

Mono: gépfüggetlen hálózati alkalmazások

Érdekel a .NET? Próbáld ki ezt az érdekes mintaalkalmazást amely a Mono GUI és XML-RPC képességeit aknázza ki.

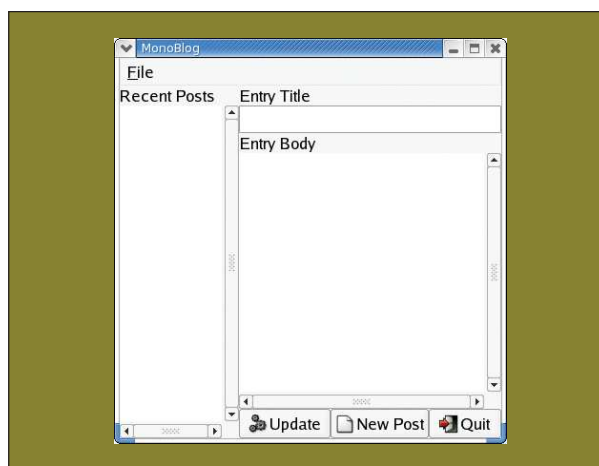
A Mono nem más mint a Microsoft .NET fejlesztői keretrendszerének Ximian által készített, nyílt forrású változata. A .NET több különböző technológiát foglal össze: fordítókat több különféle nyelvhez (többek közt a Microsoft új programnyelvéhez a C#-hoz) amelyek gépfüggetlen bájtkódot készítenek; a *Common Language Runtime* (CLR) nevezetű virtuális gépet, amely értelmezi a bájtkódot, valamint hasznos programokkal teli osztálykönyvtárakat a fájlkezelési szolgáltatásoktól kezdve a GUI készítésig és kezelésig. A Mono megvalósítás tartalmazza a Linux, BSD-alapú rendszerekhez (ideértve a Mac OS X rendszert) és Windows alatt működőképes CLR-t, valamint fordítókat a C# és Basic nyelvekhez. A Mono fejlesztés alatt áll, és a .NET osztálykönyvtár sok része még nem készült el, különösen igaz ez a Windows.Forms csoport esetében amely a Windows GUI-val dolgozó osztályokat tartalmazza. Ugyanakkor a Mono fejlesztői kiadtak egy csomagot a GTK felhasználói felület eszközkészletéhez, így 100% .NET kompatibilitás nélkül is tudunk gépfüggetlen grafikus alkalmazásokat fejleszteni. Cikkünkben bemutatjuk, hogyan lehet a C#, Mono és a Linux hármásával olyan hasznos programot készíteni mint a MonoBlog, amely bármely rendszeren képes futni ahol a Mono és a GTK megtalálható. Valamennyi tájékozottságot feltételezünk a Glade és a C# terén, persze csak teljesen alapszinten. A hálózati források közt hasznos útmutatókat találunk.

Mono beszerzése

A Mono weblap útmutatójából megtudhatjuk, hogyan telepítsük Linux, Mac OS X és Windows rendszerek alá (lásd a forrásokat). Két további C# könyvtárra is szükségünk lesz, nevezetesen a *GTK#* és *XmlRpcCS* könyvtárakra. A MonoBlogot futtató rendszeren fenn kell lenniük az alap GTK könyvtáraknak, melyek egyébként a legtöbb Linux rendszeren megtalálhatók. Windows és Mac OS X rendszerek alatt azonban valószínűleg telepítenünk kell, a csomagokat a GTK weblapján találjuk (lásd a források között). A könyvtárak telepítésére vonatkozó útmutatókat a megfelelő honlapokon találjuk.

