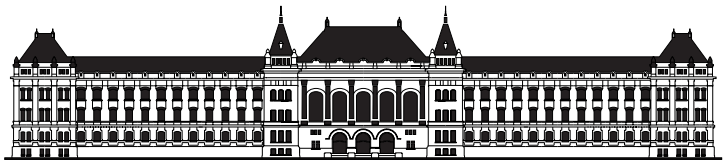


BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM
VILLAMOSMÉRNÖKI ÉS INFORMATIKAI KAR

Életviteli eszközök elosztott környezetben

Szerző
Vajna Miklós

Konzulens
dr. Hanák Péter

2009.

Nyilatkozat

Alulírott Vajna Miklós, a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, és a szakdolgozatban csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen a forrás megadásával megjelöltem.

Declaration

I, the undersigned, hereby declare that this thesis is the product of solely my own efforts except where otherwise indicated. Everything written in this document is the combination of my knowledge of the domain and the resources mentioned in the list of references.

Budapest, 2009. december 11.

Vajna Miklós

Kivonat

Az életvitelt segítő infokommunikációs eszközök (digitális vérnyomásmérő, mérleg, hőmérő, mobiltelefon, segélyhívó gomb, aktivitásfigyelő stb.) rendszerbe kapcsolása napjaink egyik aktuális műszaki kihívása. E rendszerek sajátossága, hogy a rendszerelemek és a közöttük fennálló kapcsolatok dinamikusan változhatnak, amit a rendszert vezérlő, felügyelő programnak támogatnia kell. Az Erlang nyelvet eredetileg telefonközpontok programozására fejlesztették ki, amelyekre jellemző az elosztottság, a párhuzamosság és a hibatűrés igénye. Ugyanezek a tulajdonságok jellemzik az életviteli rendszereket is. Ebben a szakdolgozatban megvizsgálom az Erlang nyelv és környezet alkalmasságát a fent nevezett területen.

Abstract

Forming a system from assistive infocommunication devices (digital blood pressure meter, balance, temperature meter, mobile phone, activity monitor etc.) is one of the current technology challenges today. The speciality of such systems is that the elements of the system and the connections between them can change dynamically, and the program controlling, monitoring the system should support this. The Erlang language was originally developed to program telephone exchange centres, which are typically distributed, concurrent and fault-tolerant. The same properties apply to assistive systems as well. In this work I investigate if the Erlang language and environment is suitable in the area mentioned above.

E lap helyére jön a kiírás eredeti példánya.

Tartalomjegyzék

1. Bevezető

Felmérések támasztják alá, hogy az idősödő emberek a lehető legtovább szeretnék megőrizni függetlenségüket, otthonukban maradni és a segítséget ott megkapni. Azaz egyre nagyobb felügyeleti, gondozási és ápolási kapacitásra lesz szükség.

Az életviteli rendszerek létrehozásakor az a feladatunk, hogy megtervezzük, hogy hogyan segíthetjük az idősebbeket és más rászorulókat abban, hogy otthon, a munkahelyen és szociális kapcsolataikban minél tovább független, hasznos tagjai maradjanak a társadalomnak.

Az egészségügyi és a szociális ellátás területén kiemelendő az időskorúak, továbbá a fogyatékkal élők, a betegségből felépülők és a rehabilitációra szorulók ellátásának, gondozásának, felügyeletének, ápolásának, önálló életvitelének segítése infokommunikációs eszközök alkalmazásával, rendszerbe állításával; a táv-diagnosztikai és távgyógyászati rendszerek elterjedésének elősegítése, valamint az otthonápolás támogatása[1].

Ahhoz, hogy infokommunikációs eszközeink valóban kiszolgáljanak - és ne kiszolgáltatottá tegyenek - bennünket, egymással együttműködni képes és egyszerűen kezelhető eszközökre van szükség, amihez elengedhetetlen egy megbízhatóan működő keretrendszer. Ha a keretrendszer stabilitási problémákkal küzd, vagy funkcionalitásában korlátozza az egyes eszközöket, akkor hiába fejlesztenek a rendszerhez kiváló modulokat.

Az életviteli rendszerbe kapcsolt infokommunikációs eszközök tehát elsősorban abban különböznek a hagyományos társaiktól, hogy az azokat használó ember felé minimális elvárásokat támasztanak. Egy vérnyomásmérő esetén a legjobb, ha maga az eszköz tudja, hogy mikor kell a mérést elvégezni, és erre figyelmezteti is az illetőt. Nem szerencsés, ha a mérés elkészülte után a mérési eredményt a betegnek kell kézzel rögzítenie, jobb, ha maga a műszer rendelkezik saját memóriával, amelynek tartalmát azonnal vagy később valamilyen adatbázisba küldi el. Ezen kívül fontos szempont, hogy a kezelés minél egyszerűbb legyen.

Például egy mérleg esetén jó lehet, ha súly ráhelyezése esetén automatikusan rögzíti a mérési eredményt, és nem kell erre kézzel utasítani.

Az összekapcsolást nehezítő technikai probléma viszont, hogy az eszközök tipikusan kevés információt árulnak el magukról, így az automatikus rendszerbe állítás abban az esetben, ha az eszköz még korábban nem volt a rendszer része, igencsak nehézkes. Ezen segíthetünk, ha megengedjük, hogy ne közvetlenül az eszközökkel kommunikáljon a rendszer, hanem az egyes eszközökhöz készített illesztőkön keresztül, és azt a célt tűzzük ki, hogy csak olyan eszközökkel működjön együtt, melyek fel vannak készítve a mi rendszerünkben való működésre.

Ezzel elérhetővé válhat az a használati eset, mikor a vásárló meglátja a boltban egy eszközön egy rendszer emblémáját, hazaviszi a terméket, és az azonnal használható, mindenféle beállítás nélkül (vagy minimális beállítással). Ez a példa kissé elrugaszkodottnak tűnik, azon-

ban ha figyelembe vesszük az idősebb emberek igényeit, az előbbi funkcióra igenis igény lenne, ha lenne ilyen rendszer.

Mielőtt azonban új rendszer tervezésébe kezdenénk, érdemes megvizsgálni, hogy jelenleg milyen megoldások, vagy megoldás-kezdemények érhetőek el a piacon. A problémát alapvetően két részre lehet bontani. Egyrészt bármilyen életviteli rendszer készítésekor érdemes felhasználni valamilyen kommunikációs keretrendszert, tehát az alternatívákat érdemes mérlegelni, figyelembe véve, hogy milyen igényeink vannak, és milyen funkcionalitást nyújtanak a jelenleg elérhető implementációk. Másrészt érdemes azt is tanulmányozni, hogy ezekre a keretrendszerekre alapozva milyen jelenleg is működő és elérhető életviteli rendszerek készültek el.

Arra sajnos már előre fel kell készülni, hogy jelenleg még nem érhető el nyílt szabvány életviteli rendszerekkel kapcsolatban, illetve esetlegesen jövőbeli ilyen rendszerhez készítendő műszerekhez, így a jelenleg rendelkezésre álló rendszerek sem használhatnak ilyet, következésképp kevés információt tudhatunk meg róluk. Ennek ellenére célunk felfedni ezen rendszerek előnyeit és hátrányait, hogy egy új rendszer tervezésekor a korábbiak példájából tanulva tudjunk mérlegelni.

A jelenleg működő életviteli rendszerekben használt keretrendszerek tanulmányozásával nem célunk tanulni azok hibáiból, hiszen a témakiírás előirja, hogy az új rendszer alapja az Erlang nyelv és környezet legyen. Ennek ellenére hasznos összehasonlítani az Erlang által nyújtott funkcionalitást más keretrendszerekkel, ezáltal is könnyebben rátapinthatunk az Erlang rendszer előnyeire és gyengeségeire.

A korábban már említett zártság miatt az elkészült rendszerekben is inkább a felhasználó oldaláról érzékelhető különbségeket, a rendszer tapasztalható előnyeit és hátrányait igyekszünk megmutatni, majd ezen elemzés figyelembevételével nekikezdeni a saját, Erlang alapú rendszer tervezésének.

Fontos leszögezni, hogy a jelen munka tárgya elsősorban az életviteli rendszer megalkotásához szükséges köztes réteg (middleware) tervezése és elkészítése Erlang környezetben. Terjedelmi okokból egy teljes életviteli rendszer megvalósítása nem lehet a célunk, de megvizsgálunk néhány jelenleg is elérhető megoldást, így reális képet kaphatunk arról, hogy a tervezett rendszer milyen is lenne teljesen megvalósított állapotában.

A számunkra szükséges funkcionalitást nyújtó keretrendszerre jó példa az OSGi[2], mely Java nyelven készült, és sok olyan problémára nyújt megoldást, amely a Java alatt futó JVM esetén különös nehézséget jelent, míg ezek jelentős hányadára az Erlang Beam virtuális gépe nyelvi szinten nyújt megoldást.

Példaként említjük itt a futási időben betölthető, újratölthető és törölhető modulokat, az egyes monitorozandó csomópontok automatikus felügyeletét, vagy a rendszerjavítási célból futási időben kiértékelt parancsok végrehajtását[3].

A jelen bevezetőnek azonban nem célja részletesen specifikálni ezeket a feladatokat, csak rá akar mutatni a szemmel látható szimmetriára az Erlang környezet által nyújtott szolgáltatások és az életviteli rendszerek által igényelt funkcionalitás között.

1.1. Életviteli rendszerek

Tunstall

A végfelhasználó számára tényleges segítséget nyújtó életviteli rendszert valósított meg az angol Tunstall[4] cég. Céljuk telekommunikációs megoldásokat felhasználni az egészséggondozás problémakörében, ezzel biztosítva idősebb emberek számára, hogy hosszabb távon önellátó módon élhessenek, és hatékonyan gondoskodhassanak saját egészségükről, jó közérzetükről.

Az általuk kiemelt főbb problémakörökre a következő fejezetben részletesen kitérünk.

Everon

Hasonló módon kulcsrakész életviteli rendszert hozott létre a finn Oy Exrei Ab[5] cég is, az Everon[6] márkajelzésű termékeivel.

Az Exrei cég megoldásai három területre bonthatók:

- Személyi biztonság. Egy modern vezeték nélküli riasztó és monitorozó rendszert hoztak létre, mely lehetőséget ad mozgásukban korlátozott és idősebb emberek számára, hogy továbbra is otthon élhessenek. A rendszernek része egy részletes öndiagnosztika, így az esetek többségében azt is érzékeli, ha maga a rendszer hibásodik meg.

  A rendszer egy alapállomásból (base station) épül fel, ehhez kapcsolódnak szenzorok, illetve vezeték nélküli riasztható eszközök. Értelemszerűen az alapállomások nem csak a hozzájuk kötődő eszközöket képesek riasztani, hanem megfelelő biztonsági beállítások mellett egy másik helyen lévő alapállomáshoz kötött eszközt is, például idős emberek gyermekeinek otthonában. A rendszert kifejezetten úgy tervezték, hogy leállások ne, vagy nagyon ritkán legyenek csak szükségesek. A értesítendő célállomások átirányíthatóak. Lehetővé válik a rendszeren belül megállapítani egy-egy ember tartózkodási helyét is. Ajtó-szenzorokkal az is beállítható, hogy az idős ember otthonát csak bizonyos időszakban hagyhassa el, például megzavarodott állapotban ne induljon neki az éjszakának egyedül.

  A rendszer előnye, hogy független a telefon-hálózattól, illetve annak kieséseitől, nem igényli személyi számítógép meglétét a monitorozandó lakhelyen, az egyes szenzorok ellenállóak (például vízzel szemben), az egyes eszközök kevés karbantartást igényelnek (például hosszú tartalmú elemek), valamint az egyes figyelmeztető jelzések bármely Everon alapállomásról bármely másik alapállomásra eljuttathatók.

- Biztonsági rendszerek kivitelezése. Az előző pontban kifejtett rendszerhez kapcsolódhatnak mozgás, ajtó- és egyéb érzékelők, melyek egyrészt segítenek az emberek védelmében (például tűz esetén kulcs nélkül is el lehessen hagyni a házat), másrészt viszont támadók ellen is védenek: ha elhagyták a házat, akkor a mozgásérzékelő riasztóként is funkcionál. Ezen kívül idősebb emberek számára segítséget nyújthat, hogy a hagyományos kulcs helyett egy engedélyező kulcsot hordhatnak magukkal, melynek jelenlétét a rendszer automatikusan érzékeli, a beléptetőkártyákhoz hasonló módon.

