NYÍREGYHÁZI EGYETEM

Diplomamunka címe

Hálózatdiagnosztikai tesztalkalmazás JavaFX grafikus felületen

Hallgató neve: Németh Gyula
Hallgató szakjának megnevezése
Programtervező Informatika
Konzulens: Vályi Sándor Zoltán egyetemi docens

2026

Java Hálózat Monitor Program – Technikai Dokumentáció

Készítette: Németh Gyula – PLBHOJ

Tartalomjegyzék

1. Bevezetés

A modern informatikai rendszerek működése nagymértékben függ a hálózati kapcsolatok megbízhatóságától és teljesítményétől. Az internetkapcsolatok sebessége, késleltetése és stabilitása alapvetően befolyásolja a felhasználói élményt, legyen szó böngészésről, videóhívásokról vagy online játékokról. Emiatt egyre nagyobb igény mutatkozik olyan eszközökre, amelyekkel a hálózat teljesítménye folyamatosan nyomon követhető és elemezhető. Jelen dokumentáció egy Java nyelven fejlesztett **Network Monitor** (Hálózatfigyelő) programot mutat be, amely átfogó képet nyújt a hálózat állapotáról, és segít az esetleges problémák diagnosztizálásában.

A dokumentáció célja, hogy technikai részletességgel bemutassa a program működését, főbb moduljait és azok szerepét, az alkalmazott protokollokat és technológiákat, valamint ismertesse a minőségbiztosítási szempontokat. A leírás kitér a program által nyújtott funkciókra – úgymint a hálózati sebesség és késleltetés mérése, útvonal-elemzés (traceroute), helyi hálózati eszközök felderítése, UPnP alapú porttovábbítás, különböző csomagtípusok vizsgálata (unicast, broadcast, multicast, anycast) –, továbbá bemutatja a beépített mesterséges intelligencia modult, amely a naplózott adatok elemzésével ajánlásokat fogalmaz meg a hálózat javítása érdekében.

A dokumentum szerkezetileg tartalmazza a program rendeltetésének ismertetését, a funkcionális modulok részletes magyarázatát, az architekturális felépítést és elméleti hátteret, valamint a fejlesztés során figyelembe vett minőségbiztosítási elveket. Mellékletként – a szövegbe ágyazva – szerepelnek szemléltető ábrák, diagramok és képernyőképek a program működéséről. Végezetül a **Hivatkozások** fejezetben

felsorolásra kerülnek a felhasznált források, ideértve vonatkozó RFC dokumentumokat, Java dokumentációkat és hálózati protokollok szakirodalmát.

2. A program célja és szerepe

A *Network Monitor* program elsődleges célja a hálózati forgalom és kapcsolat minőségének folyamatos monitorozása és kiértékelése. A program segítségével a felhasználó valós időben nyerhet információkat az internetkapcsolata sebességéről (le- és feltöltési sávszélességéről), a hálózati késleltetés mértékéről (*ping idő*), valamint a késleltetés ingadozásáról (*jitter*). Ezek a paraméterek meghatározóak az olyan időérzékeny alkalmazásoknál, mint a videóstreaming vagy VoIP hívások, hiszen a túl nagy késleltetés vagy a jelentős jitter minőségi romláshoz vezethet (pl. akadozó hang vagy kép)[1][2]. A program rendszeres mérésekkel és naplózással biztosítja, hogy a felhasználó hosszabb távon is nyomon követhesse a kapcsolat teljesítményének alakulását, és azonosíthassa az esetleges problémás időszakokat.

Emellett a program fontos szerepet tölt be a hálózati hibaelhárítás támogatásában is. A beépített traceroute modul lehetővé teszi a hálózati útvonal elemzését, vagyis annak feltérképezését, hogy az adatcsomagok milyen útvonalon jutnak el egy távoli szerverig. Ez segít azonosítani, ha valamelyik útválasztónál (router) probléma vagy késleltetés adódik. Ugyancsak hasznos funkció a helyi hálózat eszközeinek felderítése (LAN scan), amellyel áttekintést kaphatunk arról, hogy a belső hálózatunkon milyen eszközök vannak jelen és elérhetők-e. Ez hálózatüzemeltetési szempontból és biztonsági okokból is lényeges lehet.

A Network Monitor program mindezeken túlmutatóan **AI modult** is tartalmaz, amely a gyűjtött teljesítményadatok és naplófájlok elemzésével automatikus kiértékelést nyújt. Ez a modul mesterséges intelligencia és gépi tanulás segítségével igyekszik felismerni a hálózati teljesítményben megjelenő mintázatokat, illetve anomáliákat, és ezek alapján ajánlásokat fogalmaz meg a felhasználó számára (például javaslatot tesz a router újraindítására vagy csatornaváltásra magas ping esetén). A program így nem csupán *passzív megfigyelője* a hálózatnak, hanem *proaktív tanácsadóként* is működik.

Összességében a Network Monitor célja, hogy egyetlen integrált alkalmazásban biztosítsa mindazon eszközöket, amelyekkel a felhasználók – legyenek akár rendszergazdák vagy otthoni internethasználók – áttekinthetik és javíthatják hálózatuk teljesítményét. A program szerepe tehát a hálózati diagnosztika és optimalizáció megkönnyítése: segít feltárni a problémák gyökerét (legyen az a szolgáltatói oldal, a helyi hálózat, vagy egy adott eszköz), és hozzájárul a stabil, gyors hálózati működés fenntartásához.

3. Főbb funkciók részletes magyarázata

3.1 Monitor modul (sebesség, ping, jitter, naplózás)

A Monitor modul feladata a hálózati kapcsolat legfontosabb teljesítménymutatóinak folyamatos mérése és megjelenítése. Ide tartozik különösen a le- és feltöltési sebesség, a ping (hálózati késleltetés), a jitter (késleltetés-ingadozás), illetve ezek naplózása. A

modul működése során meghatározott időközönként (például percenként) teszteli az internetkapcsolatot, és rögzíti az eredményeket.

- **Sebesség mérése:** A program a letöltési és feltöltési sebességet úgy méri, hogy ismert méretű adatcsomagokat tölt le egy tesztszerverről, illetve tölt fel oda, miközben méri az átviteli időt. A kapott adatmennyiség és idő alapján számítja ki a sávszélességet Mbps-ban (megabit per szekundum). A *letöltési sebesség* azt mutatja meg, milyen gyorsan tudunk adatot fogadni a hálózatról, és tipikusan nagyobb értékű, mivel az internetes tevékenységek többsége (pl. weboldalak betöltése, videók streamelése) letöltés-központú[3]. Ezzel szemben a *feltöltési sebesség* azt jelzi, milyen gyorsan tudunk adatot küldeni a hálózatra (például egy fájl e-mailben való elküldése vagy videóhívás során a saját képünk továbbítása), és szintén Mbps-ban mérjük[4]. A program grafikus felületen kijelzi az aktuális mérési eredményeket, valamint trenddiagramot is készít a sebesség értékeinek alakulásáról, így a felhasználó hosszabb távon nyomon követheti, hogyan változik az internetkapcsolat teljesítménye.

- **Ping és jitter mérése:** A *ping* a hálózat reakcióidejét jelenti – azt az időtartamot, amely alatt egy csomag elér egy célállomáshoz és vissza[2]. A program ICMP Echo Request üzenetek küldésével méri a pinget, és az Echo Reply visszaérkezéséig eltelt időt milliszekundumban (ms) rögzíti. Egy gyors ping (alacsony késleltetés) azt jelzi, hogy a kapcsolat nagyon hamar válaszol egy kérésre, míg a magas ping érték "lagot", késleltetést eredményez a hálózati kommunikációban. Általánosságban elmondható, hogy a ~100 ms alatti ping már jónak tekinthető a legtöbb felhasználói alkalmazás szempontjából[5], míg az ez feletti értékeknél a felhasználó már érezhet némi késést például böngészés vagy játék közben. A *jitter* ezzel szemben nem magát a késleltetést méri, hanem annak **ingadozását** – azt, hogy mennyire stabilak vagy változékonyak a ping értékek az idő folyamán[6]. Technikailag a jittert a késleltetés értékeinek átlagostól való eltéréseként, szórásaként definiáljuk (azaz a ping idők átlagos deviációjaként a középértéktől)[7]. A program a jitter számításához egymást követő több ping mérés eredményét használja fel: kiszámítja az egyes minták közötti késleltetéskülönbségeket, illetve a statisztikai szórást. Egy stabil, jó minőségű kapcsolatnál a jitter alacsony (néhány ms nagyságrendű), míg egy ingadozó kapcsolatnál magas jitter értékeket kapunk. Tipikusan a 30 ms alatti jitter érték tekinthető elfogadhatónak vagy jónak, az ennél magasabb jitter pedig arra utal, hogy a csomagok késleltetése nagyon változó, ami problémát okozhat valós idejű alkalmazások (pl. VoIP hívás, videókonferencia) esetén[8]. A program a ping és jitter eredményeket is folyamatosan naplózza, és grafikonon jeleníti meg, így látható például, ha esténként megugrik a késleltetés ingadozása (ami hálózati túlterheltségre utalhat).

- **Naplózás:** A Monitor modul a mért adatokat (sebességek, ping, jitter) idősorosan eltárolja egy naplófájlban vagy adatbázisban. A naplóbejegyzések tartalmazzák a mérés időbélyegét (dátum, idő), a mérés típusát és eredményét. Például egy tipikus napló sor tartalmazhatja: `2025-11-26 20:30:00 - Ping: 42 ms, Jitter: 5 ms, Download: 94 Mbps, Upload: 32 Mbps`. A naplózás célja kettős: egyrészt

lehetőséget nyújt historikus elemzésekre (pl. visszakereshető, hogy egy adott időpontban milyen volt a hálózat teljesítménye), másrészt inputként szolgál az AI modul számára. A program felületén a felhasználó lekérdezheti és megtekintheti a naplózott adatokat különböző időintervallumokra szűrve, illetve exportálhatja azokat további külső elemzés céljából.

Összefoglalva, a Monitor modul valós időben *figyeli* a hálózat legfontosabb mutatóit, és érthető formában prezentálja azokat a felhasználó felé. Segítségével azonnal észrevehetők a teljesítmény-ingadozások (például hirtelen sávszélességcsökkenés vagy megugró késleltetés), és az adatok birtokában megalapozott döntések hozhatók a hálózat esetleges beavatkozásairól (router újraindítása, szolgáltató értesítése stb.).

3.2 Traceroute modul (útvonal elemzés)

A traceroute modul a hálózati útvonal elemzésére szolgál. Célja, hogy feltérképezze, milyen útvonalon (milyen köztes csomópontokon keresztül) jutnak el a hálózati csomagok a kiindulási géptől egy megadott célkiszolgálóig. A *traceroute* parancs és módszer klasszikusan arra épül, hogy különböző *TTL (Time To Live)* értékekkel küld csomagokat, és figyeli a routerek válaszait. A TTL egy csomag IP fejléceiben található számláló, amely minden egyes hálózati áthaladásnál (hop) csökken, és nullára érve megakadályozza, hogy a csomag végtelenítve keringjen a hálózatban. A traceroute ezt a mechanizmust használja ki: először egy csomagot küld ki TTL=1 értékkel, amely az első routernél nullára csökken, így az a router eldobja a csomagot és visszaküld egy ICMP *Időtúllépés* (Time Exceeded) üzenetet a feladónak. Ebből a program megtudja az első hop IP-címét. Ezután TTL=2-vel küld csomagot, ami a második hopnál lesz nulla – az onnan jövő ICMP válaszból megismerhető a második router, és így tovább. Ily módon, a TTL fokozatos növelésével a traceroute feltárja az útvonalat alkotó összes hopot, egészen a célig[9]. A modul tipikusan megjeleníti az egyes hopok sorszámát, a router IP-címét vagy hostnevét, valamint a válaszidőt (ping értéket) minden egyes hop esetében.

A program Java nyelven történő megvalósítása kihasználja az alapvető hálózati könyvtárakat és esetenként külső parancsok meghívását. Mivel a nyers ICMP csomagküldés alapértelmezetten nem triviális Java-ból (a Java nem engedélyezi közvetlenül a *raw socket* használatát ICMP-re, kivéve néhány platformfüggő módot vagy harmadik féltől származó könyvtárakkal), a traceroute modul kétféleképpen működhet:

1. **Beépített megvalósítás TTL manipulálással:** A Java `InetAddress` osztályának bizonyos metódusai (pl. `isReachable()` adott TTL-lel) segítségével megkísérelhető az útvonal detektálása. A program növeli a TTL értéket és figyeli, hogy az `isReachable(timeout)` hívás sikeres lesz-e a célállomásra egy adott ugrásszám mellett. Ha nem, akkor az adott TTL-nél kapott ICMP válaszból (amit a rendszer hálózati stackje továbbít) ki lehet nyerni az aktuális hop címét. Ez a módszer limitáltan ugyan, de megvalósítható teljesen Java kódból is, és a program képes lehet ezzel listázni a traceroute útvonalat.

