot, ne kelljen több lépcsőben létrehozni legenerált osztályokat, amik hozzáillesztik a saját kódunkat a keretrendszeréhez.

3.2.1.3 Proxy, stub átadása

A távoli objektumok legyenek átadhatóak.

3.2.1.4 Átvitel elfedése

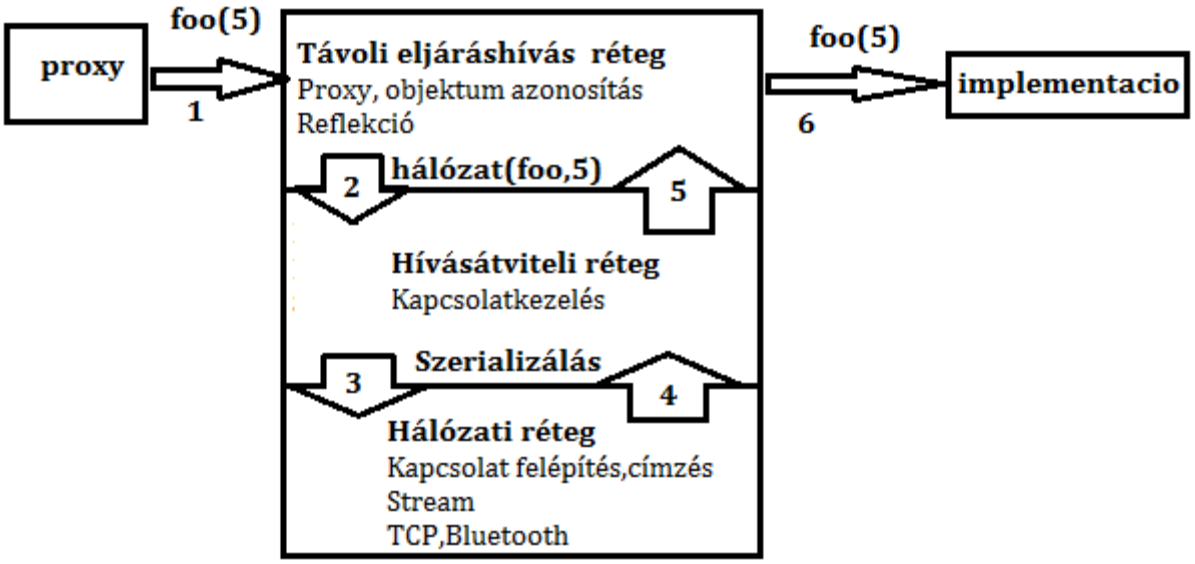
Az adatátvitelt teljesen fedje el a keretrendszer. Objektumokat kezeljen, ne bájtokat.

3.2.1.5 Bluetooth és TCP/IP

A keretrendszer támogassa mind a Bluetooth feletti, mind a TCP/IP feletti kommunikációt.

3.2.2 Általános megoldás

A keretrendszer tervezése során próbáltam a rétegszerkezetre törekedni annak érdekében, hogy a keretrendszer moduláris és újrahasznosítható legyen. Interfészeket és absztrakt osztályokat definiáltam a rétegek kapcsolódási pontjaihoz. A tiszta rétegszerkezetet nem sikerült elérni, ugyanis a címzés és a hibakezelés ezt nem tette lehetővé.



9. ábra AJAR rétegek

A 9. ábra a tervezett rétegek vázlatát tartalmazza. A foo(5) meghívását a távoli eljáráshívás réteg objektumokba kódolja, alakítja és továbbhív a hívásátviteli rétegbe hálózat(foo,5). A hívásátviteli réteg a megkapott foo és 5 objektumokat a kliens oldalon szerializálja. A hálózati réteg a szerializált bináris adatokat eljuttatja a célcímre, szerverre. A szerver oldalon a vett adatokat a hívásátviteli réteg deszerializálja és továbbhív a távoli eljárásrétegbe hálózat(foo,5). A távoli eljárásréteg megkeresi híváshoz tartozó implementációs osztályt és meghívja rajta a foo(5) metódust.

A hálózati réteg feladata a kapcsolatok és a hozzátartozó streamek felépítése és átadása a hívásátviteli rétegnek technológia független módon.

A hívásátviteli réteg az adatátvitelt fedi el egy speciális interfész segítségével, amihez egy általános metódus tartozik. Az interfész egyik implementációja a hívásátviteli rétegben a kiindulási objektum, a másik a távoli eljáráshívás rétegben a cél objektum. A hívásátviteli réteg feladata, hogy az egyik címtérben meghívott metódust meghívja egy másik címtérben egy megadott objektumon. A metódushívás során az objektumokat szerializálja a megkapott kapcsolat streamjébe.

A távoli eljárás réteg fedi el az egész távoli metódushívást proxyk létrehozásával. Feladata, hogy egy konkrét proxy objektumon meghívott konkrét metódust lefuttasson a megadott implementáción egy másik címtérben. Kódolja a metódushívást és a paramétereket is. A hívásátviteli réteg interfészét implementálja, hogy megkaphassa a hálózaton átküldött objektumokat.

A keretrendszer tervezéséhez megnéztem milyen java osztályokat támogat az Android, és azokból építettem fel a keretrendszert, így bármilyen Android alkalmazásból lehet használni azt.

A dinamikus működés során arra van szükségünk, hogy egy szokásos Java metódushívás kezdeményezése során egy általánosan implementált metódus hívódjon meg, amely képes arra, hogy stub-ként viselkedve a hívást a hálózati kapcsolaton keresztül egy távoli objektumhoz juttassa. Ez a szokásos keretrendszerek esetén (CORBA,RMI) tipikusan kódgenerálással van megoldva. Az én megoldásomban azonban a Java egy beépített dinamikus proxy osztályát (java.lang.reflect.Proxy) használtam fel, aminek a működését saját osztályokkal bővítettem ki. Ezáltal lehetővé vált, hogy kódgenerálás nélkül is a CORBA, RMI funkcionalitását biztosítsuk.

Proxy átadás során a létező kapcsolatokat nem lehet szerializálni, ezért amikor egy proxy osztályt átadunk az összes olyan információt is át kell adnunk, ami a referált objektum eléréshez szükséges. Tipikusan ez egy cím és azonosító. Ennek érdekében egy olyan proxy osztályt terveztem, ami tartalmazza a referált objektum címét és azonosítóját. A címet egy olyan absztrakt objektumban tárolja, amit a hálózati réteg aktuális technológiája értelmezni tud és általa fel tudja építeni a kapcsolatot.

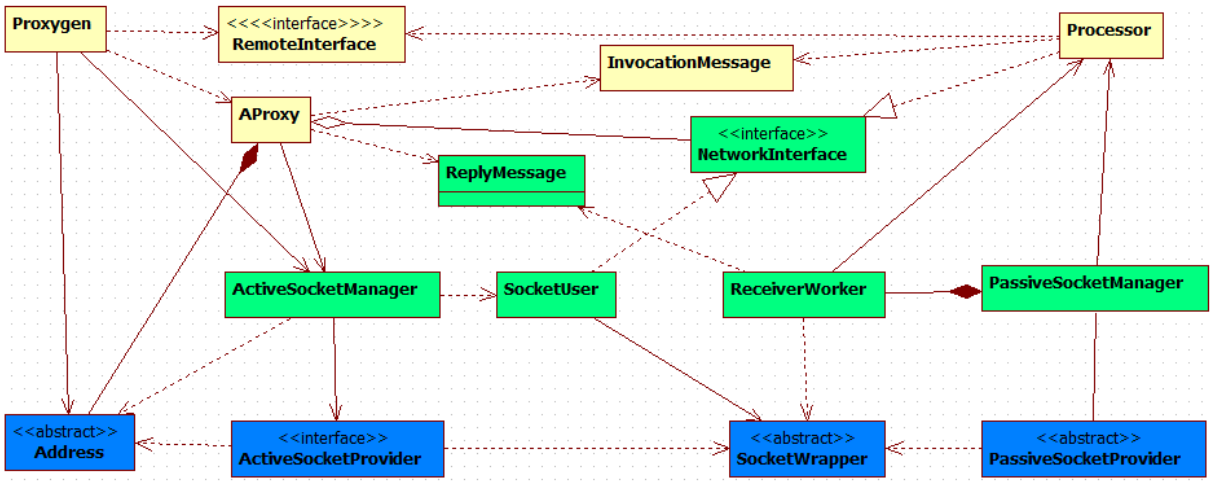
A TCP/IP és Bluetooth támogatásához csak meg kell valósítani a hálózati réteg absztrakt osztályait és interfészeit. Azért a TCP/IP és a Bluettotht választottam a keretrendszer elsődleges hálózati rétegének, mert az Androidot futtató készülékek általában Wifi és Bluetooth segítségével kapcsolódnak a külvilághoz.

A kapcsolatkezelő osztályokat viselkedés szerint csoportosítottam. Az aktív osztályok valamilyen kérésre, metódushívásra cselekednek. A passzív osztályok erőforrás takarékosan passzívan várakoznak egy eseményre, ami ha bekövetkezik, akkor cselekednek. Ilyen esemény lehet egy új kapcsolat létrejötte, vagy adat érkezése. A passzív osztályokat külön szálban várakoztatom, amik egységes és takarékos kezeléséhez poolt használok.

Minden proxy objektumhoz egy dedikált csatorna tartozik. Ennek előnye, hogy nem kell a különböző objektumok hálózaton küldött üzenetit objektumcímzéssel ellátni, ami csökkenti a hálózaton elküldött adatok mennyiségét, továbbá nem kell a címeket feloldani, ami meg a procedurális overheadet csökkenti. Hátránya viszont a megnövekedett csatornaszám, ami a projekt során nem okozott problémát.

A konfiguráció dinamikus változtathatósága és a különböző feladatot ellátó osztályok közötti kapcsolat központi kezelésére manager osztályokat hoztam létre.

3.3 Architektúra



10. ábra Az architektúra

A 10. ábrán a keretrendszer architektúrája látható. Kékkel jelölve a hálózati réteg, zölddel a hívásátviteli és sárgával távoli eljáráshívás rétegeit.

3.3.1 NetworkInterface

A **NetworkInterface** a hálózati hozzáférést elfedő interfész. A hívásátviteli réteg speciális interfésze.

Távoli eljáráshívás során, a kliens oldalon egy ezt realizáló objektumon történik egy meghatározott metódushívás, amit a szerver oldalon egy ezt megvalósító objektum kap meg.

3.3.2 Address

Az **Address** osztály absztrakt ősosztálya a címzést megvalósító osztályoknak, amik alapján el lehet érni egy távoli metódushívás célpontját. A konkrét hálózati rétegek, mint például TCP vagy Bluetooth megvalósítják saját címzésük szerint.

3.3.3 InvocationMessage

Az **InvocationMessage** osztály egy távoli eljárás során átküldendő adatokat tartalmazó szerializálható osztály.

Tartalmazza a meghívandó metódus azonosításához szükséges összes információt és a konkrét argumentumokat, amikkel meg kell hívni a metódust.

Továbbá tartalmazhat egy ID-t, ami az objektum példányt azonosítja, amin a metódust meg kell hívni.

Ha nem tartalmaz ID-t, akkor lehetőség van arra, hogy a metódushívás idejére az azonosított osztályból példányosodjon egy objektum paraméter nélküli konstruktorral.

Az **AProxy** küldi az **InvocationMessage-eket** a **Processor**nak.

3.3.4 ReplyMessage

A **ReplyMessage** osztály egy távoli eljárás eredményét, visszatérési értékét tartalmazó szerializálható osztály. Becsomagol egy objektumot, ami a visszatérési értéket reprezentálja és egy hiba objektumot, ami az távoli eljárás során keletkező hibát tartalmazza. A szerveroldalon a keretrendszer határain kívül bármilyen hiba keletkezhet, ezt a hibát a legkésőbb a hívásátviteli rétegnek el kell kapnia és jelezni a kliens oldal felé. A **Receiverworker** küldi **ReplyMessage**-eket az **AProxy** objektumnak**.**

3.3.5 Processor

A **Processor** osztály azonosítja távoli eljáráshívás során a megfelelő objektumot, amin meghívja a megfelelő metódust.

A **Processorba** osztályok példányait lehet beregisztrálni, amiknek reflection segítségével kigyűjti azon interfészeit, amik a **RemoteInterface** jelölő interfész leszármazottai. Ezen interfészek metódusai lesznek távolról elérhetőek. Végül elmenti a metódus, példány párokat.

Távoli eljáráshívás során egy **InvocationMessageben** elküldött azonosító paraméterek alapján megnézi, hogy az elmentett metódusok között megtalálja e, a meghívandó metódust. Ha megtalálja, akkor reflection segítségével meghívja a metódushoz tartozó példányon a metódust, az **InvocationMessageben** lévő argumentum értékekkel.

Beregisztrálás során azonosítót, **ID**-t lehet rendelni a példányokhoz. Az **ID** szerepe, az azonos osztályú példányok megkülönböztetése.

ID nélküli regisztrálás esetén kötelező, hogy legyen a példányhoz osztályának paraméter nélküli konstruktora, ugyanis ebben az esetben, minden hívás esetén példányosodik egy új objektum.

A **Processor** kétféle módon működhet. Az egyik üzemmódban, ha nem talál az **ID-**hoz példányt egyből hibát dob. A másik üzemmódban **ID**-nélkül megismétli a keresést. Ha ismételten nem talált példányt, akkor hibát dob, ellenkező esetben létrehoz egy új objektumot paraméter nélküli konstruktorral és folytatja az eljárást.

3.3.6 SocketWrapper

A **SocketWrapper** osztály absztrakt ősosztálya a különböző streameket nyújtó osztályoknak. Feladata elfedni, hogy konkrétan milyen osztály szolgáltatja a streameket. Példányai létező streameket reprezentálnak, amikbe lehet írni, olvasni.

3.3.7 ReceiverWorker

A **ReceiverWorker** osztály egy **SocketWrapper** használó osztály. Deszerializálja a kéréseket és szerializálja a válaszokat.

Feladata, hogy passzívan várakozva figyelje a megadott **SocketWrapper**t. Az érkező adatokat deszerializálja és velük meghív egy **NetworkInterfacet**, aminek visszatérési értékét szerializálja és visszaküldi.

Hiba esetén előállít egy **ReplyMessaget**, amibe belerakja a hiba objektumot és elküldi a kliens oldal felé.

3.3.8 ActiveSocketProvider

Az **ActiveSocketProvider** interfésze, a kérésre **SocketWrapperek** előállító osztályoknak.

Realizáló osztályainak feladata a címfeloldás, a kapcsolatok kezdeményezése és a létrejövő kapcsolatokhoz csomagoló **SocketWrapperek** létrehozása.

3.3.9 PassiveSocketProvider

A **PassiveSocketProvider** osztály absztrakt ősosztálya a várakozva **SocketWrappereket** előállító osztályoknak.

Leszármazott osztályainak feladata, a kapcsolatok fogadása és a létrejövő kapcsolatokhoz csomagoló **SocketWrapperek** létrehozása, amiket átad egy **PassiveSocketManagernek**.

3.3.10 SocketUser

A **SocketUser** osztály egy **SocketWrapper** használó osztály, ami megvalósítja a **NetworkInterfacet**.

Feladata, hogy metódushívás esetén szerializálja az argumentumokat és elküldje a megadott **SocketWrapperen** keresztül. Ezek után passzívan várja a választ, amit deszerializálás után visszaad.

3.3.11 ActiveSocketManager

Az **ActiveSocketManager** osztály **NetworkInterface** példányokat hoz létre kérésre.

Feladata, hogy kérésre létrehozzon hálózati hozzáférést biztosító **NetworkInterface**-t implementáló **SocketUsereket**, amiknek a **SocketWrappert** egy megadott **ActiveSocketProvidertől** kéri**.**

Kérésenként meg kell adni egy **Address** objektumot, amit továbbít az **ActiveSocketProvider** felé.

A proxy osztályok exportálhatósága miatt, meg kell adni egy statikus alapértelmezett **ActiveSocketManagert**, amit ismeretlen környezetben elérnek a proxy osztályok.

3.3.12 PassiveSocketManager

A **PassiveSocketManager** osztály kezel egy megadott **PassiveSocketProvidert** és kezeli a **ReceiverWorkereket**.

Feladata, hogy elindítson és futtasson egy megadott **PassiveSocketProvidert**, amitől ha megkap egy **SocketWrappert**, akkor létrehoz és elindít egy **ReceiverWorkert**, aminek átadja a **NetworkInterface** referenciáját.

3.3.13 Proxygen

A **Proxygen** osztály **AProxykat** hoz létre. Lehet alapértelmezett **Address** és **ActiveSocketManagert** beállítani, amit **AProxy**k generálása során beállít, ha szükséges.

AProxy osztályok generálásakor, meg lehet adni konkrét **Addresst**, **ActiveSocketManagert** és ID-t, ami példányok azonosítására szolgál és az **InvocationMessageben** elküldésre kerül.

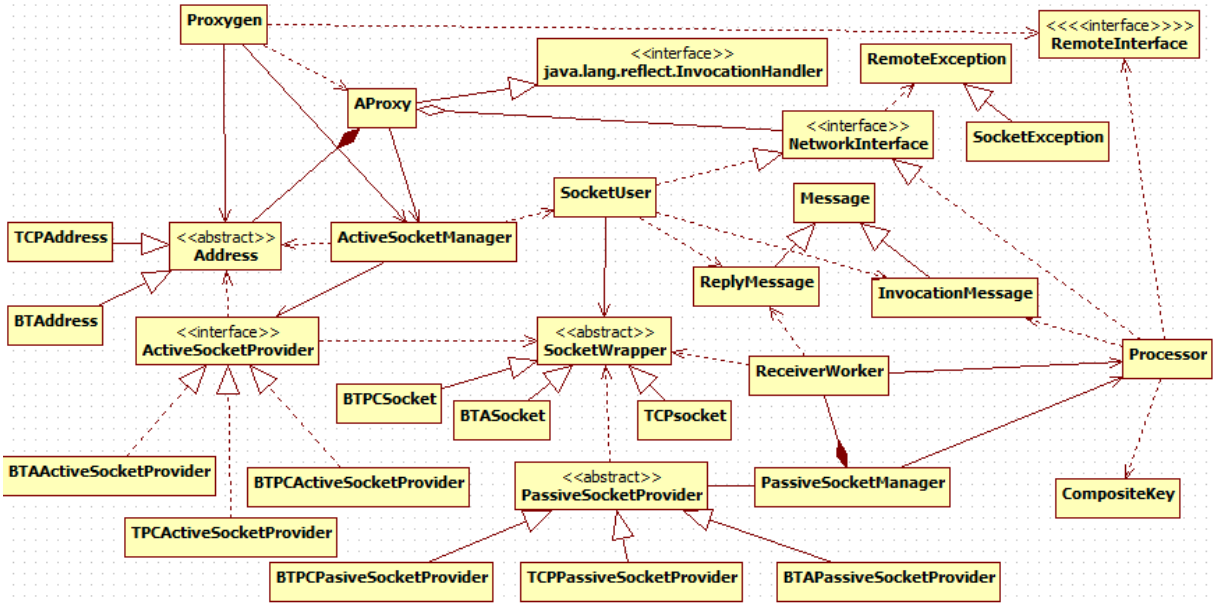
3.3.14 AProxy

Az **AProx** osztály viselkedik proxyként a távoli eljárás során, ezáltal teljesen elfedi a programozó elöl, hogy az eljárás hol fut le. Tárolja, hogy milyen címre kell eljuttatni a metódushívásokat, **Address**, továbbá, hogy melyik objektumtól lehet hálózati elérést kérni, **ActiveSocketManager**.

Csak akkor kér hálózati elérést biztosító objektumot, **NetworkInterfacet**, ha még nincs neki.

Amikor meghívnak rajta egy metódust, akkor reflection segítségével összerak egy csomagot, **InvocationMessage**, és elküldi a hálózatra, egy **NetworkInterfacen** keresztül.

3.4 Részletes Architektúra

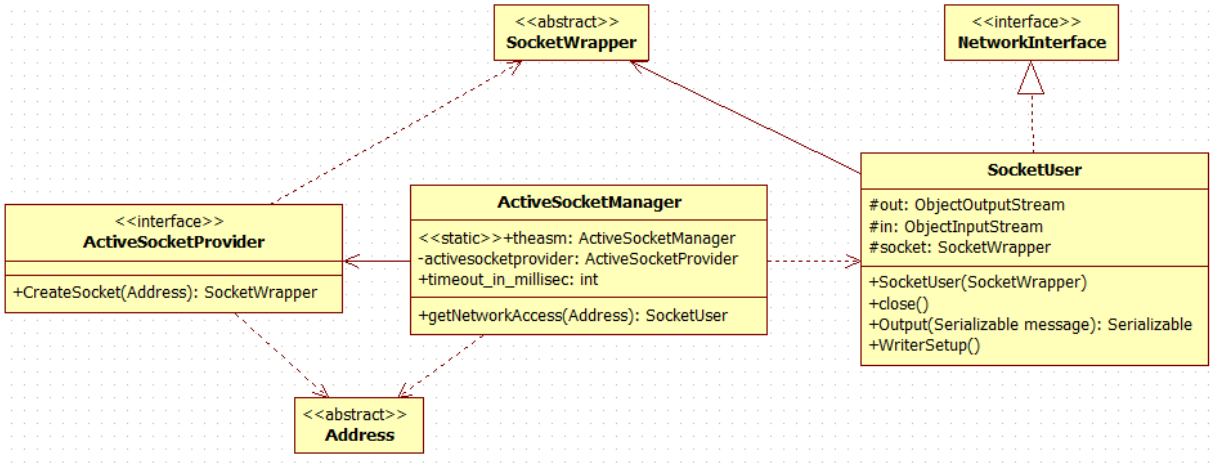


11. ábra a részletes megoldás osztályai és interfészei

A 11. ábrán a részletes megoldás osztályait és interfészeit tartalmazza, metódusok és attribútumok nélkül.

