Netspotter

Rendszerterv

írta

Dojcsák Sándor
Kasza Miklós
Roszik György Attila

BEVEZETÉS

Ez a dokumentum a *Netspotter* szoftverrendszer felépítésének és működésének leírását tartalmazza. A Netspotter rendszer *IPv6 multicast* hálózatok monitorozása céljából készült. Ezt a feladatot az *SNMP-protokoll* felhasználásával valósítja meg.

A rendszer továbbá lehetőséget biztosít tetszőleges méretű hálózat funkcionális és terhelés tesztelésére tetszőleges forgalommal, különös tekintettel az IPv6 multicast forgalomra.

A rendszer a fent említett feladatainak elvégzésére elosztott működési modellt alkalmaz és a következő technológiákat használja:

- *Java 2 Standard Edition 5.0*
- *LibPcab* és *WinPcap* könyvtárak az alacsony szintű hálózati forgalom kezelésére
- saját C függvénykönyvtár és *Java Native Interface* további alacsony szintű hálózati feladatokra
- *XML*, *XML Schema* és *XSLT* a belső adatreprezentáció kezelésére, valamint ezen technológiákat kezelni tudó külső programcsomagok (*Saxon*, *XStream*, stb.)
- *SNMP4J* külső programcsomag az SNMP-specifikus feladatok ellátására
- *Prefuse* külső programcsomag a felderített topológia grafikus megjelenítéséhez
- *Quartz* feladatütemező külső programcsomag az ügynökök feladatainak időzítéséhez
- *Sun Java System Application Server 9.1 (Glassfish v2)* és az általa támogatott *EJB 3.0* szabvány az alkalmazás szerver komponensének elkészítéséhez

TARTALOMJEGYZÉK

Bevezetés....1
Tartalomjegyzék....2
A rendszer részei....4
GUI Applet....8
Az applet célja....8
Az applet működése nagy vonalakban....8
A szerver....11
A szerver céljai....11
Agent....12
Az ügynökök céljai....12
Az ügynökök működése nagy vonalakban....12
A rendszer részletes működése....13
Kliens oldal....14
Automatikusan generált kliens oldali proxy osztályok....14
Segédosztályok és az absztrakciós réteg....14
GUI Applet....15
Ügynökök....15
Szerver oldal....16
A szerver oldal felépítése....16
Adattároló osztályok (egyedek)....16
Műveleti osztályok (session beanek)....16
Webszolgáltatások....17
Automatikusan generált szerver oldali proxy osztályok....21
Perzisztencia....21
Függőségbeszúrás és CMP....22
Ütemezett feladatok....22
Feladat (Task)....23
FeladatIdőzítés (TaskSchedule)....23
FeladatArgumentum (TaskArgument)....23
FeladatArgumentumÉrték (TaskArgumentValue)....23
FeladatPéldány (TaskInstance)....24
Távoli fájlok....24
FájlLeíró....24

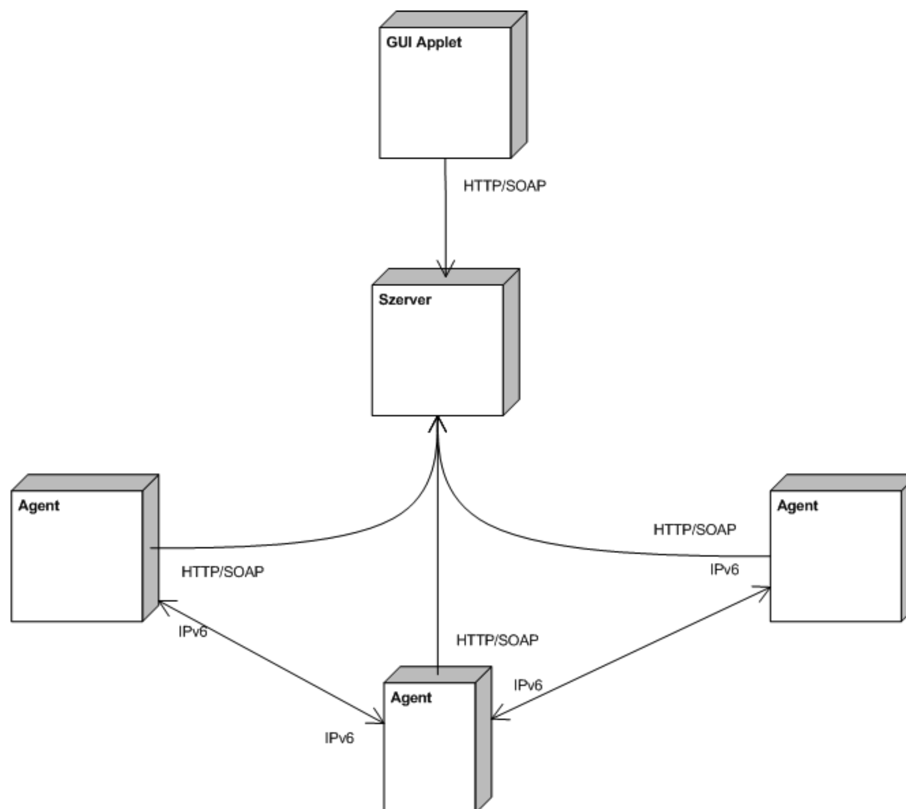
FájlTartalom....25
Hálózati csomópontok....25
Feladatütemezés....25
Bővíthető tevékenységhierarchia....25
EJB3 beépített időzítő....25
Távoli fájlrendszer működése....29
Működési elv és elérési útvonalak szintaxisa....29
Szerkezeti felépítés....29
Topológia-felderítés....30
A felderítési folyamat lépései....30
A felderítő alrendszer felépítése....30
Ügynökök kezelése....38
Hálózati információ kinyerése....38
A WinPcap és a LibPcap függvénykönyvtár....38
A csomagkapcsolt kommunikáció és az üzenetek szerkezetének ábrázolása....38
A Sequence editor modul felépítése és feladatai....46
Az Agent logika által felhasznált hálózati réteg....48

A RENDSZER RÉSZEI

Mint a bevezetésben olvasható a rendszer elosztott modell felhasználásával valósítja meg feladatait. Ennek megfelelően három különböző komponenst különböztethetünk meg:

- **GUI Applet:** felhasználói és információmegjelenítő felületként funkcionál
- **Ügynök (agent):** a többi ügynöknek adatokat küld, illetve tőlük adatokat fogad.
- **Szerver:** tárolja az adatokat, feladatokat oszt az ügynököknek, biztosítja a felhasználói felület által megjelenített információt

E három komponens kapcsolatát az *1. ábra* mutatja be. A részleteket a következő fejezetek tartalmazzák.



1. ábra: A Netspotter legfelső szintű komponensei

GUI APPLET

AZ APPLET CÉLJA

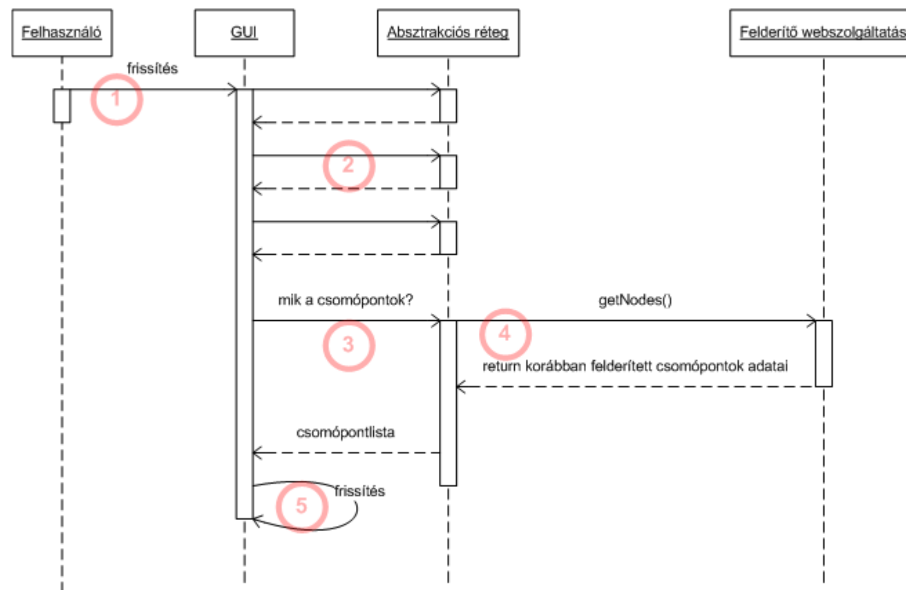
A rendszer felhasználói felületét egy tetszőleges (Java appleteket támogató) webböngészőből használható *Java Swing applet* valósítja meg. Ez az applet az egyszerű információmegjelenítésen túl további összetett feladatokat lát el a használhatóság érdekében.

Lehetőséget biztosít a rendszer által felderített hálózati topológia szemléletes, grafikus megjelenítésére; az ügynökök által elvégzendő feladatok pontos leírására (összetett szerkesztőfelület formájában) és minden olyan érték beállítására, amire a rendszernek a helyes működéshez szüksége van.

Az applet működése nagy vonalakban

Az aláírt applet egy Java *Jar* fájlban található, az általa használt külső modulok pedig szintén *Jar* fájlok formájában kapnak helyet az applet *mellett*. A GUI az alkalmazás központi részével (a *szerverrel*) *webszolgáltatásokon* (HTTP/SOAP) keresztül kommunikál. A webszolgáltatások ugyanis nagyon laza kötést tesznek lehetővé a szolgáltató (szerver) és az ügyfél (GUI Applet) között, ami lehetővé teszi, hogy a későbbiek folyamán teljesen más (akár nem Java) technológiát alkalmazva lehessen az alkalmazáshoz grafikus felhasználói felületet készíteni.

Az applet és a szerver együttműködését a *2. ábra* szemlélteti egy példán (az ismert hálózati csomópontok adatainak frissítése) keresztül. Természetesen ilyen rövid példában nagyon sok részlet rejtve marad, amiket e dokumentum későbbi fejezetei tárgyalnak.



2. ábra: a felhasználói felület és a szerver együttműködése

Az ábrán szerepel egy ún. *absztrakciós réteg* is. Ez egy logikai fogalom, ami magába foglalja azokat a segédosztályokat (automatikusan generált és kézzel írott osztályokat egyaránt), amelyek az applet és a webszolgáltatás egyszerűbb együttműködése érdekében készültek.

Az ábrán a következő lépések vannak számozott piros körökkel jelölve:

1. **Felhasználói interakció.** A felhasználó egy adott funkció elvégzését kéri valamely grafikus elem kiválasztásával (pl. Frissítés gomb)

2. **Hívás előkészítése.** Az applet egy közbülső absztrakciós réteg felhasználásával előkészíti az aktuálisan használni kívánt webszolgáltatás hívását. Ez magában foglalja a paraméterek webszolgáltatás által használható alakra alakítását is.