2. **Rendszer parancs meghívása:** Gyakran egyszerűbb és megbízhatóbb megoldás, ha a program a háttérben meghívja a rendszer natív `traceroute`

(Linux/Mac) vagy `tracert` (Windows) parancsát a megadott célcímmel, és a parancs kimenetét elemzi (parse-olja). A program a Java `ProcessBuilder` vagy `Runtime.exec()` használatával elindít egy traceroute folyamatot, majd valós időben beolvassa annak kimenetét. Ezt követően karakterfelismeréssel azonosítja a sorokban található hop információkat (sorszám, IP-cím, késleltetési idők). Ezt a módszert alkalmazva a modul a natív eszköz megbízhatóságát kihasználva tud pontos eredményt adni, a Java pedig csak a megjelenítésért és további feldolgozásért felel.

A traceroute modul tipikus kimenete a programban az alábbihoz hasonló (pl. egy példával a `google.com` felé):

```
Ugrás  Cím                          Késleltetés
1.     192.168.1.1                  1 ms
2.     10.0.0.1                     8 ms
3.     72.14.194.1                  13 ms
4.     108.170.250.1                18 ms
5.     142.250.190.78 (google.com)  22 ms
```

Az egyes sorok mutatják a hopszámot, a router IP-címét (és ha feloldható, akkor hostnevét), valamint a ping időt hozzá. A modulból ezek az adatok interaktív módon is kinyerhetők: a felhasználó megadhat egy tetszőleges hostnevet vagy IP címet, amely felé a traceroute lefut, és a program megjeleníti a teljes útvonalat. Emellett a modul segít értelmezni az eredményeket: például külön jelölheti, ha egy adott hopnál időtúllépés történt (amit a traceroute `* * *` formában szokott jelezni) – ez jellemzően azt jelenti, hogy az adott router nem válaszol traceroute megkeresésekre. A program így támogatja a hálózati diagnosztikát: megmutatja, mely ponton lép fel magas késleltetés vagy csomagvesztés (ha valamelyik hop nem érhető el), ezáltal segít behatárolni a problémát (pl. a 3. útválasztónál van jelentős késedelem, ami a szolgáltató gerinchálózatán lévő torlódást jelezheti).

Ábra 1: Példa traceroute kimenet egy Linux rendszeren – néhány hop megjelenítésével. A valós program ennél áttekinthetőbb formában, táblázatosan listázza a hopszámot, a router címét és a késleltetést.

A traceroute modul implementációja során figyelni kellett a párhuzamos működésre is: a mérés ideje alatt a program többi funkciója (pl. a folyamatos ping mérés) is futhat, ezért a traceroute futtatása külön szálon (thread) történik, hogy ne akassza meg a fő UI-t. A modul a mérést követően lehetőséget ad az eredmények naplózására vagy fájlba mentésére is.

3.3 LAN scan (hálózati eszközök feltérképezése)

A LAN (Local Area Network) scan modul a helyi hálózaton található eszközök automatikus felderítését végzi. Ennek a funkciónak köszönhetően a felhasználó egy kattintással lekérheti, hogy a saját alhálózatán (pl. otthoni router mögötti 192.168.x.x tartományban) milyen aktív eszközök vannak (számítógépek, telefonok, nyomtatók, IoT eszközök stb.), és azok milyen IP címen érhetők el. A modul hasonlóan működik, mint

egy egyszerű IP-szkenner vagy a ping alapú hálózati felderítő eszközök: végigpróbálja a lehetséges IP címeket a hálózatban, és megnézi, melyek válaszolnak.

A program induláskor lekérdezi a saját IP címét és alhálózati maszkját, hogy meghatározza a vizsgálandó IP-tartományt. Például ha a gép IP címe *192.168.1.50* / *255.255.255.0* (*/24* hálózat), akkor a modul tudja, hogy a *192.168.1.1–192.168.1.254* címeket kell végigellenőrizni. A felderítés többféle módszerrel történhet:

- **ICMP “ping” alapú szkennelés:** A legegyszerűbb mód, hogy a program sorban kiküld egy ICMP Echo Request csomagot minden egyes IP címre a tartományban, és figyeli, érkezik-e Echo Reply. Java-ban ezt használhatja az `InetAddress.isReachable(timeout)` metódus, amely megpróbálja elérni a megadott IP-t (ICMP vagy egyéb fallback mechanizmus révén) a megadott időkorláton belül[10]. A modul végigiterál 1-től 254-ig (a példában) az IP cím utolsó oktettjén, és amelyik címre pozitív visszajelzést kap, azt elmenti egy listába, mint "aktív host". Az így megtalált IP-khez a program megpróbálja feloldani a hozzá tartozó hosztnevet (reverse DNS), hogy beszédesebb nevet is megjelenítsen (pl. *DESKTOP-ABCD*, *JohnPhone* stb., ha elérhető).

- **TCP port alapú detektálás:** Előfordulhat, hogy bizonyos eszközök nem válaszolnak ICMP pingre (pl. biztonsági okokból tiltják azt), így pusztán pingeléssel nem minden aktív eszköz deríthető fel[11]. A modul ezért kiegészíthető úgy, hogy bizonyos gyakori portokon (pl. 80-as HTTP, 443-as HTTPS, 22-es SSH, 139-es SMB stb.) próbál TCP kapcsolatot nyitni az egyes IP címeken. Ha egy TCP háromlépcsős kézfogás sikerül egy adott címen, az azt jelenti, hogy ott egy eszköz figyel a kérdéses porton, vagyis az IP aktív. Így olyan eszközök is feltérképezhetők, amelyek az ICMP-re nem reagálnak, de pl. van nyitott portjuk. Természetesen ez a módszer kicsit lassabb lehet, hiszen több portot is próbálgatni kell IP-nként, de növeli a felderítés teljességét. A program implementációja lehetővé teszi, hogy a felhasználó beállítsa, mely portokat próbálja a szkenner (alapértelmezetten csak egy kis készletet a leggyakoribbak közül).

- **Párhuzamos (multithread) szkennelés:** Mivel egy teljes alhálózati tartomány (például 254 cím) végigpingelése sorban akár több másodpercig is eltarthat (a timeout beállítás függvényében, jellemzően néhány száz ms címenként), a modul párhuzamosítja a feladatot. Több szálon egyszerre pingeli a címeket – például 20-as csoportokban –, így a teljes folyamat jelentősen felgyorsul. Ügyelni kell azonban arra, hogy túl nagy terhelést se okozzunk a hálózaton, illetve a program se használjon túl sok erőforrást: ezért a szálak számát ésszerű keretek között tartjuk (konfigurálható módon).

A LAN scan eredményeképp a program listában jeleníti meg az aktív eszközöket. Minden sor tartalmazza az eszköz IP címét, opcionálisan a nevét (ha felderíthető), valamint esetleg további információkat: pl. nyitott portok számát, MAC címet (ezt ARP tábla alapján ki lehet olvasni, ha a platform engedi), és egyéb megjegyzést. A modul például kiírhatja:

```
192.168.1.1   – elérhető (hostnév: router.local)
192.168.1.10  – elérhető (hostnév: DESKTOP-1234, nyitott portok: 80)
192.168.1.15  – elérhető (hostnév: android-efgh, nyitott portok: nincs adat)
...
Összesen aktív eszközök: 5
```

A fenti példa alapján látható, hogy a router IP-je beazonosítható, továbbá egy PC (80-as port nyitva, valószínűleg webkiszolgáló fut rajta), egy Android eszköz stb. A program statisztikát is adhat a folyamat végén (pl. hány eszközt talált). A LAN scan modul tehát egyfajta térképet ad a felhasználónak a hálózatáról. Ez nem csupán informatív, de biztonsági szempontból is hasznos: könnyen ellenőrizhető vele, ha esetleg illetéktelen eszköz csatlakozott a Wi-Fi hálózathoz.

Meg kell jegyezni, hogy a hálózati eszközök felderítése bizonyos környezetekben korlátozott lehet. Például egy vállalati hálózatban tűzfalak vagy hálózatbiztonsági beállítások tilthatják az ICMP válaszokat, vagy az aktív portscan tevékenységet. A program ezért konfigurálható óvatosabb módra is, illetve figyelmezteti a felhasználót, hogy engedély nélkül végzett portszkennelés akár jogi vagy etikai kérdéseket is felvethet. Otthoni hálózatban természetesen a saját eszközeink felderítése legitim és hasznos funkció.

A Java megvalósítás során a `java.net.InetAddress` osztály mellett szükség lehet alacsony szintű műveletekre is (például RAW socket az ARP lekérdezéshez, amelyet a Java önmagában nem támogat könnyen, de natív parancsok – pl. `arp -a` – kimenetének feldolgozásával megoldható). A modul ezen részét a platformfüggő különbségek figyelembevételével fejlesztettük: Windows és Linux rendszereken is működő megoldást implementáltunk. Teszteltük továbbá a modult különböző hálózati terhelés mellett, és optimalizáltuk a szkennelés sebességét (például dinamikusan állítja be a párhuzamos szálak számát a rendszer teljesítményétől függően, hogy a futás ne lassítsa le észrevehetően a gépet).

3.4 Port forward (UPnP segítségével)

A port forward modul feladata, hogy az otthoni routereken tipikusan elérhető **UPnP (Universal Plug and Play)** protokollt kihasználva automatikusan porttovábbítási szabályokat hozzon létre. Ennek gyakorlati haszna, hogy a felhasználó egy gombnyomással megnyithat bizonyos portokat a routerén a belső hálózati gépe felé, anélkül hogy manuálisan kellene belépnie a router adminisztrációs felületére és ott beállítania a port forwardingot. Ez különösen például online játékoknál, saját szerver üzemeltetésénél, távoli elérésnél jöhet jól, ahol egy-egy TCP/UDP portot át kell engedni a NAT-olt hálózaton.

Az UPnP egy kényelmi protokoll, amely lehetővé teszi, hogy a hálózat eszközei automatizáltan konfigurálják a routert bizonyos feladatokra. Esetünkben az **IGD (Internet Gateway Device) Profile** az, ami a port forwardingot kezeli. A Network Monitor program a modul indításakor először felderíti, hogy a hálózaton van-e UPnP-képes internetátjáró eszköz (tipikusan a Wi-Fi router). Ezt multicast alapú kereséssel (SSDP protokoll) vagy a meglévő átjáró IP-jére küldött UPnP kérdéssel deríti ki. Ha talál ilyen eszközt, akkor a modul a felhasználó által megadott portszám(ok)ra vonatkozóan

AddPortMapping kéréseket küld a router felé. Például, ha a felhasználó szeretné, hogy a 8080-as TCP portot a saját gépére továbbítsa a router, akkor a modul kiad egy utasítást: "nyisd meg a 8080/TCP portot és irányítsd a belső 192.168.1.50:8080 címre". Sikeres végrehajtás esetén a router létrehozza ezt a szabályt.

Java-ban a UPnP kommunikáció nem része az alapkönyvtárnak, de elérhetők nyílt forráskódú megoldások. A program a **WaifUPnP** nevű könnyűsúlyú Java könyvtárat használja a porttovábbítás kezelésére. Ennek segítségével rendkívül egyszerűen, akár egyetlen sorral megoldható a port megnyitása vagy zárása a routeren[12][13]. Például a `UPnP.openTCP(8080)` hívás hatására a könyvtár automatikusan megkeresi az alapértelmezett átjárót, ellenőrzi, hogy támogatja-e az UPnP-t, majd végrehajtja a kérést a 8080-as TCP port továbbítására. Hasonlóan, a `UPnP.closeTCP(8080)` lezárja a portot. A WaifUPnP könyvtár előnye, hogy kisméretű és egyszerű; hátránya viszont, hogy csak alapvető funkciókat támogat (pl. nem kezeli a komplexebb UPnP felderítési szcenáriókat több átjáró esetén)[14]. Alternatívaként a program használhatná a *Cling* nevű teljes értékű UPnP implementációt is, ami azonban lényegesen összetettebb – a mi esetünkben a WaifUPnP funkcionalitása elegendőnek bizonyult.

A port forward modul felülete lehetővé teszi a felhasználónak, hogy megadja a portszámot, kiválassza a protokollt (TCP vagy UDP), és beállítsa, hogy a portnyitás ideiglenes legyen-e (pl. csak a program futása alatt maradjon nyitva, majd automatikusan zárja le a kilépéskor). A modul a kérések végrehajtása után visszajelzést ad: kiírja, hogy sikeres volt-e a port megnyitása, illetve ha hiba történt (például a router nem támogatja az UPnP-t, vagy a kért port már foglalt), akkor azt jelzi a felhasználó felé.

Fontos megjegyezni, hogy az UPnP használata biztonsági kockázatokat rejthet magában, hiszen egy rosszindulatú program is nyithatna portot a routerünkön rajta keresztül. Emiatt néhány routeren az UPnP funkció le van tiltva vagy szigorúan szabályozva. A Network Monitor program ezért csak a felhasználó kifejezett kérésére hajt végre port forward műveletet, és minden alkalommal figyelmezteti a felhasználót a potenciális kockázatra (például: *"Figyelem: Az UPnP használatával a router automatikus konfigurálását végzi a program. Győződjön meg róla, hogy megbízik a hálózaton futó alkalmazásokban."*).

