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 mikroikozmoszt, 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 allapota 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 mellékhatásmentesek (*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 függvény-

hívásokkal, feltételes elágazás helyett a függvény deklarációba írt kifejezéssel (jelen esetben pozitív számokra értelmezett a függvény).

A funkcionális pogramnyelvek fontos ismérve az is, hogy a fügvények 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 inputokar 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.

Ehhez hasonlóan 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 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).

2.4. Az Erlang programozási nyelv

2.5. Open Telecom Platform

Behaviours. Supervisor hierarchies.

3. fejezet

Soft realtime messaging – a simple demo

4. fejezet

A messaging szerver

4.1. Felépítés

4.2. Hibatűrés

4.3. Elosztott rendszer – kell a redundancia

5. fejezet

Web interfész

5.1. RESTful webservices

5.2. HTML5 – websockets

6. fejezet

Üzemeltetés, karbantartás

6.1. Naplózás

Event manager, event handler.

6.2. Hibakeresés

Trace.

6.3. Hibajavítás, verzióléptetés

Upgrade.

6.4. Szállítás

7. fejezet

Alkalmazási lehetőségek, kitekintés

8. fejezet

Összegzés

Listing 1. Map reduce module

```
-module(mapreduce).

start() ->
  spawn(fun() -> init() end).

init() ->
  loop().

loop() ->
  receive
    {From, To, What} ->
      io:format("~p sent to ~p a message ~p~n", [From,To,What]),
      loop();
    _ -> % avoid full msg box
      io:format("Nothing. Finish.")
  end.
```

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`