Egy webkiszolgáló tündöklése és bukása

A cikk első feléből megtudhatod hogyan lehet öt perc alatt HTTP kiszolgálót írni. A másik felében rájössz, miért nem érdemes.

Szoftverfrissítések, hírek a nagyvilágból, tv-műsor, no és az ebédem. Ezeket nap mint nap az internetről szerzem be. Egészen pontosan meglátogatok a böngészőmmel egy olyan oldalt, ahol a web nyújtotta korlátlan multimédia lehetőségeket kihasználva választom ki a kényes ízlésemnek megfelelő programokat, híreket, filmeket és feltételeket. A háttérben a böngészőm egy távoli web kiszolgálóval beszélget a HTTP protokollt használva de ez engem hidegen hagy. Engem csak a pizzám érdekel. Viszont lehet, hogy írsz egy programot, ami egy-két adminisztrációs munkát tesz önműködővé, és jól jönne hozzá egy webes felület. A vezetőség talán azt se tudja, mi az az SSH, de szeretne új felhasználókat hozzáadni a rendszerhez. Vagy csak azt akarod tudni, hogy a csillogó-villogó külső alatt milyen fogaskerekek működtetik a webet. Mindkét esetben jó kiindulási pont lehet belekóstolni egy ilyen kiszolgáló lelki világába.

Az egyszerűség kedvéért egy olyan nyelvet fogunk használni, ami felépítésének köszönhetően gyors alkalmazásfejlesztést tesz lehetővé. Perlben fogjuk elkészíteni első szerverünket, így a fő irányelvek nem vesznek el a részletek tengerében. Ha még nem használtad ezt a szkriptnyelvet, itt az ideje, hogy ellátogass a ➲ http://www.perl.org/ címre. Ajánlom továbbá a perl kézikönyv oldalait, melyek Debian rendszer alatt a ***perl-doc*** csomagban találhatóak.

Mindössze egy Perl modulra fogunk támaszkodni a fejlesztés során, ez az *IO::Socket*. Ez része az általános Perl könyvtárnak (***Standard Perl Library***), így minden Perl terjesztés tartalmazza. A munkához tehát mindössze egy megfelelően telepített Perl változatra, illetve kedvenc szövegszerkesztődre lesz szükség. Utóbbiak közül grafikus felület alatt erősen tudom ajánlani a jedit-et (➲ http://www.jedit.org/). Számtalan kényelmi funkciója mellett hihetetlen nagy számú nyelvet támogat színkiemeléssel is, ezért egy próbát mindenkinek megér.

Akkor vágjunk neki! Ha már olvastad A Perl és a hálózat (Linuxvilág, 2002. március) című cikkemet, az első néhány sor nem fog meglepetést okozni.

Az első sorban megadjuk a szkript parancsértelmezőjének elérési útvonalát, és kezdeti paramétereit. Ennek segítségével a szkriptnek futtatási jogot adva, közvetlenül indítható parancssorból. A `-w` hatására minden figyelmeztető üzenet megjelenik az szabványos hibacsatornán (***standard error***).

Ezután két modult is használatba veszünk. Az első, a `strict` minden szabványos Perl program alapköve. Ennek hatására hibaüzeneteket kapunk, ha az egységesítés útjára lépett nyelvben nem szép szerkezetekkel dolgozunk. Az `IO::Socket` modul pedig a hálózatkezelést egy kellemesen használható, objektum-orientált felület mögé rejti.

```
my $docroot = "./web";

die "Usage: " . $0 . " {port}\n" unless (@ARGV > 0);
```

A változók deklarációja kötelező, ha strict üzemmódban vagyunk. Erre szolgál a `my` kulcsszó. A `$docroot` egy globális változó lesz, mely annak a főkönyvtárnak az elérési útját tartalmazza, amelyben az összes HTML állományt tárolni fogjuk. Ezzel ugyanazt szeretnénk elérni, mint az Apache a DocumentRoot direktívával. A következő sor ellenőrzi, hogy van-e átadott parancssori paraméter. Így várjuk ugyanis azt a kapuszámot, ahol a kiszolgáló figyelni fog és várja a beérkező kéréseket. A program használat alakját írja ki a képernyőre és megszakítja futását, amíg az átadott paraméterek száma nem nagyobb nullánál. Ehhez csak annyit kell tudni, hogy Perl alatt egy tömb (jelen esetben `@ARGV`) skalárként értelmezve a tömb elemeinek számát adja vissza.

```
my $server = new IO::Socket::INET (
        LocalPort => $ARGV[0],
        Proto => "tcp",
        Listen => SOMAXCONN,
        ReuseAddr => 1
);

die "Error creating socket.\n" unless defined
➥$server;
```