3.4.1 Active package

Az **Active** a packagebe tartoznak a kérésre cselekvő osztályok.



12. ábra Az Active package

A 12. ábra, az Active package osztályait mutatja be részletesen.

3.4.1.1 ActiveSocketProvider

Az **ActiveSocketProvider** a kérésre **SocketWrappert** előállító osztályok interfésze.

3.4.1.1.1 SocketWrapper CreateSocket(Address address) throws IOException

A **CreateSocket** metódus, a paraméterében átadott **Address** címhez ad vissza egy **SocketWrappert**.

3.4.1.2 ActiveSocketManager

3.4.1.2.1 public static ActiveSocketManager theasm

A ***theasm*** attribútum referencia az alapértelmezett **ActiveSocketManager** objektumra. Ha egy **AProxynak** nem állítunk be **ActiveSocketManagert**, akkor megpróbálja a ***theasm***et elérni.

3.4.1.2.2 ActiveSocketProvider activesocketprovider

Az **activesocketprovider** attribútum egy referencia egy **ActiveSocketProvider** objektumra, ami kérésre **SocketWrappereket** nyújt.

3.4.1.2.3 ActiveSocketManager(ActiveSocketProvider activesocketprovider)

Konstruktor. Paramétere a **SocketWrappereket** nyújtó **ActiveSocketProvider**.

Továbbá ha nincs beállított alapértelmezett **ActiveSocketManager**, **theasm**, akkor beállítja az éppen létrejövőre.

3.4.1.2.4 NetworkInterface getNetworkAccess(Address address) throws IOException,RemoteException

A **getNetworkAcces** metódus hálózati hozzáférést biztosító objektumot ad vissza. Az ***address*** paraméterében meg kell adni a címet.

Először az **activesocketprovidertől** kér egy **SocketWrappert**, amihez létrehoz egy **SocketUsert** amivel visszatér.

3.4.1.3 SocketUser

3.4.1.3.1 ObjectOutputStream out

Az **out** attribútumba egy **ObjectOutputStream** referencia, amibe a távoli eljáráshívás során a szerializált objektumokat beleírjuk.

3.4.1.3.2 ObjectInputStream in

Az **in** attribútum egy **ObjectInputStream** referencia, ahonnan a távoli eljáráshívás során érkező válaszokat kiolvassuk és deszerializáljuk.

3.4.1.3.3 SocketWrapper socket

A **socket** attribútum szolgáltatja a **streameket**, amikre ráépül az **in** és **out** attribútum.

3.4.1.3.4 public SocketUser(SocketWrapper socket) throws IOException

Konstruktor, beállítja a **socketet** és meghívja a **WriterSetup**-ot.

3.4.1.3.5 public void WriterSetup() throws IOException

A **WriterSetup** metódus létrehozza az **out** és **in** attribútumok **ObjectOutput** és **ObjetInputStreamjeit** a beállított **socket Output** és **InputStreamjeire**.

Fontos, hogy először **outot** állítja be és utána rögtön hív rá egy **flusht**, mert az **ObjectOutputStream** beleír egy headert a **Streambe** és erre egy **ObjectInputStream** blokkolva vár. Ha erre nem ügyelünk, holtpont alakulhat ki.

3.4.1.3.6 void close() throws IOException

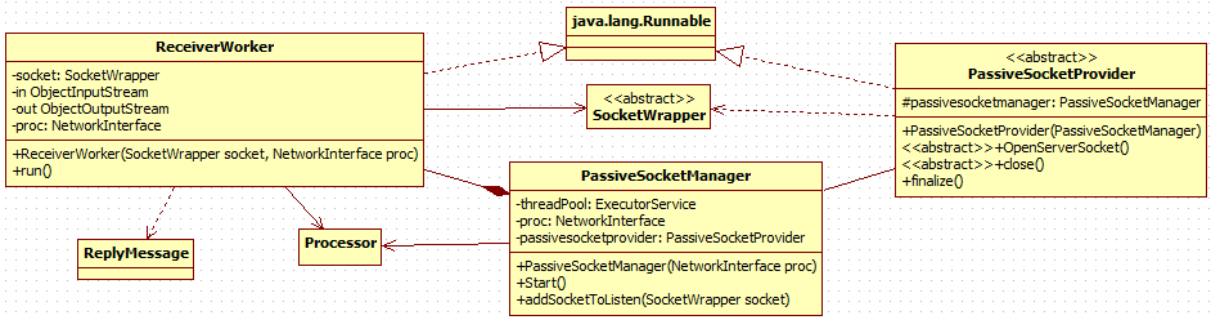
A **close** metódus meghívja a **socket close** metódusát, ezáltal lezárva azt.

3.4.1.3.7 Object Output(Serializabe message) throws RemoteException

Az **Output** metódus küldi el a hálózatra az adatokat. A **message**-t szerializálja az **out**-ba, majd az **in**-t blokkolva olvassa a válaszért.

3.4.2 Passive package

Ebbe a packagebe tartoznak a várakozva cselekvő osztályok.



13. ábra a Passive package

A 13. ábra a Passive package osztályait mutatja be részletesen.

3.4.2.1 PassiveSocketManager

3.4.2.1.1 PassiveSocketProvider passivesocketprovider

A **passievsocketprovider** attribútum referencia arra **PassiveSocketProvider** objektumra, amelyik az új kapcsolatok létrejöttekor **SocketWrappereket** ad át.

3.4.2.1.2 ExecutorService threadPool

A **threadPool** a különböző szálak futtatását oldja meg egységesen és gazdaságosan szálak újrahasznosításával. A **passivesocketprovidert** és a **ReceiverWorkereket** futtatja.

3.4.2.1.3 NetworkInterface proc

A **proc** attribútum referencia egy **NetworkInterface** objektumra, amit átad a **ReceiverWorkereknek**, hogy oda továbbítsák a deszerializált objektumokat a távoli eljáráshívás során. Tipikusan a proc referencia egy **Processorra** mutat.

3.4.2.1.4 PassiveSocketManager(NetworkInterface proc)

Konstruktor. A **proc** attribútumot állítja be.

3.4.2.1.5 void Start() throws IOException, RemoteException

A **Start** metódus inicializálja a **passivesocketprovidert** az **OpenServerSocket**() metódusának meghívásával. Ezek után átadja a **passivesocketprovidert** a **threadPoolnak** futtatásra.

3.4.2.1.6 void addSocketToListen(SocketWrapper socket)

Az **addSocketToListen** egy callback metódus. A **passivesocketprovider** hívja meg, ha létrejött egy új kapcsolat, ami köré a létrehozott **SocketWrapper** referenciáját paraméterként átadja.

3.4.2.2 ReceiverWorker

A **ReceiverWorker** a **Runnable** osztályból származik, ugyanis egy külön szálban fut, amiben figyeli a megadott **SocketWrappert**.

3.4.2.2.1 SocketWrapper socket

A **socket** attribútum referencia a figyelendő streameket tartalmazó **SocketWrapperre**.

3.4.2.2.2 ObjectInputStream in

Az **in** attribútum referencia egy **ObjectInputStream**re.

3.4.2.2.3 ObjectOutputStream out

Az **out** attribútum referencia egy **ObjectOutputStreamre**.

3.4.2.2.4 NetworkInterface proc

A **proc** attribútum referencia, a feldolgozás során a következő **NetworkInterface** objektumra. A kapott objektumokat neki továbbítjuk. A proc tipikusan egy **Processorra** mutat.

3.4.2.2.5 ReceiverWorker(SocketWrapper socket, NetworkInterface p)

Konstruktor, beállítja a **socket** és **proc** attribútumokat.

3.4.2.2.6 void run()

A **run** metódus elején létrehozzuk az **ObjectInput** és **ObjectOutput** Streameket a **socket Input** és **Output Streamjeire**.

Ez után egy végtelen ciklusban, blokkolva olvasunk az **in**-ből, majd a vett objektumot továbbítjuk a **proc** felé az **Output** függvényének meghívásával, ahol paraméterként átadjuk a vett objektumot-t. Végül az **Output** visszatérési értékét beleírjuk **out**-ba és kezdődik elölről a ciklus.

3.4.2.3 PassiveSocketProvider

A **PassiveSocketProvider** a **Runnable** osztályból származik, ugyanis egy külön szálban fut, amiben várja a kapcsolatokat.

3.4.2.4 protected PassiveSocketManager passivesocketmanager

A **passivesocketmanager** attribútum referencia arra **PassiveSocketManager** objektumra, ahova a létrejött új kapcsolatokat reprezentáló **SocketWrapper** objektumokat kell küldeni.

3.4.2.4.1 public PassiveSocketProvider(PassiveSocketManager passivesocketmanager)

Konstruktor, beállítja a **passivesocketmanager** attribútumot.

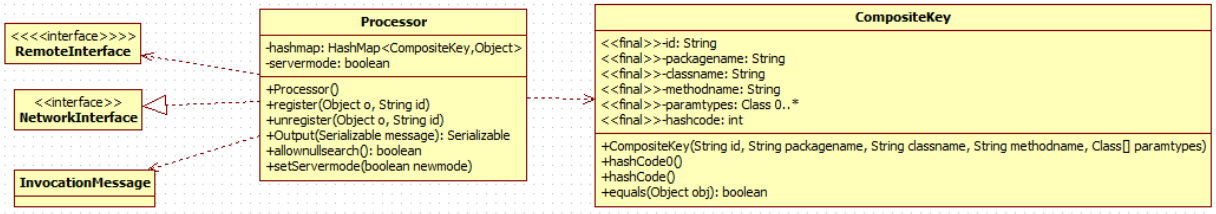
3.4.2.4.2 abstract void OpenServerSocket() throws RemoteException, IOException;

Az absztrakt **OpenServerSocket** metódus hivatott létrehozni és inicializálni azokat az objektumokat, amiken keresztül később a kapcsolatokat várni fogjuk. Tipikusan ez szerver socketek példányosítását jelenti.

3.4.2.4.3 abstract void close()

A **close** metódus, a kapcsolatokat váró objektumokat zárja be, hogy már ne jöhessen létre újabb kapcsolat.

3.4.3 Processor package



14. ábra a Processor package

A 14. ábra a Processor package osztályait ábrázolja részletesen.

A **Processor** package osztályainak célja, hogy azonosítsák a megfelelő cél objektumot távoli eljáráshívás során, és meghívják rajta a megfelelő metódust, a megadott paraméterekkel.

3.4.3.1 CompositeKey

A **CompositeKey** implementálja a Serializable jelölő interfacet, így szerializálható. Egy összetett kulcsot reprezentál, ami alapján azonosítani lehet egy távoli objektumot, és a meghívandó metódusát.

3.4.3.1.1 final String id

Az **id**, egy távoli objektumot azonosít.

Ha van értéke, akkor pontosan azonosítja az objektumot.

Ha nincs értéke (értéke null), akkor egy osztályt azonosít, ilyenkor metódushívásonként példányosodik egy objektum paraméter nélküli default konstruktorral.

Fontos, hogy ha az ID null is lehet, akkor legyen paraméter nélküli konstruktora a távoli objektumnak.

3.4.3.1.2 final String packagename

A **packagename**, egy távoli objektum packagenek a neve, azonosítására szolgál.

3.4.3.1.3 final String classname

A **classname**, egy távoli objektum osztályának a neve, azonosításra szolgál.

3.4.3.1.4 final String methodname

A **methodname**, egy távoli objektum metódusának a neve, metódusazonosításhoz kell.

3.4.3.1.5 final Class[] paramtypes

A **paramtypes**, egy Class tömb, amibe a meghívandó metódus paramétereinek **Class** leírója szerepel, metódusazonosításhoz kell.

3.4.3.1.6 final int hashcode;

A **hashcode** egy derivált attribútum a hashcode tárolására. Mivel minden attribútum final, ezért a hash kódot elég egyszer kiszámolni a konstruktorban. Teljesítményt nyerünk vele.

3.4.3.1.7 CompositeKey(String id, String packagename, String classname, String methodname, Class[] paramtypes)

Konstruktor, beállítja az attribútumokat. Lefuttatja a **hashCode0** függvényt, amiben kiszámolja a hash-t és eltárolja a hashcode attribútumban.

3.4.3.1.8 int hashCode0()

A **hashCode0** metódus, a hash kódot számolja ki.

3.4.3.1.9 int hashCode()

A **hashCode** metódus, visszaadja a hashcode attribútum értékét, amiben már korábban kiszámoltuk a hash-t. A hashmapban való tárolás miatt kell felüldefiniálni.

3.4.3.1.10 boolean equals(Object obj)

Az **equals** metódus két objektumról dönti el, hogy egyenértékű e. A hashmapban való tárolás miatt kell felüldefiniálni.

3.4.3.2 Processor

A **Processor** nyilvántartja a távoli objektumokat.

3.4.3.2.1 HashMap<CompositeKey, Object> hp;

A **hp** egy **HashMap**, amiben kulcs érték párokat tárolunk. A kulcs egy **CompsiteKey**, az érték egy objektum referencia, amelyik objektumban implementálva van az a távoli metódus, amit a **CompositeKey** azonosít.

3.4.3.2.2 boolean servermode

A **servermode** attribútum határozza meg, hogy mi történjen akkor, amikor az érkező ID-val rendelkező kéréshez nem tartozik objektum.

Igaz esetben, megpróbál ID nélküli objektumot keresni, ha megint nem talált, akkor hibát dob.

Hamis esetben, hibát dob.

3.4.3.2.3 Processor()

Konstruktor. A **hp** referenciához példányosít egy üres HashMapet és **servermodet** igazba állítja.

3.4.3.2.4 void register(Object o, String id)

A **register** metódus, a paraméterében szereplő **o** objektum interfészeit lekéri reflection segítségével. Ezek után kiszűri azokat az interfészeket, amik a **RemoteInterface** jelölő interfész leszármazottai, és ezeknek az interfaceknek a metóduasihoz létrehoz egy-egy **CompositeKey**-t.

Végül a **hp** HashMapbe beilleszti a kulcs érték párokat, ahol a kulcsok a létrehozott **CompositeKey**ek, az érték az **o**-Objektum.

Az **id** paraméter objektumazonosításhoz kell. Ha van értéke, akkor konkrét objektumot azonosít, ha nincs (értéke null), akkor osztályt, ekkor a paraméter nélküli konstruktor alapján példányosodik egy objektum, minden metódushívás során.

Fontos, hogy ha az ID lehet érték nélküli (null), akkor legyen a beregisztrált osztálynak paraméter nélküli konstruktora.

3.4.3.2.5 void unregister(Object o, String id)

Az **unregister** metódus, hasonlóan működik, mint a register metódus, ugyanis lekéri az **o** objektum interfészeit, megvizsgálja minden egyes interfészre, hogy a **RemoteInterface** jelölő interfész leszármazottja e, ezek után létrehozza a **CompositeKey**-eket a megfelelő interfészek metódusaihoz.

Végezetül eltávolítja a kulcs érték párokat, ahol a kulcsok a létrehozott **CompositeKey**-ek.

3.4.3.2.6 Object Output(InvocationMessage message) throws RemoteException, Exception

Az **Output** metódus a **NetworkInterfaceből** származó metódus. Megvalósításával a **Processor** beilleszthető a feldolgozási láncba.

A metódus, a message alapján létrehoz egy **CompositeKeyt**. Megnézi, hogy a **hp** HashMapben megtalálható e.

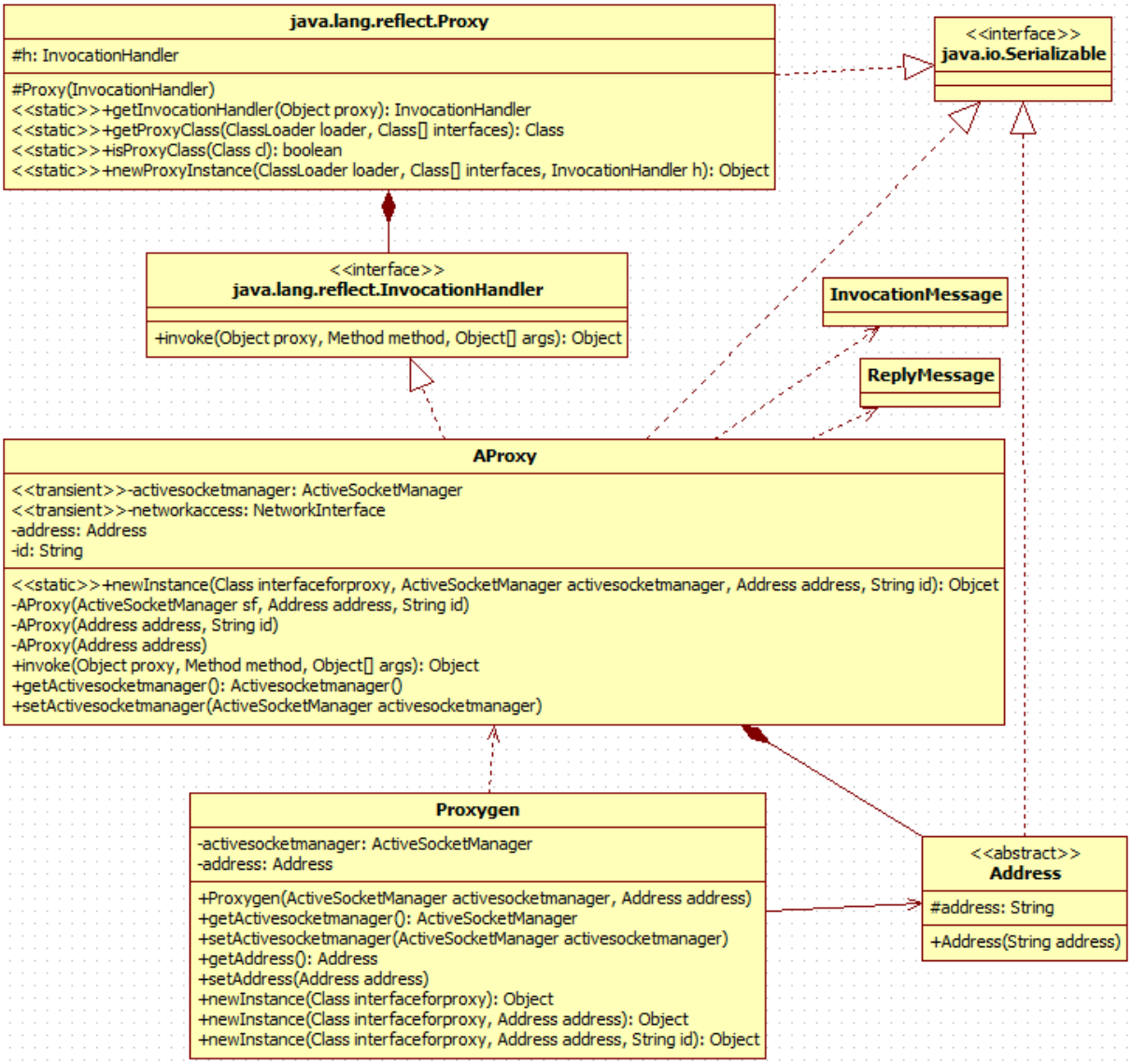
Ha nem található az adott kulcshoz objektum, akkor megnézi mi a szerver beállítása.

Ha a servermode logikai változó igaz, akkor létrehoz egy új kulcsot, amiben az **ID** értéke nem létezik (null). Ha ismételten nem találja, akkor hibát dob. Ha, a servermode logikai változó hamis, akkor hibát dob.

Ezek után, a **CompositeKey** alapján lekéri az objektumot. Ha van ID, akkor ezen az objektumon reflekszió segítségével meghívja a megfelelő metódust. A megfelelő metódust a message ben átadott paraméterek alapján keresi meg az objektumon. Ha nincs ID, akkor példányosít egy objektumot, default paraméter nélküli konstruktorral és meghívja rajta a megfelelő metódust. Végül a meghívott metódus visszatérési értékével visszatér.

Ha hiba keletkezett a metódus végrehajtása során, tovább dobja.

3.4.4 Proxy package



15. ábra A Proxy package

A 15. ábra a Proxy package osztályait mutatja be részletesen. Ezen felül még a releváns Java osztályokat is feltünteti.

A Proxy package azokat az osztályokat tartalmazza, amik egy elosztottan átadható proxyhoz kellenek.

3.4.4.1 java.lang.reflect.Proxy

A **java.lang.reflect.Proxy** osztály a JDK1.3-tól kezdve létezik. Ez egy proxy osztály, ami dinamikusan implementálja a megadott interfaceket. Az osztályon meghívott metódusokat egy **InocationHandlernek** továbbítja, aminek le kell kezelnie a metódushívásokat, gyakorlatilag az implementációt tartalmazza. [6]

Implementálja a Serializable interfacet, ez szükséges a hálózati átadás szempontjából.