3. **Hívás kérése.** Az applet szintén az absztrakciós rétegnek jelzi, hogy milyen információra van szüksége, és hogy a korábban előkészített paraméterek közül melyeket kívánja a webszolgáltatás felé továbbítani.

4. **Távoli hívás lebonyolítása.** Az absztrakciós réteg meghívja a távoli webszolgáltatást a megadott paraméterekkel, majd az eredményt az applet által kezelhető formában adja vissza.

5. **Módosítások (kért feladat) elvégzése „helyileg”.** Az applet az absztrakciós réteg által visszaadott eredményt felhasználva módosítja a megjelenített tartalmat.

A SZERVER

A szerver céljai

A szerver legfőbb célja, hogy központi adatbázist és azt kezelő műveleteket biztosítson mind a felhasználói felület, mind az ügynökök felé. Természetesen az ügynökök teljesen más műveleteket igényelnek, mint a felhasználói felület, ezért vannak olyan funkciók, amelyeket kizárólag az ügynökök és vannak olyanok, amelyeket kizárólag a felhasználói felület használnak.

Másik nagyon fontos célja a szervernek, hogy az általa kezelt hálózati csomópont-adatbázist önállóan tudja frissíteni különböző kritériumok alapján, tehát önálló logikával is rendelkezik (Ilyen például az adott időnként végzett topológia-felderítés. Ez a műveletsor ugyanis kizárólag a szerveren történik, a többi komponens csak a művelet eredményét kapja meg.)

AGENT

Az ügynökök céljai

Az ügynökök feladata, hogy egyes hálózati csomópontokon futva megfelelő IPv6 hálózati forgalmat generáljanak valamint a más ügynökök által generált forgalom rájuk vonatkozó részét figyeljék. Az ügynökök a szervertől kapják meg, hogy milyen hálózati

csomagokat kell kiküldeniük és milyeneket fogadniuk. Feladatvégzés után annak eredményeit ugyanígy a szervernek kell elküldeniük.

A RENDSZER RÉSZLETES MŰKÖDÉSE

KLIENS OLDAL

A rendszer kliens oldala a GUI appletből és az ügynökökből áll, hiszen ezek azok a komponensek, amelyek legfőként a hagyományos „asztali” Java osztályokat használják.

AUTOMATIKUSAN GENERÁLT KLIENS OLDALI PROXY OSZTÁLYOK

A kliens oldali komponensek legalacsonyabb szintű kapcsolatát a szerver oldallal az ún. automatikusan generált kliens oldali proxy osztályok jelentik. Ezek olyan annotált Java osztályok, amelyeket adott segédeszközökkel (**wsimport** nevű program) hozhatunk létre a szerveren futó webszolgáltatások leírásai (WSDL és XSD leíró dokumentumok) alapján.

Ezen osztályok legfőbb célja kettős: először is elrejtik a programozó elől a webszolgáltatáson keresztül felhasznált XML-ábrázolás részleteit (egyedtároló osztályok), másodszor pedig a teljes távoli hívási mechanizmust becsomagolják segédosztályokba, amelyeket később úgy használhatunk, mintha a webszolgáltatást megvalósító osztály közvetlenül a kliensen lenne.

SEGÉDOSZTÁLYOK ÉS AZ ABSZTRAKCIÓS RÉTEG

Az automatikusan generált osztályok tulajdonsága, hogy a webszolgáltatás interfészének megváltozásakor újra kell őket generálni. Ez még nem lenne hátrány, az viszont már mindenképpen az, hogy újragenerálás esetén az ezekre az osztályokra hivatkozó kódrészletek érvénytelenné válhatnak. Ez egyedtároló objektumok esetén még hasznos is lehet, hiszen egy egyed esetén nincs értelme egy nem létező tulajdonságra hivatkozni, vagy egy tulajdonságnak más típust feltételezni, mint ami valójában, így viszont a programozók nem hagyhatják figyelmen kívül a változásokat, mert akkor nem tudnák az érintett osztályokat lefordítani. Viszont a távoli hívásokat

megvalósító proxy osztályokra való közvetlen hivatkozás nagyon érzékennyé teheti a kliens oldali kódot az egyes webszolgáltatások interfészének változásaira. Ezen felül pedig az automatikusan generált osztályok nagy hátránya, hogy automatikusan nem származtathatók kliens oldali interfészekből vagy osztályokból, hanem újragenerálás után ezeket az öröklési kapcsolatokat kézzel kell beállítani.

E problémák elkerülésére a kliens oldali kód egy részét képezik azok a segédosztályok, amelyek becsomagolják az automatikusan generált kliens oldali proxy osztályokat. Ezek a segédosztályok ugyanis már származtathatók kliens oldali interfészekből vagy osztályokból, ezért nagyobb az (ilyen segédosztályokat használó) kliens oldali kód rugalmassága, valamint a webszolgáltatás interfészének változása esetén is gyakran elegendő csak a megfelelő segédosztály módosítása.

A most említett segédosztályok és az előzőleg tárgyalt automatikusan generált kliens oldali proxy osztályok képezik az absztrakciós réteget. Ez tehát egy logikai csoport a kliens oldali osztályok közül, melyek azt hivatottak elősegíteni, hogy a kliens oldal rugalmas legyen és minél toleránsabb a szerver oldali módosításokkal szemben.

GUI APPLET

Topológia-felderítés

A topológia-felderítés szerkezetét a *10. ábra* szemlélteti. A **MainFrame** modul a kliens oldali GUI kezelésért felelős, tulajdonképpen egy **JFrame**-ből származtatott osztály. Az alkalmazás kliens oldali fő szálában ez a kód fut. Mivel a hozzáférési listát a szerver taszk ütemezésének kliens oldali vezérlője küldi el szervernek webszolgáltatáson keresztül, ezért a **MainFrame** komponens egyedül a csomópontok lekérését teszi meg a **ProxyLogic** modulon keresztül, ezenkívül csak a megjelenítésért felelős. A topológia grafikus megjelenítésben felhasználja a **Topology** modul szolgáltatásait, mely a *Prefuse API*-ra épít. A **Topology** modul egy külön szálat indít a topológia karbantartására.

Szekvenciaszerkesztő

A Sequence editor modul felépítése és feladatai

A szekvenciaszerkesztő feladata biztosítani egy kényelmesen és könnyen kezelhető felületet, amely segítségével kommunikációs folyamatok modellezhetők. Működése alapvetően a következő: a grafikus felületen végrehajtott egyes műveleteknek megfelelően a szekvencia XML karbantartásáért felelő osztály elvégzi a szükséges műveleteket és változtatásokat az XML-ben, majd a grafikus felület az új szekvencia XML alapján megjeleníti az aktuális szekvenciát.

A szekvenciaszerkesztő felületnek a következő szolgáltatásokat kell nyújtania:

- A szekvenciaszerkesztő feladata üzenetszekvenciák készítése illetve szerkesztése:
    1. Egyedek definiálása és törlése, áthelyezése
    2. Az egyedek közé üzeneteket reprezentáló nyilak definiálása, törlése, áthelyezése
    3. Minden üzenethez megadható legyen egy előre elkészített üzenetszerkezet, amelynek érték mezőit kitöltve kapunk egy teljesen specifikált üzenetet, továbbá olvasott üzenetnél lehetőségünk legyen bármilyen (ANY) értékű mezőket specifikálni.
    4. Az olvasott üzenetekhez időtúllépést lehessen rendelni, amely megmondja, hogy mennyi ideig kell várni a beérkezésére.
- A cím-alias kezelő feladata, hogy a címekhez (főleg a hosszú IPv6-os címek miatt) aliast lehessen rendelni, ezzel könnyítve a felhasználó dolgát. Valamint

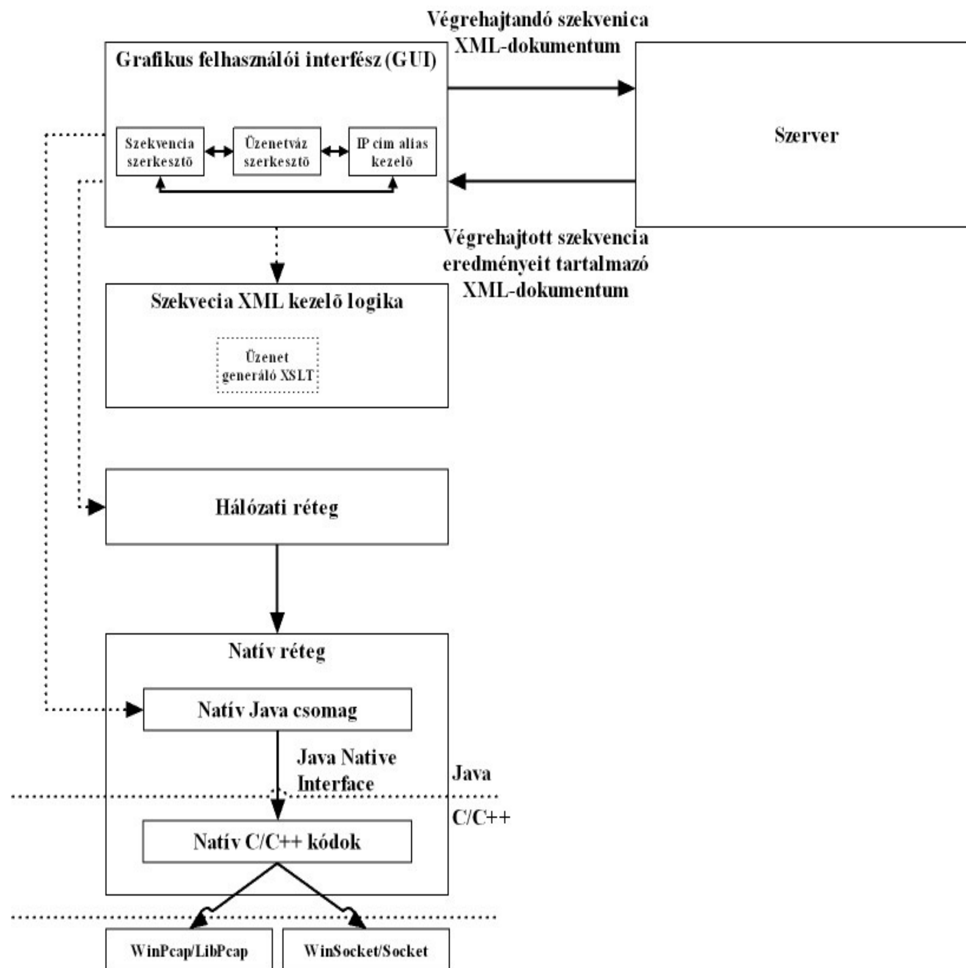
külön oldalakon tárolja az egyes agentek címeit, növelve ezzel az áttekinthetőséget.

