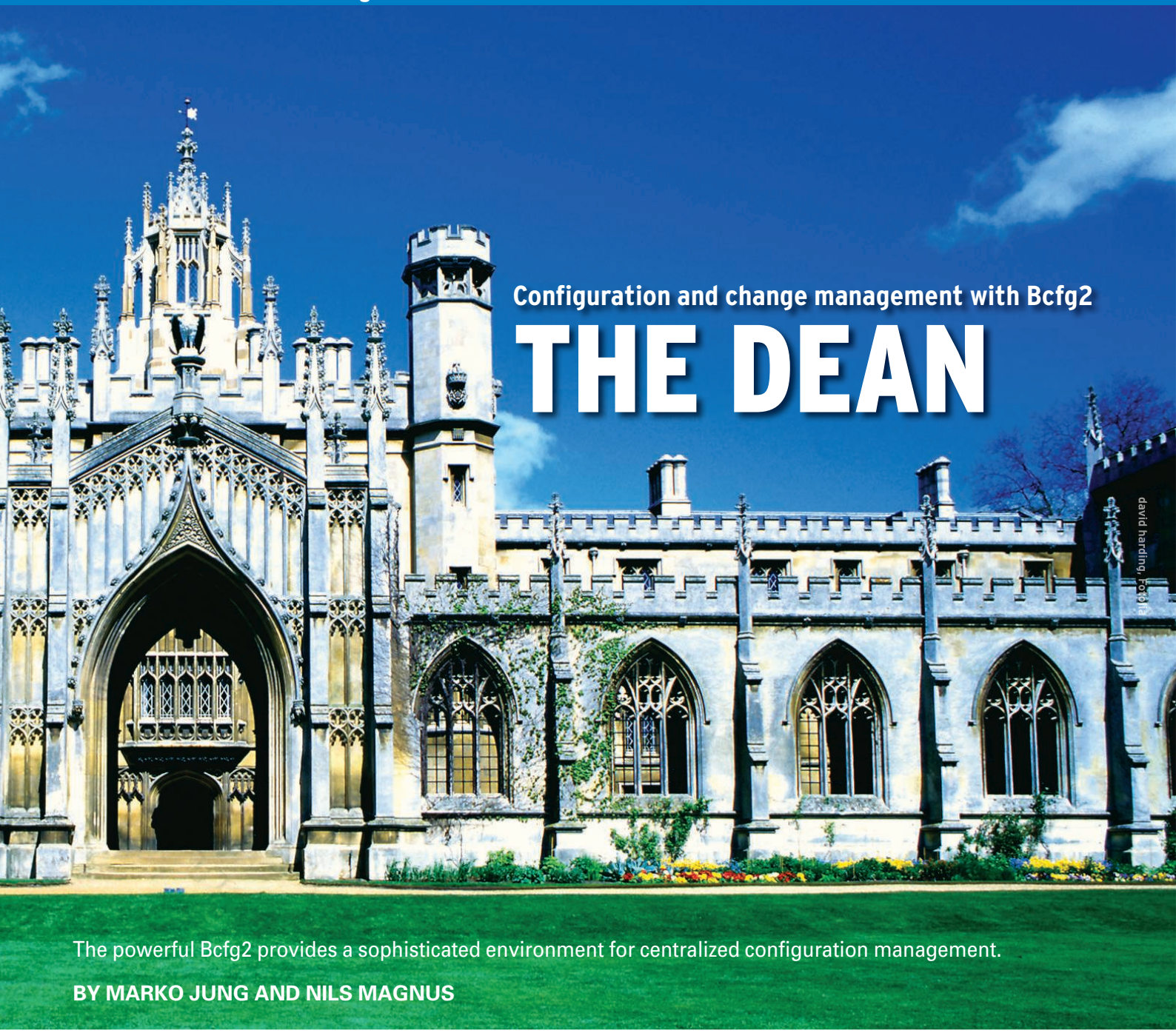

Configuration and change management with Bcfg2

THE DEAN

david harding, fotolia

The powerful Bcfg2 provides a sophisticated environment for centralized configuration management.

