# Gentoo másképp

A Gentoo-t legtöbben a fejlesztői munkaállomások egyik vezető terjesztésnek tekintik, de az egyszerű csomagkezelési rendszer miatt olykor jó választás lehet bármely folyamatosan naprakész termelési rendszerben.

Be kell vallanom valamit. ***Gentoo Linuxot*** használok. Kollégáim a különféle Linux felhasználói csoportok találkozóin úgy gondolják lökött vagyok. Mindenki tudja, hogy a ***Gentoo*** forráskód alapú ***Linux***-terjesztés. A ***Gentoo*** általános elképzelés szerint (amit igen nagy mértékben befolyásoltak a terjesztést fejlesztő emberek) olyanoknak való, akik extrém mértékű optimalizálást szeretnének, és igazából csak azoknak felel meg, akik asztali gépen használják. Valójában a ***Gentoo*** meglepő módon rengeteg más feladatra is ideális. Nem kis meglepetésemre néhányan már manapság is ***Gentoo***-t használnak termelési környezetben pontosan ezek miatt az okok miatt.

## Sebesség

Minthogy az eredeti ***i386*** processzor leszármazottai binárisan együttműködőek, más ***Linux*** terjesztések (nem beszélve azokról akik ***Microsoft Windowshoz*** írnak programokat) programjuk előrecsomagolt bináris változatát a gyakori ***i386*** rendszerre készítik el kihasználva, hogy így az mindenhol fut. A dolog másik oldala, hogy ezek a terjesztések nem tudják kihasználni azokat az új képességeket amelyeket okos processzorunk nyújthatna, ami azért kár.
A ***Gentoo*** ezzel szemben forrásból-épített terjesztés ahol beállíthatjuk a fordítókapcsolókat amikor rendszerünkhöz alkalmazásokat telepítünk. A ***GCC*** ugyanis lehetővé teszi, hogy meghatározzuk a ***CPU*** típusát, amelyen a kódot futtatni szándékozzuk. A processzortípus meghatározásával (például ***Intel Pentium III*** vagy ***AMD Athlon Thunderbird***) a fordító képes processzorfüggő kódot készíteni, amely (legalábbis elméletben) jobb, gyorsabb gépi kódot eredményez. Gyorsabb tehát a ***Gentoo*** rendszer? Anekdotákon alapuló bizonyítékok elég változóak. Úgy tűnik a ***Gentoo*** rendszerek valamivel gyorsabban futnak mint egy azonosan beállított népszerűbb terjesztés rendszere. Azonban bármilyen apróbb teljesítmény előny azonnal értelmét veszti, ha a rendszer nincs helyesen telepítve beállítva és finomhangolva. Minthogy a legtöbben nem tudjuk hogyan kell ezt csinálni és mivel a ***Gentoo*** olyan nagy mozgásteret nyújt a saját megoldásainknak, ha valami butaságot csinálunk, könnyen elveszíthetjük a némiképp gyorsabb programok előnyét.
Így sebesség tekintetében voltaképp nem igazán számít, hogy forrásból-felépített vagy bináris alapú terjesztést használunk-e. De ha a nagyobb sebesség nem érv a ***Gentoo*** mellett, akkor meg mi értelme van egyáltalán ennek a forrásból telepítés dolognak?