- Tulajdon és berendezés monitorozása. Az előzőekhez kapcsolódóan egyéb szenzorokat is fogalmaz a cég, melyek lehetővé teszik, hogy autók vagy hasonlóan értékes objektumok hollétét akár a cég online weblapján keresztül ellenőrizzük, vagy bizonyos paramétereik megváltozásakor, ill. megadott érték felvétele esetén szöveges értesítést kapjunk.

Vivago

Magyarországon a Sonaris Kft[7] működik együtt a finn IST International Security Technology Oy[8] céggel, mely a Vivago[9] aktív betegfelügyeleti rendszer megalkotója.

A Vivago rendszernek a magukat csak részben ellátni képes idősebb emberek, a krónikus betegségben szenvedők, fogyatékkal élők a célközönsége. A rendszer 24 órás felügyelettel segíti az embereket, és automatikusan is képes segítséget hívni, a sérült ember közreműködése nélkül.

A korábbiakhoz képest újdonság, hogy a Vivago egy-egy felügyelt beteg esetén először tanuló módban működik, tehát méri a személy különböző aktivitásait, majd a tanuló időszak után képes érzékelni az anomáliákat, és riasztást eszközölni. Ezáltal lehetővé válik, hogy ne kelljen kézzel állítani a riasztási paramétereket, melyek pontos megadása a rendszer bevezetése során nehézségeket okozhat.

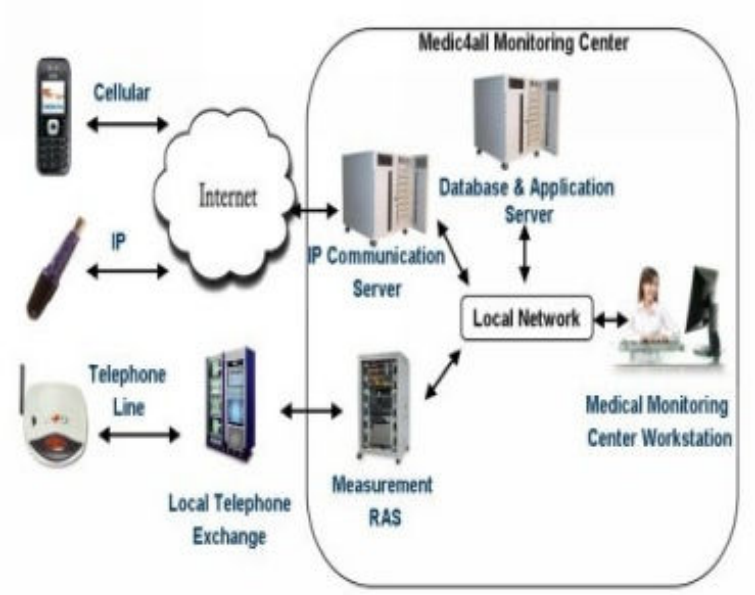
A rendszer konkrétan egy karkötő jellegű szenzorból, valamint egy, a lakásban elhelyezendő segélyállomásból áll, mely egyben kihangosított telefonként is használható szükség esetén.

A magyarországi forgalmazó ezen kívül ügyeletes ápolót is biztosít, így megoldódik az is, hogy az értesített családtag a felmerült probléma megoldásához nem rendelkezik elegendő kompetenciával. Természetesen egy rendszer tervezése szempontjából ez ugyanúgy csak egy értesíthető végpontként jelenik meg – azonban a rendszerek összehasonlítása szempontjából e részlet nem elhanyagolható különbség.

Telcomed

A Telcomed[10] egy ír kereskedelmi cég, pulzusmérőt, lázmérőt, vérnyomásmérőt, vércukormérőt és hasonló eszközöket forgalmaz. Egy teljes infrastruktúrát nyújtanak, a betegek otthonában elhelyezett egységtől a web-alapú Telcomed monitorozó központig. Zárt kommunikációs

protokollt használnak, és a megoldásuk nem arra lett tervezve, hogy harmadik fél által gyártott megoldások integrálhatóak lehessenek a rendszerbe.



1. ábra. A Telcomed rendszer áttekintése

1.2. Keretrendszerek

Egy életviteli rendszer számos csomópontból és végpontból áll. A végpontok információt fogadhatnak, ha riasztás érkezik, valamint döntéshozatal vagy szenzorok esetén információt küldhetnek is. A rendszer létrehozásakor a legtöbb probléma már itt jelentkezik: a végpontokat valamilyen módon a csatlakozási pontokhoz kell kötni, erre nem volna hátrányos valamilyen szabványos megoldást használni, valamint el szeretnénk kerülni a kerék feltalálását is.

Ebből következik, hogy a rendszert mindenképpen érdemes valamilyen jelenleg is elérhető keretrendszerre építeni, és nem teljesen önálló egységként létrehozni.

E bevezető alfejezet célja ismertetni ezek közül néhány jelenleg is használatban lévő megoldást, hogy ezek után legyen viszonyítási alapunk az Erlang környezet által nyújtott funkcionalitás megítéléséhez.

RPC

Az egyik legelső[11] ilyen megoldást a Sun cég specifikálta 1988-ban. A szabványt ma is aktívan használják, (e munka írásának pillanatában) legutolsó verzióját[12] 2009-ben adták ki. A megcélzott probléma a következő: adott egy C függvény, amelyet szeretnénk transzparens módon meghívni, de úgy, hogy egy másik gépen fusson le. Tehát a kliens oldalon kell egy csonk (stub), mely azonos interfésszel rendelkezik, mint az eredeti függvény, de az implementáció

magában foglalja a távoli kiszolgálóhoz való csatlakozást, a távoli kódfuttatást, valamint az eredmény lekérését. A kiszolgáló oldalán szintén szükség van egy csonk függvényre, mely lehetővé teszi, hogy az RPC kiszolgáló kérés esetén meghívja a tényleges implementációját a függvénynek. A távoli eljáráshíváshoz egy interfészt kell definiálni egy interfész-leíró nyelvvel (Interface Description Language, IDL), majd ebből a csonkok generálhatóak például az *rpcgen* programmal.

A módszernek manapság az egyik fontos felhasználási területe a hálózati filerendszerek (Network File System, NFS). Hátrányai közül leginkább azt érdemes megemlíteni, hogy heterogén környezetben, illetve nyitott hálózaton limitált a funkcionalitása.

DCOM

A DCOM[13] (Distributed Component Object Model) technológiát a Microsoft alkotta meg, a szintén általuk kidolgozott COM (Component Object Model) kiterjesztéseként. A DCOM nem titkolt szándéka versenytársként megjelenni a később említésre kerülő CORBA architektúrával szemben.

A DCOM mögötti gondolat az volt, hogy a népszerű COM technológiát továbbfejlesszék, és lehetővé tegyék az eljáráshívást távoli számítógépen is. A COM legfontosabb tulajdonsága, hogy lehetővé tette alkalmazások számára, hogy egy publikus interfészt definiáljanak, és az ebben megjelölt függvényeket a gépen futó más alkalmazások meghívják. Az RPC-hez képest újdonság, hogy az interfészt nem kell előre tudniuk a klienseknek, hanem egy `QueryInterface()` függvényen keresztül az lekérdezhető.

Annak ellenére, hogy alternatív implementációk elérhetőek más rendszerekre is, ez a technológia igazán csak Windows rendszereken terjedt el.

CORBA

A CORBA[14] szabványt az Object Management Group (OMG) dolgozta ki, célja pedig, hogy több nyelv, több gépen együtt tudjon működni, tehát lehetővé tegye közöttük a távoli eljáráshívást. Fontos újdonság még benne, hogy teljes mértékben hordozható.

Az interfész megadása itt is IDL segítségével történik, viszont az is a szabvány része, hogy az IDL-t hogyan kell leképezni számos nyelvre. Természetesen ezen kívül az előre meghatározott nyelv-listán kívül is még több nyelvhez elérhető a CORBA, de ott az implementáció már nem szabványosított.

A tényleges távoli eljáráshívás két legfontosabb komponense itt a objektumkérés-közvetítő (Object Request Broker, ORB – az alkalmazás más objektumokkal csak ezen keresztül léphet interakcióba), valamint az objektumadapter (Object Adapter), amelynek feladata kezelni az objektumok referencia-számlálóját, figyelni az élettartalmát, s.í.t.

XML-RPC

Végezetül megemlítjük az XML-RPC[15] szabványt, mely HTTP protokoll felett, XML használatával oldja meg a távoli eljáráshívást. Ez a megközelítés rendkívül népszerű napjainkban webszolgáltatások esetén, valamint a Microsoft által a DCOM leváltására megalkotott WCF (Windows Communications Framework) is XML-RPC-t használ belül.

OSGi

E Java alapú keretrendszerre a következő fejezetben részletesen kitérünk.

2. Egy-egy kiválasztott rendszer részletes ismertetése

2.1. A Tunstall életviteli rendszer

A számos, jelenleg elérhető életviteli rendszer közül az angol Tunstall cég életviteli rendszerét választottuk részletesebb ismertetés céljából. Ennek oka, hogy az általuk nyújtott szolgáltatásokról részletes információt tettek elérhetővé, valamint számokkal, valós életből vett példákkal és esettanulmányokkal szolgálnak az érdeklődőnek, így engedve betekintést bármely, (akár nem is kompetens) érdeklődőnek.

2. ábra. A Tunstall Telehealth Monitor egy Bluetooth alapú vérnyomásmérővel

A cég által létrehozott életviteli rendszer számos, a napi életben előforduló problémát igyekszik megoldani:

- Elmezavar. Gyakori probléma, hogy az ilyen problémákkal küzdő emberek elhagyják otthonukat, majd indokolatlanul sok ideig nem térnek vissza. A cég forgalmaz olyan szenzort, amely adott idő elteltével riaszt, ha nem történt meg a hazaérkezés. Egy hasonló probléma, hogy az ágyból kikelve nem találják az utat a fürdőszoba felé vagy vissza, ehhez is létrehoztak egy megoldást, mely érzékeli, hogy mikor kelt ki az ágyból az ember, és ekkor felkapcsolja a világítást az útvonal könnyebb megtalálásához. Végül az ilyen emberek sokszor képtelenek felmérni, hogyha veszélybe keveredtek, és nem teszik meg a szükséges óvintézkedéseket. A cég füst-, széndioxid- és földgáz-érzékelőket is forgalmaz, melyek egy központban riasztanak, ha segítségre van szükség.

- Esések. Naponta 8000 idősebb vagy gyenge ember esik el. [1] Ezek 70 százaléka éjszaka történik. Minden ötödik ember, aki eltöri a csípőjét, 6 hónapon belül hal meg. Ebből következik, hogy ha egy embernek félnie kell az eleséstől, az magával vonja azt az érzést, hogy már nem képes függetlenül mozogni, valamint hogy már nem él teljes életet. A cég így két problémakört céloz meg: egyrészt minimálisra csökkenteni az esések hatását, másrészt növelni az indokolatlanul kis mértékű önbizalmat. A cég olyan szenzort forgalmaz, mely érzékeli, ha az ember elhagyta az ágyát, és nem tért vissza meghatározott időn belül, ami vélhetőleg arra utal, hogy az illető elesett. Az esés-érzékelő egy derékszíjon

[1] Ez és a többi szám a Tunstall cég által a jelen írás pillanatában (2009 október) közölt adatokból való, és Angliára vonatkozik.

viselt szenzor, mely egyrészt automatikusan érzékelni képes az esést, valamint egyszerű megoldást ad az elesett ember számára is az esés jelzésére, így a gyors segítség nyújtása könnyebbé válik. A harmadik műszer pedig egy mozgásérzékelő, mely akkor riaszt, ha az ember hosszabb ideje nem mozgott még lakáson belül, mely nagy valószínűséggel elesést jelent. Ezek az eszközök segítenek megelőzni olyan eseteket, mikor például egy idős ember a fürdőszobából visszatérve elesik a küszöbön, majd az egész éjszakát ott kell töltse, mivel nem tud segítséget kérni.

- Átmeneti biztosítás. Ennek célja, hogy olyan emberek akik kis segítséggel már otthon gyógyulhatnának, sokszor a kórházban maradnak, és ott lassabban gyógyulnak, csupán néhány könnyen kiküszöbölhető probléma miatt. Ilyen például az előző pontban említett esés-érzékelő, e nélkül egy lábadozó beteget hazaengedni felelőtlenség lenne.

