Erlang/OTP: magas rendelkezésre állású, elosztott rendszerek fejlesztése

Czinkos Zsolt

2012-04-28

Kivonat

Az Erlang programozási nyelvet az Ericssonnál hozták létre hálózati eszközök, telefonrendszerek programozására. A magas telekom elvárások tükröződnek az Erlangban fejlesztett rendszerek architektúrájában, amely az Open Telecom Platform szoftverkönyvtárban ölt testet. A dolgozat bemutatja az Erlang programozási nyelvet, a fejlesztési elveket és alkalmazási lehetőségeit a webes technológiára épülő alkalmazások területén.

Mindenkinek, aki szereti

Tartalomjegyzék

1. fejezet

Bevezetés

Az Internet – azon belül különösen a Web – terjedésével párhuzamosan nőtt az igény kiszámítható, jó minőségű szolgáltatásokra. A szolgáltatóknak egyre magasabb elvárásoknak kell megfelelnie – nem utolsó sorban azért, mert a felhasználók fejében a web és az ingyenes tartalom összeforrt. Még színvonalas termékekért is nehezen adnak ki pénzt, nemhogy hibás, elavult tartalomért, akadozó és kiszámíthatatlanul működő szolgáltatásokért. Ma már a hálózat nem csupán mérnököknek, kutatóknak érdekes kábelezést jelent, amely így-úgy hasznos a tudományos kutatásaik során, hanem a mindennapi élet részét képező társadalmi kapcsolatok *reprezentációját* is. A felhasználó egyre aktívabb *cselekvője* ezeknek a valós vagy virtuális világban létrejött hálózatoknak, egyre inkább itt keresi (és többnyire találja meg) azt a teret, ahol ismerik, és ő is ismer, ahol ura annak az eszköztárnak, amelynek birtokában különböző – rövid, prompt, aszinkron, szöveg, hang vagy videó – *üzenetek* segítségével ápolni tudja kapcsolatait. Ez a kapcsolatrendszer és eszköztár jelenti azt az új mikrokozmoszt, amelyben a felhasználó – cselekvő- és befolyásolóképessége tudatában – kényelemben és biztonságban érzi magát.

Ez a kényelem és biztonság *függővé* tesz: világunk megszokott működésének zavarait nehezen vagy egyáltalán nem tudjuk tolerálni, kiszolgáltatottnak és tehetetlennek érezzük magunkat. Ilyenkor derül ki, hogy bár mikrokozmoszunkat ismerni véljük, az azt működtető rendszer elemeit meg sem tudjuk nevezni, csak azt tudjuk, hogy „van” (ez a valami pillanatnyilag a legtöbb ember számára néhány cég szolgáltatásában ölt testet: Facebook, Google, Twitter). A szoftverfejlesztőknek, tervezőknek ennek a világnak a működtetéséhez szükséges rendszert kell tudniuk megépíteni és üzemeltetni úgy, hogy a felhasználók a lehető legkevesebb alkalommal szembesüljenek azzal, hogy kihúzták alóluk a talajt. Nem emberbaráti, hanem üzleti megfontolások miatt.

A weben a sikerhez elengedhetetlen a folyamatos és megbízható szolgáltatás, nagyon alacsony az ingerküszöb, ha egy oldal betöltődése tovább tart mint 4 másodperc, már

odébb is állt a felhasználó (Akamai felmérés, 2006). Ha túl sokszor kap hibaüzenetet – amitől jobb esetben ingerült lesz, rosszabb esetben halálra rémül, hátha ő rontott el valamit –, keres mást. Éppen ezért nagyon fontos, hogy olyan rendszert építsünk, amely

1. folyamatosan, megszakítás nélkül működik;
2. megfelelő válaszidővel, sebességgel működik;
3. funkcionálisan jól működik;
4. a felmerülő hibák nyomon követhetők, kezelhetők.

A fentebb már említett vezető webes cégek mind megfelelnek ezeknek a követelményeknek, persze nem kevés munka és pénz árán. A felhasználót azonban a legkevésbé sem érdekli, hogy a szolgáltatást nyújtó üzleti vállalkozás hogyan tudja működtetni rendszerét, hány embert alkalmaz, stb. Őt az érdekli, hogy neki ingyen vagy elérhető áron a lehető legtöbbet nyújtsa. Ez az elvárása sajnos (vagy szerencsére) nem csak a mammutcégekkel szemben áll fent, hanem minden webes céggel szemben, mindenhol szeretné megkapni azt a minőséget, amihez hozzászokott. Azt a céget tekinti profinak, jónak, amely ugyanazt tudja nyújtani. Ha egy cég sikert akar, akkor már induláskor fel kell készülnie arra, hogy ha elsül a kapanyél, és özönlenek a felhasználók, akkor tartani tudja az iramot, ki tudja szolgálni ugrásszerűen megnőtt ügyfélkörét; miközben egy kezdő vállalkozás nem engedhet meg magának földrajzilag diverzifikált többtízezer gépes szerverparkot: kicsiből indulva kell képesnek lennie a növekedésre.

Hogyan lehet olyan rendszert építeni, amellyel neki lehet vágni egy webes vállalkozásnak úgy, hogy ne kelljen attól félni, mi lesz, ha holnap regisztrál még 10 ezer felhasználó (4 másodperc!), vagy ha tönkremegy az egyik gép?

Számos programozási nyelv és környezet közül lehet ma már választani, amely alkalmas erre a feladatra, ez a dolgozat az Erlang programozási nyelvet és a hozzá kapcsolódó Open Telecom Platform-ot (OTP) mutatja be. Az Erlang egy funkcionális programozási nyelv, amelyet az Ericsson fejlesztett ki mintegy 20 évvel ezelőtt telefonrendszerek, szoftveres kapcsolóközpontok programozásához, a telekommunikációs iparban szokásos rendkívül magas elvárásoknak megfelelően.

Az Erlang megalkotásánál az elsődleges cél magas rendelkezésre állású (*highly available*), hibatűrő (*fault tolerant*) redundáns rendszerek építése volt. Ez az, amire az Erlang igazán alkalmas, ez az a terület, ahol az Erlangnak évtizedes múltja van: akár 99,999%-os rendelkezésre állás biztosításában. Az hozzávetőleg 5 perc kiesés évente (**?**).

1998-ban open source-szá vált a nyelv és a platformot adó szoftverkönyvtárak, azóta bárki használhatja bármilyen feladatra, számos önkéntes és cég teszi be a közösbe a maga alkalmazását: HTTP szervert, NoSQL adatbáziskezelőt, stb.

2. fejezet

Elméleti alapok

Az Erlang nyelvet rendkívül magas rendelkezésre állású, elosztott rendszerek készítéséhez hozták létre. Nem akadémiai környezetben született, a legfőbb cél az volt, hogy profitot termeljenek a segítségével: a szoftverek hamarabb készüljenek el; a megrendelő azt kapja, amit szeretett volna; a termékek karbantarthatóak legyenek; redundáns, elosztott környezetben működjenek (folyamatosan); kapacitásuk növelhető legyen. A vezérelv az volt, hogy „minden szoftver hibás", mindig előfordul olyan hiba, amelyet nem kellő körültekintéssel, nem megfelelő specifikáció birtokában megírt szoftver okoz. Nem lehet úgy tenni, mintha egy szoftver valaha is „készen" lenne, és utána már nem kellene javítani, új igényeknek megfelelően bővíteni (újabb hibákat elkövetni). Azt is figyelembe kellett venni, hogy a magas rendelkezésre állás megkövetelte redundáns architektúra párhuzamos programozást igényel, nem egy processzoron kell futni, hanem többön, sőt több gépen. Az Erlang egyik megalkotója, Joe Armstrong, az itt leírt elvekre, módszerekre a *concurrency-oriented programming* kifejezést használta.

Az Erlang lényegében egy magas szintű nyelv konkurens rendszerek fejlesztéséhez. A párhuzamos programozás komplex feladatok esetében sokkal képzettebb, tapasztaltabb (így drágább) fejlesztőket kíván meg, egyáltalán nem kicsi a lépés a nem párhuzamos pogramozástól (*sequential programming*). Egyszerűen fogalmazva: egynél több cselekvő dolgozik egyazon rendszeren belül, ezért az egyes állapotváltoztatásoknál gondoskodni kell arról, hogy a több helyről kezdeményezett műveletek után a rendszer állapota konzisztens maradjon.

A Java nyelvben például a párhuzamos programozást a *thread*-ek teszik lehetővé, a konzisztens állapot megőrzését pedig a *synchronized* kulcsszóval jelölt metódusokkal, programblokkokkal lehet elérni. Az Erlanggal más utat választottak, két alapra építettek: a *funkcionális programozásra* és az *aktor modellre*.

2.1. Funkcionális programozás

Az Erlang funkcionális programozási nyelv. Az imperatív nyelvekkel szemben, ahol egy implicit állapot (*state*) változik adott nyelvi konstrukciók eredményeképp, a tisztán funkcionális nyelvekben nincs implicit állapot, a függvények mellékhatás-mentesek (*side-effect free*). Minden függvényhívás tartalmaz minden szükséges argumentumot, ami az eredményhez szükséges (akár explicit állapotot is, ha szükséges), és az azonos paraméterekkel rendelkező függvényhívások mindig ugyanazt az eredményt adják (*referential transparency*). Egyszerű matematikai példával élve: az `f(x) = x + 1` függvény azonos x értékekre mindig azonos eredményt ad.