## A termelési környezetek általános problémái

Az embereket több módon is felbosszanthatja számítógépük. Amire mi most koncentrálni próbálunk egy újabb keletű probléma, amelybe az új operációs rendszerek két esetben futnak bele. Az első eset, amikor a felhasználó valami olyan eszközkészletet szeretne, ami a terjesztésben nem talál meg. Eredményképpen a felhasználónak magának kell összeállítania azt. A második eset amikor a terjesztésben rendelkezésre álló csomagnak új verziója jelenik meg. Mindkét eset a kulcsa: az egyszerű megoldhatóság. Kérdés, hogy mennyire segít bennünket az operációs rendszer a karbantartást kísérő buktatók során? Legnagyobb meglepetésemre, a ***Gentoo Linux*** úgy tűnik igazán jónak bizonyult ebben a tekintetben. A ***Gentooval*** új csomagokat úgy telepítünk, hogy letöltjük a forrást és befordítjuk. Egy programot szeretnénk? Semmi gond, kiadunk egy utasítást és kicsivel később már fenn is van. Olyan mintha ***Debiant*** használnánk.
***Gentoo*** akkor csillogtatja meg igazán tudását, amikor a felhasználónak újabb verziójú programra van szüksége. Tegyük fel a ***Bluefish HTML*** szerkesztőt használom, ám egy hiba idegesít benne. Könnyen lehet, hogy a ***Bluefish*** újabb verziója már megtalálható a ***Gentoo Portage*** nevű csomagkezelő rendszerében, így lehet, hogy máris lekérhetem a frissítést. A ***Gentoo*** hasznos `etcat` parancsa megmutatja nekünk mit érhetünk el:

```
# etcat versions bluefish
```

Az `etcat` azt mondja, hogy a 0.9-es ***bluefish*** van telepítve viszont már elérhető a 0.12-es is. A ***bluefish*** weblapjáról megtudom, hogy a 0.12-esben már javították a hibát, tehát biztosan frissíteni szeretnék.
A `emerge` parancs megmutatja mi fog történni ha frissítek:

```
# emerge --pretend bluefish
```

Úgy tűnik ennek a ***bluefish*** verziónak szüksége van a ***libpcre*** nevű könyvtárra. A ***Portage*** azt mutatja nekünk,

```
*  app-editors/bluefish :
        [  I] app-editors/bluefish-0.9 (0)
        [   ] app-editors/bluefish-0.11 (0)
        [   ] app-editors/bluefish-0.12 (0)
        [M~ ] app-editors/bluefish-0.13 (0)
```

*1. ábra* Az etcat eszközzel lekérdezhetjük milyen bluefish verzió áll rendelkezésünkre

```
These are the packages that I would merge, in order:

Calculating dependencies ...done!
[ebuild  N    ] dev-libs/libpcre-4.2-r1
[ebuild     U ] app-editors/bluefish-0.12 [0.9]
```

*2. ábra* Az emerge – pretend segítségével ellenőrizhetjük a telepítendő program függőségeit

hogy a ***bluefish*** frissítésén felül a ***libpcre*** könyvtárat is le fogja tölteni. Lássunk neki:

```
# emerge bluefish
```

A ***Portage*** először letölti, lefordítja és telepíti a ***libpcre***-t, végül ugyanezt végrehajtja a ***bluefish*** esetében is. Négy perccel később kész is a frissítésem. Hát ez nem volt túl nehéz. Biztosan észrevették, hogy nem a 0.13-as verziót ajánlott fel telepítésre. Ez azért van, mert jelenleg a 0.13-as verzió maszkolt, ezért volt vörössel jelölve. Ebben a felállásban a 0.13 éppen csak kijött, de már van hozzá `ebuild`. Ez az `ebuild` azonban még tesztelés alatt áll, hogy kiderüljön, a program ténylegesen telepítődik-e és nincs-e véletlenül valami durva hiba benne. Amennyiben tényleg szükségünk lenne rá, felülbírálva a ***Portage*** döntését utasíthattuk volna, hogy a 0.13-ast hozza le. Ugyanígy, ha okom van rá, választhattam volna a 0.11-est is. Ez a rugalmasság a ***Gentoo*** legnagyobb ereje.