- Tanulási nehézségek. Hetente 200 ember születik tanulási nehézségekkel. Ez azt jelenti, hogy az átlagos embernél nehezebben tanulnak önállóságot, új vagy összetett információkat. A cég célja, hogy ezeknek az embereknek a tanulási nehézségein segítve ne kerüljenek távol a családjuktól, vagy valamilyen közösség részeként önállóbb életet tudjanak élni.

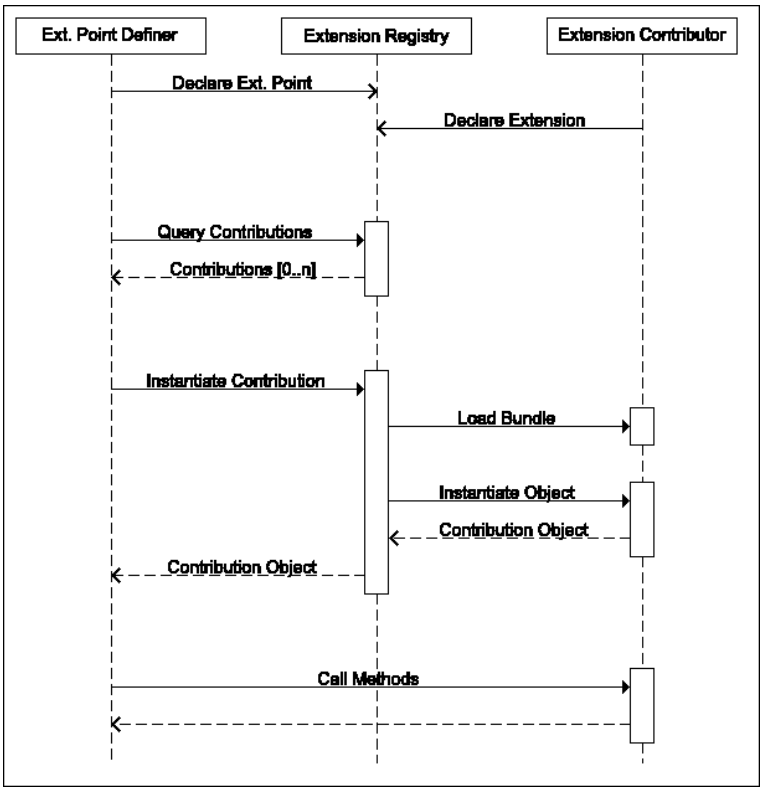
  Példaként lehetne említeni az előző pontokban is részletezett szenzort, mely az ágy elhagyásakor és adott időn belül való vissza nem téréskor jelez, így értesítve a családtagokat, ha probléma merült fel, míg az esetek nagy részében a rokonok nyugodtan alhatnak, hiszen a szenzor jelzésére biztosan felébrednek.

A technikai részletekről kevesebbet lehet tudni, de azért akadnak kivételek. A British Telecom[16] angol telefontársasággal együttműködve számos IP-alapú szolgáltatást tettek elérhetővé. A BT 21 CN[17] (21st Century Network) néven elindított programjuk technikai szempontból leginkább a hagyományos telefonhálózat (PSTN) lecserélése VoIP megoldásokra. Ez lehetővé teszi, hogy az egyes végpontokon ne csak beszédet, hanem sokféle adatforgalmat is lebonyolítsanak az ügyfelek. Ez a megoldás kiválóan használható a szenzorok által gyűjtött adatok továbbítására, valamint szükség esetén riasztásra. A Tunstall egy listát vezet azokról a szenzorokról, melyek a 21CN hálózat felett használt, általuk bevezetett protokollt támogatni tudják. Sajnálatos módon azonban nem érhető el részletes információ az általuk kifejlesztett, és éles üzemben is használt protokoll részleteiről.

2.2. Az OSGi keretrendszer

Az OSGi (régebbi nevén Open Services Gateway initiative) egy nyílt szabványügyi szervezet. A szövetség tagjai egy Java-alapú szolgáltatás-platformot definiáltak, mely távolról kezelhető.

A szabvány mögött olyan neves cégek állnak, mint a Nokia, Motorola és az IBM. Leginkább mobil eszközökben használják, de például a népszerű Eclipse IDE is ezt használja a beépülő moduljainak kezelésére. A szabvány szót itt a hagyományos értelemben használhatjuk, mivel csak a nyílt forrású implementációkat számolva is többesszámról beszélhetünk.



3. ábra. UML szekvenciadiagram, amely a kiterjesztések kezelését mutatja be az OSGi-t használó Eclipse-ben.

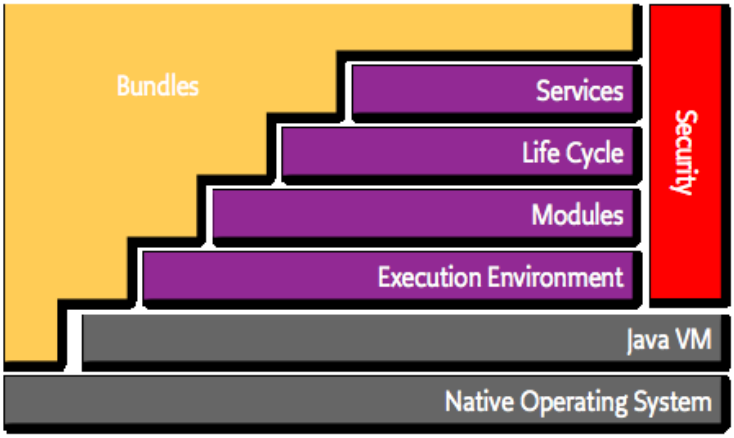
Az OSGi először 2000-ben jelent meg, a jelenleg aktuális 4-es verzióját pedig 2009 szeptemberében adták ki, így méltán tekinthetünk rá, mint aktívan fejlesztett és karbantartott megoldásra.

A keretrendszer legfontosabb elemei:

- Alkalmazások életciklusának kezelési modellje: az alkalmazásokat batyuk (bundle) formájában lehet terjeszteni, majd a JVM leállítása nélkül telepíteni, elindítani, frissíteni vagy eltávolítani.
- Szolgáltatás-adatbázis: ez az adatbázis lehetővé teszi az alkalmazások számára, hogy értesüljenek, ha új szolgáltatások érhetőek el, vagy egy szolgáltatás megszűnt, és így alkalmazkodhatnak az új környezethez.
- Végrehajtási környezet: A végrehajtási környezet azt definiálja, hogy egy környezetben milyen osztályok és metódusok érhetőek el. A Java Platform, Standard Edition is egy

elfogadott OSGi végrehajtási környezet, de a CDC keretrendszer (amit például a legtöbb mai mobiltelefon is használ) is támogatott, valamint van saját, általuk definiált környezet is. Minden egyes OSGi implementáció maga döntheti el, hogy milyen végrehajtási környezetet támogat, az előbbieket a legtöbb támogatja.

- Modulok: Azt is szabványosították, hogy az egyes batyuk milyen módon tudják az egymás közt fennálló függőségeket, valamint a kívülről elérhető API-kat definiálni, lehetővé téve ezáltal, hogy egységes módon lehessen az egyes egységekből és egységekbe metódus-hívásokat végezni.



4. ábra. Az OSGi rétegei

Az egyes batyuk olyan Java könyvtárak (JAR file-ok), amelyek OSGi-specifikus fejléccel rendelkeznek a JAR MANIFEST.MF file-jában.

Az egyes alkalmazásoknak a következő állapotai lehetnek:

- Telepítve: Ekkor még elképzelhető, hogy egy batyu függőségei nem állnak rendelkezésre.
- Feloldva: Minden függőség elérhető, az alkalmazást vagy leállították, vagy kész az indulásra.
- Indul: Az alkalmazás aszinkron módon el lett indítva, de még nem tért vissza.
- Aktív: Az alkalmazás elindult, és még nem kezdeményezték a leállítását.
- Leáll: Az alkalmazás leállítása aszinkron módon elkezdődött, de még nem fejeződött be.
- Eltávolítva: Az alkalmazás újabb telepítésig nem indítható el.

A szolgáltatások Java interfészeket jelentenek. Ezeket az API-kat implementálhatják a batyuk, majd ha a batyut regisztrálták a szolgáltatás-adatbázisban, akkor a kliensek itt megtalálhatják őket, és innentől értesítést kérhetnek a szolgáltatás elérhetősége és el nem érhetősége esetén.

Az OSGi szabvány is tartalmaz előre definiált szolgáltatásokat, melyeket három csoportra oszthatunk:

- OSGi System Services: ide tartoznak az olyan rendszerszolgáltatások, mint az eseménykezelő, a beállításkezelő, alkalmazások adminisztrációja, I/O műveletek, felhasználókezelés.
- OSGi Protocol Services: Előre definiált protokollok használatát segíti – HTTP feletti kommunikációt, UPnP-t támogató és mobil eszközök felé való kommunikációt.
- OSGi Miscellaneous Services: Ide tartozik minden egyéb előre definiált szolgáltatás, például XML értelmező.

3. Az Erlang-alapú életviteli rendszer terve

Az általunk tervezett életviteli rendszer legfontosabb ismérve tehát az lesz, hogy Erlang környezetben valósítjuk meg. Ezen fejezet célja specifikálni a követelményrendszert, és e specifikáció néhány jellegzetes elemét fogjuk megvalósítani a következő fejezetben.

3.1. Fő követelmények

3.1.1. Elosztottság

Elosztott rendszert tervezünk, ez a jelen esetben azt jelenti, hogy nincs központi eleme a rendszernek, és ez által nincsen egyetlen olyan elem sem, amelynek a meghibásodása esetén az egész rendszer leállna. Egy egyszerű példa: az idős ember elhagyja az ágyát, majd a lakását is, és megadott időn belül nem tér vissza. Két szenzor is jelez a türelmi idő letelte után. A rendszer lehetőséget ad arra, hogy egy szenzor több központnak is jelezzen, valamint azt, is, hogy egy központban ugyanarról a jelzésről többen is értesítést kapjanak. Természetesen a redundanciát tovább is fokozhatjuk, de ha ezt nem is tesszük, akkor is a következőkre számíthatunk:

- Legrosszabb esetben is kieshet egy elem, és a rendszerünk tökéletesen fog tovább üzemelni.
- Két elem kiesésekor már fennáll a veszélye annak, hogy gond lesz, például ha a két szenzor esik ki.
- Legjobb esetben három elem is kieshet: ha pont az egyik szenzor, az egyik központ és az egyik terminál esik ki.

A redundancián, mint az életviteli rendszerek szempontjából a leginkább szembetűnő előnyön kívül azonban más pozitívumai is lehetnek egy elosztott rendszernek. Említésre méltó, hogy a terminálok akár leválasztott (hálózati kapcsolat nélküli) állapotban is hozzáférnek a korábban megkapott üzenetekhez, valamint ha ezek között keresni akarnak, az szintén gyorsabb lesz, mint egy központosított rendszer esetén, hiszen nem kell hálózati késleltetéssel számolni.

3.1.2. A rendszer elemei

Csomópontok

Elosztott rendszer lévén a rendszer *csomópontokból* épül fel. A csomópontoknak két fontosabb fajtáját különböztetjük meg: a *központokat* és a *végpontokat*.

Központok

Egy központ bekapcsolás után nem tesz semmit, csak vár arra, hogy végpontok keressék. Ha egy végpont központot keres, akkor válaszol. Ha egy végpont regisztrál, akkor nevet ad neki, és erre a névre más végpontok feliratkozhatnak. Ilyen módon a végpont úgy küldhet üzenetet, hogy nem tudja, pontosan ki fogja megkapni. Ténylegesen csak a központnak küldi el, majd a központ küldi tovább a feliratkozott végpontoknak.

A központ nem rejti el az eredeti feladót, az üzenetet kapott végpontnak lehetősége van válaszolni az eredeti feladónak, ha például döntés szükséges. Ebben az esetben az üzenet közvetlenül kerül átvitelre a két végpont között, központok igénybevétele nélkül.

A központ ezenkívül hajlandó kiadni egy név mögött álló szenzor címét is, ez akkor hasznos, hogyha a végpont tudja a számára érdekes szenzor nevét, de az aktuális címét nem, valamint a szenzor sose ad ki magáról adatot implicit módon, így más módon a végpont nem szerezhetne tudomást a szenzor címéről.