Ezután újabb két sor következik, amit csupán a könnyebb olvashatóság érdekében tagoltam több sorba. Előbb egy új változót hozunk létre, majd ellenőrizzük, hogy sikerrel jártunk-e. Első lépésben a new kulcsszóval az `IO::Socket::INET` osztályból hozunk létre egy példányt, melynek segítségével majd felépítünk egy TCP/IP kapcsolatot. A konstruktornak egy olyan asszociatív tömböt kell átadni, mely kulcs-érték párjaival leírja a felépítendő foglalat

(*socket*) típusát. A legfontosabb a parancssorban átadott kapuszám. A várakozási sort a lehető legnagyobbra állítjuk, illetve beállítjuk, hogy minden kapufoglalás előtt próbálja meg felszabadítani azt, ha szükséges (lásd: `IO::Socket::INET(3 perl)`).

```
$SIG{INT} = sub {
        print "Stopping WEB server.\n";
        close $server;
        exit;
};
```

Következő lépésben a megszakítás szignálhoz (***interrupt signal***) hozzárendelünk egy névtelen eljárást, amely a kiszolgáló biztonságos leállításáért felel. Ez azért szükséges, mert az elindított webszerver majd csak a Ctrl+C billentyűkombináció hatására fejezi be működését, azonban ekkor is fontos a biztonságos leállítás. A %SIG szintén egy asszociatív tömb, melyet a szignálok rövid neveivel lehet indexelni, és minden szignálhoz egy függvényt rendel. Jelen esetben Ctrl+C esetén kiírja a képernyőre az elköszönő üzenetet, lezárja a program elején nyitott foglalatot, majd kiszáll.

```
print "Starting WEB server (port " . $ARGV[0]
➥. ").\n\n";

for (my $client; $client = $server -> accept ();
➥close $client) {
...
}
```

Ez a ciklus lesz a kiszolgáló lelke. A '.' operátor hatására összefűzött karakterfüzér megjelenítése után valójában egy végtelen ciklus indul el. Ebből csak szignálok használatával lehet kilépni. A ciklus kezdetekor deklarálunk egy `$client` nevű változót, mely a csatlakozó ügyfél foglalatát fogja tárolni. Minden iteráció elején meghívjuk a `$server` objektum `accept ()` elemfüggvényét. Ez addig nem adja vissza a vezérlést, amíg egy új ügyfél nem érkezik. Ekkor felépíti a kapcsolatot, és visszaadja a kliens foglalatát. Az iteráció végén, amikor az ügyfelet kiszolgáltuk, ez utóbbit természetesen le kell zárnunk.

```
print localtime () . " Connect from "
➥. $client -> peerhost () . "\n";
my $line = <$client>;
print "  " . $line;
...
print localtime () . " Closing connection.\n\n";
```

A ciklus minden lefutásának elején kiszolgálóoldalon kiíratjuk az aktuális dátum mellett az ügyfél csatlakozásának tényét, illetve IP-címét. Ezt az adatot a `$client` objektum `peerhost ()` elemfüggvényével határozzuk meg. Ezután fogadjuk az ügyfél kérését, ami azt jelenti, hogy egy sort olvasunk be a kliens foglalatról a gyémánt (***diamond***) operátor segítségével. Ezt a `$line` változóban tároljuk, és szemléletesség kedvéért azonnal ki is íratjuk a kiszolgáló oldalán. Természetesen az iteráció végén a kapcsolat lezárásának ténye is megjelenik a képernyőn.

```
if ($line =~ /^GET (.*) HTTP.*\r$/) {
   my $file = $1;
   $file .= "index.html" if ($file =~
   ➥/\/$/);
   print $client "HTTP/1.0 200 OK\r\n";
   print $client "Server: BigWig WEB
   ➥server\r\n";
   print $client "Connection: close\r\n";
   print $client "Content-Type:
   ➥text/html\r\n\r\n";
   ...
}
```