## Csináld magad csomagok

Egy kicsit trükkösebb eset amikor olyan programot szeretnénk felrakni amit nem támogat a rendszer. Az egyik legfontosabb ok amiért terjesztésekben bevezették a csomagkezelést, hogy egyetlen egységes rendszerben láthassák, milyen programok vannak a rendszerre telepítve. Minden egyes programhoz, legyen az alapprogram, rendszerkönyvtár, kiszolgáló program vagy felhasználói alkalmazás, egy csomagot készítenek. Ahogy a csomagot a rendszerre telepítjük, az operációs rendszer rögzíti, hogy mely állományok hova kerülnek, majd a csomag a rendszerre kerül. Ezzel a módszerrel a csomagtól függő más csomagok már tudni fogják, hogy az előkövetelményeik a helyén vannak.
De mi történik, ha újabb verziót akarunk feltenni egy programból, amihez nem készült megfelelő csomag az operációs rendszerünkhöz? Általában ugyanazt az utat kell bejárnunk mint a csomag eredeti készítőjének, kivéve ha a következő két dolog valamelyikét választjuk:

1. Egyedi helyre telepítjük, valószínűleg a ***/usr/local/bin*** könyvtárba, majd biztosítjuk, hogy a régi helyett az új program fusson.

2. Vakon telepítjük a programot a gyökér fájlrendszerre, reménykedve hogy nem nyiffantunk ki valamit közben, és imádkozunk, hogy a később telepítendő dolgok nem fognak semmit felülírni a most feltett dolgok közül.

Gondolkozzunk csak el ezen egy kicsit. Nem találjuk furának ha valaki nem aggódik ilyesmi miatt? Végül is nem éppen ez az, amit a csomagkezelésnek meg kellene előznie? Nem az a kérdés, hogy „lehet-e csomagokat készíteni”, mert erre a válasz „igen, teljeskörűen”, és nem is az, hogy „létre tudjuk-e hozni saját csomagjainkat”. Sokkal inkább azt szeretném tudni, hogy milyen nehéz mindezt végrehajtani. Tegyük fel, hogy fenn van az OS-ból feltett `bogofilter` programunk és a 0.16.1-es verzióhoz való ***rpm*** csomagunk. Hirtelen a `bogofilter` alkotója rádöbben, hogy egy buta, ám igen súlyos hiba rejtőzik a változatban, ezért gyorsan kiad egy 0.16.2-os frissítést.
A probléma ott kezdődik, hogy kénytelenek vagyunk megvárni amíg a terjesztésünk kiadja a saját ***rpm***, ***.deb***, ***.pkg*** vagy más egyéb változatát, ami akár sokáig is eltarthat, így rákényszerülünk, hogy saját változatot készítsünk. Nos, ekkor jönnek a problémák. Elméletileg az ***.rpm*** vagy ***.deb*** csomagok létrehozása egyszerű. „Alapnak egyszerűen csak vegyük a már meglévő 0.16.1 csomagot”. Csakhogy a legtöbb embernek, legalábbis akik nem állnak mágus szinten (de néha még nekik is) elég komoly kihívást jelent az ilyesmi. Saját magunknak kell:

- Letölteni a csomag leírást vagy valahogy kiszednünk egy meglévő csomagból.

- Kézzel letöltenünk és kicsomagolnunk a legújabb ***.tar.gz*** (vagy éppen ami) forrást.

- Át kell alakítanunk a leírásokat (a ***Debian*** esetében a legfrissebb forrásokba) sőt előfordulhat, hogy meg is kell foltoznunk azokat.

- Lehet, hogy meg kell változtatnunk az behúzó-parancsfájlt, hogy helyesen működjön az új verzió esetében is.

- Ezután megpróbálhatjuk létrehozni a csomagot. Ehhez először is le kell fordítanunk, ami valószínűleg jó néhány ***-dev*** csomag feltelepítését jelenti amelyekről azelőtt azt sem tudtuk, hogy léteznek.

- Végül telepítünk és tesztelünk.

Mindez természetesen kivitelezhető ugyan, de a szükséges ismeretek megszerzése igencsak meredek tanulási görbét jelent (különösen az újaknak). Akárhogy is nézzük, ez elég sok munka amit szívesen kihagynánk.