3.4.4.1.1 protected InvocationHandler h

A **h** attribútum egy referencia egy **InvocatonHandler** objektumra, amit metódushívásonként meghív.

3.4.4.1.2 static InvocationHandler getInvocationHandler(Object proxy) throws IllegalArgumentException

A **getInvocationHandler** metódus, a paraméterében átadott proxy objektumnak adva vissza az **InvocationHandlerét**.

3.4.4.1.3 static Class getProxyClass(ClassLoader loader, Class[] interfaces) throws IllegalArgumentException

A **getProxyClass** metódus, visszaadja az osztály leíróját a megadott loaderből, a megadott interfacekkel rendelkező proxy osztálynak.

3.4.4.1.4 static boolean isProxyClass(Class cl)

Az **isProxyClass** metódus, megvizsgálja, hogy **cl** proxy osztály e.

3.4.4.1.5 static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h) throws IllegalArgumentException

A **newProxyInstance** metódus létrehoz egy új objektumot.

A ClassLoader **loader** lesz a proxy class ClassLoadere.

Az Class[] **interfaces** paraméter, egy tömb, amiben az implementálandó interface osztályok vannak.

Az **InvocationHandler h**, referencia arra az objektumra, ami a metódushívásokat lekezeli.

3.4.4.2 InvocationHandler

Az **InvocationHandler** interface a proxy osztályok metódushívásának implementációinak. [7]

Minden proxy példányhoz tartozik egy **InvocationHandler**. Amikor a proxy példányon meghívódik egy metódus, akkor a metódushívást a proxy lekódolja és meghívja az **InvocationHandler invoke** metódusát.

3.4.4.2.1 Object invoke(Object proxy, Method method, Object[] args) throws Throwable

Amikor egy **java.lang.reflect.Proxy** osztályon metódust hívnak, akkor az **InvocationHandlernek** az **invoke** metódusa hívódik meg.

A **proxy** paraméter, a proxy példány referenciája, amin a hívás történt.

A **method**, a meghívott metódust leíró **Java.lang.reflect.Method** osztály.

Az **args**, egy objektum tömb, amiben a metódusparaméterek konkrét értékek vannak, amivel a metódust meghívták.

A metódus visszaad egy objektumot, ami a kívánt visszatérési érték. Ha rossz a visszatérési érték típusa hiba lesz. Ha a futás során kivétel keletkezik, akkor a függvény azt tovább dobja.

3.4.4.3 Address

Az **Address** absztrakt osztály ősosztályául szolgál a különböző kommunikációs technológiák címzésének. Azért van rá szükség, hogy a proxy magával tudja hordozni a célcímet, így elosztottan átadhatóvá válik.

Implementálja a Serializeable jelölő interfacet, azért hogy szerializálható legyen és át lehessen adni a hálózaton keresztül.

3.4.4.3.1 protected String address

Az **address**, egy String attribútum, ami a cél címét tartalmazza.

3.4.4.3.2 Address(String address)

Konstruktor, a beállítja az **addresst.**

3.4.4.4 AProxy

Az **AProxy** osztály implementálja az **InvocationHandler** interfacet. A metódushívásokat elküldi a hálózatra, a választ visszaadja, ha hibát kap a hálózatról, akkor tovább dobja.

Implementálja a Serializable interfacet, ez által szerializálható.

3.4.4.4.1 transient ActiveSocketManager activesocketmanager

Az **activesocketmanager**, egy referencia egy **ActiveSocketManager** példányra, akitől egy olyan objektumot lehet kérni, ami hálózati elérést biztosít. Azért transient, mert ezt nem sorosítjuk egy hálózati átadás során, ugyanis egy másik gépen más **ActiveSocketManagerek** vannak.

3.4.4.4.2 transient NetworkInterface networkaccess

A **networkacces**, egy referencia egy olyan **NetworkInterface** példányra, ami a hálózati kommunikációt végzi. Ennek a példánynak hívjuk meg az Output metódusát.

3.4.4.4.3 Address address;

Az **address** attribútum, a szerializálható címet **Address** tárolja. Feladata, hogy hálózati átadás esetén, a benne tárolt információkkal újra felépíthető legyen a kapcsolat a céllal.

3.4.4.4.4 String id

Az **id** attribútum, azt az objektumot azonosítja, akinek a hívást továbbítani kell.

3.4.4.4.5 static Object newInstance(Class interfaceforproxy, ActiveSocketManager activesocketmanager, Address address, String id)

A **newInstance** metódus, létrehoz egy **AProxy** osztályt a megadott paraméterekkel és meghívja a **java.lang.reflect.Procy.newProxyInstance** metódust, aminek átadja a létrehozott **AProxy** osztályt **InvocationHandlernek**.

3.4.4.4.6 AProxy(ActiveSocketManager sf, Address address, String id)

Konstruktor, a megadott paramétereket beállítja, a többi null lesz.

3.4.4.4.7 AProxy(Address address, String id)

Konstruktor, a megadott paramétereket beállítja, a többi null lesz.

3.4.4.4.8 AProxy(Address address)

Konstruktor, a megadott paramétereket beállítja, a többi null lesz.

3.4.4.4.9 Object invoke(Object proxy, Method method, Object[] args)

Az invoke mettódus, az **InvocationHandler** interface metódus implementációja.

Ha kell, akkor beállítja az activesocketmanager attribútumot, tipikusan az **ActiveSocketManager**.**theasm**-re.

Ha nincs **networkacces** (null), akkor kér az **activesocketmanagertől**.

Ezek után a paraméterekből létrehoz egy **InvocationMessaget**, amit elküld a hálózatra a **networkaccess** Output metódusának meghívásával.

A válasz **ReplyMessage** objektumot megvizsgálja, ha volt kivétel, akkor eldobja, ha nem, akkor visszatér az eredménnyel.

3.4.4.5 Proxygen

A **Proxygen** osztály főleg kényelmi szempontokból létezik. Segítségével egyszerűbb **AProxy** objektumokat generálni, mert a beállított default értékekkel példányosítja az **AProxy**-kat, így nem kell újra meg újra megadni ugyanazokat a paramétereket.

3.4.4.5.1 ActiveSocketManager activesocketmanager

Az **activesocketmanager**, referencia egy **ActiveSocketManager** objektumra, ezzel példányosítja a generált **AProxykat**, ha mást nem adunk meg a példányosítás során.

3.4.4.5.2 Address address

Az **address**, egy referencia egy **Address** objektumra, ezzel példányosítja a generált **AProxykat**, ha mást nem adunk meg a példányosítás során.

3.4.4.5.3 Proxygen(ActiveSocketManager activesocketmanager, Address address)

Konstruktor, beállítja az attrubútumokat.

3.4.4.5.4 Object newInstance(Class interfaceforproxy, Address address, String id)

Meghívja az **AProxy** statikus **newInstance** metódusát a megadott paraméterekkel.

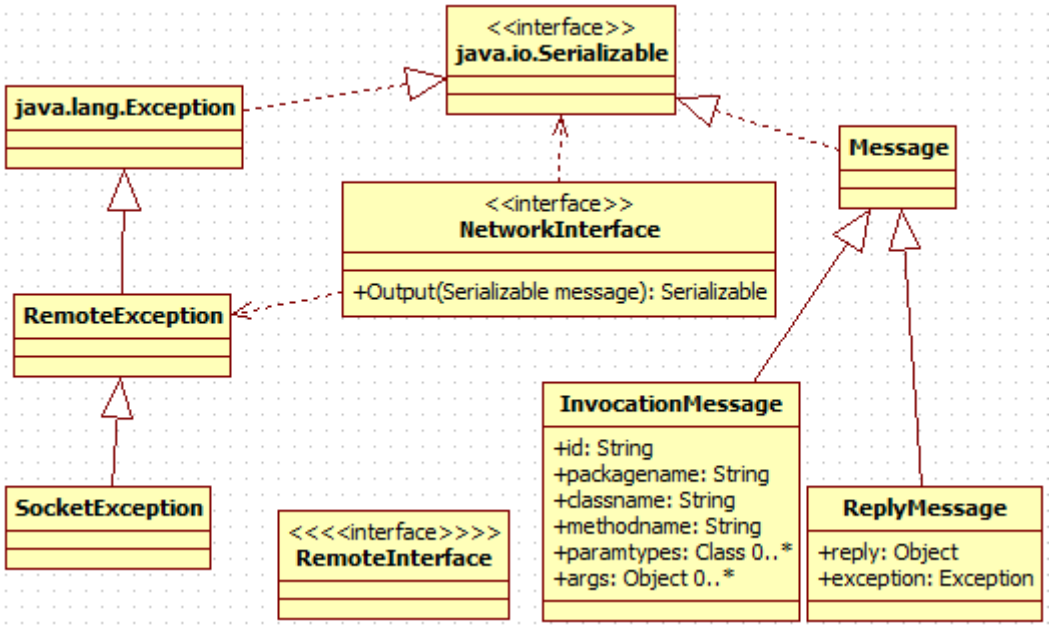
3.4.4.5.5 Object newInstance(Class interfaceforproxy, String id)

Meghívja az **AProxy** statikus newInstance metódusát a megadott paraméterekkel. A saját **address** címét adja át.

3.4.4.5.6 Object newInstance(Class interfaceforproxy)

Meghívja az **AProxy** statikus newInstance metódusát a megadott paraméterekkel. A saját **address** címét adja át, az **id** helyére null-t ad.

3.4.5 Remoteing package



16. ábra a Remoteing package

A 16. ábra a Remoteing package osztályait mutatja be részletesen, kiegészítve a releváns Java osztályokkal.

A **Remoteing** package osztályai a hálózati elérés keretét adják.

3.4.5.1 NetworkInterface

A **NetworkInterface** egy interfész, ami definiálja a hálózat határát.

Ezen az interfészen kell meghívni az egyik gépen az Output metódust, hogy a másik gépen egy ezt az interfészt implementáló osztály kapja meg a hívást.

3.4.5.1.1 Serializable Output(Serializable message) throws RemoteException, Exception

Az **Output** metódus az alacsony hálózati adatátvitel elfedő metódus, általa nem bájtokat kell átküldeni, hanem objektumokat.

Bemenő paramétere, és visszatérési értéke is egy szerializálható objektum

3.4.5.2 RemoteException

A **RemoteException** az **Excepiton** ből származik. Ősosztálya a távoli eljárás során keletkező hibáknak. Szerializálható, így a távoli gépen keletkező hibákat is át lehet küldeni.

3.4.5.3 RemoteInterface

A **RemoteInterface** egy jelölő interfész. Csak a **RemoteInterfacet** kiegészítő interfészek metódusai lesznek távolról elérhetőek, ugyanis a **Processor** csak azokat a metódusokat regisztrálja be, amik olyan interfészhez tartoznak, ami kiegészíti a **RemoteInterfacet**.

3.4.5.4 Message

A **Message** osztály, ősosztálya a hálózaton átküldendő csomagoló objektumoknak.

3.4.5.5 InvocationMessage

Az **InvocationMessage** osztály becsomagolja a távoli eljáráshívás során, a hálózaton átküldendő adatokat.

3.4.5.5.1 String id

Az **id** mező a távoli objektum azonosítója. Ha null, akkor nem egy konkrét objektumot azonosít, hanem egy osztályt, aminek meghívódik a paraméter nélküli konstruktora. Szükséges az objektumazonosításhoz.

3.4.5.5.2 String packagename

A **packagename** mező a távoli objektum packagenek a nevét tartalmazza. Szükséges az objektumazonosításhoz.

3.4.5.5.3 String classname

A **classname** mező a távoli objektum osztályának a nevét tartalmazza. Szükséges az objektumazonosításhoz.

3.4.5.5.4 String methodname

A **methodname** mező a távoli objektumon meghívandó metódus nevét tartalmazza. Szükséges a metódusazonosításhoz.

3.4.5.5.5 Class[] paramtypes

A **paramtypes** mező, a távoli objektumon meghívandó metódus paramétereinek az osztálytípusait tartalmazza egy tömbben. Szükséges a metódusazonosításhoz.

3.4.5.5.6 Object[] args

Az **args** mező, a távoli objektumon meghívandó metódus konkrét paramétereinek az értékeit tartalmazza egy tömbben. Szükséges a metódus futtatásához.

3.4.5.5.7 InvocationMessage(String id, String packagename, String classname, String methodname, Class[] paramtypes, Object[] args)

Konstruktor.

3.4.5.6 ReplyMessage

A **ReplyMessage**, a hálózaton átküldendő válasz objektumokat csomagolja be.

3.4.5.6.1 Object reply

A **reply** mező, a távoli eljárás visszatérési objektuma.

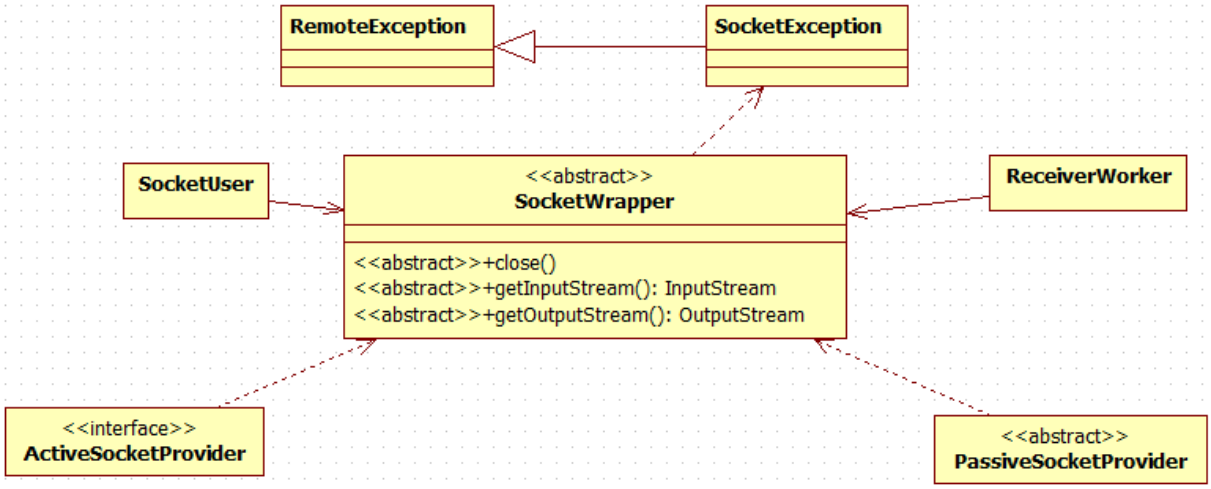
3.4.5.6.2 Exception exception

Az **exception** mező, a távoli eljárás során keletkező hiba objektum.

3.4.5.6.3 public ReplyMessage(Object reply, Exception exception)

Konstruktor.

3.4.6 Socketing package



17. ábra Socketing package

A 17. ábra a Socketing Packaget mutatja be a többi osztályhoz viszonyítva.

3.4.6.1 SocketWrapper

A **SocketWrapper** a különböző **streameket** szolgáltató kommunikációs objektumok absztrakt ősosztálya.

3.4.6.1.1 void close() throws IOException

A **close** metódus lezárja a kommunikációt.

3.4.6.1.2 abstract InputStream getInputStream() throws IOException

A **getInputStream** metódus visszaadja a kommunikációs objektum **Inputstreamjét**.

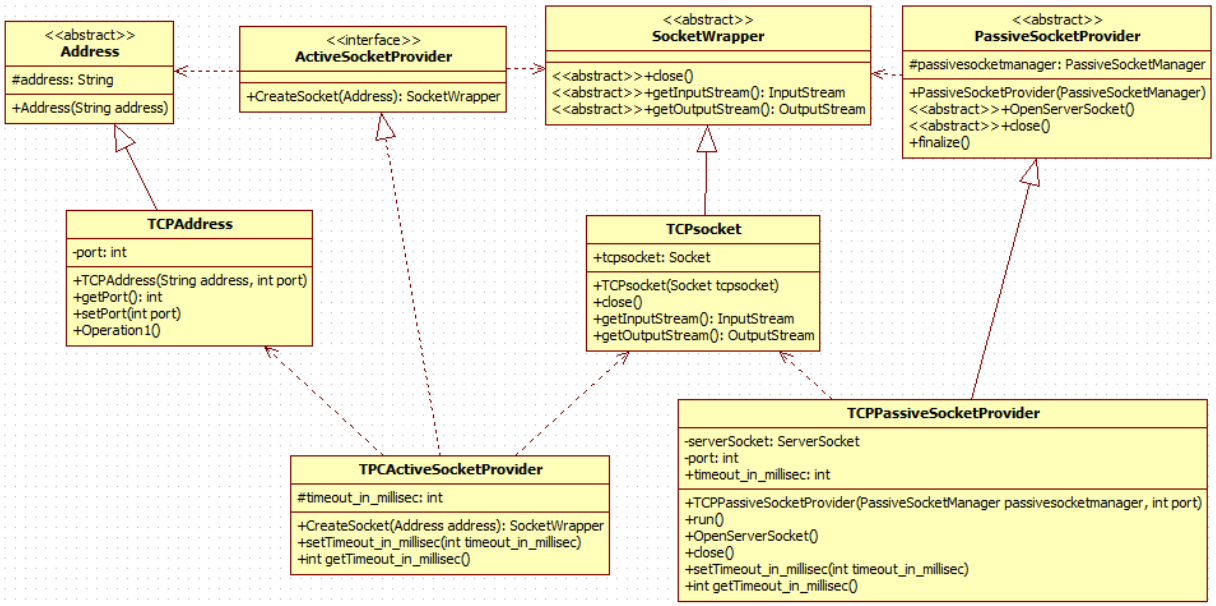
3.4.6.1.3 abstract OutputStream getOutputStream() throws IOException

A **getOutputStream** metódus visszaadja a kommunikációs objektum **Outputstreamjét**..

3.4.6.2 SocketException

A **SocketException** osztály a socket kezelés során azokat hibákat jelzi, amiket a standard java Exception osztályokkal nem lehet lefedni.

3.4.7 TCP Package



18. ábra TCP Package

A 18. ábrán a TCP package látható azokkal az osztályokkal együtt, amik a kapcsolati pontok a keretrendszerhez.

A TCP package, a legalacsonyabb szintű hálózati elérést biztosítja TCP-felet. A kapcsolatok felépítése, adatok továbbítása és kapcsolatok lebontása a feladata.

3.4.7.1 TCPActiveSocketProvider

A **TCPActiveSocketProvider** osztály feladata, hogy létrehozzon egy Java tcp socketet és becsomagolja egy **TCPsocket** be. Implementálja az **ActiveSocketProvider** interfészt.

3.4.7.1.1 int timeout_in_millisec

A **timeout_in_millisec**, mező a létrehozott **socket SO_TIEMOUT** paraméterét állítja be, ami azt mondja meg, hogy meddig blokkol maximum az olvasás a socektről.

3.4.7.1.2 public SocketWrapper CreateSocket(Address address) throws IOException, UnknownHostException

A **CreateSocket** metódus létrehoz egy socketet a megadott address-hez, amit **TCPAddress**sé kasztol. Ha szükséges, a létrehozott socketnek beállítja az SO_TIMEOUT paraméterét. Végül a létrejött **socketet** becsomagolja egy **TCPsocket** be és visszaadja.

3.4.7.2 TCPPassiveSocketProvider

A **TCPPassiveSocketProvider** várakozik a kapcsolatokra és létrehozza a **TCPsocketeket**. Leszármazik a **PassiveSocketProvider** osztályból.

3.4.7.2.1 ServerSocket serverSocket

A **serverSocket** mező, egy **ServerSocket**, ami a kapcsolatokat várja. Ezen a **ServerSocketen** figyeli a bejövő kapcsolatokat.

3.4.7.2.2 int port

A **port** mező határozza meg, hogy a **serverSocket** melyik TCP-porton figyelje a bejövő kapcsolatokat.

3.4.7.2.3 TCPPassiveSocketProvider(PassiveSocketManager passivesocketmanager, int port)

Konstruktor. Meghívja az ősosztály konstruktorát átadva a **passivesocketmanager** paramétert.

3.4.7.2.4 void run()

A **run** metódusban vár a **serverSocket** a bejövő kapcsolatokra. Új kapcsolat esetén becsomagolja egy **TCPsocket**-be majd továbbadja a passivesocketmanager-nek.

3.4.7.2.5 void OpenServerSocket() throws RemoteException, IOException

Az **OpenServerSocket** metódus létrehozza a **serverSocket** objektumot, a port attribútum által meghatározott portra.

3.4.7.2.6 void close() throws RemoteException, IOException

A **close** metódus lezárja a **serverSocket**-et így az több kapcsolatot nem tud fogadni.

3.4.7.3 TCPAddress

A **TCPAddress**, egy TCP/IP címet reprezentál. Az **Address** ből származik. Az address öröklött attribútumot használja az IP cím tárolására.

3.4.7.3.1 int port;

A port mező, a TCP/IP kapcsolat cél portját határozza meg.

3.4.7.3.2 TCPAddress(String address, int port)

Konstruktor. Meghívja az ősosztály konstruktorát **address** paraméterrel.