Végpontok

A végpontok bekapcsolás után szórt üzenetet küldenek, majd várnak, amíg legalább egy központ válaszol a kérésre. A végpontoknak két fajtáját különböztetjük meg. Ezek egymástól sokkal inkább logikailag, mintsem technikailag különböznek.

Szenzorok

A *szenzorok* olyan érzékelők, melyek a környezetről szolgáltatnak információt.

Ez az információ-adás lehet implicit vagy explicit. Implicit esetben azt értjük, ha például egy hőmérő a mért hőmérsékletet óránként elküldi az általa ismert központoknak. Explicit esetben viszont egy másik végpont kérésére, közvetlenül a másik végpontnak bármikor elküldheti a kért adatot. Láthatjuk, hogy az implicit esetet nem előzi meg kérés-üzenet, míg explicit esetben mindig csak egy végpont kap értesítést.

Egy érdekes probléma annak a jelenségnek a kezelése, mely olyan szenzorok integrálásakor merül fel, melyek folyamatosan mérnek. Ezeket nem lehet explicit módon lekérdezni, viszont az összes mért adat implicit közlése indokolatlanul nagy hálózati forgalmat és adatmennyiséget generálna. Ezt a problémát úgy küszöbölhetjük ki – ezáltal ezt a szenzor-típust beillesztve a fenti két szenzor-kategóriába –, hogy a műszer elé teszünk egy modult, mely mindig tárolja az utoljára mért értéket. Így a legutóbbi mért adat explicit módon bárkinek bármilyen időpontban elérhetővé válik.

Egy másik tulajdonság, melyben a szenzorok különbözhetnek, a vezérelhetőség.

Például egy hőmérő esetében a legtöbbször nincs mit vezérelni, ellenben egy ház ajtajába szerelt zár esetén megoldható, hogy ne csak lekérdezni tudjuk, hanem a zár állapotát vezérelni is lehessen.

Terminálok

A *terminálok* olyan végpontok, melyek elsősorban üzenetek fogadására hivatottak, tehát bekapcsolás után keresnek legalább egy központot, valamint vagy előre beállított, vagy a felhasználó által interaktívan beállítható módon feliratkoznak a központ(ok) névszolgáltatása által adott eseményekre. A terminál lehet passzív, mint például egy képernyő, vagy aktív, például egy mobiltelefon. Az aktív terminálok reagálhatnak egy-egy üzenetre, míg a passzívok csak tájékoztató üzeneteket képesek megjeleníteni.

Egy lehetséges szcenárió például a következő: a tűzjelző jelez a központnak, a központ továbbítja az üzenetet egy telefonra, ott a célszemély egy üzenetet küld a tűzjelzőt tartalmazó lakás ajtajában lévő zárnak, hogy az adja meg az állapotát, az válaszol közvetlenül a telefonra, hogy zárva van, majd a célszemély úgy dönt, hogy ez az állapot nem kívánatos, és egy olyan vezérlő-üzenetet küld a zárnak, hogy az nyíljon ki. Ez a folyamat akár meg is mentheti egy idős ember életét, aki nem képes a nagy füstben kinyitni a zárat, ellenben az egyszerűen kilinccsel nyitható ajtón keresztül már képes elhagyni a lakást.

3.1.3. Dinamizmus

Volt már szó arról, hogy minden csomópont futási időben képes változtatni azon csomópontok listáját, amelyekkel kommunikálni képes. Központok esetén ez azt jelenti, hogy egy új szenzor regisztrációjakor nem kell a rendszert újraindítani, a végpontok pedig bármikor lekérhetik egy üzenetből az eredeti feladót, vagy egy központ névszolgáltatásán keresztül egy, a név mögött álló szenzor címét, majd annak üzenetet küldhetnek. Ezek az igények nélkülözhetetlenek a rendszer működéséhez.

Amire viszont első körben nem biztos, hogy gondolnánk, az az, hogy a központoknak különböző típusú szenzorokat kell kezelniük, és ez korántsem egyszerű feladat. A probléma az, hogy minden szenzor más módon hajlandó adatokat szolgáltatni. Még ha feltételezzük is, hogy minden szenzor egyben Erlang csomópont is, akkor is más üzenetet kell küldeni egy hőmérőnek (például `getTermperature()`), és mást egy vérnyomásmérőnek (például `getBloodPressure()`). Ezek egységes kezeléséhez a központban meghajtóprogramok szükségesek.

A megoldás az, hogy minden szenzortípus egyedi azonosítóval rendelkezik, ezt elküldi a központnak, a központ az azonosító alapján letölt egy meghajtóprogramot, betölti, és onnantól tudja, hogy hogyan kell kezelni.

3.1.4. Biztonság

Az előző alfejezet címéről, a dinamizmus szóról még egy jelenség juthat eszünkbe: amíg működik, addig jó, de ha valami gond van a rendszerrel, akkor bajban vagyunk, hiszen egy dinamikusan működő rendszerben hibát keresni nem kellemes feladat. Ha tovább keressük a problémákat a dinamizmussal, akkor felmerül az a kérdés is, hogy milyen biztonsági kockázatokat hozunk be ezzel a rendszerbe.

A két probléma látszólag összefügg, de valójában független. Az intelligens, plug-and-play rendszerekre valós igény van, különösen idős emberek esetében, hiszen ők nem szakemberek, így nem várható el, hogy hosszabb tanulás előzi meg a rendszer használatát. Tekintve, hogy elsősorban őértük jön létre a rendszer, ezt nem hagyhatjuk figyelmen kívül. A másik cél – a rendszer üzemeltetőinek szemszögéből – természetesen az, hogy minél inkább kontroll alatt legyen a rendszer, manuálisan beállítva a paramétereket, hogy a nem várt viselkedést elkerüljük. Sajnos a két célt nem lehet kifogástalanul teljesíteni egyszerre, de találhatunk olyan kompromisszumos megoldást, mely mindkét fél tűréshatárán belül helyezkedik el.

A biztonság kérdése annyiban kapcsolódik az előző problémához, hogy egy biztonságos rendszer egyik alapfeltétele, hogy minden, a rendszer számára érdekes objektum azonosítva legyen, ami jelen esetben azt jelentené, hogy a felhasználók és az eszközök is valamilyen autentikációs mechanizmus teljesítése után válhassanak csak a rendszer részévé. Ez problémát jelenthet például egy idős embernél, aki telefonálni se tud, mikor elesett, nemhogy jelszavakat megadni, mielőtt értesítené a központot. Ezzel ellentétes igény, hogy ne helyezhessen el bárki egy terminált az ablakunk alatt, mely azonnal értesíti a támadót, ahogy elhagytuk a lakást.

Jelen munkában ezt a felmerülő két problémát úgy oldjuk meg, hogy feltételezzük a következőket:

- A központoknak van interaktív felhasználói felülete.
- A központokhoz fizikailag csak olyan személy fér hozzá, akinek van is jogosultsága ehhez.
- A rendszer minden végpontja egyedi azonosítóval rendelkezik, melyet nem tud megváltoztatni.
- A kommunikációra használt csatorna biztonságos. (Vagy titkosított, vagy zárt a hálózat.)
- Ha a központokhoz új szenzor vagy terminál próbál csatlakozni, akkor azt első alkalommal a központban jóvá kell hagyni. Legalább az első végpont jóváhagyását fizikailag a központban kell elvégezni. (Innentől az engedélyezett eszköz távolról is engedélyezhet más eszközöket.)

Ezekkel a feltételekkel idős emberek is könnyen integrálhatnak új, gyárilag a rendszerrel kompatibilisnek tervezett eszközöket a rendszerbe, anélkül, hogy potenciális biztonsági réseket hagynánk abban.

Összehasonlításképpen megemlítjük, hogy hasonló jellegű probléma merül fel Bluetooth rendszerek esetén, egy „önlejátszó" CD számítógépbe tétele esetén, és még sok más példát lehetne hozni. A Bluetooth rendszer esetén a megoldás az lett, hogy ha két eszköz kommunikálni akar, akkor azokat egyszer párosítani kell, és ehhez a mechanizmushoz egy korábban egyeztetett jelszót kell megadni. Ha mindkét oldalon ugyanazt a jelszót adják meg, akkor a párosítás sikerült. Az CD-k esetében Windows operációs rendszer esetén úgy döntöttek, hogy alapértelmezésben figyelmeztetés nélkül elindul a program, amint behelyeztük a CD-t a meghajtóba. Ezt természetesen tiltani lehet, és a biztonsági kérdésekre kicsit komolyabban odafigyelő felhasználók ezt meg is teszik. Ellenpéldaként lehetne felhozni a legtöbb UNIX operációs rendszert, ahol a CD automatikus csatolása vagy fel se merül problémaként, vagy a felhasználóbarátabb rendszerekben is alapértelmezésként csak jelzést kap a felhasználó a CD behelyezéséről, de automatikus csatolásra felhasználói interakció nélkül soha nem kerül sor.

3.1.5. Határok

A fő követelmények áttekintésének végén megjegyezzük azt, amire már a bevezetőben is utaltunk: jelen munka célja egy életviteli rendszer köztes rétegének kidolgozása. Szándékosan nem foglalkozunk tehát a következőkkel:

- Skálázhatósági kérdések. A rendszerben típusonként kis számú elem található meg, kisebb finomítások szükségesek lehetnek, ha a rendszert típusonként nagyságrendekkel több elemmel használjuk, ezekre nem térünk ki.
- Felhasználói felület. Az egyszerűség kedvéért az összes csomópont a standard kimenetre (stdout) írja az üzeneteit. Egy tényleges rendszerben ezt célszerű valamilyen felhasználóbarátabb grafikus vagy webes felületre cserélni.
- Távoli karbantartás. A meghajtók automatikus letöltésén kívül egyéb automatikus kódletöltéssel nem foglalkozunk, de megjegyezzük, hogy a meghajtó-letöltéshez hasonló módon az egyes csomópontok teljes szoftverét frissíthetővé lehetne tenni. Az Erlang rendszer használata esetén – ahol futás közben lehet modulokat betölteni vagy frissíteni – ehhez nincs is szükség komolyabb erőfeszítésekre.

3.2. Erlang és UML

A rendszer tervezésekor formális jelölésrendszerként az UML (Unified Modeling Language) jelöléseit használjuk. Az UML elsősorban objektumorientált rendszerek tervezésére készült,

míg Joe Armstrong[18] szerint az Erlang nem objektumorientált nyelv, így a jelölésrendszer nem használható magyarázat nélkül.

Anélkül, hogy általános megfeleltetést állítanánk fel az objektumorientált nyelvek fogalmai és az Erlang rendszerben elérhető elemek között, a jelen életviteli rendszer tervezése során a következőket feltételezzük:

- Az objektumok az Erlang rendszerben Erlang processzek lesznek.
- Ha egy Erlang processz üzenetet kap, és az üzenet típusa szerint az üzenetre más-más módon reagál, azt megfeleltethetjük az objektumok metódusainak.
- Szekvenciadiagramok esetén objektumok létrehozásán `spawn()` hívásokat értünk, metódushíváson pedig adott típusú üzenet küldését.
- Állapotdiagramok esetén az egyes Erlang processzek életciklusát értjük, hiszen az Erlangban nincsenek frissíthető változók, hacsak nem számolunk egy külső adatbázissal.

A következő alfejezetekben tehát az előző alfejezetben kifejtett fő szempontokat pontosítjuk, az UML jelöléseit használva.

3.3. Az elemek katalógusa

A rendszer tehát a következő elemekből fog állni:

- Node: a rendszerben lévő bármilyen csomópont
- Center: olyan csomópont, mely központ
- Endpoint: olyan csomópont, mely végpont
- Sensor: olyan végpont, mely elsősorban üzeneteket küld
- Terminal: olyan végpont, mely elsősorban üzeneteket fogad
- Message: a csomópontok közötti üzenetek formátumát definiálja
- Driver: a letölthető eszközmeghajtók interfészét definiálja

3.4. Az elemek leírása

Node

- Leírás:

  Egy Erlang processzt jelöl, mely egyedi, nem megváltoztatható azonosítóval rendelkezik.

- Változók:

  Id – Egyedi azonosító, mely nem változtatható meg.

- Szolgáltatások:

  ping() – Egy pong atommal válaszol, jelezve, hogy a processz fut.