Csak az ügyfél legegyszerűbb GET kéréseit dolgozzuk fel. A `$line` változót, amely a kliens által küldött sort tartalmazza, próbáljuk meg az =~ operátorral egy reguláris kifejezésre illeszteni. A '/' jelek között szereplő kifejezés a `GET /eleresi_ut/file HTTP/x.x` alakra illeszkedik, ahol `x.x` a protokoll változatszáma. A feltételes szerkezet magjába akkor lép be, ha a minta illeszkedett, sőt mi több a `$1` változó a zárójelezett `.*` miatt tartalmazni fogja a kért állományt. Ezt egy `$file` változóba helyezzük át az átláthatóság és a módosíthatóság végett, és a végére illesztünk egy „`index.html`" szöveget, ha '/' jelre végződik. Továbbá kiíratjuk a szabályos HTTP fejlécet az ügyfél oldalára.

```
$file = $docroot . $file;
if (-f $file and -r $file) {
  print "  Serving " . $file . "\n";
  open (INPUT, $file);
  while (<INPUT>) { print $client $_
  ➥. "\r\n"; }
  close (INPUT);
} else {
  print "  404 Not found\n";
  print $client "<HTML><HEAD><TITLE>
  ➥\r\n";
  print $client "404 Not found\r\n";
  print $client
  ➥"</TITLE></HEAD><BODY>\r\n";
  print $client "<FONT size=+2>Sorry,
  ➥the page you\r\n";
  print $client "have requested cannot
  ➥be found.\r\n";
  print $client "</FONT></BODY></HTML>
  ➥\r\n";
}
```

Végezetül az állománynév elé tesszük a főkönyvtár elérési útját, ellenőrizzük a file meglétét, és kiíratjuk az ügyfélnek. A `-f` és `-r` különleges függvények (lásd `perlfunc(1)`). Nemnulla visszatérési értékkel jelzik, ha a file közönséges, illetve ha olvasható. Ez esetben megtesszük a szokásos bejegyzést kiszolgálóoldalon, megnyitjuk az állományt, soronként kiíratjuk, majd lezárjuk. Ellenkező esetben pedig egy olyan HTML oldalt nyomtatunk a kliensnek, ami arról tájékoztatja, hogy a keresett oldal nem található. Ennek megfelelően már egy kis HTML tudással tetszőleges dinamikus tartalom készíthető.

```
print "  Creating dinamic content.\n";
print $client "<HTML><HEAD><TITLE>\r\n";
print $client "Dinamic page\r\n";
print $client "</TITLE></HEAD><BODY>\r\n";
print $client "<TABLE border=1>\r\n";
for (my $i = 1; $i <= 10; $i++) {
   print $client "<TR>";
   for (my $j = 1; $j <= 10; $j++) {
      print $client "<TD>" . $i * $j
      ➥. "</TD>";
   }
   print $client "</TR>\r\n";
}
print $client "</TABLE></BODY></HTML>
➥\r\n";
```

A fenti példa dinamikusan hoz létre egy 10x10-es szorzótáblát. Látható, hogy a HTML fejléc kiíratása után egy kettős for ciklussal készítjük el a táblázatot. A külső végigmegy a sorokon, és a belső kitölti azokat. Csak arra kell figyelni, hogy a nyomtatás ne a szabványos kimenetre (standard output) történjen, hanem az ügyfél foglalatára. Az eddigi példákból könnyen látható, hogy a foglalatok ugyanolyan egyszerűen írhatók-olvashatók, mint a közönséges fileleírók (***file handler***).
Utóbbi, dinamikus rész úgy van beillesztve, hogy még a kiszolgálás előtt történik egy esetszétválasztás. Ha a kérés *„**/dinamikus/index.html**”*-re vonatkozik, ez a rész fut le, egyéb esetben az állomány kiíratása. Tetszőleges böngészővel a ***http://localhost:port/eleresi_ut*** címre mutatva érhetjük el a helyi gépről a kiszolgálót, de ne felejtsük el, hogy a hálózat tetszőleges pontjáról megtalálható a megfelelő cím megadásával. A mellékelt képen jól látható, hogy kiszolgálónk támogatja az összes népszrű böngészőt.

Ha ez mind ilyen szépen működik, akkor mi a baj?