MonoBlog, a weblogszerkesztő

A MonoBlog egy weblogszerkesztő program, amely új küldeményeket tud elhelyezni és régiakat tud szerkeszteni



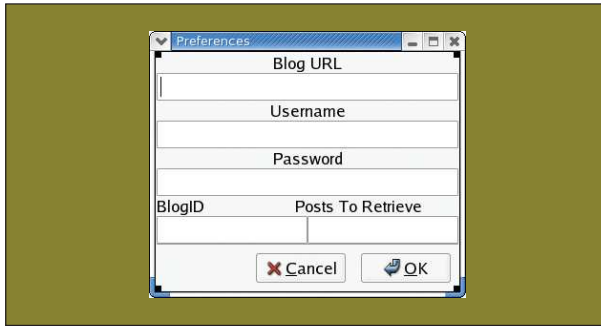
1. ábra A főablak

a naplóban, továbbá lehetővé teszi a felhasználónak, hogy beállításait megváltoztassa. A legtöbb weblog-rendszeren megtaláljuk a MetaWeblog API-ként ismert közös funkciókészletet. A MonoBlog ennek segítségével kommunikál a különféle weblog programokkal, és nem készít külön-külön háttérprogramot a *Movable Type*, *LiveJournal* vagy *Radio Userland* rendszerekhez. A példa teljes C# forráskódját a *Linux Journal* FTP oldaláról tölthetjük le (lásd a forrásokat).

Az 1. és 2. ábra a MonoBlog kezelőfelületét mutatja be, amelyet Linux alatt Glade-del készítettünk. Az 1. ábra főablakába a weblog címének és tartalmának felviteléhez használható szöveges mezőket, valamint a weblog frissítésére, az úrlap törlésére és a kilépésre szolgáló gombokat találjuk. A baloldali fehér rész a GTKTreeView vezérlőelem, amely a régebbi küldeményeket jeleníti meg, és ahol a felhasználó kiválaszthatja, melyiket szeretné frissíteni. A 2. ábrán látható ablakban a felhasználó a MonoBlog és a saját weblogja közötti párbeszédhez szükséges adatokat adhatja meg.

GUI készítés libglade-del

A GTK egyik hasznos szolgáltatása a *libglade* könyvtár, amely lehetővé teszi, hogy a Glade által készített XML állományból készítsük el a program GUI felületét, a grafikus



2. ábra A beállítások ablak



3. ábra MonoBlog Red Hat 9 környezetben

elemek (widgets) kinézetét közvetlenül a kódban adva meg. A `GTK#` változat is tartalmazza ezt a képességet, így a GUI elkészítése nagyon egyszerű. A `MonoBlog` indítása-
kor a `using` paranccsal importáljuk a `GTK` és `Glade` névtereket, majd a konstruktorban megadjuk a következőket:

```
Application.Init();
```

```
Glade.XML gxml = new Glade.XML("monoblog.glade",
    null, null);
gxml.Autoconnect(this);
```

```
Application.Run();
```

Az `Application` osztály meghívása minden `GTK` programban kötelező. Az `Application.Init()` a `GTK`-t állítja alaphelyzetbe, majd az `Application.Run()` átadja vezérlést a `GTK` főciklusnak, amely figyel az eseményeket és üzenetet küld a jelzések (*signals*) bekövetkezésekor. A szabványos `Glade.XML` konstruktor három paramétert vár: a `Glade`-fájl nevét tartalmazó szöveget, egy olyan szöveget, amely megmutatja az objektumnak, hogy a `Glade` fában melyik csomóponton kezdje el építeni a felületet, végül, a harmadik szövegben a kérdéses `Glade` fájlhoz tartozó fordítási tartományt adhatjuk meg.

A `MonoBlog`-nak az `Xml` állományban található valamennyi csomópontot el kell tudnia érni, azaz a főablakot és a „*preferences*” ablakot is. Fordításra nincs szükség, így

a második és harmadik paraméter üres érték marad. Az `Autoconnect()` tagfüggvény (method) a paraméterként megkapott objektumhoz fűzi a `Glade` állományban megadott jelzésekkezelőket (*signal handlers*) és objektumokat, így az objektum válaszolni tud az eseményekre és kezelheti a grafikus elemeket. Mivel a `MonoBlog` aprócska program, valamennyi jelzést a főablakban végeztem el. Egy nagyobb, összetettebb rendszerben, általában jobb megoldás a jelzést más osztályba sorolni.