- Az üzenetváz szerkesztő feladata üzenetvázak készítése és szerkesztése:

    1. Tetszőleges üzenet szerkezetének felépítése.
    2. Minden üzenet mezőkből áll, továbbá mindegyik tartalmaz egy *body* mezőt, amely vagy az adott üzenet adattagját tartalmazza, vagy egy másik üzenetet, vagyis egy beágyazást.
    3. Az üzenetekhez speciális mezők létrehozása. Ellenőrző összeget vagy üzenet hosszt tartalmazó mező létrehozása.
    4. Új üzenetvázak létrehozásának biztosítása az üzenetvázak között öröklődéssel.
    5. Az üzenetvázak XML-ben történő tárolása.

A szekvencia XML karbantartásáért felelős osztály szolgáltatásai ennek megfelelően a következők:

- Egyedek leírásának beszúrása, törlése, átmozgatása a szekvencia XML-ben a megfelelő helyen.
- Üzenetek leírásának beszúrása, törlése, átmozgatása a szekvencia XML-ben a megfelelő helyen.
- Üzenet tartalom leírásának beszúrása, törlése, átmozgatása a szekvencia XML-ben.



3. ábra: A szekvenciaszerkesztő felépítése

Az Agent logika által felhasznált hálózati réteg

Feladata olyan hálózati szolgáltatások nyújtása, amely segítségével megvalósítható adathálózati protokollok tesztelése. Az alapvető cél tetszőleges üzenet olvasásának és kiküldésének biztosítása.

A létrehozott kommunikációs folyamat leírását XML-dokumentumot feldolgozva elő kell állítania a dokumentumban specifikált üzenetek bájttömbjét, illetve kinyerni az üzenetekhez tartozó egyéb információkat, majd átadnia a natív réteg számára. Ezután a natív réteg elvégzi a szekvenciának megfelelően az üzenetek küldését, illetve olvasását.

A hálózati réteg osztályai a *net* csomagban találhatóak meg. Röviden tekintsük át a fontosabb osztályokat.

A legtöbb osztály őse a *network* osztály, amelyben az alapvető műveleteket tartalmazza: bit és bájt manipulációk és a CRC32 ellenőrző összeg számító algoritmus.

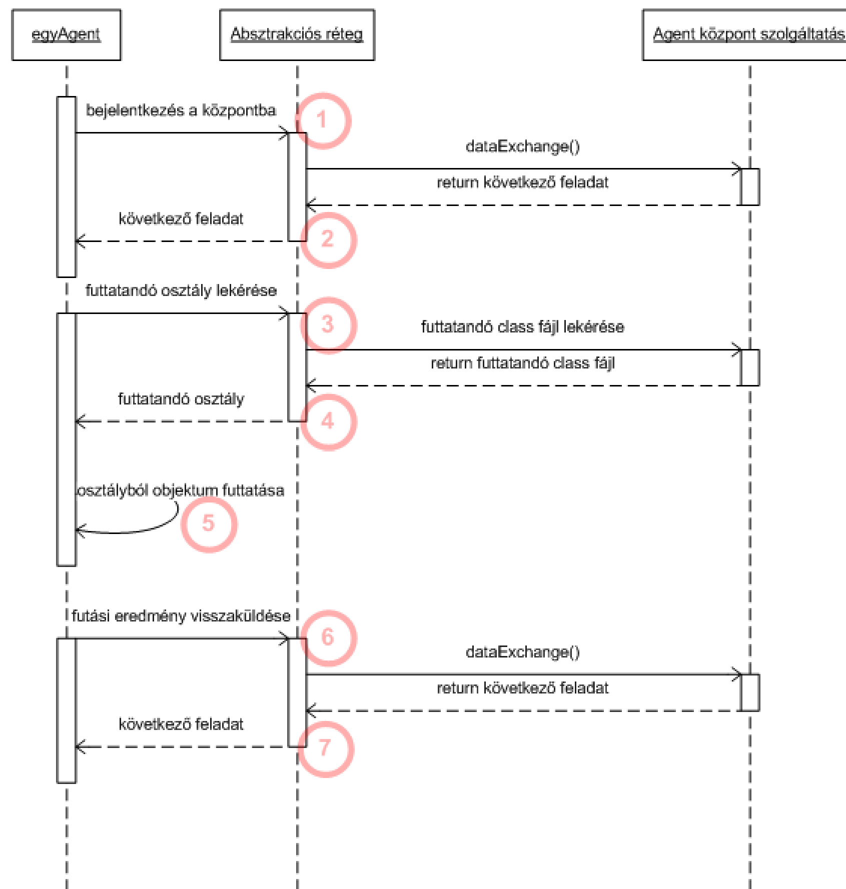
A *MACaddress*, az *ip4address* és az *ip6address* osztályok a különböző címek kezelését végzik.

A kimaradt osztályok az általános üzenetszerkezet bájt tömbjének és a visszajelzés XML-dokumentumainak előállításáért felelős osztályok.

ÜGYNÖKÖK

Az ügynökök a következő módon végzik feladatukat.

1. **Az ügynök jelenlétének jelzése a központ felé.** Első lépésként az ügynökök jelzik a szerver számára, hogy aktívak. Ezt adott időközönként meg kell tenniük ahhoz, hogy a szerver „tudomást vegyen róluk”, azaz felhasználhatónak ítélje az adott ügynököt.
2. **Következő tevékenység kiosztása az ügynöknek.** Az ügynök első bejelentkezése eredményeként megkapja a számára kiosztott feladatot.
3. **Az ügynök által futtatandó osztály lekérése.** Ha az ügynök által végzendő feladat egy adott szekvencia végrehajtása, akkor az ügynök az absztrakciós réteg közreműködésével lekéri a szerverről azt a – szerver által generált – Java osztályt, amelyik a végrehajtandó szekvencia utasításait tartalmazza.
4. **Az ügynök által futtatandó osztály betöltése.** Az absztrakciós réteg visszaadja az ügynöknek a számára kiosztott (és betöltött) osztályt.
5. **Szekvencia végrehajtása.** Az ügynök végrehajtja a számára kiosztott szekvenciát, amit a kapott osztály egy adott metódusának futtatásával ér el.



4. ábra: ügynökök feladatainak kiosztása

6. **Szekvencia eredményeinek jelentése.** Amikor az ügynök befejezte a szekvenciának megfelelő osztály adott metódusának futtatását, létrejön a szekvencia végrehajtásának eredménye. Tárolás és későbbi megjelenítés céljából a végrehajtás eredményét visszaküldi a szervernek az absztrakciós rétegen keresztül.

   Az eredmények visszaküldése a szerver számára azt is jelenti, hogy az ügynök még aktív, azaz az *1. pontban* leírt *bejelentkeztetést* is elvégezheti egyúttal.

7. **Következő tevékenység kiosztása.** Amennyiben rendelkezésre áll konkrét következő utasítás az adott ügynök számára, úgy az vissza lehet küldeni az általa

elvégezendő következő feladatot. Amennyiben nincs következő utasítás, az ügynök várakozik további feladatokra.

SZERVER OLDAL

A SZERVER OLDAL FELÉPÍTÉSE

Adattároló osztályok (egyedek)

Minden olyan szerveren tárolt adat, amelyet hosszú távon (akár a szerver újraindítása után is) meg kell őrizni, egyedobjektumok (*entity-k*) által tárolódik. Az egyedek adatbázisba (és onnan) történő szinkronizálását az alkalmazásszerver EJB-konténere illetve perzisztencia-kezelője végzi (*Container Managed Persistence*).

Az egyed-osztályok hagyományos *JavaBean* osztályok, azaz megadott szabályokat követnek. Ez érinti a tulajdonságokat meghatározó *get/set* metóduspárokat, és még néhány más szabályozást. Azokat az egyedtulajdonságokat, amelyeket nem lehet (vagy nem célszerű) ilyen hagyományos úton meghatározni, a Java 5-ös (és ezzel együtt az EJB 3.0-ás) verziójában bevezetett *annotációk* segítségével lehet megadni. Ilyenek például az egyedeket tároló adattáblák és a bennük szereplő attribútumok nevei, az egyedek közötti kapcsolatok és még jó néhány egyéb speciális tulajdonság.

Műveleti osztályok (session beanek)

A szerver oldali logika megvalósítása főként *állapotmentes (stateless) session beanekkel* történik. Ezek legfőbb feladata az egyedek kezelése, de más funkciókat is ellátnak. A különböző session beanek feladatai a következő fejezetben kerülnek leírásra.

Az említett session beanek kizárólag a szerver alkalmazáson belülről elérhető komponensek. A rendszer többi részével való kapcsolattartás a következő fejezetben tárgyalt webszolgáltatások feladata.

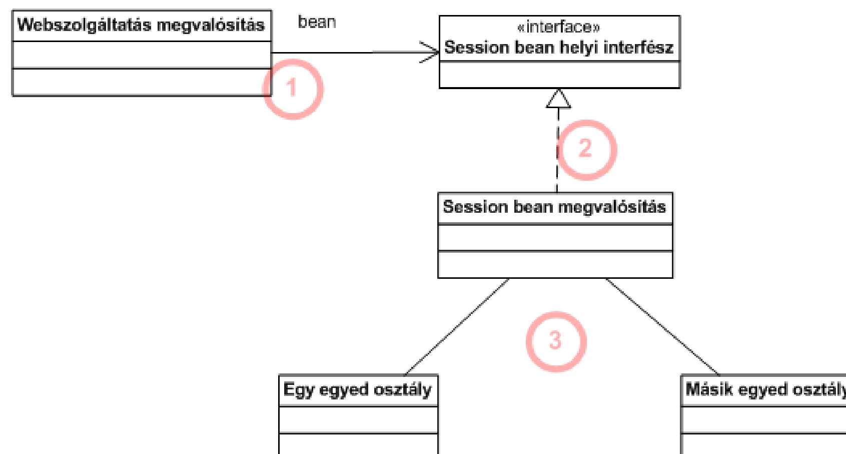
Webszolgáltatások

A szerver komponens és a rendszer többi komponense (GUI applet, ügynökök) közötti kapcsolat ***webszolgáltatások*** *segítségével* van megvalósítva. Mint már korábban

szerepelt, ennek előnye, hogy *nagyon laza kötést* tesz lehetővé a szerver és a funkcióit használó kliens komponensek között. Ennek következménye, hogy a szolgáltatások felhasználása platform- és programozási nyelv független, bár kétségtelen, hogy a megfelelő Java nyelvű kliens oldali segédosztályok megléte a Java platform használatát sugallja a kliensek platformjaként. Elvi lehetőségként azonban megmarad például a felhasználói felület önálló *asztali* alkalmazásként történő megvalósítása (például akár C++ nyelven is).