## A Gentoo csomagleírásai

Bár alapvetően nem különbözik a fent leírt folyamattól, csomagokat telepíteni egy ***Gentoo*** rendszerre könnyebb. A trükk a ***Gentoo*** egyszerű formátumú `ebuild` csomagleírásaiban rejlik. Ezek lényegében héjprogramok (az ***ebuilds***-al később még foglalkozunk a cikkben). A folyamat közben megadjuk honnan kell beszerezni a forrás tarlabdát. Amikor fordítunk, a Portage letölti a forrást majd

elkezdi kitömöríteni és lefordítani. Mivel héjprogramokról beszélünk, nagy hatékonysággal tudják használni a héjváltozókat. A verziószámot az ***ebuild*** fájlnév értelmezésével nyeri ki majd egy olyan változóba teszi, amit a parancsfájl fel tud használni.
A fenti `bogofilter` példánkban, a csomagállomány (melynek neve ***bogofilter-0.16.1.ebuild***) a következő sort tartalmazza:

```
SRC_URI="http://sourceforge.net/downloads/
➥bogofilter-${PV}.tar.gz"
```

Amikor belefogunk a `bogofiltert` felkészítésebe és fordításába, a ***Portage*** a fájlnév alapján a `$PV` változót 0.16.1-re állítja és a megfelelő ***.tar.gz*** állományt tölti le. Ezek után kicsomagolja, majd a

```
./configure
make
make install
```

parancsok futtatása után az utasításoknak megfelelően felépíti a csomagot. Amennyiben létre szeretnénk hozni a kívánt 0.16.2-es új verziónkhoz tartozó ***ebuild*** állományt a következőket kell tennünk:

```
# cd /usr/portage/net-mail/
# cp bogofilter-0.16.1.ebuild
➥bogofilter-0.16.2.ebuild
# ebuild bogofilter-0.16.2.ebuild digest
```

Feltételezve, hogy a csomagleírásban és a kicsomagolási útmutatókban semmi frissítendő nincsen, mindössze ennyit kell tennünk.
Van még pár dolog amire érdemes odafigyelni. Például, a fentieket valószínűleg inkább a ***/usr/portage*** fa külön másolatán szeretnénk végrehajtani, hogy ne veszítsük el változtatásainkat ha az elsődleges fa esetleg megváltozik. A ***Portage*** közvetlenül is támogatja ezt. Ha kíváncsiak vagyunk, hogyan kell megmutatni a ***Portage***-nek hol tároljuk a saját ***ebuild***-jeinket, nézzük meg a `PORTAGE_OVERLAY` változó leírását a hálózati dokumentációban vagy közvetlenül a ***/etc/make.conf*** állományban. Most már kiadhatjuk a ***Portage***-nek a

```
# emerge bogofilter
```

parancsot és máris megvan az új verziónk.
A ***Gentoo*** valamennyi csomagjának forrás tarlabdájából másolatokat tart fenn a világszerte megtalálható tükrein. Normál esetben a ***Portage*** az egyik ilyen gépről tölti le a forrást. Akkor sincs semmi probléma, ha esetleg valami olyasmit fordítanánk ami nincs fenn a ***Gentoo*** tükrein. A ***Portage*** egyszerűen az eredeti, fejlesztői letöltőoldalt fogja használni.
A ***Portage md5sum*** összegekkel ellenőrzi a letöltött állományok hibátlanságát. A fenti harmadik parancs (`ebuild ... digest`) éppen ezért került oda. Miután letöltöttük a forrást így az md5sum ellenőrzést is rögtön elvégezzük. Mivel most mi hajtjuk végre a verzióugrást, nekünk kell ellenőrizni, hogy biztosan hibátlan-e a letöltésünk. Ezért aztán érdemes először az `ebuild ... unpack` paranccsal megszerezni a letöltést, meggyőződni a hibátlanságáról, és csak ez után feldolgozni a parancsot.
Végül, ha olyan programcsomagot szeretnénk, amit az operációs rendszerünk nem támogat, meg kell írnunk a sajátunkat. A ***Gentooval*** nagyon könnyen készíthetünk saját ***ebuild***-et.

