Budapesti Műszaki és Gazdaságtudományi Egyetem
Közlekedésautomatika Tanszék

Felügyelő és adatküldő rendszer fejlesztése ACTROS forgalomirányító berendezéshez

Kopcsányi Sándor **2009.május**

Tartalomjegyzék

1. Bevezetés

A közúti közlekedés a közelmúltban, és jelenleg is egyre nagyobb szerepet kap, térhódítása megállíthatatlan. Az egyre több jármű, utazási igény, mind növeli a forgalmat, így ez a folyamatosan növekedés egyre nagyobb feladat elé állítja a közúti közlekedési infrastruktúrát. A forgalom növekedésének legszembetűnőbb negatív hatásai a forgalmi torlódások, amelyeknek további gazdasági negatívumai, költségei is vannak: késések miatti termeléskiesés, fokozott üzemanyagfogyasztás, balesetveszély, környezet-, zajszennyezés költségei. Ezek együttesen globális, szükségszerűen megoldandó problémát jelentenek. Megoldásként két mód kínálkozik: újabb utak építése, és a forgalom szabályozása. Az infrastruktúrák bővítése egyrészt igen költséges, másrészt vannak helyek, ahol nem kivitelezhető hely hiányában. A forgalomszabályozás ellenben olcsóbb, és hatékony megoldás ott, ahol nincs mód újabb utak, sávok megépítésére, valamint a kiépítése is gyorsabb. A fentieknek megfelelően egyre inkább a forgalom szabályozás irányába terelődött, terelődik a hangsúly. Napjainkban a forgalom szabályozását forgalomirányító központok végzik, amelyek detektorokkal (hurokdetektor, kamera, ultrahang, infrasugár) végzett méréseken keresztül figyelik a forgalmat. A mérésekből kapott, illetve az azok alapján számított értékek (forgalomnagyság, átlagsebesség, stb.) alapján lehetőség nyílik a forgalom szabályozására, ezáltal optimalizálható a forgalom lefolyása, mérsékelhetők a torlódások. Ilyen korszerű csomóponti forgalomirányító berendezés az ACTROC VTC 3000 is. 2006 óta a Közlekedésautomatika Tanszék is rendelkezik ilyen berendezéssel. Ez a berendezés a hagyományos jelzőlámpás forgalomirányításon felül, szinte bármilyen közúti forgalomszabályozás vezérlésére alkalmas.

Jelenleg az ACTROSnem menti folyamatosan a detektorértékeket. Egy olyan megoldás van, amelynek során el lehet indítani egy adott intervallum forgalmi értékeinek mentését, majd az intervallum eltelte után az adatok lekérdezhetők.

A diplomamunka célja egy olyan programrendszer elkészítése, amely az ACTROS berendezéshez beérkező detektorértékeket feldolgozza, majd internet alapú kapcsolat segítségével elküldi azt egy szervernek. A szerver fogadja és merevlemezre menti az adatokat. A szerver oldalon továbbá egy kiegészítő alkalmazás is helyet kap, amely a mentett adatok alapján különböző statisztikai adatok megjelenítésére szolgál.

Az adatküldés folyamatos, így az értékek visszamenőleg lekérdezhetők egy később jelentkező igény esetén is, valamint az adatárrolással párhuzamosan (realtime) is megjeleníthetők a paraméterek.

A diplomaterv javaslatban foglaltaknak megfelelően öt fő rész különíthető el. A második fejezetben az ACTROS forgalomirányító berendezés rendszertechnikai bemutatására kerül sor. A harmadik fejezet a jelenleg alkalmazott felügyelő, felprogramozó szoftvereket tekinti át a hazai forgalomirányító berendezések területén, és mivel a berendezésen Java alapú

szoftver fut, erről is szó esik. A negyedik fejezetben a kifejlesztésre került program funkciói, a program működésének folyamatábrája kerül bemutatásra. Az ötödik fejezet a program Java nyelven történő megírását, tesztelését, kipróbálását mutatja be, végül a hatodik fejezet a futtatások eredményeit és a továbbfejlesztés lehetőségeit taglalja.

2. Az ACTROS forgalomirányító berendezés rendszertechnikai bemutatása

2.1. A berendezés innovativitása

Az ACTROS forgalomirányító berendezés lehetővé teszi az intelligens forgalomtechnikai szabályozás megvalósítását. A technológiai kialakítás szabványosított, tovább fejleszthető, valamint figyelembe veszi az aktuális és jövőbeli követelményeket.
Az ACTROS belső kommunikációját CAN-bus-on keresztül végzi, így igen gyors, mind a feldolgozási sebességet, mind a rendszerkiszolgálást illetően. A moduláris felépítésnek köszönhetően rugalmas, az egyedi igényekhez formálható, és egyszerre több, akár három csomópont vezérlésére alkalmas. További pozitívum a felhasználóbarát kialakítás: internet alapú technológia, felhasználói adatok kezelése, program fel- és letöltési funkciók, hibatárolás. Támogatja az egyedi berendezésként történő alkalmazást, csakúgy, mint a hálózatba történő bekötést. Tartalmazza az M-Tech Csoport specifikus és szabványosított interfészprotokollját, valamint az OCIT nyílt központi interfészét. Szervízelése, üzemeltetés egyszerű, továbbá a biztonsági rendszer is igen fejlett, a hardvereken kívül a szoftver ellenőrzése is folyamatosan történik a berendezés működése közben.

2.2. A berendezés részei

2.2.1. A berendezés rendszer-hardverei

CPU LS2000 vezérlőegység
Részei:

- Flash EPROM memória (16 MB / 32 MB)
- RAM memória (16 MB / 32 MB): forgalomtechnikai adatok tárolása
- SRAM memória (1 MB): hibatárolás
- RTC órajeladó (Real Time Clock)
- Szerviz interfész
- További soros interfészek

- Opcionális csatlakoztatási lehetőség Compact Flash Card vagy Micro Drive számára
- Ethernet interfész

DCF rádióóra modul

Az ACTROS rendszeridejének előállítását végzi.

CPU 020

Részei:

- 32 bites Motorola processzor
- EPROM a meghajtó rendszer számára (2 MB)
- EPROM a nem változtatható, biztonsági adatok számára (256 kB)
- Képernyőinterfész
- Óra
- SRAM memória (2 MB)
- Flash EPROM (128 kB)
- RAM (2 MB)

CPU-V55 ellenőrző egység

Ez egy függetlenül ellenőrző egység, a lámpák állapotát CAN buszon keresztül olvassa ki, majd összehasonlítja a referencia adatokkal.

NKK hálózati ellenőrző kártya

Funkciói:

- Bekapcsolási funkció
- Részcsomópont relék a részcsomópontok meghajtásához
- Hálózati feszültség mérés és ellenőrzés
- Rendszerfeszültség figyelés
- Rendszer-újraindítás generálás
- 100 Hz-es rendszerütem generálás

ASK automata és védelmi kártya

A hálózatra kapcsolódást és az arról való lekapcsolást irányítja.

A kártya két automata biztosítékot tartalmaz a lámpafeszültség és a vezérlés számára.

2.2.2. A berendezés funkció-hardverei

SK 24 kapcsolókártya

A jelzőberendezések jelkimeneteit ellenőrzi és vezérli. A kártya 24 konfigurálható jelcsatorna irányítására alkalmas. Minden csatornán áram és feszültségellenőrzés folyik.
A lámpakimenetek kapcsolására csatornánként egy-egy kapcsoló szolgál.
A rendelkezésre állás növelése érdekében minden lámpakimenetbe egy lassú megszakítású biztosíték van beiktatva, így hiba esetén a főbiztosíték lekapcsolása elkerülhető.
A lámpaellenőrző a kimenő áramot és feszültséget méri, majd egy A/D átalakítón keresztül végzi az ellenőrzést.
A lámpahibák és lámpaállapotok mellett a hardverek belső állapotainak ellenőrzése is folyamatosan történik, a teljes biztonsági rendszer érdekében. A hibajelentések átadása CAN buszon történik, ill. ezzel párhuzamosan a hozzátartozó LED-ek is kigyulladnak.
A mikrokontroller szoftvere Flash EPROM-ban van eltárolva, így bármikor aktualizálható.
A kapcsolókártya külső 24 V-os feszültséggel van ellátva.

IO 24 ki-és bemeneti kártya

Ez a hardver egy kombinált ki- és bemeneti kártya.
Csatornái:

- 16 bemenet (8 optikai csatlakozós bemenet, 8 szenzoros bemenet)
- 8 kimenet (relés kimenet)

A bemenő jelek 10 milliszekundumos ciklusban kerülnek lekérdezésre. A kártya 8 opto-csatlakozós és 8 szenzoros - a vezérléstől galvanikusan leválasztott - bemenettel rendelkezik.
Az opto-csatlakozós bemenetek 5 és 24 V közötti egyenfeszültséggel, amíg a szenzorosak 15 és 230 V közötti egyen- vagy váltófeszültséggel működtethetők.
A relés kimenetek is galvanikusan leválasztottak. 5 V-tól 230 V-ig egyen- vagy váltófeszültséggel működtethetők.
A kártya tápfeszültségének ellenőrzése folyamatosan történik. A hibajelentések átadása CAN buszon történik, ill. ezzel párhuzamosan a hozzátartozó LED-ek is kigyulladnak.
Mind a 24 csatornához tartozik egy-egy LED, amely sárgán, pirosan és zölden is meggyújtható, ill. villogtatható.
A kártya szoftvere Flash EPROM-ban van eltárolva. A szoftver a meghajtó és az ellenőrző funkciókat tartalmazza.
A berendezés a következő oldalon, az 1. ábrán látható.

1.ábra ACTROS VTC 3000

2.3. A berendezés kezelése

2.3.1. Elindítás

Fázishelyesség

A berendezés normál hálózati feszültségről (230 V) üzemeltethető. A berendezés azonban csak fázishelyes csatlakoztatással működik. A táphálózatra való kapcsoláskor tehát ügyelni kell a fázishelyességre.

Biztosítékok

A berendezés többszörös védelemmel van ellátva. A bekapcsolás a főbiztosítékkal történik. Négy automata kisbiztosító is helyet kapott, amelyek az automata- és biztosítókártya biztosítékai, a másik kettő pedig szabadon felhasználható.
Található még a készüléken egy 230 voltos aljzat is és egy hozzátartozó biztosíték.

2.3.2. A kezelőfelület

A készülék közvetlenül vezérelhető a rajta található kezelőfelületének nyomógombjaival. A kezelőfelület CAN buszon kommunikál a kapcsolóművel. A gombok lenyomásával járó feladatokat a gép a FIFO elv szerint végzi el.

TK 1 / TK 2 / TK 3 gombok

A három TK 1 / TK 2 / TK 3 (Teilknoten) gombokkal kiválaszthatjuk, hogy melyik részcsomóponttal foglakozunk.

AUS gomb

A piros színű AUS kapcsolóval a sárga villogó programot közvetlenül kivezéreljük az adott részcsomópontra. A gombba integrált piros LED ekkor villogni kezd, illetve a villogó program elindulását követően folyamatosan égni fog.

A kijelző

A kijelző három különböző funkcióban dolgozik:

- kijelző
- üzemmód beállítások
- hibatároló

Kijelző funkció

Ez a berendezés alap üzemmódja, amely a következő információkat mutatja (adott részcsomópontra vonatkozóan):

- Idő
- Programszám
- Programszámláló
- Irányítószint:
    - Man (manuális)
    - MExt (külső manuális)
    - VTAnw (VT felhasználó)
    - Zentr (központi)
    - Uhr/DCF (időterv alapú)
    - Sys (Rendszer)
- Fázisszám
- Fázisszámláló

Üzemmód funkció

Az üzemmód funkció választásához a Ba gombot kell megnyomnunk. Ez a funkció gyakorlatilag a menü.
A menüből a TK 1/2/3 ill. az AUS gombokkal tudunk kilépni.

A funkció főmenü-pontjai között is a Ba gombbal tudunk haladni:

- Automatik
- Lokal-Prg
- Hand-Prg
- Sonder-Prg
- Betriebsmodus
- Fehlerspeicher

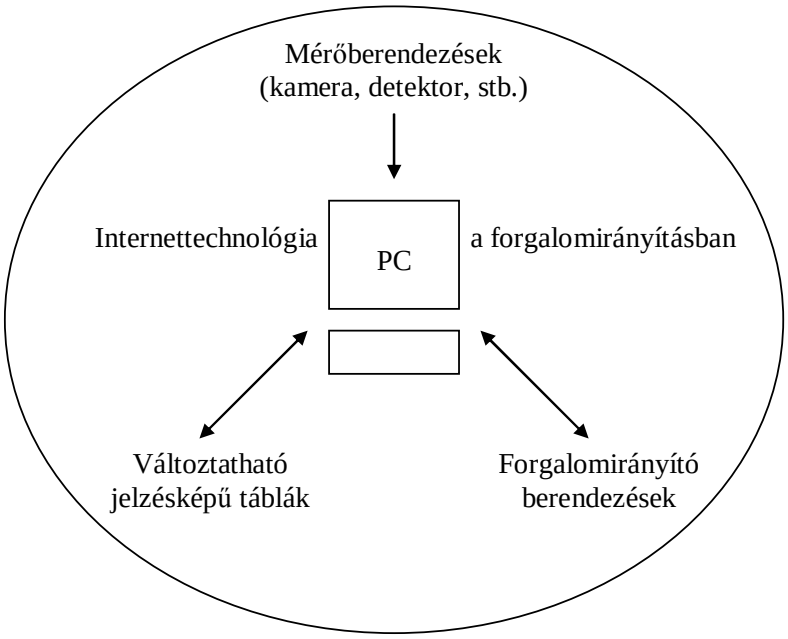
Az almenü-pontok között a +/- gombokkal tudunk lépegetni. A kiválasztott funkciót a TK 1/2/3 gombokkal tudjuk indítani. Ezután az adott TK gomb LED-je addig fog villogni, amíg a kívánt állapot be nem következik.
Az F (Fortschaltung) gombot a Hand-Prg menüpontban tudjuk alkalmazni, amennyiben fel akarunk oldani egy stop-pontot. Ennek használatával a berendezés vissza fog térni abba a programba és irányítási szintre, amelyben a Handprogramm előtt működött.

Hibatároló funkció

A hibatároló funkcióba a kijelző funkcióból tudunk átlépni a + gombbal. A funkciót a Ba ill. az AUS kapcsolókkal tudjuk elhagyni.
Ebben a funkcióban tudjuk a hibatároló tartalmát kiolvasni, amely maximum 100 bejegyzést tud tárolni. A hibák között a +/- gombokkal tudunk lépegetni.

2.4. OCIT

Az ACTROS-hoz kapcsolt detektorok OCIT technológiával csatlakoznak a berendezéshez.
OCIT=Open Communication Interface for Road Traffic Control Systems.
Ez a rendszer internet technológia alapú, nyílt interfészek megvalósítására szolgál a közúti közlekedési forgalomirányítás számára. Az internettechnológia forgalomirányításban való megjelenését a következő oldal 2. számú ábrája szemlélteti. A technológia kialakítása 5 német cég nevéhez fűződik: Dambach, Siemens, Signalbau Huber, Stoye, Stührenberg. Az OCIT kizárólag ipari felhasználású, forgalomirányításban alkalmazott protokoll. OCIT technológiát alkalmaznak jelenleg a következő országokban: Németország, Ausztria, Hollandia, Magyarország, Svájc, Spanyolország. Az OCIT lehetővé teszi, hogy a forgalomirányító berendezéseket, a központi elemeket és irányítóterületet egy hálózatba fogjuk össze.



2.ábra Internettechnológia a forgalomirányításban

Az OCIT rendszer

Az OCIT a nyitott rendszer-architektúra alapját képezi. Ezek az interfészek szabványosított kapcsolatot valósítanak elosztott központi és nem központi elemek között. Mivel internet-technológiát használ, lehetőség nyílik a forgalomirányítási rendszerek és központok össze-kapcsolására.

Az OCIT felépítése:

- rendszer-architektúra
- szabályok
- OCIT-protokoll
- funkciók
- átviteli protokoll

Az OCIT-architektúrája

Az OCIT rendszer három interfészterületet különböztet meg:

- belállomás: szabványosított interfészek a központi elemek között
- külállomás: szabványosított interfészek a központ és a terepi berendezések között
- OCIT-LED: elektromos interfész a jelzőberendezés és a LED-jeladómodul között

Az OCIT adatátvitel és protokoll

Az OCIT adatátviteli technika a szabványos TCP/IP protokollra épül, amivel egy megbízható adatkapcsolatot nyújt. Az OCIT saját protokollja a BTPPL (Basis Transport Paket Protokoll Layer), ami együttműködik az internetes szabvánnyal. A BTPPL kétcsatornás adatfolyam. Az egyik – magasabb prioritással rendelkező - csatornát kapcsolási parancsok és jelentések küldésére használják. Az alacsonyabb prioritású csatornát pedig az alapvető működés ellátását közvetlenül nem befolyásoló adatok cseréjére. Az adatátvitel működése aszinkron.

Az OCIT kommunikáció az OSI (Open Systems Interconnection) referenciamodellen alapul. Az OSI rétegmodell – amelynek rétegeit az 1. táblázat tartalmazza - implementálásával a legkülönbözőbb hálózati rendszerek (pl. különböző gyártók esetén) egy nyitott, egymással kompatibilis kommunikációs hálózattá kapcsolhatók össze.

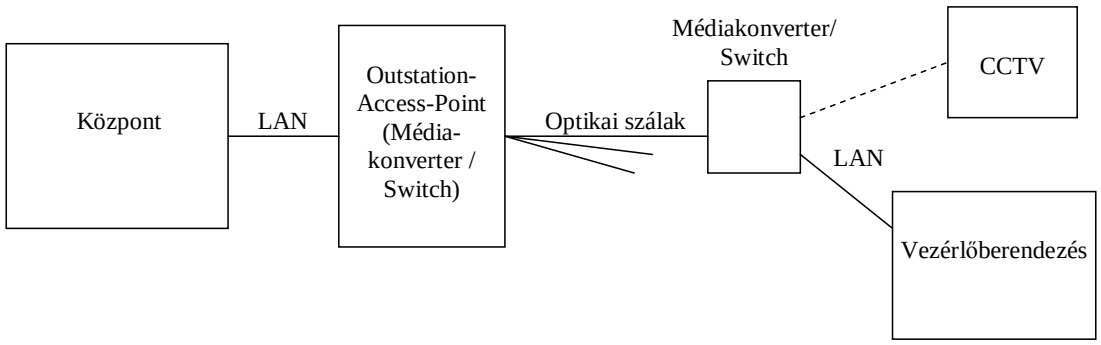
7.	Alkalmazás
6.	Megjelenítés
5.	Kommunikációvezérlés
4.	Transzport
3.	Közvetítés
2.	Biztonság
1.	Fizikai réteg

1.táblázat OSI rétegek

A BTPPL protokoll az 5., 6. és 7. réteg funkcióit fogja át. A 3. és 4. szint protokollja a TCP/IP. Az 1-es és 2-es rétegen keresztül telekommunikációs szolgáltatók kapcsolhatók a rendszerre.

3. Napjaink forgalomirányító berendezéseinek, szoftvereinek áttekintése, Java programozási nyelv ismertetése

3.1. Napjaink forgalomirányító berendezései



3.ábra Jelenlegi modern forgalomirányítási rendszer

A jelenlegi közúti forgalomirányító rendszerek elvi vázlata a 3. ábrán látható. A forgalomirányító berendezések piacán megtalálhatók többek közt a Siemens cég termékei is. A budapesti forgalomirányító központok is Siemens gyártmányúak. Jelenlegi legmodernebb rendszerük:

SITRAFFIC CANTO

TCP/IP technológián alapszik, ez lehetővé teszi a mai modern hálózatok felhasználását, illetve az átviteli útvonalak többszörös felhasználása is lehetséges, például kamera képének továbbítására.

A CANTO szinte minden Siemens vezérlőberendezéshez csatlakoztatható. Így a régi analóg technika egyetlen átállással a legmodernebb színvonalra emelhető. Több, jelenleg különálló alközpont számítógépei egy berendezéssel helyettesíthetők, ez növeli a rendelkezésre állást, csökkenti a költségeket. A vezérlésnél elérhető funkciók a forgalomfüggő programozás, gépnapló lekérdezés, távprogramozás. A gyakran módosítandó programrészletek a központból a beágyazott SITRAFFIC Control szoftverrel menüvezérelten szerkeszthetőek, pl. a helyi programok, vagy a forgalomfüggő paraméterek.

A magyarországi forgalomtechnikai eszközök piacának egy másik jelentős képviselője a VILATI Signalbau Huber Forgalomtechnikai Kft. A cég legmodernebb berendezése az

ACTROS VTC 3000, amelynek ismertetése a 2. fejezetben megtörtént, azonban a cég forgalmaz további forgalomirányító rendszereket is:

VRS 5000

A VRS 5000 roppant innovatív rendszerfelépítéssel rendelkezik. A hardverbázis és a vezérlés fejlesztése során modern technológia került alkalmazásra. A rendszer alapja egy hardverfüggetlen szoftverplattform. A rendszer összeállítása során mind a hardver, mind a szoftver esetében jövőbe mutató szabványok lettek figyelembevéve. Az adatátvitelhez és a hálózat felépítéséhez internet-technológiák kerülnek alkalmazásra, ennél fogva a VRS 5000 hiánytalanul integrálható jövőbeli rendszereket átfogó hálózati struktúrákba. Sokféle átviteli eljárás alkalmazható, ezáltal az átviteli kapacitás egyértelműen nő. Az értéknövelő szolgáltatások támogatása megvalósítható. A rendszerintegráció lehetővé teszi a különféle közlekedési és infrastrukturális rendszerekkel történő adatcserét.

UZ 2000

A modern közlekedésbefolyásoló eszközök nagy teljesítőképességű vezérlőközpontokat igényelnek. A VRZ 2000/UZ 2000 fejlesztésének fő tulajdonságai a következők:

- megbízható
- bővíthető
- forgalomtechnikailag rugalmas
- felhasználóbarát kezelőfelület
- operációs rendszer függetlenség

A függetlenség révén lehetőség nyílik az egyes szoftvermodulok különböző operációs rendszereken (pl. MS-Windows NT, Unix) történő használatára. Az alkalmazott szoftvertermékeket széles körű hitelesítésnek vannak alávetve, így garantálva a berendezések megbízhatóságát. Az alapul szolgáló szoftverarchitektúra nyitott és moduláris felépítésű. A VRZ 2000/UZ2000 központtal a következő üzemmódok lehetségesek:

- teljesen automata, autonóm üzemmód
- teljesen automata, integrált üzemmód
- teljesen automata üzemmód manuális beavatkozási lehetőséggel
- félautomata üzemmód nyugtázással, kézi vezérlés a kezelőfelületen keresztül.