3.4.7.4 TCPsocket

A **TCPsocket**, egy becsomagolt TCP/IP socketet reprezentál. A **SocketWrapperből** származik.

3.4.7.4.1 Socket tcpsocket

A **tcpsocket**, a becsomagolt Socket referenciája.

3.4.7.4.2 int getTimeout_in_millisec() throws SocketException

A **getTimeout_in_millisec**, metódus visszaadja a **tcpsocket** SO_TIMEOUT paraméterét.

3.4.7.4.3 TCPsocket(Socket tcpsocket)

Konstruktor.

3.4.7.4.4 void close() throws IOException

A **close** metódus meghívja a **tcpsocket** close() metódusát, ez által lezárva azt.

3.4.7.4.5 InputStream getInputStream() throws IOException

A **getInputStream** metódus visszaadja a **tcpsocket InputStreamjét**.

3.4.7.4.6 OutputStream getOutputStream() throws IOException

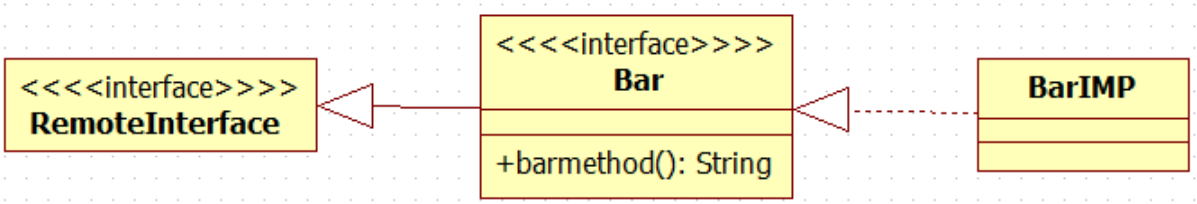
A **getOutputStream** metódus visszaadja a **tcpsocket OutputStreamjét**.

3.5 Dinamika

A következő fejezetben a keretrendszer dinamikus viselkedését mutatom be.

3.5.1 A környezet

A hálózat alacsony szintű bemutatására a TCP packaget használom. A keretrendszer használatának bemutatására létrehoztam egy teszt osztályt és interfacet, amit a későbbiekben egy példaalkalmazásban forráskód szinten is bemutatok a 3.7. fejezetben, és mérésre is felhasználok a 4. fejezetben.



19. ábra A példaprogram osztályai

A 19. ábrán, a példaprogram osztályai láthatóak.

3.5.1.1 Bar

A **Bar** interfész egy szokásos interfész, ami kiegészíti a **RemoteInterface** jelölő interfészt, ezáltal távolról elérhetővé válnak metódusai.

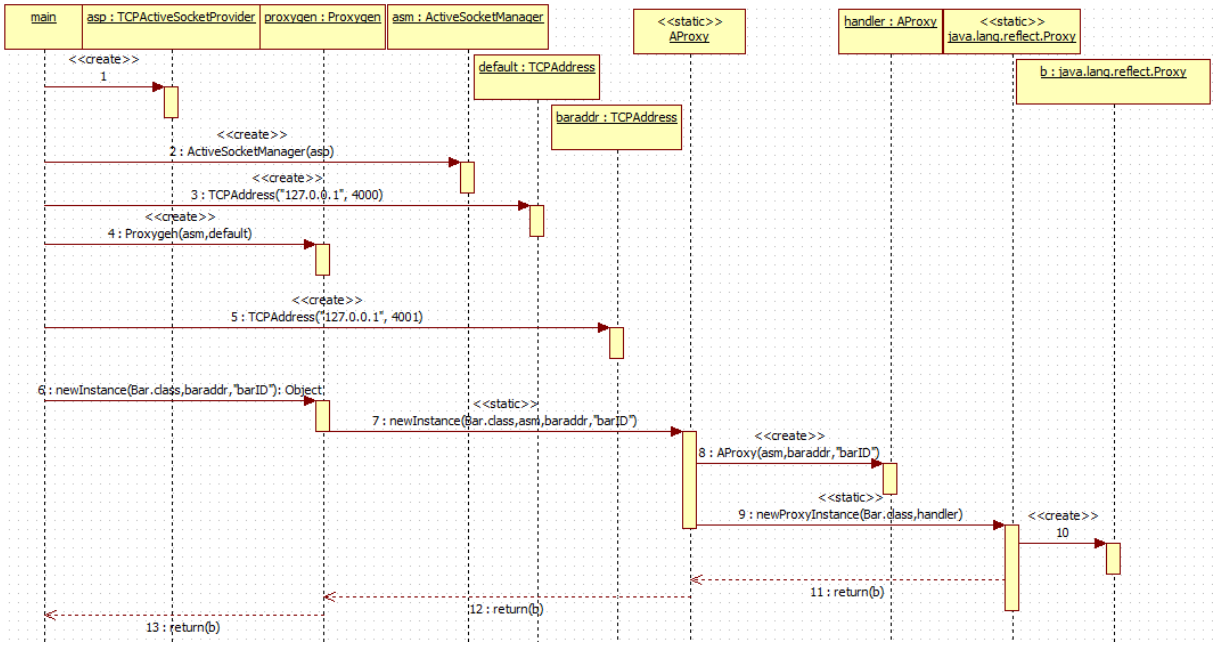
Egyetlen metódusa **barmethod**, egy Stringet ad vissza.

3.5.1.2 BarIMP

A **BarIMP** osztály implementálja a **Bar** interfészt.

A **barmethod** metódusban, vissza ad egy "Barimp" Stringet.

3.5.2 Kliens oldali inicializálás



20. ábra Kliens oldali inicializálás

A 20. ábrán a Kliens oldali inicializálást látjuk, amiben létrejönnek a keretrendszer objektumai, és a proxy osztály.

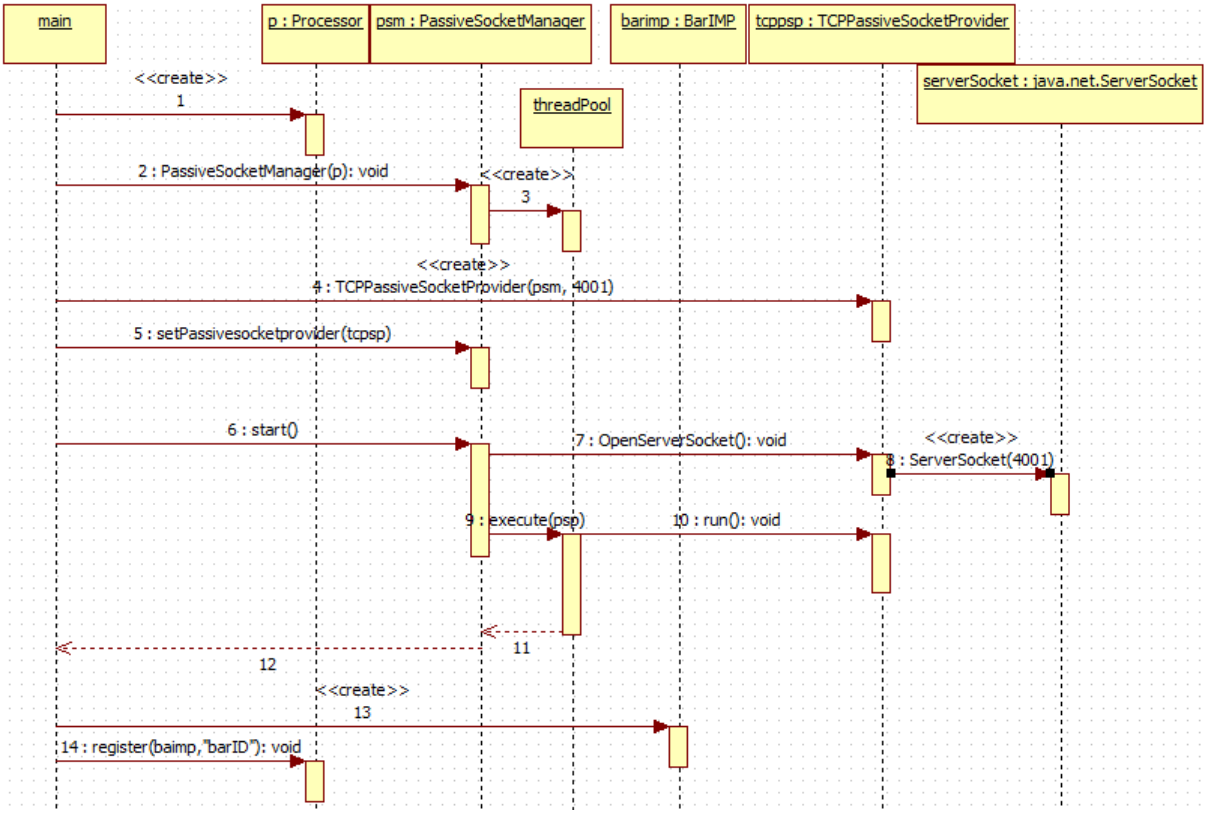
Először létrehozzuk a keretrendszer objektumait, és beállítjuk a referenciákat. Ez után létrehozunk egy címet (**baraddr**), ami a proxy címe lesz

A **Proxygen** osztálynak meghívjuk a **newInstace** metódusát paraméterként, hogy a **Bar**.class osztályból szeretnénk egy példányt, a (**baraddr**) címre, "**barID**" vel.

A **Proxygen** meghívja a Statikus **AProxy newInstance** metódust, ami létrehoz egy **AProxy** objektumot (**handler**), majd meghívja a statikus **java.lang.reflect.Proxy newInstance** metódust, paraméterként átadva a (**handler**)-t.

A **java.lang.reflect.Proxy newInstance** metódus visszatér egy objektummal, ami implementálja a **Bar**.class-t és a metódushívásait a (**handler**)-nek továbbítja, ami egy **AProxy**.

3.5.3 Szerver oldali inicializálás



21. ábra Szerver oldali inicializálás

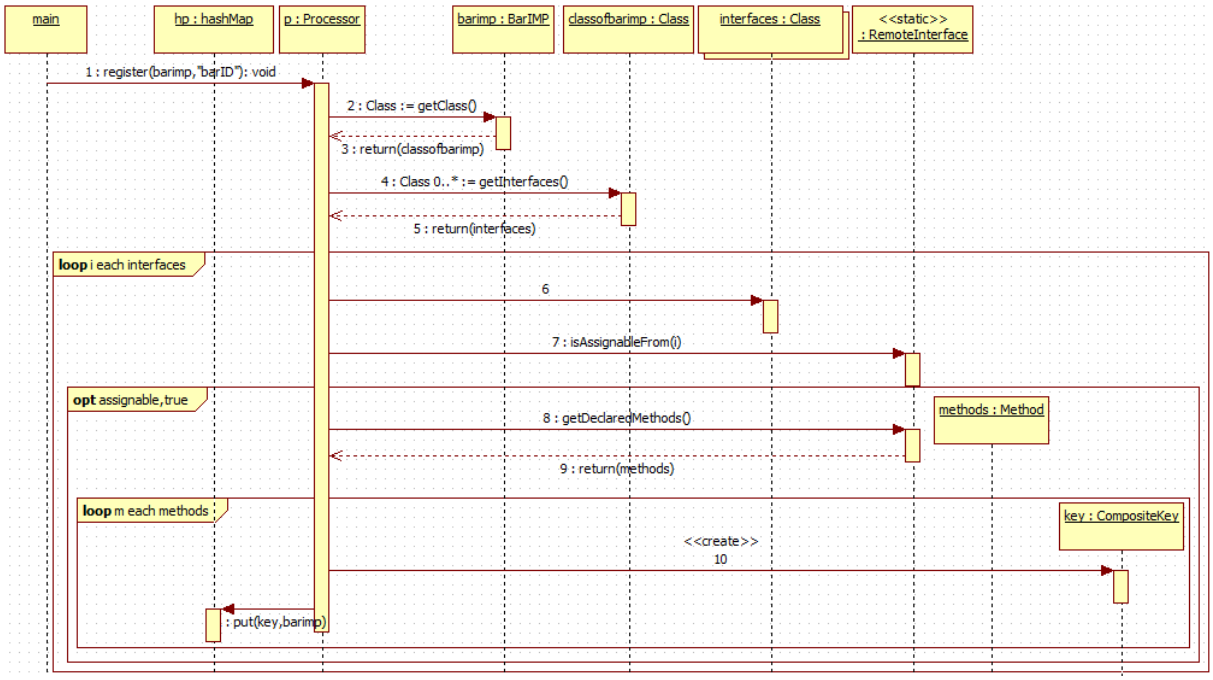
A 21. ábrán a szerver oldali inicializálás látható.

Először létrehozzuk a keretrendszer objektumait, és beállítjuk a referenciáit.

Ezek után a **PassiveSocketManager** (**psm**) **start** metódusát meghívjuk. Ez a **PassiveSocketProvider** (**tcpsp**)-en keresztül, létrehozza a szerver socketet a beállított porton. Majd a (**psm**) **threadpooljában** futtatjuk (**tcpsp**)-t, ami a **run**() metódusában passzívan várja a kapcsolatokat. A **run**() metódust a 3.5.7 fejezetben ismertetem.

Végezetül beregisztráljuk az implementációs osztályt,"**barID**" ID-val.

3.5.4 Register

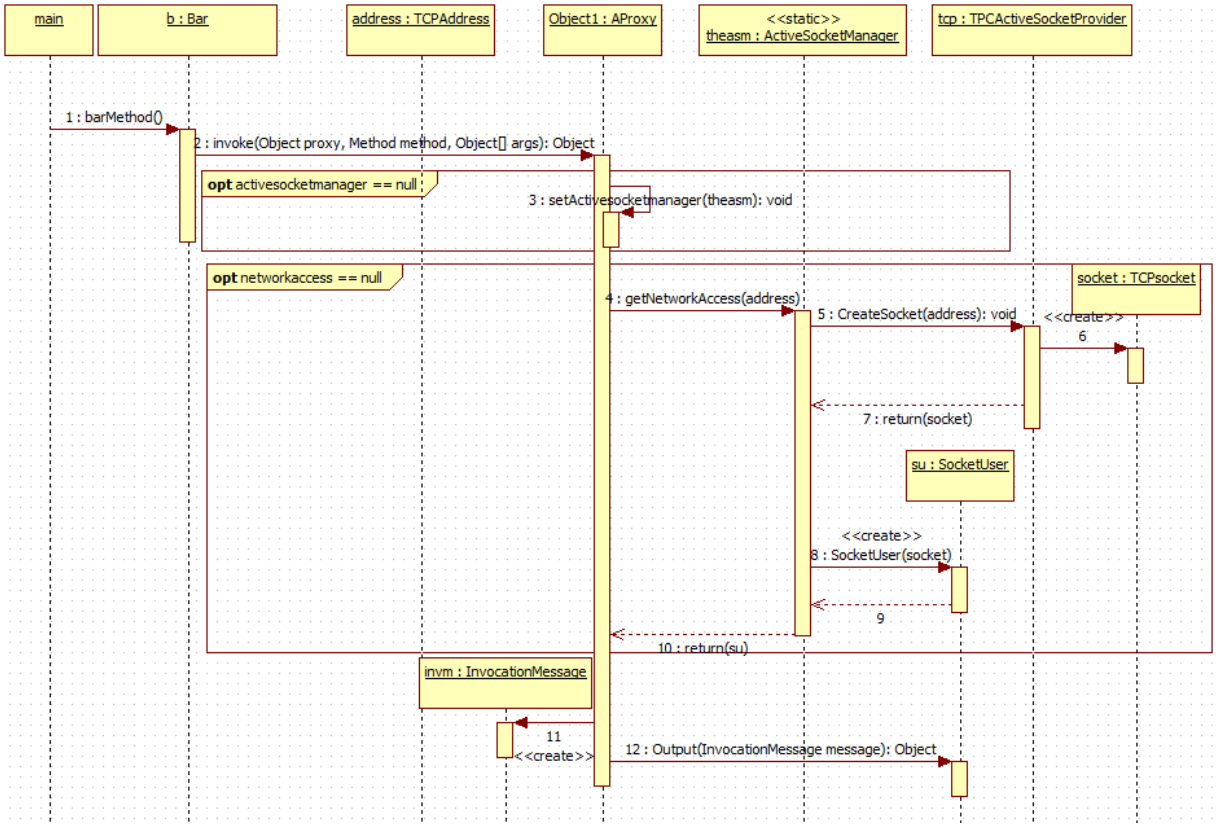


22. ábra Register metódus

A 22. ábrán a **Register** metódus futása látható.

Az átadott (**barimp**) objektumnak, lekéri az osztály leíróját, aminek lekéri az implementált interfészeit. Egy ciklusban végigmegy az interfaceken, és minden interfacre megnézi, hogy a jelölő **RemoteInterface**-t kiegészíti e. Ha kiegészíti, akkor végigmegy az interface metódusain, minden metódushoz létrehoz egy **CompositeKey** (**key**)-t és belerakja a **HashMapbe** (**hp**), a (**barimp**) objektummal együtt.

3.5.5 Kliens oldali metódushívás

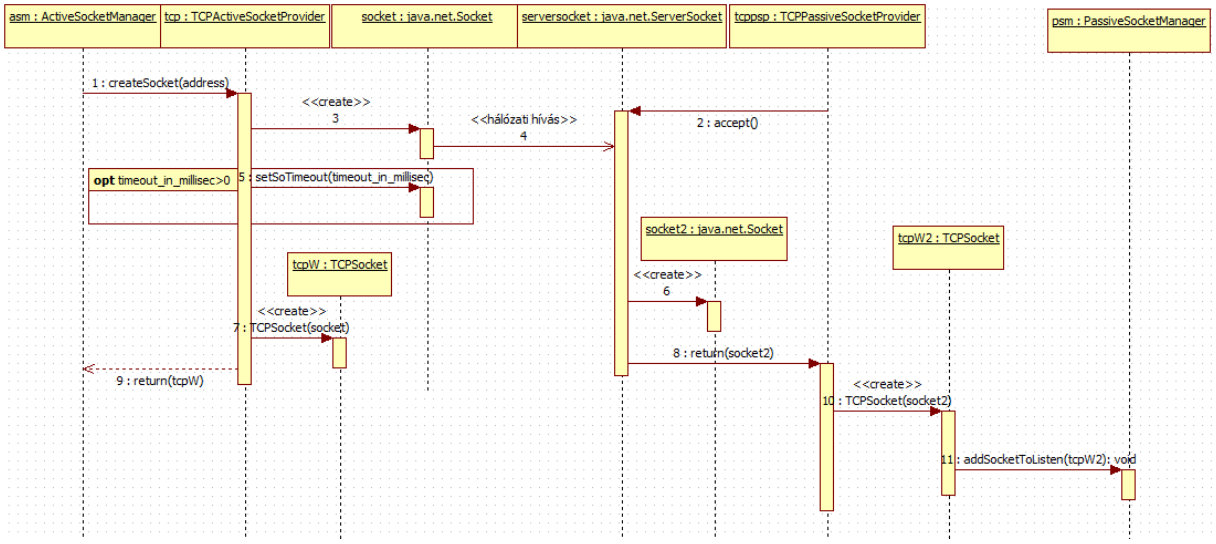


23. ábra Kliens oldali metódushívás felépítése

A 23. ábrán a kliens oldali metódushívás felépítése látható.

A **Bar** interfészt implementáló proxy osztályon (b), meghívjuk a **barMethod**-ot. Ezt a hívást továbbítja az **InvocationHandlernek**, ami egy **AProxy** (object1). Ha nincs **ActiveSocketManager**, akkor beállítja az alapértelmezettet (theasm). Ha nincs **networkacces**, akkor kér, az **activesocketmanagertől**. Jelen esetben a (theasm)-től. A (theasm), a **TCPActiveSocketProvider** (tcp)-től kér egy **SocketWrappert** (socket). A (tcp) létrehozza a socketet, ha kell, akkor beállítja a time-outot, majd létrehozza egy **TCPsocket** (socket) csomagolót és visszaadja. A (theasm) ezután példányosít egy **SocketUsert** (su), amit vissza ad az (object1)-nek. Az (objec1) ezután létrehozza az **InvocationMessaget** (invm), amit elküld a hálózaton.