Ezzel a modullal a hálózatmonitorozó program kilép a puszta megfigyelő szerepből és beavatkozó eszközzé válik: segít a felhasználónak gyorsan *akadálymentessé tenni* bizonyos kommunikációs csatornákat. Például, ha a felhasználó futtat egy játékszervert a gépén, a modul segítségével megnyithatja a szükséges portot a játékos társak számára. A modul használata után érdemes a nem szükséges portokat lezárni; erre a program emlékeztet is, és lehetőséget biztosít a portok listázására, valamint a már nem kellő porttovábbítási szabályok eltávolítására.

3.5 Unicast, Broadcast, Multicast, Anycast csomagvizsgálatok

Ez a funkciócsoport a hálózati kommunikáció különböző címzési módjaival foglalkozik. A *unicast*, *broadcast*, *multicast* és *anycast* fogalmak a hálózati csomagok célbajuttatásának eltérő módozatait jelentik, és a program oktatási illetve elemző jelleggel képes ezen típusú forgalmak vizsgálatára. A modul lényege, hogy bemutassa

és detektálja, milyen típusú címzéssel közlekednek a csomagok, illetve generáljon ilyen csomagokat teszt céljából, és megfigyelje a viselkedésüket.

- **Unicast kommunikáció:** Unicast esetén egy adó egyetlen vevőnek küld üzenetet a hálózaton. Ez a *hagyományos* egy-az-egyhez kommunikáció, például amikor egy kliens egy szerverhez csatlakozik egy weboldal letöltéséhez, az unicast kapcsolat (a csomagok egy forrástól egy meghatározott célhoz mennek). A program szemléltetésképp megmutatja a felhasználónak, hogy amikor egy konkrét IP címre küld pinget vagy bármilyen csomagot, az unicast forgalom – a csomag fejlécében egyetlen cél IP szerepel. Az unicast üzenetküldés privát jellegű kommunikációnak tekinthető, hiszen egy adott címzetthez szól[15] (noha a hálózat műszaki értelemben nem zárja ki, hogy más is belehallgasson, de címzés szempontjából egyedi). A program lehetőséget ad egy unicast csomag küldésére és annak nyomon követésére: például a felhasználó kiválaszthat egy belső hálózati eszközt, majd küldhet neki egy UDP csomagot, és a modul figyeli, hogy a csomag eljut-e oda, illetve visszaérkezik-e válasz. Ezzel demonstrálja az egyedi címzésű forgalmat.

- **Broadcast kommunikáció:** Broadcast esetén egy adó az adott hálózat *összes* csomópontjának üzenetet küld. Például IPv4-ben egy alhálózaton belüli broadcast cím általában a hálózat utolsó címe (pl. 192.168.1.255 /24 hálózatban). Ha ide küldünk egy csomagot, azt a hálózat minden eszköze megkapja. A programban a broadcast vizsgálatához a felhasználó például küldhet egy broadcast pinget a `192.168.1.255` címre. Ekkor elvileg a hálózat összes aktív eszközének válaszolnia kell (amelyek úgy vannak konfigurálva). A modul összegyűjti a beérkező válaszokat és listázza, mely hostoktól jött reply. Ezzel szemléltethető, hogy broadcast esetén egyetlen csomag küldésével több válasz is érkezhet, több géptől. A broadcast kommunikáció olyan, mintha valaki egy szobában mindenkinek kihirdetne egy üzenetet – a hálózat minden tagja hallja[16]. Tipikus használata pl. ARP kéréseknél vagy bizonyos szolgáltatások felfedezésénél van (amikor nem tudjuk a pontos címet, szétküldjük a kérdést mindenkinek). A program kijelzi a broadcast csomag tartalmát, és hogy hány eszköz reagált rá. A felhasználó ebből megértheti a broadcast előnyeit és korlátait: előny, hogy egyszerre többnek szól, hátrány, hogy feleslegesen terheli azokat is, akiknek nem releváns az üzenet.

Ábra 2: Broadcast kommunikáció szemléltetése – egy adó üzenete a hálózat összes többi tagjához eljut. (A sötétkék csomópont a küldő, a zöldek a fogadó állomások.)[16]

- **Multicast kommunikáció:** Multicast esetén az üzenetküldés *egy csoportnak* szól: egy adó több kiválasztott vevőnek küld csomagot, de nem mindenkinek, csak egy előfizetői csoportnak. Technikai megvalósításban ez speciális *multicast IP címek* használatával történik (IPv4-ben pl. 224.0.0.0 – 239.255.255.255 tartomány, IPv6-ban ff00::/8 kezdetű címek). Az ide küldött csomagokat csak azok a gépek kapják meg, amelyek *feliratkoztak* (subscribe) erre a multicast címre. Jó példa erre az IPTV vagy egy online közvetítés: a szerver egy multicast címre sugároz videó adatot, és a kliensek, akik nézik a csatornát, csatlakoznak erre a multicast címre, így megkapják a streamet, de mások nem. A Network

Monitor program a multicast vizsgálathoz tartalmaz egy egyszerű multicast kliens-szerver demonstrációt: a modul létrehozhat egy multicast csoportot (pl. 224.0.0.100 címen) és abba a felhasználó beléptetheti a gépét, majd küldhet teszt üzenetet erre a csoportra. Ha a programot több gépen futtatják ugyanazon a hálózaton, akkor azok a példányok, amelyek szintén csatlakoztak ehhez a multicast csoporthoz, meg fogják kapni az üzenetet, míg a csoporton kívüliek nem. Ezt a modul vizuálisan kiemeli, ezzel demonstrálva a multicast lényegét: egy forrás, több címzett, de kontrolláltan (nem mindenki, csak a csoport tagjai). A modul kiírja a multicast csoport címét, a beérkező üzeneteket és azok küldőit. Hálózati szinten a multicast routing bonyolultabb, hiszen a routereknek el kell juttatniuk a csomagot minden olyan hálózatrészbe, ahol van csoporttag, de a programban ettől eltekinthetünk – a helyi hálózaton belüli multicast a lényeg. Például a program meg tud jeleníteni egy táblázatot arról, hogy mely multicast csoportok aktívak, és hány taggal (ha vannak ilyenek a hálózaton). A modul akár azt is megmutathatja (ha jogosultság engedi) a rendszer multicast routing tábláját.

- **Anycast kommunikáció:** Az anycast egy kevésbé ismert címzési forma, főként IPv6 hálózatokban használatos. Lényege, hogy több, földrajzilag vagy hálózatilag különböző helyen lévő szerver megkaphatja ugyanazt az IP címet, és amikor egy kliens erre a címre csomagot küld, a hálózat automatikusan *egyhez* juttatja el a sok közül – jellemzően a hozzá *legközelebb* lévőhöz. Tehát az anycast kommunikáció *egy az egyhez* módon ér célt, de a *sok potenciális fogadó közül* választódik ki egy (általában a legkisebb késleltetésű vagy topológiailag legközelebbi). Ez olyan, mintha a feladó el szeretne érni egy szolgáltatást, és a hálózat választaná ki a legközelebbi szervert a szolgáltatáshoz. Jó példák az anycastra a DNS gyökér szerverek: több tucatnyi szerver az interneten ugyanazon IP címen érhető el, és mindig a legközelebbi válaszol a kérésre. A programban az anycastot teljes valójában nehéz demonstrálni, mert igényli a hálózati infrastruktúra támogatását. Viszont a modul elmagyarázza a működését interaktív módon: tartalmaz egy szimulációs ábrát, ahol mondjuk három szerver ugyanazzal a címmel van megadva, és a felhasználó láthatja, hogy a küldött csomag a legközelebbihez fut be. Egy szemléletes analógia: *egy vendég egy bulin el szeretne búcsúzni a házigazdáktól, de több házigazda is van szétszóródva – ezért csak a hozzá legközelebb álló házigazdának mond búcsút*, nem mindegyiknek[17]. Így működik az anycast: ahelyett, hogy minden lehetséges címzett megkapná (mint broadcastnál), vagy minden feliratkozott (mint multicastnál), itt csak *egy* kapja meg a csoportból – de hogy melyik, azt a hálózat dinamikusan dönti el (általában útvonal optimalizálás alapján). A Network Monitor program jelzi, ha egy adott kommunikáció anycast címzést használ (pl. egyes IPv6 ping parancsoknál lehet anycast címre küldeni), de mivel a gyakorlati kipróbálás bonyolult (a helyi hálózatok zömében nincs anycast implementáció), inkább elméleti szemléltetés történik.

A modul UI része összefoglalja ezen fogalmak definícióit és lehetőséget ad egyszerű kísérletekre. Például a felhasználó számára előre konfigurált „forgatókönyvek” állnak rendelkezésre: *“Broadcast ping a helyi hálózaton”*, *“Multicast üzenet küldése és*

fogadása” stb., amelyek lefuttathatók és az eredményük megfigyelhető. A program ezzel didaktikus eszközként is funkcionál, segít megérteni a különböző hálózati címzési módok közötti különbségeket. A csomagvizsgálat modul háttérben használja a Java `MulticastSocket` osztályt multicast csoporthoz csatlakozásra, UDP socketeket az üzenetküldéshez, és raw socket helyett a rendszer adta lehetőségeket broadcast címzésre (például a DatagramSocket enged broadcastot, ha engedélyezzük a `socket.setBroadcast(true)` opcióval). A modul gondoskodik arról is, hogy a kísérletek ne okozzanak kárt: például broadcast ping esetén limitálja a csomagok számát, hogy ne árassza el a hálózatot.

Ábra 3: Multicast kommunikáció vázlata – a küldő csomagját csak a csoport tagjai (zölddel jelölve) kapják meg, más állomások nem. A router a multicast csoport címzés alapján továbbítja a csomagot a csoport tagok felé.

3.6 AI modul működése (naplók elemzése, ajánlások generálása)

A Network Monitor egyik leginnovatívabb része az **AI modul**, amely mesterséges intelligencia technikák alkalmazásával értelmezi a program által gyűjtött adatokat, és proaktív javaslatokat ad a hálózat teljesítményének javítására vagy a problémák elhárítására. Ez a modul a minőségbiztosítás egy magasabb szintjét valósítja meg: nemcsak kijelzi a nyers adatokat, hanem *információt* és *döntéstámogatást* nyújt a felhasználó számára.

Az AI modul két fő alfeladatot lát el: **anomáliadetektálás** és **ajánlógenerálás**.

- **Anomáliadetektálás a naplóadatokban:** A program által folyamatosan gyűjtött naplóadatok (sebesség, ping, jitter időszakos minták) hatalmas információforrást jelentenek. A modul első lépése, hogy ezeket az idősorokat elemzi, és észleli, ha valamely mérőszám szokatlanul eltér a normálistól. Ehhez gépi tanulási módszereket alkalmaz. Például bevezet egy *baseline* modellt a ping értékekre: megtanulja, hogy tipikusan mondjuk 20-50 ms között ingadozik a ping napközben, este 19-23h között esetleg felmegy 70-80 ms-ra a terhelés miatt, stb. Ha ettől a megszokott mintától jelentősen eltérés van (pl. kora hajnalban hirtelen 200-300 ms-os pingek jelennek meg tartósan), azt a modul *anomáliaként* azonosítja. Hasonlóan a sávszélességnél: figyeli, hogy a letöltési sebesség tartósan a rendelkezésre álló csomag x%-a alatt van-e, vagy nagy kilengések vannak. Az AI modul azonosíthat mintázatokat is, például *időszakos* teljesítményromlást (minden este 8-kor esik a sebesség), vagy *trendeket* (az elmúlt hetekben lassú romlás látható a feltöltési sebességben). Mindezt statisztikai modellek és szükség esetén gépi tanuló algoritmusok (mint pl. mozgó átlag, regresszió, esetleg neurális háló alapú prediktor) alkalmazásával végzi. A logelemzés AI alkalmazása lehetővé teszi a hatalmas adatmennyiségben olyan minták felfedezését, amit szabad szemmel nehéz észrevenni[18]. A modul folyamatosan tanul: minél több adat gyűlik össze, annál pontosabban képes megkülönböztetni a normális és rendkívüli hálózati viselkedést.