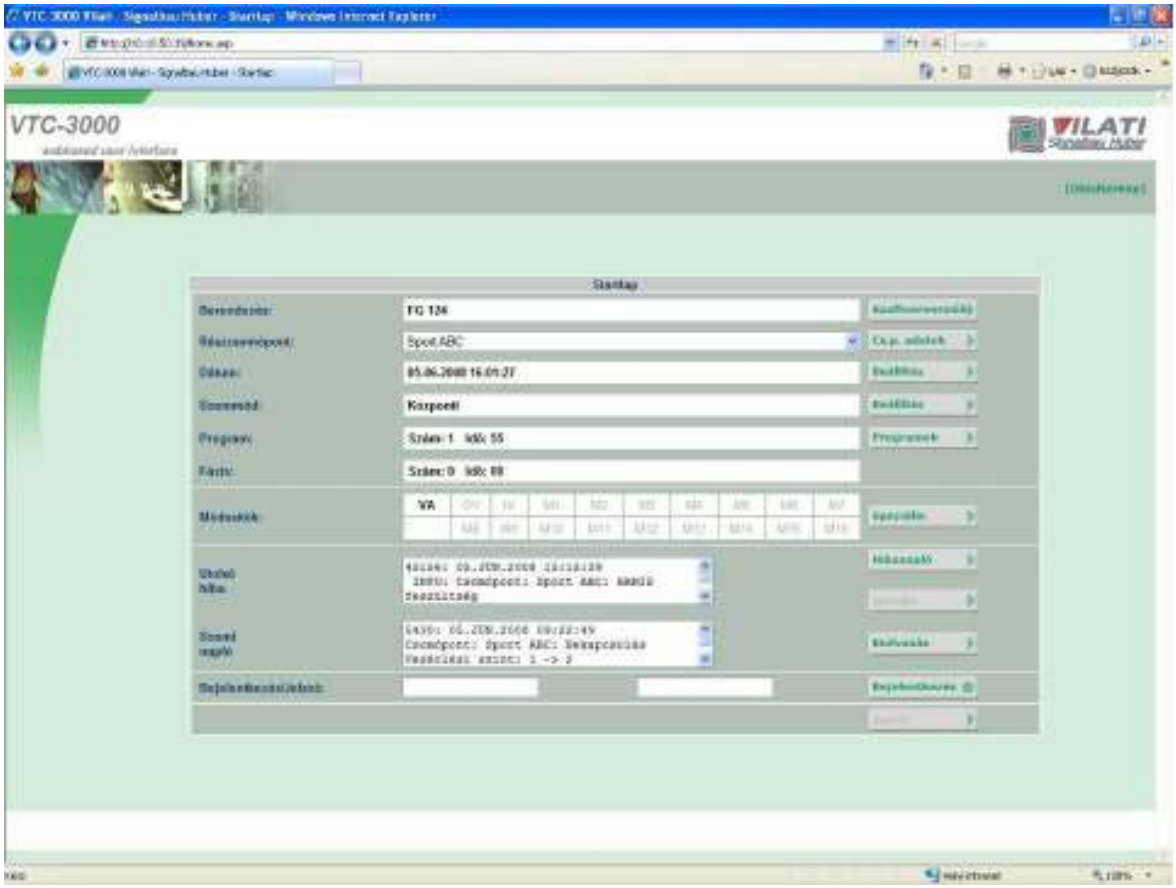
3.2. Felügyelő, felprogramozó szoftverek

A jelenlegi modern rendszerek meghatározó rendszertechnikai elemei a rajtuk futó programok. Igen sok biztonsági és forgalomtechnikai funkció szoftveresen van megvalósítva. Az így felépített rendszerek előnye, hogy könnyen újrakonfigurálhatók, akár az egyes hardverelemek módosítása nélkül.

Alapvetően a berendezéseken futó szoftvereknek 3 fajtáját különböztethetjük meg:

- Működtető szoftverek: a berendezés vezérlését végzik. Képesek kapcsolatot létesíteni a felprogramozó eszközökkel, képesek a letöltött programot feldolgozni, futtatni.
- Felhasználói, felügyelő szoftverek: olyan futtatható programok, web-es felületek, amelyek az adott csomópontokhoz tartozó paramétereket, mérési eredményeket, és egyéb adatokat tartalmaznak, illetve fogadni képesek. További funkció az egyes alapprogramok, és az egyes jelzőcsoportok befolyásolása, valamint alkalmas egyes védelmi ellenőrzések végrehajtására, például a nem kívánt állapotok figyelésére.
- Felprogramozó szoftverek: A Siemens és a Vilati berendezések felprogramozása is PC-ről történik. Az összekapcsolás általános vagy speciális illesztőkábellel történik. A közvetlen (online) kapcsolat következtében a gép futás közben tesztelhető. A szoftverek hiányossága, hogy a forgalomirányító berendezés nélkül (offline) nem képesek a programokat lefuttatni, tesztelni, így a fejlesztőnek szüksége van egy külön berendezésre, hogy programokat tudjon fejleszteni.

Az ACTROS is rendelkezik webes felhasználói felülettel, mely működését tekintve megfelel a fent említetteknek, kinézete a 4. ábrán látható.



4.ábra Az ACTROS felhasználói felülete

A felprogramozható berendezések igen komoly hányadának programja napjainkban Java nyelven íródik, mint ahogy ez az ACTROS esetében is történt. A következő pontok erre a nyelvre nyújtanak betekintést, sorra véve azon tulajdonságokat, amiért oly nagy előszeretettel használják a programozási feladatok széles területén.

3.3. Java programozási nyelv

A programnyelv mindenki számára ingyenesen elérhető, így a SUN weboldaláról letölthető a Java Fejlesztői Csomag (Java Development Kit), ami tartalmaz egy fordítóprogramot (javac), egy nyomkövetőt, hibakeresőt (jdb) a futtatáshoz szükséges virtuális gépet és egy a programkákat (Applet) futtató alkalmazást (AppletViewer).

3.3.1. A kezdet

Az 1990-es évek elején a SUN vállalatnál elindult egy kevéssé fontos projekt azzal a céllal, hogy betörjön a processzorral vezérelt, programozható (smart) készülékek területére. E készülékcsalád jellegzetes képviselője a kábel-TV társaságok által használt vezérlő (set-top box). Az ilyen készülékek programozásához igény volt olyan architektúra-független technológiára, amely lehetővé tette a kész programok hálózaton keresztül a készülékbe letöltését, megbízható futtatását. A projekt kezdetben a C++ nyelvet használta, ám a fejlesztők ezt is, a többi, akkor hozzáférhető programozási nyelvet is alkalmatlannak találták a célkitűzések maradéktalan megvalósítására. Kiindulási alapként a C++ nyelvet használták, helyenként egyszerűsítve, esetenként bővítve azt. A felhasználói elektronikai alkalmazások lassabban fejlődtek, mint azt előre várták, a projekt szép csendben el is halt volna, ha közben az Internet hálózat nem indul rohamos fejlődésnek. Észrevették, hogy az Internet hálózat hasonló körülményeket teremt és hasonló igényeket támaszt egy új programozási technológiával szemben, így a Java nyelv „összeházasodott” az Internet hálózattal, s megkezdte világméretű terjedését.

3.3.2. A nyelv legfontosabb tulajdonságai

3.3.2.1. Egyszerű

A nyelv szintaxisa és szemantikája nagyban hasonlít a sokak által ismert C illetve C++ programozási nyelvhez. A C++ nyelvből elhagytak bonyolult elemeket, de nem mindent, például megmaradt a futásidejű hibák lekezelése, az ún. kivételkezelés (exception handling), erről később részletesen. Az egyszerűsítések számos programozási hibát szüntetnek meg: nincs goto, és nincsenek pointerek sem, amik rendszeres hibaforrások voltak.

Van helyettük szemétgyűjtés (felszabadítja a már nem használt tárterületeket - garbage collection), és mindenre kiterjedő kivételkezelés.

3.3.2.2. Objektumorientált

Az objektumorientáltság fő tulajdonságai:

- Egy objektumorientált program együttműködő objektumok (object) összessége.
- A program alap építőkövei az objektumok. Ezek olyan, a környezetüktől jól elkülöníthető, viszonylag független összetevők, amelyeknek saját viselkedésük, működésük és lehetőleg rejtett, belső állapotuk van. Egy objektumra a környezetben lévő egyéb objektumok hatnak és ennek hatására saját állapotuk megváltozhat.
- Minden objektum valamilyen osztályba (class) tartozik. Az osztályok megfelelnek az absztrakt adattípusoknak, minden objektum valamely típus példánya, egyede (instance). Az osztályok definiálják az egyes objektumok állapotát leíró adatszerkezetet és a rajtuk végezhető műveleteket, az úgynevezett módszereket (method). Az egyes egyedek csak az állapotukat meghatározó adatszerkezet tényleges értékeiben különböznek egymástól, a módszerekkel definiált viselkedésük közös.
- Az egyes osztályokat az öröklődés hierarchiába rendezi. Az öröklődés olyan eljárás, amely segítségével egy osztály felhasználhatja a hierarchiában felette álló osztályokban definiált állapotot (adatszerkezeteket) és viselkedést (módszereket). Így a közös elemeket elegendő egyszer, a hierarchia megfelelő szintjén definiálni.

Általában csak az első három követelménynek eleget tevő programozási nyelvet, technológiát objektum-alapúnak (object based), amíg a negyedikkel kiegészülőket objektumorientáltnak (object-oriented) hívják. Az első 3 építőelem már korábbi programozási nyelvekben is megjelent, ezt leggyakrabban absztrakt adattípusoknak nevezték. A Java lényegében a C++ objektumorientált tulajdonságait tartalmazza. A programozó absztrakt adattípusként viselkedő osztályokat definiálhat, az osztályok műveleteket - módszereket - tartalmaznak, amelyek a rejtett adatreprezentáción (egyedváltozók) operálnak. Létrehozhatunk objektumokat, azaz egyes osztályokba tartozó egyedeket. Osztályok definiálásánál felhasználhatunk már meglévő osztályokat, az új osztály (a leszármazott) örökli a szülő adatait, módszereit. A módszerek hívása mindig futási időben, az objektum aktuális típusának megfelelően kerül kiválasztásra (virtuális módszerek, polimorfizmus). Az egyes osztályokban definiált változók és módszerek láthatóságát a C++-hoz hasonlóan lehet megadni, amelyre később részletesen kitérünk. Eltérést jelent a C++-hoz képest, hogy a Java-ban a beépített, egyszerű adattípusú - numerikus, logikai és karakter típus - változók kivételével minden objektum, így a string is, az egyetlen összetett adattípus, a tömb (array) teljes értékű osztályként viselkedik. Mivel a String objektum, így a programkódban a String mindig nagy kezdőbetűs, amíg az integer változó, így kisbetűvel írjuk. A program nem tartalmaz globá-

lis változókat és globális eljárásokat, minden adat és eljárás valamilyen objektumhoz, esetleg osztályhoz kötődik. A Java-ban minden módszerhívás - a fent említett statikus módszerek kivételével - virtuális. A C++-hoz hasonlóan lehetőségünk van az egyes objektumok típusát futási időben lekérdezni (ez a C++-ban is viszonylag új nyelvi elem az ún. RTTI, Run-Time Type Interface), sőt itt akár az osztályok forrásprogramban definiált nevét futás közben is felhasználhatjuk például objektumok létrehozására. Az osztályok mellett a Java az Objective-C programozási nyelvből átvette az Interface fogalmat. Az Interface nem más, mint módszerek egy halmaza - adatszerkezeteket, egyedváltozókat nem tartalmaz -, amelyet egyes osztályok megvalósíthatnak (implementálhatnak). A Java a C++-szal ellentétben nem engedi meg a többszörös öröklődést, viszont Interface-ek használatával, egyszerűbben, kevesebb implementációs problémával hasonló hatást lehet elérni.

3.3.2.3. Architektúra független, hordozható

Napjaink hálózatait heterogén hardver- és szoftver architektúrájú számítógépek alkotják. A programok fejlesztését nagymértékben megkönnyíti, ha a forráskódból előállított program bármely architektúrán azonos módon fut. Ezen cél elérése végett a Java nyelv nem tartalmaz architektúra- vagy implementációfüggő elemeket. Ahhoz, hogy a lefordított program változtatás nélkül futtatható legyen különböző hardver architektúrákon, a fordítóprogram a programot nem egy konkrét processzor gépi kódjára, hanem egy képzeletbeli hardver - virtuális gép (virtual machine) - utasításrendszerére fordítja le. Az így létrejött közbenső, ún. Byte kódot töltjük le a célarchitektúrára, ahol a virtuális gépet megvalósító program értelmezi és végrehajtja azt. A hordozhatóság nem csak a virtuális gépi utasítások, hanem a nyelv mellet szabványosított rendszerkönyvtárak szintjén is jelentkezik, ezek a könyvtárak valósítják meg a legfontosabb, operációs rendszerekhez kötődő feladatokat, mint például a be- és kiviteli rendszert, vagy a programok grafikus kezelői felületét. Egy új architektúrán akkor futtathatók a Java programok, ha már implementálták rá a virtuális gépet, beleértve a rendszerkönyvtárakat is.

3.3.2.4. Interpretált és dinamikus

Az interpretált végrehajtás a fejlesztési ciklust nagymértékben felgyorsítja. Elég csak a megváltozott állományokat lefordítanunk, a program máris futtatható. Egyébként a Java támogatja a nagybani programozást, összetartozó osztályok egy csomagba (package) foghatók, egyszerre fordíthatók. Bár a virtuális gép elég ügyesen lett kitalálva és a fordítóprogram is mindent megtesz azért, hogy ezt a virtuális architektúrát a lehető legjobban kihasználja, de még így sem vetekedhet a gépi utasítások sebességével, becslések szerint 10-20-szor lassabban futnak, mint a gépi kódú megfelelőik. Persze a Java programok meghív-

hatnak gépi kódú (native) eljárásokat is, de ezzel elvesztjük az architektúra- függetlenség tulajdonságot. Ez történt az ACTROS-ban is, olyan C függvényeket hív meg, melyek csak ott léteznek, így a PC-n történő valós szimulációja jelenleg nem megoldott, így a bemutatásra kerülő program is csak az ACTROS-on volt tesztelhető, ill. a korai szakaszban a program által kezelt detektorértékeket egy későbbiekben bemutatandó, egyszerű random függvénygenerátor szolgáltatta.

3.3.2.5. Robusztus és biztonságos

Robusztus egy nyelv, ha megakadályozza vagy futás közben kiszűri a programozási hibákat, biztonságos, ha megakadályozza, hogy rosszindulatú programok kerüljenek a rendszerünkbe. A Java nyelv elkészítésekor mindkettőre igen nagy hangsúlyt fektettek.

3.3.2.6. Többszálú

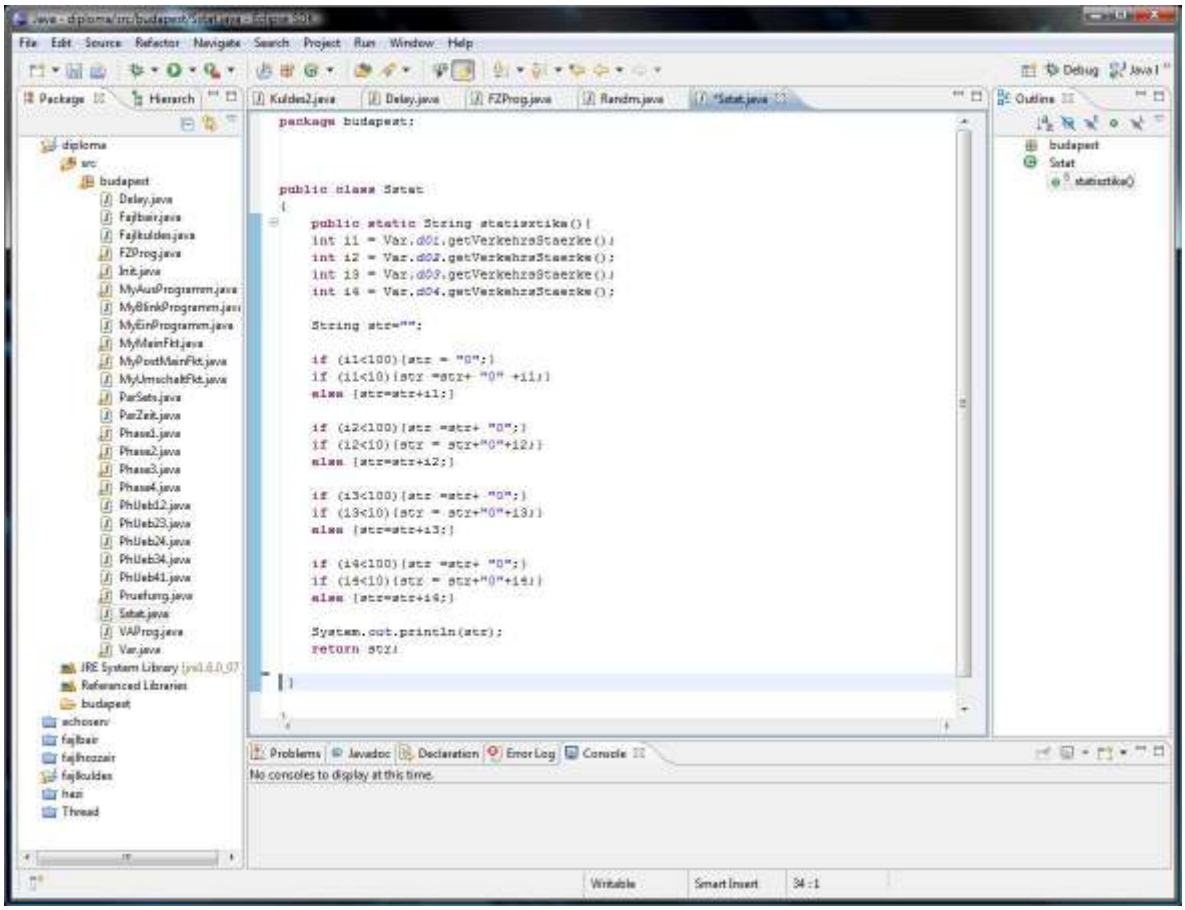
A programok jelentős része párhuzamosan végrehajtható részletekre - vezérlési szálakra - bontható. Így jobban kihasználható a számítógép központi egysége, a programok a külső - például felhasználói - eseményekre gyorsabban reagálhatnak. Az egymással kommunikáló, viszonylag laza kapcsolatban álló szálakra bontott feladat könnyebben áttekinthető, megvalósítható. A többszálú programozáshoz a Java nyelvi szinten biztosítja az automatikus kölcsönös kizárást: szinkronizált (synchronized) módszerek vagy utasítások, valamint a szálak létrehozásához, szinkronizációjához a rendszerkönyvtár tartalmaz egy ún. Thread osztályt. A Thread osztály a Hoare-féle feltételes változók (conditional variable) modelljét követi. Természetesen az összes, a rendszerkönyvtárakban definiált osztály használható többszálú programokból anélkül, hogy aggódnunk kellene az esetleges hibás működés miatt (thread-safeness).

3.3.2.7. Elosztott

Az elosztottság jelenleg két formában jelenik meg. Egyrészt az osztály-betöltő képes Java Byte kódot a hálózaton letölteni. Másrészt a rendszerkönyvtárak tartalmaznak a TCP/IP protokollcsalád alacsonyabb, szállítási (TCP és UDP), valamint magasabb, alkalmazói (pl. FTP, HTTP) szintű protokollok kezelésére szolgáló osztályokat.

3.3.3. Java fejlesztői környezet - Eclipse

A programok fejlesztésére egy ingyenes, internetről letölthető programozói felületet, az Eclipse-t használtam.
Az Eclipse egy integrált fejlesztői környezet (IDE) a Java programozás számára. Számos segédeszközt, fejlett szövegszerkesztőt, valamint saját hibakeresőt (debugger) tartalmaz, amelyek mind segítik a kód írását. A fejlesztői környezet az 5. számú ábrán látható.



5.ábra Eclipse fejlesztői felület

3.3.4. Objektumorientált programozás Java módra

Az objektumorientáltság manapság az egyik legkedveltebb programozási alapelv, amit a Java is követ. Igen fontos már az elején végigvenni azon építőköveket, módszereket, melyekből egy objektumorientált program felépül.

3.3.4.1. Absztrakt adattípusok létrehozása

A Java nyelvben csaknem minden objektum. Egy programban az import utasításokon kívül minden utasítás osztályok belsejében szerepelhet csupán, nincsenek globális változók, globális eljárások. A nyelv ugyan tartalmaz néhány egyszerű adattípust, de ezeken felül minden egyéb adatszerkezet vagy valamilyen osztályba tartozó objektum, vagy tömb, amely, ugyancsak speciális osztály. Mellesleg a nyelv tartalmaz a beépített típusokat becsomagoló osztályokat (wraper class) is: Boolean, Character, Double, Float, Integer.

3.3.4.2. Objektumok

Az objektumok olyan programösszetevők, amelyek szoros egységbe foglalják az állapotukat leíró belső adatszerkezetüket és a rajtuk értelmezhető műveleteket. Ebben az értelemben az egyes objektumok nagyon hasonlítanak egy adott típusba tartozó értékekhez. Egy int típusú érték is elrejti előlünk a belső reprezentációját, csak az előre definiált műveletek végezhetők rajta.
A Java-ban minden egyes objektumnak meghatározott típusa kell, hogy legyen, azaz valamelyik osztályba kell tartoznia. Az objektumokra változókon keresztül hivatkozhatunk.
Nagyon fontos, hogy a Java-ban minden változó hivatkozást tartalmaz csupán. Ezt szem előtt tartva érthető, hogy az értékadás (=) operátor nem készít másolatot, a bal oldali változó ugyanarra az objektumra hivatkozik majd, mint a jobb oldal. Ha valakinek mégis másolatra van szüksége, többnyire használhatja a csaknem minden osztályban megtalálható clone módszert. Hasonlóképpen az egyenlőség vizsgálatának C-ből ismert operátora (==) is a referenciák egyenlőségét ellenőrzi, az objektumok strukturális ellenőrzésére az equal módszer való, már amennyiben létezik ilyen. A referenciák sérthetetlenek, semmiféle művelet, konverzió nem végezhető rajtuk. Új objektumokat a new utasítással lehet létrehozni, megszüntetni, törölni viszont nem kell. A Java virtuális gép, amely a programokat futtatja, tartalmaz egy szemétgyűjtő (garbage collection, gc) - jelenleg az ún. mark-and-sweep típusú - algoritmust, amely általában a háttérben, a CPU szabadidejében, a programmal aszinkron futva összegyűjti a már nem hivatkozott objektumokat és azok tárterületét felszabadítja, hogy a new újra felhasználhassa majd. A szemétgyűjtés teljesen rejtetten, automatikusan történik, a programozónak általában nem is kell törődnie a szemétgyűjtéssel.