BY MARKO JUNG AND NILS MAGNUS

The experts at the department of mathematics and computer science at the Argonne National Laboratory [1] were so unhappy about having to configure their numerous computer systems manually that they started an internal research project, dubbed Bcfg2 [2], that they later released under the BSD license.

Getting Started

Bcfg2 provides a sophisticated system for describing and deploying complete client configurations. This flexible tool uses a comprehensive XML format to describe the configurations, and RPC to communicate with the clients.

Experts will find Bcfg2 easy to extend, but the learning curve is steep – the program uses a powerful, abstract approach.

Supported Platforms

Bcfg2 supports various platforms, including openSUSE, Fedora, Gentoo, and Debian, as well as their many derivatives. The tool also runs on FreeBSD, AIX, Solaris, and Mac OS X through the use of the developer's distribution-independent Encap packages [3] on top of the ready-to-run client and server packages. As an alternative, you can check out the openSUSE Build Service [4],

Table 1: Entry Configuration Types

Tag	Description	Example
Package	Name of the software package	*<Package name='bcfg2'/>*
ConfigFile	Absolute path of the configuration file	*<ConfigFile name='/etc/bcfg2.conf'/>*
Service	Name of the init script for the service (not of the respective daemon)	*<Service name='ntpd'/>*
Directory	Absolute path of the directory in the filesystem	*<Directory name='/var/tmp'/>*
SymLink	Absolute path of the smybolic link	*<SymLink name='/dev/MAKEDEV'/>*
Permissions	Permissions of the absolute path	*<Permissions name='/etc/passwords'/>*

which offers the package for a number of other platforms.

Architecture

To manage specifications, Bcfg2 uses a server that communicates with a fairly lean counterpart on the client. To install both the server and the client, which comprises just a couple of lines of Python code, you can use the package manager for your distribution.

To get the client to request updates from the server at regular intervals, you need to create a crontab entry. Alternatively, you can run Bcfg2 as a background process. In that case, the server will actively contact the agent.

Settings on managed systems are configured at the Bcfg2 server. For each client, the server stores a description, which it generates from a central specification.

At the highest level, Bcfg2 works with profiles that describe classes of identical computers, such as desktop systems or web servers. Each managed machine has exactly one profile.

Specific logical system areas within the profile are organized into groups, such as office software or network settings. The groups, which can be nested recursively, let administrators organize configuration specifications in a meaningful way.

Each group in turn is made up of an arbitrary number of bundles with scope that typically extends to a single software product, such as Postfix, OpenOffice, or the *nameswitch* mechanism.

Profiles, groups, and bundles are defined in the *metadata/groups.xml* file (see Listing 1).

Making Packages

Bundles can be broken down still further: Bcfg2 groups the configuration files and software packages belonging to a specific system service in a file below the *Bundler* directory. For example, the *Bundler/motd.xml* file defines the configuration bundle for the */etc/motd* welcome message:

```
<Bundle name=➥
'motd' version='2.0'>
   <Package name='login' />
   <ConfigFile name=➥
   '/etc/motd' />
</Bundle>
```

Some nicely commented examples of bundles are available from the project's wiki [5]. Bundles contain a number of configuration items, which Bcfg2 aptly refers to as entries.

The system defines six different entry types (see Table 1): The *ConfigFile* type manages a file and its content; the *Directory*, *Permission*, and *SymLink* types let administrators manage or monitor objects. The *Package* and *Service* types of entries have more in the way of artificial intelligence.

Bcfg2 abstracts the distribution-specific methods used to, for example, install a package on a client; that is, it will call *aptitude* on a Debian system and *rpm* or *yum* on an openSUSE system.

The same principle applies to system services that are typically listed in */etc/init.d*.

Installing the Server

The abstract structure that describes the client is managed entirely on the server side. Typing *bcfg2-admin init* initializes a new configuration.