A kiszolgáló első és legszembetűnőbb hibája, hogy Perlben íródott. A Perl egy szkriptnyelv, vagyis nem előrefordított. Nagy terhelés mellett nem nehéz megfektetni egy szervert, amely olyan nyelven íródott, ami futásidőben értelmeződik. Valódi, nagy erőforrásigényű alkalmazásokat érdemes inkább C/C++-ban készíteni. Ez a példa viszont egy állatorvosi ló. Azért íródott Perlben, hogy a legfontosabb részek jól felnagyíthatók legyenek, és ne zavarjanak minket az apró részletek. Ha az irányelveket elkaptad, nézz utána, hogyan írhatnád meg ugyanezt pl. C-ben.
Sajnos a szerverünk a dinamikus tartalom készítése miatt HTML kódot tartalmaz! Ez egy súlyosan hibás elgondolás. Valamivel tűrhetőbb megoldás lett volna, ha az oldal készítését egy külső szkriptre bízzuk, a szerverünknek így azt csak meghívnia kellene. Ezt az eljárást hívják CGI-nek (***Common Gateway Interface***), amikor egy külső program készíti a teljes választ az ügyfél GET kérésére. Ez azonban szintén egy túlhaladott elképzelés, ma már sokkal szívesebben használnak oldalba ágyazott szkripteket (ilyen a PHP).
A szerver nem tud egy időben egynél több ügyfelet kiszolgálni. Ha két kliens egyszerre csatlakozik, előbb a gyorsabbat kiszolgáljuk, ezalatt a másik várakozási sorba kerül (***listen queue***). Ezt nagyjából úgy lehet elképzelni, mint egy fogorvosi rendelőt, amelyben egyetlen orvos van. A várószoba méretét adhattuk meg a ***$server*** objektum létrehozásakor a Listen paraméterrel. Ahhoz, hogy több orvosunk legyen, gyermek folyamatokat kellett volna létrehozni minden egyes csatlakozó ügyfélnek, ám ez szintén elbonyolította volna a példát. Ha ennek a mikéntje érdekel, olvasd el a vonatkozó kézikönyv lapot: `perlfork(1)`.
Elég gyermekded módon bántunk a hibakezeléssel is. Ha az állomány nem létezik, a HTTP szabvány szerint nem szabad OK kódot adni az ügyfélnek. Mi a HTTP fejlécben kiadtunk egy HTTP 200 OK-t, majd nyomtattunk egy olyan HTML oldalt, ami leírja, hogy mennyire sajnáljuk, amiért nem tudjuk kiszolgálni. Már a fejlécben illett volna 404-es kóddal jelezni, hogy az oldal nem található. Ezeket a dolgokat nem szabad elnagyolni, mert a szabványtól való eltérés miatt semmi sem garantálja, hogy a böngésző tényleg megjeleníti a tartalmat.
Egy portál gyakran tartalmaz képet. Máskor hangot, videót, flash animációt, letöltésre váró állományt. A sokféle tartalom egységes kezelésére találták ki a MIME információt. Ez is a HTTP fejlécben utazik, és egy főcsoport/alcsoport formában szöveges leírással szolgál az adott állományról. Mi egyszerűen azt mondtuk, hogy minden text/html. Ennek az az eredménye, hogy ha egy képet kér egy kliens, azt is ezzel a MIME jelzéssel adjuk át, ezért nem a kép fog megjelenni a böngészőben, hanem a bináris tartalom.
Végül, de nem utolsó sorban alapvető biztonsági hibától szenved a szerver. A főkönyvtáron kívüli tartalom is elérhető, mindössze olyan GET kérést kell fogalmazni, amely kellő számú „..”-t tartalmaz.Mivel ez nem valódi chroot, ezért elég a böngésző címsorába azt írni, hogy `http://localhost:port/../web.pl`, és máris olvasható a webszerver forrása. Ez azonban nem az egyetlen olyan információ, amely bizalmas lehet, gondoljunk csak /etc/passwd állományra, amely bárki számára olvasható lesz, amint a támadó eltalálta a megfelelő számú “..”-ot.

Ha ez ennyire rossz, akkor mire volt jó?

Okos ember mások hibáiból tanul – szokták mondani. Ha ez egy informatikusra igaz, az hatalmas szerencse mindenkinek. Olvass utána azoknak a részeknek, amelyekről most hallottál először. Utána próbáld meg átírni a szervert úgy, hogy eggyel kevesebb hiba legyen benne – legyen az elvi, kényelmi, vagy biztonsági.
A fő az, hogy meglásd a saját kódodban is az ilyen hibákat és arra törekedj, hogy kijavítsd őket, ahelyett, hogy elfednéd.

Írj bátran, ha kérdésed van.
Sok sikert és örömet a kísérletezgetéshez!

Fülöp Balázs (admin@guardware.com)
Imádja a Túró Rudit, a Debian Linuxot és a teheneket. Kedvenc írója Slawomir Mrożek. Leginkább a számítógépes hálózatok biztonsága érdekli. A BME VIK műszaki informatikus szak hallgatója.