Ha már vannak objektumaink, nézzük meg, hogyan tudunk hatni rájuk. Az egyes objektumok módszereit az
Obj.method (parameter, parameter, ...)
szintaxissal hívhatjuk meg. Az objektumorientált terminológia ezt gyakran üzenetküldésnek (message passing) nevezi, ahol az utasítás "megkéri" az objektumot, hogy hajtsa végre az ’x’ műveletet az ’y’ paraméterrel. Az ilyen módszerhívások vezethetnek az objektum belső állapotának, azaz belső változói értékének megváltozásához is, de gyakran csak egy belső változó értékének lekérdezése a cél.
Bár a tiszta objektumorientált elvek szerint egy objektum belső állapotváltozóinak értékéhez csak módszerein keresztül lehetne hozzáférni, a Java - a C++-hoz hasonlóan - megengedi a programozónak, hogy bizonyos változókat közvetlenül is elérhetővé tegyen.

3.3.4.3. Osztályok

A Java programozók osztályokat a
class ClassName { az osztály törzse }
programszerkezettel definiálhatnak, ahol a ClassName az újonnan definiált osztály neve lesz, az objektumok belső állapotát és viselkedését pedig a kapcsos zárójelek közötti programrészlet írja le. A class alapszó előtt, illetve az osztály neve után opcionálisan állhatnak még különböző módosítók, de ezekről majd később.

3.3.4.4. Egyedváltozók

Az osztály törzsében szerepelnek azok a változó-deklarációk, amelyek az egyes egyedek belső adatszerkezetét valósítják meg. Ezeket gyakran egyedváltozóknak (instance variable, member variable) nevezzük, jelezve, hogy az osztály minden egyede ezekből a változókból saját készlettel rendelkezik.

3.3.4.5. Módszerek

A típus által értelmezett műveleteket a módszerek testesítik meg. Az egyes módszereknek lehet visszatérési értéke – amennyiben nincs, azt a kötelező void jelzi -, illetve adott típusú és számú bemenő paramétere. A módszer nevét, visszatérési értékének és paramétereinek számát, típusát a módszer lenyomatának (signature) nevezzük.
A módszerek meghívásánál a lenyomatban definiált formális paraméterek aktuális értéket kapnak. A Java-ban minden paraméterátadás érték szerint történik, azaz a paraméter helyén szereplő értékről másolat készül, a módszer ezt a másolatot látja, használja. Persze, ha a másolat módosul, az az eredeti értéket nem érinti. Kicsit becsapós a "mindig érték szerinti

paraméterátadás" szabálya. Amennyiben a paraméter nem valamelyik beépített, egyszerű típusba tartozik, úgy az átadásnál a referenciáról készül másolat, azaz a C++ fogalmai szerint ilyenkor referencia szerinti átadás történik, tehát a hivatkozott objektum a módszer belsejében megváltoztatható. A visszatérési érték sem feltétlenül egy egyszerű típusba tartozó érték, hanem lehet objektum referencia is.
Az osztály belsejében azonos névvel, de egyébként különböző lenyomattal több módszert definiálhatunk (többes jelentés, overloading), ilyenkor a fordító a módszer hívásánál a paraméterek számából és típusából dönti el, hogy melyik módszert akarjuk meghívni, de a kiválasztásnál a módszer visszatérési értékének típusát a fordító nem veszi figyelembe.
A módszerek törzsébe utasításokat írhatunk. A kifejezésekben, értékadások bal oldalán használhatjuk az eljárás paramétereit, lokális változóit, de az objektum egyedváltozóinak értékét is. Amennyiben a saját objektumra akarunk hivatkozni, úgy használhatjuk a this szimbólumot.
A módszerek fejében használhatjuk a native módosítót, ez a fordítóprogramnak azt jelenti, hogy a módszer törzsét nem Java-ban írtuk meg, hanem az aktuális platformtól függő módon, pl. C-ben. Az ilyen módszerek paraméterátadási módja, esetleg az elnevezési konvenciói az aktuális platformtól függhetnek. Természetesen ilyenkor az osztálydefinícióban a módszernek csak a lenyomatát kell megadni. Az ilyen módszereket használó programok sajnos elvesztik a Jáva programok architektúra-függetlenségét (lásd ACTROS), de egyébként a native módszerek mind a láthatóság, mind az öröklődés szempontjából a többi módszerrel teljesen azonos módon viselkednek.

3.3.4.6. Konstruktorok

Az osztályban definiált módszerek közül kitűnnek az ún. konstruktorok (constructor), amelyek szerepe az objektumok létrehozása, belső állapotuk kezdeti értékének beállítása. Minden konstruktor neve megegyezik az osztálya nevével, visszatérési értéke pedig nincs. Természetesen a konstruktor is lehet többes jelentésű módszer. A konstruktorokat a new utasítás kiadása hívja meg, a konstruktorok közül a new-nál megadott paraméterek száma és típusa szerint választunk. Kitüntetett szerepű a paraméter nélküli, ún. alap (default) konstruktor, ha mi nem definiáltunk ilyet, a Java fordító automatikusan készít egyet az osztályhoz. Az így generált konstruktor a helyfoglaláson kívül nem csinál semmit.
A szemétgyűjtés miatt a C++-ból ismert destruktorok itt nem léteznek, viszont ha egy objektum végleges megszűntetése előtt valami rendrakó tevékenység végrehajtására lenne szükség, akkor ezt írjuk egy void finalize() lenyomatú módszer törzsébe, a Java virtuális gép biztosan végrehajtja, mielőtt az objektum területét a szemétgyűjtő felszabadítaná, azonban az aszinkron szemétgyűjtés miatt azt soha nem tudhatjuk biztosan, hogy ez mikor következik be.

3.3.4.7. Osztályváltozók és módszerek

Amíg az egyedváltozókból minden példány saját készlettel rendelkezik, addig az ún. osztály-, vagy statikus (static) változókból osztályonként csak egy van. Természetesen mivel ezek nem egy példányhoz tartoznak, ezért a hozzáféréshez sincs szükség egy konkrét objektumra, az osztály nevével is hivatkozhatunk rájuk.
A csak statikus változókat használó módszerek a statikus, avagy osztálymódszerek.

3.3.4.8. Tömbök

A Java nyelv tömbjei is objektumok, ha kissé speciálisak is. Deklarálásuk nem csak a C-szerű szintaxissal megengedett, hanem tehetjük a szögletes zárójeleket a "logikus helyére" is, azaz a következő két deklaráció ekvivalens:

```
int a[] = new int[10];
int[] a = new int[10];
```

Mint a többi objektumnál, a név itt is csak egy referencia, a tömb deklarációja után azt a new paranccsal létre is kell hozni. Lényeges, hogy a virtuális gép ellenőrzi, hogy ne nyúljunk túl a tömb határán: kivétel(erről később) keletkezik, ha mégis megtesszük. Minden tömb objektumnak van egy length nevű egyedváltozója, amely a tömb aktuális méretét adja vissza. Ez azért is fontos, mert a program futása során ugyanaz a tömbváltozó más és más méretű tömbökre hivatkozhat.
Természetesen nem csak a beépített, egyszerű típusokból képezhetünk tömböket, hanem tetszőleges típusból, beleértve egyéb tömböket. A Java többdimenziós tömbjei is mint tömbök tömbje jönnek létre. Még egy érdekesség: a referenciák használatának következménye, hogy nem csak "négyszögletes" kétdimenziós tömböt lehet készíteni, de olyat is, ahol az egyes sorok nem azonos hosszúak, például háromszög alakú tömböt.

3.3.4.9. Szövegek

A Java-ban a szövegek (String) is teljes rangú objektumok. A szövegek gyakori módosításának hatékonyabb elvégzésére használhatjuk a StringBuffer osztályt is.

3.3.4.10. Csomagok

Összetartozó osztályokat a csomag (package) segítségével a programozók egyetlen fordítási egységgé foghatnak össze. Ezzel osztály-könyvtárakat építhetünk, és nagy szerep jut a láthatóság szabályozásánál is. A csomagot, amennyiben használjuk egyáltalán, a forrás első nem megjegyzés sorában kell megneveznünk, pl.:

```
package budapest;
```

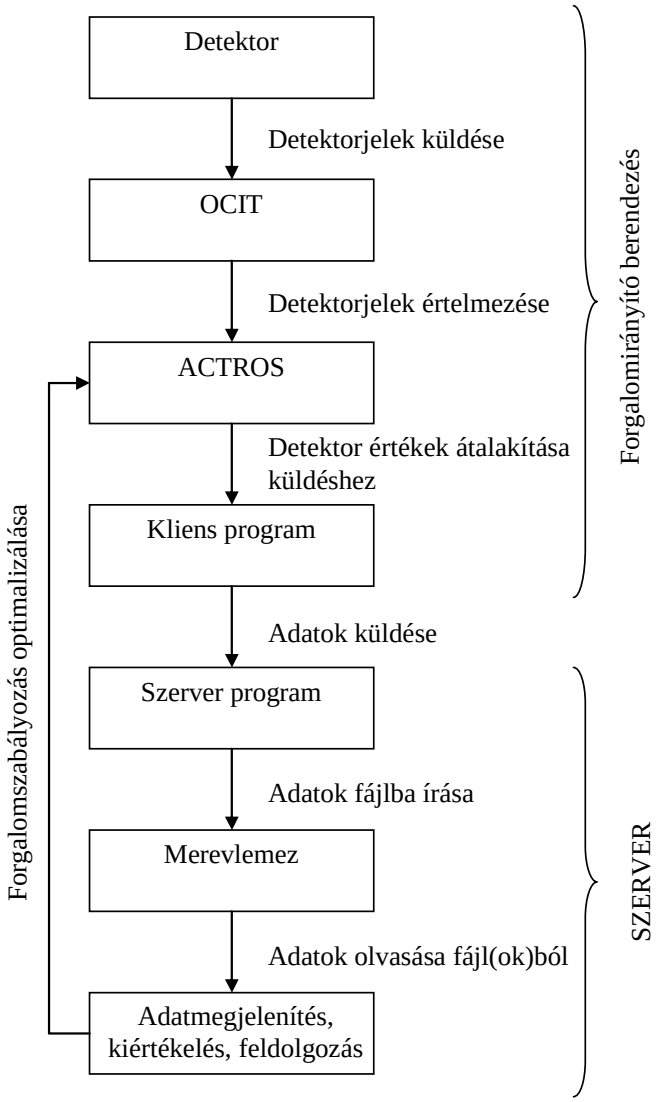
A csomagoknak programozási konvenciók szerint általában több komponensű, ponttal elválasztott nevet adunk, a fordító a névnek megfelelő könyvtár-hierarchiába helyezi el a lefordított osztályokat. Létezik egy név nélküli csomag arra az esetre, ha nem adtuk meg a package utasítást.

Az így létrehozott csomagokból lehet a korábban megismert import utasítással egy vagy több osztályt átvenni, használni.

4. A kifejlesztésre kerülő Java programok bemutatása, tesztelése

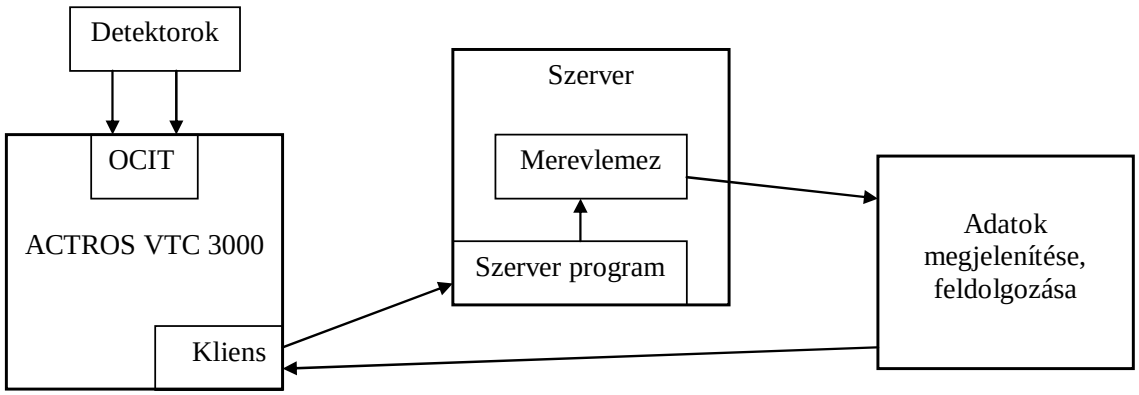
4.1. A programfunkciók bemutatása

A tanulmány célja az ACTROS szoftverének kibővítése, amely megoldja a járműérzékelő detektorok által fogott jelek feldolgozását, internet alapú csomagonként ezen adatokat elküldi egy központi szervernek, ahol azok tárolására kerül sor, valamint a szerveroldal statisztikai feldolgozásra is lehetőséget nyújt. A kliensoldalon, ami jelen esetben maga az ACTROS berendezés, a bemeneti jeleket, azaz a detektorok jeleit egy OCIT kártyán keresztül az ACTROS-hoz kapcsolt detektorok szolgáltatják. A rendszer hatásvázlata a 6. ábrán látható.



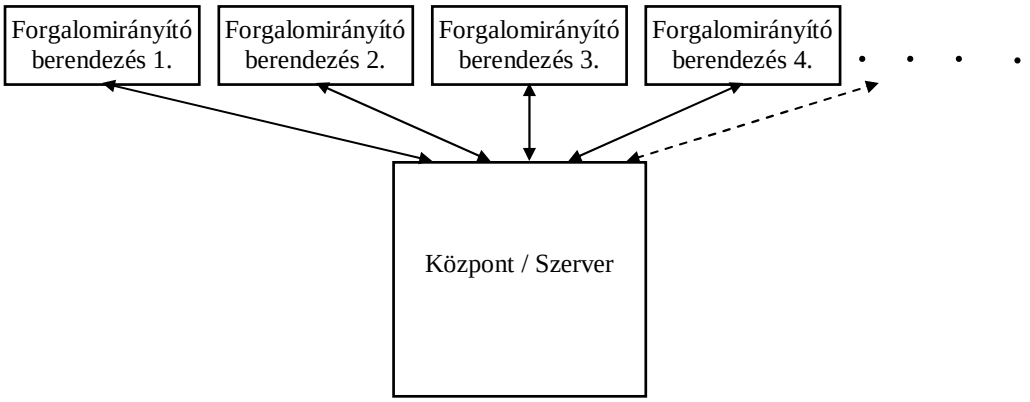
6.ábra Adatküldő rendszer hatásvázlata

A kliens oldalon az forgalomirányító programot módosítani szükséges, egyrészt meg kell oldani a detektoroktól érkező jelek észlelését, feldolgozását, ezenkívül indítani kell a program indulásakor egy kliensprogramot, ami az eredeti programmal párhuzamosan futva, interneten keresztül az adott IP-című szerver felé továbbítja az adatokat.
A kliensoldali alkalmazások bemutatását a 4.2. fejezet tartalmazza.
A szerveroldali program fogadja a csomagokat, s az adott kliens IP-címe alapján elmenti a szükséges adatokat a merevlemezen. Jogosan merülhet fel a kérdés, miért van szükség szerverre és adatküldés funkcióra, miért nem tárolhatók az információk közvetlenül az ACTROS-on?
Ennek két oka is van: egyrészt az ACTROS ROM memóriája nem alkalmas nagyobb mennyiségű adat tárolására, másrészt egy központi szerveren megoldható, a több ACTROS-t, több forgalomirányítási egységet érintő statisztikai jellegű lekérdezés.
A szerver-oldali programokat a 4.3. fejezet mutatja be.
Az elkészült statisztikai lekérdezéseket támogató program a 4.4. fejezetben szintén bemutatásra kerül. Az alkalmazás a szerver oldalon futtatható, eredményként pedig grafikus statisztikai ábrákat jelenít meg a felhasználó számára. A rendszer felépítését az alábbi, 7. számú ábra szemlélteti:



7.ábra Az adatküldő rendszer felépítése

Teljes forgalomirányítási rendszerben természetesen a szerverhez több forgalomirányító berendezés kapcsolódik - mint az a következő oldal 8. számú ábráján látható - tovább növelve a rendszer optimalizáló hatásfokát.



8.ábra Teljes forgalomirányítási rendszer

4.2. ACTROS – kliens

Ez a program, mint az összes Java alapú program osztályokból (class) áll. Ilyen osztályok az inicializáló osztály (Init.class), a változók osztálya (Var.class), a program-, és rendszer-program osztályok, a fázis-, és fázisátmenet osztályok és a berendezés-tesztelő osztály.
A berendezés bekapcsolásakor a program inicializálással indul, ha ez sikeresen, hiba nélkül fut le, elindul a berendezés. Ezután a főprogram és az egyéb programok ciklikusan futnak.
Az inicializálás 10 másodpercet vesz igénybe, ezután sárga-villogó program következik, ezt pedig a kikapcsoláskor utoljára futó program követi.
Inicializálás során az Init.java osztály main metódusa hajtódik végre. Ez futása során olyan visszatérési érték nélküli metódusokat hív meg, melyek a működéshez szükséges kezdeti értékeket állítja be. Itt két helyen történt változtatás:
protected void initialisiereDet() metódus - itt megadásra kerültek a d01..d04 OCIT kártyán keresztül a berendezéshez kapcsolt detektorok kezdeti értékei:

```
Var.d01  = new Detektor( Var.tk1, "OCIT_DET_01", 200, 10,
           Detektor.KEINE_UEBERWACHUNG, 1, null, 0 ,0 );
Var.d02  = new Detektor( Var.tk1, "OCIT_DET_02", 200, 10,
           Detektor.KEINE_UEBERWACHUNG, 2, null, 0, 0 );
Var.d03  = new Detektor( Var.tk1, "OCIT_DET_03", 200, 10,
           Detektor.KEINE_UEBERWACHUNG, 3, null, 0 ,0 );
Var.d04  = new Detektor( Var.tk1,"OCIT_DET_04", 200, 10,
           Detektor.KEINE_UEBERWACHUNG, 4, null, 0, 0 );
```

protected void initialisiereHW() metódus – itt a detektorértékek átadásához szükséges adatátvieli csatorna (Kanal) beállítása történt:

```
IoKanal io1_1 = new IoKanal (  Var.d01  , io1,  1);
IoKanal io1_2 = new IoKanal (  Var.d02  , io1,  2);
IoKanal io1_3 = new IoKanal (  Var.d03  , io1,  3);
```

```
IoKanal io1_4 = new IoKanal (  Var.d04  , io1,  4);
IoKanal io1_9 = new IoKanal (  Var.t21  , io1,  9);
IoKanal io1_10= new IoKanal (  Var.t22  , io1, 10);
IoKanal io1_17= new IoKanal (  Var.sk21 , io1, 17);
IoKanal io1_18= new IoKanal (  Var.sk22 , io1, 18);
```

Az ACTROS kódjában az angol és német nyelvű részek felváltva követik egymást, mivel a programozás nyelve az angol, de a német fejlesztők amit lehetett, azt anyanyelven írtak, ezzel kissé bonyolítva a további fejlesztést. Miután ezen metódusok lefutnak, visszatérve a main-be, értéket kell adni a program kódjában már meglévő anzZykl nevű integernek. Ezzel tudjuk beállítani, hogy a detektorértékeket hány ciklusra visszamenőleg nézze a program. A ciklus itt a berendezés saját ciklushosszát jelenti, ami 0,5 másodperc. Az anzZykl értékét ennek megfelelően 120 ra lett beállítva, így 0,5 másodpercenként frissítődik az elmúlt 1 percről kapott információ. Ezt az értéket minden egyes d01..d04 detektorhoz hozzá kell rendelni, melynek módja:

```
Var.d01.addVerkehrsStaerke(anzZykl);
Var.d02.addVerkehrsStaerke(anzZykl);
Var.d03.addVerkehrsStaerke(anzZykl);
Var.d04.addVerkehrsStaerke(anzZykl);
```

Ezek után indítódik a kliens program. Ezt nem lehet egyszerűen indítani az inicializálás közben, mivel ekkor megakadna a folyamat a kliens program várakozásával. Ehhez egy külön szálat (thread) kell létrehozni a kliens számára, így ez párhuzamosan futhat az alapfunkciókkal.
A Jáva nyelvben a szálak számára a java.lang csomag tartalmaz egy külön osztályt, a Thread-et. Minden párhuzamos tevékenység vagy egy Thread, vagy ebből leszármazott osztály egy példánya. Szálakat tehát úgy hozunk létre, hogy leszármaztatunk a Thread osztályból egy saját osztályt. Persze meg kell adnunk, hogy a szálunk milyen tevékenységet hajtson végre, ezt a Thread-ből örökölt run módszerben teheljük meg.
A kliens szál létrehozása, maj elindítása:

```
Fajlkuldes cli = new Fajlkuldes();
cli.start();
```

Innentől elkezdődik a kliens szál futása. A számára szükséges adatok az Sstat.java-ban jönek létre, ami a detektorértékek lekérdezését végzi. Ez az osztály minden működési programból – FZProg, MyAusProgramm, MyBlinkProgramm, MyEinProgramm - meghívódik, így minden program során működik az adatfolyam, A meghívés mindig a ProgrammFunktion() metóduba kerül, mivel ez minden cikluban lefut:

```
Var.data=Sstat.statisztika();
```

Meghívásakor nincs szüksége konstruktorra, ellenben van visszatérési értéke. Az Sstat.statisztika() metódus négy integert állít elő (i1..i4), melyek a már beállított anzZykl alapján az elmúlt egy perc alatt a detektorok felett áthaladt járművek számát kapják meg:

```
int i1 = Var.d01.getVerkehrsStaerke();
int i2 = Var.d02.getVerkehrsStaerke();
int i3 = Var.d03.getVerkehrsStaerke();
int i4 = Var.d04.getVerkehrsStaerke();
```

Ezen i1..i4 értékek kerülnek majd küldésre, illetve tárolásra. Ezek az integerek az egyszerűbb küldés érdekében 1 db str nevű stringbe lettek fűzve:

```
String str="";
```