First, you are prompted for the details of the repository path, an SSL certificate for secure communications with the clients, and an access password. The Bcfg2 admin script then goes on to create several files and directories and populate the */etc/bcfg2.conf* configuration file with meaningful defaults. You might need to modify the entry in the *[compo-*

Listing 1: Bundle and Subgroup Definitions

```
01 <Groups>
02   <Group name='desktop' profile='true'>
03     <Bundle name='motd' />
04     <Bundle name='networking' />
05     <Group name='office-workstation' />
06     <Group name='debian-stable' />
07   </Group>
08
09   <Group name='webserver' profile='true'>
10     <Group name='apache' />
11     <Bundle name='networking' />
12     <!-- ... --!>
13   </Group>
14
15   <Group name='office-workstation'>
16     <Group name='gnome-desktop' />
17     <!-- ... --!>
18   </Group>
19
20   <Group name='debian-stable'
21         toolset='debian' />
22 </Groups>
```

Listing 2: Registering Clients

```
01 01 <Clients>
02 02   <Client profile="desktop"
03             name="alpha.example.com" pingable="N"/>
04 03   <Client profile="desktop"
05             name="beta.example.com"  pingable="Y"/>
06 04   <Client profile="webserver"
07             name="gamma.example.com" pingable="N"/>
08 05   <!-- .. --!>
09 06 </Clients>
```

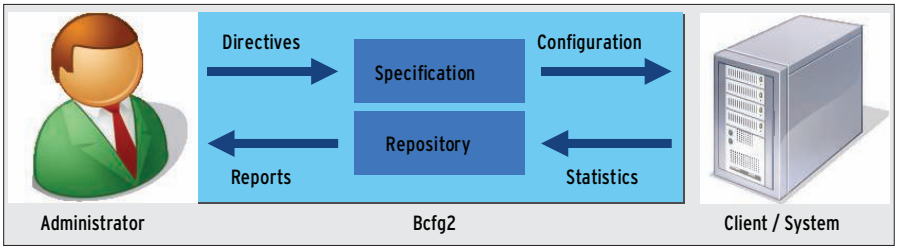

Figure 1: The administrator uses Bcfg2 to create a specification defining the required client configuration. Bcfg2 attempts to perform the changes and documents the results.

nents] section to match the hostname, which your clients can resolve with DNS.

This name is also the perfect choice for the *Common Name* when you generate a certificate. *bcfg2-admin* might create the file as read-only, but you will want to change this because it contains the client access password.

Additionally, the tool sets up an empty configuration repository below */var/lib/bcfg2*. Calling */etc/init.d/bcfg2-server start* launches the server.

Client Configuration

The only component you need to install on the client is the *bcfg2* client package. Additionally, you need to store the configuration file in */etc/bcfg2.conf*.

The next step is to add each Bcfg2 client to the *Metadata/clients.xml* file on the server (Listing 2). All specifications use the XML format. The administrator can later assign profiles to hosts by using this file.

Bcfg2 identifies clients by their DNS names, which makes it important to have a working DNS system. The example assigns the *desktop* profile to the hosts *alpha* and *beta.example.com* and *webserver* to *gamma.example.com*. Bcfg2 is now ready for action.

Multi-Stage Process

An update process consists of multiple steps (see Figure 1): The client first retrieves its XML-formatted configuration from the server, performs an inventory, and compares the current and target states. After completing all the changes associated with the update, the client again checks the vectors and informs the server of the results. The server generates reports and statistics on the basis of the information.

To check the reports, the system administrator can trigger a test update. To trigger an update on the client side, type the following:

```
bcfg2 -q -v -n
```

This command launches the update process in a non-invasive, no-op mode (*-n* option). The *-q* option, which stands for *quick*, omits some checks. The *-v* option provides more information. Listing 3 shows the results of this kind of query.