A széles körben elterjedt imperatív nyelvekben (Java, C) az állapot változtatására szolgáló, a vezérlést végző nyelvi elemek egymásutáni használatával lehet leírni a működés „hogyanját". Változók értékadása, feltételes elágazások, ciklusok adják a működés logikáját, mit mi után milyen feltételek teljesülése esetén kell végrehajtani. Például a Fibonacci-sor Java megvalósítása:

Listing 2.1. Fibonacci – Java

```
int a=0, b=1;
public static int fib(int n) {
  for(int i=0; i<n; i++) {
    int c = a;
    a = b;
    b = c + b;
  }
  return a;
}
```

A fenti egyszerű példában szerepel értékadás, feltétel vizsgálat és ciklus is. A végrehajtás során az `a` változó többször kap új értéket, a ciklus futásának végén tartalmazza az eredményt.

Az Erlang deklaratív nyelv, a „hogyan" helyett a „mit" írja le: a fejlesztő kifejezéseket ad meg mintaillesztésekkel – kijelentéseket tesz. Funkcionális nyelveknél a függvény alapvető nyelvi elem, amelynek segítségével a működés leírható. A Fibonacci példa Erlang változata:

Listing 2.2. Fibonacci – Erlang

```
fib(0) -> 0;
fib(1) -> 1;
fib(N) when N > 0 -> fib(N-1) + fib(N-2).
```

Ebben az esetben a fib függvény deklarációja három állítás (*function clause*), melyik argumentum esetén mit kell tenni. Az imperatív nyelvekben megszokott struktúrák helyett más megoldásokkal kell élni. Ciklus helyett rekurzív függvényhívásokkal, feltételes elágazás helyett a függvény deklarációba írt kifejezéssel (jelen esetben pozitív számokra értelmezett a függvény).

A funkcionális pogramnyelvek fontos ismérve az is, hogy a függvények teljes jogú elemei a nyelvnek (*first class functions*). Bárhol, ahol valamilyen érték szerepelhet, függvény is: listák, rekordok, adatszerkezetek elemeként. Függvény argumentuma illetve visszatérési értéke is lehet függvény (*higher order functions*). Például egy lista elemeit többféle szempont szeretnénk szűrni. Először a páros számokat, majd utána a páratlanokat:

```
Paros = fun(N) when N rem 2 == 0 -> true;
           (_)                   -> false.

Paratlan = fun(N) when N rem 2 == 1 -> true;
              (_)                   -> false.


Lista = [1, 2, 3, 4, 5, 6, 7, 8, 9].
ParosSzamok = lists:filter(Paros, Lista).
ParatlanSzamok = lists:filter(Paratlan, Lista).
```

A `lists:filter` függvény a megadott lista minden elemére lefuttatja a `Paros` vagy a `Paratlan` függvényt, és ha igaz értéket kap vissza, akkor bennehagyja a listában a vizsgált elemet. A `Paros` illetve `Paratlan` függvények változók (amik nem igazi változók, csak egyszer kaphatnak értéket), amelyek értéke egy-egy névtelen függvény (λ *function*).

Az Erlang nem tisztán funkcionális környezet, lévén ipari alkalmazásra készült, lehet írni olyan függvényeket, amelyek nem mellékhatás mentesek, például fájl- és adatbázis műveletek. A funkcionális programozás alapelvei azonban jelen vannak, lehet tisztán funkcionális alkalmazást is írni, és a fentebb ismertetett elvek maradéktalanul alkalmazhatók. Nem léteznek globális változók, amelyek értékét több függvényből is módosítani lehetne, minden változó (pontosabban értékadás egy névhez) csak az adott függvényen belül értelmezett, a szükséges állapotot a függvény argumentumai közt explicit kell megadni, és a visszatérési értékbe is betenni. A párhuzamos programozást ez nagyban megkönnyíti: nem kell attól félni, hogy egy értéket egy másik folyamat vagy függvényhívás felülír, mindig minden „kézben van”. Hogyan lehet így bármi használhatót írni? Hogyan lehet egy komplex rendszert felépíteni, amelynek egyes moduljai inputokat

várnak más moduloktól?

Az egyes folyamatok (*process*) üzeneteket küldenek egymásnak, amelyek egy-egy másolatát (nem csak egy referenciát!) kapja meg a címzett fél, az üzenetváltás után a küldő és a fogadó fél is birtokában van az adatnak, nincs megosztott állapot (*shared state*), még véletlenül sem fordulhat elő, hogy egyik folyamat módosít valamit, amire egy másik folyamat is épít (kivéve, ha *side-effect*-eket használ a program – például adatbázist).

Az Erlang architektúra alapja a fentebb leírt üzenet alapú modell: az aktor modell.

2.2. Aktor modell

Az aktor modell létrehozását több száz, több ezer mikroprocesszorból álló rendszerek készítése motiválta. 1973-ban írta le először Carl Hewitt. Az aktor modell alapeleme az *aktor*, azaz a cselekvő, amely önálló entitás, saját memóriával, viselkedési mintával (*behaviour*), üzenetküldési és fogadási képességgel. Az egyes aktorok közötti kommunikációs csatorna biztosítja az összeköttetést. Akárcsak a valós világban: a cselekvők cselekszenek valamilyen viselkedési minta szerint, egymással üzenetet váltanak, észlelik a másik cselekvő halálát, diszfunkcionalitását. A világ konkurens, az aktorok egymással egy időben működnek, kommunikálnak, megfelelő protokoll szerint megértik egymás üzeneteit, esetleg – megállapodás alapján – figyelnek egymásra. Ha egyikkel történik valami, a másik észreveszi, és ha tud, csinál valami hasznosat.

Például egy telefonbeszélgetés során két aktor üzeneteket vált egymással (hangcsatornán keresztül):

– Jó napot! Béla vagyok.
– Jó napot! Mondja a számot.
– 42.
– Köszönöm.

Vagy:

– Jó napot! Az adóhivataltól vagyok.
– Sajnálom, ez biztos téves.

A fenti kis párbeszédekben a résztvevők értelmezni tudják a bejövő üzenetet, és annak megfelelően adnak választ. Ugyanígy a számítógépes aktorok (Erlangban: *process*) is képesek:

- üzeneteket küldeni;

- üzeneteket fogadni;
- a beérkező üzeneteket minta alapján szűrni, és azokra adekvát választ adni;
- meghatározni azt a viselkedés mintát, amellyel a beérkező üzeneteket kezelni fogja;
- további aktorokat létrehozni;
- észlelni a megfigyelt aktorok leállását.

A kommunikáció nagyon fontos eleme az aktor modellnek, az üzenetküldés elkülöníti a kommunikáció megoldását az aktortól, lehetővé téve az aszinkron üzenetküldést (*asynchron message passing*). Az Erlang architektúra ezen a modellen alapszik, nagyon gyorsan és hatékonyan lehet *process*-eket nagyon nagy számban indítani és futtatni; minden *process* saját memóriával, levelesládával (*mailbox*) rendelkezik, egymással aszinkron módon tudnak kommunikálni. A processzek azonosítóval rendelkeznek, ez a cím, ahova küldeni lehet az üzenetet, és – ami nagyon fontos a redundáns rendszerek építésénél – az üzenetküldés két külön gépen lévő process között teljesen transzparens, a programozónak nem kell külön erőfeszítést tennie, ha másik gépen lévő process-szel akar kommunikálni. A szintaxis ugyanaz helyben mint gépek között.

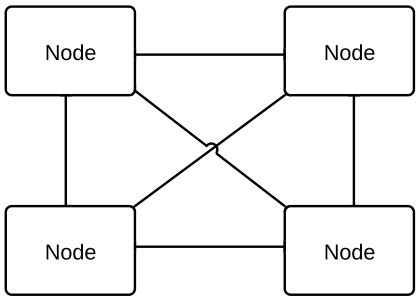
A párhuzamos programozás lehetőségét ez az aktor modell nyújtja az Erlang platformon. Nincs közös állapot (*shared state*), az egyes folyamatok aszinkron üzenetküldéssel kommunikálnak egymással, az üzenetek másolata kerül a címzett birtokába, a feladó „eredeti példánya" megmarad. A valós életben sem törlődik az agyunkból semmi, csak mert elmondjuk másnak (kivéve esetleg az egyetemi vizsgákat). Ez a másolás teszi lehetővé a gépek közti transzparens kommunikációt.

2.3. Elosztott (*distributed*) Erlang

Egy gép nem gép, ha magas rendelkezésre állást kell biztosítani. Hiába van remekül megírva a szoftver, ha a hardver meghibásodik alatta vagy kimegy az áram. Egy gép nem gép akkor sem, ha a feladat olyan időigényes, vagy olyan nagy adatigénye van, hogy egyetlen mai számítógép sem elegendő hozzá önmagában. A feladatot ilyenkor partícionálni kell, és az egyes részfeladatokat egymással párhuzamosan kell elvégezni. Több gépre kell szoftvert írni, és ez az Erlang erőssége. A fentebb röviden ismertetett funkcionális programozás elve és az aktor modell egyszerűbbé, átláthatóbbá teszi a programok szerkezetét, magas szintű nyelven lehet elosztott rendszert fejleszteni, az egyes Erlang egységek (*node*) közti kommunikáció mikéntjéről mit sem kell tudni (az OSI hálózati hierarchia applikációs szintjén biztosított protokoll és a nyelvbe épített

egyszerű üzenetküldési, fogadási szintaxis biztosítja a teljes transzparenciát a programozó számára).