A grafikus elemek eléréséhez különleges megadási mód szükséges. Mindegyiket példányváltozóként (*instance variable*) kell megadnunk:

```
[Glade.widget] GtkWidgetType widgetname;
```

ahol a `GtkWidgetType` helyére az éppen aktuális objektum-típust írjuk, és a `widgetname` pedig a `Glade` fájlban megadott megfelelő elemnév. Az `Autoconnect()` visszatérése után az elemeket pontosan ugyanúgy használhatjuk, mint ha maga program hozta volna létre őket.

Régi bejegyzések lekérése

A program betöltése során első lépésként lekérdezi a weblogot és letölti a friss küldeményeket, ezután pedig megjeleníti őket a `TreeView` elemben. Mindezt a `MonoBlog` osztály `getRecentPosts()` tagfüggvénye kezeli; a fő konstruktor hívja meg, feltéve, hogy a beállítások már megvannak, hiszen a tagfüggvénynek tudnia kell milyen webloghoz akar csatlakozni. A `MetaWeblog API` egyik függvényhívása, a `metaweblog.getRecentPosts`, megadott számú régi küldeményt ad vissza, illetve, annyit amennyit talál, ha többet kértünk le mint amennyi létezik.

A webloggal folytatott párbeszéd nagyon egyszerű:

```
XmlRpcRequest client = new XmlRpcRequest();
client.MethodName = "metaweblog.getRecentPosts";
client.Params.Add(BlogID);
client.Params.Add(ServerUser);
client.Params.Add(ServerPass);
client.Params.Add(NumberOfPosts);
XmlRpcResponse response = client.Send(ServerURL);
```

Új `XmlRpcRequest` objektum létrehozásához csak a kiválasztott API függvény nevét kell megadnunk, majd feltöltenünk a szükséges paramétereket és elküldeni a weblognak. A weblog visszaküldi a választ, jelen esetben a küldemények tömbjét, amit a `XmlRpcResponse` objektum `Value` mezőjében tárolódik. Ezután a `GTKTreeView` vezérlőelemet kell frissítenünk.

A `GTK 2.0` és újabb rendszer esetében a vezérlő modell-nézet-vezérlő (*model-view-controller*) megközelítést alkalmaz. Itt létrehozunk az új objektummodell, és átadjuk a vezérlőnek:

```
System.Type[] ListTypes = new System.Type[3];
ListTypes[0] = typeof(string);
ListTypes[1] = typeof(string);
ListTypes[2] = typeof(string);
ListStore store = new ListStore(ListTypes);
treeview1.Model = store;
```

Ez az objektummodell egy háromszlopos táblázatot hoz létre. A `ListStore` objektumnak `Type` objektumok tömbjét

kell átadnunk; a tömb minden egyes eleme megfelel az adott oszlop típusának. A weblog-küldemények három elemet tartalmaznak – a címet, a tartalmat és egy egyedi azonosítót. Minthogy mindhárom elem szöveg, az összes oszlopnak `String` típust adunk meg. A tagfüggvény további része végiglépdel a tömbön és feltölti a modellt:

```
TreeIter iter = new TreeIter ();
foreach (Hashtable post in results) {
    String title = (String) post ["title"];
    String postid = (String) post ["postId"];
    String description = (String) post
        ↳ ["description"];

    store.Append (out iter);
    store.SetValue (iter, 0, new GLib.Value(title));
    store.SetValue (iter, 1, new
        ↳ GLib.Value(postid));
    store.SetValue (iter, 2,
        new GLib.Value(description));
}
```

Mindez önmagában még nem elegendő ahhoz, hogy a címeket is megjelenítsük a fában. Ezért egy kis kódot szúrunk a konstruktorba, a `getRecentPosts()` meghívása után:

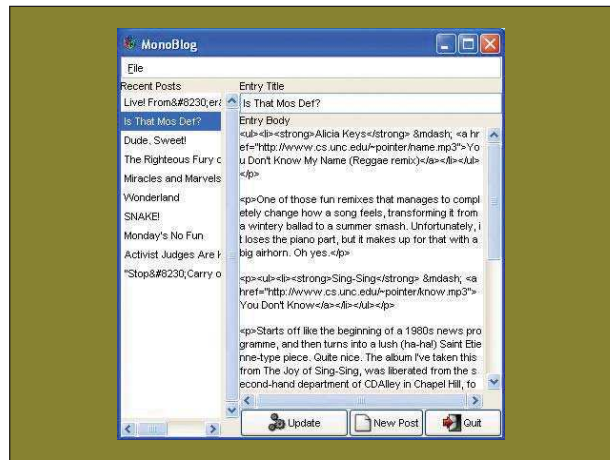
```
TreeViewColumn titleCol = new TreeViewColumn();
CellRenderer titleRenderer = new
↳ CellRendererText();
titleCol.AddAttribute (titleRenderer, "text", 0);
treeview1.AppendColumn (titleCol);
```

Ezzel az új oszlopnézetet hozzáadtuk a fához. Az `AddAttribute()` függvény a modell első oszlopához (`title`) kapcsolódik a 0 paraméterrel. A felhasználónak csak a bejegyzések címét kell látnia a `TreeView` vezérlőelemenben; több oszlopnézetre nincs szükségünk. Az információt azonban a modellben tároljuk, így programunk hatékonyabban működik.

Régi küldemények szerkesztése

Amikor a felhasználó egy bejegyzésre kattint, azt szeretnénk ha a program a régi bejegyzést megjelenítené az ablak jobb oldalán található szöveges mezőben. A `MetaWeblog` API-ban találunk egy `metaweblog.getPost` nevű függvényt, amely a küldeményeket kéri le a weblogból. Mivel azonban mi már korábban letöltöttük őket a `getRecentPosts()` függvénnyel, a program a modelltől is lekérheti az adatokat, nem kell ismét a webloggal társalognia. A `Glade` segítségével a `row_activated` jelzést összekapcsoltuk a `selectOldPost` függvénnyel, így amikor az elemre duplán kattintunk, a következő kód fut le:

```
public void selectOldPost(System.Object obj,
↳ EventArgs e) {
    TreeSelection currentSelection =
↳ treeview1.Selection;
    TreeIter iter;
    TreeModel model = treeview1.Model;
    currentSelection.GetSelected (out model,
↳ out iter);
    String selected = (string) model.GetValue
↳ (iter,1);
```



4. ábra MonoBlog Windows XP környezetben

```
String oldTitle = (string)
↳ model.GetValue(iter,0);
String oldEntry = (string)
↳ model.GetValue(iter,2);

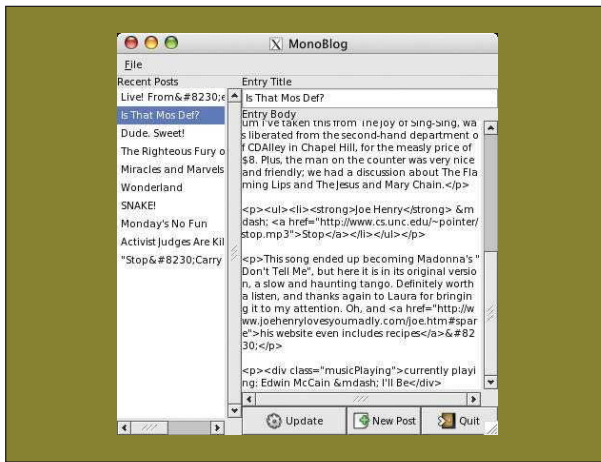
TextBuffer buffer = textview1.Buffer;
entry1.Text = oldTitle;
buffer.SetText(oldEntry);

OldPostID = selected;
EditingOldPost = true;
}
```

A függvény lekéri a `GTKTreeView` vezérlőben aktuálisan kiválasztott elemet, majd közelítő (*iterator*) segítségével címzi meg a modellt és keresi ki a szükséges értéket. Ezután kitölti a szöveges mezőket a kapott információkkal, majd frissít két példányváltozót amelyekre akkor lesz szükségünk ha a felhasználó a `Post` gombra kattint. Ha a program új bejegyzés készítése helyett egy régit szerkeszt, másik `MetaWeblog` API hívást kell kiadnunk, amihez szükségünk lesz a küldemény egyedi azonosítójára. Ezért kell beállítanunk a `OldPostID` és `EditingOldPost` változókat.