## Minden, amit az ebuildről tudni kell

A ***Gentoo*** csomagleírásai ***Bash***-ben készültek. A különféle utasítások függvényekbe kerülnek amelyeket a ***Portage*** hív meg a folyamat során. A főbb elemek:

```
pkg_setup()
src_unpack()
src_compile()
src_install()
pkg_preinst()
pkg_postinst()
```

Ezek sorrendben hívódnak meg. Ha meg akarjuk mutatni a ***Portagenek***, hogyan készítse el a programunkat, írjuk meg a lépésekhez tartozó függvényeket, mindegyiket ellátva némi információval, mint például a korábban említett `SRC_URI`.
A forrásaink fordításához például a következőket használhatjuk:

```
src_compile () {
        ./configure --prefix=/usr
        make
}
```

Ezekben a héjprogramokban az a csodálatos, hogy felüldefiniálható, alapértelmezett változattal rendelkeznek.
Az alapértelmezett `src_compile()` valójában igen hasonló az itt bemutatotthoz, ami sok csomaghoz tökéletes. Tulajdonképpen akár olyan ***ebuild***-et is készíthetünk, amely teljes egészében az alapértelmezett változatokat használja és egyáltalán nem rendelkezik saját függvényekkel.
Néha előfordul, hogy egy csomagot rendszertől függően másképpen szeretnénk beállítani a `./configure`-al. A ***Portage*** ***/etc/make.conf***-ban található és parancssorból felülbírálható `USE` környezeti változója olyan tokeneket tartalmaz, amelyek a rendszerünk testreszabásában és leírásában segíthetnek. Tegyük fel, hogy van egy csomagunk, amit többféleképpen is le lehet fordítani attól függően, hogy szeretnénk-e mondjuk ***X Window System*** vagy ***IPv6*** támogatást. Az `src_compile()` függvényünk a következőképpen nézne ki:

```
src_compile () {
        use X    && conf="${conf} --with-x"
        use ipv6 || conf="${conf} --without-ipv6"
        ./configure --prefix=/usr ${conf}
        make
}
```

Különféle héjprogramozási megoldásokkal találkozhatunk itt. A fenti példánkban ha a rendszerünkön van ***X***, a csomagnak megüzenjük, hogy készítsen ***X*** támogatást.

*1. lista* ssh2-3.2.9.1.ebuild

```
DESCRIPTION="ssh.com's implementation of SSH2"
SRC_URI="
ftp://mirror.aarnet.edu.au/pub/ssh/
➥ssh-${PV}.tar.gz
ftp://ftp.ssh.com/pub/ssh/ssh-${PV}.tar.gz"
HOMEPAGE="http://www.ssh.com/products"
SLOT="0"
LICENCE="free-noncomm"
KEYWORDS="x86 ~ppc"
RDEPEND="virtual/glibc
        !net-misc/openssh
        >=sys-libs/zlib-1.1.4"
DEPEND="${RDEPEND}
        dev-lang/perl
        >=sys-apps/sed-4"
PROVIDE="virtual/ssh"
IUSE="X ipv6"
RESTRICT="nomirror"
# A csomagot ssh2-nek hívjuk; a forrás
# tarlabdák alakja ssh-x.y.z Ezért felül kell
# írnunk az S-t, hogy kicsomagoláskor a helyes
# könyvtárra mutasson.
S="${WORKDIR}/ssh-${PV}"
# Itt valószínűleg hagyatkozhattunk volna az
➥alapértelmezésre is
src_unpack() {
    unpack ${A}
}
# A nagyszámú configure kapcsoló közül
# az X windows és az Ipv6 befordításának
# lehetőségét kezeljük le itt.

src_compile() {
    local conf
    use X    && conf="${conf} --with-x"
    use ipv6 || conf="${conf} --without-ipv6"
    ./configure ${conf} --host="${CHOST}" \
        --prefix="/usr" \
        --with-ssh-agent1-compat \
        --with-etcdir="/etc/ssh2" \
             || die "configuration failed"
    make || die "compile failed"
}
# ez is szinte azonos az alapértelmezettel,
# de az rc parancsfájl nevét meg szeretnénk
➥változtatni
src_install() {
    make DESTDIR=${D} install
    exeinto /etc/init.d
    newexe ${FILESDIR}/sshd2.rc sshd2
}
```