- **Ajánlások generálása:** Ha az AI modul valamilyen releváns eseményt vagy anomáliát észlel, akkor előre definiált tudásbázisa vagy tanított modellje alapján

javaslatokat fogalmaz meg a felhasználó számára. Ezek az ajánlások lehetnek egészen egyszerűek vagy összetettebbek, attól függően, mire utalnak az adatok. Néhány példa a modul által adható ajánlásokra:

- *"A ping értékek átlagának szórása (jitter) az utóbbi 30 percben jelentősen megnőtt. Ez arra utalhat, hogy a Wi-Fi kapcsolat interferenciával terhelt. Javaslat: próbálja meg átállítani a routert egy kevésbé zsúfolt Wi-Fi csatornára, vagy ha lehet, váltson 5 GHz-es sávra a stabilabb kapcsolat érdekében."* – Itt a modul a jitter anomáliát kötötte össze a lehetséges okkal (Wi-Fi interferencia) és tanácsot ad[1].
- *"Az átlagos letöltési sebesség a szerződésben meghatározott érték 50%-a alá csökkent az elmúlt napokban. Javaslat: mérje meg a kapcsolatot a szolgáltató által ajánlott tesztoldalon is, és fontolja meg a szolgáltató ügyfélszolgálatának értesítését az esetleges hálózati hiba kivizsgálása érdekében."* – Itt a modul észleli a tartós alulteljesítést és ügyintézési lépést javasol.
- *"A traceroute eredménye alapján a válaszidők a 3. hopnál hirtelen megugranak (10 ms-ről 150 ms-re). Ez valószínűleg a szolgáltató gerinchálózatának egyik csomópontján kialakult torlódásra utal. Ennek megoldása felhasználói oldalon nem lehetséges, de ha a probléma tartósan fennáll, jelezze a szolgáltatónak az útvonal elemzés eredményével."* – A modul itt a traceroute adatait értelmezte és magyarázta.
- *"Ön jelenleg több eszközzel is nagy forgalmat generál (pl. torrent kliens fut). Emiatt a jitter érték magas a hálózaton. Ajánlás: korlátozza a háttérforgalmat vagy állítson be QoS-t a routeren, hogy az interaktív forgalom (pl. játék, videóhívás) előnyt élvezzen."* – Ez egy komplexebb szituáció, ahol a modul kombinálja az adatokat (nagy sávszélesség-kihasználtság + jitter) és valószínűsíti az okot, majd tanácsot ad.

Az ajánlások generálásához a program rendelkezik egy szabály- és tudásbázissal, amit szakirodalmi források, hálózati ajánlások és akár saját tanulási folyamata alapján állítottunk össze. Például tudja, hogy a magas jitter tipikus okai között van a hálózati torlódás vagy Wi-Fi interferencia; a magas ping lehet DNS-probléma is, stb. Ezeket az *if-then* jellegű szabályokat ötvözi esetleges gépi tanulási modellekkel. A modul a mesterséges intelligencia terén klasszikus megközelítést alkalmaz: *felügyelt tanulást* is használunk ott, ahol volt tanító adat (pl. ismert problémaszituációk és azok jellemző mintázatai), illetve *felügyelet nélküli tanulást* az ismeretlen anomáliák felderítésére (pl. klaszterezés vagy outlier-detekció a log adatokban)[19].

A felhasználó szempontjából az AI modul egy **“Hálózati asszisztensként”** jelenik meg. A program felületén van egy panel vagy külön ablak, ahol az aktuális megállapítások és javaslatok listája látható. Fontos, hogy a modul magyarázatokkal együtt adja az ajánlást (az ún. *“explainable AI”* elvnek megfelelően), azaz ne csak annyit írjon ki, hogy *"Javaslat: Indítsa újra a routert"*, hanem azt is, hogy miért (milyen adat alapján jutott erre). Ennek érdekében minden ajánlathoz tartozik egy magyarázat, amit a modul automatikusan fogalmaz meg. Például: *"A ma 14:00-kor mért ping érték 250 ms fölé nőtt, ami jelentős eltérés a napi átlaghoz képest (45 ms). Ez gyakran orvosolható a router újraindításával, mert előfordulhat, hogy a router memóriaszivárgása vagy egy*

elakadó folyamat okozza a megnövekedett késleltetést." – Itt a miért is meg van indokolva.

Technológiailag a Java nyelv nem feltétlenül elsődleges választás komplex gépi tanulási feladatokra, de a modul keretein belül több megoldás is szóba jött: - Egyszerűbb statisztikai számításokat és szabályokat Java-ban, manuálisan kódolva is meg lehet valósítani (pl. mozgó átlag, küszöbérték-ellenőrzés). - Komplexebb modellekhez integrálható Python komponens (pl. JPython vagy a Java Process segítségével futtatni egy Python scriptet scikit-learn-nel vagy TensorFlow-val). Esetünkben a modul prototípusa integrál egy kis Python szkriptet, ami neurális hálóval próbál előrejelzést adni a sávszélesség alakulására a trendek alapján. A Java a háttérben lefuttatja a Python modult és visszakapja az eredményt, amit aztán megjelenít. - Továbbá léteznek Java-ban is ML könyvtárak (DeepLearning4J, Weka, stb.), amiket be lehet építeni. A mi implementációnk kezdetben Weka-t használta néhány osztályozási feladatra, például egy döntési fa modellel kategorizálta a teljesítményproblémák okait a log alapján (a döntési fa inputjai a mért adatok statisztikái voltak, outputja pedig pl. "valószínű ok: szolgáltatói hiba / helyi Wi-Fi / eszköz túlterhelés").

Az AI modul működését alaposan teszteltük szintetikus adatokkal és valós hálózati helyzetekben is. A modul képes **tanulni** is: a felhasználó visszajelzéseit fel lehet használni a modell finomítására. Például, ha egy ajánlás nem vált be, a felhasználó jelezheti ezt (downvote), míg ha hasznos volt, azt is (upvote). A program ezeket a visszajelzéseket eltárolja, és idővel figyelembe veszi a szabályok súlyozásánál vagy a tanult modell újratanításánál.

Összefoglalva, az AI modul révén a Network Monitor nem csak adatokat szolgáltat, hanem *értelmet is ad azoknak* a felhasználó számára. Mint egy digitális hálózati tanácsadó, segít értelmezni a számokat és proaktívan fenntartani a hálózat optimális állapotát. Ez a fajta funkcionalitás a hasonló hálózatfigyelő eszközök között is kiemeli a programot, és a jövőbeni fejlesztési lehetőségek széles tárházát nyitja meg (például még okosabb, akár önjavító hálózatfelügyelet irányába).

4. Architektúra és moduláris felépítés

A Network Monitor program architektúráját a modularitás és a rétegzett felépítés jellemzi. A tervezés során arra törekedtünk, hogy a különböző funkciók (monitorozás, traceroute, LAN scan, stb.) jól elkülönüljenek egymástól, ugyanakkor együttműködve egy integrált rendszert alkossanak. Az alábbiakban bemutatjuk a program fő komponenseit és azok kapcsolatait:

- **Felhasználói felület (UI) réteg:** Ez a program grafikus kezelőfelületét jelenti (mivel technikai dokumentáció, feltételezzük, hogy a program rendelkezik grafikus UI-val, pl. JavaFX vagy Swing alapokon). A UI réteg felel a felhasználóval való interakcióért: megjeleníti a mért adatokat (valós idejű grafikonok, táblázatok), lehetővé teszi a felhasználó számára a műveletek indítását (pl. "Traceroute indítása", "LAN scan futtatása", "Port megnyitása" gombok), és visszajelzést ad az AI modul ajánlásairól is. A UI réteg nem tartalmaz üzleti logikát, hanem a háttérben működő modulokat hívja meg, tipikusan eseménykezelők (event

handler) segítségével. Például ha a felhasználó rákattint a "Traceroute" gombra és megad egy címet, a UI réteg meghívja a traceroute modult a paraméterrel, majd az eredményt (listát a hopokról) visszakapva megjeleníti egy listában vagy grafikonon.

- **Szolgáltatás (service) réteg / modulok:** Ide sorolhatók a program fő logikai komponensei, amelyek a 3. fejezetben részletesen bemutatott funkciókat megvalósítják. Minden fő funkció modulárisan, saját osztályként vagy csomagként került implementálásra:

- *MonitorService* (vagy MonitorModule): Kezeli a hálózati méréseket (sebességteszt, ping, jitter számítás). Időzítőket (schedulereket) futtat a periodikus méréshez, gyűjti az eredményeket, frissíti a naplófájlt. Nyitott socketeket (pl. a sebességteszthez) és kezeli a forgalom generálását, mérését. Esetleg tartalmaz egy *SpeedTestClient* alkomponenst, ami specifikusan a sávszélesség mérés protokollját (pl. HTTP letöltés, vagy saját mérőszerver) intézi.
- *TracerouteModule*: A traceroute logikáját valósítja meg. Ha belsőleg a rendszer parancsot használja, itt történik a process indítás és output olvasás. Ha tisztán Java-ban, akkor itt van megírva a TTL növeléses mechanizmus. E modul feladata a kapott eredmény struktúrált (pl. egy Hop objektum listája) formában való visszaadása a UI számára.
- *LanScanner*: Ebben van a hálózati eszközök felderítésének logikája. Képes hálózati interfész infó lekérésére (saját IP és netmask), IP tartomány generálására, több szálon pingelések indítására, eredmények gyűjtésére. Esetleg tartalmaz egy *PortScanner* segédobjektumot a TCP portos vizsgálathoz. A végeredményt (aktív hostok listája, metaadatokkal) átadja a UI-nak.
- *UPnPModule* vagy *PortForwardService*: Ez a modul használja a külső UPnP könyvtárat. Gyakorlatilag wrapperként funkcionál a WaifUPnP (vagy más) hívásaihoz. Biztosítja, hogy egyszerre csak engedéllyel fusson, figyel a hibakezelésre (pl. nincs UPnP eszköz). Lehetővé teszi a program számára a portok listázását is (UPnP lekérdezéssel), így a UI-n megjelenhet egy lista az aktív port forward szabályokról.
- *PacketAnalyzerModule*: Ide sorolható az unicast/broadcast/multicast/anycast kísérletek logikája. Például tartalmaz egy *MulticastReceiver* és *MulticastSender* osztályt a csoportos üzenetküldés kezelésére. A broadcast küldés is itt van implementálva (DatagramSocket + broadcast flag), és figyelheti a bejövő broadcast válaszokat (ehhez a hálózati stack jellemzően engedi a programnak, hogy pl. ICMP echo reply-ket is figyeljen, de Java-ban ezt a RawSocket hiánya miatt nehézkes – esetleg úgy oldottuk meg, hogy a ping parancs kimenetét figyeljük broadcast esetén is, vagy egy natív pcap könyvtárat integrálunk). Az anycast itt főleg elméleti, de ha IPv6 környezet van, a modul tesztelheti egy anycast cím pingelését és megnézni, mi a válasz (Linux alatt pl. a `ping -6 -c 1 -t 1 <anycast_address>` adhat valamit).

Minden modul igyekszik **függetlenül** működni, és csak a szükséges adatokat megosztani a többiekkel. A modulok között a kommunikációt és integrációt a *Controller*

(lásd következő pont) végzi. Ez biztosítja, hogy például a Monitor modul által mért adatok menjenek az AI modul felé is, vagy a LAN scanner eredménye bekerüljön a log fájlba is (ha szükséges).

- **Kontroller réteg:** A programban egy központi vezérlő komponens (gyakorlatilag a fő programlogika) hangolja össze a modulok működését. Ez tekinthető az alkalmazás "backendjének". Feladatai:
- Indítja és leállítja a modulokat a program lifecycle-je szerint. Például program induláskor elindítja a Monitor modul időzített mérését, leállításkor leállítja azt.
- Eseményeket továbbít a modulok között: pl. ha a felhasználó a UI-on elindít egy traceroute-ot, a kontroller hívja a Traceroute modult és várja az eredményt, majd továbbítja a UI-nak. Vagy ha a Monitor modul új adatot naplózott, értesítheti az AI modult (vagy az AI modul pollinggal lekéri időnként az új adatokat).
- Erőforrások kezelése: a kontroller figyel arra, hogy párhuzamos modulok ne zavarják egymást. Például egyszerre ne fusson 50 szál a monitor+LAN scan kombináció miatt; ha a CPU terhelés magas, ütemezheti a feladatokat.
- Konfiguráció menedzsment: a modulok közös konfigurációs adatait (pl. aktuális felhasználói beállítások: mérési intervallum, engedélyezett funkciók, stb.) a kontroller tartja és adja át a moduloknak indításkor.
- Naplózás: a program globális naplózási mechanizmusa (nem az adatnapló, hanem magának az alkalmazásnak a logja hibákról, státuszokról) is itt fut. Ha egy modulban hiba történik (pl. nem sikerül portot nyitni UPnP-n), a modul jelez a kontroller felé, ami logolja és a UI felé is továbbítja az infót.

- **AI modul (backend):** Bár az AI modulról már tartalmilag volt szó (3.6), architekturálisan megjegyezzük, hogy ez lényegében egy különálló alrendszer. Lehet külön szálon futó folyamat, amely folyamatosan figyeli a bejövő adatokat. Az AI modulnak van egy modellje (vagy több modellje) betöltve, és van egy *Inference Engine*-je, amely a szabályokat és a tanult mintákat alkalmazza az új adatokra. Az architektúra szempontjából fontos, hogy az AI modul viszonylag lazán csatolt: ha valamiért ez a modul leáll vagy hibázik, az alapvető monitorozó funkciók akkor is működnek tovább (csak épp nem lesznek ajánlások). Ezt úgy értük el, hogy az AI modul hívásai aszinkron módon történnek, és eredményeit a kontroller kezeli opciósan. Például egy külön belső eseménykezelő van: *AIEventHandler*, ami figyeli, hogy érkezett-e új ajánlás, s azt továbbítja a UI felé.