Fontos megjegyezni, hogy a webszolgáltatások semmilyen *feldolgozásra vonatkozó* szerver oldali *logikát nem tartalmaznak*. Egyedüli céljuk, hogy a műveleteket végző belső session beanek távoli eléréséhez *interfészt biztosítsanak*. Így tulajdonképpen a webszolgáltatások homlokzatként szolgálnak a szerver alkalmazás többi részéhez (ezek alapján ez a kialakítás nevezhető *web service facade-nak*, azaz ***webszolgáltatás homlokzatnak***).

A webszolgáltatás homlokzat egyszerűsített szerkezetét a következő ábra szemlélteti. Az ábra alapján az egyszerűsített szerkezet a következőképpen épül fel.



5. ábra: az egyszerű webszolgáltatás homlokzat elrendezés szerkezete

1. **Session bean helyi interfész metódus továbbítása.** Az előzőek alapján tehát a webszolgáltatás megvalósítás tartalmaz egy *hivatkozást* egy megfelelő helyi

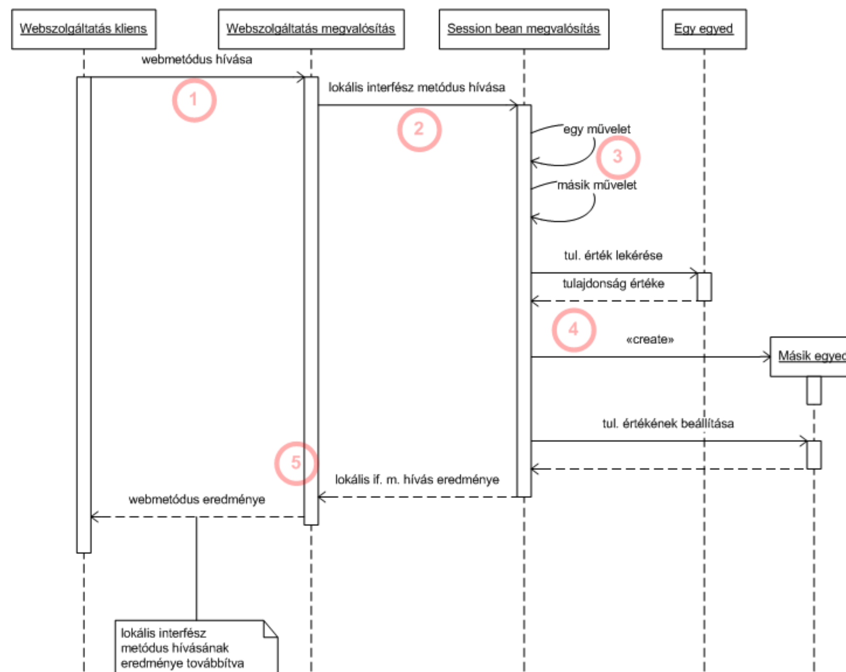
interfésszel rendelkező *stateless session beanre*. Minden webszolgáltatás metódus legfeljebb a kliens felé dobott kivételek megfelelő alakra alakításával foglalkozik közvetlenül, ezen kívül a metódus törzse a session bean megfelelő metódusát hívja, és szükség esetén az azáltal szolgáltatott eredményt adja vissza a kliensnek.

2. **Session bean helyi interfész megvalósítása.** Az EJB specifikációnak megfelelően a session beaneknek rendelkezniük kell legalább egy *interfésszel* (ami lehet *helyi* vagy *távoli*) és egy *megvalósítással*. Számunkra elegendő *helyi interfészek* használata, hiszen közvetlenül nem engedjük meghívni a beanek metódusait. A helyi interfészeket pedig a webszolgáltatás megvalósítások tudják használni.

3. **Session bean megvalósítás belső működése.** A konkrét session beanek részletes működése az adott bean által végzett feladattól függ, de általános, hogy ezek a beanek egyedek kezelését végzik közvetlenül (az EJB-konténer által biztosított egyedkezelő felhasználásával) vagy segédosztályok metódusain keresztül.

   A session bean megvalósítások néha hivatkoznak más session beanekre is, ezt az EJB technológia lehetővé teszi.

Az imént tárgyalt szerkezet a feladatait a következő ábrán szemléltetett módon végzi el.



6. ábra: az egyszerű webszolgáltatás homlokzat elrendezés működése

1. **Webmetódus hívása.** A kiszolgálási folyamat akkor kezdődik, amikor a kliens meghívja az általa használni kívánt webszolgáltatás megfelelő metódusát (a szükséges argumentumokkal együtt). A hívás módja a klienstől függ (leggyakrabban absztrakciós réteg segédosztályain keresztül), a szerver oldali viselkedés szempontjából ez nem lényeges mindaddig, amíg helyes argumentumokkal felparaméterezett metódushívások történnek.
2. **Hívás továbbítása a megfelelő session beannek.** A webszolgáltatást megvalósító osztály az előző (szerkezeti) ábra 1. pontjánál leírtak szerint a megfelelő session bean megfelelő metódusát hívja meg az átadott argumentumokat is továbbítva.
3. **Feladatspecifikus műveletek végrehajtása.** A szerkezeti ábra 3. pontjánál leírtak szerint a session bean a kért feladatnak megfelelő utasításokat hajtja végre.
4. **Egyedműveletek végrehajtása.** Ld. szerkezeti ábra 3. pontja

5. **A művelet eredményének továbbítása a kliens felé.** A hívott session bean metódus eredményét a webszolgáltatás megvalósítás a kliens felé továbbítja. Amennyiben a session bean metódus kivételt dobott, a webszolgáltatás metódus a kivételt a kliens felé továbbítható alakra hozhatja,

Automatikusan generált szerver oldali proxy osztályok

A webszolgáltatások használatát az EJB 3.0 technológia nagyban elősegíti. Annotációk segítségével meghatározhatók a webszolgáltatások tulajdonságai. Az annotált osztályok az alkalmazás alkalmazásszerverre történő telepítése során ***előfeldolgozásra*** *kerülnek*, amely folyamán az alkalmazásszerver automatikusan elkészít egy sor olyan osztályt és XML-leírást, ami korábbi verziójú EJB technológia alkalmazásakor sok mechanikus munkát követelt a programozóktól.

Az előfeldolgozás során készülnek el az ún. **s*zerver oldali webszolgáltatás proxy osztályok*** is. Ezek az osztályok transzparens módon elfedik a webszolgáltatás készítői elől az objektum-XML leképezéseket. A szerver alkalmazás fejlesztése során ezekkel az osztályokkal közvetlenül nem találkozunk, de ettől függetlenül léteznek és segítik a munkánkat.

PERZISZTENCIA

Függőségbeszúrás és CMP

Az EJB 3.0-val (szintén az előző fejezetben tárgyalt annotációs előfeldolgozás eredményeként) lehetővé vált az ún. függőségbeszúrás (dependency injection) mechanizmus. Ez lehetővé teszi, hogy egyes, az EJB-konténer által biztosított erőforrásokat körülményes JNDI utasítások sorozata helyett egyszerű tagváltozókként használhassuk.

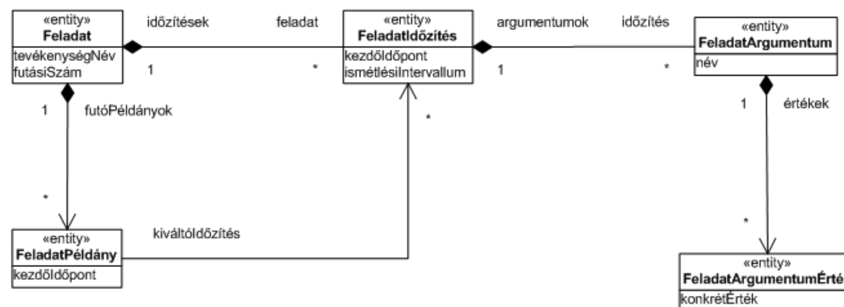
Ugyan nem EJB 3.0 újítás, de nagyon hasznos funkció az ún. *konténer által kezelt perzisztencia* (*CMP – Container Managed Persistence*) is. Ez a mechanizmus elintézi, hogy az egyes bean metódusok futása tranzakciókba legyen foglalva és hogy az egyedek adatbázissal történő szinkronizációja a lehető legegyszerűbb módon legyen kezelhető.

Az imént említett függőségbeszúráson keresztül kerülnek felhasználásra az EJB-konténer által biztosított egyedkezelő (`EntityManager`) objektumok is (a `@PersistenceContext` annotáció használható erre). Ezek a CMP mechanizmus felhasználásával együtt könnyen megvalósítható perzisztenciakezelést tesznek lehetővé.

Az egyedkezelő objektumok (és a CMP) ilyen módon való felhasználása gyakorlatilag minden session bean esetén előfordul, hiszen szinte mindegyik ilyen beannek hozzá kell férnie egyedekhez.

Ütemezett feladatok

A szerver oldalon ütemezett feladatok a következő ábrán szereplő szerkezetbe vannak rendezve.



7. ábra: szerver oldali ütemezett feladatokhoz kapcsolódó egyedosztályok szerkezetei

Feladat (Task)

A Feladat osztály egy adott tevékenység adott időzítéseit kezeli.

FeladatIdőzítés (TaskSchedule)

Egy feladat adott időpontra történő időzítését tárolja az időzítéshez tartozó argumentumokkal együtt.

FeladatArgumentum (TaskArgument)

Egy adott FeladatIdőzítéshez tartozó argumentumot tárol. Egy feladat esetén a feladat által végzett tevékenységben használt paramétereknek lehet értéket adni a FeladatArgumentum objektumok segítségével.

FeladatArgumentumÉrték (TaskArgumentValue)

Mivel a feladatok argumentumainak típusa tetszőleges lehet, ezért szükséges volt bevezetni a FeladatArgumentumÉrték egyedet is, amelynek egyedüli feladata, hogy egy adott FeladatIdőzítés egy adott argumentumának konkrét értékét tárolja.

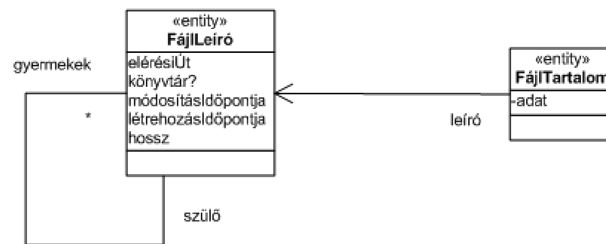
FeladatPéldány (TaskInstance)

Egy adott feladat futó példánya. Ezek az egyedek csak addig léteznek, amíg a megfelelő példány fut, arról tárolnak különböző adatokat. Amint a feladat (tevékenység) futása befejeződik, a hozzárendelt FeladatPéldány megszűnik.

Távoli fájlok

A szerver alkalmazás elkészítése közben kifejlesztésre került egy egyszerű felépítésű távoli fájlrendszer is. Ennek célja, hogy a GUI Applet hagyományos fájl- és fájlfolyamműveleteket felhasználva legyen képes adatainak tárolására, valamint hogy az ügynökök az általuk futtatandó osztályokat tartalmazó fájlokat szintén hagyományos fájlokként tudják elérni.