A weblog frissítése

Az `Update` gomb `clicked` jelzését az `OnUpdateClicked` tagfüggvényhez rendeltük. Ez a függvény sajnos túl hosszú, így nem tudjuk teljes egészében bemutatni a cikkben, de működése nem annyira bonyolult. Először is lekéri a szöveget a két szöveges mezőből és elkészíti a küldemény hash-tábla megfelelőjét; erre a `MetaWeblog` API híváshoz lesz szükségünk. Attól függően, hogy az `EditingOldPost` jellet beállítottuk-e, a függvény `metaweblog.newPost` vagy `metaweblog.editPost` hívással XML-RPC kérelmet küld a weblognak. Amikor a weblog sikeres választ ad vissza, jelezve, hogy a frissítés megtörtént, a függvény tisztítja a mezőket és lehetővé teszi a felhasználónak, hogy új bejegyzést készítsen. A főablak további gombjai, a `New Post` és a `Quit` rövid programocskát tartalmaznak. Akárcsak a `Post` gomb esetében itt is a `clicked` jelzéseket csatoltuk a `MonoBlog`hoz. A `New Post` a szöveges mezőket töröl és az `EditingOldPost` jellet hamisra állító függ-



5. ábra MonoBlog Mac OS X környezetben

vényhez kapcsoljuk, így a felhasználó újratekesheti a munkát. A Quit, ahogy annak lennie kell, az Application.Exit() GTK hívás segítségével kilép a MonoBlogból.

Beállítások (preferences)

A .NET osztálykönyvtár osztályokat is tartalmaz, amelyekkel XML állományokból olvashatjuk be beállításainkat. Az 1. listában a MonoBlog beállításállományára láthatunk példát. Ezeket az értékeket az alább bemutatott getConfig() függvény olvassa be:

```
private bool getConfig {
    try {
        AppSettingsReader config = new
            AppSettingsReader();

        ServerURL = (string)
            config.GetValue("ServerURL",
                typeof(string));

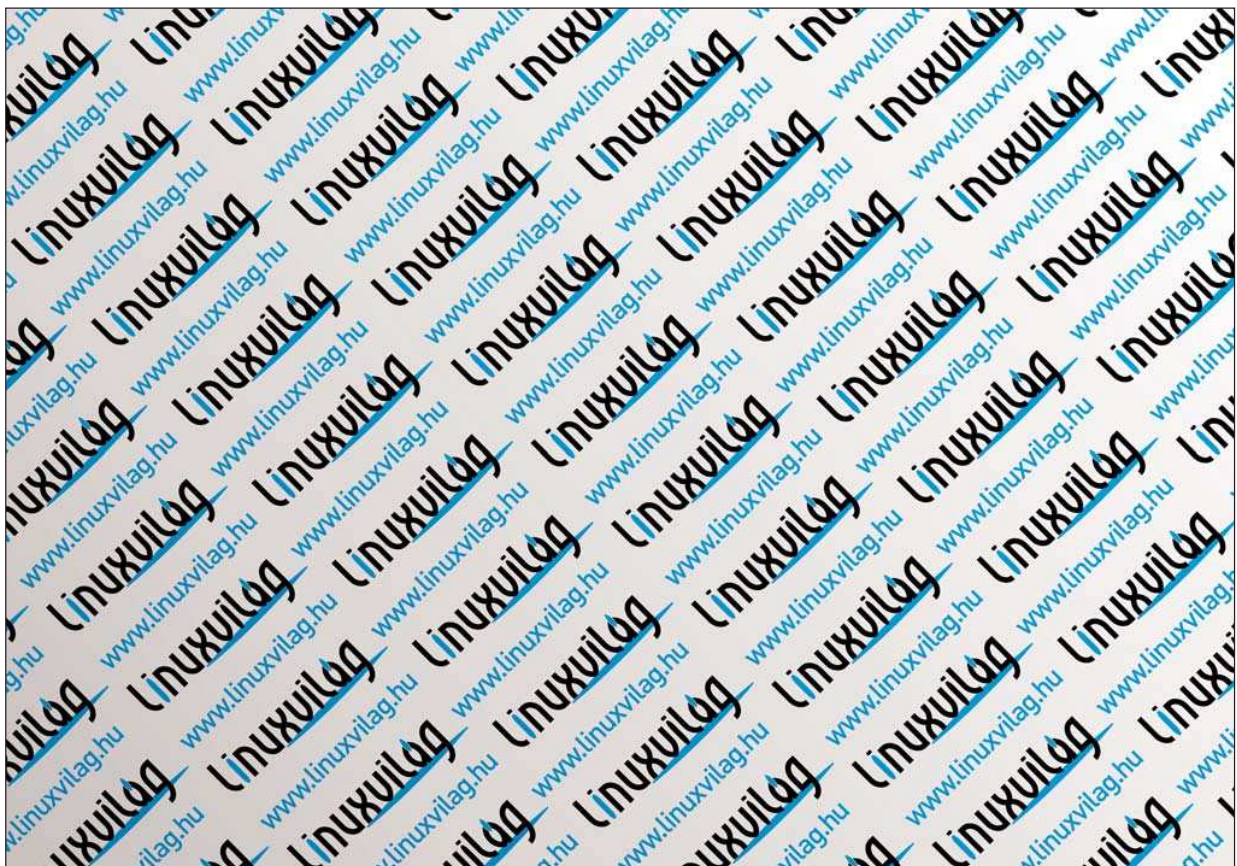
        ServerUser = (string)
            config.GetValue("ServerUser",
                typeof(string));

        ServerPass = (string)
            config.GetValue("ServerPass",
                typeof(string));

        BlogID = (string) config.GetValue("BlogID",
            typeof(string));

        NumberOfPosts = (string)
            config.GetValue("NumberOfPosts",
                typeof(string));

        catch(Exception problem) {
            return false;
        }
        return true;
    }
}
```