So far, the specification is still empty. Bcfg2 will not display correct (line 5) or incorrect (line 6) entries because the server does not having a single configuration entry in its central repository (line 7). The number of *unmanaged entries* in line 8 is more interesting: The Bcfg2 client has found 242 configuration objects that the administrator has not explicitly configured. The server will want to register and manage these objects in the future. These entries are mainly *Service*- and *Package*-type items added on the client side by the Linux distribution. Line 2 tells more about them.

Bcfg2 has automatically identified the distribution as one that uses the *Chkconfig* service to configure init services and *RPM* as its package manager. A Debian system would use *update-rc.d* and the *DEB* package format.

Because the *bcfg2* command has just requested an update without changes, the details in lines 10 through 14 match the values in the lines above. It is now up to the system administrator to view the unmanaged entries one by one and add any appropriate definitions. Stipulating the *-e* option tells the system to detail the entries. One of Bcfg2's main strengths is its ability to gradually complete an incomplete configuration.

The update process is the core element in any configuration management system, and it makes sense to examine this process in more detail. The process comprises three phases.

Probing

In the *initial* phase, the client connects with the server and performs a number

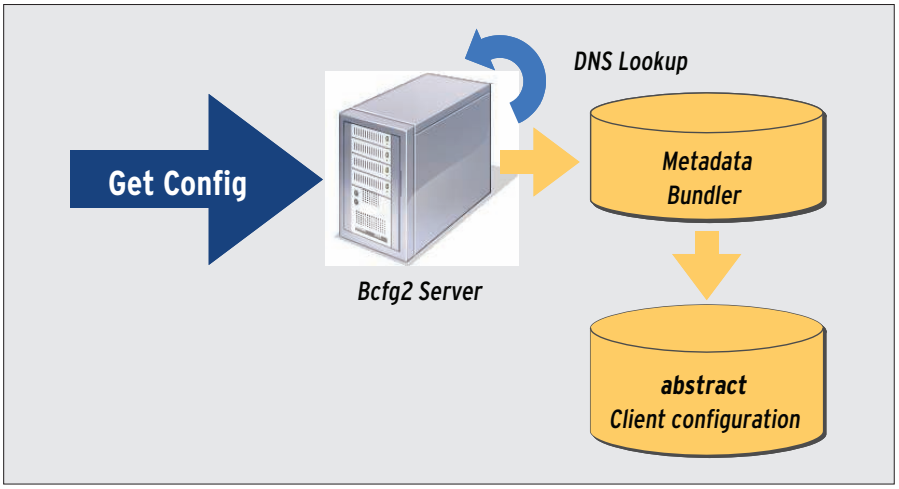

Figure 2: During the first stage, the Bcfg2 server generates the abstract client configuration from the metadata.

Listing 3: Non-Configured Server

```
01 Loaded tool drivers:
02  Chkconfig    POSIX
   PostInstall  RPM
03
04 Phase: initial
05 Correct entries:        0
06 Incorrect entries:      0
07 Total managed entries:  0
08 Unmanaged entries:      242
09
10 Phase: final
11 Correct entries:        0
12 Incorrect entries:      0
13 Total managed entries:  0
14 Unmanaged entries:      242
```

Anzeige
wird
separat
angeliefert

of tests, known as probes. These probes take the form of short shell scripts that run on the client side. Depending on the results, the server might add groups or bundles to the client specification. A test will simply return a result of *group:group name*.

The server automatically assigns the client to additional groups. This means that it will automatically add, say, a matching modem bundle if it discovers *lspci* on the client. For more examples and an exhaustive HOWTO, see the Bcfg2 website [6].

Configuring

In the second phase, the client receives the configuration description from the server. This description contains many configuration entries of the six basic types. The client does a local inventory to create a comparable structure and then goes on to compare the local inventory with the description from the server. In case of deviations, the client will perform changes depending on the corresponding configuration element types: The Bcfg2 client will resolve differences between configuration files by using the version defined on the server.

If a service is not running, the client will start it. Dependencies between configuration elements are resolved automatically.

The client repeats this process until no further progress is made – that is, until the next call to the update process fails to make any changes.