3.5.6 A hálózati kapcsolat létrehozása



24. ábra a hálózati kapcsolat létrehozása

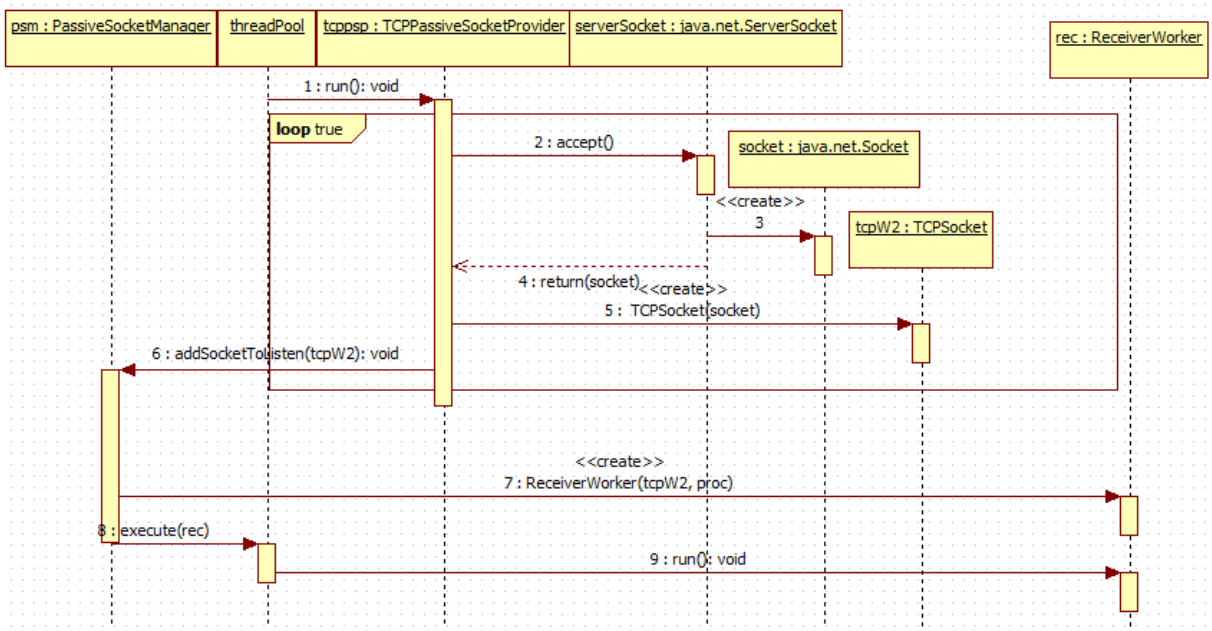
A 24. ábrán a hálózati kapcsolat létrehozásának lefutása látható.

A kliens oldalon az (asm) meghívja a (tcp) **createSocket** metódusát és átadja a cél címet (address), aminek statikus típusa **Address**, dinamikus típusa **TCPAddress**. Az (address) értékei alapján megpróbál létrehozni egy socketet (socket). **A 4-es számú hívás a létrejövő kapcsolatot jelöli**.

A szerver oldalon a (tcppsp) passzívan várakozik új kapcsolatokra a (serversocket) **accept** metódusban. Amikor létrejött egy új kapcsolat 4-es számú hívás, akkor a (serversocket) **accept** metódusa visszatér a kapcsolatot reprezentáló sockettel (socket2). Ezután a (tcppsp) létrehozza a csomagoló osztályt (tcpW2) és átadja a (psm)-nek, **addSocketToListen** metódushívással. A 3.5.7 fejezetben található a szerveroldal további kifejtése.

A kliens oldalon miután létrejött a (socket), ha kell a (tcp) beállítja a time-out értékét, majd létrehozza a csomagoló objektumot (tcpW) a (socket) köré és visszaadja.

3.5.7 Szerver oldali várakozás, run() és hatása



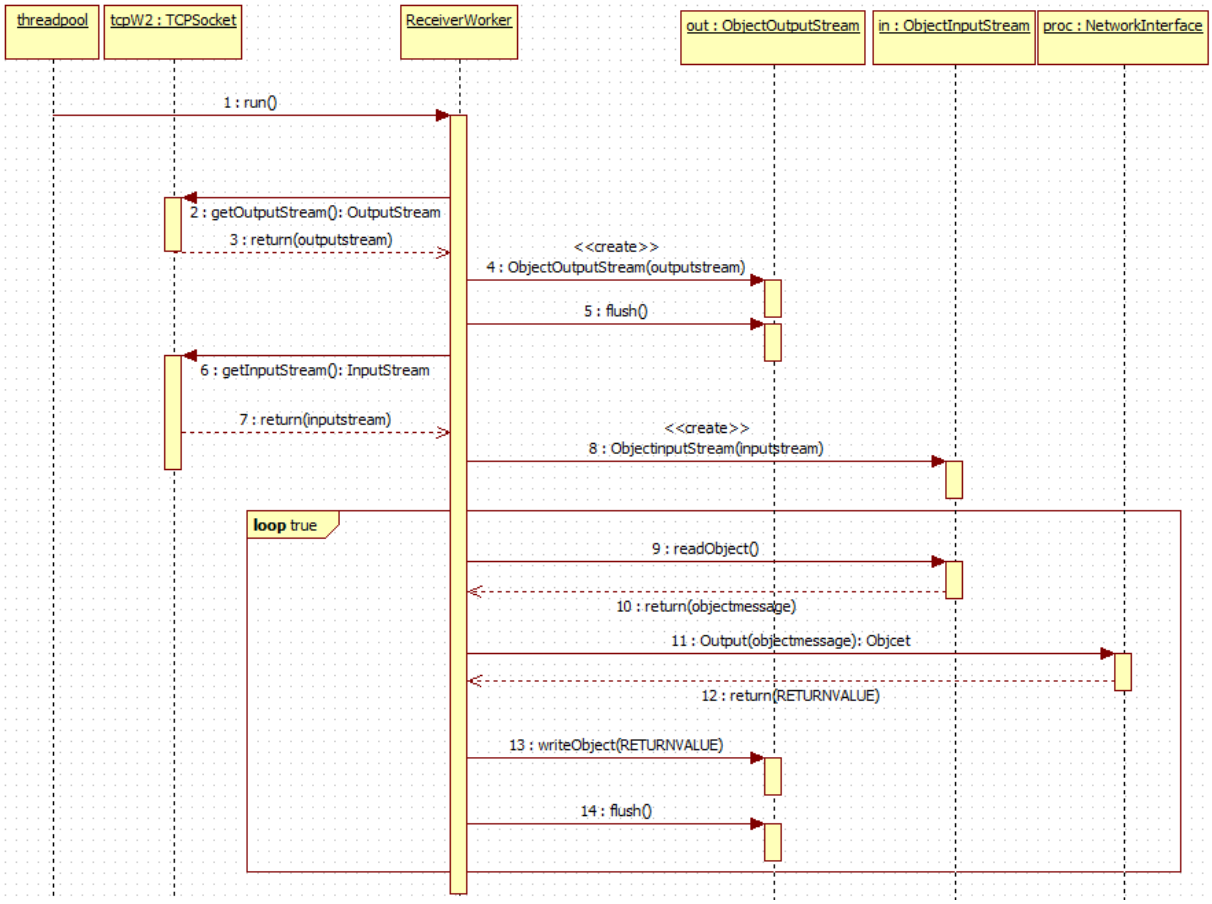
25. ábra Szerver oldali várakozás

A 25. ábrán a szerver oldali várakozó szál futását láthatjuk.

Az **run** metódus egy végtelen ciklust futtat, amiben a (serverSocket)-en hívott **accept** metódus addig blokkol, amíg fel nem épül egy kapcsolat. Amint ez megtörtént, létrejön egy **java.net.Socket** objektum (socket), amihez létrehoz egy csomagolót (tcpw2), amit átad a (psm)-nek **addSocketToListen** metódussal. A (psm) létrehoz egy **ReceiverWorkert** (rec), (tcpw2)-vel és a **proc** mező referenciájával, amit korábban egy **Processorra** állítottunk.

Végezetül a (rec)-et átadja a (threadpoolnak), ami elkezdi futtatni.

3.5.8 A Receiverworker run() metódusa



26. ábra A Receiverworker run() metódusa

A 26. ábrán a Receiverworker futását látjuk.

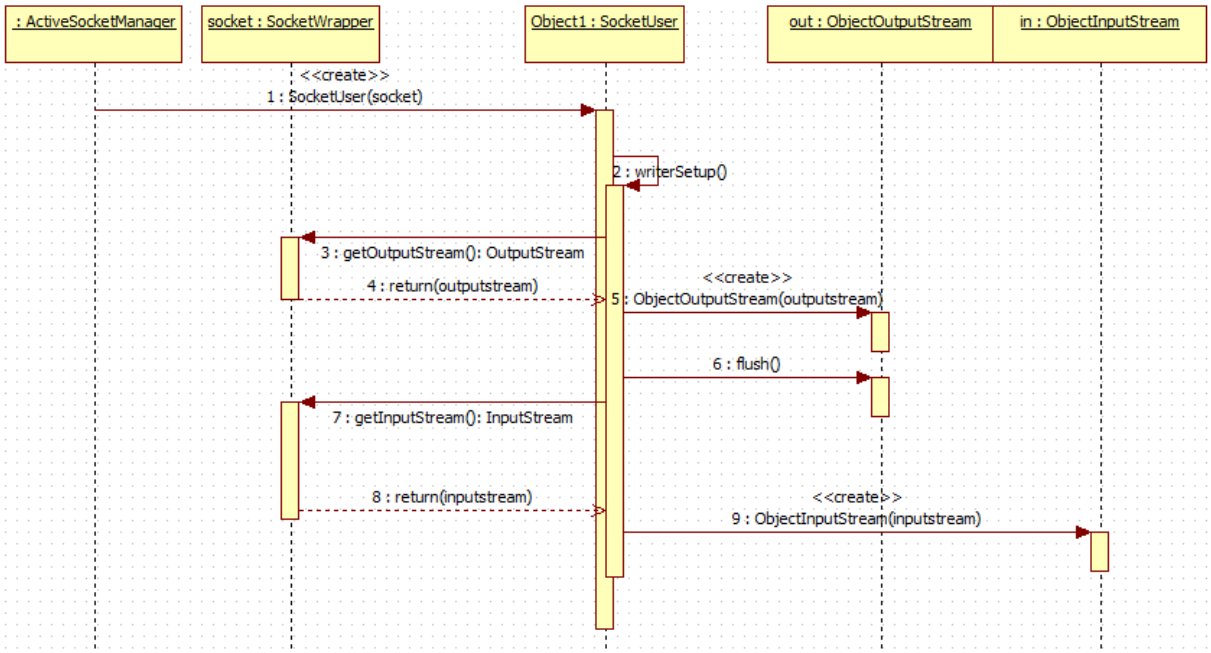
A **run** metódus elején, létrehozzuk az **ObjectOutputStreamet** (out), **flush**-t hívunk rá, mert egy inicializáló headert bele kell írni a streambe, amit a kliens oldali **ObjectInputstream** kiolvas.

Majd létrehozzuk az **ObjectInputStreamet** (in), ami blokkolva vár addig, amíg nem kapja meg a klienstől érkező inicializáló headert.

Miután beállítottuk az **ObjectStreameket**, egy végtelen ciklusba kerülünk.

Először blokkolva kiolvasunk egy objektumot (objectmessage), (in)-ből, majd továbbítjuk a (proc) felé az **Output** meghívásával. A válasz objektumot (RETURNVALUE) beleírjuk (out)-ba, majd **flush**-t hívunk rá és a ciklus újrakezdődik.

3.5.9 Egy SocketUser létrehozása



27. ábra Egy SocketUser létrehozása

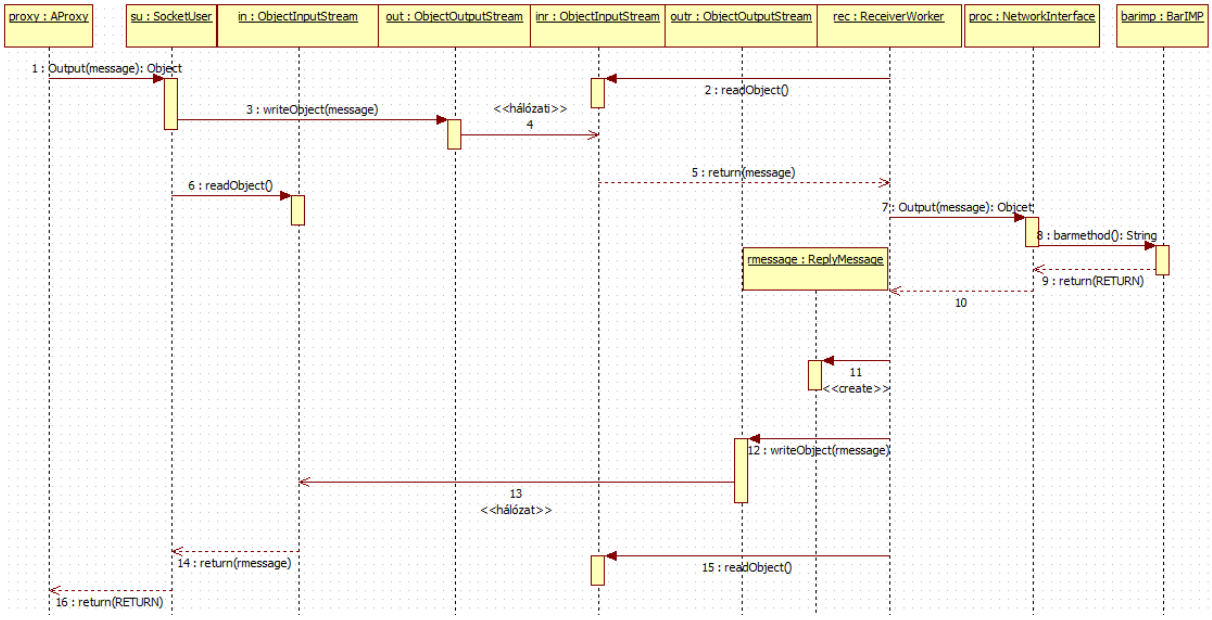
A 27. ábrán egy **SocketUser** konstruktorának lefutását látjuk, amiben beállítja a streameket.

A **SocketUser** (Object1), először lekéri a (socket) **OutputStreamjét**, létrehoz rá egy **ObjectOutputStreamet**(out), majd **flush**-t hív rá, ugyanis az **ObjectOutputStreamnek** bele kell írni egy headert, amit a szerver oldali **ObjectInputStream** olvas majd ki.

Végezetül lekéri a (socket) **InputStreamjét** és létrehoz rá egy **ObjetInputStreamet** (in), aminek a létrehozása addig blokkol, amíg nem olvassa ki a szervertől jövő headert.

Azért kell először az **ObjectOutputStreameket** létrehozni, hogy ne legyen holtpont.

3.5.10 Egy távoli eljárás hívás átfogó lefutása



28. ábra Átfogó lefutás

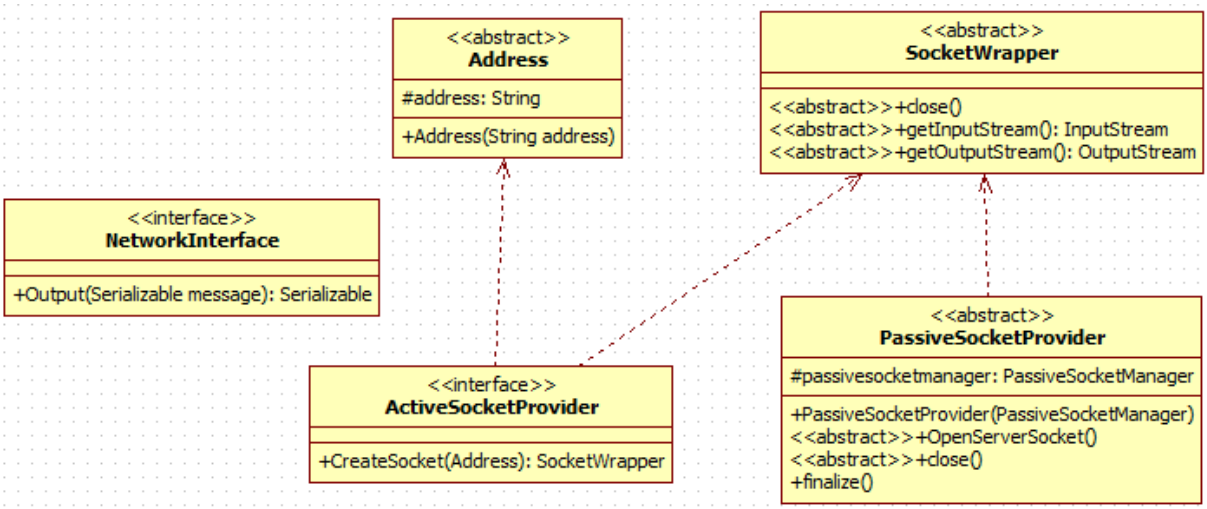
A 28. ábra egy távoli eljáráshívás átfogó lefutását mutatja be, amin látszik mind a kliens, mind a szerv oldal.

Először a (su) nek meghívjuk az **Output** metódusát, (message) objektum a paraméter. (su) beleírja (out) -ba (message)-t, majd (in) olvasásával **readObject** blokkolva vár.

A szerver oldalon a (rec) már korábban **readObject** –hívott (inr)-re, így várja az újabb objektumot. Amikor megkapja (message)-t, akkor meghívja (proc) **Output** metódusát, amiben paraméterként átadja (message)-t. (proc) ezután a megfelelő implementáció (barimp) megfelelő metódusát meghívja **barmethod**. A visszatérési értéket (RETURN) visszaadja. (rec) a visszatérési értékhez létrehoz egy **ReplyMessage**-et (rmessage), amit beleír (outr) -be, majd folytatja a 3.5.8-fejezetben ismertetettek szerint a futását.

A kliens oldal **readObject** hívása visszatér (rmessage) objektummal. (su) kicsomagolja (rmessage)-ból a visszatérési értéket (RETURN) és visszatér vele.

3.6 A keretrendszer határai



29. ábra a keretrendszer határai

A 29. ábrán a keretrendszer határait jelölő osztályok láthatóak.

3.6.1 A hívásátvitel határai

A hívásátvitel határait a **NetworkInterface** jelöli. A kliens oldalon egy **NetworkInterfacet** implementáló objektumon, tipikusan **SocketUser** objektumon, meghívott **Output** metódus, a szerver oldalon egy szintén **NetworkInterface**-t implementáló objektumon fog meghívódni, ami bármi lehet

3.6.2 A hálózati réteg határai

Hálózati szinten, a határokat az **Address**,**SocketWrapper**, **ActiveSocketProvider** és **PassiveSocketProvider** jelentik.

Új hálózati réteg használatához, a technológiára jellemző címzéshez meg kell valósítani egy **Address**-t.

Kliens oldalon a kapcsolat kezdeményezéséhez egy **ActiveSocketProvidert** kell megvalósítani, ami megkapja a megfelelő **Address** objektumot.

Szerver oldalon a kapcsolatok fogadásához egy **PassiveSocketProvidert** kell megvalósítani.

Végezetül a **SocketWrappert** is meg kell valósítani, ugyanis ez által lehet átadni a streameket a keretrendszer számára.

3.7 A példaalkalmazás, RMI-ben is

Ez a fejezet programkód szintjén mutatja be a keretrendszer használatát. Összehasonlítás alapjául RMI-ben is létrehoztam a példaalkalmazást. A létrehozott példaalkalmazás a 3.5.1 fejezetben megemlítettel azonos.

3.7.1 Interfacek deklarálása

3.7.1.1 ARPC

Először azokat az interfaceket kell létrehozni, amik a távoli eljárás során meghívandó metódusokat deklarálják.

```
import remoteing.RemoteException;
import remoteing.RemoteInterface;

public interface Bar extends RemoteInterface {
	String barmethod() throws RemoteException;
}
```

Annyi módosítás kell egy szokásos helyben használt interfacehez képest, hogy ki kell egészíteni a **RemoteInterface** jelölő interfacet, továbbá jelezni kell, hogy a metódus dobhat **RemoteException**-t.

3.7.1.2 RMI

RMI-ben az első lépés ugyanaz. Létrehozzuk az RMI-s **RemoteInterfacet** kiegészítő saját interfészeinket.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Bar extends Remote {
	String barmethod() throws RemoteException;
}
```

Ebben a lépésben minimális a különbség a két keretrendszer használata között.

3.7.2 Implementációk megírása

3.7.2.1 ARPC

A következő lépésben megírjuk az implementációs osztályokat.

```
import remoteing.RemoteException;
import test.interfaces.Bar;

public class BarIMP implements Bar {

	@Override
	public String barmethod() throws RemoteException {
		return "Barimp";
	}

}
```

Itt egy csak helyben hívható osztályhoz képest csak annyi az eltérés, hogy az implementált metódusnak **RemoteExceptiont** kell dobnia.

3.7.2.2 RMI

```
import java.rmi.RemoteException;
import remote.Bar;

public class BarIMP implements Bar {

	@Override
	public String barmethod() throws RemoteException {
		return "Barimp";
	}
}
```

Ebben a lépésben is minimális a különbség a két keretrendszer használata között.

3.7.3 Kliens oldal létrehozása

3.7.3.1 ARPC

Létre kell hoznunk a hálózati objektumokat és be kell állítanunk a paramétereket.

```
public class PeldaalkalmazasClient {

	public static void main(String[] args) {
		TCPActiveSocketProvider asp = new TCPActiveSocketProvider();
		ActiveSocketManager asm = new ActiveSocketManager(asp);
		TCPAddress _default = new TCPAddress("127.0.0.1", 4000);
		Proxygen proxygen = new Proxygen(asm, _default);
		TCPAddress baraddr = new TCPAddress("127.0.0.1", 4001);
		Bar b = (Bar) proxygen.newInstance(Bar.class, baraddr, "barID");
		try {
			b.barmethod();
		} catch (RemoteException e) {
			e.printStackTrace();
		}
	}
}
```

3.7.3.2 RMI

```
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

import remote.Bar;

public class PeldaRMICLIENT {
	public static void main(String[] args) {
		try {
			Registry registry = LocateRegistry.getRegistry(4001);
			Bar b = (Bar) registry.lookup("BAR");
			b.barmethod();
		} catch (RemoteException e) {
			e.printStackTrace();
		} catch (NotBoundException e) {
			e.printStackTrace();
		}
	}
}
```

Ebben a lépésben az általam írt keretrendszerrel több a teendő, ugyanis létre kell hozni az alacsony szintű hálózati objektumokat, míg az RMI elfedi azt.

Fontos azonosság, hogy miután megtörtént az inicializálás, az proxy, illetve stub példányon (b) **ugyanúgy történik a metódushívás** mindkét esetben, **b.barmethod**.

3.7.4 Szerveroldal létrehozása

3.7.4.1 ARPC

Létre kell hoznunk a hálózati objektumokat és be kell állítanunk a paramétereket.