Amennyiben kiszolgálóról van szó és erre semmi szükségünk, a programunk a felesleges részek nélkül készül el. Az `USE` változókat a parancssorból felülbírálhatjuk, így ha szükséges még pontosabb vezérléssel is dolgozhatunk.

Az `src_unpack()` ugyanígy működik. Amennyiben nem adunk meg semmit, a ***Portage*** nekilendül, az alapértelmezett helyre kitarolja a forrás tarlabdát, belép a könyvtárba és megfelelően beállítja a `$WORKDIR` munkakönyvtár környezeti változót. Másrészről, ha valami szokatlan dolgot is kell csinálni, például egy foltot kell feltenni- magunk is létrehozhatjuk az egyszerű kicsomagoló függvényt:

```
src_unpack () {
        unpack ${A}
        epatch ${FILESDIR}/fixit.patch
}
```

Végül nézzünk egy teljes példát. Volt egy ügyfelem, amely igen sokat használta az ***SSH2*** protokoll ***ssh.com*** által készített változatát. Ezért aztán sok gépre kellett feltelepítenem. Lásd az *1. listát*.
Az ***ebuild*** néhány környezeti változó beállításával kezd:

- `SLOT`: ezeket általában a programkönyvtárakhoz használjuk. Amikor az ***ebuild*** szerzője tudja, hogy ugyanazon csomag különféle verziói lehetnek fenn egy időben a gépen, egy „slot"-szám hozzárendelésével tehet különbséget közöttük. Az egyik rendszeremen fenn van a ***Berkeley DB 1.85***-ös verzió (`SLOT 1`), a 3.2.9-es verzió (`SLOT 3`), a 4.0.14-es verzió (`SLOT 4`) sőt még a 4.1.25_p1-es verzió is (`SLOT 4.1`). Elég sok program létezik, amit a régebbi ***API***-hoz fejlesztettek, és nincs semmi okunk rá, hogy ezeket ne engedjük telepíteni. Amikor aztán a 4.0-ás sorozatban megjelenik egy újabb stabil kiadás, mondjuk a 4.0.17-es verzió, amíg a `SLOT` 4-et használjuk hozzá, a többi verzió eltávolítása nélkül is nyugodtan frissíthetek a 4.0.14-esről. Valójában a ***Berkeley DB*** ennél valamivel összetettebb példa, de jól bemutatja a ***Gentoo*** „slot" elképzelésének erejét. A legtöbb ***ebuildnek*** persze semmi ilyesmire nincs szüksége és a `SLOT="0"` értéket használja.

- `KEYWORDS`: itt jelezhetjük a támogatott architektúrákat. A példában azt mutatom be, hogy ez az ***ebuild*** az ***x86*** sorozaton működőképes és elismerten stabil. A `~ in ~ppc` azt jelzi, hogy ezek maszkoltak. Tudom, hogy a korábbi verziók lefordultak a ***PowerPC*** rendszeren, de nem áll rendelkezésemre egy sem amin kipróbálhatnám, ezért másoknak óvatosnak kell lenniük, ha ezt a verziót választják. A hivatalos ***Portage*** fában az ilyen ***ebuild***-ek néhány hétig ebben az állapotban lehetnek, míg a ***PowerPC*** használók ki nem próbálják azt. Néhány pozitív jelentés után az ***ebuild*** maszkolatlanná válik.