A távoli fájlrendszerhez tartozó egyedeket a következő ábra szemlélteti.



8. ábra: távoli fájlrendszerhez kapcsolódó egyedosztályok szerkezetei

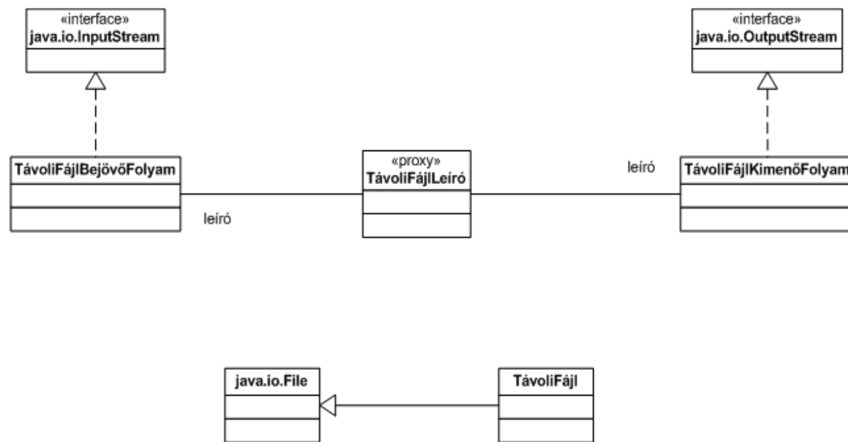
Az egyedek leírása a továbbiakban következik.

FájlLeíró

A FájlLeíró egyedek információs egyedek, amelyek *a fájlrendszer* ***struktúráját*** hivatottak tárolni. Ezek az egyedek az adatfájlok tartalmát nem tartalmazzák. Ez azért lett így kialakítva, mert a legtöbb fájlművelethez nem szükséges a fájlok tartalmának ismerete. Ha ezen egyedek tárolnák a fájlok tartalmát is, ez a legtöbb esetben felesleges hálózati forgalmat jelentene a szerver és a kliensek (GUI applet, ügynökök) között.

FájlTartalom

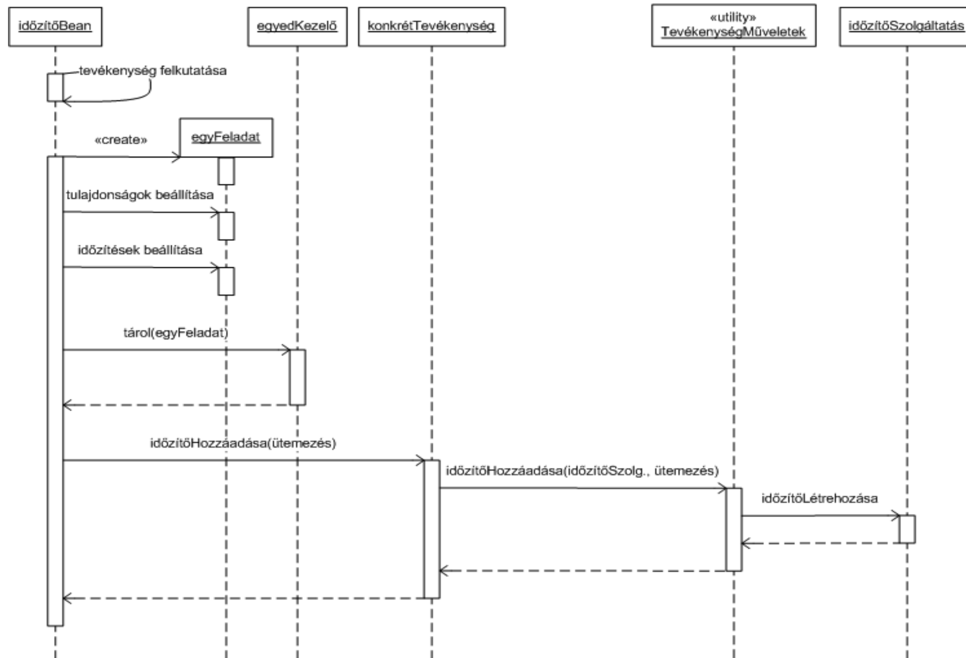
A FájlTartalom egyedek egyetlen célja, hogy tárolják a tartalmat azokhoz a távoli fájlokhoz, amelyekhez tartalom tartozik. Természetesen könyvtárak esetén ennek nincs értelme.



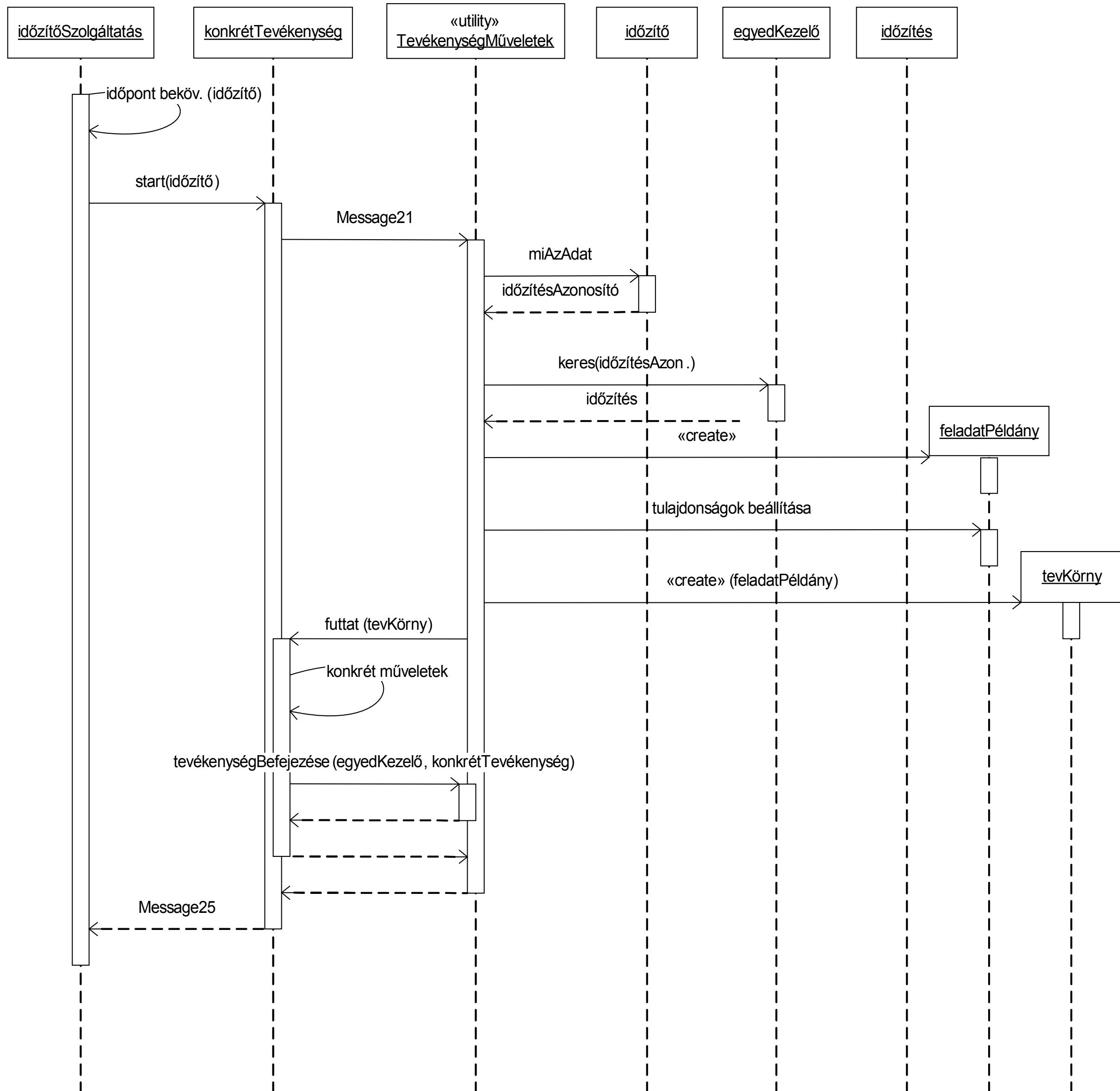
9. ábra: távoli fájlok használatát segítő kliens oldali osztályok

FELADATÜTEMEZÉS

A szerver oldalon háttérben futó feladatok egy bővíthető osztályhierarchiát képeznek. A tevékenységek időzítésének folyamatát az alábbi ábra szemlélteti.



A feladatok futását beépített EJB-időzítő idézi elő az alábbi diagramnak megfelelően.



TOPOLÓGIA-FELDERÍTÉS

A felderítési folyamat lépései

A topológia felderítése meghatározott csomópontokból kinyert információval történik. A monitorozni kívánt hálózati eszközök listáját a felhasználónak kell megadnia. Az