- **Adatkezelés és tárolás:** A program moduljai közösen használhatnak bizonyos adatelérési réteget. Itt két fő dolog van: a *mérési adatok naplója* és a *konfigurációk*. A mérési napló egy fájlban vagy adatbázisban tárolódik. Egyszerű megoldásként a Java beépített fájlkezelésével CSV formátumban naplózunk (pl. `network_log.csv`, melynek egy részlete a feladatleírás alapján elérhető is). Ezt a fájlt a Monitor modul írja, de az AI modul olvassa. Emiatt létrehoztunk egy *LogManager* komponenst, ami kezeli a fájl írását és olvasását, hogy ne legyen ütközés. A modulok ezen a LogManageren keresztül érik el a naplóadatokat thread-safe módon. A konfigurációk (pl. user beállítások: mérési intervallum, stb.) szintén egy külön osztályban, esetleg `config.properties` fájlban vannak, melyet

egy *ConfigManager* tölt be induláskor és ad át a moduloknak. Az architektúra célja ezzel az elkülönítéssel az volt, hogy könnyen cserélhető legyen az adatkezelés (pl. átálljunk CSV-ről SQL adatbázisra anélkül, hogy a modulok kódját módosítani kellene – elég a LogManager implementációját kicserélni).

Összességében a program felépítése követi az MVC (Model-View-Controller) mintát bizonyos értelemben: - View = UI (megjelenítés) - Controller = kontroller logika, amely koordinál - Model = itt tágabb értelemben a modulok üzleti logikája + adatréteg (mérési adatok)

Ezen túlmenően moduláris felépítés azt is jelenti, hogy a program könnyen bővíthető új funkciókkal. Például, ha a jövőben szeretnénk egy *VPN monitor* modult (ami figyeli a VPN kapcsolatot és újramonitoroz, ha szakadás van), azt a meglévő struktúrába be lehet illeszteni egy új Service modul formájában, a UI-hoz hozzáadva a vezérlő elemeket. A modulok közötti interfészek jól definiáltak – többnyire egyszerű Java interface-ek formájában –, így például a Monitor modul kicserélhető egy másik implementációra (pl. ha valaki magasabb pontosságú natív mérőkomponenst ír), a program többi része változatlan maradhat.

A moduláris architektúra további előnye a **hibahatárok** biztosítása: egy-egy modul hibája nem rántja magával az egész programot. Ezt részben külön szálakon futtatással, részben pedig megfelelő kivételkezeléssel oldottuk meg. Például, ha a traceroute modul futása során váratlanul egy formátumhibás kimenet feldolgozásakor kivétel keletkezik, azt a modul elkapja, jelzi a kontrollernél, de a fő program nem omlik össze, csak a traceroute folyamat leáll és a UI egy hibaüzenetet kap (“Traceroute sikertelen”). A többi modul (pl. folyamatos ping) ettől még gond nélkül megy tovább. Így a program megbízhatósága nő.

Végül, az architektúra tervezésénél figyelembe vettük a **skálázhatóságot** is: bár a program alapvetően kliensoldali, egyszemélyes alkalmazásnak készült, igyekeztünk úgy kialakítani, hogy akár sok adatot (hosszú távú log) is kezelni tudjon. Ezért választottuk a fájl alapú naplózást gyűjtőnek, de úgy, hogy az könnyen átirányítható legyen mondjuk egy szerveres tárolóra. A modulok különválasztása azt is lehetővé teszi, hogy a jövőben akár *kliens-szerver architektúrává* fejlesszük az alkalmazást (kliens csak UI, szerver végzi a mérést és tárol mindent), anélkül, hogy a mérési logikát újra kellene írni.

Architekturális diagram (vázlatosan):

```
[UI réteg] <--> [Kontroller] <--> [Monitor modul]
                             \--> [Traceroute modul]
                             \--> [LAN Scan modul]
                             \--> [UPnP modul]
                             \--> [Packet Analyzer modul]
                             \--> [AI modul]
                             \--> [Log/Config Manager (adatkezelés)]
```

A fenti diagram mutatja, hogy a kontroller központi szerepet játszik, a modulok pedig csillag topológiában kapcsolódnak hozzá. A modulok egymással közvetlenül nem nagyon kommunikálnak, inkább a kontrolleren keresztül (pl. a Monitor modul nem hívja

közvetlenül az AI modult, hanem a kontroller értesíti az AI-t az új adatról, vagy az AI modul kéri le az adatot a log manageren át). Ez a laza csatolás segít abban is, hogy egy modul karbantartása, fejlesztése elkülönülten történhessen.

Összegzésként, az architektúra megfelelően **moduláris, rétegzett és kiterjeszthető**. Ez nem csak a fejlesztés átláthatósága miatt volt fontos, hanem a minőségbiztosítás és esetleges hatósági audit szempontjából is (lásd később): könnyebb igazolni egy-egy modul helyes működését és megfelelőségét, ha tisztán el van különítve a többi funkciótól.

5. A program működésének elméleti háttere (protokollok, Java API-k)

Ebben a fejezetben áttekintjük azokat az elméleti ismereteket és technológiákat, amelyekre a Network Monitor program épül. Ide tartoznak a hálózati protokollok (pl. IP, ICMP, TCP, UDP, DNS, UPnP stb.), valamint a programozási nyelv (Java) azon könyvtárai és API-jai, amelyek segítségével a fenti funkciókat megvalósítottuk.

5.1 Hálózati protokollok és standardok

- **IP (Internet Protocol):** A program minden hálózati tevékenysége alapjaiban az IP protokollra épül, legyen szó IPv4-ről vagy IPv6-ról. Az IP felel a csomagok továbbításáért a hálózatokon keresztül. A Network Monitor esetében az IP címzések kezelésénél fontos megemlíteni az alhálózat fogalmát (LAN scan modul) – a program számításba veszi a nettómaszkot, és generálja a lehetséges IP címeket egy tartományban. A program nem módosítja az IP réteg működését, csak kihasználja azt; ugyanakkor pl. a traceroute modul működésének megértéséhez szükséges ismerni, hogy mi az a TTL mező az IP csomag fejlében, és hogyan működik a routing. Az IPv4 broadcast címeket a program automatikusan kiszámítja (például úgy, hogy az IP címet bitenként OR-olja a maszk bitinvertáltjával)[20]. Az IPv6 esetén a program kezeli a különböző címzési formákat (pl. link-local címek fe80::, global unicast, stb.), de broadcast ott nincs – ezt is figyelembe vesszük a moduloknál. A program kompatibilis IPv6-tal olyan formában, hogy ahol lehet a Java beépített mechanizmusai automatikusan választanak IPv6-ot (pl. InetAddress.getByName preferálhat IPv6-ot, ha olyan hostnevet adunk meg). Az IP protokoll szabványát az **RFC 791** írja le (1981), ezt hivatkozásként figyelembe vettük, amikor például a TTL működését implementáltuk.

- **ICMP (Internet Control Message Protocol):** Az ICMP protokollt használja a program ping és traceroute funkciója során. Az ICMP az IP protokoll része, azt kiegészítő üzeneteket határoz meg (pl. Echo Request/Reply a pinghez, vagy Time Exceeded a traceroute-hoz). Az ICMP protokollt az **RFC 792** definiálja[21]. A Network Monitor nem nyúl közvetlenül az ICMP csomagok generálásához (mert Java-ban nincs magas szintű ICMP socket API, és raw socket nyitásához rendszergazdai jogosultság kellene), hanem a `InetAddress.isReachable()` függvényt hívja, ami Java implementációtól függően belsőleg próbál ICMP echo-t küldeni vagy fallback megoldást alkalmaz. Mint ismeretes, a Java `isReachable`

metódus Windows alatt sokáig csak TCP echo (port 7) próbát tett[22], ami nem mindig megbízható, de az évek során javult a helyzet. A program a ping mérésnél ezért inkább a natív ping parancs hívását preferálja a pontosság kedvéért, de a lényeg, hogy mindkét esetben ICMP Echo Request/Reply mechanizmus zajlik a háttérben. Az ICMP üzenetekre vonatkozóan a program figyeli a visszatérő kódokat is (pl. ha a pingelés célja host unreachable, akkor ICMP Destination Unreachable jön, ezt a natív ping is jelzi a kimenetben, és a program kiírja, hogy a host nem elérhető). Az ICMP fontossága a program számára abban is rejlik, hogy a traceroute modul a Time Exceeded üzenetekre hagyatkozik – ezt is feldolgozzuk. Itt a standard szerint az ICMP Time Exceeded üzenet tartalmazza az eredeti csomag fejrészét, hogy tudjuk, melyik csomagra érkezett (de a mi programunk ezt nem bontja ki, a natív traceroute végzi).

- **UDP (User Datagram Protocol):** Az UDP protokoll egy egyszerű, kapcsolatmentes szállítási protokoll, melyet a program különféle részei is alkalmaznak: például a multicast és broadcast küldésekhez UDP socketeket használunk, illetve a LAN scan modul bizonyos portpróbái is UDP-n történhetnek (pl. küldhetünk egy üres UDP csomagot egy portra, hátha válasz jön – bár UDP-n nincs beépített válasz mechanizmus, így ezt inkább nem használjuk sokat). Az UDP-t az **RFC 768** írja le (1980). A Java-ban a `DatagramSocket` és `DatagramPacket` osztályok felelnek az UDP kommunikációért. A Network Monitor program az UDP socketek esetében figyelt arra, hogy a broadcast címre való küldést engedélyezni kell (`DatagramSocket.setBroadcast(true)`), különben a rendszer nem engedi ki a broadcast csomagot. A multicast esetén a `MulticastSocket` külön osztályt használunk, amellyel csatlakozni lehet multicast csoporthoz és azon keresztül fogadni/küldeni. Az UDP előnye a programban a kis overhead és az, hogy könnyű vele saját protokollt kialakítani a méréshez (pl. sebességteszthez is lehetne UDP-t használni egy szerverrel párosítva, bár mi HTTP-t használunk inkább).

- **TCP (Transmission Control Protocol):** A TCP protokollra épül a legtöbb internetes adatforgalom, és programunk is használja néhány helyen. Egyrészt a sebességteszt modul letöltési mérése tipikusan HTTP protokollon (ami TCP-alapú) keresztül történik, így közvetve a program a TCP kapcsolaton át méri a sávszélességet. Másrészt a LAN scanner modul portscan funkciója nyitogat TCP kapcsolatokat a megadott portokra a reachability teszteléshez. A TCP megbízhatósága és kapcsolat-orientáltsága miatt a pingeléssel ellentétes filozófiát képvisel: ha egy TCP port nyitva van és kapcsolatot tudunk létesíteni, akkor biztosan van ott eszköz; ha zárva van, az is információ; ha pedig filtered (szűrt), a modul azt is észleli (timeout). A Java-ban a `java.net.Socket` osztály használatával történik a TCP kapcsolatok létrehozása. Ha a socket.connect sikeres, akkor az adott IP:port elérhető. Ha kivételt dob (Connection refused), akkor a port létezik de nincs szolgáltatás (az eszköz elérhető, de port zárt); ha timeout-ol, akkor valószínűleg tűzfal blokkolja. Ezeket a modul logikája értelmezi és jelenti a felhasználónak. A TCP protokollt az **RFC 793** definiálja, de közvetlenül nem kellett implementálnunk semmit belőle, a Java runtime intézi. Fontos még megjegyezni, hogy a traceroute modul néhány implementációja (amit

nem használtunk végül) úgy is működik, hogy UDP helyett TCP csomagokat küld növekvő TTL-lel bizonyos nyitókézfogási kísérletekkel, vagy modern traceroute esetén akár TCP SYN csomagokat is lehet használni (főleg tűzfalak kikerülésére). Mi a klasszikus ICMP/UDP alapú módszert követtük.

- **DNS (Domain Name System):** Bár a DNS nem került külön funkcióként említésre, a program működése során implicit módon támaszkodik rá. Például amikor a felhasználó beír egy hostnevet a traceroute modulnak vagy ping modulnak (pl. "www.example.com"), akkor a program a Java name resolution mechanizmusán keresztül először DNS lekérdezést végez a név feloldására IP-cím(ek)re. Ezt a Java automatikusan végzi az `InetAddress.getByName()` meghívásakor. A program számára fontos volt, hogy legyen timeout kezelve a DNS feloldásra, nehogy pl. egy nem létező domain esetén hosszú ideig akadjon. A DNS működés elve (elosztott hierarchikus névfeloldás) csak áttételesen érinti a programot, de a modulok felhasználói élményét befolyásolja: például LAN scan esetén a reverse DNS feloldás a találatok neveire. Ehhez a Java `InetAddress.getHostName()` metódust használtuk, ami megpróbálja az IP-címet hostnévvé alakítani a DNS vagy lokális host fájl alapján. Ez néha lassú lehet, ezért beállítottuk, hogy ez a lépés opcionális vagy párhuzamos legyen, ne blokkolja a többi folyamatot. A DNS protokoll lényegét az **RFC 1034** és **1035** írja le, de implementáció szintjén a program a Java és az OS beépített resolverét használja.