A string pozitívuma, hogy kezelésüknél a nullákat nem hagyja el a program. Itt arra kell gondolni, hogy minden detektorérték három karakteren foglal helyet, és például integer esetén, a 7 érték kerülne trolásra, nem pedig a 007. Ez a karaktersor eltolódásához vezet. A string-ben való mentés tehát nem vágja le a nullákat, de megoldandó feladat, hogy egyáltalán legyenek. Erre egy dupla if ciklus szolgál detektorértékenként, ahonnan az első detektorhoz tartozó részlet:

```
if (i1<100){str = "0";}
if (i1<10){str =str+ "0" +i1;}
else {str=str+i1;}
```

A metódus a fenti műveletek elvégzése során egy 12 karakterből álló str karaktersorozatot hoz létre, amely egyben a metódus visszatérési értéke:

```
return str;
```

Ez azt jelenti, hogy futása után a meghívó oldalon ezt adja vissza eredményül, tehát a fent említett Var.*data*=Sstat.*statisztika*();egyenlőség alapján a Var.data az str értékét kapja meg. A Var.data egy osztályváltozó.
Az egyes osztályok által használt változóknak tehát 2 csoportja van, egyedváltozók, és osztályváltozók.
A teljes program számára számára fontos összes változó a Var osztályban van definiálva. Ahhoz, hogy más osztályok ezekhez hozzáférhessenek, minden egyes változót a „puplic static” jelzővel hozunk létre, példaként az újonnan létrehozott detektorok változói, és a fenti példából a data létrehozása:

```
public static Detektor d01;
```

```
public static Detektor d02;
public static Detektor d03;
public static Detektor d04;
public static String data="";
```

Itt a public egy hozzáférést szabályozó módosító. A programozó megadhatja, hogy az általa definiált egyes osztályok, illetve az osztályok változói, módszerei milyen körben használhatók. Erre a célra az ún. hozzáférést szabályozó (access specifier) módosítók használatosak, melynek 3 fajtája lehet:
nyilvános (public), védett (protected), baráti (friendly). Ez utóbbi az alapértelmezés, ha nem adjuk meg a hozzáférés módját, akkor ez mindig baráti. A második típusról akkor beszélhetünk, ha az egyes osztályok között öröklődés is van, és ez a protected (védett) alapszó. A védettnek definiált változók és módszerek csak az osztályban, annak valamelyik leszármazottjában, vagy a csomagban láthatók, illetve létezik ennek egy módosított változata, a privát védett (private protected) változók az előzőhöz képest a csomagokban sem látszanak.
Végül a static, ilyen a példabeli d01 változó is: publikus, minden osztály számára hozzáférhető.
A másik alapszó a static. Az egyed-, és osztályváltozókról már volt szó: amíg az egyedváltozókból minden példány saját készlettel rendelkezik, addig az ún. osztály- vagy statikus (static) változókból osztályonként csak egy van. Természetesen mivel ezek nem egy példányhoz tartoznak, ezért a hozzáféréshez sincs szükség egy konkrét objektumra, az osztály nevével is hivatkozhatunk rájuk. Ha tehát más osztályból szeretnénk elérni az adott változót, akkor az osztálynév után, amit mindig nagy betűvel kezdünk, ponttal elválasztva írjuk a kívánt változót. Egy másik megoldás, amikor importáljuk az adott osztályt a kód elején, ekkor nem kell a változó elá az osztálya, erről a későbbiekben bővebben is lesz szó.
A változók deklarálásának harmadik tagja a detektor illetve a string a fenti péndákban, ezek változó típusok, majd ezt követi a változó neve.
Az egyes programrészek működésének ismertetésekor nem mindig szerepel a teljes kód, ellenben a mellékletekben megtalálhatók.

Az adatok küldése a Fajlkuldes.java osztályban történik, melynek kódja a xy.számú mellékletben található. Az egyes sorok jelentése:

```
package budapest;
```

Egy programozási feladat megoldása során fontos az áttekinthető és könnyen módosítható kód előállítása. Ennek megvalósításához a programot szerkezetileg megfelelően tagolni kell. A megfelelő tagolás célja az, hogy olyan programegységek jöjjenek létre, amelyek valamilyen módon összefüggő elemeket tartalmaznak, viszont egymáshoz csak lazán kap-

csolódnak. A Java nyelvben ilyen célt szolgálnak a csomagok (package). Számunkra azonban nincs jelentősége, az Actros minden egyes oszálya egyetlen budapest nevű csomagba lett összefogva.

```
import java.io.*;
import java.net.*;
```

A programok import utasításokkal kezdődhetnek, felsorolva a programban felhasznált könyvtárak nevét. Ez nem kötelező, az egyes könyvtári elemekre való hivatkozásnál is megadhatjuk a könyvtár nevét, mint az a fenti Var.data-nál látszott. Az import a kapcsolatszerkesztőnek, betöltőnek szól, már lefordított kódú könyvtárakra hivatkozik. Importálásnál csomagokból (*package*) osztályokat (*class*) importálunk, azaz a programunkban felhasználhatóvá tesszük. A csillag az adott csomag összes osztályát jelenti, jelen esetben a java.io.* egy olyan java könyvtár mely fájlok, illetve általánosan bemenetek és kimenetek kezelésére alkalmas.
Bemenet: adatokat lehet egymás után olvasni róla, például konzol, fájl, hálózati kapcsolat, adatbázis egy rekordja, stb.
Kimenet: adatokat lehetegymás után kiírni rá, például képernyő, fájl, nyomtató, hálózati kapcsolat, adatbázis egy rekordja, stb.
Bemenet és kimenet kezelésének szintjei:

- csatorna (stream)
- csatornaobjektum

Csatornák iránya szerint:

- Bemeneti: InputStream, FileReader
- Kimeneti: OutputStream, FileWriter

A második importálás a java hálózati könyvtárára (java.net) mutat, ami a TCP/IP protokollcsaládra épül - ez a szorosan vett Internet hálózat alapja. Esetünkben a szállítási szintű protokollok közül a TCP kapcsolatorientált protokoll kerül használatra, de a java.net könyvtár a datagrammokra épülő UDP protokollt is támogatja. Ezeken túl jelenleg az alkalmazói szintű protokollok közül csak a Web alapját jelentő HTTP (Hypertext Transfer Protocol) protokoll közvetlen használatát támogatja.
Az importálások után az osztálydeklarációnak kell következnie, mivel a Java programnyelvben minden változó, minden utasítás csak osztályok törzsében szerepelhet:

```
public class Fajlkuldes extends Thread
```

Az extends alapszó az öröklődés kifejezésére szolgál, jelentése bővíteni, itt tehát létrejön a Fajlkuldes osztaly, mely „örökli” a Thread minden „tudását”, ami alatt a szálak kezelését értjük.

```
public void run()
```

A kliens kapcsolata a szerverrel

Mivel a szerver és a kliens között folyamatos kapcsolattartás szükséges, ezért az összeköttetés-alapú kapcsolatot kell kiépíteni közzöttük. Ez TCP (Transmission Controll Protocoll) transzport protokoll segítségével valósítható meg. Egy hálózati csatlakozóhoz több különböző kommunikációs végpontot lehet rendelni a TCP protokoll segítségével. Ezeket nevezik TCP portoknak. Minden egyes folyamat lefoglalhat magának egy-egy portot, és azon adatokat fogadhat és küldhet. A TCP portazonosító egy 16 bites szám, ami azt jelenti, hogy egy hálózati csatlakozónak maximum 65535 darab portja lehet. Egyes portokat ezzel a portazonosítóval címzünk meg. Egy számítógépnek természetesen több hálózati csatlakozója lehet. Minden alkalmazás lefoglal magának egy ilyen portot. Vannak szabvány portazonosítók, ami azt jelenti, hogy egy adott számú port minden gépen ugyanazt a szolgáltatást jelenti (pl. 21-FTP, 23-telnet, 80-WWW, stb.)

Egy összeköttetés-alapú kapcsolat létrehozásának kezdetén, a kliensen és a szerveren az egymással kommunikálni kívánó folyamatok lefoglalnak egy-egy kommunikációs végpontot, azaz TCP portot. Ezeket a portokat összekötve adatokat tudnak küldeni egymásnak. Miután a résztvevők úgy döntenek, hogy befejezik a kommunikációt, szabályosan le kell bontaniuk a kommunikáló felek között kialakított kommunikációs csatornát.

Egy kliens alkalmazás az alábbi lépések végrehajtásával vehet igénybe egy a szerver által kínált szolgáltatást összeköttetés-alapú protokoll alkalmazásával (ezáltal válik a szolgáltatást nyújtó szerver kliensévé):

1. Az alkalmazás első lépésként lefoglal egy még nem használt TCP-portot magának.
2. Az alkalmazás ezen a lefoglalt porton keresztül egy hálózati összeköttetést hoz létre saját maga és a szerver kommunikációs végpontja (TCP-portja) között, azaz kapcsolódik a szerverhez.
3. Ha sikerült az összeköttetést létrehozni, akkor ezen az összeköttetésen keresztül az alkalmazás (kliens) adatokat tud küldeni a szervernek, és adatokat tud fogadni a szervertől. Igénybe tudja venni a szerver által nyújtott szolgáltatásokat.
4. Miután nincs többé szüksége a szolgáltatás igénybevételére, a kliens szabályosan lebontja a szerverrel az a. pontban felépített kapcsolatot. Ha a kapcsolat lebontása után a kliensnek nincs szüksége a lefoglalt portra, akkor azt felszabadítja, és visszaadja az operációs rendszernek.

Összeköttetés-alapú kliens megvalósítása Javában

Egy hálózati összeköttetésnek mind a kliens-, mind pedig a szerveroldali részét a java.net.Socket osztályba tartozó objektumokkal reprezentálhatjuk. Egy összeköttetés-alapú kliens alkalmazás a Socket osztály segítségével építhet fel egy összeköttetést a hálózaton keresztül egy szerverrel. A felépített összeköttetés kliensoldali jellemzőit kijelölheti maga

a kliensalkalmazás, az összeköttetést létrehozó konstruktor művelet paramétereiben. Ha pedig a kliensalkalmazás ezt nem teszi meg, akkor az operációs rendszer TCP/IP szoftvere határozza meg azt.
Egy kliens-szerver összeköttetést reprezentáló java.net.Socket osztálynak több konstruktora van. A létrehozandó összeköttetésről rendelkezésre álló információk alapján a programozónak kell eldöntenie, hogy melyiket akarja használni. A konstruktor paraméterezése a következőképpen történhet:
Socket(InetAddress address, int port, InetAddress localAddr, int localPort)
Az első két paraméterben kell megadni az összeköttetés szerveroldali jellemzőit: annak a szervernek az Internet-címét és TCP-port azonosítóját, amellyel a kliens fel akarja venni a kapcsolatot. A harmadik és negyedik paraméterben adhatjuk meg annak a helyi hálózati csatlakozónak az Internet-címét, amelyikhez a létrehozott összeköttetés helyi végpontját kötni akarjuk, és ott jelölhetjük ki a helyi kommunikációs végpont TCP-port azonosítóját is. A harmadik paraméterben null, a negyedikben 0 értéket megadva az adott jellemző kiválasztását az operációs rendszerre bízhatjuk. A konstruktor művelet végrehajtásának sikertelenségét a Java-futtató rendszer egy kivétel generálásával jelzi.
Miután egy hálózati összeköttetés felépült, rajta adatok küldhetők a kommunikációs partnerhez.A hálózati adatátvitelt a kommunikációs végpont getInputStream() és getOutputStream() metódusa által visszaadott I/O csatornán keresztül végezhetjük (műveletek sikertelen végrehajtása esetén java.io.IOException kivételt generálhatnak).
Egy kétirányú kommunikációs kapcsolatot biztosító összeköttetésen keresztül az adatátvitel az összeköttetésre ráültetett csatornákon keresztül történhet: egy kimeneti csatorna a ráírt adatokat eljuttatja a kommunikációs partnerhez, egy bemeneti csatornán pedig a kommunikációs partnertől érkező adatokhoz lehet hozzáférni. Mivel egy Java I/O csatorna egyirányú adatátviteli útvonalat biztosít, ezért egy kétirányú kommunikációs adatátviteli vonal szolgáltatásainak az igénybevételéhez két csatorna kell: a kliens és a szerver közötti adatátvitel mindkét irányát egy-egy önálló Java I/O csatornára képezik le. A Socket objektum getInputStream() és getOutputStream() metódusaival kaphatjuk meg rendre az összeköttetésen a kommunikációs partnertől jövő adatokat tartalmazó, illetve a kommunikációs partner felé adatokat közvetítő csatornákat.

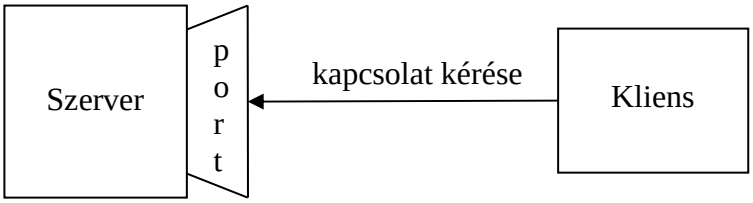
A kapcsolat osztály

A kliens program a kapcsolatot a szerverrel akkor veszi fel, mikor az ACTROS inicializál. A kapcsolatot a megadott szerver címre, és portra a Kapcsolat osztály építi ki a konstruktorában. A kommunikáció a kliens és a szerver között objektumokkal történik. A konstruktor létrehozza a hálózati kapcsolatot a szerverrel, elküldi a bejelentkező csomagot (csomag-ról a 6.6.1. fejezetben lehet olvasni). Majd megvárja a szerver válaszát, ami kétféle lehet. Vagy sikerült bejelentkezni a szerverre, vagy pedig sikertelen volt a bejelentkezés. A siker-

telen bejelentkezés oka eldönthető a szervertől kapott csomag kódjából. Ilyenkor a beépített metódus vár a megadott ideig, majd a kliens újra próbálja a csatlakozást.
Ha sikeres volt a bejelentkezés, akkor a Kapcsolat osztály futása szálként folytatódik tovább. Ekkor a kapcsolattartás a szerverrel a run() metódusban van megvalósítva. A run() metódus addig vár, amíg nem kap egy objektumot az adatátviteli csatornán. Mikor kap egy objektumot, akkor instanceof operátorral megnézi, hogy az objektum csomag osztály egy példánya, vagy java.util.Vector osztály példánya. Ha csomag osztály egy példánya, jelen program-rendszernél mindig, akkor a csomag.kod alapján meghívja a megfelelő objektum megfelelő metódusát. Mikor meghívta a megfelelő metódus(oka)t, akkor újra a ciklus elejére kerül, ahol várja a következő objektumot a szervertől.
A ciklus egészen addig fut, amíg a hálózati kapcsolat meg nem szűnik. Ez két okból történhet meg. Az egyik amikor a kliens szabályosan kilép a szerverről. Ez úgy történik, hogy küld egy 99-es kódú csomagot a szervernek. Erre a szerver egy 199-es kódú csomagot küld válaszként és szabályosan lezárja a hálózati kapcsolatát a klienssel. Mikor a kapcsolat objektum megkapja a 199-es kódú csomagot, akkor szabályosan lezárja a megnyitott hálózati kapcsolatát, majd pedig run() metódusban lévő ciklus befejezi a futását. Másik oka a kapcsolat megszűnésének, amikor a kapcsolatban valamilyen hiba lép fel, ami egy kivételt vált ki, ami a kivétel kezelésekor befejezteti a szál futását. A csomag, és a Vector összeállítása mindig a metódusokat hívó programrész feladata.
A Fajlkuldes szál, start parancsra való indításakor tehát mindig a run() metódus hívódik meg, melynek tartalma:

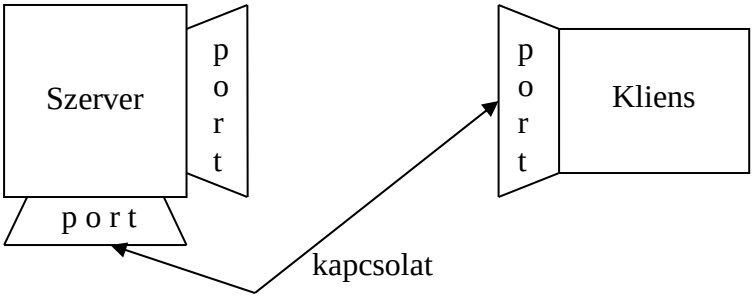
```
Socket SOCKET = null;
try{
	SOCKET = new Socket("x.x.x.x",7200);
}
catch (UnknownHostException e) {
	e.printStackTrace();
}
catch (IOException e) {
	try {
		Delay.ujra();
	}
	catch (InterruptedException e1) {
		e1.printStackTrace();
	}
	catch (IOException e1) {
		e1.printStackTrace();
	}
	e.printStackTrace();
}
```

Első lépésként konstruktorával létrehozzuk a socket-et. A konstruktorok (constructor) osztályban definiált módszerek, szerepük az objektumok létrehozása, belső állapotuk kezdeti értékének beállítása. Minden konstruktor neve megegyezik az osztálya nevével, visszatérési értéke pedig nincs. A konstruktorokat a *new* utasítás kiadása hívja meg, a konstruktorok közül a new-nál megadott paraméterek száma és típusa szerint választunk. Kitüntetett szerepű a paraméter nélküli, ún. alap (default) konstruktor, ha mi nem definiáltunk ilyet, a Java fordító automatikusan készít egyet az osztályhoz. Az így generált konstruktor a helyfoglaláson kívül nem csinál semmit.
A socketek kizárólag konstruktorral hozhatók létre, mivel meg kell adni a szerver host nevét vagy IP címét, illetve a szerver által figyelt port számát. A socket, definíció szerint végpontja egy kétvégű kommunikációs hálózatnak, amin két program fut. A socket egy port számhoz van kötve, ezért a TCP réteg azonosítani tudja a kérést, amit ahhoz kértek, hogy elküldhessék az adatot. Amíg a szerver várakozik, figyeli a socket-et, hogy van-e felkérés egy klienstől a kapcsolódásra, addig a kliens oldalon, mivel ismeri annak a számítógépnek a host-nevét, amelyiken a szerver fut, és annak a port-nak a számát, amelyiken a szerver figyel, kapcsolatra való felkérést küld, a szerver gép megadott port-jára, amely a 9. ábrán látható.



9.ábra A kliens elküldi a kapcsolatra való felkérést

Ha minden jól megy, a szerver elfogadja a kapcsolatot. Az elfogadás során a szerver egy új socket porthoz kapcsolódik (10. ábra). Azért kell egy új socket (és természetesen egy különböző port szám is), mert csak így folytatódhat a figyelés az eredeti socket-en a kapcsolatkérések irányába, amíg a kliensnek egy másik porton kapcsolódik.



10.ábra Létrejön a kapcsolat a kliens és a szerver között

A kliens oldalon, ha a kapcsolat elfogadásra került, a socket sikeresen létrejött, akkor a kliens használni tudja a socket-et a szerverrel való kommunikálásra.
Megjegyzés: a kliens oldali socket nincs összekötve azzal a port számmal, amit a szerverrel való kapcsolattartásra használ. A kliens a helyi kijelölt port számmal azonosítható azon a gépen, amin fut.
A java.net csomag tartalmaz egy Socket osztályt, ami alkalmas egy kétirányú kapcsolat egyik oldalának vezérlése egy, a hálózaton lévő másik program felé, ez kell a klienshez, illetve tartalmazza a ServerSocket osztályt, amely figyel és elfogadja a kapcsolatot a kliensektől, ez pedig a szerver-funkciókhoz szükséges.
A kód try és catch elágazásokkal folytatódik, ami a java programozási nyelv egy nagyon fontos részéhez tartozik:

Kivételkezelés

A kivételkezelés egy utasításcsoport, a try-catch-throw csoport , amely a program végrehajtásnak sorrendjét befolyásolhatja.
A programok legnagyobb problémáját az egyes eljárások végrehajtása közben esetleg előbukkanó hibák, ún. kivételek (exception) lekezelése jelenti. Legtöbbször igen időigényes a programok hibatesztelése, legegyszerűbben az eljárások visszatérési értékének tesztelésével ellenőrizhető, sikerült-e a végrehajtott művelet. Ez általában működik, ám a hiba előbukkanását és okát "felfelé" terjeszteni és a megfelelő helyen lekezelni azt, már sokkal bonyolultabb probléma. Ezen segít a fenti utasításcsoport.
Egy try blokkba zárt utasításokon belül bárhol előforduló kivétel hatására a program normális futása abbamarad és a vezérlés automatikusan a catch blokkra kerül. Itt a catch paramétereként megkapjuk a hiba okát - általában egy Exception típusú objektumot. A hiba lekezelése vagy a catch blokkban történik, vagy a lekezelhetetlen kivételeket - a throw utasítással felfelé tovább lehet „dobni”, amíg „valaki” - legrosszabb esetben a virtuális gép - "lekezeli".
Kivétel keletkezésekor a program futása megszakad és megszakad a meghívóé is, ha nem definiált try-catch blokkot a kivételre, hanem a throws-al azt jelezte, hogy tovább akarja dobni. Így megy ez addig, amíg a hívási lánc tetejére nem ér, vagy bele nem ütközik egy aktív try-catch blokkba. Ekkor a try blokk megszakad és a futás a megfelelő catch blokkon folytatódik. A fentiekből következik, hogy ha egy metódust meghívunk, ami adhat valami kivételt és ezt jelzi is a throws-ban, akkor két eset lehetséges: vagy elkapja egy try-catch, vagy továbbdobja egy throws.
A Java nyelv a kivételek lekezelését megköveteli, ennek hiányában szintaktikai hibát jelez.
A try-catch szintaktikája a következőképpen néz ki:

```
try {
   ... utasítások ...
}
catch( Exception1 e1 ) {
  ... Exception1 típusú hibát lekezelõ utasítások ...
}
catch( Exception2 e2 ) {
  ... Exception2 típusú hibát lekezelõ utasítások ...
```

A try után következő blokkban vannak az utasítások, amelyek Exception1 vagy Exception2 típusú hibát generálnak. A fordító ellenőrizni fogja, tényleg megvan-e erre a lehetőség (minthogy a meghívott metódusok throws-aiból tudja, azok milyen kivételt dobhatnak), ha valamelyik kivétel nem következhet be, hibaüzenetet ad. Ha nem történik hiba, a try blokk rendben. Ezen blokk végrehajtása után a szerkezet véget ér. Ha azonban a try blokkban kivétel következik be, a blokk futása megszakad és a kivételobjektum (amit a throw-val generáltunk) beíródik a megfelelő catch blokk referenciaváltozójába. Ha például Exception1 következett be, az e1 mutat a kivételobjektumra, amikor a catch blokkba kerülünk és a futás az Exception1 catch blokkjában folytatódik. Ezt elvégezve elhagyjuk a szerkezetet. Egynél több catch blokk opcionális. Alapesetben a szerkezet így néz ki:

```
try {
  ... utasítások ...
} catch( Exception1 e1 } {
  ... Exception1 típusú hibát lekezelõ utasítások ...
}
```

A kapcsolat kiépítésére, és a kivételkezelésre való kitérő után, továbbnézve a korábbi kódot, a második try blokkban található egy Delay.ujra() metódus. Ez minden kivételkezelésnél megtalálható lesz a kliens szálban. Arra szolgál, hogy amennyiben valamilyen hiba folytán a működés megszakad, ez 5 másodpercig várakozik, majd újraindítja a klienst. A budapest csomagba sorolással kezdődik a kód:

```
package budapest;
import java.io.IOException;
public class Delay {
      public static void ujra() throws InterruptedException, IOException{
            Thread.sleep(5000);
            Fajlkuldes cli = new Fajlkuldes();
        cli.start();
      }
}
```

A Thread.sleep(5000) a várakozás idejének beállítására szolgál, mértékegysége miliszekundum. Miután ez letelt, a főprogramszálakkal megegyező módon elindítja a kliens szálat.
Amennyiben nem volt hiba, létrejön a socket, az átvitelre két stream objektum szolgál, egy a bemeneti, egy a kimeneti irányhoz, és amelyeket rendre a getInputStream, illetve a getOutputSteam eljárással hívhatunk meg. A kód felépítése a socket hez hasonlóan:

```
OutputStream KIMENO = null;
try {
	KIMENO = SOCKET.getOutputStream();
}
catch (IOException e) {
	try {
		Delay.ujra();
	}
	catch (InterruptedException e1) {
		e1.printStackTrace();
	}
	catch (IOException e1) {
		e1.printStackTrace();
	}
	e.printStackTrace();
}
```

Létrehoztuk a kapcsolatot, az adatcsatornát, elküldhető az első üzenet a szervernek:

```
PrintWriter IR = new PrintWriter(KIMENO);
String szoveg = null;
```

Ez önmagában nem elég, a stream tartalma csak a flush metódus hatására indul útjára:

```
while(true){
	IR.println(Var.data);
	IR.flush();
	Var.data="";
	try {
		Thread.sleep(60000);
	}
	catch (InterruptedException e) {
		e.printStackTrace();
	}
	if(szoveg=="kilep"){
		break;
	}
}
```

```
try {
	SOCKET.close();
}
catch (IOException e) {
	try {
		Delay.ujra();
	}
	catch (InterruptedException e1) {
		e1.printStackTrace();
	}
	catch (IOException e1) {
		e1.printStackTrace();
	}
	e.printStackTrace();
}
```

A kivételek lekezelése itt is meg kell történjen. Az adatok küldése a while cikluson belül folyamatos, kizárólag a berendezés leállítása, vagy a „kilep” parancs hatására áll meg.