Center

- Leírás:

  Olyan Node-ot jelöl, melybe regisztrálhatnak szenzorok, valamint a regisztrált nevekre feliratkozhatnak terminálok. Csak továbbítja az üzeneteket, nem tényleges feladó vagy címzett.

- Változók:

  Sensors – Regisztrált szenzorok név-cím párjait tartalmazó lista.

  Subscriptions – Nevekre feliratkozott terminálok név-cím párjait tartalmazó lista.

- Szolgáltatások:

  start() – Center indítása.

  stop() – Center leállítása.

  reg(Address, Name) – Sensor címének regisztrálása névként.

  subscribe(Name, Address) – Névre feliratkozás egy Terminal adott címével.

  lookup(Name) – Sensor nevének feloldása címre.

  notify(Message) – Message feladása továbbítás céljából.

Endpoint

- Leírás:

  Olyan Node-ot jelöl, mely csak küld vagy fogad üzeneteket, nem továbbít.

- Változók:

  Messages – Beérkezett üzenetek listája.

- Szolgáltatások:

  start(Config) – Endpoint indítása adott beállításokkal.

  stop() – Endpoint leállítása.

Sensor

- Leírás:

  Olyan végpontot jelöl, mely a külvilág valamely változásának hatására üzenetet küld. Támogathat még vezérlést, illetve explicit lekérdezést is.

- Változók:

  Centers – Azon központok listája, melyeket értesíteni kell, ha változott a környezet.

- Szolgáltatások:

  query(Message) – Lekérdez egy adott funkciót, és a feladó címére megküldi.

  control(Message) – Beállít egy adott funkciót.

Terminal

- Leírás:

  Olyan végpontot jelöl, mely elsősorban üzenetek fogadására hivatott. Opcionálisan üzeneteket is lehet vele küldeni, válaszként egy korábban egy szenzortól kapott üzenetre.

- Változók:

  Centers – Azon központok listája, melyekre fel kell iratkozni induláskor.

- Szolgáltatások:

  notify(Message) – Üzenet átadása a terminál számára.

Message

- Leírás:

  Egy Erlang ennest jelöl, mely az adat mellett tartalmazza az adat típusát, feladóját, címzettjét, továbbítóját.

- Változók:

  Data – Egy szám, a mért érték vagy döntés.

  Description – Ha a szenzor több típusú értéket is mérne, ez mondja meg, hogy melyik típust jelöli a Data mező.

  From – A feladót jelöli.

  To – A címzettet jelöli.

  Receiver – A központot jelöli.

- Szolgáltatások:

  Nincsenek.

Driver

- Leírás:

  Egy Erlang függvényt ír le, lehetővé téve, hogy különböző szenzorokat egységes interfészen át kezeljünk. Az eszközmeghajtónak megküldjük, hogy melyik Node hányadik funkcióját akarjuk lekérdezni/vezérelni, majd az megmondja a funkció nevét, amit már az eszköz megért.

  Például a `translate(sensor0@clevo.local, 0)` hatására az adott meghajtó válasza lehet a `sugar`, mely egy vérnyomás- és vércukormérő esetén a vércukor lekérdezését teszi lehetővé.

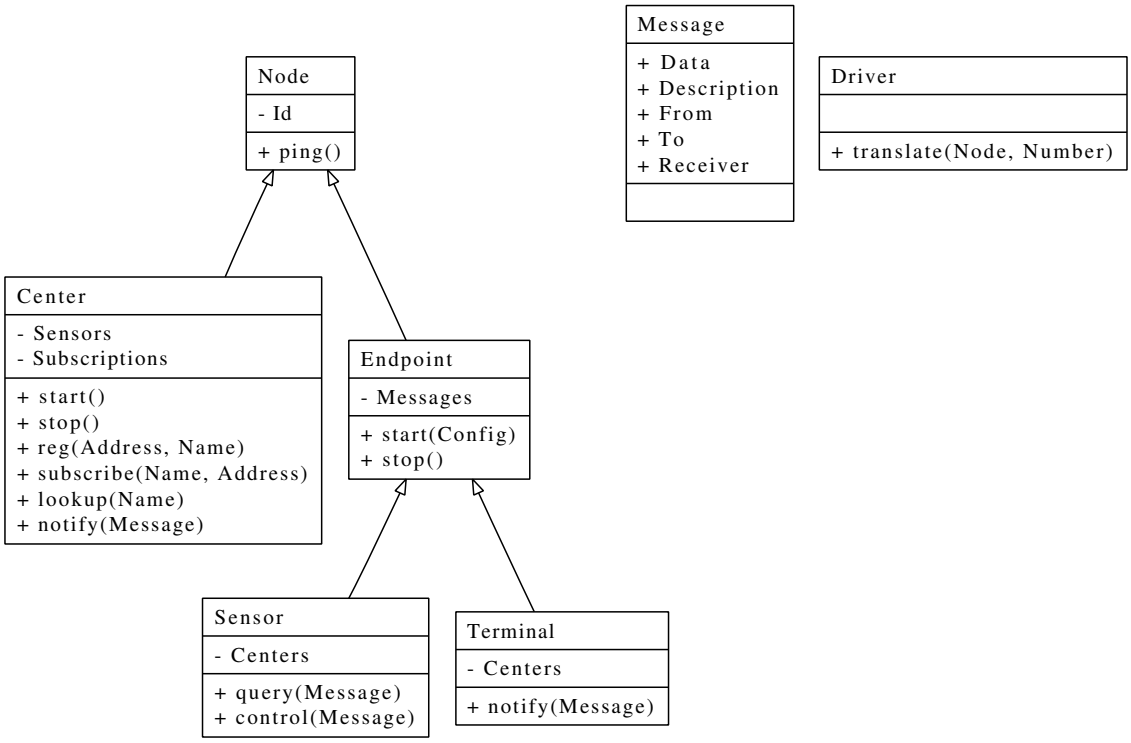
- Változók:

  Nincsenek.

- Szolgáltatások:

  translate(Node, Number) – Megad egy atomot, melyet metódusnévként használhatunk ha a szenzort explicit módon akarjuk lekérdezni.

3.5. Statikus struktúradiagram

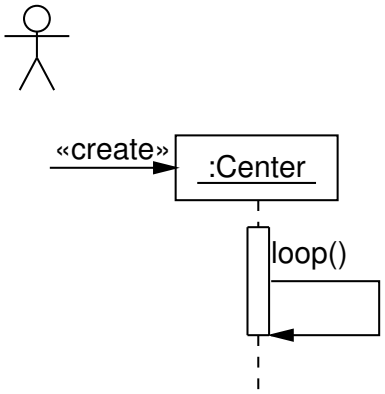


5. ábra. A rendszer statikus struktúradiagramja

A statikus struktúradiagram a rendszer elemeiről tárolt adatokat, azok összefüggéseit és kapcsolatait mutatja.

3.6. Szekvenciadiagramok

Központ indulása

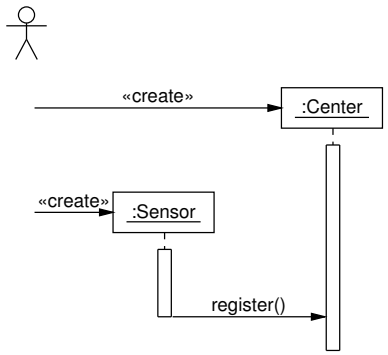


6. ábra. Szekvenciadiagram: központ indulása

Az ábrán látható, hogy a központot mindig a felhasználó helyezi üzembe, és a központ bekapcsolás után belép a várakozási hurokba.

Ebből a helyzetből aztán majd később a regisztráló vagy riasztó szenzorok és feliratkozó terminálok mozdíthatják ki.

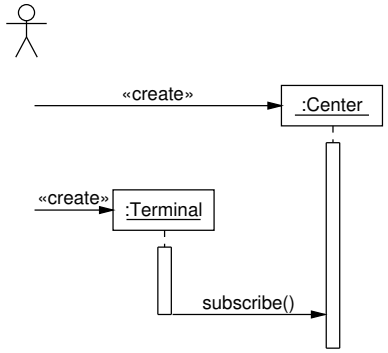
Szenzor regisztrációja



7. ábra. Szekvenciadiagram: szenzor regisztrációja

Az ábra mutatja, hogy a szenzor bekapcsolásakor már legalább egy központnak bekapcsolt állapotban kell lenni a rendszerben. Ekkor a szenzor regisztrál, majd a központ elküldi a címét, melyre a szenzor jelezhet, ha az szükséges.

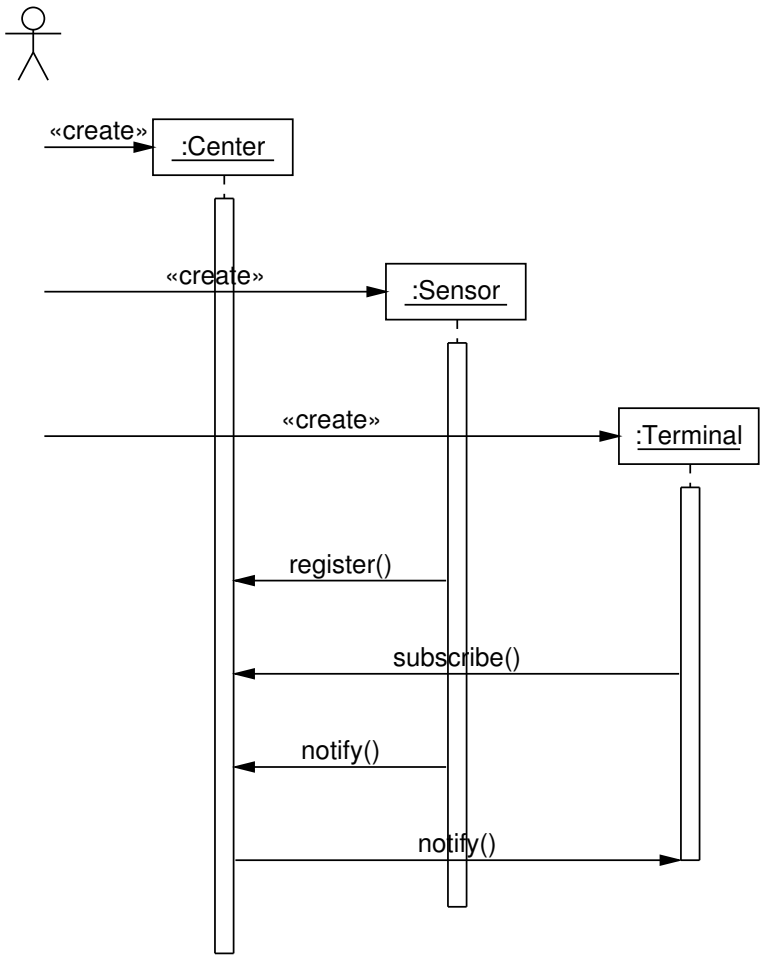
Terminál regisztrációja



8. ábra. Szekvenciadiagram: terminál regisztrációja

A terminál regisztrációja esetén is előkövetelmény legalább egy központ működése, ahova feliratkozhat a terminál, de itt a központnak nem kell azonosítót küldenie, hiszen a terminál nem fogja értesíteni a központot eseményekről.