Az Erlang hálózat alapegysége a *node*. A node egy teljesen önálló Erlang rendszer, saját névvel, virtuális géppel (*virtual machine, VM*), külön operációs rendszer folyamatban. Egy gépen egy vagy több node is futtatható, egymáshoz való viszonyuk nem különbözik attól, mintha külön gépen futnának. Ahhoz, hogy a node-ok egy Erlang hálózatot alkossanak, egy közös *cookie*-val kell rendelkezniük, az Erlang VM-et ezt az értéket megadva kell elindítani. Az Erlang hálózatban minden node tud minden noderól, közvetlen kapcsolatban vannak egymással. Ez a hálózati felépítés korlátozza a node-ok számát egy hálózaton belül, bizonyos szám felett már túl nagy terhelést jelent a node-ok egymás közti adminisztrációs kommunikációja. Szerencsére node-ot lehet rejtett üzemmódban is indítani, így nem jelenik meg a többi node listáján.



Erlang hálózatok belső, biztonságos környezetben használatosak, a klaszteren (*cluster*) kívüli világgal, az interneten keresztül, valamelyik IP alapú protokollal – TCP, UDP, SCTP – lehet kommunikálni. Egy webszolgáltatás esetében például a belső hálózat Erlang node-okból áll, a külvilág – a böngésző – pedig HTTP protokollon keresztül éri el a HTTP szervert futtató gépet.

Elosztott rendszereknél bizonyos méret fölött, különösen földrajzilag is diverzifikált rendszereknél a CAP-elméletnek megfelelően a következő háromból valamelyiket fel kell adni:

Consistency

A rendszer által tárolt adatok mindig, minden pillanatban konzisztensek, a referenciák érvényesek és minden felhasználó ugyanazt látja. Ha egy darab könyv van raktáron, és azt valaki megveszi, akkor egy másik vásárló ugyanezt a könyvet már nem tudja megvenni, azt fogja látni, hogy elfogyott.

Availability

A kliens mindig kap választ, a rendszer mindig elérhető. A válasz lehet sikeres vagy sikertelen végrehajtás eredménye is, a lényeg, hogy a kérést intéző fél mindig kap információt az eredményről.

Partition Tolerance

Csak a teljes rendszer hibás működése esetén fordulhat elő, hogy a kliens rossz választ kapjon, vagy egyáltalán ne kapjon választ.

A fenti korlátozás a rendszer növekedésével párhuzamosan válik szembetűnővé. Amíg egy gép van (sok-sok redundáns diszkkel), addig minden konzisztens, viszont a legegyszerűbb hardver hiba is elég, hogy ne legyen elérhető. Nő a forgalom, a működtető nem akar kiesést, egyre több gépet állít csatasorba, az adatok nem férnek már el semmilyen számítógépen, elosztott architektúrára van szükség. Ha az is meg van, akkor felmerül, hogy ha tűz üt ki az épületben, akkor nem lesz elérhető semmi. Földrajzilag is el kell különíteni az egyes részeket: ekkor az a probléma, hogy meg kell oldani a konzisztens adatmódosítást úgy, hogy közben az elérhetőség (elfogadható válaszidő) ne sérüljön. A CAP-elmélet ezt állítja, valamelyik feltétel a háromból mindenképpen sérül.

Az ebben a dolgozatban a .. fejezetben bemutatott alkalmazás esetében az elérhetőségről (*availability*) mondunk le. A rendszer egy Erlang node-okból álló klasztert alkot egy biztonságos helyi hálózaton belül, földrajzilag nincs elkülönítve egyetlen része sem (ez az irány – nevezetesen a konzisztencia feladása – egy későbbi dolgozat tárgya lehet). A rendszer állapota mindig konzisztens, és a rendszer mindaddig válaszképes, amíg akár egyetlen node is üzemel.

A dolgozat célja nem egy Google vagy Amazon-méretű cég technológiai hátterének elemzése, hanem az Erlang/OTP platform bemutatása, ezért ezt a feltételt dobjuk el. Egy így felépíthető rendszer kapacitása, képességei is messze meghaladják egy átlagos magyar vállalkozás vagy akár egyetemi tanulmányi rendszer követelményeit. Biztosítani képes a folyamatos működést és az elfogadható válaszidőt (*soft-realtime*).

2.4. Az Erlang programozási nyelv alapjai

Az ebben a részben bemutatásra kerülő Erlang szintaxis nem fedi le a teljes nyelvet, a cél, hogy a következőkben bemutatott alkalmazás felépítése, a modulok érthetőek legyenek.

Minden Erlang folyamathoz tartozik egy *shell*, amely egy REPL (*Read Eval Process Loop*) alkalmazás a futó környezethez. Ennek segítségével egy egyszerű parancssoros felületen ki lehet próbálni parancsokat, műveleteket, sőt – mint később még látni fogjuk – „éles” (*production*) környzeteknél is hozzáférhetünk a rendszerhez (például hibakeresés céljából).

Az Erlang shell indítása után egy *prompt*-ot ad, a sor eleji szám a beírt kifejezések ('.'-tal bezárólag) sorszámát jelenti. Induláskor:

```
\$ erl -sname a
Erlang R13B03 (erts-5.7.4) [source] [smp:2:2] ...

Eshell V5.7.4  (abort with ^G)
(a@notebook)1> 1 + 2 + 3.
6
(a@notebook)2>
```

2.4.1. Típusok

Az Erlang dinamikus típusos nyelv (*dynamic typed programming language*). A statikus típusos nyelvekkel (pl. Java) szemben a típusellenőrzés futásidőben (*runtime*) történik, nem fordításkor, nem a változóknak van típusa, hanem az értékeknek. Például egy függvény argumentumánál nem kell deklarálni, hogy milyen típusú paramétert fog kapni, kaphat bármilyet, az futásidőben derül ki, hogy mit kapott, és azt tudja-e kezelni.

Funkcionális nyelv lévén a változó nem igazi változó, csak egyszer kaphat értéket, utána nem lehet módosítani. Az egyszerűség kedvéért használjuk az angol *variable* szó magyar megfelelőjét. Az Erlangban a változó nevét mindig nagy betűvel kell kezdeni.

Egy függvény definíció példa. Egy téglalap területét számolja ki a megadott oldalaiból.

```
terulet(A, B) ->
  A * B.
```

A függvényen belül sem az `A` sem a `B` változóhoz nem lehet új értéket rendelni. Amennyiben ciklusra lenne szükségünk, rekurzióval kell megoldani.

Integer. Egész számok: `100, 3.1415, -5`. Méretét a memória mérete határozza meg, nincs felső korlát.

Float. Lebegőpontos számok az IEEE 754 64-bites ábrázolásnak megfelelően.

Binary. Byte sorozat.

Atom. Kis betűvel kezdődő alfanumerikus karaktersorozat. Például: `ok, error, true, false`. Nem *string*!

Fun. Függvény típus. A függvények a többi típushoz hasonlóan használhatók változók értékeként. `fun(N) -> N * N end.`

Reference. Egyedi azonosító, amely összehasonlítás céljából állítható elő az Erlang `make_ref()` függvényével.

Pid. Process azonosító.

Port. Az Erlangon kívüli kommunikáció `port`-okon keresztül zajlik. Az Erlang `open_port()` függvényével hozható létre például C-ben megírt könyvtárak használatához.

List. Lista. Eleme bármilyen változó lehet. `[1,2,3,4,5,6]`. Az üres lista jelölése: `[]`.

Tuple. Több változó egy egységben való kezelésére szolgál. Az egyes elemeknek nincs nevük, nem lehet közvetlenül hivatkozni rájuk. Egy *tuple* bármilyen más változót tartalmazhat, újabb *tuple*-t is. A szintaxis: `{teglalap, {2, 3}}`. Ez esetben a tuple eleme egy atom és egy tuple, amely két egész számot tartalmaz.

Record. A rekord szintaktikai egyszerűsítés tuple-ök használatához. Egy rekord egy olyan tuple, amelynek az első eleme a rekord nevét tartalmazó atom. A rekord mezőinek neve van, amelyekkel lehet hivatkozni az egyes elemekre. Futás közben semmiben sem különböznek a tuple-től, a programozás megkönnyítésére szolgál. Például:

```
-record(sikidom, {nev, a_oldal, b_oldal}).

terulet(S) ->
  S#sikidom.a_oldal * S#sikidom.b_oldal.
```

String. Szintaktikai egyszerűsítés egész számok listájához. A "kakukk" kifejezés ekvivalens a `[107,97,107,117,107,107]` és `[$k,$a,$k,$u,$k,$k]` listákkal.

2.4.2. Mintaillesztés (*pattern matching*)

A mintaillesztés nagyon fontos fegyver az Erlang arzenálban. Mintaillesztéssel lehet egy változónak értéket adni, komplex adatszerkezetből adatot kinyerni, a vezérlést irányítani. A mintaillesztés általános formája:

```
Pattern = Expression
```

A baloldali minta (*pattern*) tartalmazhat bármilyen adattípust, változót, a jobboldalon azonban csak olyan változó szerepelhet, amelynek van értéke. A kifejezés ezen kívül még tartalmazhat műveletet, függvényhívást is. A mintában szereplő, még értékkel nem rendelkező változó a jobboldali kifejezés alapján kap értéket. Amennyiben a baloldalon álló változónak már van értéke, a mintaillesztés csak akkor lesz sikeres, ha a jobboldali kifejezésben, a megfelelő helyen, ugyanaz az érték szerepel.