Alapértelmezés szerint az AppSettingsReader objektum *programfájl*-neve.config nevű állományt keres, ami jelen esetben *monoblog.exe.config* lesz. Ezután a GetValue() függvény segítségével kaphatjuk meg a szükséges beállítás-értékeket. A MonoBlog ezt a függvényt a konstruktorában hívja meg, még mielőtt megpróbálná letölteni a régi küldeményeket a weblogról, így megkapja a szükséges információkat. Ha az állomány nem létezik vagy az adatok betöltése nem sikerül, a függvény hamis értéket ad vissza.

A konstruktor csak akkor hívja meg a getRecentPosts() függvényt ha a visszatérési érték igaz, így biztosan nem használunk fel sérült értékeket. A beállítások frissítése már kicsit nehezebb feladat. Először is a főablak menüsorában található Preferences pontot a Glade Menu Editor-ral az OnPrefsActivate függvényhez kötöttük. Ez a 2. ábrán látható párbeszédablakot hozza fel majd a mezőket kitölti az aktuális értékekkel, ha van ilyen. Amikor a felhasználó a párbeszéd OK gombjára kattint, a MonoBlog frissíti a változókat, és a friss adatokat visszaírja a beállításállományba. Sajnos a .NET osztálykönyvtár nem rendelkezik beállításfájl frissítő osztállyal. Minthogy a jelen beállítás nem túl bonyolult, készítettem egy saveConfig() nevű függvényt, amely megnyitja az alapértelmezett beállításfájl és a frissített információkat write() parancsok sorozatával visszairja a lemezre. Ezt valami kifinomultabb megoldással kellene helyettesíteni amely felépíti a helyes XML dokumentumot, de ennél az alkalmazásnál egyszerűbb volt körülményeskedés nélkül kiírni az értékeket.