Reporting

In the third and *final* phase, the client generates a report containing the system status and other details, including the number of correct and incorrect configuration entries and the number of non-managed objects on the system. The client sends this message to the server, which then processes it to create web pages, RSS feeds, and email.

The core of any Bcfg2 system is the configuration specification. Administrators use it to describe the target configurations for the systems they manage. This process occurs in two stages: Bcfg2 refers to the structure looked at earlier, as well as the profiles, groups, and bundles it contains as metadata. The metadata define the elements that Bcfg2 needs to configure for a client.

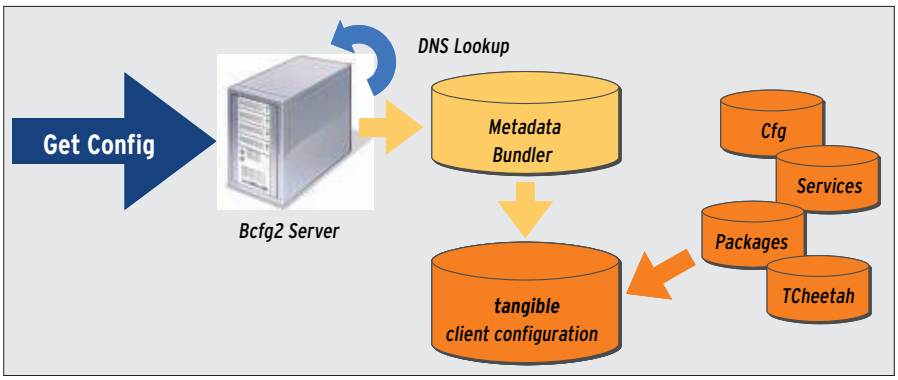

Figure 3: Bcfg2 uses multiple generators to fill abstract specification entries with content. This process creates complete information from bundles and metadata.

When a client sends a request to a server, the server generates the abstract configuration from the matching metadata (see Figure 2). This template contains all the required configuration entries for the target system, but without any content. For example, a *ConfigFile* element contains a file name, but not the file content.

Generators

After the server has created the framework of the configuration, it uses generators to bind tangible information to each entry. Bcfg2 uses a number of generators written in Python.

The administrator needs to run the following command

```
generators = Cfg, Pkgmgr, ➥
Rules, TCheetah
```

to add them to the */etc/bcfg2.conf* file. Each registered generator is capable of instantiating a number of configuration elements.

In simple cases, the Cfg generator will return a static file, and in more complex cases, a TCheetah generator will use a template and script language to retrieve the file content from a database entry. Other generators handle configuration elements, such as services and packages. Consequently, Bcfg2 is capable of using almost any data source to compile a tangible configuration (see Figure 3).

Cfg

The Cfg plugin mainly generates content for *ConfigFile*-type entries. To configure an element, the administrator creates a subdirectory below *Cfg* in the repository with the same name as the corresponding bundle. Then you create a static file in the directory and let Bcfg2 distribute the file to all your clients.

To define the target clients, you can add suffixes to file names. The *H_Hostname* suffix distributes the file to the specified machines only; *GPrio_Group* sends the file to all systems that have the specified group profile. If a host belongs to multiple groups, Bcfg2 applies the highest priority file.

An *:info* file in the same directory defines permissions, with entries like:

```
owner: root
group: admin
perms: 0644
```

Also, you can specify values such as the Bcfg2 encoding or the behavior for local changes. The functionality provided by

Listing 4: Script for an Automtic motd

```
01 Welcome to $self.metadata.hostname!
02
03 This system is managed by Bcfg2. It is a
04 member of the following groups:
05
06 #for $group in $self.metadata.groups:
07  * $group
08 #end for
```

the Cfg generator helps you manage a major part of the system; however, it does not always offer the flexibility required to manage large-scale systems.

To manage large networks, the Bcfg2 developers created the TCheetah generator, based on the Cheetah template language [7]. Cheetah supports instructions that range from simple string operations, to flow control, to Python code embedded directly in configuration files.