A mintaillesztésnek két eredménye van: vagy sikeres vagy nem. Néhány példa az egyszerű értékadástól, a komplexebb szerkezetig.

```
1> A = A + 2.
* 1: variable 'A' is unbound
2> A = 2.
2
3> A = A + 2.
** exception error: no match of right hand side value 4
4> B = A + 2.
4
5> B.
4
6> B = A + A.
4
7> B = 2 * A.
4
8> C = A + B.
6
9> C.
6
```

Az 1. utasítás nem értelmezhető, mert az `A` változónak nincs értéke (*unbound*), és a jobb oldalon nem szerepelhet így (nincs is értelme). A 3. utasításnál az `A` változónak van értéke (2), de a mintaillesztés nem sikeres, mert a bal oldalon így 2 szerepel, a jobb oldalon meg 4.

Listáknál a mintaillesztésnél használható az alábbi szintaxis a lista első elemének (*head*) kinyerésére.

```
[Head|Tail] = [1,2,3,4,5]
```

Ebben az esetben a `Head` változóban lesz a lista első eleme (1), a `Tail` változóban a többi elem listája (`[2,3,4,5]`). Néhány példa listás mintaillesztésre.

```
1> List = [1,2,3,4,3,2,1].
[1,2,3,4,3,2,1]
2> [Head|Tail] = List.
[1,2,3,4,3,2,1]
3> Head.
1
4> Tail.
[2,3,4,3,2,1]
5> [H1,H2|T] = List.
[1,2,3,4,3,2,1]
6> H2.
2
7> T.
[3,4,3,2,1]
8> A = 3.
3
9> [1,2,E,4,E,2,1] = List.
[1,2,3,4,3,2,1]
10> E.
3
11> [1,2,F,4,3,F,1] = List.
** exception error: no match of right hand
side value [1,2,3,4,3,2,1]
```

A 9. utasítás sikeres, mert az `E` változónak nincs értéke, és a jobb oldalon lévő lista 3. és 5. eleme megegyezik. A mintaillesztés sikeres, és az `E` változó a 3-as értéket kapja. A 11. utasításnál azért nem sikerül az illesztés, mert az `F` változó két olyan helyen szerepel a listában, ahol különböző érték áll a jobboldali kifejezésben.
Hasonlóképpen a *tuple*-öknél.

```
1> C = {degree, celsius, 37}.
{degree,celsius,37}
2> F = {degree, fahrenheit, 99}.
{degree,fahrenheit,99}
3>
3> {degree, Type, Value} = C.
{degree,celsius,37}
4> {Type, Value}.
{celsius,37}
```

A rekordok valójában tuple-ök, amelyeknek az első eleme a rekord neve, így a rekordokat is lehet használni a mintaillesztésekben. A fenti példa rekordra átírva.

```
1> rd(degree, { type, value}).
degree
2> T = #degree{ type=celsius, value=37 }.
#degree{type = celsius,value = 37}
3> #degree{type=celsius} = T.
#degree{type = celsius,value = 37}
4> #degree{type=Type} = T.
#degree{type = celsius,value = 37}
5> Type.
celsius
6> {degree, celsius, Value} = T.
#degree{type = celsius,value = 37}
7> Value.
37
```

A rekord szintaxis több mezőnél válik hasznossá, amikor csak egy-egy mező értékét szeretnénk kinyerni (és a rekord szintaxisú mintaillesztések akkor is működni fognak, ha a mezőszám megváltozik, míg tuple-nél az összes helyen módosítani kellene a kódot, ahol mintaillesztésben szerepel). A 6. utasításnál látszik, hogy a tuple szintaxisú minta is megfelel a jobboldali rekordnak.

2.4.3. Függvény definiálása

A mintaillesztés a függvények definiálásánál is nagyon hasznos, a függvény argumentumainak mintát megadva lehet a komplexebb paraméterből változóknak értéket adni, illetve külön állításokat (*function clause*) megfogalmazni a függvényhez, milyen input esetén mit csináljon. Egy névvel és azonos számú argumentummal (*function arity*) egy függvényt lehet definiálni, de több állítást lehet tenni azon belül. Egy névvel, de eltérő számú argumentummal akármennyi különálló függvényt lehet definiálni. Az általános szintaxis jól látszik a következő egyszerű példán:

```
convert(celsius, T) ->
  9 * T / 5 + 32;
convert(fahrenheit, T) ->
  5 * (T - 32) / 9;
convert(_,_) ->
  unkown.

convert() ->
```

```
    nothing.

convert(A) ->
  A.
```

A fenti példa három függvény-definíciót tartalmaz. Az egyik a `convert` függvény két paraméterrel (Erlangban a jelölése: `convert/2`), paraméter nélkül (`convert/0`) és egy paraméterrel (`convert/1`). A két paraméteres függvény három *function clasue*-ból áll: a kapott paraméter alapján mintaillesztéssel dől el, hogy melyik rész fut le. Az '_' jel azt jelenti, hogy „bármi" – mindenre illeszkedik. Ha tehát a függvényhívás a következő: `convert(celsius, 36)`, akkor a legelső számítás eredményét adja vissza, mert a `celsius` atom megegyezik a mintával, a `T` változó pedig megkapja értéknek a 36-ot.

Anoním függvények definiálása hasonló módon történik, csak ott a név helyett a `fun` kulcsszó szerepel, az egyes *function clause*-oknál pedig csak a zárójelbe írt argumentum lista. Anoním függvényt bárhol definiálhatunk, ahol változót is írhatnánk.

```
Convert = fun(celsius, T) ->
                9 * T / 5 + 32;
            (fahrenheit, T) ->
                5 * (T - 32) / 9;
            (_,_) ->
                unkown
          end.
```

Ebben az esetben a `Convert` változó értéke lesz a függvény, és a következő módon lehet meghívni: `Convert(celsius, 36)`. Az anoním függvény definiálása az `end` kulcsszóval zárul.

Végezetül egy példa egy jellemző rekurzív függvény bemutatására. Az alábbi `insert` függvény egy elemet illeszt be egy rendezett listába.

```
insert(I, []) ->
  [I];
insert(I, [H|T]) when I > H ->
  [H | insert(I, T)];
insert(I, [H|T]) ->
  [I, H|T].
```

A `when` kulcsszó utáni feltétel neve: *guard*. Ezzel lehet tovább finomítani a mintában megfogalmazott feltételeket. Jelen esetben addig hívja meg újra és újra az insert függvényt (2. *function clause*), amíg a beillesztendő elem nagyobb az aktuális listaelemnél (folyamatosan a lista elejére téve a megvizsgált listelemet). Amikor ez a feltétel nem teljesül, akkor beilleszti a fennmaradó lista elejére az elemet, és nem vizsgálódik tovább (3. *function clause*). Az egyes lépések eredményei egy rövid kis listán.

```
1> list:insert(4, [1, 3, 4, 6]).
3. function clause [4,4,6]
2. function clause [3,4,4,6]
2. function clause [1,3,4,4,6]
[1,3,4,4,6]
2> list:insert(7, [1, 3, 4, 6]).
1. function clause [7]
2. function clause [6,7]
2. function clause [4,6,7]
2. function clause [3,4,6,7]
2. function clause [1,3,4,6,7]
[1,3,4,6,7]
```

A `list:insert` kifejezés jelentése: a `list` modulban lévő `insert` függvény.

2.4.4. Modulok

A modulok szolgálnak logikailag összetartozó függvények csoportosítására. A modulok nem alkotnak hierarchiát (*flat namespace*), elnevezéssel lehet megoldani a szükséges tagolást. A modulokban az alábbi részek szerepelhetnek:

- modul attribútumok, verziószám, szerző, név, interfész függvények deklarálása;
- rekord, makró definíciók, külső fájlok (*header*) beillesztése;
- függvények.

Egy egyszerű modul.

```
-module(temp).
-export([convert/1]).

convert({celsius, T}) when is_number(T) ->
  {fahrenheit, c2f(T)};
convert({fahrenheit, T}) when is_number(T) ->
  {celsius, f2c(T)};
```

```
convert(_) ->
  bad_parameter.

f2c(F) ->
  5 * (F - 32) / 9.
c2f(C) ->
  9 * C / 5 + 32.
```

A fenti modul az egy argumentummal rendelkező `convert` függvényt exportálja, csak ezt lehet meghívni más modulokból. A modul neve `temp`, kötelező ugyanilyen nevű fájlba tenni.

2.4.5. Konkurens programozás

A fentebb röviden bemutatott aktor modellen alapul a párhuzamos programozás az Erlangban. Az aktorok *process*-ek, amelyek tudnak üzenetet fogadni (*receive*) és küldeni (*send*). A process az Erlang VM fennhatósága alá tartozik, nem külön operációs rendszer folyamat, indítása nem igényel sok erőforrást, egy-egy rendszeren belül akár többszázezer process-t is lehet indítani. Például egy HTTP szerver megvalósítása lehet az, hogy minden kapcsolathoz tartozik egy önálló webszerver process. Ha ebben valami hiba történik (pl. 0-val osztás), akkor ez a process leáll, a többi működik tovább.