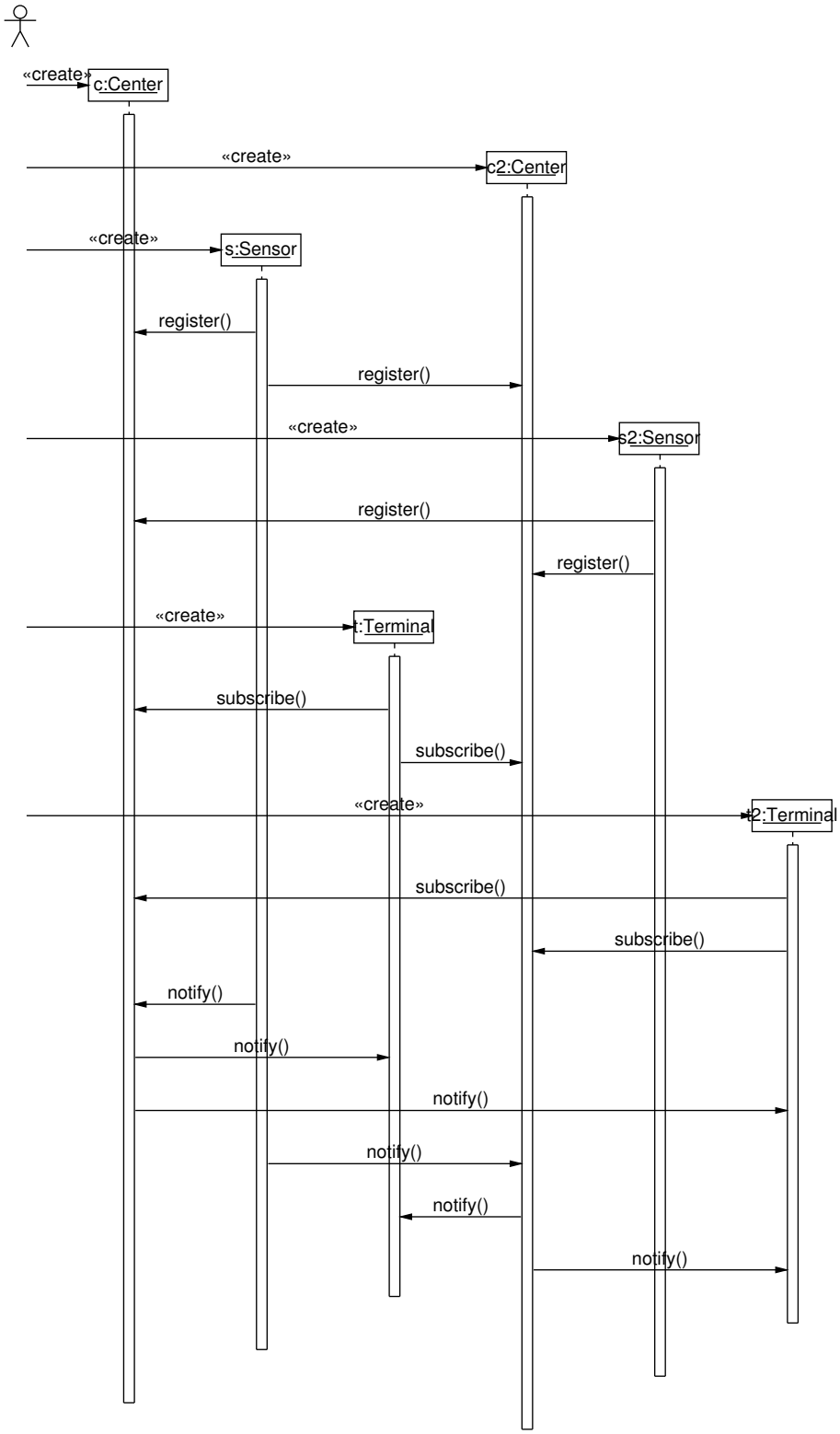
Üzenetküldés szenzorról



9. ábra. Szekvenciadiagram: üzenetküldés szenzorról

Az ábra azt mutatja, hogy az üzenetküldés akkor értelmes, ha egy központ bekapcsolása és a szenzor regisztrációja után legalább egy terminál feliratkozott az üzenetekre az üzenetküldés előtt. Ilyenkor a szenzor a központot értesíti, a központ pedig a terminált.

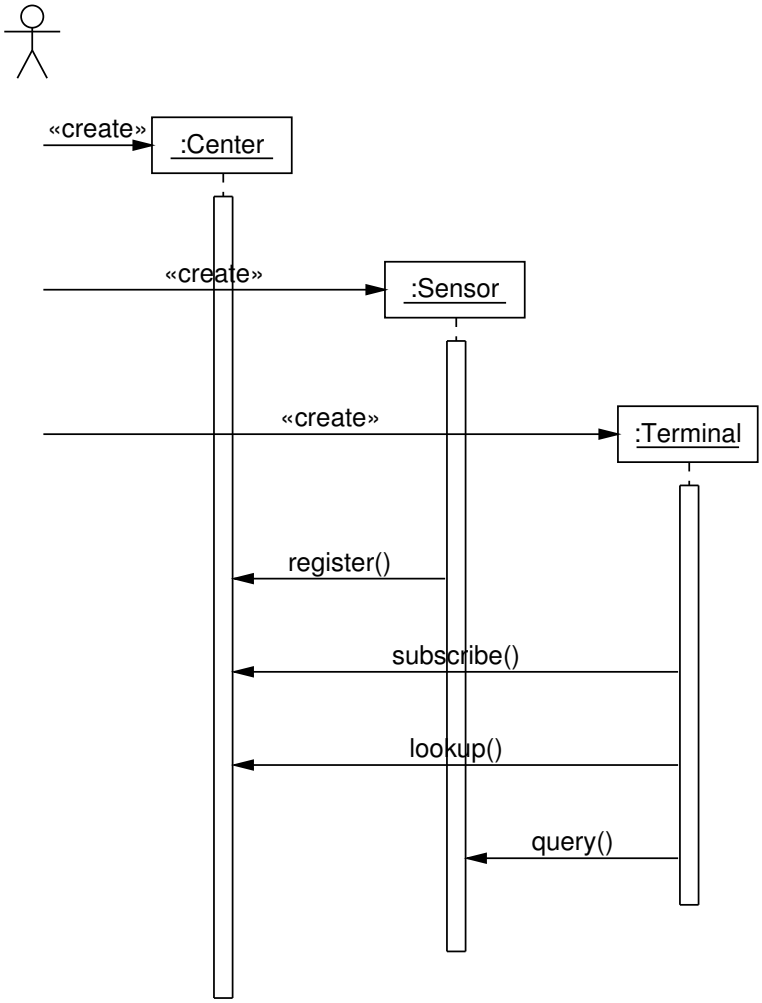
Üzenetküldés szenzorról, redundáns eset



10. ábra. Szekvenciadiagram: üzenetküldés szenzorról redundáns esetben

Az előző eset általánosítása, ha több szenzort, központot és terminált helyezünk el a rendszerben. Az ábra két-két példány kommunikációját mutatja be abban az esetben, ha olyan esemény következik be, melynek hatására mindkét szenzor jelez.

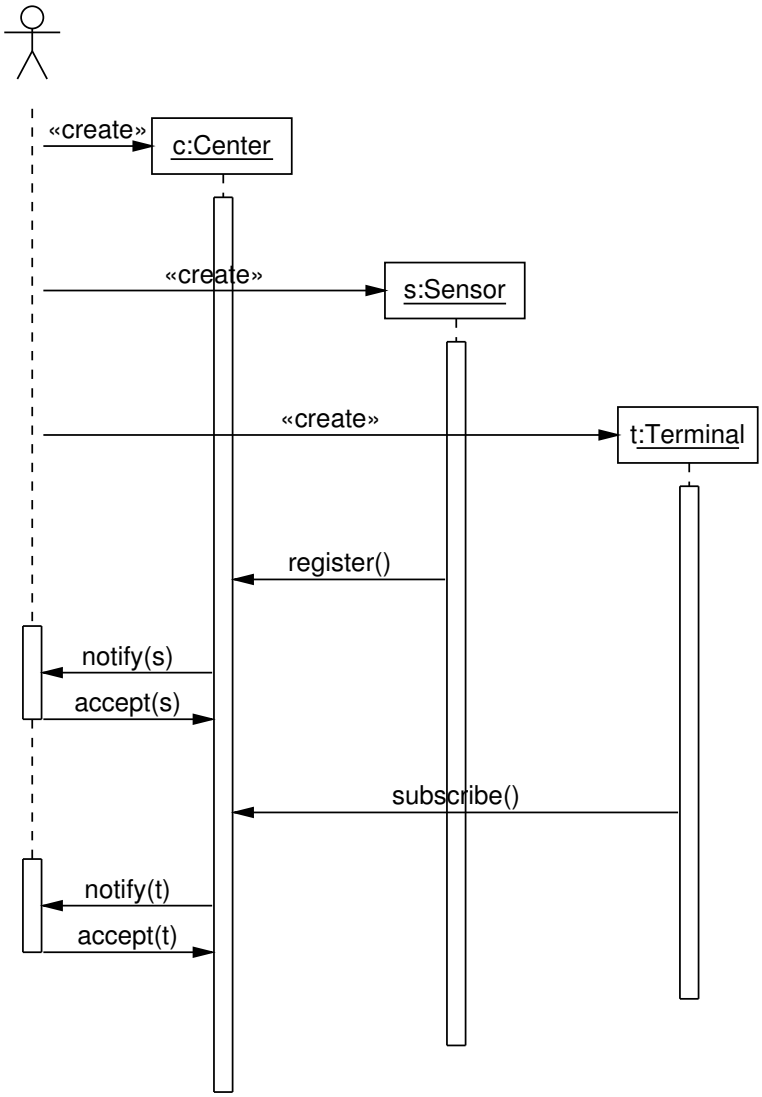
Explicit lekérdezés terminálról



11. ábra. Szekvenciadiagram: explicit lekérdezés terminálról

Explicit lekérdezés esetén a terminál először a központhoz fordul a szenzor címéért, majd a cím birtokában közvetlenül a szenzornak küld üzenetet az adatok lekérdezése céljából.

Eszköz engedélyezése



12. ábra. Szekvenciadiagram: eszköz engedélyezése

Az eszközök engedélyezése esetén az ábrán látható módon első alkalommal, ha egy szenzor regisztrálni vagy egy terminál feliratkozni akar, akkor a felhasználónak azt egyszer engedélyeznie kell.

4. Megvalósítás

A megvalósítás során a tervezés folyamán elemzett használati eseteket fogjuk sorra implementálni. Terjedelmi okokból nem valósítjuk meg a teljes funkcionalitást, így célszerű mindjárt az elején definiálni, hogy mely részletek esetén alkalmazunk egyszerűsítéseket, és hol cél a teljes megvalósítás.

A tervezés során definiált osztályok, az párhuzamosság, hibatűrés és elosztottság támogatása a program részét képezik. Az elkészítés során nyomon fogjuk követni, ahogy a tervezés szakaszában leírt használati esetek sorra működőképessé válnak.

Amit nem vagy csak egyszerűsítve valósítunk meg:

- A szenzorok nem automatikusan fedezik fel a központokat, hanem egy konfigurációs állomány előre definiálja ezek listáját.
- A korábban felvetett biztonsági kérdésekkel nem foglalkozunk, feltételezzük, hogy a rendszer zárt, a rendszer elemei pedig megbízhatóak.
- Eltekintünk a grafikus felhasználói felület létrehozásától, egyszerű parancssoros interfészt adunk a felhasználónak.

4.1. Első, már működő változat

A legkisebb működő változata a rendszernek az az eset, mikor a rendszerben egy szenzor, egy központ és egy terminál van. Mivel ez lesz minden későbbi változat alapja, tekintsük át tüzetesen a működését. A későbbiekben csak az ehhez a változathoz képest eszközölt módosításokat ismertetjük.

E változat működése során a felhasználó egy központot indít, majd egy terminált, mely regisztrál a központba, végül egy szenzort, mely indulás után azonnal értesítést küld. Ezt az értesítést kapja meg a központ, majd továbbítja a terminálnak.

Központ

A központ három olyan függvényt definiál, mely a processz életciklusát érinti:

```
start() -> register(center, spawn(fun() -> put(sensors, []), put(
    subscriptions, []), loop() end)).

stop() -> center ! stop.

ping() -> rpc({ping}).
```

A `start/0` függvény egy új processzt indít az aktuális csomóponton, mely inicializálja a szenzorok és feliratkozások listáját, majd várakozó állapotba kerül, ld. később.

A `stop/0` függvény ennek az új processznek küld egy üzenetet, melynek hatására az leáll.

A `ping/0` függvény kizárólag hibakeresési célokat szolgál a lejjebb ismertetésre kerülő `rpc/1` függvény felhasználásával, lehetővé téve, hogy megvizsgáljuk, hogy a központ válaszképes-e.

További négy függvény pedig a központ tényleges vezérlésére szolgál:

```
reg(Address, Name) -> rpc({reg, Address, Name}).

lookup(Name) -> rpc({lookup, Name}).

subscribe(Name, Address) -> rpc({subscribe, Name, Address}).

notify(Message) -> rpc({notify, Message}).
```

Láthatjuk, hogy ezek mind a lenn ismertetett `rpc/1` függvény köré épített csomagoló függvények.

A csomagoló függvények által hívott `rpc/1` függvény a következő:

```
rpc(Q) ->
 center ! {self(), Q},
 receive
  {center, Reply} ->
   Reply
 end.
```

A függvény a korábban center néven regisztrált [2] processznek küld egy két elemű ennest, megküldve a saját címét, valamint a tényleges adatokat, majd a válaszként kapott értékkel tér vissza.