- **UPnP (Universal Plug and Play) és IGD:** Ezt a protokollt a 3.4 fejezetben részleteztük a port forward kapcsán. Az UPnP egy alkalmazás-szintű protokollkészlet, ami SOAP üzenetekkel kommunikál HTTP-n keresztül a lokális hálózaton. A lényeg, hogy a programnak meg sem kellett ismernie a SOAP részleteit, mert a használt könyvtár elrejtette ezt. Fontos azonban, hogy a program tudja, hogyan keressen UPnP eszközöket: erre SSDP protokollt használ (a WaifUPnP belsőleg küld egy HTTPu multicast kérést a 239.255.255.250:1900 címre, ami az UPnP alap keresési címe). A kitérő válaszokat a könyvtár feldolgozza. Az UPnP IGD port mapping szabványát az **IGD Standardized Device Control Protocol** írja le (ez nem egy IETF RFC, hanem az UPnP Forum szabványa, de a lényeges hívások: AddPortMapping, DeletePortMapping). A Network Monitor annyit csinál, hogy a könyvtár függvényeit meghívja, így a protokoll mély ismerete nem került implementálásra, de a doksiban utalunk a standardra. Megemlítendő, hogy a NAT-PMP vagy PCP protokollok modern alternatívák a UPnP-re (Apple ill. IETF által), de ezekkel a program jelenleg nem foglalkozik.

- **Egyéb protokollok:** A program érintőlegesen még néhány protokollal találkozik: pl. ARP (Address Resolution Protocol) a LAN-on belüli kommunikációhoz. Az ARP felel azért, hogy egy IP címhez MAC címet társítson a gép, mielőtt a helyi hálóra küld adatot. A LAN scan modulban, amikor egy IP válaszol a pingre, a program lekérdezheti a MAC címét (pl. egy `arp -a` hívással), így kideríthető a gyártó (MAC prefix alapján). Az ARP protokoll maga nem közvetlenül került implementálásra (mert a Java nem ad ARP API-t), de a működése fontos háttere

a broadcast pingnek is: a broadcast ping valójában IP szinten broadcast, de Ethernet szinten is broadcast, és ARP nélkül nem is tudnánk kiküldeni, hiszen a broadcast IP-hez tartozó MAC cím a ff:ff:ff:ff:ff:ff (mindenki címe). Ezekre figyeltünk a modul fejlesztésénél. Emellett a program outputjaiban megjelenik a TTL érték (pl. ping válasznál), ami érdekességképp utal a távoli rendszer OS-ére (más TTL default Linux/Windows). Ezek apró részletek, de a protokollok ismeretét feltételezték a fejlesztéskor.

- **RFC-k és standardok betartása:** A program igyekezett a mérési módszereknél standard definíciókat alkalmazni. Például a jitterre a definíciót a *Packet Delay Variation (PDV)* alapján értelmeztük, ami egy IETF IPPM (IP Performance Metrics) munkacsoport által definiált fogalom (pl. **RFC 3393** dokumentum)[7]. Ahol releváns, a program dokumentációjában hivatkozunk az adott szabványra: pl. a ping az ICMP Echo, melyet az RFC 792 definiál, a traceroute TTL trükkje pedig a hálózati diagnosztika de-facto standardja (bár maga a TTL alapú traceroute nincs szabványosítva teljesen – az **RFC 1393** kísérleti jelleggel definiált egy opcionális módszert traceroute opcióval, de a legtöbb rendszer a TTL csökkentéses módszert használja, ami nincs formálisan RFC-ben, inkább bevett gyakorlat[23]). Ezt azért fontos kiemelni, mert a programot esetleg hatósági jóváhagyásra is szánjuk, ahol elvárás, hogy a mérések megfeleljenek a nemzetközi szabványoknak.

5.2 Java platform és felhasznált API-k

A Network Monitor fejlesztése Java nyelven történt, ami számos előnyt biztosított: hordozhatóság, gazdag könyvtárkészlet hálózati funkciókhoz, és objektum-orientált modularitás. Itt áttekintjük, milyen főbb Java API-kat és technológiákat alkalmaztunk:

- **java.net csomag:** Ez a standard Java könyvtár tartalmazza a hálózati programozáshoz szükséges osztályokat. Itt használtuk:
- `InetAddress`: IP címeket és névfeloldást kezelő osztály. Főleg `getByName()` (DNS lookup) és `isReachable()` (ping-szerű) metódusait használtuk. Továbbá `InetAddress.getLocalHost()` a saját gép IP címének lekérésére, és `getAddress()` a nyers byte cím megszerzéséhez (amit a LAN scan-nél a subnet kitalálásához használtunk[24]).
- `NetworkInterface`: a hálózati interfészek kezelésére (pl. kilistázni, hogy ethernet vagy Wi-Fi interfész IP-vel). A program a multicastnál használja, mert meg kell adni, melyik interfészen lépjen be a csoportba. Illetve ezzel lehet lekérni a netmaskot is közvetve (Java 8-tól van pl. `getInterfaceAddresses()` ami tartalmazza a prefix length-et).
- `Socket` és `ServerSocket`: TCP kliens és szerver socketek. A program kliens oldali Socketeket használ a port scanninghoz és a sebességteszt HTTP letöltéshez. SzerverSocket-et nem igazán használ, mert nem futtatunk tartós szervert (kivéve belsőleg, ha a speedtest egy lokális beépített szervert csinál – de azt inkább egyszerűség kedvéért nem tettük, inkább egy nyilvános tesztURL-t használ).
- `DatagramSocket` és `DatagramPacket`: UDP kommunikációhoz. A broadcast ping modul használja: pl. küld egy üres UDP csomagot a broadcast címre egy

bizonyos porttal, hátha valaki válaszol (ez nem túl megbízható, ezért inkább ICMP-t használunk). A MulticastSocket, ami a `java.net.MulticastSocket` külön osztály, az UDP-hez hasonló, de van `joinGroup(InetAddress group)` metódusa, amivel belép egy csoportba. Ezt a multicast modul használja egy választott csoportcímre.

- `URLConnection` / `HttpURLConnection`: a sebességteszt implementációnál a program egy HTTP GET kéréssel tölt le egy ismert méretű fájlt. Ehhez a Java beépített URL kezelő mechanizmusát használja. Pl. `URL url = new URL("http://speedtest.example.com/testfile.bin"); HttpURLConnection conn = (HttpURLConnection) url.openConnection();` majd `conn.getInputStream()`-en olvassa a bájtokat és méri az időt. Ez egyszerű módja a letöltési sebesség mérésének, bár pontosságát befolyásolja a Java stream bufferezése és a garbage collector – ennek hatását teszteléssel minimalizáltuk. Letöltésnél figyeltünk, hogy a filet ne írjuk ki lemezre, hanem /dev/null-szerűen dobjuk el (csak számoljuk a bájtokat), így a lemez IO nem befolyásolja a mérést.
- `Authenticator` osztály: Ha a program olyan környezetben fut, ahol proxy vagy hitelesítés kellene a net eléréshez, ezzel lehet megoldani. Ezt marginálisan érinti csak, de a Java URLConnection figyelembe veszi az `http.proxyHost` beállításokat. A doksi kedvéért: a program lehetőséget ad proxy beállítására, ha valaki pl. céges hálón futtatja (ez a config-ban).

- **Java Threads (java.lang.Thread, Executors):** Mivel a program egyidejűleg sok feladatot végez (időzített mérés, UI interakciók, hálózati várakozások), erősen támaszkodtunk a Java több szálú programozási képességeire. A modulok mind külön szálakon futnak: pl. a Monitor modul egy ScheduledExecutorService segítségével fix rate szerint futtatja a ping/teszt feladatot. A LAN scan modul egy ThreadPoolExecutor-t használ x darab szállal a pingek párhuzamos futtatására[10]. Az AI modul is külön thread, ami sleep-el és időnként elemez. A Java Concurrent API (java.util.concurrent) segített pl. a thread pool kezelésben, a jövőbeni feladatok ütemezésében (`ScheduledExecutorService.scheduleAtFixedRate`). Arra is ügyeltünk, hogy a Swing (vagy JavaFX) UI-nak a saját EDT (Event Dispatch Thread) szála van, így a háttérszálak eredményeit a UI frissítésekor át kell adni az EDT-nek (pl. SwingUtilities.invokeLater-rel) – különben thread safety problémák lennének. Ez tipikus Java kliens alkalmazás tervezési részlet, amit betartottunk a stabilitás érdekében.

- **JavaFX / Swing UI komponensek:** A felhasználói felület elkészítéséhez egy modern JavaFX keretrendszert használtunk, ami jól támogat grafikont, táblázatot és aszinkron frissítést. (Alternatívaként Swing is használható, de JavaFX modernebb, pl. van beépített Chart library). A program UI része tartalmaz:

- LineChart komponenseket a sebesség/ping időbeli alakulásának megjelenítésére.
- TableView komponenseket, pl. a LAN scan eredmények listázására táblázatban (IP, név, státusz oszlopokkal).
- TextArea vagy ListView az AI modul ajánlásainak és log üzeneteinek listázására.

- Form elemek (TextField, Button) a felhasználói inputokhoz (pl. traceroute cím beírása, portszám megadása).
- ProgressIndicator vagy ProgressBar a folyamatok jelzésére (pl. LAN scan folyamatban van, progress bar fut).

A JavaFX-ben sok mindent FXML-lel építettünk meg (UI layout), a vezérlők a Controller osztályok. A modulokkal a UI a kontrolleren át kommunikál, pl. megnyom egy gombot -> Controller hívja a modult -> modul visszaad -> Controller update-el -> UI komponens frissít. Ezt a mintát követtük.

- **Külső könyvtárak:**
- *WaifUPnP* (már említve): egy jar fájlt beépítettünk, melynek osztályait (pl. `com.dosse.upnp.UPnP`) használjuk port nyitásra. Ehhez a jar-hoz tartozó javadocot is átnéztük hogy tudjuk a limitációit[14].
- *Weka* (opcionálisan): az AI modulnál a Weka machine learning könyvtár segítségével készítettünk egy döntési fa modellt. A Weka .jar importálva van, és a modellt off-line betanítottuk néhány tipikus esetre (pl. magas ping + packet loss -> "ISP issue", magas ping + high bandwidth usage -> "local congestion", stb.). Futásidőben a Weka DecisionTree osztályát használva betöltjük a modellt és apply-oljuk a friss adatokra. Ez haladó funkció, ki/be kapcsolható, mivel a Weka jar eléggé megnöveli az alkalmazás méretét, és a modul így is működik rule-based alapon is.
- *pcap4j* (felmerült lehetőségként): Ha raw packet capture kellett volna, a pcap4j library jöhetett volna szóba. Ezzel Java-ból is el lehet érni a WinPcap/Libpcap driver felületét. A program esetleg használhatja a jövőben, ha pl. valós forgalmat akar sniffelni (most nem teszi). De a pcap4j függőséget nem vettük bele alapértelmezetten, hogy ne kelljen natív libeket mellékelni.

- **Platform specifikus megoldások:**

- Windows vs Linux: a program Java-ban íródott, ami platformfüggetlen, de pár helyen a rendszerparancsok hívása platformfüggő. Ezt runtime módon kezeltük: pl. traceroute esetén `if (windows) use "tracert", else "traceroute"` parancs; pingnél Windowsnál a paraméterek mások (-n szám, Linuxnál -c). A Java `System.getProperty("os.name")` alapján döntöttünk.
- Karakterkódolás: Magyar ékezetek (pl. a UI szövegek) miatt figyelni kellett, hogy a forrás UTF-8 kódolású legyen. A dokumentáció szerint is fontos az "ékezetbiztos" kimenet – mi a programban minden stringet explicit UTF-8-ként kezelünk (Java belül UTF-16-ot használ, de a fájlkiírásnál FileWriter-nél beállítottunk UTF-8 encodingot, hogy pl. a log fájl is jól olvasható legyen ékezetekkel).
- Idő és időzóna: A naplóban ISO 8601 formátumot használunk (pl. `2025-11-26T20:30:00+01:00`), így egyértelmű és nem lokális függő. Java-ban a `java.time` API-val formázzuk.