4.3. Szerver

A szerver oldalon két program futhat. Egyik állandó jelleggel, ez maga a szerver feladatait ellátó alkalmazás. A másik a statiszikai program, ez természetesen csak lekérdezésekkor fut. Ebben a fejezetben a szerverprogram kerül bemutatásra. Két osztályból áll, ezek a Szerver.class és a Fajlbair.class.
A class (osztály) kiterjesztés azon fájlokat jelenti, melyeket a javac nevű fordító program már lefordított, tehát a Szerver.java fordítás után Szerver.class lesz.
Sikeres fordítás esetén tehát tárgykódú fájlt kapunk, a *.class fájlok már platform függetlenek, hordozhatóak, és osztályaik bármelyik JVM-en (Java Virtual Machine - Java virtuális gép) futtathatóak, utóbbi fogalom a már korábban tárgyalt java programok platformfüggetlenségéhez köthető.

Fordítási/futtatási szabályok:

- Egy fájl általában egy osztályt tartalmazzon. Amennyiben több osztályt tartalmaz, csak egy osztálya lehet publikus, és ez tartalmazhatja csak a main metódust.
- Egy program több fordítási egységből is állhat (fájl, osztály), de a programnak csak egy belépési pontja lehet:
    - public static void main(String[] args) {}).
- A fordító minden egyes osztályból készít bájtkódot (.class).
- Futtatni azt az osztályt/bájtkódot lehet, amely tartalmazza a main-t.

- A lefordított osztályoknak szükségük van a Java osztálykönyvtáraira (API), ezért a futási környezetnek (JRE) jelen kell lennie.

Visszatérve a Szerver.class fájlra, és annak kódjára, felépítését tekintve általános elrendezésű, importálásokkal kezdődik:

```
import java.io.*;
import java.net.*;
import java.util.*;
```

Az első két imortálás a kliens oldali fejezetben már ismertetett bemenetek és kimenetek, és a TCP/IP kapcsolatok kezelésére alkalmas könyvtárakra irányul.
a java.util.* a rendszer által támogatott adatszerkezeteket kezeli, mint a verem (Stack), a dátum (Date) vagy a szótár (Dictionary). Itt található a véletlen-számgenerátor (Random) is.

```
public class Szerver extends Thread {
	public static String inputipcim;
	public static void main(String[] args) throws IOException {
		Szerver szerver = new Szerver();
		szerver.start();
	}
```

A fenti sorok létrehozzák a Szerver osztályt, mely a már ismert extend Threads miatt örökli a szálak kezeléséhez szükséges információkat.
Létrejön az inputipcim nevű string típusú változó, itt tárolódik a kliens ip címe, public static-ként hozzuk létre, mivel máshol el akarjuk majd menteni a bejövő adatokkal együtt, hogy tudjuk, melyik IP-című forgalomirányító berendezés küldte az adatokat, s így a fájlbaíró osztály is hozzáfér ehhez a változóhoz. A public static void main kezdetű sor mindig az adott program főrészének első sora, a throws IOException a kivételkezelés. A főrészen belül létrehozzuk a szerver szálat, majd a start metódussal elindítjuk azt. Ekkor a Szerver visszatérési érték nélküli run metódusa indul:

```
public Szerver() {}
public void run() {
	try {
		ServerSocket serverSocket = new ServerSocket(7200);
		while (true) {
			Socket socket = serverSocket.accept();
			System.out.println("Bejovo kapcsolat. " +
			socket.toString());
			reqs.put(socket, new Request(this, socket));
		}
```

```
        }
        catch (IOException e){}
}
public void removeReq(Socket sock) {
        System.out.println("Kapcsolat vege. " + sock.toString());
        reqs.remove(sock);
    }
```

A szerver programok hálózati kommunikációért felelős része minden programban hasonlóan működnek, s az alábbi lépéseket hajtják végre futásuk során:

1. A szerver lefoglal egy még nem használt kommunikációs végpontot magának a futása kezdetén. Ezen a porton fogja várni a kliensek csatlakozását.
1. A szerver elkezd várni addig, amíg egy kliens ezen a lefoglalt porton meg nem próbál összeköttetést létesíteni.
2. Amikor egy kliens kapcsolatot próbál meg létesíteni a szerverrel, akkor a szerver létrehozza az összeköttetést a klienssel.
3. Ezután a kliens, és a szerver a létrehozott összeköttetésen keresztül adatokat cserélhet egymással.
4. Ha már nincs szükség a felépített összeköttetésre, akkor a szerver, vagy a kliens szabályosan lebontja a kapcsolatot.

Mivel a szerverek egy időben nem egy klienst, hanem többet szolgálnak ki, ezért a szerverek, amikor felépítik a kapcsolatot egy klienssel, akkor a klienssel történő kapcsolattartásra egy külön programszálat indítanak el, ami már önállóan tartja a kapcsolatot a klienssel. Ezután a szerver a b. pontban várja az újabb csatlakozni szándékozó klienst.

Az összeköttetés-alapú szerver feladatait ellátó kommunikációs végpontokat a Java-ban a ServerSocket osztály tartalmazza. Ez egy szerver feladatokat is ellátó TCP kommunikációs port modellje. A ServerSocket osztály konstruktorának sikeres végrehajtása után a kliensek már kezdeményezhetik a szerverrel való kapcsolatfelvételt. A ServerSocket a konstruktorában paraméterként megadott TCP porton várja a kliensek csatlakozását, ami jelen programnál a 7200-as portot jelenti. A szerverrel a kapcsolatot felépíteni szándékozó kliensek egy várakozási sorba kerülnek érkezési sorrendjükben, a szerver ebből a sorból mindig a legelső klienssel hozhat létre kapcsolatot, a ServerSocket objektum accept metódusának meghívásával, ami egy Socket osztálybeli objektumot ad vissza. Ez a visszaadott objektum azonosítja a kialakított szerver-kliens kapcsolatot, adatátviteli csatornát. Ez az osztály egy hálózati kommunikációs csatorna egy-egy végpontjának modellje, rendelkezik egy getInputStream, és egy getOutputStream metódussal, ezekkel az adatátviteli I-O csatornákhoz férhetünk hozzá, melyek a java.io csomagban definiált DataInputStream és DataOutputStream osztályba tartoznak. Az adatátvitelt ezen osztályok felhasználásával valósíthatjuk meg.

Ha tehát egy kliens kapcsolódik a szerverhez, akkor a szerver a System.out.println utasítással üzeneteket ír a képernyőre, ami a bejövő kapcsolat + kliens IP-címe formmában jelenik meg, és elindul a Request (kérés) szál, ami a fent bemutatott önálló klienssel való kommunikációt végzi.

```
class Request extends Thread {
```

Itt létrejön a Request szál,majd megtörténik az adatátviteli csatornák létrehozása, kezelése:

```
private final Szerver owner;
private Socket socket;
private BufferedReader bejovoCsatorna;
private PrintWriter kimenoCsatorna;

public Request(Szerver srv, Socket sock) throws IOException,
UnsupportedEncodingException {
      setDaemon(true);
      owner = srv;
      socket = sock;
      bejovoCsatorna = new BufferedReader(new
      InputStreamReader(socket.getInputStream(), "ASCII"));
      kimenoCsatorna = new PrintWriter(socket.getOutputStream());
      start();
}
```

Mint minden szál, ez is a run metódusában tartalmazza a végrehajtandó utasításokat, melyek:

```
public void run() {
      String msg;
      try {
            while ((msg = bejovoCsatorna.readLine()) != null) {
                  System.out.println(socket.toString() + " <- " + msg);
                  Szerver.inputipcim = ""+socket.getInetAddress();
                  Fajlbair.adatkiir(msg);
            }
      }
      catch (Throwable t) {}
      owner.removeReq(socket);
}
public void send(String msg) throws IOException {
      kimenoCsatorna.print(msg + "\r\n");
      kimenoCsatorna.flush();
}
```

Létrejön az msg (message - üzenet) nevű, string típusú változó, majd egy try blokkban, egy while cikluson belül történik a bejövő adat „elkapása”.
Amikor adat érkezik a socketen keresztül, akkor a bejovoCsatorna nevű, BufferedReader típusú objektum, a socket.getInputStream() metódus által megkapja ezt az értéket. Amikor értéke van, olyankor értékét továbbadja az msg-nek. A szerver minden egyes adatcsomag érkezésekor kiírja azt a kijelzőre:

```
Socket[addr=/152.66.223.70,port=52607,localport=7200] <- 013004006012
```

A nyíl után magát a beérkezett adatot olvashatjuk.
A konzolra való kiíratás után minden ciklusban konstruktorával meghívódik a Fajlbair.adatkiir(msg) metódus, amely az msg stringet kapja meg kezdeti értékként. Mint neve is mutatja a Fajlbair osztály tartalmazza az adatok merevlemezre írásáért felelős utasításokat:

```
import java.io.*;
import java.util.*;
```

Az első sorokban az IO és a Util osztálykönyvtárak importálása történik, szerepük már korábban bemutatásra került.

```
public class Fajlbair {
      public static void adatkiir(String a){…
```

Itt az eddigiekhez hasonlóan létrejön a Fajlbair osztály, és annak metódusa, az adatkiir mely után a zárójel nem üres, mivel ha a metódust konstruktorral hívják meg, akkor ide történik az érték átadása. A metódus meghívása jelen esetben a Szerver.run()-ból történik, a Fajlbair.*adatkiir*(msg); konstruktorral.
Ekkor tehát Szerver osztályban lévő msg értékét megkapja az ’a’ nevű String. Természetesen az ’msg’ is String, ellenkező esetben hibát kapnánk, illetve típuskonverzióval lenne megoldható a probléma, aminek módjáról a statisztikai alkalmazásnál lesz szó.

```
try{
      Calendar actDate = Calendar.getInstance();
      int y=actDate.get(Calendar.YEAR);
      String fajlnev;
      fajlnev = "d:\\"+y;
      if (actDate.get(Calendar.MONTH)<9)
            {fajlnev=fajlnev+ "0" +(actDate.get(Calendar.MONTH)+1);}
      else {fajlnev=fajlnev+(actDate.get(Calendar.MONTH)+1);}
      if (actDate.get(Calendar.DATE)<10)
            {fajlnev=fajlnev+ "0" +actDate.get(Calendar.DATE);}
```

```
      else {fajlnev=fajlnev+(actDate.get(Calendar.DATE));}
      fajlnev=fajlnev+".txt";
      …
```

Try blokkba kerültek az utasítások (kivételkezelés), a calendar utasítás lekérdezi a rendszer időt, melyet ezután komponensenként (year-év, month-hónap, stb.) kérdezhetünk le. A fájl neve, amibe az adatokat menti a program, mindig az aktuális dátum, kiterjesztése txt, így jön létre például a 20090214.txt, amely fájl a 2009.február 14-ei adatokat tartalmazza.
A merevlemezen való fájlcímzéshez a fajlnev típusú String kerül alkalmazásra, amely a fent látható módon D:\\-el kezdődik, majd hozzáfűzve az év, hó, nap komponenseket, végül a '.txt'-t készül el.
Az if-else elágazások a karakterfüzér '0'-kal való kibővítésére szolgál, illetve a month-hónap komponensnél mindig hozzá kell egyet adni, mivel a hónapok számozása '0'-val kezdődik. Az adatátviteli csatorna létrehozása történik a következő részben:

```
FileOutputStream outfile = new FileOutputStream(fajlnev,true);
DataOutputStream outdata = new DataOutputStream(outfile);
PrintStream wr = new PrintStream(outfile);
```

Többfajta konstruktor közül választhatunk, megadhatjuk zárójelben egyszerűen csak a fájlnevet, ekkor a program minden egyes fájlbaíráskor újraírja a fájlt. Ez természetesen itt nem alkalmazható, s ekkor a vesszővel elválasztott 'true'-t használjuk, amitől a metódus nem ír felül, hanem mindig a legelső üres sorba menti az adatot.

```
String fuzer="";
if (actDate.get(Calendar.HOUR_OF_DAY)<10)
      {fuzer="0" +actDate.get(Calendar.HOUR_OF_DAY);}
else {fuzer=fuzer+(actDate.get(Calendar.HOUR_OF_DAY));}
if (actDate.get(Calendar.MINUTE)<10)
      {fuzer=fuzer+ "0" +actDate.get(Calendar.MINUTE);}
else {fuzer=fuzer+(actDate.get(Calendar.MINUTE));}
fuzer=fuzer+" "+a+" "+Szerver.inputipcim;
wr.println(fuzer);
outdata.close();
```

A fuzer nevű String-be hozzuk létre az elmentdő adatsort, az =”” a változó üres kezdeti értékének beállítására szolgál. Mint az feljebb látható, a karaktersor első két karaktere az aktuális idő órája, majd két karakteren a perc, egy üres karakter (space), majd az 'a' változó, amely maga a beérkezett detektorértékek 12 karakteren, ismét egy space, végül a kliens ip címe, mely a lementett txt-ben a következőképp néz ki:

```
1416 012013007007 /152.66.223.70
```

A metódus catch blokkal zárul, mely kiírja a konzolra a hibakódot:

```
catch (Exception e){
      System.err.println("Error: " + e.getMessage());
}
```

4.4. Statisztika

A szerver oldalon, a szever-funkciókat betöltő program futásával párhuzamosan, lehetőség van egy adatokat lekérdező alkalmazás futtatására.
Itt meg kell említeni, hogy a diagram megjelenítés JfreeChart segítségével készült.

JfreeChart osztálykönyvtár

A JfreeChart egy ingyenes diagram könyvtár Java platformra. Alkalmazásokban, appletekben servletekben és JSP-ben való használatra tervezték. A JfreeChart a teljes forráskódjával együtt hozzáférhető a GNU Lesser General Public License szerint.
A JfreeChart képes kördiagramok, hisztogramok, vonaldiagramok, idősorok, Gantt-diagramok, számlap diagramok, szimbólum-diagramok, széltérképek, összetett diagramok, stb. megjelenítésére.
További jellemzői:

- adathozzáférés a definiált interfészek bármely implementációjával
- exportálás PNG és JPEG formátumokba
- exportálás bármely Graphics2D implementációval rendelkező formátumba, pl.:
    - PDF az iText segítségével
    - SVG a Batik segítségével
- tooltipek
- interaktív zoomolás
- eseménykezelés
- széljegyzetek
- HTML image map generálás
- LGPL licensz

A JfreeChart teljes egészében Java-ban készült, a Java 2 platform bármely implementációján működik (JDK 1.3.1 vagy régebbi).

Az adatlekérdezés törzse a Foprog osztályban kapott helyet, ez kérdezi meg a felhasználótól a lekérdezéshez szükséges adatokat.

```
import java.io.BufferedReader;
```

```
import java.io.IOException;
import java.io.InputStreamReader;
import org.jfree.ui.RefineryUtilities;
```

Az importálás tartalmazza az IO könyvtárat, illetve egy új, nem java által tartalmazott osztálykönyvtárat, az org.jfree.ui.RefineryUtilities-t, ez a JfreeChart-ból importál.

```
public class Foprog {
      public static void main(String[] args) throws IOException{
            System.out.println("FORGALOM STATISZTIKA");
            System.out.print("Lekérdezés típusa: ");
            BufferedReader standard = new BufferedReader(new
            InputStreamReader(System.in));
            Var.lekerdtipus = standard.readLine();
            System.out.print("Kérem az évet: ");
            Var.evout = standard.readLine();
            System.out.print("Kérem a hónapot: ");
            Var.hoout = standard.readLine();
            System.out.print("Kérem a napot: ");
            Var.napout = standard.readLine();
```

A fenti sorokban létrejön a Foprog osztály, s annak főrésze, melyet a public static void main... kezdetű sor jelöl.
Létrehozunk egy standard nevű BufferedReader-t, amely figyeli a billentyűzetről bevitt információkat. A bevitel mindig az ENTER lenyomásával végződik.
A program futásának kezdetekor kiírja a képernyőre: FORGALOM STATISZTIKA, majd új sorba: 'Lekérdezés típusa: '. Új sorba a println használatával tudunk írni, azonos sorba íráshoz a print-et használjuk.
Ezután a program várja a BufferedReader segítségével az adatbevitelt, majd az ENTER lenyomásakor a bevitt adatot a Var.lekerdtipus változóba menti. Látható, hogy ez egy osztályváltozó, méghozzá a Var osztályé. Ez a Var osztály nem azonos a kliens oldalival, külön osztály, és csupán néhány string, és integer változóval rendelkezik:

```
public static String inputipcim, datum, lekerd,ev, ho, nap, ora, perc,
      evout, hoout, napout, oraout, det1s, det2s, det3s, det4s,
      lekerdfajlnev, lekerdtipus;
public static int det1, det2, det3, det4, det1v, det2v, det3v, det4v,
      lekerdtipusint;
```

A Foprog osztály a fent említetteken túl lekérdezi az évet, a hónapot, a napot. Ekkor a program már tudja, hogy melyik fájlban kell keresnie, így össze is állítja a lekérdezéshez szükséges elérési útvonalat:

```
Var.lekerdfajlnev="D:\\"+Var.evout+Var.hoout+Var.napout+".txt";
```

A továbbiakhoz kell a lekérdezés típusa, ehhez először egy már korábban hivatkozott típuskonvertálásra van szükség, melynek formája:

```
Var.lekerdtipusint = Integer.parseInt(Var.lekerdtipus);
```

Itt a Var.lekerdtipus nevű String integerré alakítása történik, amennyiben az csak számokat tartalmaz - ellenkező esetben hibaüzenet jelenik meg – az értéket megkapja a Var.lekerdtipusint.
A további kérdés már attól függ, hogy milyen lekérdezés típust választottunk, mivel ha egy adott óra forgalomeloszlását kérdezzük, akkor szükség van az adott órára, amennyiben napi eloszlás, úgy erre nincs szükség:

```
if (Var.lekerdtipusint<3){
      System.out.print("Kérem az órát: ");
      Var.oraout = standard.readLine();}
```

Itt látható, hogy az 1-2-es számú lekérdezés adott órára mutat, a 3-4-es lekérdezés pedig napra. Miután megvannak a szükséges információk a lekérdezéshez, a főprogram, meghívja szükséges osztályt, illetve eljárást.
Az 1-3-as metódushívás ugyanazt a metódust hívja meg, csak más adatokkal számol (az 1. órás-, a 3. napi lekérdezést tartalmaz), és ez érvényes a 2-4-es metódushívások kapcsolatára is, így csökkenteni lehet az if elágazások számát:

```
if (Var.lekerdtipusint==1 || Var.lekerdtipusint==3){
      Forgalom demo = new Forgalom("Forgalom statisztika");
      demo.pack();
      RefineryUtilities.centerFrameOnScreen(demo);
      demo.setVisible(true);
}
if (Var.lekerdtipusint==2 || Var.lekerdtipusint==4){
      XYSplineRendererDemo1 appFrame = new XYSplineRendererDemo1(
      "Forgalom statisztika");
      appFrame.pack();
      RefineryUtilities.centerFrameOnScreen(appFrame);
      appFrame.setVisible(true);
}
```

A fent látható kódrészletben a ’||’ jel a vagy kapcsolatot jelzi.
Az első esetben a Forgalom osztály metódusaival dolgozik a program:

```
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.CategoryPlot;
import org.jfree.chart.plot.PlotOrientation;
```

```
import org.jfree.chart.renderer.category.BarRenderer;
import org.jfree.data.category.CategoryDataset;
import org.jfree.data.category.DefaultCategoryDataset;
import org.jfree.ui.ApplicationFrame;
```