- `DEPEND, RDEPEND`: itt soroljuk fel a függőségeket. Viszonylag teljes nyelvtant és a csomagverziók listáját adhatjuk meg itt. A leggyakoribb módosítók: a `>=`, amely azt jelenti, hogy legalább ilyen verziót kell telepítenünk, általában valamilyen ***API*** miatt amitől a programunk függ; valamint a `!`, amit akkor használunk, ha az adott csomag jelenléte ütközik egy másikkal. Mindkettőt tehát nem telepíthetjük egy időben.

- `RDEPEND`: futásidejű függőségek, ezeket a dolgokat telepítenünk kell a csomag használatához. A `DEPEND` a felépítéshez szükséges függőségeket tartalmazza; a különbség csak akkor mutatkozik meg, ha máshol fordított bináris csomagokat telepítünk.

- `RESTRICT`: itt a *Portage* képességeinek finomhangolását végezhetjük el. Esetünkben, mivel saját készítésű ***ebuild*** csomagról van szó, a `nomirror` segítségével meg tudtam mondani az `emerge`-nek, hogy ne keresgéljen a *Gentoo* tükrein. Ez önmagában még nem jelenti azt, hogy nem használhatom a fejlesztők tükreit. Tulajdonképpen, ha vetünk egy pillantást a `SRC_URI` változóra, látni fogjuk, hogy egy hozzám közel eső tükröt írtam ide, ahonnan a szükséges ***.tar.gz*** állományoka le lehet tölteni.

Ezek után elkezdhetjük felülírni a csomag építését vezérlő függvényeket. A minket érdeklő rész az `src_compile()` függvény lesz. Fogtam a fenti példát és csinosítottam egy kicsit. Láthatjuk, hogy néhány lehetőséget a `USE` változók szabályoznak, míg másokat mi adunk meg, mint például, hogy hol szeretnénk a beállításfájlokat elhelyezni. A ***leállási hiba*** (***die***) üzenetekre nem igazán van szükségünk, viszont jól mutatja, hogy a héjprogramok minden képességét és erejét ki tudjuk használni.
Végül, az `src_install()` függvény helyett is használhattuk volna az alapértelmezettet, de az én rendszeremen az ***/etc/init.d*** állományainak végén nem volt .rc utótag. Ennél is fontosabb, hogy ennek a parancsfájlnak az *OpenSSH*-t kellett lecserélnie a célrendszereken. Ezért biztos akartam lenni benne, hogy az *RC* parancsfájl semmiképp sem ugyanaz, mint ami az *OpenSSH*-t indítja.
A *Portage* gazdag segítő-függvény készlettel rendelkezik amelyek egyszerűsítik az gyakran szükséges feladatokat. Az egyiket ki is használjuk, amikor megmutatjuk hová kell kerülnie az *RC* parancsfájlnak majd ellenőrizzük, hogy végrehajthatónak van-e kijelölve. Ezek után az *ebuild*-et a helyi *Portage* fa másolatunkba helyezve és kiadva az *emerge* parancsot végrehajthatjuk a lépéseket.
Példánk tulajdonképpen csak a felszínt karcolgatja. További részleteket találunk a ➲ *www.gentoo.org/proj/en/devrel/handbook/handbook.xml?part=2&chap=1#doc_chap2* lapon illetve bármely *Gentoo* rendszeren az `emerge --help` kimenetében vagy az `ebuild(1)` és `ebuild(5)` kézikönyv-oldalain.