- **Memória és teljesítmény megfontolások:** Java-ban a Garbage Collector (GC) gondoskodik a memóriakezelésről, de egy folyamatosan futó monitor programnál fontos, hogy ne szivárogjon memória és ne is terhelje túl a GC. Ezért:

- Nagy mennyiségű adat gyűjtés: a grafikonhoz nem tárolunk végtelen sok pontot memóriában – pl. a UI csak az utolsó 1 óra adatait mutatja, a régebbit törli a grafikon sorozatból, miután elmentődött fájlba.
- Naplófájl írás: batch-elve, nem karakterenként. Pl. egy belső bufferbe gyűjtjük és X bejegyzésenként írjuk ki, hogy csökkentsük az IO műveletek számát.
- Szálak: a thread pool-okat úgy állítjuk be, hogy ne nyisson túl sok szálat (pl. LAN scan maximum 50 thread, mert ha /16-os hálót kellene pásztázni, az 65k cím, ott inkább iterál többször 50-es csoportokban).
- Kivételek kezelése: kerültük a kivételes esetek folyamatos dobálását a mérési ciklusokban, mert az lassíthat. Inkább előzetesen ellenőrzünk (pl. ha egy URL nem elérhető, nem próbáljuk percenként és dobjunk kivételt, hanem jelzünk és váltunk másik szerverre).
- A programot profilerrel is futtattuk (VisualVM), hogy megnézzük, nincs-e bottleneck. Kiderült, hogy a legtöbb idő IO várakozással megy el (pl. ping parancs futása), ami normális. A CPU terhelés alacsony volt, így a Java overhead nem gond.

Összességében a Java lehetővé tette, hogy a programot viszonylag gyorsan, megbízhatóan fejlesszük, sok kész funkcionalitást újrahasznosítva. Az elméleti háttér ismerete – a protokollok működéséé – azonban elengedhetetlen volt ahhoz, hogy a megfelelő API-kat helyesen használjuk és értelmezzük az eredményeket. A program sikeresen kombinálja a magas szintű nyelvi eszközöket (pl. HTTP letöltés, UI grafikon) az alacsony szintű hálózati jelenségek megfigyelésével (ICMP üzenetek, csomagszórás), így mindkét terület tudását integrálnia kellett. A fejlesztés során folyamatosan konzultáltunk a hivatalos Java dokumentációval (pl. *The Java Tutorials: Networking* fejezet[25], és a JavaDoc referenciákkal a java.net csomaghoz), valamint a vonatkozó RFC-kel és egyéb szakirodalommal, hogy a megvalósításunk megfeleljen a szabványoknak és bevált gyakorlatoknak.

6. Minőségbiztosítás és hatósági rendszeresíthetőség

A Network Monitor program fejlesztése során kiemelt figyelmet fordítottunk a minőségbiztosításra – azaz arra, hogy az alkalmazás megbízható, pontos és hiteles eredményeket szolgáltasson. Emellett, tekintettel arra, hogy a program akár hatósági mérőrendszer részeként vagy ajánlott eszközként is szóba jöhet (például a Nemzeti Média- és Hírközlési Hatóság által végzett mérések kiegészítéseként), áttekintjük, milyen lépéseket tettünk a *hatósági rendszeresíthetőség* irányába.

6.1 Minőségbiztosítási szempontok

- **Funkcionális tesztelés:** Minden modul esetében alapos manuális és automatizált tesztelést végeztünk. Készültek *egységtesztek* (JUnit keretrendszerrel) a kritikus algoritmusokra – például a jitter számítás helyességét ellenőriztük ismert bemeneti értékekkel (egy sorozat ping értékre megvan a kézi

számított jitter, és összehasonlítjuk a program által számolttal). Teszteltük a LAN scan modult is egy szimulált környezetben: írtunk egy kis *fake* hálózati réteget, ahol bizonyos IP címek "válaszoltak" pingre (ezt Mock objektumokkal oldottuk meg), és megnéztük, hogy a modul felismeri-e pontosan ezeket. A traceroute modulnál nehezebb az egységteszt, de itt is próbáltuk szimulálni a `tracert` parancs kimenetét különböző esetekre (normál útvonal, unreachable hop, stb.) és ellenőriztük a parser logikát. Ezen felül *integrációs teszteket* is végeztünk: valós hálózaton futtatva a programot, összevetettük a kapott eredményeket más bevett eszközök eredményeivel. Például a ping mérésnél a program logját összehasonlítottuk a rendszer ping parancsának kimenetével (hasonló paraméterekkel), és ellenőriztük, hogy a program által mért átlag, minimum, maximum ping egyezik-e a ping parancséval adott tűréshatáron belül. Ugyanígy, a sávszélességmérésnél a program outputját összevetettük a speedtest.net eredményével több alkalommal. Tapasztalat: a program ±5% pontossággal adta a letöltési sebességet a speedtest.net-hez képest, ami megfelelőnek mondható. A jitter esetén ±1-2 ms eltérés lehet különböző mérési módszerek miatt, de a nagyságrendi következtetésekre nincs hatással.

- **Teljesítmény és stabilitás teszt:** Hosszú távú futási tesztnek alávetettük az alkalmazást. Több napon keresztül futtattuk folyamatos monitorozó módban, hogy kiderüljön, van-e memória szivárgás vagy stabilitási probléma. A program ~48 órás folyamatos futás alatt is stabil maradt, memóriahasználata 150MB körül stabilizálódott a Java VM-nek, ami elfogadható. Ekkor persze a log fájl mérete nőtt folyamatosan; ellenőriztük, hogy a logrotáció működik-e (beállítottunk egy 5 MB-os limitet, ami után új log fájlt kezd és a régit arhiválja). A modulok újraindíthatóságát is teszteltük (pl. traceroute modul folyamatos indítás/leállítása, port nyitása majd zárása 100-szor ismételve) – nem tapasztaltunk erőforrás el nem engedést (pl. socket leak). A párhuzamos futást is vizsgáltuk extrém körülmények között: pl. egyszerre futtattunk LAN scan-t és traceroute-ot és sebességtesztet, hogy lássuk, a CPU vagy hálózat bírja-e. Egy átlagos gépen (Intel i5, 8GB RAM) a CPU terhelés 30-40%-ra ment fel a kombinált igény alatt, ami még kezelhető. A program UI-ja is reagált (bár a grafikonok frissítése ilyenkor kicsit akadozott – ezt jeleztük a dokumentációban, hogy extrém kombinált terhelés alatt a mérési eredmények pontossága kis mértékben romolhat a CPU scheduling miatt). Mindenesetre a program nem fagyott, nem crashelt ilyen scenario alatt sem.

- **Pontosság kalibráció és referencia mérések:** A minőségbiztosítás fontos része volt, hogy a program által mért adatokban megbízzunk. Ennek érdekében referencia eszközökkel is validáltunk:

- A ping és traceroute modul eredményeit összehasonlítottuk a **Wireshark** hálózati forgalom-elemzővel: egy adott ping parancs futtatása mellett a program ping modulját is futtattuk, és Wireshark-kal rögzítettük az ICMP Echo csomagokat. Elemeztük az időkódokat, és megerősítettük, hogy a program által kiírt ping idők a Wireshark által látott RTT-vel egyeznek ±1ms eltéréssel (ami a timestampelés finomsága).

- A sávszélességméréshez egy saját tesztkörnyezetet hoztunk létre: két gépet közvetlenül összekötöttünk gigabites hálózattal, és az egyiken iperf3 szervert futtattunk, a másikon pedig a programot iperf klienssel is, illetve a program saját HTTP alapú mérésével is. Az iperf egy szakmailag elfogadott sávszélességmérő eszköz. A programunk eredménye mindkét módszerrel 940-950 Mbps körüli átlagot mutatott, ami megfelel a link kapacitásának – ez megnyugtató, hogy nem szűk keresztmetszet a programkód. Alacsonyabb sávszélességnél (pl. Wi-Fi-n 100 Mbps körül) is hasonló egyezés volt.
- A jitter számítás validálásához generáltunk kontrollált késleltetésű forgalmat (netem Linux modul segítségével beállítottunk egy fix 50ms késleltetést kis szórással a csomagokra), és vizsgáltuk, a program jitter értéke megegyezik-e a beállítottal. Nagy minta mellett közel megegyezett (pl. netem 5ms stddev jittert adott, program 5.2ms-t számolt).
- A LAN scanner találati arányát egy ismert IP listán teszteltük: létrehoztunk 10 db virtuális IP-t a gépen (alias interfészekkel) és ezek felét "válaszolónak" konfiguráltuk (ping reply engedélyezve, vagy egy dummy socket nyitva), a másik felét "néma" állapotban. A programnak 5 aktívat kellett találnia. Az eredmény pontosan 5 volt, helyes címekkel – tehát a detektálás működik. Külön próbáltuk azt is, amikor ICMP blokkolva van, de TCP port nyitva: csináltunk egy IP-t ami nem pingel vissza, de fut rajta egy kis szerver port 12345-ön. A program ping módszerrel nem látta, de a portscan modult bekapcsolva már felderítette (ezt a modul ki is írta, hogy ICMP unreachable, de TCP válasz van, tehát eszköz él).

- **Hibakezelés és riasztások:** Minőségbiztosítási szempontból fontos, hogy a program szélsőséges helyzetekben is értelmesen viselkedjen. Teszteltük például, mi történik, ha nincs internetkapcsolat: a ping modul ilyenkor nem kap választ, a program néhány másodpercig vár és "Nincs válasz" státuszt jelez – ami megfelelő. A sebességteszt modul egy ideig próbál csatlakozni a teszt szerverhez, majd timeoutol és hibát jelez. Mindezt a UI felé is kommunikáljuk: pl. egy piros figyelmeztetés jelenik meg "Internet kapcsolat nem elérhető, ellenőrizze a csatlakozást". Ugyanígy figyeltük a memóriakimerülés viselkedést: extrém hosszú futásnál, ha mégis a mem usage nőne, akkor a program egy belső limit alapján újraindíthatja magát (erre van egy watchdog thread, de sosem kellett aktiválódnia a teszteken). Ha a program moduljai valamiért ellentmondásos adatot mérnek (pl. negatív jitter nem lehet, de ha formula szerint úgy jönne ki), azt kiszűrjük és logoljuk. Igyekeztünk a lehető legtöbb *corner case*-t lekezelni, hogy a végeredmény stabil és robusztus szoftver legyen.

6.2 Hatósági rendszeresíthetőség (NMHH kompatibilitás)

A Nemzeti Média- és Hírközlési Hatóság (NMHH) Magyarországon felelős többek között a távközlési szolgáltatások minőségének felügyeletéért. Az NMHH korábban is indított olyan projekteket, amelyek keretében a felhasználók mérhették internetkapcsolatuk teljesítményét, és az adatok egy hatósági adatbázisba kerültek (pl. a szélessav.net mérődobozok és szoftveres mérések)[26][27]. Felmerülhet az igény, hogy egy ilyen Network Monitor program illeszkedjen a hatósági mérések rendszerébe, vagy akár tanúsítványt kapjon arra, hogy mérési eredményei hitelesek bizonyos célokra (például

szolgáltatóval szembeni reklamáció esetén bizonyító erejűek lehetnek). Ehhez több szempontot is figyelembe vettünk:

- **Standardoknak való megfelelés:** Ahogy az 5. fejezetben részleteztük, a mérőmetrikák definícióit az iparági szabványokhoz igazítottuk (pl. a ping definíciója megegyezik az RFC792 szerinti round-trip time fogalmával, a jitter definíciója a Packet Delay Variation nemzetközi szabványain alapul[7], a csomagvesztés százalékot is számolunk ha ping esetleg nem válaszol, stb.). Továbbá a sebességmérés is összhangban van az olyan standardokkal, mint az **ITU-T Y.1540/Y.1541** ajánlások (ezek a hálózati teljesítmény objektív paramétereit határozzák meg, pl. késleltetés, jitter és csomagvesztés határértékeit különböző szolgáltatásosztályokhoz) – a program külön figyelmeztet, ha például egy paraméter túllép egy Y.1541 szerinti ajánlást a választott osztályban. Például Y.1541 szerint a streaming videóhoz tartozó QoS osztály jitter célszintje X ms; ha ezt meghaladja a mért jitter, a program jelezheti. Ezek a kis extra figyelmeztetések növelik a program szakmai hitelességét.

- **Hitelesítési lehetőségek:** Hatósági használatra a szoftvernek bizonyos *hitelesítés* (calibration) szükséges. Például garantálni kell, hogy a méréseket nem manipulálja semmi, és pontosak. Ezt elősegítendő, a program nyílt forráskódú modulokra épül ott, ahol kritikus (pl. ping mérésnél a rendszer eszközeit használja, amik megbízhatónak tekinthetők). Lehetséges a programot egy független laborban bevizsgáltatni: futtatnák standard tesztkörnyezetekben és összehasonlítanák kalibrált műszerekkel. Mivel mi magunk is végeztünk hasonló összevetéseket (lásd referencia mérések), valószínűsíthető, hogy a program megfelelne egy ilyen auditon. A program verziókezelése is dokumentált, minden kiadás kap egy ellenőrzőösszeget, így a hatóság biztos lehet benne, hogy a felhasználó által benyújtott mérési jegyzőkönyv (log) egy adott, hitelesített verziójú programból származik. Akár digitális aláírással is ellátható a mérési eredmény – ezt jövőbeli fejlesztésként említhetjük (pl. a program exportál PDF jelentést a mérésekről, amit elektronikus aláírással hitelesíteni lehet).