Process-ek programozásához az eddigieken bemutatottakon túl mindössze három műveletre van szükség:

Process indítása. Process-t a `spawn` függvénnyel lehet indítani helyben vagy akár egy másik node-on.

```
P = spawn(module, function, [P1, P2 ...]).
```

A process-ek egyedi azonosítót kapnak, amely a címük, ide lehet küldeni az üzenetet. Van lehetőség arra is, hogy a process nevet kapjon, ezt hosszabb ideig futó processeknél érdemes megtenni.

Üzenet küldése. Üzenetet küldeni a következők szerint lehet:

```
P ! "Hello".
```

ahol `P` a process azonosítója vagy regisztrált neve, a `!` a küldés operátor. A jobboldalon álló kifejezés az üzenet, amely nem csak *string* lehet, mint a fenti példában,

hanem bármilyen Erlang kifejezés (tuple, fun, lista, stb). Másik node-ra ugyanígy lehet üzenetet küldeni, nincs különbség.

Üzenet fogadása. A `receive` utasítás szolgál az üzenetek fogadására.

```
receive
  Pattern1 ->
        ...;
  Pattern2 ->
        ...;
  _ ->
        ...;
after Time ->
        ...
end.
```

A függvényekhez hasonlóan *clause*-okat tartalmaz, és itt is mintaillesztés alapján dől el, melyik ág fog lefutni. Minden process-hez tartozik egy *mailbox*, amelyből a receive utasítással lehet kiolvasni az üzeneteket. Az utasítás addig olvassa a mailbox-ot, amíg nem talál egy olyan üzenetet, amely illeszkedik valamelyik ág mintájára. Ekkor kiolvassa, és az üzenet törlődik a mailbox-ból. Ha egyik mintára sem illeszkednek a beérkező üzenetek, akkor előbb-utóbb megtelik a memória. Éppen ezért szokás a legtöbb esetben az utolsó ág mintáját '_'-ként megadni. Ez a minta mindenre illeszkedik, így végül minden üzenet törlődik a mailbox-ból. Az `after` utasítással meg lehet adni, hogy ha `Time` ideig nem érkezett olyan üzenet, ami valamelyik mintára illeszkedett, akkor mit hajtson végre a program (*timeout*).

Az egyes process-ek – azonkívül, hogy tudnak egymásnak üzeneteket küldeni – monitorozni is tudják egymást, nem múlt-e ki a másik. Ennek módja a process-ek összekapcsolása a `link` függvénnyel.

```
start() ->
  process_flag(trap_exit, true),
  Pid = spawn(module, function, []),
  link(Pid),
  receive
    {'EXIT', Pid, Reason} ->
      io:format("Process ~p has just died.~n", [Pid])
  end.
```

A fenti `start` függvény elindít egy másik process-t, és összekapcsolja a jelenlegivel. Amikor a másik process leáll, érkezik egy üzenetet erről, aminek a formátumát a fenti `receive` blokkban szereplő mintán lehet látni.

Ahhoz, hogy egy process folyamatosan fusson, újra és újra meg kell hívni a `receive` utasítást (hogy várakozzon üzenetre). Az alábbi modul egy process-t futtat, amíg egy `stop` atomot nem kap üzenetként. A `?MODULE` makró a modul nevét tartalmazza.

```
-module(temp).
-export([start/0, loop/0]).

start() ->
  spawn(?MODULE, loop, []).

loop() ->
  receive
    {From, celsius, T} ->
        From ! {fahrenheit, c2f(T)},
        loop();
    {From, fahrenheit, T} ->
        From ! {celsius, f2c(T)},
        loop();
    stop ->
      stopped;
    _ ->
      loop()
  end.

f2c(F) ->
  5 * (F - 32) / 9.

c2f(C) ->
  9 * C / 5 + 32.
```

A puding próbálja az evést – az Erlang shellben. A `self()` függvény az aktuális process azonosítóját adja vissza, jelen esetben az shell-ét; a `flush()` shell-utasítás pedig kilistázza és törli a shell mailbox-át.

```
1> c(temp).
{ok,temp}
2> P = temp:start().
<0.42.0>
3> P ! {self(), fahrenheit, 99.9}.
{<0.35.0>,fahrenheit,99.9}
```

```
4> flush().
Shell got {celsius,37.72222222222222}
ok
5> P ! stop.
stop
6> flush().
ok
7> P ! {self(), fahrenheit, 99.9}.
{<0.35.0>,fahrenheit,99.9}
8> flush().
ok
```

Az ebben a részben bemutatott téglákból építkezik az Erlang részét képező, a leggyakoribb feladattípusok megoldására létrehozott Open Telecom Platform (OTP) szoftverkönyvtár, amely központjában a magas rendelkezésre állású, párhuzamos rendszerek építése áll.

2.5. Open Telecom Platform (OTP)

Nagyobb szoftverek fejlesztésekor elkerülhetetlen, hogy a fejlesztők valamilyen közös implementációs gyakorlatot kövessenek. Enélkül minden egyes fejlesztő kitalálja magának a világ legjobb megoldását (és ebben rendkívül kreatívak tudnak lenni), amit rajtuk kívül senki más nem ért. Az OTP formalizált keretet ad a process-fejlesztéshez. Hasonlóan a Java EE *servlet*-eihez, egy interfészt definiál, amelyet minden Erlang fejlesztő ért, így a rendszer különböző emberek által fejlesztett részei egy koherens egészet alkotnak. Ezek mögött az interfészek mögött ráadásul általános problémák megoldásai állnak, nem kell újra és újra kitalálni például a hibakezelés módját.

Az OTP viselkedésmintákat definiál (*behaviour*), amelyekhez a megadott *callback* függvényeket (a Java interfész függvényeinek implementálásahoz lehet hasonlítani) implementálva egy process-modul hozható létre. Ezt a modult az OTP kezeli, és adott eseménynél meghívja a megfelelő függvényt. Az OTP-ben a process-ek hierarchiába rendezhetők, ennek megfelelően van háromféle típusú process.

Application. Az OTP application *behaviour* egy Erlang/OTP alkalmazás, amely összefogja a modulok egy csoportját, és a process hierarchia csúcsán áll.

Supervisor. Olyan *behaviour*, amelynek feladata más processek felügyelete, folyamatosan monitorozza azok állapotát, észleli, ha a processben hiba történt, és a megadott újraindítási stratégiának megfelelően újraindítja vagy hagyja leállni. Supervisor

processek alá tartozhatnak *worker* és további supervisor processek.

Worker. Olyan process, amely valamilyen alkalmazás-specifikus feladatot végez, nem felügyel más process-eket. Az OTP az alábbi *behaviour*-öket biztosítja:

- *Generic server (gen_server)*: általános szerver;
- *Generic Finit State Machine (gen_fsm)*: véges automata;
- *Generic Event Manager (gen_event)*: események, alarmok kezeléséhez.

A fenti celsius-fahrenheit átváltó modul *gen_server* átirata:

```
-module(temp_otp).
-behaviour(gen_server).

-export([start/0]).
-export([init/1, handle_call/3, handle_cast/2,
         handle_info/2, code_change/3, terminate/2]).

start() ->
  gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

init(_Options) ->
  {ok, 0}.

handle_cast({From, celsius, T}, State) ->
  From ! {fahrenheit, c2f(T)},
  {noreply, State + 1};
handle_cast({From, fahrenheit, T}, State) ->
  From ! {celsius, f2c(T)},
  {noreply, State + 1};
handle_cast(stop, State) ->
  io:format("Server handled ~p requests.~n", [State]),
  {stop, normal, State};
handle_cast(_Info, State) ->
  {noreply, State}.

handle_call({celsius, T}, _From, State) ->
  {reply, {fahrenheit, c2f(T)}, State + 1};
handle_call({fahrenheit, T}, _From, State) ->
  {reply, {celsius, f2c(T)}, State + 1}.

handle_info(_Info, State) ->
  {noreply, State}.

terminate(_Reason, _State) ->
  ok.

code_change(_OldVsn, State, _Extra) ->
  {ok, State}.

f2c(F) ->
  5 * (F - 32) / 9.
c2f(C) ->
  9 * C / 5 + 32.
```

A fenti modul egy szerver, amely kétféleképpen használható: szinkron (`handle_call` callback függvény) és aszinkron (`handle_cast`) módon. Indításkor a modul nevén regisztráljuk a process-t (`gen_server:start_link`), így process azonosító helyett erre a névre is lehet címezni az üzeneteket. A szervernek háromféleképpen lehet üzenetet küldeni: a `gen_server:cast()` és `gen_server:call()` függényekkel, és a standard módon a `!` operátorral (ezt az esetet a `handle_info()` függvénnyel lehet kezelni).

A függvények megkapják az init függvényben inicializált `State` paramétert (jelen esetben egy egész számot) minden egyes hívásnál. Ez a változó használható a szerver állapotának tárolására, a példában az elvégzett konverziók számát tartalmazza, amit leállításkor kiír. Próba a shell-ben:

```
1> c(temp_otp).
{ok,temp_otp}
2> temp_otp:start().
{ok,<0.42.0>}
3> gen_server:cast(temp_otp, {self(), celsius, 37}).
ok
4> flush().
Shell got {fahrenheit,98.6}
ok
5> {fahrenheit,F}=gen_server:call(temp_otp, {celsius, 37}).
{fahrenheit,98.6}
6> F.
98.6
7> gen_server:cast(temp_otp, stop).
Server handled 2 requests.
ok
```

A fenti példában a szerver nem része semmilyen process hierarchiának, nem felügyeli supervisor a működését. A kód bonyolultabbnak tűnhet a nem OTP-s megoldásnál, de ahogy nem egy ennyire egyszerű, teljesen egyedülálló szerver process-ről van szó, az OTP-s környezet hamar behozza az árát. Erre nézünk egy kicsit összetettebb, de azért még követhető példát a következő fejezetekben.