Az első sorok a JFreeChart-ból való importálásokat jelentik.

```
public class Forgalom extends ApplicationFrame {
       public Forgalom(String title) throws IOException {

              super(title);
              CategoryDataset dataset = createDataset();
              JFreeChart chart = createChart(dataset);
              ChartPanel chartPanel = new ChartPanel(chart, false);
              chartPanel.setPreferredSize(new Dimension(1280, 1024));
              setContentPane(chartPanel);
      }
```

Létrejön a Forgalom osztály, és elkészül egy alapbeállítású Windows-os ablak, esetünkben 1280*1024-es méretben.
A Forgalom osztály oszlopdiagramot készít, s a következő sorok ezen oszlopok létrehozását, elnevezését tartalmazzák.

```
private static CategoryDataset createDataset() throws IOException {

      String series1 = "1.detektor";
      String series2 = "2.detektor";
      String series3 = "3.detektor";
      String series4 = "4.detektor";
```

A következő lépés az adatbeállítás(dataset):

```
DefaultCategoryDataset dataset = new DefaultCategoryDataset();
if (Var.lekerdtipusint==1){
      int i = 0;
      Var.perc = null;
      String category = null;
      while(i!=60 ){
            category = null;
            if (i<10){Var.perc= "0" +i;}
            else {Var.perc=""+i;}
            Lekerdezes.melyiksor();
            category=""+i;
            dataset.addValue(Var.det1v, series1, category);
            dataset.addValue(Var.det2v, series2, category);
            dataset.addValue(Var.det3v, series3, category);
            dataset.addValue(Var.det4v, series4, category);
            i++;
      }
}
if (Var.lekerdtipusint==3){
      int i = 0;
      Var.perc = null;
      Var.ora = null;
      String category = null;
```

```
      while(i!=24){
            category = null;
            if (i<10){Var.ora= "0" +i;}
            else {Var.ora=""+i;}
            Lekerdezes.melyiksor();
            category=""+i;
            dataset.addValue(Var.det1v, series1, category);
            dataset.addValue(Var.det2v, series2, category);
            dataset.addValue(Var.det3v, series3, category);
            dataset.addValue(Var.det4v, series4, category);
            i++;
      }
}
return dataset;
```

Mint látható, az adatbeállítás a lekérdezés tíusától függően kettéágazik, majd az értékek beállítása után a dataset objektumot kapjuk.

```
String ora="";
if (Var.lekerdtipusint==1){
      int oraszam;
      oraszam = Integer.parseInt(Var.oraout);
      ora =Var.evout+"."+Var.hoout+"."+Var.napout+".  "+Var.oraout+" és
      "+(oraszam+1)+" óra között";
}
if (Var.lekerdtipusint==3){
      ora =Var.evout+"."+Var.hoout+"."+Var.napout+". napja";
}
```

Az oszlopdiagram létrehozásakor megnevezzük magát a diagramot, melyet az ora nevű Stringben tárolunk, ami a lekérdezéstől függően kerül előállításra.
A konstruktor:

```
JFreeChart chart = ChartFactory.createBarChart(
ora, "Idő (perc)", "Forgalomnagyság (jármű)", dataset,
PlotOrientation.VERTICAL, true, true, false );
```

A konstuktor paraméterei közül számunkra az első öt lényeges, melyek sorrendben:
az egyel feljebb létrehozott ora nevű string, a vízszintes tengely neve, a függőleges tengely neve, az adattábla, majd az elhelyezés.
A diagram lényegi része ekkor már készen van, innentől az opcionális beállítások történnek, például színek, vonaltípusok, stb., amelyre jelen tanulmány nem tér ki, de a melléklet tartalmazza a kódot, és ahol lehet, tartalmaz hozzáfűzéseket (comment) is.
Amennyiben a 2. vagy a 4. lekérdezést választottuk, úgy az XYSplieRendererDemo1 osztályában fut tovább a program, melynek importálásokat tartalmazó kezdeti része szintén megtalálható a mellékletben.

```
public class XYSplineRendererDemo1 extends ApplicationFrame {
    static class MyDemoPanel extends DemoPanel {

        private XYDataset data1;
```

```
        public MyDemoPanel() throws IOException {
            super(new BorderLayout());
            this.data1 = createSampleData();
            add(createContent());
        }
```

A szokásos Java program kezdet: importálások, osztály létrehozása, metódus létrehozása történik, illetve elkészül a diagramhoz szükséges adattábla, amelyet a következő sorok töltenek föl adatokkal:

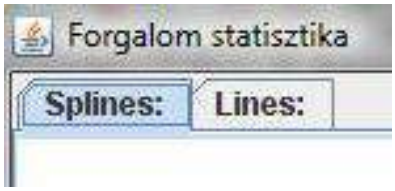
```
private XYDataset createSampleData() throws IOException {

XYSeries series = new XYSeries("Detektor 1");
int i = 0;
Var.perc = null;
if (Var.lekerdtipusint==2){
      while(i!=60 ){
            if (i<10){Var.perc= "0" +i;}
            else {Var.perc=""+i;}
            Lekerdezes.melyiksor();
            series.add(i,Var.det1v);
            i++;
      }
}
if (Var.lekerdtipusint==4){
      Var.ora = null;
      while(i!=24){
            if (i<10){Var.ora= "0" +i;}
            else {Var.ora=""+i;}
            Lekerdezes.melyiksor();
            series.add(i,Var.det1v);
            i++;
      }
}
XYSeriesCollection result = new XYSeriesCollection(series);
…
```

Ezek a ciklusok az 1. számú detektor adatait teszik be az adattáblába, a másik három detektor adatai ugyanezen rendszer szerint kerülnek feldolgozásra, végül a programrész visszatérési értékként adja az adattáblát:

```
return result;
```

Innentől a program két metódussal folytatódik, így eredményként a windows-os ablak két fület tartalmaz (lásd. 11. ábra), az egyik egyenes vonalakkal köti össze az értékeket, amíg a másik ezen pontokra egy-egy regressziós görbét fektet.



11.ábra A megjelenítési módokhoz tartozó fülek

A kód további részének felépítése megegyezik a Forgalom osztályéval, így ennek ismertetésére külön nem kerül sor, de a programkód megtalálható axy mellékletben.

4.5. Tesztelés, kipróbálás valós berendezésen

A program tesztelése személyi számítógépen jelenleg nem megoldott, a korábban említett, pc-n nem létező C-függvények miatt.
A valós berendezésen való tesztelés a korlátozott hozzáférés, illetve a feltöltési-, és beállítási időigények miatt igen hosszadalmas.
Fentiek okán készült egy egyszerű random detektorértékeket előállító program, mely az ACTROS-t helyettesítheti a tesztelések kezdeti szakaszában. Melynek lényegesebb részei:

```
static Random gen = new Random();  Létrejön a random függvény,
int i1= gen.nextInt(x)+y;
```

i1..i4 detektorértékek képzésének első sora, ahol x és y segítségével y és x+y közötti értékeket állít elő a függvény, lényeges, hogy az adott változó ne csorduljon túl, integer esetén a maximális érték: 2147483647.
A további rész az Sstat-hoz hasonlóan létrehozza a fenti értékek tartalazó string típusú karakterláncot.
Ütemezni itt is a thread.sleep-el tudunk:

```
Thread.sleep(x);
```

Az ’x’ a várakozás idejét jelöli(msec).
Végül a metódus továbbadja a visszatérési értékét:

```
return str;
```

További pozitívum, hogy így a szerveroldal fejlesztése is egyszerűsödött, valamint a kliens szál tesztelése is megtörténhetett, s utóbbi átültetése a forgalomirányító berendezésre csak apróbb módosításokat igényelt, köztük a véletlenszerű (random) értékekről, a valós detektorértékekre való átállás.
A kész program feltöltése Crosslink UTP kábelen keresztül történik.
Az UTP (Unshielded Twisted Pair - árnyékolatlan csavart érpár) egy hálózati kábeltípus a számítástechnikában. Külső zavarok ellen védtelen adatátviteli közeg. Leggyakrabban alkalmazott kábeltípus az ethernet hálózatokon. Az UTP kábel számos hálózatban használt, 4 érpárból álló réz alapú átviteli közeg. Az UTP kábeleknek mind a 8 rézvezetéke szigetelőanyaggal van körbevéve. Emellett a vezetékek párosával össze vannak sodorva, így csökkentve az elektromágneses és rádiófrekvenciás interferencia jeltorzító hatását. Az árnyéko-

latlan érpárok közötti áthallást úgy csökkentik, hogy az egyes párokat eltérő mértékben sodorják. Maximális átviteli távolsága 100 m. Típusai:

- Egyeneskötésű (link):
    - PC – Switch
    - Router – Switch
    - HUB -- PC
- Keresztkötésű (cross-link):
    - Router -- PC
    - PC -- PC
    - Switch -- Switch
- Konzol (cross-over): Számítógép soros portja és router/switch konzol portja (DB-9 - RJ-45 átalakító) közötti átvitelhez.

Pozitívum a viszonylagos olcsósága, könnyű telepíthetősége, kis átmérője, hátrányaként említhető külső interferencia-források elleni viszonylagos védtelensége, valamint kis átviteli távolsága.A kábel végein 8P8C (RJ45) csatlakozók találhatók, amellyel a hálózati interfészekhez csatlakozik. A teszteléshez összekötött ACTROS-PC rendszer a 12. ábrán látható:

12.ábra ACTROS tesztelés közben

A Crosslink kábel csatlakoztatása után két fajta kommunikáció közül választhatunk:

- LAN kapcsolat: ekkor internetes böngésző segítségével érhetjük el a berendezés webes felületét, a 10.10.50.39.-es IP címen keresztül. Ez a kapcsolat leginkább a berendezés felhasználókénti vezérlésére alkalmas. A honlap egyik látványos tulajdonsága, hogy az éppen futó fázis-idő tervet valós időben (realtime) jeleníti meg.
- FTP kapcsolat: az FTP kapcsolatra a berendezés programozásához van szükség. Az FTP szerver tárolja a működéshez szükséges kódokat.

Az FTP kapcsolat TCP/IP beállításai:

- IP cím: 10.10.50.xx. (az utolsó kettő bármi lehet a 39 kivételével, mivel az a berendezésé)
- Maszk: a mezőbe kattintva automatikusan kitöltődik
- A többi mezőt üresen kell hagyni

Az új program feltöltése során az ACTROS ROM-jában lévő kódot felül kell írni, majd újraindítás után már az új programmal indul a berendezés.

5. Eredmények, továbbfejlesztési lehetőségek

5.1. Futtatási eredmények

A tesztelések során az ACTROS berendezés elküldte az adatokat, és a szerver sikeresen lementette azokat a merevlemezre. Statisztikai szempontokból a programhoz órás és napi forgalomnagyság grafikonok készültek, azonban az ezekhez szükséges adatok valós bevitele idő hiányában nem történhettek meg, így ezekhez random függvény szolgáltatta az adatokat.
Az átvitt adatok helyességének ellenőrzésére is sor került, mely szerint az adatküldés nem tartalmazott hibát, és a konzolon is helyes értékek jelentek meg.
Az egyik teszt során a konzolon megjelent sorok:

```
Bejovo kapcsolat. Socket[addr=/152.66.223.70,port=52607,localport=7200]
Socket[addr=/152.66.223.70,port=52607,localport=7200] <- 013004006012
Socket[addr=/152.66.223.70,port=52607,localport=7200] <- 013011003007
…
Socket[addr=/152.66.223.70,port=52607,localport=7200] <- 013012004004
Socket[addr=/152.66.223.70,port=52607,localport=7200] <- 006004010010
Socket[addr=/152.66.223.70,port=52607,localport=7200] <- 010011005007
Kapcsolat vege. Socket[addr=/152.66.223.70,port=52628,localport=7200]
```

Az első sor értelemszerűen a kapcsolat létrejöttét, a továbbiak az adatfogadásokat, majd az utolsó a kapcsolat bontását jelzik. A kapcsolat megszakadására való válasz is helyesen működött, érzékelte a kapcsolat szakadását, ennek oka a szerver leállása, vagy az átviteli közeg hibája, mindkettő kipróbálásra került, s a szerverprogram jelezte a kapcsolat szakadását, illetve a kliensoldali problémamegoldás is működött, a szál nem szakadt meg, hanem próbált újracsatlakozni, és a szerver újraindítása, vagy a vezeték visszakötése után ez sikerült is, és folytatta az adatküldést.
Az adatok mentése a megfelelő fájlba, a megfelelő módon történt. Az egyik mentett txt tartalma:

```
1503 009009006006 /152.66.223.70
1504 010008006006 /152.66.223.70
1505 009010007005 /152.66.223.70
1506 011009007006 /152.66.223.70
1507 008009006006 /152.66.223.70
1508 009010006007 /152.66.223.70
1509 010009007006 /152.66.223.70
1510 011008008007 /152.66.223.70
```

A statisztikai alkalmazás eredményeinek megjelenése felugró ablakban történik meg, ilyen grafikonok a .. számú mellékletben láthatók.

5.2. Továbbfejlesztési lehetőségek

Az összehangolt forgalomszabályozás, a közúti forgalomirányító rendszerek számtalan továbbfejlesztési lehetőséget tartogatnak magukban.
Egyrészt számos átviteli módot, adattárolási és adatfeldolgozási módszert fel lehet még használni, hogy hatékonyabbá tegyük a működést. Másrészt az ACTROS is rengeteg további felhasználási lehetőséget kínál.
Adatátviteli módok között a vezetéknélküli (wireless) adatátviteli technika említendő, ehhez mind szoftveres fejlesztés, mind hardveres bővítés szükséges.
Adattárolás terén további fejlesztési lehetőség az adatok SQL adatbázisban való tárolása. A jelenlegi ACTROS szakadt kapcsolat esetén nem tárolja el a saját memóriájában az adatokat, azok elvesznek. Ide lehetőség van olyan programszál készítésére, amely érzékeli a kapcsolat hiányát, és ezen időtartam alatt elmetni az adatokat a memóriába, majd ha a kapcsolat újra létrejött, elküldi azokat a szervernek.
A lekérdezések típusa is bővíthető természetesen, illetve további paraméterek bekérésével sokrétűbb lekérdzések is megvalósíthatók. A statisztikai adatok megjelenítésekor az adatok bevitele egy legördülő menükből, gombokból álló felület esetén sokkal felhasználóbarátabb lenne, a kezelés is egyszerűsödne. További fejlesztési lehetőség, amennyiben az adott ACTROS ip címéhez kapcsoljuk annak helyét, úgy a fent említett legördülő menüben az egyes berendezések helye alapján is lehetne választani. Az általánosan vett szerver-kliens rendszer alapján a lekérdezések alkalmazásnak át kellene kerülnie egy felhasználói számítógépre, ekkor a szerver feladatai közül kikerül a statisztikai megjelenítés, helyette a szerverfunkciók bővülnének, méghozzá kiszolgálója lenne a lekérdezést végző felhasználónak, kliensnek.
A biztonsági funkciók szintén fejleszthetők, valós felhasználás esetén ez egy szükséges lépés is, amely a csatorna biztonságának ellenőrzésén kívül, az adatok kódolását, illetve a hozzáférés korlátozását jelentheti.

6. Összefoglalás

A magyar utakon napjainkban még elég korlátozott mértékben használnak forgalomszabályozást. A motorizáció egyértelmű növekedési tendenciája, és a városi közlekedésben egyre jellemzőbb torlódások azonban ezen eszközök létjogosultságát vetítik elő.

A diplomatervem célja a modern forgalomirányítás egy lehetséges adtfeldolgozási folyamatának megvalósítása volt. A felépített rendszer bemutatja a detektorokból vett információk küldésének, eltárolásának, feldolgozásának, megjelenítésének egy lehetséges változatát, amely jó kiindulási alapja lehet bármilyen más hasonló célú fejlesztésnek.
Az elkészített rendszer nyilvánvalóan nem kizárólagos módja a folyamat megvalósításának, de mindenképpen egy fejlesztési irányt mutat.
Elmondható, hogy a dolgozatban bemutatott renszer a forgalomszabályozás optimalizálását segíti elő. Az utakban rejlő kapacitástartalékok kiaknázása pedig igen fontos, - mint az a bevezetőben is olvasható volt - a forgalom szabályozása több szempontból is ideálisabb az infrastruktúrális bővítésnél, különösen ha ezt a szabályozást minél nagyobb mértékben optimalizáljuk, így maximalizálva az adott útvonal, csomópont forgalomkapacitását. Természetesen több forgalomszabályozó eszköz összehangolása további kapacitás-növekedést jelenthet.

Köszönetet szeretnék mondani konzulenseimnek Tettamanti Tamásnak és Varga Istvánnak a dolgozat megírásához nyújtott szakmai segítségért, valamint Nagy Tibornak az ACTROS kezeléséhez és programozásához adott tanácsokért.

7. Felhasznált irodalom

Vilati-Signalbau Huber:
ACTROS VTC 3000 teljes dokumentáció (német nyelvű)

Tettamanti Tamás:
Autópálya forgalomszabályozás felhajtókorlátozás és változtatható sebességkorlátozás összehangolásával és fejlesztési lehetőségei című diplomamunka, 2007.

Uwe Wilbrand:
Java forgalomtechnika az ACTROS VTC 3000 berendezésben,
fordította Tettamanti Tamás, 2007.

dr. Katkó László, Molnár Géza, Varga István:
Közúti forgalomirányító berendezések, 2000.

Dévényi Károly:
Java dokumentációk, 2002.

Kiss István:
A Java programozási nyelv rejtelmei, 1996.

www.sun.java.com
A Java programnyelv készítő vállalat, a Sun hivatalos weboldala

www.javagrund.hu/javasite/dokument
Magyar és idegen nyelvű Java dokumentációk, segédletek

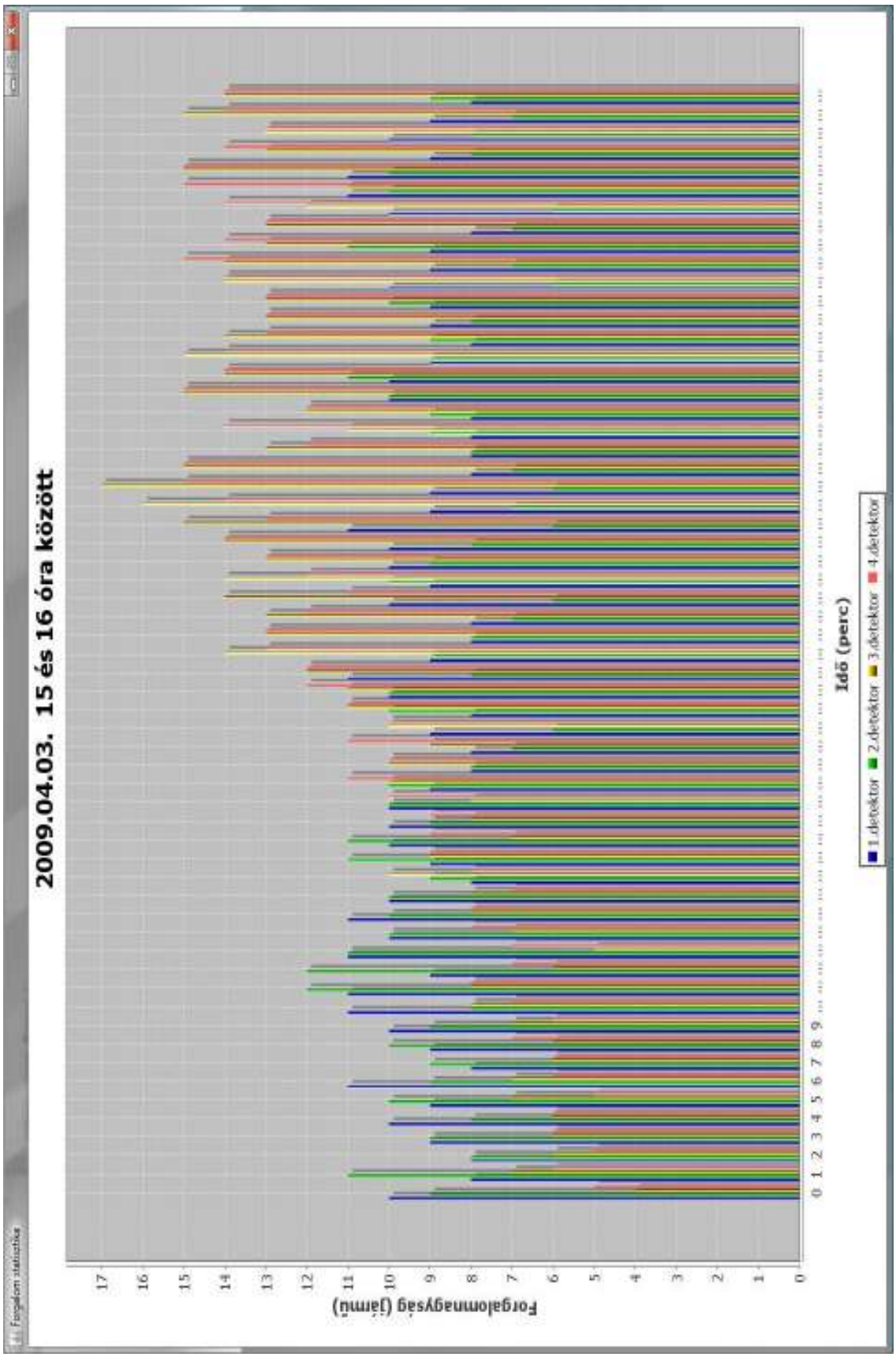
www.prog.hu – informatikáról szakszerűen
Programozói és informatikai fórum

www.jfree.org/jfreechart
A JfreeChart, ingyenes diagram könyvtár hivatalos weboldala