Végül a center néven regisztrált processzünk fő függvénye a következő:

```
loop() ->
 receive
  {From, {ping}} ->
   From ! {center, pong},
   loop();
  {From, {reg, Address, Name}} ->
   io:format("[~p] reg(~p,~p)~n", [node(), Address, Name]),
   Sensors = get(sensors),
   put(sensors, [{Address, Name}| Sensors]),
   From ! {center, ok},
```

[2] Regisztráción és regisztráció törlésén itt az Erlang rendszer `register/2` és `unregister/1` függvényeit értjük, ami lehetőséget ad arra, hogy az aktuális csomóponton Pid-ekhez atomokat rendeljünk. Ezek a regisztrációk más csomópontokról nem láthatóak, a teljes rendszer szintjén nem egyediek, és ezt ki is fogjuk használni.

```
    loop();
  {From, {lookup, Name}} ->
    io:format("[~p] lookup(~p)~n", [node(), Name]),
    Sensors = get(sensors),
    A = [A || {A, N} <- Sensors, N =:= Name],
    From ! {center, {ok,A}},
    loop();
  {From, {subscribe, Name, Address}} ->
    io:format("[~p] subscribe(~p,~p)~n", [node(), Name, Address]),
    Subscriptions = get(subscriptions),
    put(subscriptions, [{Name, Address}| Subscriptions]),
    From ! {center, ok},
    loop();
  {From, {notify, Message}} ->
    io:format("[~p] notify(~p)~n", [node(), Message]),
    {Data, Description, Fro, To, _Receiver} = Message,
    lists:foreach(fun(I) -> rpc:call(I, terminal, notify, [{Data,
        Description, Fro, To, node()}]) end,
     [A || {N, A} <- get(subscriptions), N =:= To]),
    From ! {center, ok},
    loop();
  stop ->
    init:stop()
end.
```

Az egyes beérkező üzenetekre tehát a következő válaszokat adja:

- A ping atomra mindig pong atommal válaszol.
- A reg atomra a paraméterként kapott szenzor címét és nevét regisztrálja a egy listába, melyet a process dictionary-ben tárol (mely az Erlang futtatórendszer része), így később is lehetővé válik a módosítása.
- A lookup atomra a paraméterként kapott névhez tartozó szenzor címet adja vissza a sensors listából.
- A subscribe atomra a paraméterként kapott névhez eltárolja a kapott címet.
- A notify atom esetén egy öt elemű ennest (melynek elemeit a tervezés során Message néven definiáltuk) vár paraméterként. Ebben szerepel az, hogy milyen névre szól az üzenet. A központ azoknak a terminálóknak továbbítja az üzenetet, amelyek ilyen névvel iratkoztak fel, beleírva saját magát mint továbbítót.
- Végül a stop atomra leállítja az aktuális Erlang csomópontot.

A másik két elemmel ellentétben itt megfigyelhető, hogy a központnak nincs konfigurációs állománya, valamint elindítása után nincs semmilyen bekövetkező esemény.

Terminál

A struktúra diagramban láthattuk, hogy a Center és a Terminal is egy Node, így itt is megtalálható a processz életciklusával kapcsolatos három függvény:

```
start(ConfigFile) ->
 {ok, Config} = file:consult(ConfigFile),
 Centers = [ {C, E} || {center, C, E} <- Config],
 lists:foreach(fun({C, E}) -> rpc:call(C, center, subscribe, [E, node()])
    end, Centers),
 register(terminal, spawn(fun() -> loop() end)).

stop() -> terminal ! stop.

ping() -> rpc({ping}).
```

A `start/1` függvény azonban itt egy konfigurációs állományt vár paraméterül. Erre egy példa:

```
{center, center0@clevo.local, event}.
```

A konfigurációs állomány szintaxisára az egyetlen megkötés az, hogy minden egyes sorban egy Erlang termnek kell lennie, mivel ezt fogadja el a `file:consult/1` értelmező függvény. A fenti file-ban az egyetlen sor egy központra való feliratkozást definiál, erre utal a center atom. Az ennes másik két paramétere a központ címét [3] és annak az eseménynek a nevét definiálja, melyre fel szeretnénk iratkozni az adott központban.

Az indítás során tehát értelmezzük a beállításokat, és a kérésnek megfelelően feliratkozunk a központoknál, azok subscribe metódusát meghívva. Ha ez megtörtént, üzenetre várunk a lenn ismertetésre kerülő `loop/0` függvény használatával.

A `stop/0` és `ping/0` feladata és működése más ismerős, a center modulban definiáltakkal megegyező.

A terminálnak egyetlen, a modellben is létező metódusa a `notify/1`:

```
notify(Message) -> rpc({notify, Message}).
```

A `center:notify/1`-hez hasonló módon ez is csak csomagoló függvény az `rpc/1` köré.

A terminal modul maradék része a korábban hivatkozott `rpc/1` és `loop/0` függvényekből áll:

[3] A következő, tesztelésről is szóló fejezetben látni fogjuk, hogy mi az oka annak, hogy a cím ebben a formátumban lett megadva. Általánosságban a cím egy csomópontot és egy gépet azonosító atomból áll, a kukac karakterrel összekötve.

```
rpc(Q) ->
 terminal ! {self(), Q},
 receive
  {terminal, Reply} ->
   Reply
 end.

loop() ->
 receive
  {From, {ping}} ->
   From ! {terminal, pong},
   loop();
  {From, {notify, Message}} ->
   io:format("[~p] notify(~p)~n", [node(), Message]),
   From ! {terminal, ok},
   loop();
  stop ->
   init:stop()
 end.
```

Az `rpc/1` feladata tehát kommunikálni a terminal néven regisztrált processzel: az üzenetküldés után válaszra várni, majd azzal visszatérni. A center modulhoz hasonló módon itt is azért van erre szükség erre a csomagolásra, mivel terminal néven más csomóponton más processzt érhetünk el.

A `loop/0` függvény pedig üzeneteket vár:

- A ping és stop atomok az életciklus monitorozására és vezérlésére szolgálnak a center modullal megegyező módon.
- A notify atomra pedig a terminál egyszerűen a standard kimenetre küldi a kapott üzenet tartalmát, nem túl felhasználóbarát módon.

Szenzor

A sensor modul implementációja a start, stop és ping metódusokra:

```
start(ConfigFile) ->
 Data = data,
 {ok, Config} = file:consult(ConfigFile),
 Centers = [ {C, E} || {center, C, E} <- Config],
 lists:foreach(fun({C, E}) ->
  rpc:call(C, center, reg, [node(), E]),
  Message = {Data, desc, node(), E, false},
  rpc:call(C, center, notify, [Message])
 end, Centers),
```

```
 register(sensor, spawn(fun() -> put(desc, Data), loop() end)).

stop() -> sensor ! stop.

ping() -> rpc({ping}).
```

Láthatjuk, hogy a start függvény itt is használ egy konfigurációs állományt, melynek azonos a formátuma a szenzoréval. A feladata természetesen más: itt azt adja meg, hogy melyik központnak milyen néven kell elküldeni a mérési adatainkat.

Az első változat tehát rögtön indulás után küld egy üzenetet a konfigurációs állományban megadott központba, a megadott eseménynevet használva, mely arról fogja informálni a terminálokat, hogy a desc típusú mérési adat értéke data.

A tervezés során a szenzorok explicit lekérdezésének igénye is felmerült, az ehhez szükséges függvények:

```
query_data(Message) -> rpc({query_data, Message}).

control(Message) -> rpc({control, Message}).

rpc(Q) ->
 sensor ! {self(), Q},
 receive
  {sensor, Reply} ->
   Reply
 end.
```

Az előzőekben megszokott módon ezek is az `rpc/1` függvényt hívják, melynek működési elve nem változott a korábbi két modulhoz képest.

A korábban hivatkozott `loop/0` függvény pedig a következő:

```
loop() ->
 receive
  {From, {ping}} ->
   From ! {sensor, pong},
   loop();
  {From, {query_data, {_Data, _Desc, Fro, _To, _Recv}}} ->
   rpc:call(Fro, terminal, notify, [{get(desc), desc, node(), Fro, false}])
       ,
   From ! {sensor, ok},
   loop();
  {From, {control, {Data, _Desc, _Fro, _To, _Recv}}} ->
   put(desc, Data),
   From ! {sensor, ok},
   loop();
  stop ->
```

```
    init:stop()
end.
```

- A ping és stop atomok kezelését már ismerjük
- A query_data atom hatására a szenzor process dictionary-jéből lehet lekérdezni.
- A control atom segítségével pedig vezérelni lehet a szenzort.

4.2. Redundáns működés

A redundáns működés két szempontból különbözik az előző alfejezetben ismertetett változattól:

- A rendszer indításakor mindhárom elemből két-két példányt kell indítani.
- A terminálok és szenzorok konfigurációs állományában két-két központot kell definiálni.

Ez utóbbit például a következőképpen tehetjük meg:

```
{center, center0@clevo.local, event}.
{center, center1@clevo.local, event}.
```

4.3. Eszközmeghajtók támogatása

Meghajtóprogramokra akkor van szükségünk, ha egy általános interfészen keresztül akarunk elérni egy-egy eszközt, melynek a speciális jellemzőit nem ismerjük. Tegyük fel, hogy egy szenzor többféle mérést is képes végezni, és ezeket számozzuk. A megvalósított példa esetén a szenzor egy `querydesc/0` függvénnyel rendelkezik, a terminál ezt szeretné explicit módon lekérdezni, de csak annyit tud, hogy ez a nulladik típusú mérési funkciója az eszköznek. A driver modul fog abban segíteni, hogy a 0 alapján megkapja a querydesc atomot.

A modul egyetlen `translate/2` függvényt definiál az interfészében:

```
sub(Str,Old,New) ->
        RegExp = "\\Q"++Old++"\\E",
        re:replace(Str,RegExp,New,[multiline, {return, list}]).

translate(Node, Num) ->
        Module = sub(sub(atom_to_list(Node), "@", "_"), ".", "_"),
        compile:file(Module),
        Modulea = list_to_atom(Module),
        list_to_atom(Modulea:translate(Num)).
```

Az első segédfüggvény csomópont nevéből a modul nevét állítja elő, kukacokat és pontokat aláhúzásjellel helyettesítve. A második pedig az előállított modulnév alapján lefordítja az eszközmeghajtót, meghívja annak translate metódusát, és a kapott értékkel visszatér.

A sensor0_clevo_local modul egyetlen `translate/1` függvényt definiál:

```
translate(Num) ->
        case Num of
         0 ->
          "querydesc";
         _ ->
          false
        end.
```

Ahhoz, hogy ezt ki is próbálhassuk, a szenzor kódjában a querydesc függvényt implementálni kell:

```
querydesc() -> rpc({querydesc}).
```

Valamint a `loop/0` függvényben egy új esetet kell felvenni:

```
{From, {querydesc}} ->
 From ! {sensor, get(desc)},
 loop();
```

4.4. Felhasználói felület

Idáig alapvetően azzal foglalkoztunk, hogy hogyan tudunk adatokat küldeni Erlang csomópontok között.

A felhasználói felület feladata, hogy az idáig elkészült köztes réteget felhasználva a felhasználók számára is hasznos támogatást nyújtson, ezáltal tényleges életvitelt segítő rendszert megvalósítva.

Felhasználói felületünk kialakítását kezdjük a szenzorral. A `start/1` függvény idáig visszatért, ha sikeres volt az üzeneteket kezelő process regisztrációja terminal néven. Most ezt megváltoztatjuk, és interaktívan, a standard bemenetről fogunk beolvasni értékeket, majd azonnal jelzünk, ha a felhasználó ENTER-t ütött.