3. fejezet

Tőzsdei „ticker” alkalmazás

Állatorvosi lóként egy egyszerű üzenetküldésen alapuló webes szolgáltatást fogunk megvizsgálni, ezen mutatjuk be az eddigiekben leírtak alkalmazását egy kicsi, ám működőképes rendszer megalkotásában. A rendszer célja, hogy a tőzsde felöl érkező pillanatnyi kereskedési adatokat (melyik részvényből milyen áron hány darabot adtak el az adott tranzakció során) a weben keresztül eljuttasa a felhasználóknak. A webes felület lehet akár egy előfizetői oldal, ahol a tőzsdei adatok real-time jelennek meg, vagy lehet egy újság, ahol a szokásos 15 perces késleltetéssel lehet látni az árakat. A rendszerrel szemben támasztott elvárások: *soft-realtime* működés, magas rendelkezésre állás, belátható üzemeltetési költségek. A dolgozat terjedelmi korlátai és az átláthatóság miatt nem térünk ki minden egyes részletre, nem faragunk csicsás weboldalt, de a fenti elvárásokat kielégítő megoldásokat mind bemutatjuk.

A magas rendelkezésre állás követelményéből fakadóan egynél több gépre lesz szükség. Az egyszerűség és a terminológia kedvéért az alábbiakban Erlang node-ként hivatkozunk az egyes gépekre.

A tőzsdei kereskedési rendszer kívül esik a látókörünkün, az adatokat egyszerűen inputként kezeljük, melyek Erlang üzenet formájában érkeznek. A tőzsdei kereskedést egy egyszerű véletlenszám-generátorral dolgozó process szimulálja.

A feladat magját jelentő üzenetküldés-kezelő rendszert egy külön, általánosan használható szerverként implementáljuk, erre építjük a webes felületet nyújtó egyszerű kis webalkalmazást.

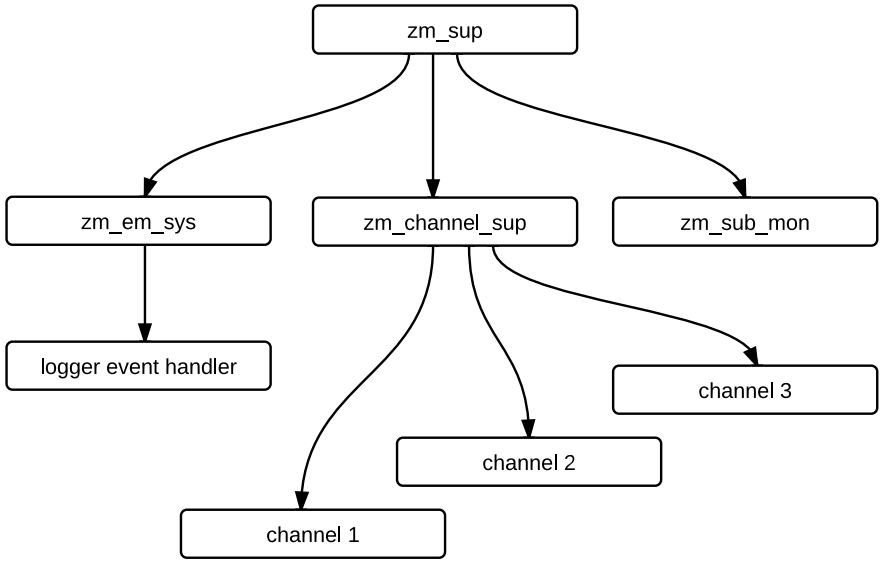
3.1. ZM – üzenetküldés, fogadás

A ZM fantázinevű alkalmazás egy *publish-subscribe* rendszer, amely csatornákat (*channel*) definiál, amelyekre fel lehet iratkozni (*subscribe*), és amelyekre üzenetet lehet küldeni (*publish*). A ZM szerver biztosítja, hogy az egyes csatornákhoz küldött üzenetet

megkapják az előfizetők („akik” maguk is process-ek) függetlenül attól, hogy melyik gépen küldték az üzenetet, és kezeli a csatornák működésében előforduló hibákat. A ZM keretet ad alkalmazásoknak, amelybe be lehet illeszteni az adott alkalmazás saját csatorna-implementációit (*gen_server*-ként). Így az üzenetküldés szolgáltatás egyszerűen beilleszthető például egy webes szolgáltatásba, használatához csak a csatornákat kell implementálni.

3.2. Felépítés

A ZM szerver felépítése az OTP alapelveknek felel meg, az egyes feladatokat önálló processek végzik, a processek egy hierarchiát alkotnak, ahol supervisor-ok felügyelik az alattuk futó folyamatokat. Az alábbi ábrán néven nevezzük az egyes elemeket.



zm_sup. A gyökere a process-fának. Ez a supervisor kezeli az alábbi három folyamatot. Újraindítási stratégiája: *one_for_one*, ami azt jelenti, hogy ha valamelyik process leáll, akkor azt az egyet indítja újra, a többivel nem foglalkozik.

zm_em_sys. *Event manager*, amely eseményeket fogad, és azokat a hozzácsatolt *event handler*-eknek továbbítja. A rendszer működése során a számottevő eseményekről értesíti (*notify*) ezt a process-t, és például egy hozzácsatolt naplózó (*logger*) handler egy fájlban rögzíti a tényt. A ZM nem tartalmaz *handler*-t, azt az alkalmazásra bízza.

zm_channel_sup. Supervisor, amely a csatorna process-eket felügyeli. Minden elindított csatorna ez alá a folyamat alá kerül, a `zm_channel_sup` indítja, állítja le, monitorozza őket.

zm_sub_mon. A feliratkozásokat kezeli, és monitorozza. Ha egy előfizető folyamat (*subscriber*) véget ér (rendben vagy rendellenesen), akkor ez a process gondoskodik arról, hogy a csatornák ne küldjenek erre a címre több üzenetet.

A fenti négy process alkotja a rendszert, ehhez lehet különböző csatornákat írni. A ZM szerverhez tartozik két egyszerű csatorna-implementáció. Az egyik egyszerűen csak továbbítja az előfizetők felé az üzeneteket, az előfizető a felirakozáskor megadhat egy szűrő függvényt (*filter function*), amelyet minden üzenetre lefuttat a csatorna, és csak akkor továbbítja, ha a függvény visszatérési értéke `true`. A másik egy késleltetett csatorna (*delayed*), amely a beérkező üzenetet a feliratkozáskor megadott időtartam után küldi csak tovább. Ezzel lehet például biztosítani a tőzsdei adatoknál szükséges 15 perces késleltetést (előfizetéstől függően ez lehet real-time adatközlés is).

A csatornák megvalósítása egy-egy *gen_server* modul. A behaviour-höz tartozó callback függvények implementálása után az alábbi paranccsal indítható.

```
zm_channel:start(zm_ch_delayed, stock_exchange, []).
```

Ebben az esetben a `zm_ch_delayed` modulban implementált csatorna indul el `stock_exchange` néven. A `zm_ch_delayed` modul `handle_cast` függvénye (aszinkron a csatornáknak való üzenetküldés).

```
handle_cast({msg, {_Timestamp, _} = Message},
            #state{ id=ChannelId } = State) ->
  Send = fun
    ({SubPid, SubscriptionRef, []}, _) ->
      SubPid ! {msg, {ChannelId, SubscriptionRef}, Message},
      sent;
    ({SubPid, SubscriptionRef, [{delay, Delay}]}, _) ->
      erlang:send_after(timer:minutes(Delay),
                        SubPid,
                        {msg, {ChannelId, SubscriptionRef},
                        Message}),
      delayed;
    ({_SubPid, _SubscriptionRef, _}, _) ->
      invalid_option
  end,
  zm_sub_mon:fold_subs(ChannelId, {0, 0, 0}, Send),
  {noreply, State};
```

```
handle_cast(_Info, State) ->
  {noreply, State}.
```

Leszámítva a `gen_server` default callback függvényeit, a fenti `handle_cast` jelenti a tőzsdei adatok szétküldésére szolgáló csatornát. A `Send` változóhoz rendelt függvényt a `zm_sub_mon:fold_subs()` hívás minden egyes előfizetőre végrehajtja. A feliratkozásnál egy opciót lehet megadni: azt, hogy hány perces késleltetéssel kapja meg az előfizető az adatokat. Egy példa a shell-ben többet mond ezer szónál.

```
(zm@notebook)1> zm_channel:start(zm_ch_delayed,
                                 stock_exchange, []).
{[{ok,started}],[]}
(zm@notebook)2> zm_channel:subscribe(stock_exchange,
                                     self(), []).
#Ref<0.0.0.443>
(zm@notebook)3> zm_channel:send(stock_exchange,
                    {msg, {erlang:now(), "SHARE_1 15"}}),
(zm@notebook)3> receive
(zm@notebook)3> M -> M
(zm@notebook)3> end.
{msg,{stock_exchange,#Ref<0.0.0.443>},
     {{1335,128575,402055},"SHARE_1 15"}}
```

A fenti *subscribe* eredménye egy *real-time* előfizetés, a beérkező üzenetet azonnal kiküldi a csatorna. A késleltetett előfizetéshez a `subscribe` függvényben meg kell adni a késleltetés időtartamát.

```
zm_channel:subscribe(stock_exchange, self(), [{delay, 15}]).
```

Így az üzeneteket 15 perccel érkezés után továbbítja a `stock_exchange` csatorna. De mi történik, ha hiba van? Hol a hibakezelés a fenti `handle_cast` függvényből?