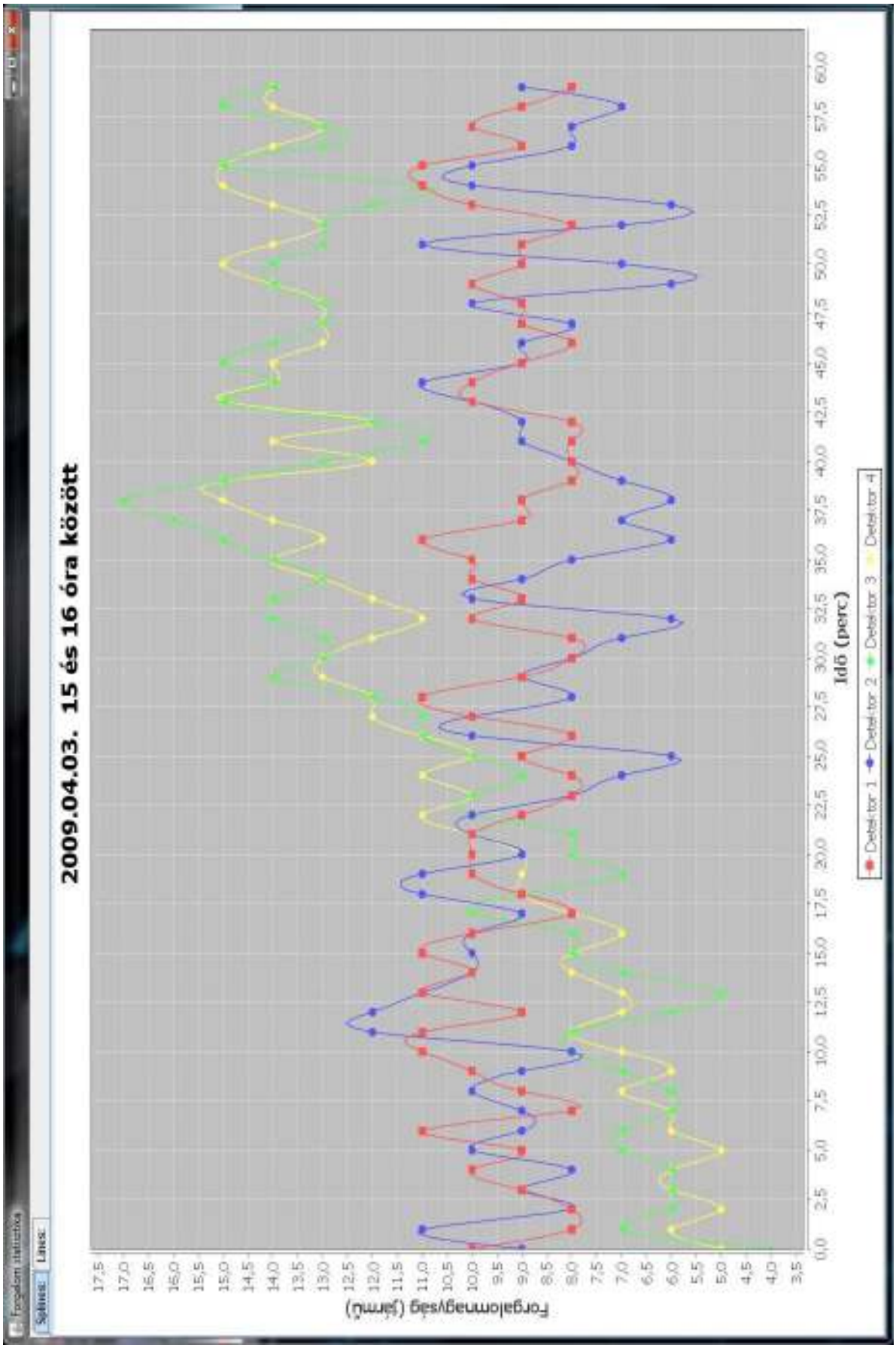
http://hu.wikipedia.org
Szabad enciklopédia

8. Mellékletek

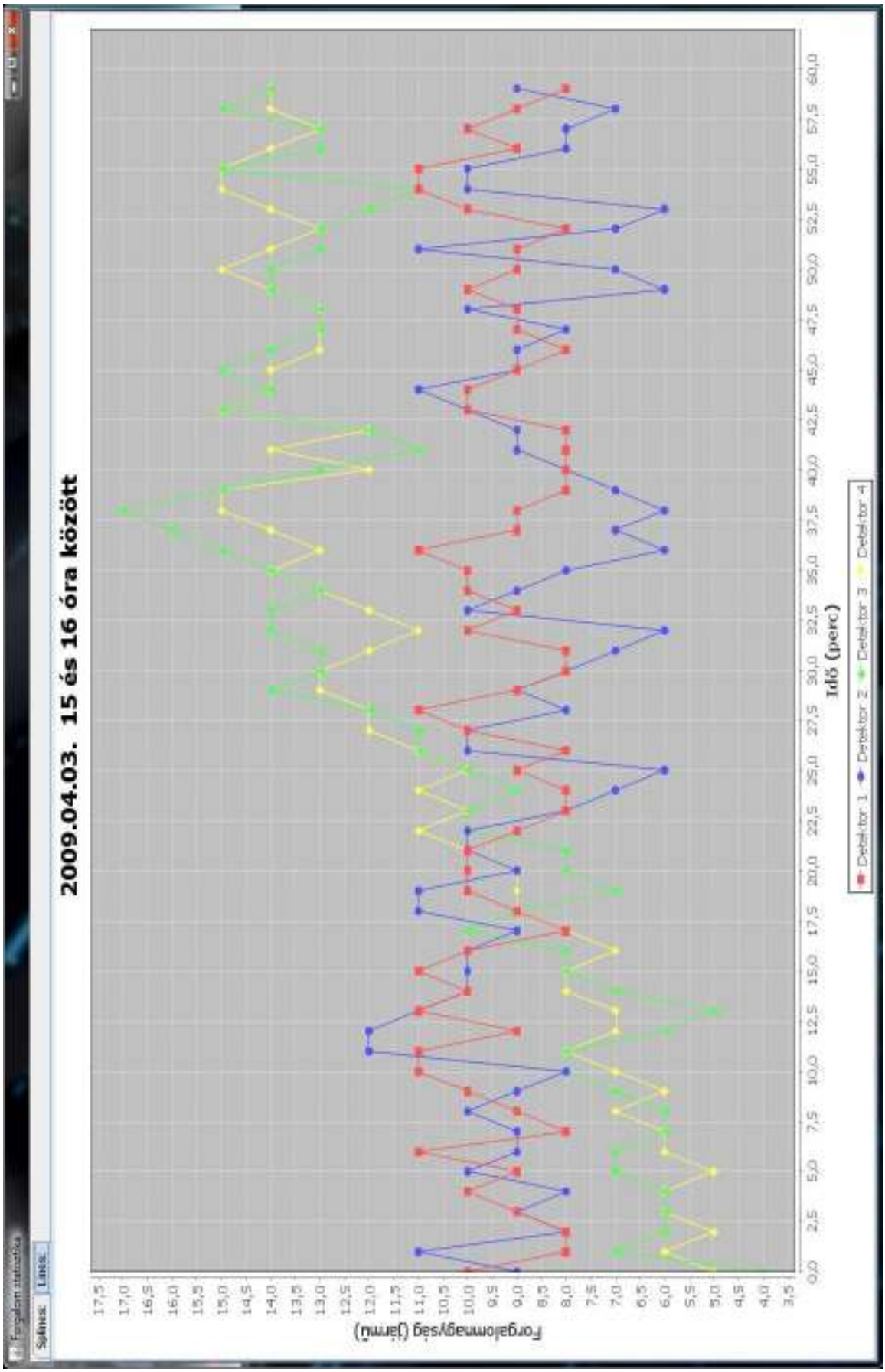
8.1. Órás forgalom - oszlopdiagram



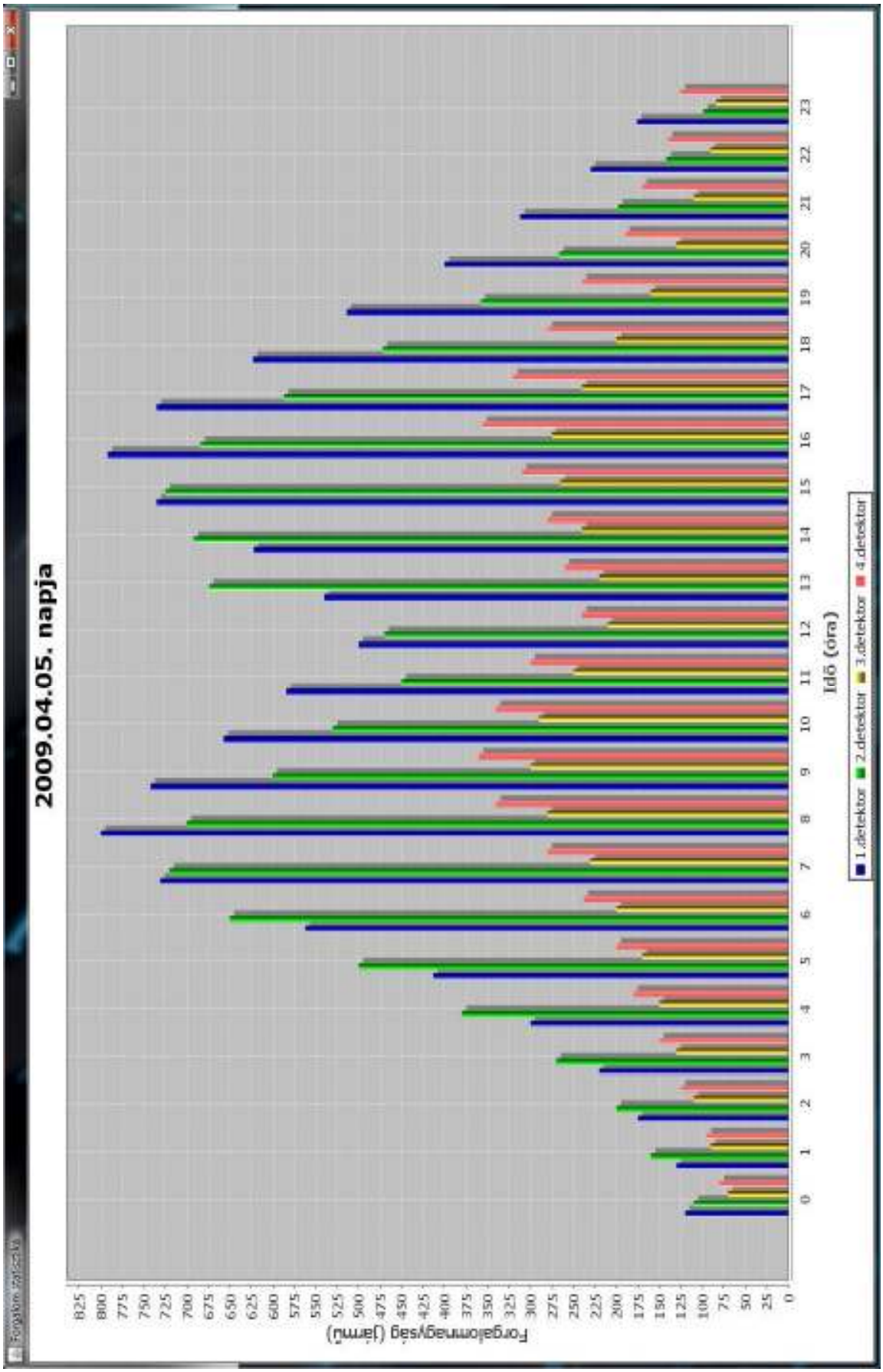
8.2. Órás forgalom – regressziós görbe diagram



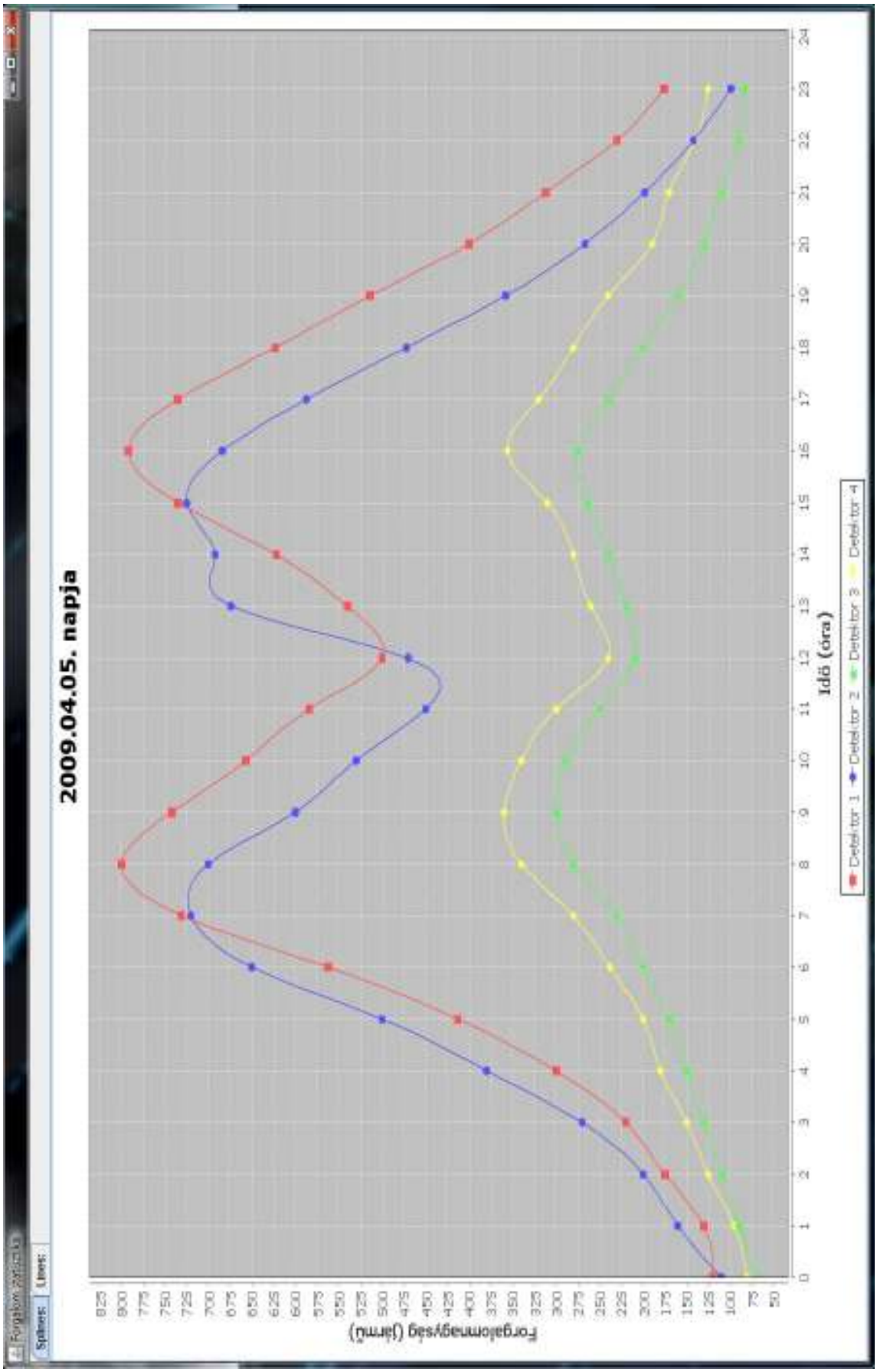
8.3. Órás forgalom – lineáris diagram



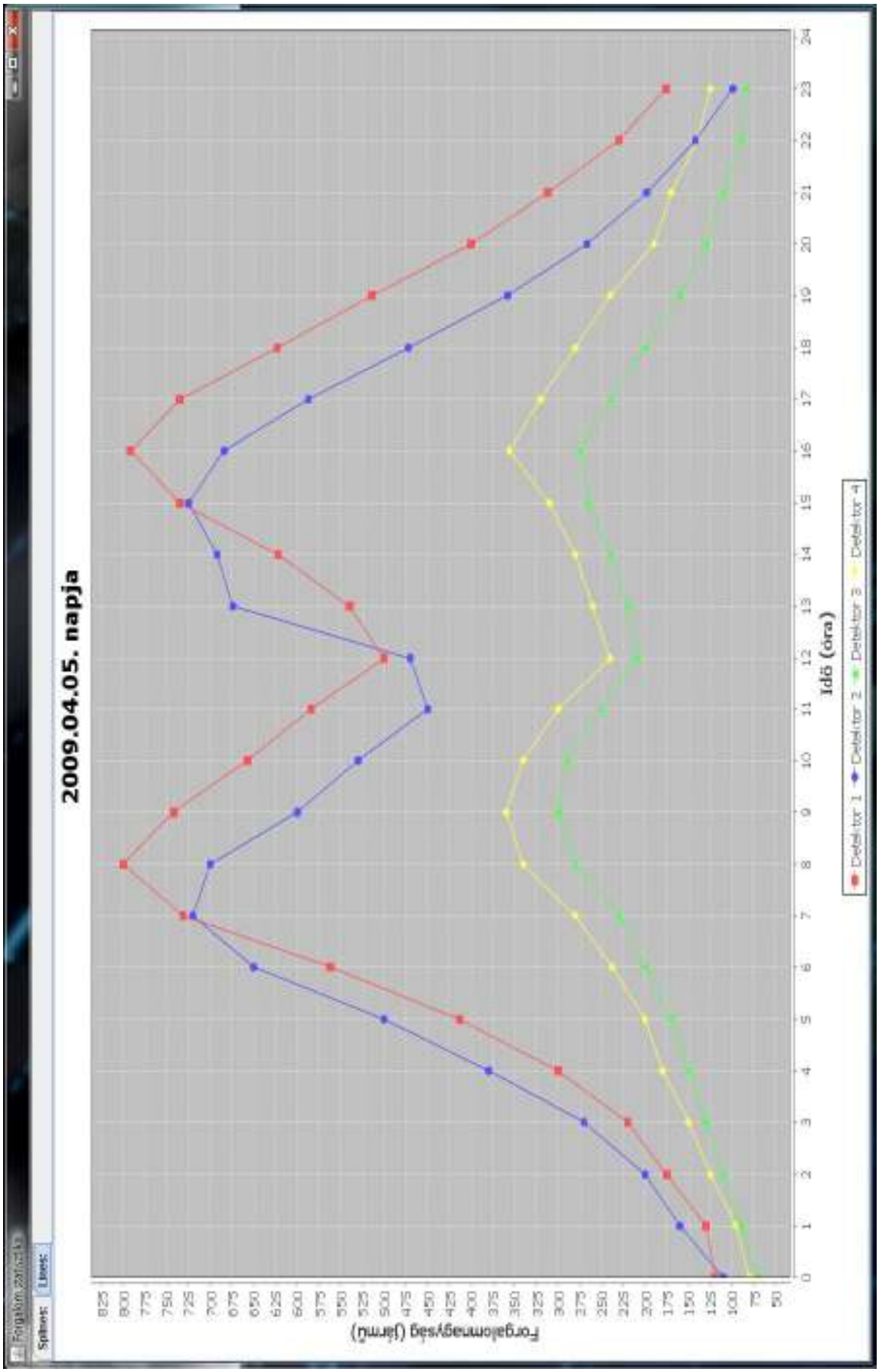
8.4. Napi forgalom - oszlopdiagram



8.5. Napi forgalom – regressziós görbe diagram



8.6. Napi forgalom – lineáris diagram



8.7. ACTROS - kliens programkódok

Terjedelmi okok miatt az ACTROS saját kódjának azon terjedelmes részei, amelyeken nem történt változtatás kimaradnak, azonban ezt mindig ’ …’ jelöli.

Randm.java

```
import java.util.*;
public class Randm {

      static Random gen = new Random();

    public static String rand() throws InterruptedException {
      int i1= gen.nextInt(10)+5;
      int i2= gen.nextInt(10)+4;
      int i3= gen.nextInt(10)+2;
      int i4= gen.nextInt(10)+3;

      String str="";
      if (i1<100){str = "0";}
        if (i1<10){str =str+ "0" +i1;}
        else {str=str+i1;}

        if (i2<100){str =str+ "0";}
        if (i2<10){str = str+"0"+i2;}
        else {str=str+i2;}

        if (i3<100){str =str+ "0";}
        if (i3<10){str = str+"0"+i3;}
        else {str=str+i3;}

        if (i4<100){str =str+ "0";}
        if (i4<10){str = str+"0"+i4;}
        else {str=str+i4;}

        Thread.sleep(100);
        System.out.println(str);
        return str;
    }
}
```

Delay.java

```
package budapest;

import java.io.IOException;

public class Delay {
      public static void ujra() throws InterruptedException, IOException{
            Thread.sleep(5000);
            Fajlkuldes cli = new Fajlkuldes();
        cli.start();
```

```
	}
}
```

Fajlkuldes.java

```
package budapest;

import java.io.*;
import java.net.*;

public class Fajlkuldes extends Thread
{

public void run()
{
Socket SOCKET = null;
try {
	SOCKET = new Socket("152.66.223.70",7200);
} catch (UnknownHostException e) {
	e.printStackTrace();
} catch (IOException e) {
	try {
		Delay.ujra();
	} catch (InterruptedException e1) {
		e1.printStackTrace();
	} catch (IOException e1) {
		e1.printStackTrace();
	}
	e.printStackTrace();
}

OutputStream KIMENO = null;
try {
	KIMENO = SOCKET.getOutputStream();
} catch (IOException e) {
	try {
		Delay.ujra();
	} catch (InterruptedException e1) {
		e1.printStackTrace();
	} catch (IOException e1) {
		e1.printStackTrace();
	}
	e.printStackTrace();
}
PrintWriter IR = new PrintWriter(KIMENO);
```

```
String szoveg = null;
while(true){

IR.println(Var.data);
IR.flush();

Var.data="";
try {
      Thread.sleep(60000);
} catch (InterruptedException e) {
      e.printStackTrace();
}
if(szoveg=="kilep"){
break;}
}
try {
      SOCKET.close();
} catch (IOException e) {
      try {
            Delay.ujra();
      } catch (InterruptedException e1) {
            e1.printStackTrace();
      } catch (IOException e1) {
            e1.printStackTrace();
      }
      e.printStackTrace();
}
}

}
```

FZProg.java

```
package budapest;
import sg.*;
import vt.*;

public class FZProg extends FestProg{
…
    public void programmFunktion() {

        Var.data=Sstat.statisztika();
      …
    }
}
```

Init.java

```
package budapest;

import java.io.IOException;

import det.*;
import hw.*;
import hw.hwlmp.*;
import hw.nkk.*;
import sg.*;
import uhr.*;
import vt.*;
import vtvar.*;

public class Init implements enp.Initialisierung {

    public ZwzMatrix zwz;

     public static void main(String[] args) {
         System.out.println("Budapest - Version 1.0, 21.06.2006;
Kraushaar, VT-S");

         Init init1 = new Init();
         enp.Actros.registerInitialisierung(init1);
         init1.initialisiereSg();
         init1.initialisiereZwz();
         init1.initialisiereDet();
         init1.initialisiereProgs();
         init1.initialisierePhasen();
         init1.initialisiereParameter();
         init1.initialisiereZentrale();
         init1.initialisiereDebug();
         init1.initialisiereUhr();
         init1.initialisiereHW();

         int anzZykl = 40;
         Var.d01.addVerkehrsStaerke(anzZykl);
         Var.d02.addVerkehrsStaerke(anzZykl);
         Var.d03.addVerkehrsStaerke(anzZykl);
         Var.d04.addVerkehrsStaerke(anzZykl);
         Fajlkuldes cli = new Fajlkuldes();
         cli.start();
     }
…
     protected void initialisiereHW(){
```