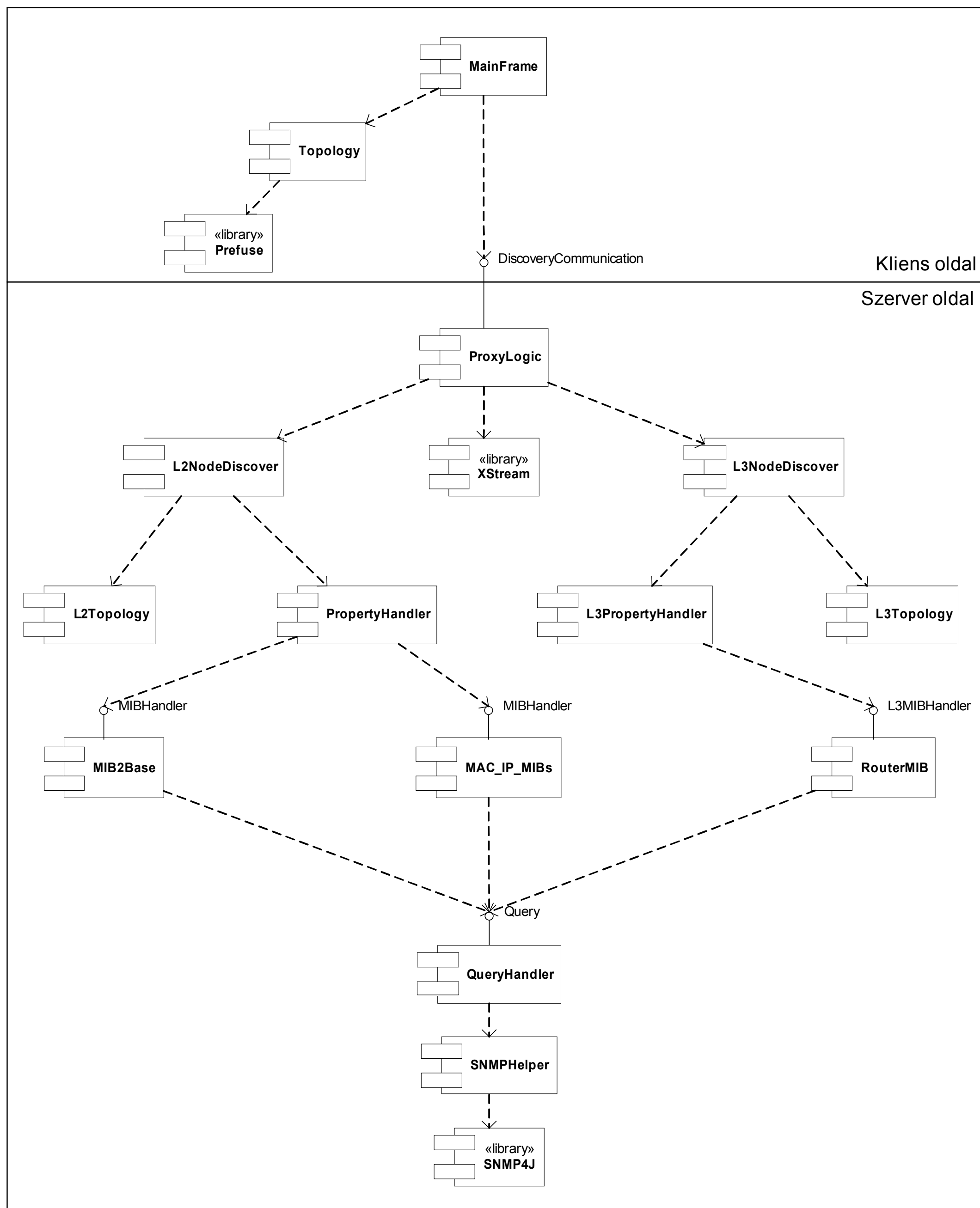
eszközök címzése *IP cím* (IPv4 vagy IPv6 cím egyaránt lehetséges) és *Community string* páros megadásával történik. Ez utóbbi az SNMP információk kinyerésére szolgáló hozzáférést biztosítja. A felderítés ezen megcímzett eszközökhöz való hozzáféréssel indul el. Ha a hozzáférés sikeres volt, azaz adott választ a menedzselt eszköz, akkor azt érvényes topológiabeli csomópontnak tekintjük és megjelenik a végső reprezentációban. Ha az eszköz nem válaszolt az SNMP kérésre, akkor úgy a továbbiakban figyelmen kívül hagyjuk, tehát a topológia csomópontjai a felhasználó által megadott lista részhalmazából képződik. A szükséges információk begyűjtése után a rendszer kiterjeszti a kapott információkat, kiszámolja a topológiát és azt a grafikus felhasználói felület részére bocsájtja. A felderítés logikája a szerver oldalán helyezkedik el, ez nem terheli a kliens CPU-t.

A felderítési folyamat befejező lépése a kliens megjelenítési parancsának kiadása. Mivel a szerver oldalról nem történik a kliens oldalra visszahívás, ezért a kliensre van bízva, hogy az éppen aktuális felderítés eredményének megjelenítését indítványozza. Tehát az aktuális csomópontlistát és az azokban tárolt topológia információkat a kliens elkéri a szervertől és azokat grafikus felületén megjeleníti, valamint a képernyőn lévő táblázatba berakja a csomópontokból kinyert információkat.

A felderítő alrendszer felépítése

A szoftver az alábbi szabadon felhasználható komponensekre támaszkodik:

- **SNMP4j:** az egyik legismertebb szabadon használható SNMP API Java környezeteben. Bővebb információ: http://www.snmp4j.org/



10. ábra: felderítési logika vázlatos felépítése

- **Prefuse:** egy szabadon használható gráf ábrázoló keretrendszer. Bővebb információ: http://prefuse.sourceforge.net/

- **XStream:** XML-objektum átalakításra fejlesztett szabadon használható API.

Bővebb információ: http://xstream.codehaus.org/

A szoftver moduláris felépítésű, tetszőlegesen bővíthető topológia/eszköz felderítő algoritmusokat megvalósító modulokkal. A *10. ábrán* látható a szoftver vázlatos felépítése. A szerveroldali felderítési üzleti logika összefogó komponense a **ProxyLogic** szerver oldali csonkja, amely felelős a teljes felderítési szerver taszk meghajtásáért. A komponens a ***DiscoveryCommunication*** interfészből van származtatva, így nyújt szabványos hozzáférést a kliens számára webszolgáltatásokon keresztül. Ezek a szolgáltatások a következők:

- **Hozzáférési lista beállítása** (IP cím – community string pár): minden egyes eszközhöz külön specifikálva van a hozzá tartozó SNMP hozzáférési azonosító (community string), ezt adja át a szerver számára a kliens.

- **Adott időbélyeggel rendelkező L2 csomópontok lekérése**: a kliens lekérheti az adott időpontban felderített második rétegbeli csomópontok listáját a topológia információkkal együtt.

- **Adott időbélyeggel rendelkező L3 csomópontok lekérése**: a kliens lekérheti az adott időpontban felderített második rétegbeli csomópontok listáját a topológia információkkal együtt.

Az interfésznek köszönhetően transzparens a kliens számára, hogy a ProxyLogic-ban implementált interfész metódusok a helyi gépen nyújtanak szolgáltatást, vagy a szerverrel webszolgáltatásokon keresztül kommunikálva elégítik a kliens kéréseit.

Az összefogó logika alatt helyezkedik el a konkrét felderítési feladat (második rétegbeli, illetve harmadik rétegbeli) logikája: **L2NodeDiscover** és **L3NodeDiscover**, melyek külön szálban futnak, így teljesen független lehet azok meghajtása. A modulok fő feladata a különböző információ kinyerő és feldolgozó modulok kezelése és a megfelelő topológia kiszámoló komponensek meghajtása. A feldolgozó modulok jelenleg

rögzítettek, de jól szeparált interfésszel rendelkeznek (***MIBHandler***). A jövőben az **L2NodeDiscover** és **L3NodeDiscover** modul képes lesz a rendelkezésre álló modulok dinamikus felfedezésére és betöltésére. Ezzel a szoftver könnyen bővíthetővé válik. A felderítő modulok nem közvetlenül vannak meghajtva, második rétegben a **PropertyHandler**, harmadikban az **L3PropertyHandler** feladata, hogy a felhasznált MIB kezelő modulokat koordinálja, ezzel különválasztva a felderítés módját az információfeldolgozás módjától.

A felderítő komponensek feladata ezután a rendelkezésre álló hálózati eszköz információk átadása a megfelelő topológia kiszámoló komponenseknek, melyek a következők:

- **L2Topology** – az OSI modell szerinti második réteg topológiáját tárja fel a következő elv felhasználásával. Mivel a hálózati eszközökön nem garantált, hogy minden eszköz támogatja a *Cisco Discovery Protocolt* (továbbiakban *CDP*), amivel teljes mértékben fel lehetne deríteni a kapcsolati réteg topológiáját, ezért az egyetlen biztosan járható út az eszközök továbbító táblázatában (továbbiakban *Forwarding Database, FDB*) található MAC címek alapján feltérképezni a csomópontok elhelyezkedését. (Még egy további megoldás lehetséges: eszközökből való információkinyerés nélkül ügynökökkel speciális forgalmat generálni és ez alapján felderíteni a topológiát – ezzel a lehetőséggel azonban a tervezés során nem foglalkoztunk.) A MAC cím alapján történő felderítés más-más elven alapszik, ha biztosan tudjuk, hogy az eszközök portjain az FDB-k teljesek, tehát valamennyi MAC előfordul valamennyi csomóponton, illetve ha ez nem garantált, akkor ugyancsak más megvalósítást kell alkalmazni. Mivel ez utóbbi eset gyakrabban fordul elő, a szélesebb körű gyakorlati alkalmazhatóság miatt a komponens ezt valósítja meg. Az algoritmus elméleti háttere a [LoOh01] cikkben található.

  Az alap ötlet a következő: *ahelyett, hogy két eszköz közvetlen kapcsolatát bizonyítanánk az ellenkezőjét bizonyítjuk: az A eszköz nincs a B eszköz X*

portjára kötve. Az alap feltevés, hogy a két eszköz össze van kötve (nem feltétlenül közvetlenül = egyszerű összeköttetés), ha egy ellentétes bejegyzés is található az eszközök továbbító táblájában, akkor ez nem igaz. Ilyen bejegyzés például, ha egy adott címet különböző irányból látnak. Olyan portpárok esetében, ahol nincs ilyen ütközés, ott pontosan lehet tudni, hogy ha képezzük A eszköz a portján az összes többi porton lévő FDB-k unióját és ugyanezt tesszük B eszköz b portjára is, akkor a két halmaz metszete üres. Ez az állítás fordítva is igaz, pontosan ez teszteli az egyszerű összeköttetést. Fontos elvi minimális követelmény, hogy két eszköz között csak egy portpárra legyen kielégítő az összeköttetés bizonyítása.

A tényleges algoritmus úgy működik, hogy kezdetben a hálózati kapcsolók közül kiválasztja a gyökérnek legalkalmasabb csomópontot és leképezi a portjaira az adott irányban lévő eszközöket. Majd a portokra képezett csomópontokra rekurzívan meghívja a leképező algoritmust. Ha bizonyítható az FDB-k alapján, hogy egy helyen létezik egy fel nem derített hálózati eszköz, akkor ott egy virtuális kapcsolót helyez el az algoritmus. Ezután következik az ügynökök és forgalomirányítók, mint második szintű topológia végpontok ráillesztése a fa leveleire, vagy ha valamely belső ponton fordul elő a MAC címe, akkor oda becsatlakoztat egy virtuális kapcsolót és arra köti a végpontot. Ennek oka egyrészt lehet osztott közeg használata, illetve nem menedzselhető eszköz jelenléte.

- **L3Topology** – a megvalósításban lényegesen szerényebb információkra hagyatkozhatunk, mint a második réteg felderítésénél. Itt a forgalomirányítókból kinyerhető IPv6-os tudás lényegesen kisebb, mint az IPv4-es esetben. Egyrészt nem nyerhető ki adott forgalomirányító közvetlen szomszédai (*next hop*) a forgalomirányító táblázatából, egyedül az interfészeken konfigurált IPv6-os címek alapján, illetve az ezekhez tartozó *prefix* bejegyzésekből deríthető ki az interfészekre kapcsolódó alhálózatok. A felderítő algoritmus az így kiterjesztett

alhálózatok közül a megegyezőket összeilleszti, létrehozva ezzel a next hop kapcsolatokat.