3.3. Hibatűrés

Az Erlang fejlesztési gyakorlatra nem jellemző a defenzív – hibavizsgálattól függő elágazások a kódban – programozás. Az ajánlott módszer az, hogy a függvényekben feltételezzük, hogy az input adatok helyesek, és nem írjuk körül a lehetőségeket, ha valami mégsem lenne megfelelő. Egyetlen kivétel a felhasználó által megadott adatok

bekérésére szolgáló függvények, ott elkerülhetetlen a vizsgálódás, hiszen ki kell tudni írni, mi a baj a megadott adatokkal.

A hibákat a kódban nem kell kezelni, egyszerűen hagyni kell a process-t elbukni, és a supervisor-ra bízni a döntést, újraindítja-e a process-t vagy sem. A hiba úgy derül ki függvényhívásnál, hogy a várható értéket adjuk meg baloldali mintaként. Amennyiben a függvény nem a várt értéket adja vissza, a minta nem illeszkedik, és az hiba – a process leáll (*crash*). A supervisor ezt észreveszi, meghívja a gen_server `terminate` függvényét, majd újraindítja a process-t. Ha adott időn belül túl sokszor fordulna elő hiba, akkor nem próbálkozik újra.

A hibakezelést az alábbi függvénnyel teszteljük. A csatornának üzenetként küldött `crash` atom nullával való osztást eredményez, amitől leáll a folyamat.

```
...
handle_info(crash, State) ->
  1 / 0,
  {noreply, State};
...
terminate(Reason, _S) ->
  io:format("Channel terminated. Reason: ~p~n", [Reason]),
  terminate.
```

A supervisor azonnal újraindítja a csatornát, és meghívja a modul `terminate` függvényét (ahol most csak egy konzol kiírás van, hogy látható legyen, valóban ez történik). Lássuk a shell-ben.

A csatorna indítása:

```
(zm@notebook)1> zm_channel:start(zm_ch_delayed,
                                 stock_exchange, []).
...
{[{ok,started}],[]}
(zm@notebook)2> zm_channel:list().
[stock_exchange]
```

A shell process feliratkozása a csatornára, majd egy üzenet küldés-fogadás – működik a csatorna:

```
(zm@notebook)3> zm_channel:subscribe(stock_exchange,
                                     self(), []).
#Ref<0.0.0.498>
(zm@notebook)4> zm_channel:send(stock_exchange,
```

```
                    {msg, {erlang:now(), "Kakukk"}}).
abcast
(zm@notebook)5> flush().
Shell got {msg,{stock_exchange,#Ref<0.0.0.498>},
                {{1335,179193,696571},"Kakukk"}}
ok
```

Crash előidézése. Az üzenetküldés utána azonnal jön a jelentés, hogy a `stock_exchange` csatornában hiba történt (jelen esetben: `badarith` – aritmetikai). A `Channel terminated` kezdetű kiírást a fentebb bemutatott `terminate` függvénynek köszönhetjük.

```
(zm@notebook)6> stock_exchange ! crash.
crash
Channel terminated. Reason: {badarith,
    [{zm_ch_delayed,handle_info,2},
    {gen_server,handle_msg,5},
    {proc_lib,init_p_do_apply,3}]}
(zm@notebook)7>
=ERROR REPORT==== 23-Apr-2012::13:07:02 ===
** Generic server stock_exchange terminating
** Last message in was crash
** When Server state == {state,stock_exchange}
** Reason for termination ==
** {badarith,[{zm_ch_delayed,handle_info,2},
              {gen_server,handle_msg,5},
              {proc_lib,init_p_do_apply,3}]}

=CRASH REPORT==== 23-Apr-2012::13:07:02 ===
 crasher:
  initial call: zm_ch_delayed:init/1
  pid: <0.121.0>
  registered_name: stock_exchange
  exception exit:{badarith,[{zm_ch_delayed,handle_info,2},
                            {gen_server,handle_msg,5},
                            {proc_lib,init_p_do_apply,3}]}
  in function  gen_server:terminate/6
  ancestors: [zm_channel_sup,<0.102.0>,<0.86.0>]
  messages: []
  links: [<0.104.0>]
  dictionary: []
  trap_exit: false
  status: running
```

```
  heap_size: 377
  stack_size: 24
  reductions: 584
neighbours:
```

A supervisor is küldi a jelentést arról, hogy észlelte a hibát, és arról is, hogy újraindította a csatornát.

```
=SUPERVISOR REPORT==== 23-Apr-2012::13:07:02 ===
     Supervisor: {local,zm_channel_sup}
     Context:    child_terminated
     Reason:     {badarith,[{zm_ch_delayed,handle_info,2},
                            {gen_server,handle_msg,5},
                            {proc_lib,init_p_do_apply,3}]}
     Offender:   [{pid,<0.121.0>},
                  {name,stock_exchange},
                  {mfa,{zm_ch_delayed,start_link,
                        [stock_exchange,[]]}},
                  {restart_type,transient},
                  {shutdown,5000},
                  {child_type,worker}]


=PROGRESS REPORT==== 23-Apr-2012::13:07:02 ===
          supervisor: {local,zm_channel_sup}
             started: [{pid,<0.136.0>},
                       {name,stock_exchange},
                       {mfa,{zm_ch_delayed,start_link,
                             [stock_exchange,[]]}},
                       {restart_type,transient},
                       {shutdown,5000},
                       {child_type,worker}]
```

Ezután megint működik minden, mint a hiba előtt.

```
(zm@notebook)8> zm_channel:list().
[stock_exchange]
(zm@notebook)9> zm_channel:send(stock_exchange,
                    {msg, {erlang:now(), "Kakukk again"}}).
abcast
(zm@notebook)10> flush().
Shell got {msg,{stock_exchange,#Ref<0.0.0.498>},
               {{1335,179242,712806},"Kakukk again"}}
```

```
ok
```

Ez a mechanizmus teszi lehetővé, hogy a modulokban ne kelljen a hibakezelést mindenhol leprogramozni. Ennek eredményeképpen a kód átláthatóbb marad, és kevesebb sorból is áll. Kevesebb kód, kevesebb hibalehetőség.

3.4. Hibakeresés

Hiba, nem dokumentált kódrész azonban mindig marad. Többféle lehetőség közül választhat a fejlesztő, amikor hibát keres:

Debug. A kód futtatása közben vizsgálhatja a fejlesztő a program-változókat. Hátránya, hogy „éles” környezetben nem igazán alkalmazható, egy webszolgáltatást nem lehet fejlesztői környezetből futtatni.

Logging. Minden rendszer elengedhetetlen része a különböző részletezettségű naplózás. A naplófájlokban rögzíti a rendszer azokat az eseményeket, amelyek vagy audit vagy hibakeresések miatt érdekesek. Fentebb már volt szó a ZM program `zm_em_sys` *event_manager*-eréről, amely rendszer-eseményeket fogad. Az alkalmazásíró hozzáadhat akármennyi *event_handler*-t, amelyekkel például naplózhatja az eseményeket.

Trace. : az Erlang lehetőséget ad arra, hogy egy működő node működését, egészen pontosan függvényhívásait kívülről meg lehessen vizsgálni. Meg lehet adni, hogy melyik modul melyik függvényére vagyunk kíváncsiak, és meg lehet adni a paraméterek mintáját is, hogy csak azokat a hívásokat kapjuk meg, amik érdekelnek. Elkerülendő a több gigabájtos, áttekinthetetlen eredményt.

A trace nagy előnye, hogy „éles” környezetben is használható, a szoftver megfigyelhető futás közben. Ez különösen akkor hasznos, amikor nem triviális vagy reprodukálható hibáról van szó, és nyomozni kell még azt is, hogy egyáltalán hol fordulhat elő hiba. A funkcionális programozás lehetővé teszi, hogy egy-egy függvény működését teljes egészében látni lehessen, hiszen minden inputot tartalmaznak a függvény paraméterei, nincs egyéb globális változó, aminek értéke befolyással lehetne a függvény működésére.

Az Erlang beépített *trace* moduljai: a `dbg` és a `ttb`. Előbbi tartalmazza a trace-hez szükséges függvényeket, utóbbi egy erre épülő eszköz elosztott (*distributed*) rendszerek nyomozásához. A `ttb` segítségével egyszerűen lehet több node-ot figyelni, és az eredményül kapott log-okat egy gépen, időrendi sorrendben olvasni.

A fenti példában szereplő, `handle_info` függvény az alábbi módon trace-elhető (a példa együgyű, de látni, amit kell).

```
(zm@notebook)21> stock_exchange ! {self(), ping}.
{<0.214.0>,ping}
(<0.149.0>) call zm_ch_delayed:handle_info(
                 {<0.214.0>,ping},{state,stock_exchange})
(<0.149.0>) returned from zm_ch_delayed:handle_info/2 ->
                 {noreply, {state, stock_exchange}}
```

Két sort ír ki a `dbg`. Az első a *call*, hogy a függvény milyen paraméterrel lett meghívva, a második a *returned*, amelyben az szerepel, milyen értéket adott vissza a függvény. A kiírást természetesen lehet fájlba is kérni, sőt grafikus felülettel rendelkező eszköz is van, amellyel nyomon lehet követni a függvényhívások, üzenetküldések láncolatát. További részletekre itt helyhiány miatt nem térünk ki.