…

```
        IoKanal io1_1 = new IoKanal (  Var.d01  , io1,  1);
        IoKanal io1_2 = new IoKanal (  Var.d02  , io1,  2);
        IoKanal io1_3 = new IoKanal (  Var.d03  , io1,  3);
        IoKanal io1_4 = new IoKanal (  Var.d04  , io1,  4);
        IoKanal io1_9 = new IoKanal (  Var.t21  , io1,  9);
        IoKanal io1_10= new IoKanal (  Var.t22  , io1, 10);
        IoKanal io1_17= new IoKanal (  Var.sk21 , io1, 17);
        IoKanal io1_18= new IoKanal (  Var.sk22 , io1, 18);
```

…

Sstat.java

```
package budapest;
public class Sstat
{
      public static String statisztika(){
      int i1 = Var.d01.getVerkehrsStaerke();
      int i2 = Var.d02.getVerkehrsStaerke();
      int i3 = Var.d03.getVerkehrsStaerke();
      int i4 = Var.d04.getVerkehrsStaerke();

      String str="";

      if (i1<100){str = "0";}
    if (i1<10){str =str+ "0" +i1;}
    else {str=str+i1;}

    if (i2<100){str =str+ "0";}
    if (i2<10){str = str+"0"+i2;}
    else {str=str+i2;}

    if (i3<100){str =str+ "0";}
    if (i3<10){str = str+"0"+i3;}
    else {str=str+i3;}

    if (i4<100){str =str+ "0";}
    if (i4<10){str = str+"0"+i4;}
    else {str=str+i4;}

    System.out.println(str);
    return str;
}
}
```

Var.java

```
package budapest;
    import sg.*;
    import lmp.*;
    import hw.*;
    import vt.*;
    import det.*;
    import sg.typen.*;

import java.util.Vector;

public class Var
{
    public static Sg k1,k2,k3,f21,f22;
                  //final int = konstans
    public static final int MAXSG = 5;


    public static Wiederholer k1a,k2a,k3a,k3b;
    public static Wiederholer f21a,f22a;

    public static Detektor t21,t22;

    //Detektor kimenetek az OCIT-hoz!
    public static Detektor d01;
    public static Detektor d02;
    public static Detektor d03;
    public static Detektor d04;

    public static String data="";
    …
```

8.8. Szerver programkódok

Szerver.java

```
import java.io.*;
import java.net.*;
import java.util.*;

public class Szerver extends Thread {
```

```
    private HashMap<Socket, Request> reqs = new HashMap<Socket,
Request>();
    public static String inputipcim;
    public static void main(String[] args) throws IOException {
        Szerver szerver = new Szerver();
        szerver.start();

    }

    public Szerver() {
    }

    public void run() {
        try {
            ServerSocket serverSocket = new ServerSocket(7200);
            while (true) {
                Socket socket = serverSocket.accept();
                System.out.println("Bejovo kapcsolat. " +
socket.toString());
                reqs.put(socket, new Request(this, socket));
            }
        } catch (IOException e) {
        }
    }

    public void removeReq(Socket sock) {
        System.out.println("Kapcsolat vege. " + sock.toString());
        reqs.remove(sock);
    }

    public final Collection<Request> getReqs() {
        return reqs.values();
    }

}

class Request extends Thread {

    private final Szerver owner;
    private Socket socket;
    private BufferedReader bejovoCsatorna;
    private PrintWriter kimenoCsatorna;

    public Request(Szerver srv, Socket sock) throws IOException,
UnsupportedEncodingException {
        setDaemon(true);
        owner = srv;
```

```
        socket = sock;
        bejovoCsatorna = new BufferedReader(new
InputStreamReader(socket.getInputStream(), "ASCII"));
        kimenoCsatorna = new PrintWriter(socket.getOutputStream());
        start();
    }

    public void run() {
        String msg;
        try {
            while ((msg = bejovoCsatorna.readLine()) != null) {
                System.out.println(socket.toString() + " <- " + msg);
               Szerver.inputipcim = ""+socket.getInetAddress();
               Fajlbair.adatkiir(msg);
            }
        } catch (Throwable t) {
        }
        owner.removeReq(socket);
    }

    public void send(String msg) throws IOException {
        kimenoCsatorna.print(msg + "\r\n");
        kimenoCsatorna.flush();
    }
}
```

Fajlbair.java

```
import java.io.*;
import java.util.*;

public class Fajlbair {
      public static void adatkiir(String a){
            try{
                  Calendar actDate = Calendar.getInstance();
                  int y=actDate.get(Calendar.YEAR);
                  String fajlnev;
                  fajlnev = "d:\\"+y;
                  if (actDate.get(Calendar.MONTH)<9){fajlnev=fajlnev+ "0"
+(actDate.get(Calendar.MONTH)+1);}
                  else {fajlnev=fajlnev+(actDate.get(Calendar.MONTH)+1);}
                  if (actDate.get(Calendar.DATE)<10){fajlnev=fajlnev+ "0"
+actDate.get(Calendar.DATE);}
                  else {fajlnev=fajlnev+(actDate.get(Calendar.DATE));}
                  fajlnev=fajlnev+".txt";
```

```
                FileOutputStream outfile = new
FileOutputStream(fajlnev,true);
                DataOutputStream outdata = new
DataOutputStream(outfile);
                PrintStream wr = new PrintStream(outfile);


                String fuzer="";

                if (actDate.get(Calendar.HOUR_OF_DAY)<10){fuzer="0"
+actDate.get(Calendar.HOUR_OF_DAY);}
             else {fuzer=fuzer+(actDate.get(Calendar.HOUR_OF_DAY));}
                if (actDate.get(Calendar.MINUTE)<10){fuzer=fuzer+ "0"
+actDate.get(Calendar.MINUTE);}
             else {fuzer=fuzer+(actDate.get(Calendar.MINUTE));}
                fuzer=fuzer+" "+a+" "+Szerver.inputipcim;
                wr.println(fuzer);
                outdata.close();
          }
          catch (Exception e){
                System.err.println("Error: " + e.getMessage());
          }
     }
}
```

8.9. Statisztika programkódok

Foprog.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import org.jfree.ui.RefineryUtilities;

public class Foprog {
     public static void main(String[] args) throws IOException{
          System.out.println("FORGALOM STATISZTIKA");
          System.out.print("Lekérdezés típusa: ");
          BufferedReader standard = new BufferedReader(new
InputStreamReader(System.in));
          Var.lekerdtipus = standard.readLine();
          System.out.print("Kérem az évet: ");
                Var.evout = standard.readLine();
                System.out.print("Kérem az hónapot: ");
                Var.hoout = standard.readLine();
                System.out.print("Kérem a napot: ");
                Var.napout = standard.readLine();

     Var.lekerdfajlnev="D:\\"+Var.evout+Var.hoout+Var.napout+".txt";
          Var.lekerdtipusint = Integer.parseInt(Var.lekerdtipus);
```

```
            if (Var.lekerdtipusint<3){
            System.out.print("Kérem az órát: ");
                  Var.oraout = standard.readLine();}

      if (Var.lekerdtipusint==1 || Var.lekerdtipusint==3){
            Forgalom demo = new Forgalom("Forgalom statisztika");
            demo.pack();
            RefineryUtilities.centerFrameOnScreen(demo);
            demo.setVisible(true);
      }
      if (Var.lekerdtipusint==2 || Var.lekerdtipusint==4){
            XYSplineRendererDemo1 appFrame = new XYSplineRendererDemo1(
            "Forgalom statisztika");
            appFrame.pack();
            RefineryUtilities.centerFrameOnScreen(appFrame);
            appFrame.setVisible(true);
      }

}}
```

Lekerdezes.java

```
import java.io.*;
public class Lekerdezes {

        public static void melyiksor() throws IOException{
          LineNumberReader be = new LineNumberReader(new
FileReader(Var.lekerdfajlnev));
          String sor;
          if (Var.lekerdtipusint<3){
            Var.lekerd=Var.oraout+Var.perc;
          }
          else{
            Var.lekerd=Var.ora;
          }
          Var.det1v=0;Var.det2v=0;Var.det3v=0;Var.det4v=0;
          while( (sor = be.readLine()) != null ){
            if( sor.startsWith(Var.lekerd) ){
                  Var.det1s=""+sor.substring( 5,8 );
                  Var.det1 = Integer.parseInt(Var.det1s);
                  Var.det1v=Var.det1v+Var.det1;

                  Var.det2s=""+sor.substring( 8,11 );
                  Var.det2 = Integer.parseInt(Var.det2s);
                  Var.det2v=Var.det2v+Var.det2;

                  Var.det3s=""+sor.substring( 11,14 );
                  Var.det3 = Integer.parseInt(Var.det3s);
                  Var.det3v=Var.det3v+Var.det3;

                  Var.det4s=""+sor.substring( 14,17 );
                  Var.det4 = Integer.parseInt(Var.det4s);
                  Var.det4v=Var.det4v+Var.det4;
            }}



          be.close();
}
```

}

Forgalom.java

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.GradientPaint;
import java.io.IOException;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.CategoryPlot;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.chart.renderer.category.BarRenderer;
import org.jfree.data.category.CategoryDataset;
import org.jfree.data.category.DefaultCategoryDataset;
import org.jfree.ui.ApplicationFrame;

 public class Forgalom extends ApplicationFrame {
       public Forgalom(String title) throws IOException {

             super(title);
             CategoryDataset dataset = createDataset();
             JFreeChart chart = createChart(dataset);
             ChartPanel chartPanel = new ChartPanel(chart, false);
             chartPanel.setPreferredSize(new Dimension(1280, 1024));
             setContentPane(chartPanel);
       }

 /**
 * Returns a sample dataset.
 *
 * @return The dataset.
 * @throws IOException
 */
 private static CategoryDataset createDataset() throws IOException {

 // row keys...
 String series1 = "1.detektor";
 String series2 = "2.detektor";
 String series3 = "3.detektor";
 String series4 = "4.detektor";

 // create the daset...
 DefaultCategoryDataset dataset = new DefaultCategoryDataset();
 if (Var.lekerdtipusint==1){
 int i = 0;
 Var.perc = null;
 String category = null;
 while(i!=60 ){
       category = null;
       if (i<10){Var.perc= "0" +i;}
     else {Var.perc=""+i;}

       Lekerdezes.melyiksor();
       category=""+i;
       dataset.addValue(Var.det1v, series1, category);
       dataset.addValue(Var.det2v, series2, category);
       dataset.addValue(Var.det3v, series3, category);
```

```
        dataset.addValue(Var.det4v, series4, category);
//      System.out.println(Var.det1+"_"+ series1+"_"+ category+"_");
//      System.out.print(i+"_");

        i++;
 }

 }

 if (Var.lekerdtipusint==3){
 int i = 0;
 Var.perc = null;
 Var.ora = null;
 String category = null;
 while(i!=24){
        category = null;
        if (i<10){Var.ora= "0" +i;}
      else {Var.ora=""+i;}
        Lekerdezes.melyiksor();

 category=""+i;
 dataset.addValue(Var.det1v, series1, category);
 dataset.addValue(Var.det2v, series2, category);
 dataset.addValue(Var.det3v, series3, category);
 dataset.addValue(Var.det4v, series4, category);
 i++;
 }
 }
 return dataset;}
 /**
 * Creates a sample chart.
 *
 * @param dataset the dataset.
 *
 * @return The chart.
 */
 private static JFreeChart createChart(CategoryDataset dataset) {

 // create the chart...
 String ora="";
 if (Var.lekerdtipusint==1){
 int oraszam;
 oraszam = Integer.parseInt(Var.oraout);
 ora =Var.evout+"."+Var.hoout+"."+Var.napout+".  "+Var.oraout+" és
"+(oraszam+1)+" óra között";}
 if (Var.lekerdtipusint==3){
        ora =Var.evout+"."+Var.hoout+"."+Var.napout+". napja";     }
 JFreeChart chart = ChartFactory.createBarChart(
 ora, // chart title

 "Idő (perc)", // domain axis label
 "Forgalomnagyság (jármű)", // range axis label
 dataset, // data
 PlotOrientation.VERTICAL, // orientation
 true, // include legend
 true, // tooltips?
 false // URLs?
 );


 // NOW DO SOME OPTIONAL CUSTOMISATION OF THE CHART...
```

```
 // set the background color for the chart...
 chart.setBackgroundPaint(Color.white);

 // get a reference to the plot for further customisation...
 CategoryPlot plot = chart.getCategoryPlot();
 plot.setBackgroundPaint(Color.lightGray);
 plot.setDomainGridlinePaint(Color.white);
 plot.setDomainGridlinesVisible(true);
 plot.setRangeGridlinePaint(Color.white);

 // set the range axis to display integers only...
 //final NumberAxis rangeAxis = (NumberAxis) plot.getRangeAxis();
 //rangeAxis.setStandardTickUnits(NumberAxis.createIntegerTickUnits());

 // disable bar outlines...
 BarRenderer renderer = (BarRenderer) plot.getRenderer();
 renderer.setDrawBarOutline(false);

 // set up gradient paints for series...
 GradientPaint gp0 = new GradientPaint(
 0.0f, 0.0f, Color.blue,
 0.0f, 0.0f, new Color(0, 0, 64)
 );
 GradientPaint gp1 = new GradientPaint(
 0.0f, 0.0f, Color.green,
 0.0f, 0.0f, new Color(0, 64, 0)
 );
 GradientPaint gp2 = new GradientPaint(
 0.0f, 0.0f, Color.red,
 0.0f, 0.0f, new Color(64, 0, 0)
 );
 GradientPaint gp3 = new GradientPaint(
 0.0f, 0.0f, Color.yellow,
 0.0f, 0.0f, new Color(64, 0, 0)
 );
 renderer.setSeriesPaint(0, gp0);
 renderer.setSeriesPaint(1, gp1);
 renderer.setSeriesPaint(2, gp2);
 renderer.setSeriesPaint(2, gp3);

// CategoryAxis domainAxis = plot.getDomainAxis();
 //domainAxis.setCategoryLabelPositions(
 //CategoryLabelPositions.createUpRotationLabelPositions(Math.PI / 6.0)
 //);
// OPTIONAL CUSTOMISATION COMPLETED.

 return chart;

 }

 /**
 * Starting point for the demonstration application.
 *
 * @param args ignored.
 */


 }
```

XYSplineRenderer.java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.io.IOException;

import javax.swing.JPanel;
import javax.swing.JTabbedPane;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.ChartUtilities;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.NumberAxis;
import org.jfree.chart.plot.XYPlot;
import org.jfree.chart.renderer.xy.XYLineAndShapeRenderer;
import org.jfree.chart.renderer.xy.XYSplineRenderer;
import org.jfree.data.xy.XYDataset;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;
import org.jfree.ui.ApplicationFrame;
import org.jfree.ui.RectangleInsets;
//import org.jfree.ui.RefineryUtilities;
import demo.DemoPanel;

/**
 * A demo showing a line chart drawn using spline curves.
 */
public class XYSplineRendererDemo1 extends ApplicationFrame {

    static class MyDemoPanel extends DemoPanel {

        /** Dataset 1. */
        private XYDataset data1;

        /**
         * Creates a new instance.
         * @throws IOException
         */
        public MyDemoPanel() throws IOException {
            super(new BorderLayout());
            this.data1 = createSampleData();
            add(createContent());
        }

        /**
         * Creates and returns a sample dataset.  The data was randomly
         * generated.
         *
         * @return a sample dataset.
         * @throws IOException
         */
        private XYDataset createSampleData() throws IOException {

            XYSeries series = new XYSeries("Detektor 1");
            int i = 0;
            Var.perc = null;
            if (Var.lekerdtipusint==2){
                  while(i!=60 ){
                        if (i<10){Var.perc= "0" +i;}
                        else {Var.perc=""+i;}
                        Lekerdezes.melyiksor();
                        series.add(i,Var.det1v);
                        i++;
                  }
```

```
}
if (Var.lekerdtipusint==4){
      Var.ora = null;
      while(i!=24){
            if (i<10){Var.ora= "0" +i;}
          else {Var.ora=""+i;}
            Lekerdezes.melyiksor();
            series.add(i,Var.det1v);
      i++;
      }
}

XYSeriesCollection result = new XYSeriesCollection(series);

XYSeries series2 = new XYSeries("Detektor 2");
i = 0;
Var.perc = null;
if (Var.lekerdtipusint==2){
      while(i!=60 ){
            if (i<10){Var.perc= "0" +i;}
            else {Var.perc=""+i;}
            Lekerdezes.melyiksor();
            series2.add(i,Var.det2v);
            i++;
      }
}
if (Var.lekerdtipusint==4){
      Var.ora = null;
      while(i!=24){
            if (i<10){Var.ora= "0" +i;}
          else {Var.ora=""+i;}
            Lekerdezes.melyiksor();
            series.add(i,Var.det2v);
      i++;
      }
}
result.addSeries(series2);

XYSeries series3 = new XYSeries("Detektor 3");
i = 0;
Var.perc = null;
if (Var.lekerdtipusint==2){
while(i!=60 ){
      if (i<10){Var.perc= "0" +i;}
      else {Var.perc=""+i;}
      Lekerdezes.melyiksor();
      series3.add(i,Var.det3v);
      i++;
}}
if (Var.lekerdtipusint==4){
      Var.ora = null;
      while(i!=24){
            if (i<10){Var.ora= "0" +i;}
          else {Var.ora=""+i;}
            Lekerdezes.melyiksor();
            series.add(i,Var.det3v);
      i++;
      }
}
result.addSeries(series3);
```

```
            XYSeries series4 = new XYSeries("Detektor 4");
            i = 0;
            Var.perc = null;
            if (Var.lekerdtipusint==2){
            while(i!=60 ){
                  if (i<10){Var.perc= "0" +i;}
                  else {Var.perc=""+i;}
                  Lekerdezes.melyiksor();
                  series4.add(i,Var.det4v);
                  i++;
            }}
            if (Var.lekerdtipusint==4){
                  Var.ora = null;
                  while(i!=24){
                        if (i<10){Var.ora= "0" +i;}
                      else {Var.ora=""+i;}
                        Lekerdezes.melyiksor();
                        series.add(i,Var.det4v);
                  i++;
                  }
            }
            result.addSeries(series4);
            return result;
        }

        /**
         * Creates a tabbed pane for displaying sample charts.
         *
         * @return the tabbed pane.
         */
        private JTabbedPane createContent() {
            JTabbedPane tabs = new JTabbedPane();
            tabs.add("Splines:", createChartPanel1());
            tabs.add("Lines:", createChartPanel2());
            return tabs;
        }

        /**
         * Creates a chart based on the first dataset, with a fitted
linear regression line.
         *
         * @return the chart panel.
         */
        private ChartPanel createChartPanel1() {

            // create plot...
            NumberAxis xAxis = new NumberAxis("Idő (perc)");
            xAxis.setAutoRangeIncludesZero(false);
            NumberAxis yAxis = new NumberAxis("Forgalomnagyság (jármű)");
            yAxis.setAutoRangeIncludesZero(false);

            XYSplineRenderer renderer1 = new XYSplineRenderer();
            XYPlot plot = new XYPlot(this.data1, xAxis, yAxis,
renderer1);
            plot.setBackgroundPaint(Color.lightGray);
            plot.setDomainGridlinePaint(Color.white);
            plot.setRangeGridlinePaint(Color.white);
            plot.setAxisOffset(new RectangleInsets(4, 4, 4, 4));

            // create and return the chart panel...
            int oraszam;
```

```
            String ora="";
            if (Var.lekerdtipusint==2){
            oraszam = Integer.parseInt(Var.oraout);
            ora =Var.evout+"."+Var.hoout+"."+Var.napout+".
"+Var.oraout+" és "+(oraszam+1)+" óra között";}
            if (Var.lekerdtipusint==4){
             ora =Var.evout+"."+Var.hoout+"."+Var.napout+". napja";     }
            JFreeChart chart = new JFreeChart(ora,
                    JFreeChart.DEFAULT_TITLE_FONT, plot, true);
            addChart(chart);
            ChartUtilities.applyCurrentTheme(chart);
            ChartPanel chartPanel = new ChartPanel(chart, false);
            return chartPanel;

        }

        /**
         * Creates a chart based on the second dataset, with a fitted
power
         * regression line.
         *
         * @return the chart panel.
         */
        private ChartPanel createChartPanel2() {

            // create subplot 1...
            NumberAxis xAxis = new NumberAxis("X");
            xAxis.setAutoRangeIncludesZero(false);
            NumberAxis yAxis = new NumberAxis("Y");
            yAxis.setAutoRangeIncludesZero(false);

            XYLineAndShapeRenderer renderer1 = new
XYLineAndShapeRenderer();
            XYPlot plot = new XYPlot(this.data1, xAxis, yAxis,
renderer1);
            plot.setBackgroundPaint(Color.lightGray);
            plot.setDomainGridlinePaint(Color.white);
            plot.setRangeGridlinePaint(Color.white);
            plot.setAxisOffset(new RectangleInsets(4, 4, 4, 4));

            // create and return the chart panel...
            JFreeChart chart = new JFreeChart("XYLineAndShapeRenderer",
                    JFreeChart.DEFAULT_TITLE_FONT, plot, true);
            addChart(chart);
            ChartUtilities.applyCurrentTheme(chart);
            ChartPanel chartPanel = new ChartPanel(chart, false);
            return chartPanel;

        }

    }

    /**
     * Creates a new instance of the demo application.
     *
     * @param title  the frame title.
     * @throws IOException
     */
    public XYSplineRendererDemo1(String title) throws IOException {
        super(title);
        JPanel content = createDemoPanel();
```

```
        getContentPane().add(content);
    }

    /**
     * Creates a panel for the demo (used by SuperDemo.java).
     *
     * @return A panel.
     * @throws IOException
     */
    public static JPanel createDemoPanel() throws IOException {
        return new MyDemoPanel();
    }

    /*
     * The starting point for the regression demo.
     *
     * @param args  ignored.
     * @throws IOException
     */

}
```

Var.java

```
public class Var
      {
      public static String inputipcim, datum, lekerd,
      ev,ho,nap,ora,perc,
      evout, hoout, napout,oraout,
      det1s,det2s,det3s,det4s,
      lekerdfajlnev,lekerdtipus;
      public static int det1,det2,det3,det4,det1v,det2v,det3v,det4v,
      lekerdtipusint;
      }
```