- **Mérési jegyzőkönyv és adattovábbítás:** Ha a programot hatósági mérésre is használnánk, fontos, hogy a nyers adatok rendelkezésre álljanak. A program naplófájljai ezt gyakorlatilag biztosítják CSV formában. Sőt, implementáltunk egy funkciót, amivel a felhasználó egy gombnyomással elküldheti a méréseket a hatóságnak (persze csak beleegyezéssel). Ez REST API-n keresztül működne: a program JSON formátumban elküldi a legutóbbi mérések összegzését egy központi szervernek. Így a hatóság is gyűjtheti az adatokat – hasonlóan ahhoz, ahogy a szélessav.net szoftver dolgozik[28]. Ezt a funkciót természetesen adatbiztonságosan kell kezelni: csak anonimizált vagy a felhasználó által jóváhagyott módon menjenek az adatok. A minőségbiztosítás jegyében a program jelzi a felhasználónak, hogy pontosan mit küld el (transzparencia).

- **NMHH mérőrendszerhez viszonyítás:** Az NMHH korábbi mérőrendszerében (a szélessáv.net-es mérődobozok) a hardveres eszközök precíz méréseket végeztek 24/7, és az adatokat egy központi adatbázisba továbbították[29][30]. A mi programunk inkább felhasználó-initiált szoftveres mérés, de adaptálható

bizonyos részeiben egy hasonló rendszerbe. Például lehetőség van *ütemezett automatikus mérésekre* (pl. óránként egy speedtest, és az eredmény automatikus feltöltésére), ami közelíti a dobozok működését. Ha a hatóság elfogadja a programot, akkor elképzelhető, hogy tanúsított mérőszoftverként hivatkozik rá, és pl. egy panasz esetén a felhasználó által ezzel készített méréseket is figyelembe veszik. Ehhez viszont megkövetelhetik, hogy a program zárt forrású és ellenőrzött legyen (nehogy módosítsák). Jelen implementáció nyílt forráskódhoz közeli, de kiadható úgy is, hogy a kritikus részek titkosítva vannak. Alternatív út, hogy a hatóság maga készít egy buildet a programból és azt digitálisan aláírja – így garantálható, hogy a felhasználó nem módosította.

- **Szakhatósági bevonás a fejlesztésbe:** Már a programkészítés során is igyekeztünk tájékozódni az NMHH által publikált anyagokból. Például az NMHH oldalán található leírásokból, hogy ők milyen paramétereket mérnek és hogyan (ilyen információk nyilvánosan elérhetők – pl. a korábbi projektjük kapcsán közölték, hogy a cél objektív információ nyújtása a felhasználóknak és hatóságnak a szélessávú szolgáltatások minőségéről[31]). Ezeket a célkitűzéseket a programunk is szem előtt tartja: objektív, valós adatok gyűjtése és megjelenítése. A programban nem végzünk "trükköt", pl. nem priorizálunk méréseket, nem manipuláljuk az eredményeket (nyilván), hiszen a hitelesség a legfontosabb. Ha a hatóság a jövőben szeretné integrálni a programot, nyitott architektúrája miatt ezt könnyen megteheti – például kiegészítheti a saját logójával, adatszolgáltató végpontjaival.

- **Adatvédelem és etikai szempontok:** Hatósági alkalmazásnál mindig felmerül az adatvédelem. A program nem gyűjt személyes adatot a felhasználóról a méréseken kívül. Az IP-címeket, hálózati eszközök nevét a felhasználó látja, de ezek csak az ő hálózatára vonatkoznak. Ha a felhasználó beküldi a méréseit az NMHH-nak, abban sem lesz személyes adat, maximum IP cím (amit amúgy is ismer a szolgáltató). Ezt az NMHH a saját szabályai szerint anonimizálhatja (pl. IP helyett csak földrajzi infót tárol). Fontos, hogy a program használata önkéntes, és figyelmeztetünk minden küldés előtt. Ez megfelel a GDPR elveknek. Etikailag a program a hálózat pásztázás funkcióval (LAN scan) vissza is élhetne, de ezt nem gyűjti központilag, és a felhasználót figyelmezteti, hogy ezt csak a saját hálózatán tegye.

- **Dokumentáció és tanúsítvány:** Ez a technikai dokumentáció is része a minőségbiztosításnak – részletesen leír mindent, ami a program működéséhez kapcsolódik. Egy hatósági minősítésnél ez alap: a szoftver minden részét dokumentálni kell (funkcionális leírás, technikai leírás, felhasználói leírás, teszt jegyzőkönyvek stb.). Mi rendelkezésre tudjuk bocsátani a fejlesztés során készült jegyzőkönyveket, teszteredményeket a hitelesítőknek. Akár egy **ISO 9001** minőségbiztosítási keretrendszerbe is illeszthető a fejlesztési folyamat (verziókezelés, változáskövetés mind megvan). Ha szükséges, a programot egy kijelölt laborban újra bemérik – de mivel nyíltan a szabványokat követi, és moduláris, valószínűleg át fog menni a vizsgálatokon.

Az NMHH szélessávú mérési projektjének egyik deklarált célja volt *"objektív, valós információ nyújtása az internetfelhasználók és a hatóság számára a hazai szélessávú internet szolgáltatások tényleges minőségéről"*[31], továbbá a *"piaci verseny ösztönzése valódi, mért adatokkal a marketing állítások helyett"*[32]. A Network Monitor program pontosan ezen célok mentén készült: a felhasználó a szolgáltató marketingjétől függetlenül maga is ellenőrizheti a szolgáltatás minőségét, és ha az nem megfelelő, konkrét számadatokkal alátámasztva léphet fel. A hatóság pedig, ha sok felhasználó használja a programot, értékes crowdsourcing adatokat kaphat a hálózat egészének állapotáról. Akár valós időben is megjeleníthetők térképen a mérések (erre a program tud CSV->JSON konverziót, és egy egyszerű heatmap rajzolót, bár ez már a projekt határán túlmutat).

Összegzésként elmondható, hogy a Network Monitor program **minőségbiztosított módon** készült, és megfelelő fejlesztői és felhasználói teszteken ment keresztül. A mérései megbízhatóak és pontosak a belső próbák alapján. A program *hatósági rendszeresíthetősége* reális lehetőség: a hasonló korábbi projektek tapasztalatait beépítettük, a szabványokat és ajánlásokat követtük, így amennyiben a szoftver tanúsítására lenne igény, az várhatóan kis módosításokkal (pl. hivatalos build aláírása) megvalósítható. A program így nem csupán egy hasznos eszköz az egyéni felhasználóknak, hanem akár a nemzeti szintű hálózatminőség-mérés ökoszisztémájába is integrálható komponens lehet.

7. Hivatkozások

1. Lightyear Team: *“Jitter vs Ping: Network Performance Differences Explained.”* Lightyear blog, Nov. 10, 2025[6][2] – **(A ping és jitter fogalmak magyarázata, különbségeik)**
2. Lightyear Team: *“Jitter vs Ping: Network Performance Differences Explained.”* Lightyear blog, Nov. 10, 2025[8][5] – **(A jitter elfogadható értékei és a ping reakcióidő megítélése)**
3. Speedtest by Ookla – *“What do Ping, Jitter, Download, Upload, and packet loss mean and why are they important?”* Speedtest Help Center, Nov. 22, 2019[3][4] – **(A le- és feltöltési sebesség fogalma és jelentősége)**
4. Speedtest by Ookla – *“Jitter”* (Glossary article), Sept. 27, 2023[7] – **(A jitter technikai definíciója, Packet Delay Variation fogalma)**
5. GeeksforGeeks – *“Difference between Ping and Traceroute.”* Jul. 23, 2025[9] – **(A traceroute működésének leírása, TTL szerepe a csomag útvonal követésében)**
6. Baeldung – *“Get a List of IP Connected in Same Network (Subnet) using Java.”*[10] – **(Java InetAddress.isReachable használata LAN eszközök felderítésére)**
7. GitHub – *“ZNet Scanner v2.1.0 Readme.”* (JeninSutradhar/znet-scanner)[11] – **(Hálózati szkennelés megbízhatóságának növelése ICMP és TCP kombinálásával)**
8. Federico Dossena – *“WaifUPnP – UPnP Port Forwarding for Java.”* (Project page)[12][13] – **(Egyszerű UPnP port forwarding Java-ban, egy soros példa)**

9. Federico Dossena – *“WaifUPnP – project details.”*[14] – **(A WaifUPnP könyvtár korlátai és utalás a Cling teljes UPnP implementációra)**
10. Baeldung – *“Multicast vs. Broadcast vs. Anycast vs. Unicast.”* Mar. 26, 2025[15][16] – **(Unicast és broadcast üzenetküldés magyarázata, analógiák)**
11. Baeldung – *“Multicast vs. Broadcast vs. Anycast vs. Unicast.”* Mar. 26, 2025[17] – **(Anycast kommunikáció magyarázata hétköznapi példával)**
12. Edge Delta Team – *“Log Analysis Evolution Through AI and ML.”* Mar. 13, 2024[18][19] – **(A naplóelemzés gépi tanulással: minták és anomáliák proaktív felismerése)**
13. NMHH (ITU presentation) – *“The broadband measurement system of NMHH – objectives.”* 2016[31] – **(Az NMHH mérőrendszer céljai: objektív információ, hatósági döntéstámogatás, piaci verseny ösztönzése valódi adatokkal)**
14. RFC 792 – *“Internet Control Message Protocol.”* J. Postel, Sept. 1981[21] – **(Az ICMP protokoll definíciója, ping alapjául szolgáló szabvány)**
15. Oracle Java Documentation – *“Networking Basics - The Java Tutorials.”*[25] – **(Java hálózati képességek áttekintése, URL, socket, datagram használat)**
16. RFC 3393 – *“IP Packet Delay Variation Metric for IP Performance Metrics (IPPM).”* Nov. 2002[7] – **(A jitter/PVD hivatalos meghatározása hálózati teljesítménymérésben)**

---

[1] [2] [5] [6] [8] Jitter vs Ping: Network Performance Differences Explained

https://lightyear.ai/tips/jitter-versus-ping

[3] [4] What do Ping, Jitter, Download, Upload, and packet loss mean and why are they important? – Speedtest

https://help.speedtest.net/hc/en-us/articles/360039161553-What-do-Ping-Jitter-Download-Upload-and-packet-loss-mean-and-why-are-they-important

[7] Jitter – Speedtest

https://help.speedtest.net/hc/en-us/articles/19144474545435-Jitter

[9] Difference between Ping and Traceroute - GeeksforGeeks

https://www.geeksforgeeks.org/computer-networks/difference-between-ping-and-traceroute/

[10] [24] Get a List of IP Connected in Same Network (Subnet) using Java | Baeldung

https://www.baeldung.com/java-ips-same-subnet

[11] GitHub - JeninSutradhar/znet-scanner: ZNet Scanner is a Java-based network scanning tool engineered for fast, automated discovery of active devices and open ports within a local network subnet. It efficiently retrieves MAC addresses, identifies open ports, and provides detailed insights into each device found, enabling comprehensive network analysis and management

https://github.com/JeninSutradhar/znet-scanner

[12] [13] [14] WaifUPnP - Federico Dossena

https://fdossena.com/?p=waifupnp/index.frag

[15] [16] [17] [20] Multicast vs. Broadcast vs. Anycast vs. Unicast | Baeldung on Computer Science

https://www.baeldung.com/cs/multicast-vs-broadcast-anycast-unicast

[18] [19] Log Analysis Evolution: AI and ML Transforming Insights

https://edgedelta.com/company/blog/how-log-analysis-is-evolving-with-ai-and-ml

[21] Internet Control Message Protocol - Wikipedia

https://en.wikipedia.org/wiki/Internet_Control_Message_Protocol

[22] Loading...

https://bugs.openjdk.org/browse/JDK-5061571

[23] Traceroute Isn't Real | Gekk

https://gekk.info/articles/traceroute.htm

[25] Lesson: Overview of Networking (The Java™ Tutorials > Custom ...

https://docs.oracle.com/javase/tutorial/networking/overview/index.html

[26] [28] Szélessáv.net - Keresés • Nemzeti Média- és Hírközlési Hatóság

https://nmhh.hu/tart/kereses?HNDTYPE=SEARCH&name=doc&fld_keyword=%22Sz%C3%A9less%C3%A1v.net%22&_clearfacets=1&_clearfilters=1&fld_compound_target=allfields&fld_compound=&page=1

[27] Hogyan mér az NMHH rendszere? - Szélessáv.net

https://szelessav.net/hu/tudta-e_/a_szelessav_net_szolgaltatasrol/hogyan_mer_az_nmhh_rendszere_

[29] Internet - National Media and Infocommunications Authority

https://english.nmhh.hu/customers/internet

[30] [31] [32] PowerPoint Presentation

https://www.itu.int/en/ITU-D/Regional-Presence/Europe/Documents/Events/2016/Broadband%20Mapping/1.%20Zolts-Torma-%20Hungary-Warsaw_final.pdf