TCheetah adopts the Cfg generator's directory structure with directories below */var/lib/bcfg2/TCheetah* representing the configuration elements. Each directory contains an *info* file with the same content as the Cfg generator, along with a *template* configuration. Cheetah code, which can contain some Bcfg2-specific extensions, is then added. Dropping Listing 4 into */var/lib/bcfg2/TCheetah/etc/motd/template* would create a dynamic message of the day. The TCheetah Generator replaces *$self.metadata.hostname* with the actual values and then runs the loop that starts with *#for*.

Connections

A database lets you retrieve configuration information from more data sources. For example, you could automatically generate the DHCP, DNS, and NIS configuration data from a source such as a directory service. The example in Listing 5 shows how TCheetah on Debian configures the network interfaces in */etc/network/interfaces* on the basis of data from a PostgreSQL database.

Services

Configurations do not just apply to file entries. System services must be configured to reflect the runlevels. The Service generator reads the administrative information in *Svcmgr/services.xml* to discover how to configure, say, the NTP service.

To do so, the server references the service referred to as *< service name = 'ntpd' / >* in the matching bundle.

Then, *services.xml* is used to specify whether or not the client should start the service:

```
<Services priority='0'>
   <Service name=
   'ntpd' status='on' />
</Services>
```

The generator converts these details to tangible configurations and sends them to the client, which then applies distribution-specific methods to enable or disable the service.

Packages

Bcfg2 does not replace the package manager, but it can have the effect of shifting more control from the package manager to the system administrator, who can use Bcfg2 to specify which version of which individual package the tool installs. XML files manage information about the available packages and synchronize the details with the installation server package selection. Multiple installation servers allow administrators to assign different priorities – for example, for security updates.

The Bcfg2 client compares global and local package versions and upgrades or downgrades accordingly. Administrators can freeze special versions by assigning them the highest priority.

Other generators configure directories, symlinks, and many other element types. Some more experienced administrators will appreciate the ability to use plugins to bind Python functions to the generators provided with the Bcfg2 distribution.

Documentation Imminent

The lively developer community that surrounds Bcfg2 integrates new designs with the system almost every week. Just recently, a new team that will be focusing on improving the documentation was founded on the mailing list [8]. Documentation is absolutely vital because Bcfg2 requires that you know so many details of the system.

Administrators will also appreciate the tool's support for gradual migration, which removes the need for drastic changes and allows step-by-step specification of the configuration. All told, the use of Bcfg2 to configure and validate clients is a very powerful tool for anyone who is undaunted by its high level of abstraction. ■

Listing 5: TCheetah-Configured Network/linterfaces

```
01 #from Bcfg2.Server.dbconnection import DBPgConnection
02 #silent result = DBPgConnection().execute(
03    "SELECT ip, netmask, broadcast, gateway \
04     FROM hosts \
05     WHERE hostname = '%s'" % $self.metadata.hostname)
06
07 auto eth0
08 iface eth0 inet static
09     address   $result[0]
10     netmask   $result[1]
11     broadcast $result[2]
11     gateway   $result[3]
```

THE AUTHOR

Marko Jung is an IT consultant who advises small to mid-sized businesses on migration projects and the deployment of free software.

INFO

[1] Argonne National Library, Mathematics and Computer Science Division: *http://www-new.mcs.anl.gov/new*

[2] Bcfg2: *http://www.bcfg2.org*

[3] Bcfg2 Encap packages: *http://trac.mcs.anl.gov/projects/bcfg2/wiki/EncapPackages*

[4] Bcfg2 packages on the openSUSE Build Service: *http://download.opensuse.org/repositories/home:/markojung:/bcfg2*

[5] Bcfg2 annotated examples: *http://www.bcfg2.org/wiki/AnnotatedExamples*

[6] Bcfg2 probes: *http://trac.mcs.anl.gov/projects/bcfg2/wiki/Probes*

[7] Cheetah template engine: *http://www.cheetahtemplate.org*

[8] Bcfg2 mailing list: *mailto:bcfg-dev@mcs.anl.gov*