Ehhez a szenzor kódjában a `start/1` függvényt változtatjuk meg:

```
start(ConfigFile) ->
 Data = data,
 {ok, Config} = file:consult(ConfigFile),
 Centers = [ {C, E} || {center, C, E} <- Config],
 lists:foreach(fun({C, E}) ->
  rpc:call(C, center, reg, [node(), E])
 end, Centers),
 register(sensor, spawn(fun() -> put(desc, Data), loop() end)),
```

```
 io:format("Adja meg a mert adatot 'tipus szam' formaban, majd usson ENTER-
     t!\n"),
 io:format("A testhomerseklet merese utan peldaul 'h 38.2'\n"),
 read_stdin(Centers).
```

Az újonnan bevezetett read_stdin függvény pedig:

```
read_stdin(Centers) ->
 L = io:get_line("> "),
 [A|[B]] = re:split(L, " "),
 Desc = list_to_atom(binary_to_list(A)),
 {Data, _} = string:to_float(binary_to_list(B)),
 lists:foreach(fun({C, E}) ->
  Message = {Data, Desc, node(), E, false},
  rpc:call(C, center, notify, [Message])
 end, Centers),
 read_stdin(Centers).
```

A feladat második fele a terminál átalakítása; azt szeretnénk, hogyha a hőmérséklet értéke meghaladja a 37.2 fokot, akkor csipogjon, egyéb esetben pedig egyszerűen adjon barátságos kimenetet.

Ehhez a terminal modul `loop/0` függvényét a következőképpen módosítjuk:

```
loop() ->
 receive
  {From, {ping}} ->
   From ! {terminal, pong},
   loop();
  {From, {notify, Message}} ->
   {Data, Desc, _Fro, _To, _Recv} = Message,
   case Desc of
    h ->
     case Data > 37.2 of
      true ->
       io:fwrite([7]),
       io:format("Figyelem, a homerseklet ~p homerseklet tobb a
           megengedettnel!~n", [Data]);
      _ ->
       io:format("Uj homerseklet adat: ~p.~n", [Data])
      end;
    _ ->
     io:format("A '~p' uj erteke: ~p.~n", [Desc, Data])
   end,
   From ! {terminal, ok},
   loop();
  stop ->
```

```
    init:stop()
  end.
```

4.5. Hibatűrés

A rendszerrel szemben elvárás, hogy hibatűrő legyen, azonban ezt idáig nem valósítottuk meg, például ha hőmérsékletnek nem lebegőpontos számot adunk meg, akkor nem kapunk semmilyen hibát, egyszerűen hibás lesz a működés.

Két feladatunk van tehát:

- Hiba esetén biztosítani, hogy leálljon az adott processz, mielőtt az hibás működéshez vezetne.
- A leállt processz helyébe újat indítani.

A leállás biztosításához a `terminal:read_stdin/0` függvényt a következőképpen módosíthatjuk:

```
read_stdin(Centers) ->
 L = io:get_line("> "),
 [A|[B]] = re:split(L, " "),
 Desc = list_to_atom(binary_to_list(A)),
 {Data, _} = string:to_float(binary_to_list(B)),
 case Data of
  error ->
   erlang:error(badarg);
  _ ->
   lists:foreach(fun({C, E}) ->
      Message = {Data, Desc, node(), E, false},
      rpc:call(C, center, notify, [Message])
    end, Centers),
   read_stdin(Centers)
 end.
```

Az új processz indítása két lépéses folyamat. Először is definiálni kell egy függvényt, mely akkor hívódik meg, ha az aktuális processz leáll:

```
on_exit(Pid, Fun) ->
 spawn(fun() ->
  process_flag(trap_exit, true),
  link(Pid),
  receive
   {'EXIT', Pid, _Why} ->
    Fun()
  end
 end).
```

Valamint a `start/1` függvényben ezt a függvényt regisztrálnunk kell:

```
start(ConfigFile) ->
 Data = data,
 {ok, Config} = file:consult(ConfigFile),
 Centers = [ {C, E} || {center, C, E} <- Config],
 lists:foreach(fun({C, E}) ->
  rpc:call(C, center, reg, [node(), E])
 end, Centers),
 catch unregister(sensor),
 register(sensor, spawn(fun() -> put(desc, Data), loop() end)),
 on_exit(self(),
  fun() ->
    io:format("A szenzor hibaval lepett ki, ujrainditas...~n"),
    start(ConfigFile)
  end),
 io:format("Adja meg a mert adatot 'tipus szam' formaban, majd usson ENTER-
    t!\n"),
 io:format("A testhomerseklet merese utan peldaul 'h 38.2'\n"),
 read_stdin(Centers).
```

5. Eredmények, jövőbeli munka

5.1. Eredmények

Tesztelési környezet

A tényleges eredmények áttekintése előtt ismertetjük, hogy milyen környezetben készült el a munka. A dolgozat tördelése a TEX 3.141592 verziójával készült, Frugalware Linux operációs rendszeren. A programok teszteléséhez az Erlang R12B-5 verzióját használtam.

A dolgozatban ismertetett kódok kipróbálásának feltétele, hogy az egyes Erlang csomópontoknak nevet adjunk. A tesztelés során a következő beállításokat használtam:

- A tesztrendszer két fizikai gépből állt: a gyártó, illetve terméknév után a gépek IP címeit a rendszer `/etc/hosts` file-jába `clevo.local` és `ibook.local` néven írtam be mindkét rendszeren.

- Az Erlang csomópontok elnevezésére parancssori kapcsolót használtam.

  Példa a `sensor0` nevű csomópont indítására a `clevo` gépen:

  ```
  erl -name sensor0@clevo.local
  ```

- Mikor a tesztelést két csomópont között fizikailag azonos gépen végeztem, akkor nem kellett tudni arról, hogy az Erlang rendszerben két csomópont akkor kommunikálhat egymással, ha azok *cookie-jai* megegyeznek. Ennek értéke parancssori kapcsolót nem használva a `~/.erlang.cookie` file tartalma, mely ha nem létezik, automatikusan létrejön és véletlenszerű tartalommal töltődik fel. Ha fizikailag két külön gépen tesztelünk, akkor ez a véletlenszerű érték különbözni fog, és így a tesztelés sikertelen lesz. Erre parancssori kapcsoló használata a megoldás:

  ```
  erl -name sensor0@clevo.local -setcookie s3cr3t
  ```

  A másik probléma abból adódott, hogy mindkét gép rendelkezett saját tűzfallal. Ezt két részproblémára lehetett bontani. Egyrészt az epmd (Erlang Port Mapper Daemon) fixen az 4369-es porton akar bejövő kéréseket kiszolgálni, ezt engedélyezni kell a tűzfalon. Másrészt az egyes távoli eljáráshívásokhoz egy port-tartományt kell kijelölnünk. Ezt a következő opciókkal érhetjük el:

  ```
  erl -name sensor0@clevo.local -setcookie s3cr3t -kernel
      inet_dist_listen_min minimum inet_dist_listen_max maximum
  ```

  A minimum és a maximum értéke egyezhet, például állíthatjuk mind a kettőt 4370-re. Ha e portok ki- és bemenő forgalma engedélyezett a tűzfalon, akkor a tűzfal több beállítást már nem igényel.

Kiindulási állapot, feladat

A munka megkezdésekor bizonyos ismeretekkel rendelkeztem már az szekvenciális Erlang programozásával kapcsolatban, valamint adott volt az fent ismertetett Erlang környezet. A cél a hasonló megoldások felkutatása, valamint az Erlang nyelv alkalmasságának vizsgálata volt abból a szempontból, hogy ha egy életviteli rendszert szeretnénk létrehozni, azt érdemes-e Erlang környezetben megvalósítani.

Elért eredmények

A munka végére világossá vált, hogy egy ilyen rendszer létrehozása lehetséges, egy kis életviteli rendszert meg is terveztem Erlang környezetben, valamint néhány jellegzetes elemét meg is valósítottam.

Mint arról korábban szó volt, a megvalósított rendszer legkomolyabb hiányossága a biztonság megfelelő kezelése: ha egy Erlang csomópont egy másik Erlang csomóponthoz hozzáférést szerez, onnantól azon bármilyen kódot futtathat.[19] Természetesen erre a problémára több részmegoldás is született, de mindegyiknek megvan a maga problémája:

- Választhatjuk, hogy nem használjuk az Erlang elosztott képességeit, és például egy saját protokollt vezetünk be, és annak értelmezésekor döntjük el, hogy a bejövő kéréseket kiszolgáljuk-e vagy sem. Ekkor sok olyan funkciót kell saját magunknak megvalósítani, melyet idáig az Erlang nyelvi szinten adott számunkra.

- Egy másik egyszerű megoldás az, hogy zárt rendszert feltételezünk, ahol nincsenek megbízhatatlan felhasználók. Ha ezt az esetet bővítjük azzal, hogy a kódokat valamilyen technikával aláírjuk, és csak aláírt kódot hajtunk végre, akkor sok szituációban elegendő a nyújtott biztonsági színvonal, például megfelelő lehet egy kis életviteli rendszer számára.

Azt azonban el kell ismernünk, hogy egyik megoldás sem ad olyan fokú biztonsági beállítási lehetőségeket, mint amilyeneket például a Java futtatókörnyezet beépítetten támogat. Ez természetesen érthető is, hiszen az Erlang környezetet eredetileg zárt, telefonos hálózatokba tervezték, ahol nem volt szükség a jelenleg rendelkezésre állónál finomabb biztonság-kezelésre.

Mindezek ellenére megállapíthatjuk, hogy egy kis életviteli rendszer megvalósítása során ki tudjuk használni az Erlang környezet által nyújtott legfontosabb szolgáltatásokat: az elosztottság hatékony kezelését, a párhuzamosság jó támogatottságát és a hibatűrési mechanizmusokat.

5.2. Jövőbeli munka

Sok olyan probléma került elő a munka során, mellyel terjedelmi okokból nem foglalkoztam, de a jövőben még foglalkozni lehetne. A legfontosabbak:

- A felhasználói felületet le lehetne cserélni egy grafikus verzióra.
- A szenzorok és terminálok jelenleg a beállítási file-jukba fixen beírt központokhoz csatlakoznak. Ezt automatizálni lehetne, például ha feltételezhetnénk, hogy a rendszer alapértelmezett átjárója egyben egy központ is, vagy ha egy újabb elemet vezetnénk be, amely az alapértelmezett átjárón fut, és a központok naprakész listáját szolgáltatná.
- A mért adatokat jelenleg nem tároljuk semmilyen módon, egy Mnesia vagy valamilyen SQL adatbázisban való tárolásnak számtalan előnye lenne.
- A központ is kaphatna felhasználói felületet, ahol az egyes eszközöket a tervezés fejezetben ismertetett módon engedélyezni lehetne.
- A rendszer működésének demonstrálása sokkal látványosabb lehetne valódi szenzorokkal. Az előző félévben az önálló laboratóriumi munka keretében egy vérnyomásmérőhöz Erlang eszközmeghajtót írtam, célszerű lenne azt integrálni ebbe a rendszerbe.

Hivatkozások

[1] eVITA program: életviteli technológiák és alkalmazások. http://de.njszt.hu/node/87.

[2] OSGi Szövetség (korábbi néven Open Services Gateway initiative, mára elavult név): Hivatalos honlap. http://www.osgi.org/.

[3] Joe Armstrong: Programming Erlang. http://www.pragprog.com/titles/jaerlang/programming-erlang.

[4] Tunstall: Hivatalos honlap. http://www.tunstall.co.uk/.

[5] Oy Exrei Ab: Hivatalos honlap. http://exrei.fi/.

[6] Everon: Always there for you. http://www.everon.fi/eng/.

[7] Sonaris Kft. http://www.sonaris.com/ceg.htm.

[8] IST International Security Technology Oy. http://www.istsec.fi/en.php.

[9] Nurse Háziápolási Szolgálat: Vivago betegfelügyeleti rendszerrel támogatott 24 órás felügyelet. http://hun.haziapolas.hu/HaziapolasHunIndex.aspx?Id=1a3e4423-106b-4753-%a516-3e1c24489d6d.

[10] Telcomed: About us. http://www.telcomed.ie/about_us.html.

[11] ONC RPC Version 1. http://tools.ietf.org/html/rfc1057.

[12] ONC RPC Version 2. http://tools.ietf.org/html/rfc5531.

[13] Microsoft Distributed Component Object Model (DCOM) Remote Protocol Specification. http://msdn.microsoft.com/library/cc201989.aspx.

[14] The OMG's CORBA Website. http://www.corba.org/.

[15] XML-RPC Home Page. http://www.xmlrpc.com/.

[16] BT Group plc. http://www.bt.com/.

[17] BT's 21st Century Network. http://www.btplc.com/21CN/.

[18] Joe Armstrong: Why OO Sucks. http://www.sics.se/~joe/bluetail/vol1/v1_oo.html.

[19] Joe Armstrong: A Fault-tolerant server. http://www.sics.se/~joe/.