3.5. Hibajavítás

Mint láttuk, a *trace* intézménye azért nagyon fontos, mert az Erlang rendszerek egyes számú ismérve a folyamatos működés, az hibakeresést meg kell tudni oldani leállás nélkül – az üzleti környezetben. Ehhez hasonlóan, ha megvan a hiba helye, a javítást is olymódon kell tudni eljuttatni a célrendszerre, hogy a folyamatos működés elve ne sérüljön. Az Erlang rendszereken az egyes modulokat futás közben is be lehet tölteni, a következő függvényhívás már az új kódot fogja futtatni. Javítsuk ki a fenti *crash* hibát a `zm_ch_delayed` modulban!

```
handle_info(crash, State) ->
  io:format("Won't crash.\n"),
  {noreply, State};
```

És a próba a shell-ben.

```
(zm@notebook)26> l(zm_ch_delayed).
{module,zm_ch_delayed}
(zm@notebook)27> stock_exchange ! crash.
Won't crash.
```

Az `l(zm_ch_delayed)` parancs betölti a javított modult, és a következő üzenetet már hibamentesen tudja kezelni.

Az Erlang ennél komplexebb futásközbeni szoftverfrissítést is tud kezelni, például futó szerverek állapotát lehet transzformálni új formátumba anélkül, hogy le kellene állni a folyamatnak. Ennek bemutatására itt helyhiány miatt nem térünk ki, annyit megjegyzünk, hogy az is a fenti mechanizmuson alapul.

3.6. Elosztott rendszer – egy gép nem elég

A fentiekben vázolt ZM szerver elosztott rendszerként való működését az alábbi kódrészletek biztosítják.

```
start(ChannelModule, ChannelId, Options) ->
  rpc:multicall(?MODULE, do_start,
                [ChannelModule, ChannelId, Options]).
...
stop(Channel) ->
  rpc:multicall(?MODULE, do_stop, [Channel]).
...
send(Channel, Message) ->
  gen_server:abcast(Channel, Message).
```

Ezen kívül minden megegyezik azzal, mintha csak egyetlen gépen futna a szerver (köszönhetően az aktor modellnek). Az `rpc:multicall` hívás minden egyes kapcsolódó node-on (`[node()|nodes()]` kifejezés adja meg a listát) lefuttatja a paraméterként megadott modul adott függvényét. A `gen_server:abcast` ugyanazt jelenti, mint a `cast`, csak minden node-ra elküldi az üzenetet.

A ZM működése tehát a következő. Minden node egyenrangú, minden csatorna elindul az összes node-on (függetlenül attól, hogy melyik gépen adták ki a start parancsot), és az elküldött üzenetet megkapja a csatorna minden node-on (függetlenül attól, hogy honnan küldték). Ez a felépítés lehetővé teszi, hogy a rendszer működőképes maradjon, ameddig egyetlen node is üzemel. Amennyiben a csatorna nem csak egyszerűen továbbküldi az üzenetet, hanem valamilyen – akár számításigényes – transzformációt is végrehajt azon, akkor a terhelés szétoszlik a node-ok között (minden csatorna elvégzi a feladatot a saját előfizetőinek). Ehhez a terheléseloszláshoz az kell, hogy az előfizető process-ek egyenletesen szóródjanak a node-ok között (az alábbiakban még lesz erre példa).

Másfajta munkamegosztásra is mód van: előfordulhat, hogy olyan mennyiségű adatot kell kezelni, ami nem fér el egy gépen. Ekkor minden csatorna a nála meglévő adatokkal dolgozik, majd a végén egy elküldi az eredményt egy másik csatornának, ami összesíteni tudja az eredményt, és azt küldi ki az előfizetőknek. Ez alapján lehet

egy egyszerű *map-reduce* rendszert implementálni. Akkor is járható ez az út, ha nem adatból van sok, hanem a számítás rendkívül időigényes, ám a feladat partícionálható (pl. π minél pontosabb meghatározása a Leibniz-módszerrel). Ehhez a feladatmegosztáshoz nem kell feltétlenül csatornát használni, elég ha egy csatorna a beérkező üzenet alapján részfeladatokat gyárt, és azokat a már ismerős `spawn` függvénnyel más és más node-okon indítja el.

Az egyes node-ok összekapcsolása egy egyszerű `net_adm:ping(Node)` hívással történet – azután felkerül a `Node` az ismert node-ok listájára. Lássuk működés közben!

A két node `zm@notebook` és `zm2@notebook` néven fut. Mindkettőn fut a ZM szerver. A `||` kezdetű sorok a másik node shell-jét jelölik.

```
(zm@notebook)2> net_adm:ping(zm2@notebook).
(zm@notebook)3> nodes().
[zm2@notebook]
  || (zm2@notebook)3> nodes().
  || [zm@notebook]
```

Csatorna elindítása és a két shell feliratkozása.

```
(zm@notebook)4> zm_channel:start(zm_ch_delayed,
                                 stock_exchange, []).
{[{ok,started},{ok,started}],[]}
(zm@notebook)5> zm_channel:list().
[stock_exchange]
  || (zm2@notebook)2> zm_channel:list().
  || [stock_exchange]

(zm@notebook)6> zm_channel:subscribe(stock_exchange, self(), []).
#Ref<0.0.0.407>
  || (zm2@notebook)3> zm_channel:subscribe(stock_exchange,
  ||                                       self(), []).
  || #Ref<0.0.0.386>
```

Üzenetküldés az egyikről.

```
(zm@notebook)7> zm_channel:send(stock_exchange,
                                {msg, {now(), "Kakukk"}}).
abcast
(zm@notebook)8> flush().
Shell got {msg,{stock_exchange,#Ref<0.0.0.407>},
               {{1335,201745,724599},"Kakukk"}}
```

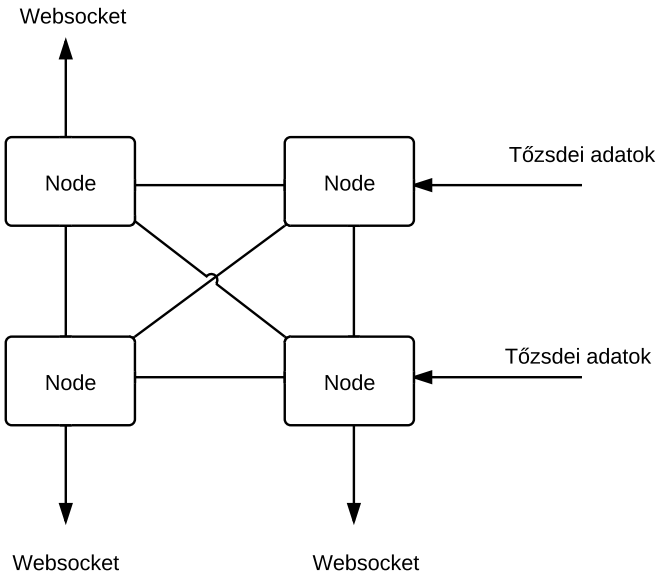
```
|| (zm2@notebook)5> flush().
|| Shell got {msg,{stock_exchange,#Ref<0.0.0.386>},
||                {{1335,201745,724599},"Kakukk"}}
```

Az egyiken elküldött üzenetet megkapta mindkét shell.

Itt az A4-es oldal adta korlátok miatt csak 2 node-dal néztünk egy példát, de ugyanilyen egyszerű további node-okat hozzáadni. Ahogy az új node elérhető, az is megkapja onnantól az összes üzenetet. Ezzel eljutottunk az alábbi felépítésig, már csak az hiányzik, hogy ne csak a shell process használja a csatornákat, hanem a külvilág is. Kapcsoljuk rá a böngésző előtt ülő felhasználókat a tőzsdéről érkező kereskedési adatokra!

3.7. Web interfész

Valami



Valami

3.7.1. RESTful webservices

3.7.2. HTML5 – websockets

4. fejezet

Összegzés

Irodalomjegyzék

Armstrong, Joe (2003): *Making reliable distributed systems in the presence of software errors*. PhD. thesis, The Royal Institute of Technology Stockholm, Sweden. Web: `http://www.erlang.org/download/armstrong_thesis_2003.pdf`, letöltés dátuma: 2012-04-01

Armstrong, Joe (2007): *Programming Erlang: Software for a Concurrent World*. USA: The Pragmatic Bookshelf.

Cesarini, Francesco – Thomson, Simon (2009): *Erlang programming*. USA: O'Reilly Media.

Logan, Martin – Merritt, Eric – Carlsson, Richard (2011): *Erlang and OTP in Action*. USA: Manning Publications.

Fielding, Roy Thomas (2001): *Architectural Styles and the Design of Network-based Software Architectures*. PhD. thesis, University of California, Irvine. Web: `http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf`, letöltés dátuma: 2012-04- 01.

Erlang documentation... (2011) Web: `http://www.erlang.org`

Akamai felmérés (2006): *Retail web site performance: Consumer Reaction to a Poor Online Shopping Experience* `http://www.akamai.com/dl/reports/Site_Abandonment_Final_Report.pdf`