Hibakezelés

Minthogy a MonoBlog az interneten dolgozik, ahol a programunktól függetlenül mindenféle gondok adódhatnak (hálózati hibák, névkiszolgáló problémák és így tovább), valamint alapvető hibakezelési megoldásra is szüksége van. A getRecentPosts() és OnUpdateClicked() tagfüggvények try...catch blokkba kerültek. Végrehajtjuk az internetet használó kódot, és ha valamilyen problémába ütközünk a következő catch blokk fut le:

```
catch(Exception problem) {
    MessageDialog md =
        new MessageDialog(MonoBlogWindow,
            DialogFlags.DestroyWithParent,
            MessageType.Error,
            ButtonType.Close,
            problem.ToString());

    md.Run();
    md.Destroy();
}
```

következményképpen a képernyőn a problémát ismertető hibaüzenete jelenik meg, szöveges üzeneteként átadva a Mono CLR-től kapott üzenetet. Így a felhasználó folytathatja a munkát és lehetőség van javítani a problémán. Ugyanakkor jelenleg a Mono CLR PPC ágán a hibakezelés nemigen működik így ha a program Mac OS X rendszeren fut, a hibakezelés nem fog működni és a program a csendben nem csinál semmit. Folyik a munka a PPC átiraton, szóval, mire ez cikk a nyomdába kerül, ez a probléma lehet, hogy már a múlté.

1. lista XML beállításfájl minta

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<appSettings>
<add key="ServerURL"
value="http://www.test.com/mt-xmlrpc.cgi"/>
<add key="ServerUser" value="example"/>
<add key="ServerPass" value="password"/>
<add key="BlogID" value="1"/>
<add key="NumberOfPosts" value="10"/>
</appSettings>
</configuration>
```

Fordítás és futtatás

A C# programok fordítását az mcs fordítóval végezzük.

A MonoBlogot a következő paranccsal fordítottuk:

```
mcs -r gtk-sharp.dll -r glade-sharp.dll
➔ -r XmlRpcCS.dll -r glib-sharp.dll monoblog.cs
```

A -r kapcsoló mutatja milyen háttérre van szüksége a programnak; itt egyszerűen csak azt kell megmutatnunk, milyen könyvtárakat használ a MonoBlog. Eredményképpen egy monoblog.exe nevű lefordított bájtkódot kapunk. A program futtatásához a Mono CLR-nek ezt a fájlt kell paraméterként megadnunk:

```
mono monoblog.exe
```

Kifejlesztettünk egy programot Linux alatt amit szinte gond nélkül tudunk Windows vagy Mac OS X környezetben futtatni. Egyszerűen másoljuk a *monoblog.exe*, *monoblog.exe.config* és *monoblog.glade* állományokat a másik rendszerre és futtasuk le az imént bemutatott módon a Mono CLR programmal. A 3., 4. és 5. ábra a MonoBlogot mutatja működés közben Linux, Windows és Mac OS X gépeken. Semmilyen kódot nem kell megváltoztatnunk; a program úgy működik ahogy van, feltéve, hogy a futásához szükséges könyvtárak elérhetők.

Összefoglalás

Remélhetőleg cikkünkben be tudtuk mutatni, hogyan lehet a Mono és a C# segítségével könnyen és gyorsan gépfüggetlen alkalmazásokat készíteni. Fejleszthetünk az egyik rendszeren és biztosak lehetünk benne, hogy a program bármely rendszeren futni fog, amely ismeri a Mono és GTK könyvtárakat. Maga a MonoBlog program megérett a további kísérletezésre. Néhány lehetséges fejlesztési irány lehet a további formázási lehetőségek, részletesebb hibajelentések kidolgozása, GtkHTML# kötésekkel készíthetnénk HTML előnézet ablakot, valamint a MetaWeblog API egyéb lehetőségeit is kihasználhatnánk, például a weblog küldeményeinek törlését is felvehetnénk lehetőségek sorába.

Linux Journal 2004. július, 123. szám

Ian Pointer munkanélküli végzős számítástechnikus az Egyesült Királyságban. A ian@snappishproductions.com címen érhető el.