```
public class PeldaalkalmazasServer {

	public static void main(String[] args) {
		Processor p = new Processor();
		PassiveSocketManager psm = new PassiveSocketManager(p);
		PassiveSocketProvider tcppp = new TCPPassiveSocketProvider(psm,
4001);
		psm.setPassivesocketprovider(tcppp);
		try {
			psm.Start();
		} catch (IOException e) {
			e.printStackTrace();
		} catch (RemoteException e) {
			e.printStackTrace();
		}
		BarIMP baimp = new BarIMP();
		p.register(baimp, "barID");
	}
}
```

3.7.4.2 RMI

```
import imps.BarIMP;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

import remote.Bar;

public class PeldaRMIServer {
	public static void main(String[] args) {
		try {
			String name = "BAR";
			Bar bar = new BarIMP();
			Bar stub = (Bar) UnicastRemoteObject.exportObject(bar, 0);
			Registry registry = LocateRegistry.createRegistry(4001);
			registry.rebind(name, stub);
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

Szerver oldal esetében is többet kell dolgozni az ARPC keretrendszer beüzemelésével.

3.7.5 Konklúzió

Az általam írt keretrendszer az inicializálásnál egy kicsit több odafigyelést igényel, de ettől eltekintve ugyanolyan kényelmesen használható, mint az RMI.

3.8 Bluetooth

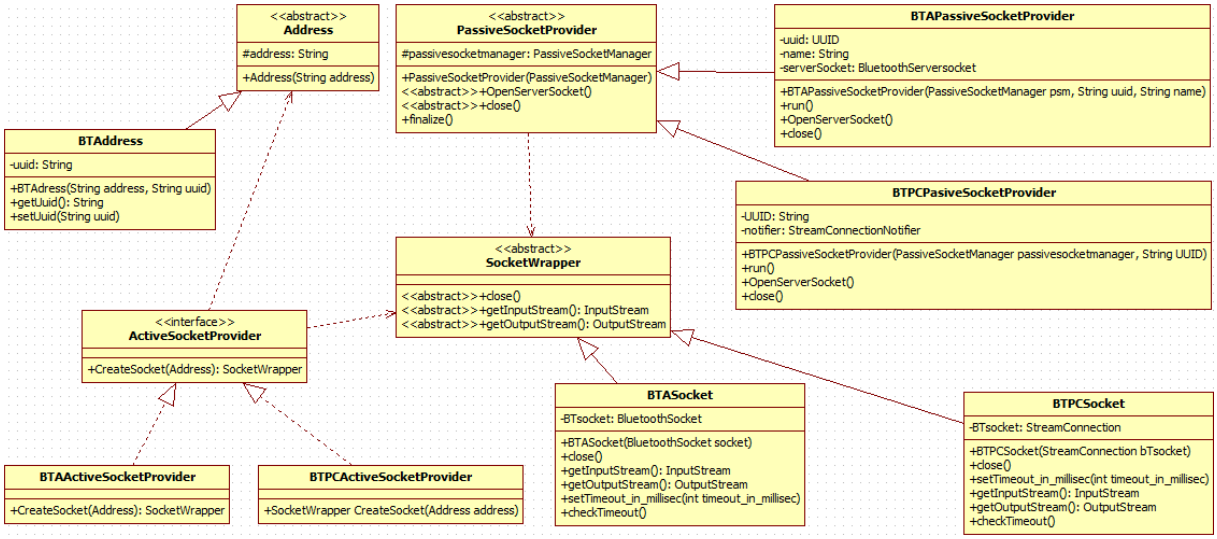
A Bluetooth egy rövidtávú vezeték nélküli adatcserére létrehozott szabvány. 1994-ben az Ericsson hozta létre.

A JSR-82 egy a Java Bluetooth API specifikációja.

A BlueCove egy osztály könyvtár, ami implementálja a JSR-82 specifikációt.

3.9 Bluetooth-Android-BlueCove

A keretrendszerem Bluetooth alapú kiterjesztését mutatom be ebben a fejezetben.



30. ábra Bluetooth implementációk

A 30. ábra, a **Bluetooth** implementációs osztályokat tartalmazza Android és Standard Java környezethez, utobbihoz a BlueCove osztálykönyvtárat használtam.

3.9.1 Általános Bluetooth kiegészítések

Az Android és a BlueCove különböző implementáció miatt a csak a **BTAddress**, Bluetooth címzést megvalósító osztály közös.

3.9.1.1 BTAddress

A **BTAddress** osztály egy Bluetooth végpontot azonosít, az **Address** osztályból származik. Az öröklött address mezőben tárolja a Bluetooth harware címét- Media Access Control address-.

3.9.1.1.1 String uuid

Az **uuid** mező, az Universally Unique IDentifier-t tartalmazó mező. [8]

3.9.1.1.2 BTAddress(String address, String uuid)

Konstruktor. Meghívja az ős **Address** konstuktorát.

3.9.2 Android Specifikus kiegészítések

3.9.2.1 BTASocket

A **BTASocket** osztály az **Android BluetoothSocket** osztályát csomagolja be.

3.9.2.1.1 BluetoothSocket BTsocket

A **BTsocket** mező, a becsomagolt **BluetoothSocket**re mutató referencia.

3.9.2.1.2 BTASocket(BluetoothSocket bTsocket)

Konstruktor.

3.9.2.1.3 close() throws IOException

A **close** metódus meghívja a **BTsocketet close** metódusát, ez által lezárja azt.

3.9.2.1.4 InputStream getInputStream() throws IOException

A **getInputStream**() metódus meghívja a **BTsocket getInputStream**jét, és ez által visszaadja az **InputStreamet**.

3.9.2.1.5 OutputStream getOutputStream() throws IOException

A **getOutputStream** metódus meghívja a **BTsocket getOutputStream**jét, és ez által visszaadja az **OutputStreamet**.

3.9.2.2 BTAActiveSocketProvider

A **BTAActiveSocketProvider** az Android kérésre **BTASocket**eket előállító osztálya.

Fontos, hogy Androidon csak párosított -paired- eszközökhöz csatlakozhatunk. A Bluetooth socketek létrehozásához ezen felül mind a Bluetooth hardware címére és az UUID-ra is szükség van.

3.9.2.2.1 SocketWrapper CreateSocket(Address address) throws IOException

A CreateSocket metódus létrehoz egy BluetoothSocketet, a megadott cím alapján, majd létrehoz egy **BTASocket** csomagoló osztályt, amibe belerakja a létrehozott BluetoothSocketet és visszatér vele.

3.9.2.3 BTAPassiveSocketProvider

A **BTAPassiveSocketProvider** az Android platform várakozva **SocketWrappereket** - **BTASocketeket**- előállító osztálya.

Fontos, hogy Androidon csak párosított -paired- eszközöktől fogadhatunk el csatlakozást.

3.9.2.3.1 UUID uuid

Az **uuid**, egy Universally Unique IDentifier, ami a szolgáltatás egyedi azonosítója.

3.9.2.3.2 String name

A **name**, a **ServiceRecord** nevét tárolja, amin a **BTAPassiveSocketProvider** figyeli a bejövő kapcsolatokat. [9]

3.9.2.3.3 BluetoothServerSocket serverSocket

A **serverSocket** mező referencia egy **BluetoothServerSocketre**, ami kapcsolatokra vár. Ezen a -BluetoothServerSocketen figyeli a bejövő Bluetooth kapcsolatokat

3.9.2.3.4 BTAPassiveSocketProvider(PassiveSocketManager passivesocketmanager, String uuid, String name)

Konstruktor. Meghívja az ősosztály konstruktorát átadva a **passivesocketmanager** paramétert.

3.9.2.3.5 void run()

A **run** metódusban vár a **serverSocket** a bejövő kapcsolatokra. Új kapcsolat esetén becsomagolja egy **BTASocket**-be majd továbbadja a **passivesocketmanager**-nek.

3.9.2.3.6 void OpenServerSocket() throws RemoteException, IOException

Az **OpenServerSocket** metódus létrehozza a **serverSocket** objektumot, a **name** attribútum által meghatározott ServiceRecordra és az **uuid** által megadott uuidval.

3.9.2.3.7 void close() throws RemoteException, IOException

A **close** metódus lezárja a **serverSocket**et így az több kapcsolatot nem tud fogadni.

3.9.3 BlueCove kiegészítések

3.9.3.1 BTPCSocket

A **BTPCSocket** osztály az **javax.microedition.io.StreamConnection** osztályát csomagolja be be.

3.9.3.1.1 StreamConnection BTsocket

A **BTsocket** mező referencia, egy becsomagolt **StreamConnection**re.

3.9.3.1.2 BTPCSocket(StreamConnection bTsocket)

Konstruktor.

3.9.3.1.3 close() throws IOException

A **close** metódus meghívja a **BTsocketet close** metódusát, ez által lezárja azt.

3.9.3.1.4 InputStream getInputStream() throws IOException

A **getInputStream** metódus meghívja a BTsocket **openInputStream** metódusát, ez által visszaadja az **InputStreamet**.

3.9.3.1.5 OutputStream getOutputStream() throws IOException

A **getOutputStream** metódus meghívja a BTsocket **openOutputStream** metódusát, ez által visszaadja az **OutputStreamet**.

3.9.3.2 BTPCActiveSocketProvider

A **BTPCActiveSocketProvider** a javax.microedition kérésre **BTPCSocket**-eket előállító osztálya.

3.9.3.2.1 SocketWrapper CreateSocket(Address address) throws IOException

A **CreateSocket** metódus létrehoz egy **StreamConnectiont**, a megadott cím alapján, majd létrehoz egy **BTPCSocket** csomagoló osztályt, amibe belerakja a létrehozott **StreamConnection**-t és visszatér vele.

Fontos kiegészítés, hogy a javaME, nem Bluetooth hardware cím alapján keres, hanem UUID alapján és nincs megkötés arra, hogy milyen eszközhöz csatlakozhatunk.

3.9.3.3 BTPCPassiveSocketProvider

A **BTPCPassiveSocketProvider** a javax.microedition várakozva BTASocketeket előállító osztálya.

3.9.3.3.1 UUID uuid

Az **uuid** mező, egy Universally Unique IDentifier, ami a szolgáltatás egyedi azonosítója.

3.9.3.3.2 String name

A **name**, a ServiceRecord nevét tárolja, amin az osztály figyeli a bejövő kapcsolatokat. [9]

3.9.3.3.3 StreamConnectionNotifier notifier

A **noitifer** mező, egy StreamConnectionNotifier, ami kapcsolatokra várakozik. Ezen a StreamConnectionNotifier figyeli a bejövő Bluetooth kapcsolatokat

3.9.3.3.4 BTAPassiveSocketProvider(PassiveSocketManager passivesocketmanager, String uuid, String name)

Konstruktor. Meghívja az ősosztály konstruktorát átadva a **passivesocketmanager** paramétert.

3.9.3.3.5 void run()

A **run** metódusban vár a notifier a bejövő kapcsolatokra. Új kapcsolat esetén becsomagolja egy BTPCSocketbe majd továbbadja a passivesocketmanagernek.

3.9.3.3.6 void OpenServerSocket() throws RemoteException, IOException

Az **OpenServerSocket** metódus létrehozza a notifier objektumot, a name attribútum által meghatározott ServiceRecordra és az uuid által megadott uuid-val.

3.9.3.3.7 void close() throws RemoteException, IOException

A **close** metódus lezárja a notifiert így az több kapcsolatot nem tud fogadni.

4 Mérés

A következő fejezetben méréseket folytatok az elkészült keretrendszeren és egy referenciamérést is elvégzek, amivel összehasonlítom az RMI-vel.

4.1 Az eszközök bemutatása

4.1.1 Samsung Galaxy S Android

A készülék megfelel a gyári specifikációknak [10] az alábbi kivételekkel.

Firmware verzió	2.3.4
Kernel-verzió	2.6.35.7-I9000XXJVR-CL425308se.infra@SEP-63 #2
Build szám	GINGERBEARD.XXJVR

4.1.2 ZTE Blade Android

A készülék megfelel a gyári specifikációknak. [11]

4.1.3 MSI GX620 Notebook Java SE

Operációs rendszer verziója	6.1.7601 (Win7 RTM)
Processor	Mobile DualCore Intel Core 2 Duo P8400, 2266
Alaplap neve	MSI MegaBook GT627/GT628/GX620/GX623/GX627/PX603 (MS-1651)
Alaplap Bios	A1651IMS.10E
Hálózati kártya	Realtek RTL8168B/8111B Family PCI-E Gigabit Ethernet NIC (NDIS 6.20)
Bluetooth HW azonosító	BTH\MS_RFCOMM
Java Verzió	1.7

4.1.4 ASUS RT-N16 Router

Az eszköz megfelel a gyári specifikációnak [12] az alábbi kivétellel.

Firmware:	DD-WRT v24-sp2 (05/08/11) mega

4.2 A tesztprogram

4.2.1 Inicializálás

Az inicializálás során létrehoztam a keretrendszer osztályait, beállítottam a megfelelő IP, Bluetooh hardware címeket, elindítottam a szerver oldalon a megfelelő PassiveSocketProvidert és legeneráltam egy Proxy osztályt a Bar interface-hez.

4.2.2 A mérés

```
double multiplier = 1;
	try {
		b.barmethod();//meleg inditas miatt
		long starttime = System.currentTimeMillis();
		for (int i = 0; i < multiplier; i++) {
			b.barmethod();
		}
```

```
            long totaltime = (System.currentTimeMillis() - starttime);
            System.out.println("avg: " + totaltime / multiplier);
        } catch (RemoteException e)
```

A tesztprogram ebben a részében végeztem el a mérést.

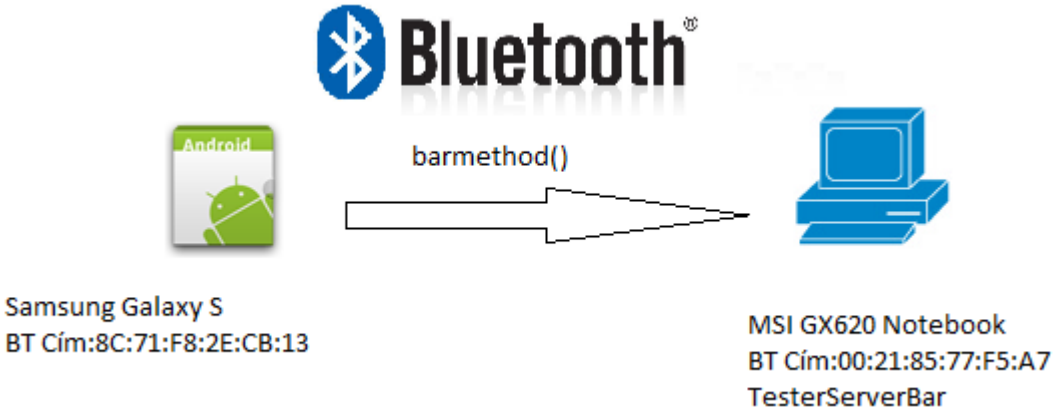
A multiplier változóban adtam meg, hogy hányszor ismételje meg a b.barmethod() hívását.

Mivel a keretrendszer, az első hívásnál építi fel a kapcsolatot, ezért különvettem a hideg indítás lemérését, amikor a kapcsolat felépítése is szükséges.

A további mérések során, a mérőciklus elindítása előtt kiépítettem a kapcsolatot egy b.barmethod() hívással.

4.3 Mérések

4.3.1 Bluetooth Android-ról PC-re



31. ábra Bluetooth Androidról-PC-re

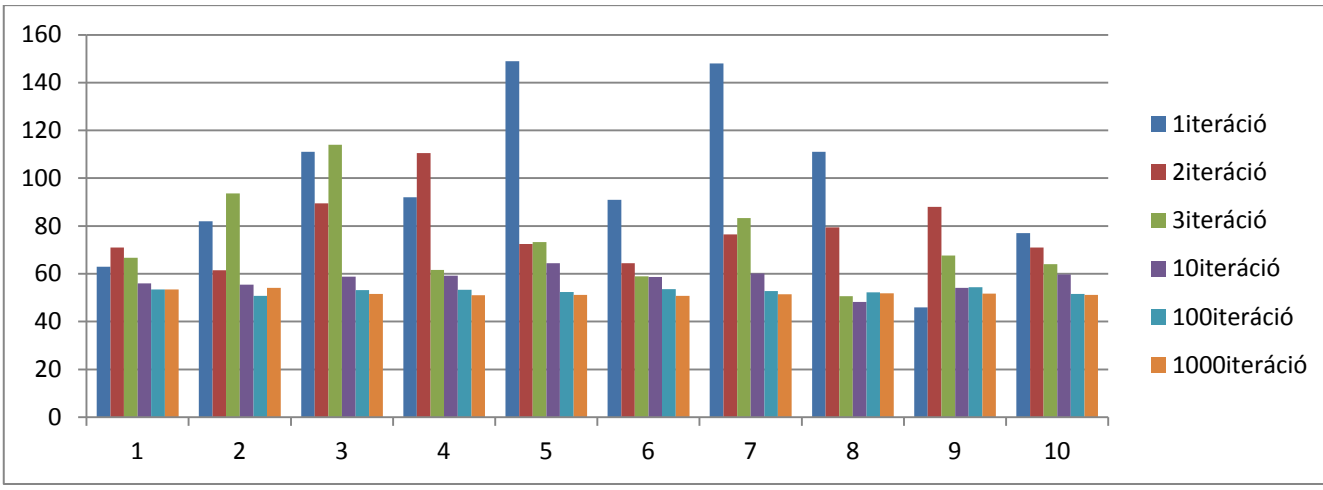
A 31. ábra azt a mérési elrendezést szemlélteti, amikor, Android a kliens, PC a szerver és a hálózati réteg Bluetooth.

4.3.1.1 Adatok

	1.mérés	2.mérés	3.mérés	4.mérés	5.mérés	6.mérés	7.mérés	8.mérés	9.mérés	10.mérés
inic	1 270,000ms	1 218,000ms	1 262,000ms	1 265,000ms	1 186,000ms	1 212,000ms	1 236,000ms	1 150,000ms	1 353,000ms	1 269,000ms
1	63,000ms	82,000ms	111,000ms	92,000ms	149,000ms	91,000ms	148,000ms	111,000ms	46,000ms	77,000ms
2	71,000ms	61,500ms	89,500ms	110,500ms	72,500ms	64,500ms	76,500ms	79,500ms	88,000ms	71,000ms
3	66,667ms	93,667ms	114,000ms	61,667ms	73,330ms	59,000ms	83,300ms	50,660ms	67,660ms	64,000ms
10	56,000ms	55,400ms	58,800ms	59,200ms	64,500ms	58,700ms	60,100ms	48,200ms	54,100ms	59,700ms
100	53,440ms	50,790ms	53,130ms	53,360ms	52,440ms	53,600ms	52,760ms	52,300ms	54,390ms	51,560ms
1000	53,472ms	54,124ms	51,513ms	51,045ms	51,209ms	50,810ms	51,448ms	51,860ms	51,762ms	51,234ms

1. táblázat Bluetooth Androidról-PC-re mérés

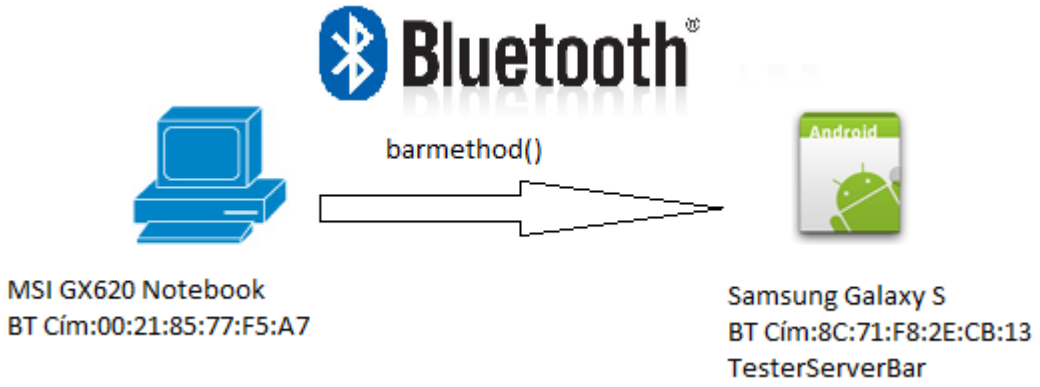
Az 1. táblázat a Bluetooth Androidról-PC-re mérés adatait tartalmazza.



32. ábra Bluetooth Androidról-PC-re mérés diagramja.

A 32. ábrán a Bluetooth feletti Androidról-Pc re irányuló mérés eredményei láthatók.