Amennyiben csak *link-local* cím van az adott csatolón, úgy nincs alhálózat információnk, más módon kell kiderítenünk, hogy mely routerekkel kommunikál ezen az oldalon. Erre szolgál az *ARP gyorstár.* Ennek hátránya, hogy csak IPv4-es forgalom alapján töltődik, előnye, hogy kideríthető, hogy mely forgalomirányítókkal kommunikál adott eszköz vagy IPv4-en vagy IPv6-on. Az ARP gyorstár alapú összekötés lényege, hogy ha egy eszköz portján lévő ARP tárban megtalálható egy másik eszköz egyik interfészének MAC címe, akkor ez a két csatolófelület egy szórási tartományba esik, közöttük nincs más forgalomirányító. Az nem garantált, hogy ők tudnak is kommunikálni IPv6 felett. Mégis, ha minkét kapcsolódó interfészen van konfigurálva legalább egy IPv6-os cím, akkor a forgalom létrejöhet. Az ilyen módon felderített alhálózatokat privátoknak tekintjük, hiszen csak a jelenlétüket tudjuk igazolni, semmilyen publikus információ nem áll rendelkezésre róluk.

A következő rétegben az információ kinyerő modulok találhatóak. Ezek definiálják a SNMP által lekérdezendő információkat és ezen információk feldolgozási módját. Az információ definiálás objektumazonosítók (*OID-k*) megadásával történik. A feldolgozás pedig ezen OID-kre adott válasz lekezelése. A **MIB2Base** modul a MIB2-ben definiált alapvető tulajdonságokat definiálja és dolgozza fel (pl.: *sysName, sorozatszám, modell, stb.*). Ezen túl az interfészeket és azok tulajdonságait is lekezeli. Az **MAC_IP_MIBs** modul az eszköz IP címeit (IPv4, IPv6) és a kapcsolók továbbító táblázatában (FDBs) lévő információkat kéri le és dolgozza fel. A **RouterMIB** modul pedig felel a forgalomirányítókból kinyerhető L3topológiához szükséges információk definiálásáért és a kinyert információk feldolgozásáért.

A fenti modulok a **QueryHandler** modul szolgáltatásait veszik igénybe a ***Query*** interfészen keresztül. A modul feladata, hogy az igényeket párhuzamosan továbbadja az **SNMPHelper** modulnak. Annyi példányban indítja el az alatta lévő modult amennyi IP

címünk van (a hozzáférési listában szerepelhetnek egyaránt IPv4-es és IPv6-os címek), vagy pedig amennyi szálunk lehet maximálisan. Ezzel a megoldással nagyságrendekkel gyorsul a rendszerünk, mivel a jelenlegi beállítások mellett akár 40 párhuzamos lekérdezést is kezelni tud. Az **SNMPHelper** modul egyszerű hozzáférést biztosít az SNMP4J API szolgáltatásaihoz. Két lekérdezés típust támogat: *GET* és *GETBULK*. GETBULK esetén képes detektálni a nem befejezett választ és a hiányzó részt letölteni. Ezt a funkciót csak szinkronizált formában valósítja meg, hiszen az SNMPv3 protokoll alapvető jellegénél fogva semmilyen információ nem áll rendelkezésre arról, hogy az adott OID-hez tartozó valamennyi érték megérkezett, vagy csak egy része. Ezt valamennyi válasz esetén ellenőrizni kell.

HÁLÓZATI INFORMÁCIÓ KINYERÉSE

A WinPcap és a LibPcap függvénykönyvtár

A *WinPcap* és a *LibPcap* függvénykönyvtár egy alkalmazás programozási interfész a hálózaton közlekedő csomagok elfogására. A Unix világban *LibPcap* könyvtár a neve, míg Windows rendszereken futó portja a *WinPcap* nevet viseli. Az újabb verziókban már lehetőségünk van ethernet keretek küldésére is, továbbá egy listát kérhetünk a hálózati interfészekről, amelyeket a LibPcap illetve a WinPcap használni tud, és szűrési feltételeket is adhatunk meg a csomagok olvasásánál.

A LibPcap és a WinPcap jó néhány nyílt forráskódú illetve kereskedelmi szoftver csomag elfogási és szűrési motorja, amelyek protokollelemző, hálózat monitorozó, hálózati forgalomgeneráló szolgáltatásokat nyújtanak. Ilyenek az *Ethereal* és *tcpdump* programok is. A NetSpotter alkalmazás szintén a LibPcap illetve WinPcap függvénykönyvtárakra épít.

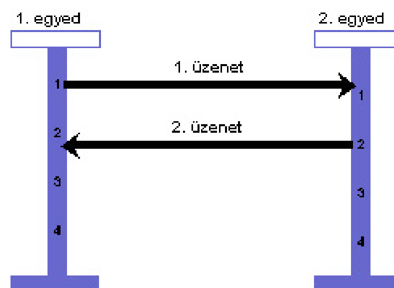
A csomagkapcsolt kommunikáció és az üzenetek szerkezetének ábrázolása

A TCP/IP protokollra épülő hálózatok úgynevezett csomagkapcsolt hálózatok,

amelyekben az adatok különálló, de önmagukban elemi egységeket képező blokkokban, csomagokban kerülnek továbbításra. Minden csomag, különböző számú bitet tartalmazó mezőből épül fel.

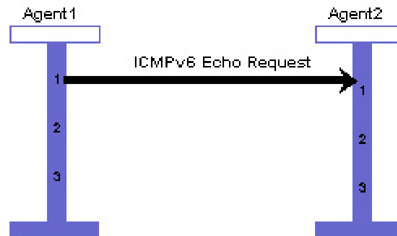
Tehát ahhoz, hogy egy csomagkapcsolt kommunikációt leírjunk meg kell mondani, hogy mikor és milyen felépítésű csomagot küldjön el az állomás, vagy várjon olvasásra. Egy MSC (ITU-T Z.120) szerű jelölésrendszert felhasználva a csomagkapcsolt kommunikáció könnyen leírható grafikai eszközökkel, amelyet aztán viszonylag egyszerűen transzformálhatunk szöveges formátumúvá és vissza.

Ez a jelölésrendszer a következő. A kommunikáló feleket egyednek nevezzük és grafikai ábrázolás esetén egy fej és farok résszel is rendelkező tengellyel jelöljük őket. Az üzeneteket vagy csomagokat az egyedek közé húzott nyilakkal ábrázoljuk. Ahogy haladunk az egyedek tengelye mentén a farok rész felé úgy haladunk előre az időben.



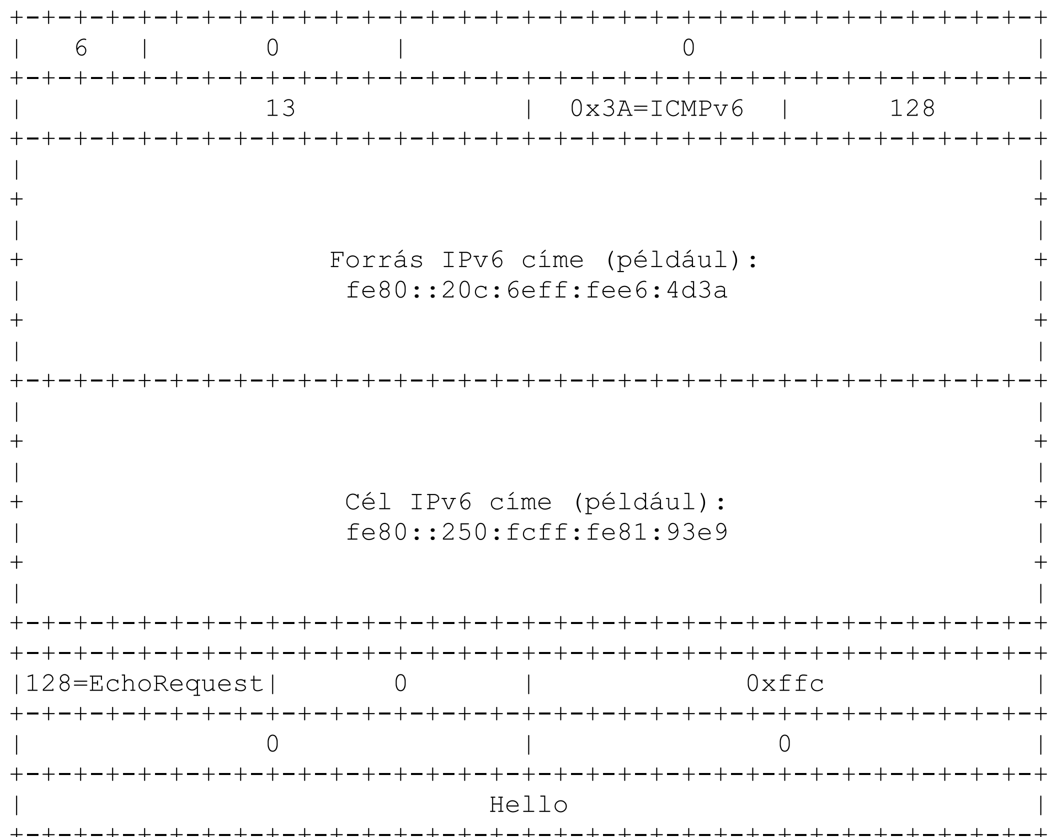
11. ábra: egyedek és üzenetek

Tehát például a *11. ábrát* az 1. egyed szempontjából nézve a folyamat a következő: először az 1. üzenetet elküldi, majd később várakozik a 2. üzenet beérkezésére. A 2. egyed ezzel ellentétben először az 1. üzenet megérkezésére vár egy ideig, majd a 2. üzenetet elküldi.