## Több gép karbantartása

Képzeljük el ugyanezeket a problémákat egyetlen gép helyett több tucat kiszolgálóval és munkaállomások ezreivel felszerelt termelési környezet esetében. Nem túl sok operációs rendszert találunk amely segíthet ilyenkor. Sok polcnyi irodalom foglalkozik az infrastruktúra kezelés témájával. Sajnos még mindig elég sok ad-hoc megoldással találkozhatunk. Bár sok gyártó forgalmaz olyan eszközöket, amelyekkel kezdetben tömegével telepíthetünk rendszereket, azonban a későbbi karbantartás feladatát általában a felhasználóra hagyják. Az új verziók problémája nem csak egyetlen gépen jelentkezhet, egész hálózatokat is sújthat.

Tehát hogyan is kerül a *Gentoo* a termelési környezetbe? Itt jön a forrás alapú terjesztés második meglepetése: a *Portage* képes bináris csomagokat is készíteni. Ezzel lehetővé válik, hogy a sarokban felállított egyetlen gép végezze az összes fordítási munkát. Az összes több gép ezeket a bináris csomagokat használhatja, így nem kell maguknak lefordítani a csomagokat. Sokan biztos azt kérdezik magukban „de hát nem éppen ezt teszi a többi *Linux* terjesztés is?" A különbség abban rejlik, hogy itt a helyes csomag-összeállítás kiválasztása a helyi döntésektől függ, és egy újabb verzió alkalmazása egyértelműen olyasmi, amit helyben kell megoldani. A *Gentoo* a z adott rendszercsapatnak olyan eszközöket ad a kezébe, amelyekkel maguk megoldhatják is az ehhez kapcsolódó különféle problémákat.
Egy fordítókiszolgáló alkalmazásával az erőinket és a verziókezelést hatékonyan koncentrálhatjuk, ugyanakkor mégis helyet biztosítunk a helyi sajátosságoknak. Könnyedén összeállíthatunk egy ilyen lépcsős rendszert. Aztán, amikor meg vagyunk elégedve a tesztelt verziókkal, egyszerűen pillanatképet készítünk a bináris csomagokról majd kiosztjuk őket a sorkatona gépeinknek. A *Portage* újabb verziói beépített lehetőséggel rendelkeznek a helyi fájlkiszolgálón készült bináris csomagok letöltésére, így az egész megoldást kulcsrakészen kapjuk.

## Összefoglalás

Saját csomagunk készítése vagy egy már létező csomag kézi verzió javítása folyton gondokat okoz. Az tűzhöz közelebb álló csomagkezelők, jóllehet kiforrottabbak, sokkal több nehézség árán képesek csak végrehajtani ezeket a feladatokat. Elméletileg a feladatok egyszerűek. Legnagyobb meglepetésemre, hiszen ezt az oldalát nem is reklámozzák a dolognak, a *Gentoo* eszközeivel magunk is könnyedén megoldhatjuk ezeket a problémákat.

## Köszönetnyilvánítás

A szerző szeretne köszönetet mondani ***Stephen Whitenak*** a ***Adelaide Egyetemről*** és ***Andrea Barisaninak*** a ***Trieszti Egyetemről*** hogy segítettek kidolgozni az itt bemutatott termelési környezetben is használható *Gentoo Linuxra* vonatkozó ötleteket. Ezen kívül voltak olyan kedvesek és átnézték a cikket, ahogy azt ***Pia Smith*** a ***Linux Australia***-tól, ***Jeff Waugh*** a *GNOME*-tól, és ***Craige McWhirter*** a ***Sydney Linux Users Group***-tól valamint ***Wade Mealing*** a *Gentoo Server Projectből* is tette.

*Linux Journal 2005. február, 130. szám*

***Andrew Cowie*** készíti az Operational Dynamics (www.operationaldynamics.com), operatív és infrastruktúra mérnöki tanácsadással foglalkozó oldalt. Szervezeteknek segít értékesíteni a technológiájukat az emberekre és az emberek körüli folyamatokra koncentrálva. Valószínűleg éppen amiatt keres állandóan egyszerűbb megoldási lehetőségeket. Elérhető az andrew@operationaldynamics.com címen vagy az irc.freenode.net AfC néven.