4.3.2 Bluetooth PC-ről Android-ra



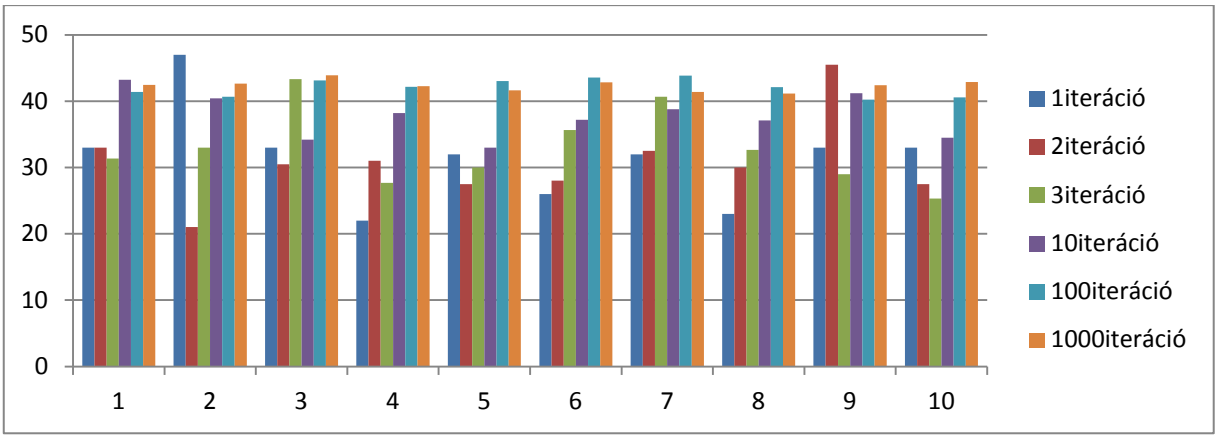
33. ábra Bluetooth PC-ről Android-ra

A 33. ábra azt a mérési elrendezést szemléltetni, amikor, PC a kliens, Android a szerver és a hálózati réteg Bluetooth.

4.3.2.1.1 Adatok

	1.mérés	2.mérés	3.mérés	4.mérés	5.mérés	6.mérés	7.mérés	8.mérés	9.mérés	10.mérés
inic	414,000ms	1 747,000ms	602,000ms	1 550,000ms	493,000ms	1 181,000ms	1 883,000ms	3 015,000ms	2 144,000ms	1 550,000ms
1	33,000ms	47,000ms	33,000ms	22,000ms	32,000ms	26,000ms	32,000ms	23,000ms	33,000ms	33,000ms
2	33,000ms	21,000ms	30,500ms	31,000ms	27,500ms	28,000ms	32,500ms	30,000ms	45,500ms	27,500ms
3	31,330ms	33,000ms	43,330ms	27,660ms	30,000ms	35,660ms	40,660ms	32,660ms	29,000ms	25,330ms
10	43,200ms	40,400ms	34,200ms	38,200ms	33,000ms	37,200ms	38,800ms	37,100ms	41,200ms	34,500ms
100	41,380ms	40,660ms	43,130ms	42,160ms	43,010ms	43,560ms	43,830ms	42,110ms	40,230ms	40,560ms
1000	42,458ms	42,640ms	43,883ms	42,276ms	41,637ms	42,855ms	41,372ms	41,125ms	42,398ms	42,882ms

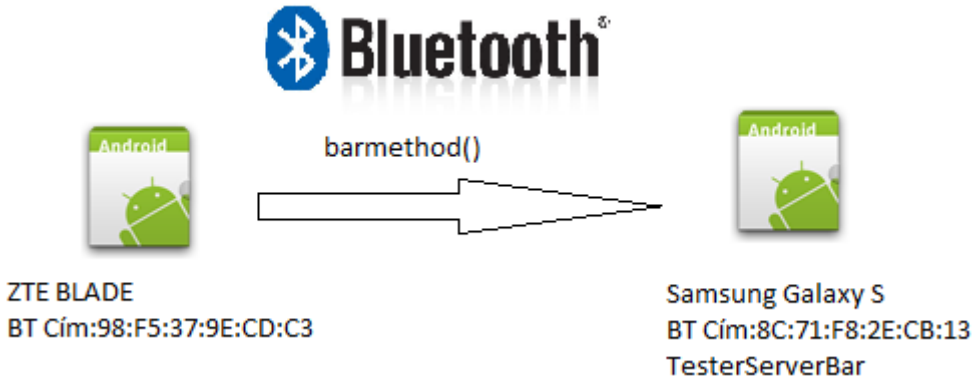
2. táblázat Bluetooth PC-ről Android-ra mérés adatai



34. ábra a Bluetooth PC-ről Android mérés diagramja

A 34. ábrán a Bluetooth feletti PC-ről Androidra irányuló mérés eredményei láthatók.

4.3.3 Bluetooth Android-ról Android-ra



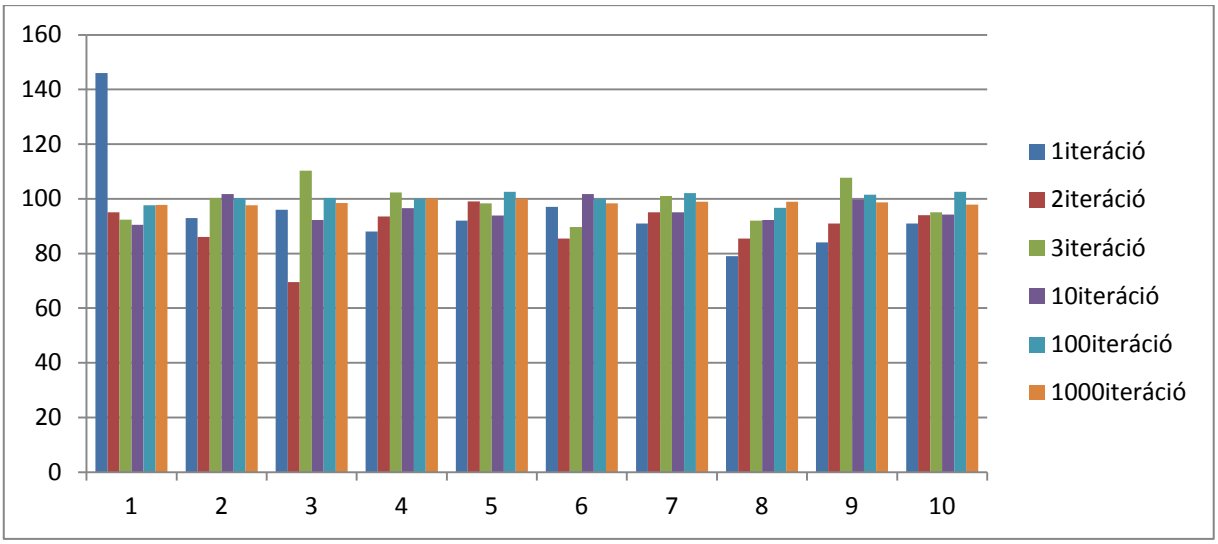
35. ábra Bluetooth Androidról-Androidra

A 35. ábra azt a mérési elrendezést szemléltetni, amikor, Androidról a kliens, Android a szerver és a hálózati réteg Bluetooth.

4.3.3.1.1 Adatok

	1.mérés	2.mérés	3.mérés	4.mérés	5.mérés	6.mérés	7.mérés	8.mérés	9.mérés	10.mérés
inic	1 761,000ms	2 540,000ms	2 512,000ms	2 319,000ms	2 439,000ms	1 590,000ms	1 465,000ms	1 392,000ms	2 236,000ms	1 655,000ms
1	146,000ms	93,000ms	96,000ms	88,000ms	92,000ms	97,000ms	91,000ms	79,000ms	84,000ms	91,000ms
2	95,000ms	86,000ms	69,500ms	93,500ms	99,000ms	85,500ms	95,000ms	85,500ms	91,000ms	94,000ms
3	92,333ms	100,330ms	110,330ms	102,333ms	98,330ms	89,666ms	101,000ms	92,000ms	107,660ms	95,000ms
10	90,500ms	101,700ms	92,300ms	96,600ms	93,900ms	101,700ms	95,100ms	92,300ms	99,700ms	94,200ms
100	97,580ms	100,160ms	100,270ms	100,020ms	102,540ms	99,950ms	102,060ms	96,730ms	101,460ms	102,500ms
1000	97,799ms	97,586ms	98,402ms	100,130ms	99,965ms	98,306ms	98,910ms	98,954ms	98,742ms	97,910ms

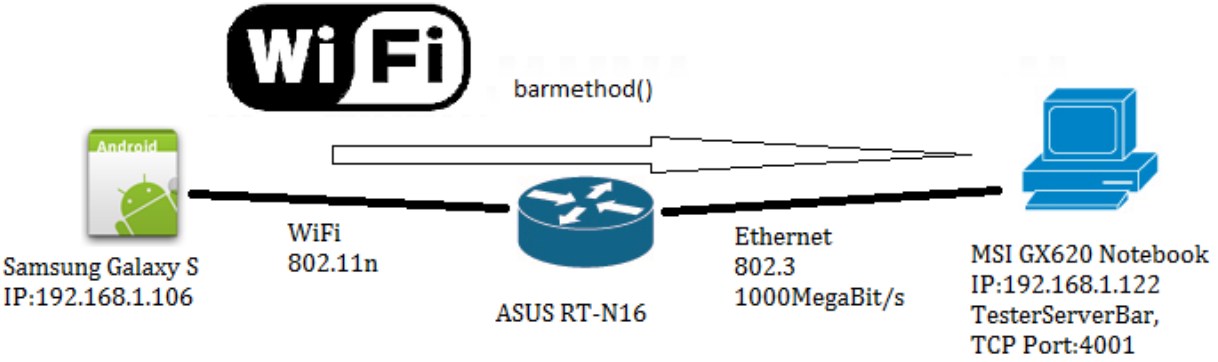
3. táblázat Bluetooth Android-ról Android-ra mérés adatai



36. ábra Bluetooth Androidról-Android mérés diagramja

A 36. ábrán a Bluetooth feletti Androidról-Androidra irányuló mérés eredményei láthatók.

4.3.4 Wifi, TCP Android-ról PC-re



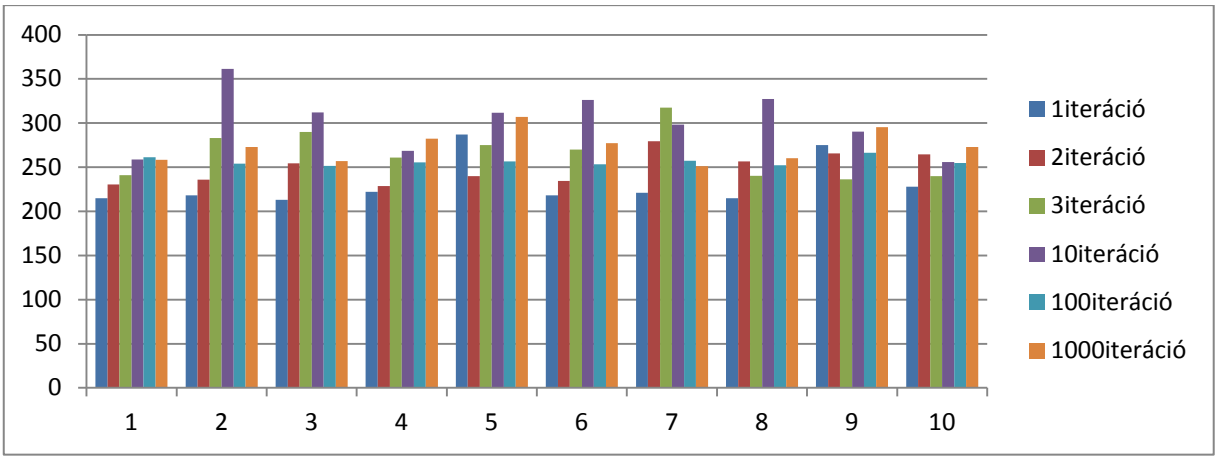
37. ábra Wifi, TCP Androidról-PC

A 37. ábra azt a mérési elrendezést szemléltetni, amikor, Android a kliens, PC a szerver és az adatkapcsolati protokollok Wifi, illetve Ethernet, a hálózati protokoll IPv4 a szállítási protokoll TCP.

4.3.4.1.1 Adatok

	1.mérés	2.mérés	3.mérés	4.mérés	5.mérés	6.mérés	7.mérés	8.mérés	9.mérés	10.mérés
inic	275,000ms	268,000ms	225,000ms	320,000ms	297,000ms	299,000ms	272,000ms	226,000ms	345,000ms	347,000ms
1	215,000ms	218,000ms	213,000ms	222,000ms	287,000ms	218,000ms	221,000ms	215,000ms	275,000ms	228,000ms
2	230,500ms	236,000ms	254,500ms	228,500ms	240,000ms	234,500ms	279,500ms	256,500ms	265,500ms	264,500ms
3	241,000ms	283,000ms	290,000ms	261,000ms	275,000ms	270,000ms	317,660ms	240,330ms	236,330ms	240,000ms
10	258,700ms	361,300ms	311,900ms	268,500ms	311,700ms	326,200ms	298,200ms	327,400ms	290,400ms	256,000ms
100	261,390ms	254,190ms	251,430ms	255,610ms	256,730ms	253,370ms	257,390ms	252,240ms	266,420ms	254,780ms
1000	258,500ms	272,980ms	256,940ms	282,430ms	306,900ms	277,320ms	251,040ms	260,360ms	295,480ms	272,990ms

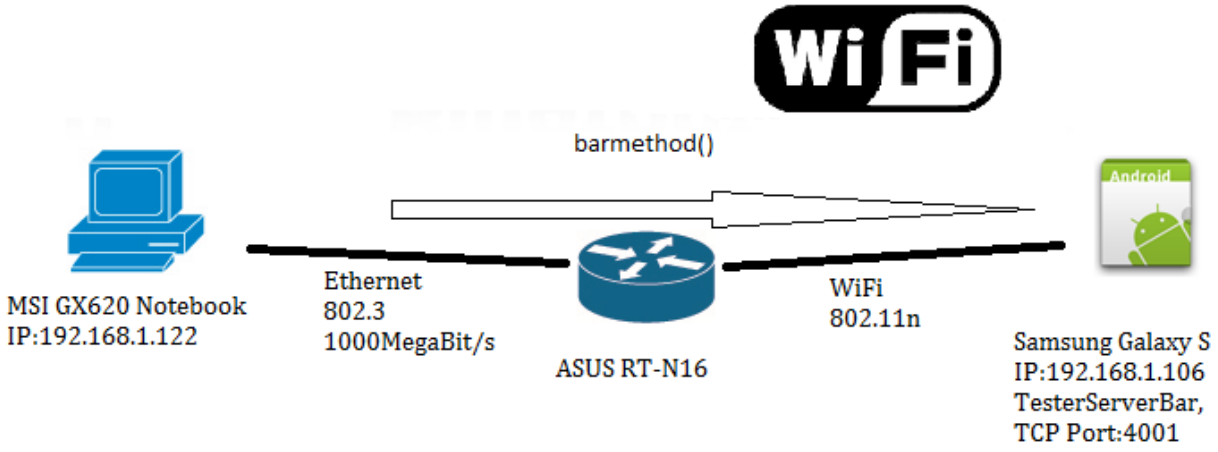
4. táblázat a Wifi, TCP Androidról-PC-re mérés adatai.



38. ábra a Wifi, TCP Androidról-PC-re mérés adiagramja

A 38. ábrán a TCP feletti Androidról-PC-re irányuló mérés eredményei láthatók.

4.3.5 Wifi, TCP PC-ről Android -ra



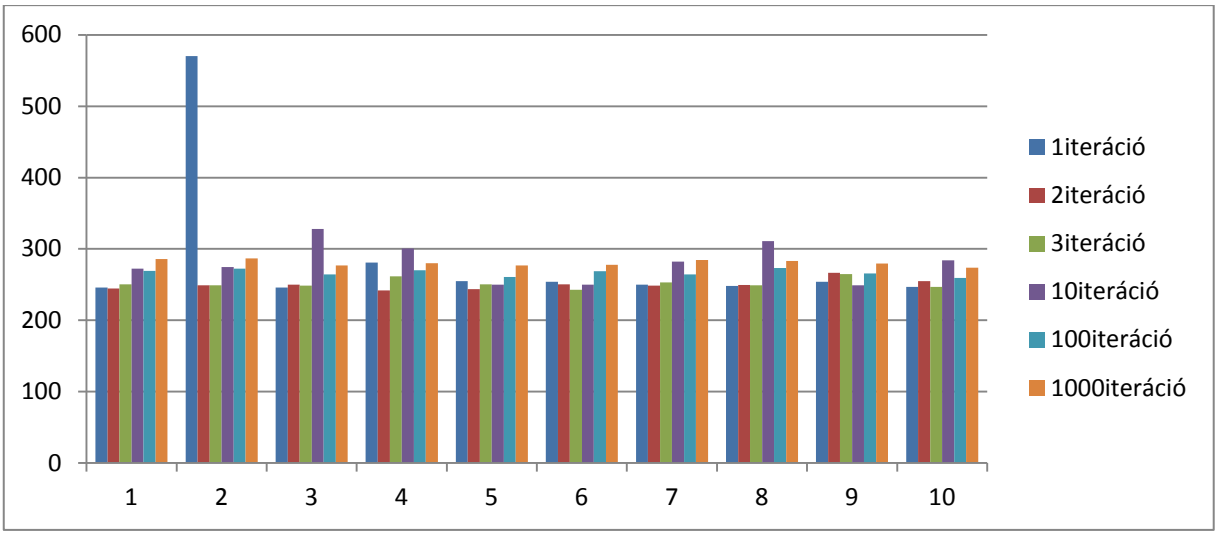
39. ábra Wifi, TCP PC-ről-Androidra

A 39. ábra azt a mérési elrendezést szemléltetni, amikor, PC a kliens, Android a szerver és az adatkapcsolati protokollok Wifi, illetve Ethernet, a hálózati protokoll IPv4, a szállítási protokoll TCP.

4.3.5.1.1 Adatok

	1.mérés	2.mérés	3.mérés	4.mérés	5.mérés	6.mérés	7.mérés	8.mérés	9.mérés	10.mérés
inic	772,000ms	726,000ms	587,000ms	789,000ms	912,000ms	1 059,000ms	531,000ms	533,000ms	667,000ms	588,000ms
1	246,000ms	570,000ms	246,000ms	281,000ms	255,000ms	254,000ms	250,000ms	248,000ms	254,000ms	247,000ms
2	244,500ms	249,000ms	250,000ms	242,000ms	243,500ms	250,500ms	248,500ms	249,500ms	266,500ms	255,000ms
3	250,666ms	249,000ms	248,666ms	261,666ms	250,333ms	243,000ms	253,333ms	249,000ms	264,666ms	246,667ms
10	272,500ms	274,500ms	328,000ms	300,600ms	250,010ms	250,000ms	282,200ms	311,100ms	249,300ms	284,100ms
100	269,080ms	272,310ms	264,390ms	270,010ms	260,980ms	268,670ms	264,180ms	273,330ms	265,740ms	259,600ms
1000	285,829ms	286,785ms	277,035ms	280,197ms	276,867ms	277,623ms	284,371ms	283,157ms	279,505ms	273,693ms

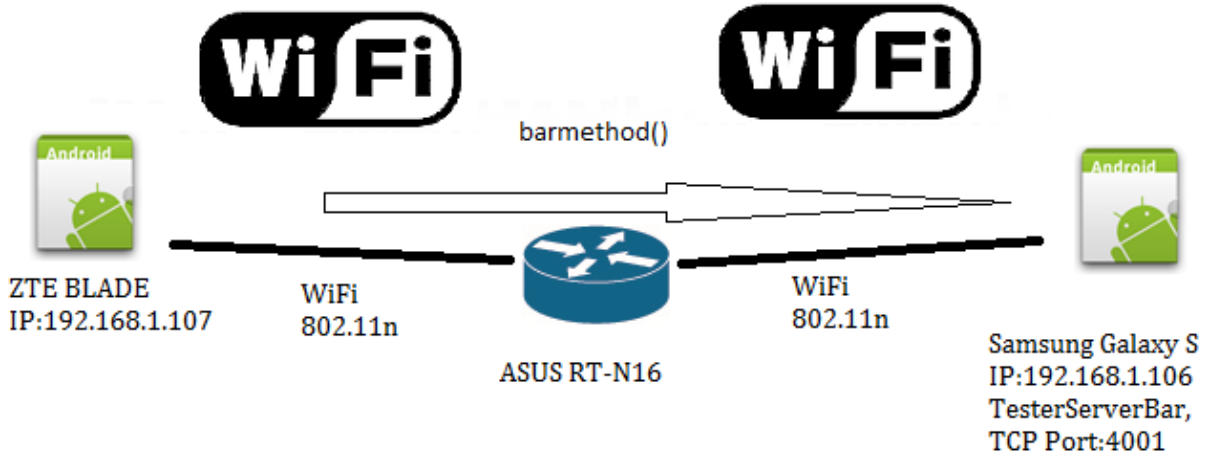
5. táblázat Wifi, TCP PC-ről-Androidra mérés adatai



40. ábra a Wifi, TCP PC-ről-Androidra mérés diagramja

A 40. ábrán a TCP feletti PC-ről-Androidra irányuló mérés eredményei láthatók.