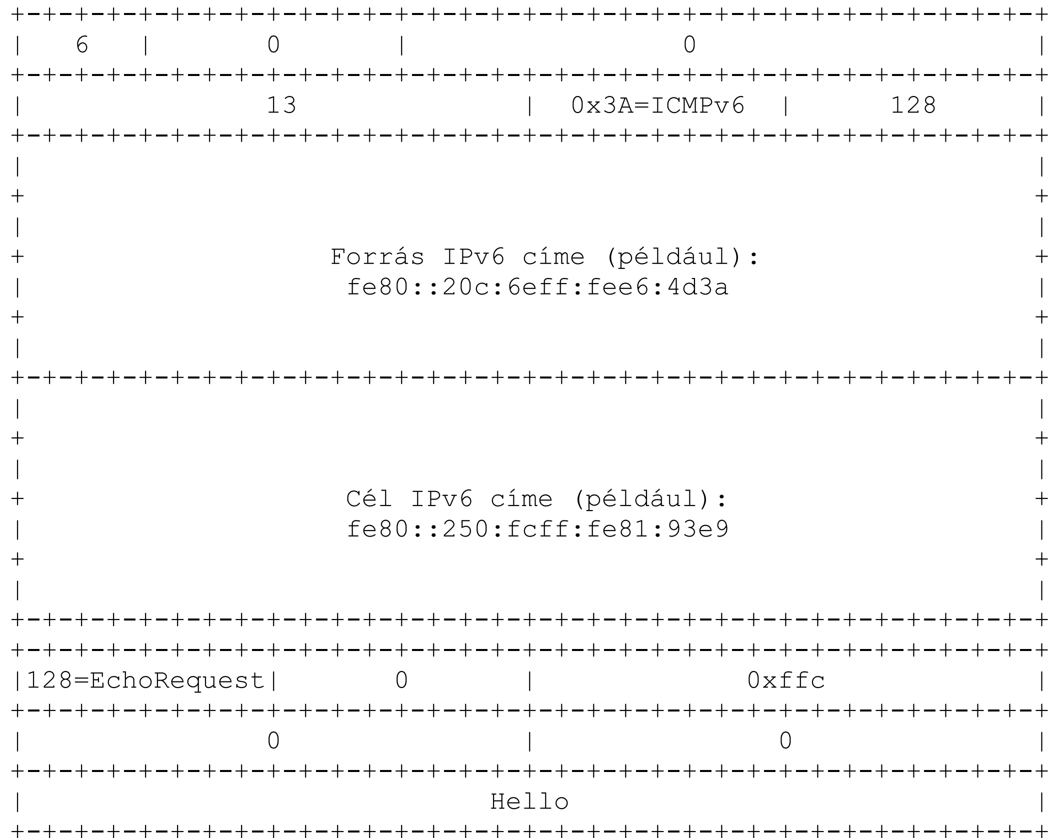


12. ábra: ICMPv6 Echo Request (Ping6)

A kommunikációs folyamat jelölésén túl le kell írni az üzenetek pontos tartalmát is. Egy konkrét példát tekintve, ha egy IPv6 csomagba ágyazott *ICMPv6 Echo Request* üzenetet akar az 1. egyed kiküldeni, akkor az üzenet szekvencián (*12. ábra*) túl meg kell adnunk az IPv6 csomag fejléc és az beágyazott ICMPv6 szegmens mezőit és azok tartalmát, ami például úgy nézhet ki, mint a *3.3. ábrán* látható.



```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   6   |       0       |                  0                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              13               |  0x3A=ICMPv6  |      128      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                   Forrás IPv6 címe (például):                 +
|                    fe80::20c:6eff:fee6:4d3a                   |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                    Cél IPv6 címe (például):                   +
|                    fe80::250:fcff:fe81:93e9                   |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|128=EchoRequest|       0       |             0xffc             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               0               |               0               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             Hello                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```



13. ábra: egy ICMPv6 Echo Request szegmens beágyazva egy IPv6 csomagba

A grafikus ábrázolás könnyen áttranszformálható a következő szöveges szerkezetbe, amelyet XML-dokumentumban tárolunk. A példát tovább folytatva a *3.4. ábrán* látható a folyamat szöveges ábrázolását tartalmazó XML-dokumentum faszerkezete.

Node	Value
?=? xml	version="1.0" encoding="UTF-8"
⊟ e tns:MSCDocument	(testcaseid, dataEntityBones?, messages, Agent*, areas?)
a xmlns:tns	http://nlab.inf.u-szeged.hu/camp6
a xmlns:xsi	http://www.w3.org/2001/XMLSchema-instance
a xsi:schemaLocation	http://nlab.inf.u-szeged.hu/camp6 sequence.xsd
e testcaseid	2
⊞ e dataEntityBones	(dataEntityBone+)
⊞ e messages	(message*)
⊟ e Agent	((entity*), (properties))
a id	1
a managed	true
a name	Agent1
a serial	1
⊟ e entity	(timer | loop | sendMessage | receiveMessage | areaRef | delay)
⊟ e sendMessage	
a adjacent	10
a body	30
a equals	true
a id	9
a name	ICMPv6 Echo Request
a serial	1
a timeout	3000
⊞ e properties	(interface*)
⊞ e Agent	((entity*), (properties))

14. ábra: az üzenet szekvencia szöveges ábrázolása

Node	Value
?=? xml	version="1.0" encoding="UTF-8"
e tns:MSCDocument	(testcaseid, dataEntityBones?, messages, Agent*, areas?)
a xmlns:tns	http://nlab.inf.u-szeged.hu/camp6
a xmlns:xsi	http://www.w3.org/2001/XMLSchema-instance
a xsi:schemaLocation	http://nlab.inf.u-szeged.hu/camp6 sequence.xsd
e testcaseid	2
e dataEntityBones	(dataEntityBone+)
e messages	(message*)
e Agent	((entity*), (properties))
a id	1
a managed	true
a name	Agent1
a serial	1
e entity	(timer | loop | sendMessage | receiveMessage | areaRef | delay)
e properties	(interface*)
e Agent	((entity*), (properties))
a id	2
a managed	true
a name	Agent2
a serial	2
e entity	(timer | loop | sendMessage | receiveMessage | areaRef | delay)
e receiveMessage	
a adjacent	9
a body	32
a equals	true
a id	10
a name	ICMPv6 Echo Request
a serial	1
a timeout	3000
e properties	(interface*)

15. ábra: az üzenet szekvencia szöveges ábrázolása (2)

Az *Agent* csomópontok tartalmazzák az egyedek adatait. Az egyed nevét, azonosítóját, sorszámát és hogy menedzselhető eszközről (vagyis Agent-ről) van-e szó. (*3.4. ábra*).

Az üzenet forrásának vagy céljának megfelelő *Agent* csomópont alatti *entity* csomópontok alatt a *sendMessage* illetve a *receiveMessage* (*3.5. ábra*) csomópontokban kap helyet az üzenetek szekvenciára vonatkozó adatai. Vagyis az üzenet párjára vonatkozó hivatkozás a fogadó vagy küldő Agent-nél, az üzenet tartalmának leírására vonatkozó hivatkozás. Jelzés, hogy az üzenet tartalma megegyezik-e a párjának tartalmával. Itt tárolódik az üzenet neve és olvasandó üzenet esetén az olvasásra maximálisan fordított idő, valamint egy azonosító és sorszám.

Node	Value
⊟ e messages	(message*)
⊟ e message	(data*)
a id	30
a name	IPv6
a serviceNeeded	IPv6
a uLink	None
⊟ e data	(value*, originValue?, encap?)
a reference	11
⊟ e value	
a reference	15
	13
⊟ e value	
a reference	16
	0x3a
⊟ e value	
a reference	18
	2001::1
⊟ e value	
a reference	19
	2001::2
⊟ e encap	(data)
a id	31
a name	Icmpv6Echo
a reference	20
a serviceNeeded	Icmpv6Echo
⊟ e data	(value*, originValue?, encap?)
a reference	22
⊟ e value	
a reference	25
	0x1be1
⊞ e message	(data*)
⊞ e Agent	((entity*), (properties))
⊞ e Agent	((entity*), (properties))

16. ábra: az üzenet szekvencia szöveges ábrázolása (3)

Az üzenet tartalmak a *messages* csomópont alatt kaptak helyet. A *3.4. ábrán* látható első üzenethez a 30-as azonosítóval rendelkező tartalom leírás tartozik. (*3.6. ábra*) A *message* csomópont attribútumaiban tárolódik az üzenet tartalom neve (jelen esetben IPv6) és még néhány a rendszer által kezelt információ. A *data* csomópont *reference* attribútuma hivatkozik az IPv6 üzenet vázára, amelyek a *dataEntityBones* csomópont alatt foglalnak helyet, majd a *value* csomópontok következnek sorban azon mezők után, amelyeknél a váz által definiált alapértelmezett értékeket a felhasználó felüldefiniálta. A példában a váz azonosítója 11, amely az IPv6 csomag vázát írja le. Minden *value* csomópont tartalmaz egy hivatkozást a váz adott mezőjére és magát a felüldefiniált értéket. A beágyazott üzenet az *encap* csomópont alatt helyezkedik el, ismételve az

előbbi szerkezetet. A példában egy Icmpv6Echo üzenet van beágyazva.

?=? xml	version="1.0" encoding="UTF-8"
⊟ e tns:MSCDocument	(testcaseid, dataEntityBones?, messages, Agent*, areas?)
a xmlns:tns	http://nlab.inf.u-szeged.hu/camp6
a xmlns:xsi	http://www.w3.org/2001/XMLSchema-instance
a xsi:schemaLocation	http://nlab.inf.u-szeged.hu/camp6 sequence.xsd
e testcaseid	2
⊟ e dataEntityBones	(dataEntityBone+)
⊟ e dataEntityBone	(chksum, lengthField, item*)
a extends	None
a id	11
a name	IPv6
a serviceNeeded	IPv6
⊞ e chksum	(chksField, chksFields, pseudoHeader)
⊞ e lengthField	(field, fields)
⊟ e item	(name, type, length, value?, dataTemplate?)
a chksum	false
a id	12
a length	false
a override	0
e name	Version
e type	bit
e length	4
e value	6
⊞ e item	(name, type, length, value?, dataTemplate?)
⊞ e item	(name, type, length, value?, dataTemplate?)
⊞ e item	(name, type, length, value?, dataTemplate?)
⊞ e item	(name, type, length, value?, dataTemplate?)
⊞ e item	(name, type, length, value?, dataTemplate?)
⊞ e item	(name, type, length, value?, dataTemplate?)
⊞ e item	(name, type, length, value?, dataTemplate?)
⊞ e item	(name, type, length, value?, dataTemplate?)
⊞ e dataEntityBone	(chksum, lengthField, item*)

17. ábra: az üzenet szekvencia szöveges ábrázolása (4)

Visszatérve az IPv6 csomag vázához, a *dataEntityBones* alatt a 11-es azonosítóval rendelkező *dataEntityBone* csomópont a következő módon írja le az IPv6 csomag szerkezetét. Tartalmazza természetesen a váz nevét és egyéb rendszer által kezelt információkat, többek között azonosítót. A *chksum* és a *lengthField* csomópontok alatt tárolódik a beállított ellenőrző összeg illetve hossz mező számításához szükséges adatok. Majd az *item* csomópont alatt sorban az egyes mezők szerkezetének leírása. Nevük, típusuk, hosszuk, és az alapértelmezett érték. Tehát az első *item* az IPv6 fejléc Version 4 bites mezőjének adatait tárolja, majd sorban a többi mező adatai.

Azáltal, hogy külön csomópont alatt kerül tárolásra az üzenetek szerkezetének leírása és a szekvencia leírása, ha többször akarunk ugyanolyan szerkezetű üzenetet felhasználni a szerkezet leírása csak egyszer kerül az XML-dokumentumba.

Összefoglalva egy csomagkapcsolt kommunikációs folyamat egy ilyen jelölés-rendszerrel felhasználóbarát módon ábrázolható grafikusan és hatékonyan leírható szövegesen is.