4.3.6 Wifi, TCP Android-ról Android-ra



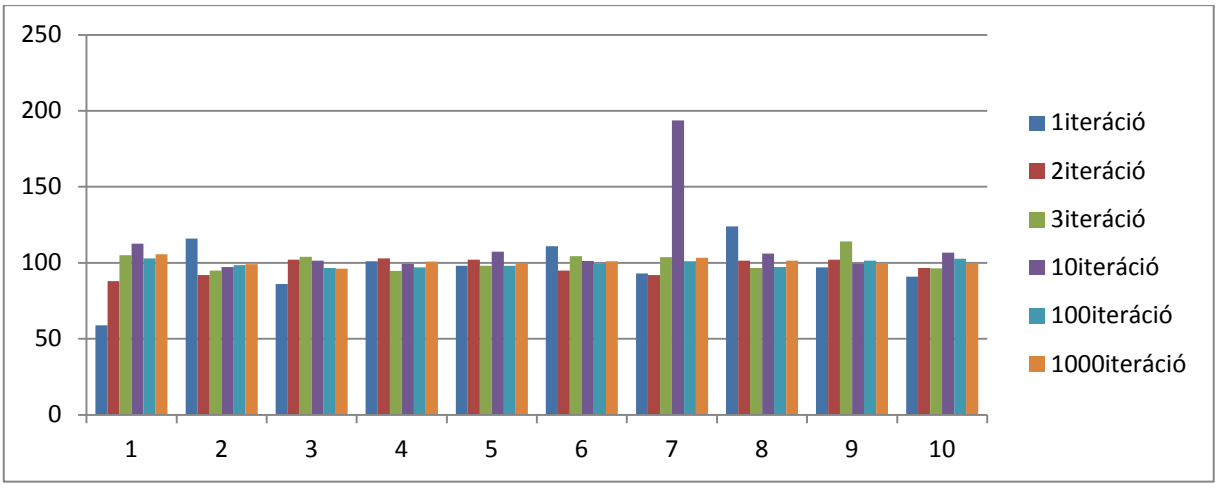
41. ábra Wifi, TCP Androidról-Androidra

A 41. ábra azt a mérési elrendezést szemléltetni, amikor, Android a kliens, Android a szerver és az adatkapcsolati protokoll Wifi, a hálózati protokoll IPv4, a szállítási protokoll TCP.

4.3.6.1.1 Adatok

	1.mérés	2.mérés	3.mérés	4.mérés	5.mérés	6.mérés	7.mérés	8.mérés	9.mérés	10.mérés
inic	1 190,000ms	515,000ms	560,000ms	478,000ms	521,000ms	676,000ms	754,000ms	320,000ms	554,000ms	910,000ms
1	59,000ms	116,000ms	86,000ms	101,000ms	98,000ms	111,000ms	93,000ms	124,000ms	97,000ms	91,000ms
2	88,000ms	92,000ms	102,000ms	103,000ms	102,000ms	95,000ms	92,000ms	101,500ms	102,000ms	96,500ms
3	105,000ms	95,000ms	104,000ms	94,667ms	98,000ms	104,333ms	103,667ms	96,667ms	114,000ms	96,333ms
10	112,700ms	97,300ms	101,500ms	99,400ms	107,300ms	101,200ms	193,600ms	106,100ms	99,500ms	106,700ms
100	102,880ms	98,590ms	96,660ms	97,120ms	98,000ms	99,500ms	101,040ms	97,140ms	101,390ms	102,780ms
1000	105,680ms	99,140ms	96,183ms	100,882ms	99,887ms	100,971ms	103,266ms	101,349ms	99,684ms	99,660ms

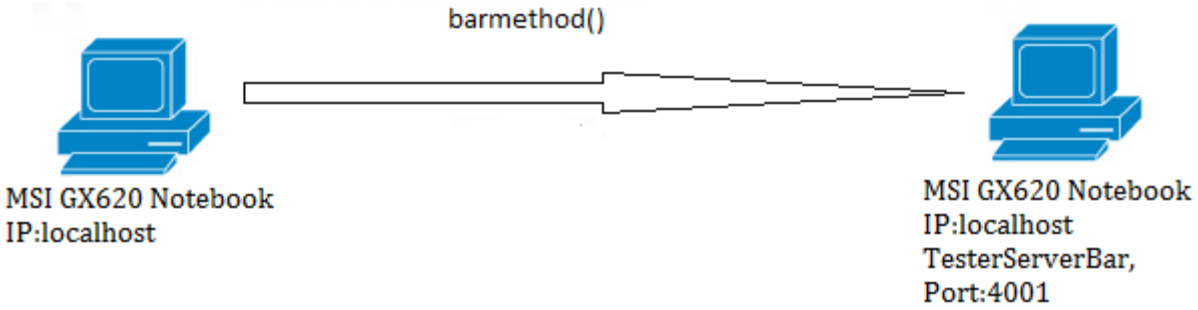
6. táblázat a Wifi, TCP Androidról-Androidra mérés adatai



42. ábra a Wifi, TCP Androidról-Androidra mérés diagramja

A 42. ábrán a TCP feletti Androidról-Androidra irányuló mérés eredményei láthatók.

4.3.7 Localhost



43. ábra Localhoston mérés

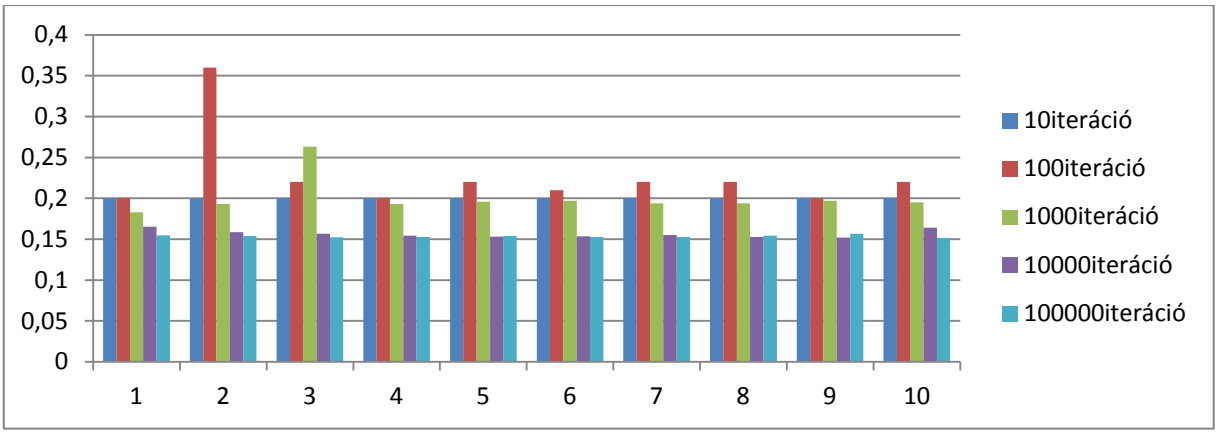
A 43. ábra a localhoston történt mérést ábrázolja.

4.3.7.1.1 Adatok ARPC

	1.mérés	2.mérés	3.mérés	4.mérés	5.mérés	6.mérés	7.mérés	8.mérés	9.mérés	10.mérés
inic	29,00000ms	33,00000ms	30,00000ms	31,00000ms	34,00000ms	42,00000ms	37,00000ms	29,00000ms	32,00000ms	28,00000ms
1	0,00000ms	0,00000ms	0,00000ms	1,00000ms	0,00000ms	0,00000ms	0,00000ms	1,00000ms	0,00000ms	0,00000ms
2	0,00000ms	0,00000ms	0,50000ms	0,00000ms	0,50000ms	0,00000ms	0,50000ms	0,00000ms	0,50000ms	0,00000ms
3	0,33000ms	0,00000ms	0,33000ms	0,00000ms	0,33000ms	0,33000ms	0,33000ms	0,00000ms	0,00000ms	0,33000ms
10	0,20000ms	0,20000ms	0,20000ms	0,20000ms	0,20000ms	0,20000ms	0,20000ms	0,20000ms	0,20000ms	0,20000ms
100	0,20000ms	0,36000ms	0,22000ms	0,20000ms	0,22000ms	0,21000ms	0,22000ms	0,22000ms	0,20000ms	0,22000ms
1000	0,18300ms	0,19300ms	0,26300ms	0,19300ms	0,19600ms	0,19700ms	0,19400ms	0,19400ms	0,19700ms	0,19500ms
10000	0,16520ms	0,15850ms	0,15660ms	0,15430ms	0,15330ms	0,15340ms	0,15500ms	0,15280ms	0,15180ms	0,16430ms
100000	0,15470ms	0,15387ms	0,15224ms	0,15279ms	0,15396ms	0,15275ms	0,15275ms	0,15438ms	0,15665ms	0,15151ms

7. táblázat ARPC keretrendszer localhoston mért adatai

A 7. táblázat a localhoston mért adatokat tartalmazza, a nagyobb sebesség miatt több iterációt kellett elvégeznem.



44. ábra ARPC keretrendszer localhoston mért adatainak diagramja

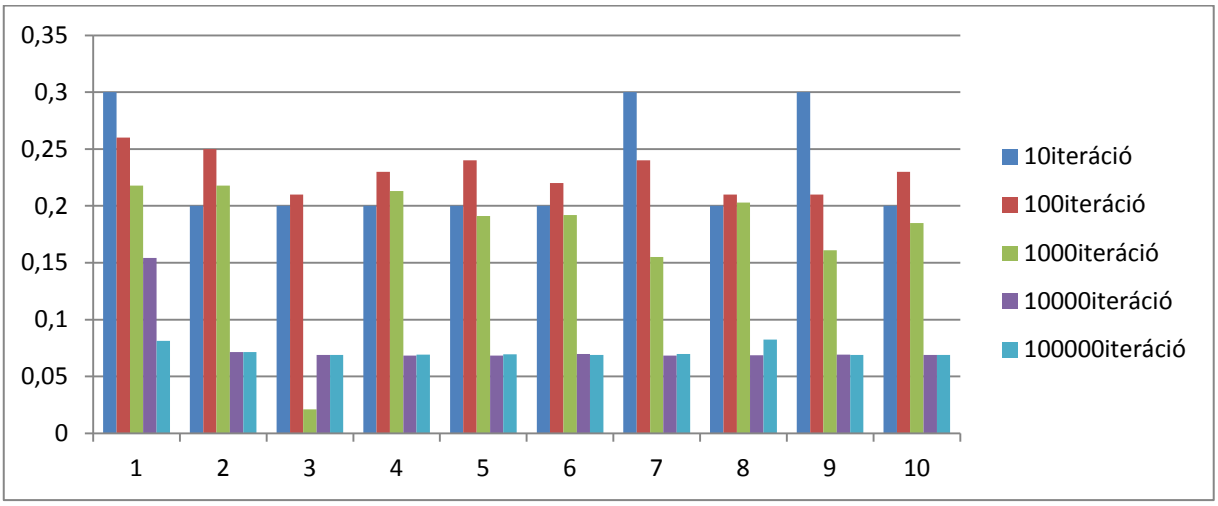
A 44. ábrán a mért eredmények diagramja látható, 10 iterációtól indulva, ugyanis kevés számú iterációra pontatlan a mérés, ami az adatokból látszik.

4.3.7.1.2 Adatok RMI

	1.mérés	2.mérés	3.mérés	4.mérés	5.mérés	6.mérés	7.mérés	8.mérés	9.mérés	10.mérés
inic	1,00000ms	1,00000ms	1,00000ms	1,00000ms	1,00000ms	1,00000ms	1,00000ms	1,00000ms	1,00000ms	1,00000ms
1	0,00000ms	0,00000ms	0,00000ms	0,00000ms	0,00000ms	1,00000ms	1,00000ms	0,00000ms	0,00000ms	0,00000ms
2	0,50000ms	0,00000ms	0,00000ms	0,00000ms	0,00000ms	0,50000ms	0,50000ms	0,00000ms	0,50000ms	0,00000ms
3	0,33000ms	0,33000ms	0,33000ms	0,33000ms	0,33000ms	0,00000ms	0,33000ms	0,00000ms	0,33000ms	0,00000ms
10	0,30000ms	0,20000ms	0,20000ms	0,20000ms	0,20000ms	0,20000ms	0,30000ms	0,20000ms	0,30000ms	0,20000ms
100	0,26000ms	0,25000ms	0,21000ms	0,23000ms	0,24000ms	0,22000ms	0,24000ms	0,21000ms	0,21000ms	0,23000ms
1000	0,21800ms	0,21800ms	0,02100ms	0,21300ms	0,19100ms	0,19200ms	0,15500ms	0,20300ms	0,16100ms	0,18500ms
10000	0,15430ms	0,07140ms	0,06910ms	0,06840ms	0,06850ms	0,06990ms	0,06850ms	0,06870ms	0,06930ms	0,06910ms
100000	0,08133ms	0,07160ms	0,06890ms	0,06922ms	0,06939ms	0,06906ms	0,06985ms	0,08237ms	0,06894ms	0,06896ms

8. táblázat a localhoston mért RMI mérés adatai

A 8. táblázat a localohoston mért RMI mérés adatait tartalmazza.



45. ábra

A 45. ábra, a localhoston mért RMI adatainak diagramját tartalmazza.

4.4 Mérési eredmények értékelése

A mérési eredmények a hálózati réteg és az irány szempontjából elég nagy szórást mutatnak.

A kapcsolatkiépítési idő nagyságrendekkel volt nagyobb, mint az átlagos késleltetés, erre a keretrendszer használata során figyelni kell, hogy az esetleges time-out értékek meghatározásánál ez ne okozzon gondot. Nagyságrendi eltérésük miatt nem is ábrázoltam

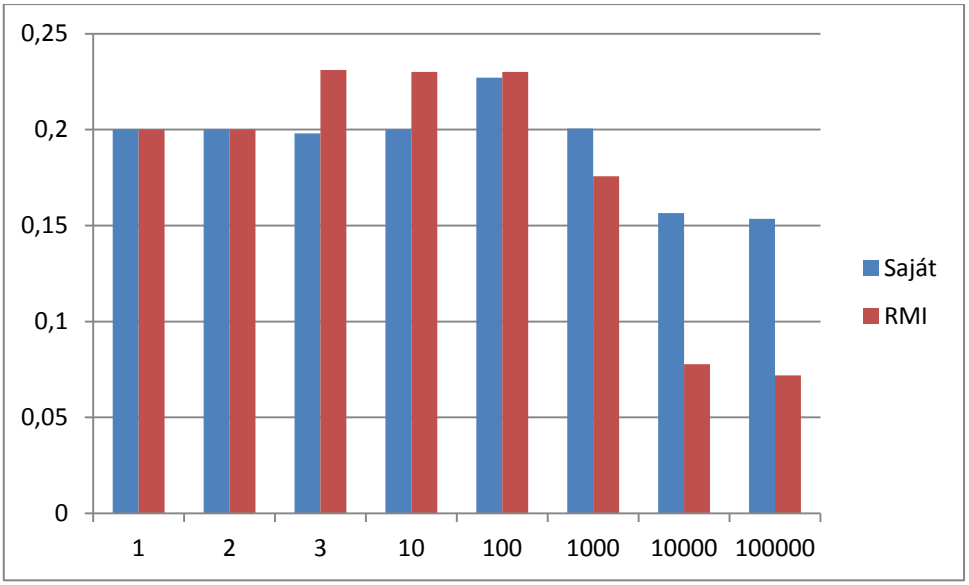
grafikonon a kapcsolatépítési időt, továbbá a méréseket úgy végeztem, hogy már léteztek a kapcsolatok.

A Bluetooth alapú hálózat esetén az Android-PC közötti 50ms körüli átlagos késleltetés akár valós idejű játékok futtatását is lehetővé teszi. Android-Android között a 100ms körüli késleltetés határeset valósidejű alkalmazások szempontjából. A különböző Bluetooth-stack implementációk miatt lehet ekkora a különbség.

A TCP rosszabb késletetései Android-PC között 300ms között voltak. Valószínűleg a TCP protokoll overheadja nagyobb és a router Wifi-Ethernet átalakítása is közbejátszik.

4.4.1 RMIvel összehasonlítás

Minden iterációs számra vettem a 10 mérés átlagát és ezeket hasonlítottam össze.



46. ábra ARPC keretrendszer és az RMI összehasonlítása.

A 46. ábra az általam írt keretrendszert és az RMI-t hasonlítja össze.

Az általam írt keretrendszer átlagos késleltetése nagyságrendileg nem marad el az RMI keretrendszertől.

5 Projektösszegzés

Az alapvető probléma az Android platform azon hiányossága, hogy nem támogat RPC megoldást, így a projekt célja egy olyan moduláris kiegészíthető keretrendszer megalkotása volt, ami fut Android platformra.

A projekt során törekedtem a tisztán réteges szerkezet elérésére, de a hibakezelés és a címzés miatt ezt nem sikerült megvalósítani, de ennek ellenére nagyfokú modularitást értem el absztrakt osztályok és interfészek definiálásával.

A projekt során meg kellett oldanom többek között a hálózatkezelés alapvető problémáit, mint kapcsolat felépítés, adatátvitel, címzés. Ezekre a problémákra sikerült absztrakt megoldást adnom. Konkrét megoldásként implementáltam TCP-re és Bluetoothra. TCP estében a standard java osztályokat használtam. Bluetooth esetén az Android platform standard Bluetooth implementációját használtam fel. Ezen felül létrehoztam egy Java ME alapú implementációt a BlueCove osztálykönyvtár felhasználásával, ami lehetővé tette a távoli eljáráshívást Android és PC közötti Bluetooth felett. Szerializálásra a java szeralizáló mechanizmusát használtam fel.

További probléma volt a metódusok és objektumok távoli azonosítása és a dinamikus metódushívás. A forráskód generálás elkerülése végett a Reflection API-t használtam, ennek a problémának a megoldására. Ez teljesítményveszteséget okozott, de cserében teljesen dinamikus és általános lett a keretrendszer. Az objektumok, metódusok azonosítása, többek között az osztályadataik alapján történik. Ugyanis annak érdekében, hogy egy osztályból több példányt külön lehessen azonosítani, az azonosításhoz még hozzáraktam egy megadható ID-t is.

A Reflection API további hozománya a dinamikus proxy osztály, aminek köszönhetően a távoli eljáráshívás teljes folyamatát sikerült elfedni és az RMI-hez hasonló kényelmet és funkcionalitást elérni.

A távoli metódusok megjelölésére, az RMI-hez hasonlóan bevezettem egy jelölő interfészt.

Tovább növelik a modularitást a manager osztályok, amik a dinamikus konfigurációkezelést teszik lehetővé, ugyanis elfedik a különböző feladatot ellátó osztályokat.

A keretrendszer implementálása után, méréseket végeztem, amiben a hálózati késletetést vizsgáltam különböző hálózati implementációk esetében. A mérések alapján a keretrendszer kapcsolatépítési ideje elég magas, így erre figyelni kell, de mivel az átlagos késleltetési ideje elfogadható, ezért a keretrendszer kevésbé késleltetés érzékeny alkalmazások futtatására tökéletesen alkalmas.

A projekt során az említett összes problémát megoldottam és létrehoztam egy működőképes keretrendszert, ami a korábban felvázolt összes követelményt maradéktalanul teljesíti.

5.1 RMI vel összehasonlítás

5.1.1 Előny

Az AJAR Android platformon, Bluetooth felett és Java platformok között is működik.

5.1.2 Hasonló

Mindkét keretrendszer proxy osztályok segítségével teljesen elfedi a távoli eljáráshívást. Mindkét keretrendszer kiegészíthető.

5.1.3 Hátrány

A AJAR lassabb.

5.2 Továbbfejlesztési lehetőségek

A projekthez újabb hálózati rétegeket lehet fejleszteni, például UDP.

A teljesen dinamikus viselkedés feladásával és forráskód generálással reflection kikerülhető, ami árán sebességnövekedést nyerhetünk.

Titkosításhoz SSL alapú hálózati rétegeket fejleszthetünk.

Web Szolgáltatások elérése új hálózati réteg implementálásával.

5.3 Hivatkozások

[1] http://www.idc.com/getdoc.jsp?containerId=prUS22653511

[2] http://www.idc.com/about/viewpressrelease.jsp?containerId=prUS22689111

[3] http://www.usstockmarket.tv/tech-blog/2011/06/needham-androids-market-share-peaked-in-march/

[4] http://www.idc.com/

[5] http://developer.android.com/guide/basics/what-is-android.html

[6] http://download.oracle.com/javase/1.4.2/docs/api/java/lang/reflect/Proxy.html

[7] http://download.oracle.com/javase/1.4.2/docs/api/java/lang/reflect/InvocationHandler.html

[8] http://en.wikipedia.org/wiki/Universally_unique_identifier

[9] http://bluecove.org/bluecove/apidocs/javax/bluetooth/ServiceRecord.html

[10] http://www.samsung.com/hu/consumer/mobile-phone/mobile-phones/touchscreen/GT-I9000HKDXEH/index.idx?pagetype=prd_detail&tab=specification

[11] http://www.gsmarena.com/zte_blade-3391.php

[12] http://www.asus.com/Networks/Wireless_Routers/RTN16/#specifications